Text Data Management and Analysis

A Practical Introduction to Information Retrieval and Text Mining

ChengXiang Zhai
Sean Massung

Text Data Management and Analysis

ACM Books

Text Data Management and Analysis

A Practical Introduction to Information Retrieval and Text Mining

ChengXiang Zhai

University of Illinois at Urbana–Champaign

Sean Massung

University of Illinois at Urbana–Champaign

ACM Books #12

To Mei and Alex

To Kai

Contents

Preface

The growth of "big data" created unprecedented opportunities to leverage computational and statistical approaches to turn raw data into actionable knowledge that can support various application tasks. This is especially true for the optimization of decision making in virtually all application domains such as health and medicine, security and safety, learning and education, scientific discovery, and business intelligence. Just as a microscope enables us to see things in the "micro world" and a telescope allows us to see things far away, one can imagine a "big data scope" would enable us to extend our perception ability to "see" useful hidden information and knowledge buried in the data, which can help make predictions and improve the optimality of a chosen decision. This book covers general computational techniques for managing and analyzing large amounts of text data that can help users manage and make use of text data in all kinds of applications.

Text data include all data in the form of natural language text (e.g., English text or Chinese text): all the web pages, social media data such as tweets, news, scientific literature, emails, government documents, and many other kinds of enterprise data. Text data play an essential role in our lives. Since we communicate using natural languages, we produce and consume a large amount of text data every day on all kinds of topics. The explosive growth of text data makes it impossible, or at least very difficult, for people to consume all the relevant text data in a timely manner. Thus, there is an urgent need for developing intelligent information retrieval systems to help people manage the text data and get access to the needed relevant information quickly and accurately at any time. This need is a major reason behind the recent growth of the web search engine industry. Due to the fact that text data are produced by humans for communication purposes, they are generally rich in semantic content and often contain valuable knowledge, information, opinions, and preferences of people. Thus, as a special kind of "big data," text data offer a great opportunity to discover various kinds of knowledge useful for many applications, especially knowledge about human opinions and preferences, which is often

directly expressed in text data. For example, it is now the norm for people to tap into opinionated text data such as product reviews, forum discussions, and social media text to obtain opinions. Once again, due to the overwhelming amount of information, people need intelligent software tools to help discover relevant knowledge for optimizing decisions or helping them complete their tasks more efficiently. While the technology for supporting text mining is not yet as mature as search engines for supporting text access, significant progress has been made in this area in recent years, and specialized text mining tools have now been widely used in many application domains. The subtitle of this book suggests that we cover two major topics, *information retrieval* and *text mining*. These two topics roughly correspond to the techniques needed to build the two types of application systems discussed above (i.e., search engines and text analytics systems), although the separation of the two is mostly artificial and only meant to help provide a high-level structure for the book, and a sophisticated application system likely would use many techniques from both topic areas.

In contrast to structured data, which conform to well-defined schemas and are thus relatively easy for computers to handle, text has less explicit structure so the development of intelligent software tools discussed above requires computer processing to understand the content encoded in text. The current technology of natural language processing has not yet reached a point to enable a computer to precisely understand natural language text (a main reason why humans often should be involved in the loop), but a wide range of statistical and heuristic approaches to management and analysis of text data have been developed over the past few decades. They are usually very robust and can be applied to analyze and manage text data in any natural language, and about any topic. This book intends to provide a systematic introduction to many of these approaches, with an emphasis on covering the most useful knowledge and skills required to build a variety of practically useful text information systems.

This book is primarily based on the materials that the authors have used for teaching a course on the topic of text data management and analysis (i.e., CS410 Text Information Systems) at the University of Illinois at Urbana–Champaign, as well as the two Massive Open Online Courses (MOOCs) on "Text Retrieval and Search Engines" and "Text Mining and Analytics" taught by the first author on Coursera in 2015. Most of the materials in the book directly match those of these two MOOCs with also similar structures of topics. As such, the book can be used as a main reference book for any of these two MOOCs.

Information Retrieval (IR) is a relatively mature field and there are no shortage of good textbooks on IR; for example, the most recent ones include *Modern Information Retrieval: The Concepts and Technology behind Search* by Baeza-Yates

and Ribeiro-Neto [2011], *Information Retrieval: Implementing and Evaluating Search Engines* by Büttcher et al. [2010], *Search Engines: Information Retrieval in Practice* by Croft et al. [2009], and *Introduction to Information Retrieval* by Manning et al. [2008]. Compared with these existing books on information retrieval, our book has a broader coverage of topics as it attempts to cover topics in both information retrieval and text mining, and attempts to paint a general roadmap for building a text information system that can support both text information access and text analysis. For example, it includes a detailed introduction to word association mining, probabilistic topic modeling, and joint analysis of text and non-text data, which are not available in any existing information retrieval books. In contrast with IR, Text Mining (TM) is far from mature and is actually still in its infancy. Indeed, how to define TM precisely remains an open question. As such, it appears that there is not yet a textbook on TM. As a textbook on TM, our book provides a basic introduction to the major representative techniques for TM. By introducing TM and IR in a unified framework, we want to emphasize the importance of integration of IR and TM in any practical text information system since IR plays two important roles in any TM application. The first is to enable fast reduction of the data size by filtering out a large amount of non-relevant text data to obtain a small set of most relevant data to a particular application problem. The second is to support an analyst to verify and interpret any patterns discovered from text data where an analyst would need to use search and browsing functions to reach and examine the most relevant support data to the pattern.

Another feature that sets this book apart is the availability of a companion toolkit for information retrieval and text mining, i.e., the META toolkit (available at https://meta-toolkit.org/), which contains implementations of many techniques discussed in the book. Many exercises in the book are also designed based on this toolkit to help readers acquire practical skills of experimenting with the learned techniques from the book and applying them to solve real-world application problems.

This book consists of four parts. Part I provides an overview of the content covered in the book and some background knowledge needed to understand the chapters later. Parts II and III contain the major content of the book and cover a wide range of techniques in IR (called Text Data Access techniques) and techniques in TM (called Text Data Analysis techniques), respectively. Part IV summarizes the book with a unified framework for text management and analysis where many techniques of IR and TM can be combined to provide more advanced support for text data access and analysis with humans in the loop to control the workflow.

The required background knowledge to understand the content in this book is minimal since the book is intended to be mostly self-contained. However, readers

are expected to have basic knowledge about computer science, particularly data structures and programming languages and be comfortable with some basic concepts in probability and statistics such as conditional probability and parameter estimation. Readers who do not have this background may still be able to follow the basic ideas of most of the algorithms discussed in the book; they can also acquire the needed background by carefully studying Chapter 2 of the book and, if necessary, reading some of the references mentioned in the Bibliographical Notes section of that chapter to have a solid understanding of all the major concepts mentioned therein. META can be used by anyone to easily experiment with algorithms and build applications, but modifying it or extending it would require at least some basic knowledge of C++ programming.

The book can be used as a textbook for an upper-level undergraduate course on information retrieval and text mining or a reference book for a graduate course to cover practical aspects of information retrieval and text mining. It should also be useful to practitioners in industry to help them acquire a wide range of practical techniques for managing and analyzing text data that they can use immediately to build various interesting real-world applications.

Acknowledgments

This book is the result of many people's help. First and foremost, we want to express our sincere thanks to Edward A. Fox for his invitation to write this book for the ACM Book Series in the area of Information Retrieval and Digital Libraries, of which he is the Area Editor. We are also grateful to Tamer Ozsu, Editor-in-Chief of ACM Books, for his support and useful comments on the book proposal. Without their encouragement and support this book would have not been possible. Next, we are deeply indebted to Edward A. Fox, Donna Harman, Bing Liu, and Jimmy Lin for thoroughly reviewing the initial draft of the book and providing very useful feedback and constructive suggestions. While we were not able to fully implement all their suggestions, all their reviews were extremely helpful and led to significant improvement of the quality of the book in many ways; naturally, any remaining errors in the book are solely the responsibility of the authors.

Throughout the process of writing the book, we received strong support and great help from Diane Cerra, Executive Editor at Morgan & Claypool Publishers, whose regular reminders and always timely support are key factors that prevented us from having the risk of taking "forever" to finish the book; for this, we are truly grateful to her. In addition, we would like to thank Sara Kreisman for copyediting and Paul C. Anagnostopoulos and his production team at Windfall Software (Ted

to thank all the people who have helped with these two MOOCs, especially TAs Hussein Hazimeh and Alex Morales, and UIUC instruction support staff Jason Mock, Shannon Bicknell, Katie Woodruff, and Edward Noel Dignan, and the Head of Computer Science Department, Rob Rutenbar, whose encouragement, support, and help are all essential for these two MOOCs to happen. The first author also wants to thank UIUC for allowing him to use the sabbatical leave in Fall 2015 to work on this book. Special thanks are due to Chase Geigle, co-founder of META. In addition to all the above, the second author would like to thank Chase Geigle, Jason Cho, and Urvashi Khandelwal (among many others) for insightful discussion and encouragement.

Finally, we would like to thank all our family members, particularly our wives, Mei and Kai, for their love and support. The first author wants to further thank his brother Chengxing for the constant intellectual stimulation in their regular research discussions and his parents for cultivating his passion for learning and sharing knowledge with others.

ChengXiang Zhai
Sean Massung
June 2016

PART I

OVERVIEW AND BACKGROUND

1 Introduction

In the last two decades, we have experienced an explosive growth of online information. According to a study done at University of California Berkeley back in 2003: ". . . the world produces between 1 and 2 exabytes (1018 petabytes) of unique information per year, which is roughly 250 megabytes for every man, woman, and child on earth. Printed documents of all kinds comprise only .03% of the total." [Lyman et al. 2003]

A large amount of online information is textual information (i.e., in natural language text). For example, according to the Berkeley study cited above: "Newspapers represent 25 terabytes annually, magazines represent 10 terabytes . . . office documents represent 195 terabytes. It is estimated that 610 billion emails are sent each year representing 11,000 terabytes." Of course, there are also blog articles, forum posts, tweets, scientific literature, government documents, etc. Roe [2012] updates the email count from 610 billion emails in 2003 to 107 *trillion* emails sent in 2010. According to a recent IDC report report [Gantz & Reinsel 2012], from 2005 to 2020, the digital universe will grow by a factor of 300, from 130 exabytes to 40,000 exabytes, or 40 trillion gigabytes.

While, in general, all kinds of online information are useful, textual information plays an especially important role and is arguably the most useful kind of information for the following reasons.

Text (natural language) is the most natural way of encoding human knowledge. As a result, most human knowledge is encoded in the form of text data. For example, scientific knowledge almost exclusively exists in scientific literature, while technical manuals contain detailed explanations of how to operate devices.

Text is by far the most common type of information encountered by people. Indeed, most of the information a person produces and consumes daily is in text form.

Text is the most expressive form of information in the sense that it can be used to describe other media such as video or images. Indeed, image search engines such as those supported by Google and Bing often rely on matching companion text of images to retrieve "matching" images to a user's keyword query.

The explosive growth of online text information has created a strong demand for intelligent software tools to provide the following two related services to help people manage and exploit big text data.

Text Retrieval.   The growth of text data makes it impossible for people to consume the data in a timely manner. Since text data encode much of our accumulated knowledge, they generally cannot be discarded, leading to, e.g., the accumulation of a large amount of literature data which is now beyond any individual's capacity to even skim over. The rapid growth of online text information also means that no one can possibly digest all the new information created on a daily basis. Thus, there is an urgent need for developing intelligent text retrieval systems to help people get access to the needed relevant information quickly and accurately, leading to the recent growth of the web search industry. Indeed, web search engines like Google and Bing are now an essential part of our daily life, serving millions of queries daily. In general, search engines are useful anywhere there is a relatively large amount of text data (e.g., desktop search, enterprise search or literature search in a specific domain such as PubMed).

Text Mining.   Due to the fact that text data are produced by humans for communication purposes, they are generally rich in semantic content and often contain valuable knowledge, information, opinions, and preferences of people. As such, they offer great opportunity for discovering various kinds of knowledge useful for many applications, especially knowledge about human opinions and preferences, which is often directly expressed in text data. For example, it is now the norm for people to tap into opinionated text data such as product reviews, forum discussions, and social media text to obtain opinions about topics interesting to them and optimize various decision-making tasks such as purchasing a product or choosing a service. Once again, due to the overwhelming amount of information, people need intelligent software tools to help discover relevant knowledge to optimize decisions or help them complete their tasks more efficiently. While the technology for supporting text mining is not yet as mature as search engines for supporting text access, sig-

nificant progress has been made in this area in recent years, and specialized text mining tools have now been widely used in many application domains.

In contrast to structured data, which conform to well-defined schemas and are thus relatively easy for computers to handle, text has less explicit structure, so the development of intelligent software tools discussed above requires computer processing to understand the content encoded in text. The current technology of natural language processing has not yet reached a point to enable a computer to precisely understand natural language text (a main reason why humans often should be involved in the loop), but a wide range of statistical and heuristic approaches to management and analysis of text data have been developed over the past few decades. They are usually very robust and can be applied to analyze and manage text data in any natural language, and about any topic. This book intends to provide a systematic introduction to many of these approaches, with an emphasis on covering the most useful knowledge and skills required to build a variety of practically useful text information systems.

The two services discussed above (i.e., text retrieval and text mining) conceptually correspond to the two natural steps in the process of analyzing any "big text data" as shown in Figure 1.1. While the raw text data may be large, a specific application often requires only a small amount of most relevant text data, thus conceptually, the very first step in any application should be to identify the *relevant text data* to a particular application or decision-making problem and avoid the unnecessary processing of large amounts of non-relevant text data. This first step of converting the raw big text data into much smaller, but highly relevant text data is often accomplished by techniques of text retrieval with help from users (e.g., users may use multiple queries to collect all the relevant text data for a decision problem). In this first step, the main goal is to connect users (or applications) with the most relevant text data.
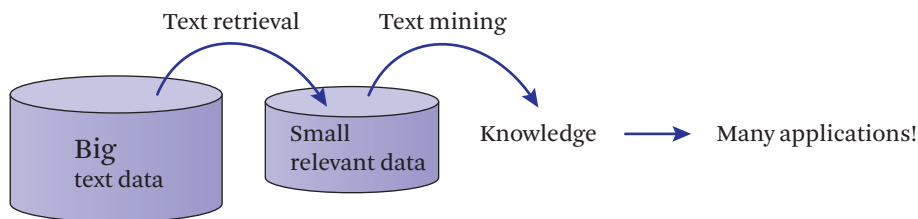
Figure 1.1   **Text retrieval and text mining are two main techniques for analyzing big text data.**

Once we obtain a small set of most relevant text data, we would need to further analyze the text data to help users digest the content and knowledge in the text data. This is the text mining step where the goal is to further discover knowledge and patterns from text data so as to support a user's task. Furthermore, due to the need for assessing trustworthiness of any discovered knowledge, users generally have a need to go back to the original raw text data to obtain appropriate context for interpreting the discovered knowledge and verify the trustworthiness of the knowledge, hence a search engine system, which is primarily useful for text access, also has to be available in any text-based decision-support system for supporting knowledge provenance. The two steps are thus conceptually interleaved, and a full-fledged intelligent text information system must integrate both in a unified framework.

It is worth pointing out that put in the context of "big data," text data is very different from other kinds of data because it is generally produced directly by humans and often also meant to be consumed by humans as well. In contrast, other data tend to be machine-generated data (e.g., data collected by using all kinds of physical sensors). Since humans can understand text data far better than computers can, involvement of humans in the process of mining and analyzing text data is absolutely crucial (much more necessary than in other big data applications), and how to optimally divide the work between humans and machines so as to optimize the collaboration between humans and machines and maximize their "combined intelligence" with minimum human effort is a general challenge in all applications of text data management and analysis. The two steps discussed above can be regarded as two different ways for a text information system to assist humans: information retrieval systems assist users in finding from a large collection of text data the most relevant text data that are actually needed for solving a specific application problem, thus effectively turning big raw text data into much smaller relevant text data that can be more easily processed by humans, while text mining application systems can assist users in analyzing patterns in text data to extract and discover useful actionable knowledge directly useful for task completion or decision making, thus providing more direct task support for users.

With this view, we partition the techniques covered in the book into two parts to match the two steps shown in Figure 1.1, which are then followed by one chapter to discuss how all the techniques may be integrated in a unified text information system. The book attempts to provide a complete coverage of all the major concepts, techniques, and ideas in information retrieval and text data mining from a practical viewpoint. It includes many hands-on exercises designed with a companion software toolkit META to help readers learn how to apply techniques of information

retrieval and text mining to real-world text data and learn how to experiment with and improve some of the algorithms for interesting application tasks. This book can be used as a textbook for computer science undergraduates and graduates, library and information scientists, or as a reference book for practitioners working on relevant application problems in analyzing and managing text data.

1.1 Functions of Text Information Systems

From a user's perspective, a text information system (TIS) can offer three distinct, but related capabilities, as illustrated in Figure 1.2.

Information Access. This capability gives a user access to the useful information when the user needs it. With this capability, a TIS can connect the right information with the right user at the right time. For example, a search engine enables a user to access text information through querying, whereas a recommender system can push relevant information to a user as new information items become available. Since the main purpose of Information Access is to connect a user with relevant information, a TIS offering this capability
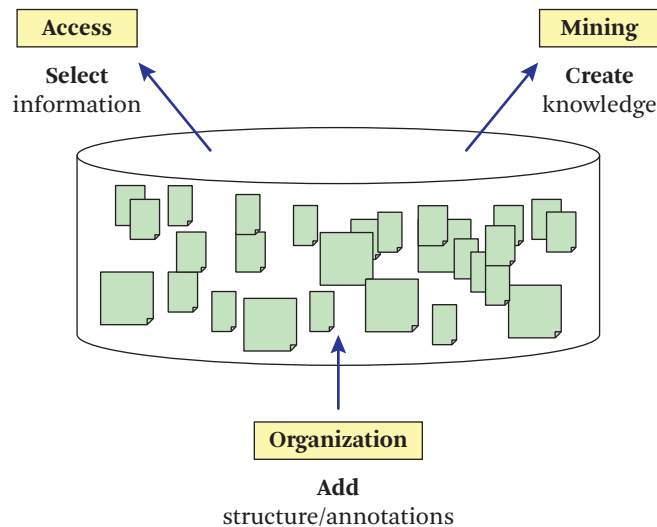
Figure 1.2 Information access, knowledge acquisition, and text organization are three major capabilities of a text information system with text organization playing a supporting role for information access and knowledge acquisition. Knowledge acquisition is also often referred to as text mining.

generally only does minimum analysis of text data sufficient for matching relevant information with a user's information need, and the original information items (e.g., web pages) are often delivered to the user in their original form, though summaries of the delivered items are often provided. From the perspective of text analysis, a user would generally need to read the information items to further digest and exploit the delivered information.

Knowledge Acquisition (Text Analysis).   This capability enables a user to acquire useful knowledge encoded in the text data that is not easy for a user to obtain without synthesizing and analyzing a relatively large portion of the data. In this case, a TIS can analyze a large amount of text data to discover interesting patterns buried in text. A TIS with the capability of knowledge acquisition can be referred to as an analysis engine. For example, while a search engine can return relevant reviews of a product to a user, an analysis engine would enable a user to obtain directly the major positive or negative opinions about the product and to compare opinions about multiple similar products. A TIS offering the capability of knowledge acquisition generally would have to analyze text data in more detail and synthesize information from multiple text documents, discover interesting patterns, and create new information or knowledge.

Text Organization.   This capability enables a TIS to annotate a collection of text documents with meaningful (topical) structures so that scattered information can be connected and a user can navigate in the information space by following the structures. While such structures may be regarded as "knowledge" acquired from the text data, and thus can be directly useful to users, in general, they are often only useful for facilitating either information access or knowledge acquisition, or both. In this sense, the capability of text organization plays a supporting role in a TIS to make information access and knowledge acquisition more effective. For example, the added structures can allow a user to search with constraints on structures or browse by following structures. The structures can also be leveraged to perform detailed analysis with consideration of constraints on structures.

Information access can be further classified into two modes: *pull* and *push*. In the pull mode, the user takes initiative to "pull" the useful information out from the system; in this case, the system plays a passive role and waits for a user to make a request, to which the system would then respond with relevant information. This mode of information access is often very useful when a user has an *ad hoc*

information need, i.e., a temporary information need (e.g., an immediate need for opinions about a product). For example, a search engine like Google generally serves a user in pull mode. In the push mode, the system takes initiative to "push" (recommend) to the user an information item that the system believes is useful to the user. The push mode often works well when the user has a relatively stable information need (e.g., hobby of a person); in such a case, a system can know "in advance" a user's preferences and interests, making it feasible to recommend information to a user without having the user to take the initiative. We cover both modes of information access in this book.

The pull mode further consists of two complementary ways for a user to obtain relevant information: *querying* and *browsing*. In the case of querying, the user specifies the information need with a (keyword) query, and the system would take the query as input and return documents that are estimated to be relevant to the query. In the case of browsing, the user simply navigates along structures that link information items together and progressively reaches relevant information. Since querying can also be regarded as a way to navigate, in one step, into a set of relevant documents, it's clear that browsing and querying can be interleaved naturally. Indeed, a user of a web search engine often interleaves querying and browsing.

Knowledge acquisition from text data is often achieved through the process of text mining, which can be defined as mining text data to discover useful knowledge. Both the data mining community and the natural language processing (NLP) community have developed methods for text mining, although the two communities tend to adopt slightly different perspective on the problem. From a data mining perspective, we may view text mining as mining a special kind of data, i.e., text. Following the general goals of data mining, the goal of text mining would naturally be regarded as to discover and extract interesting patterns in text data, which can include latent topics, topical trends, or outliers. From an NLP perspective, text mining can be regarded as to partially understand natural language text, convert text into some form of knowledge representation and make limited inferences based on the extracted knowledge. Thus a key task is to perform *information extraction*, which often aims to identify and extract mentions of various entities (e.g., people, organization, and location) and their relations (e.g., who met with whom). In practice, of course, any text mining applications would likely involve both pattern discovery (i.e., data mining view) and information extraction (i.e., NLP view), with information extraction serving as enriching the semantic representation of text, which enables pattern

finding algorithms to generate semantically more meaningful patterns than directly working on word or string-level representations of text. Due to our emphasis on covering general and robust techniques that can work for all kinds of text data without much manual effort, we mostly adopt the data mining view in this book since information extraction techniques tend to be more language-specific and generally require much manual effort. However, it is important to stress that information extraction is an essential component in any text information system that attempts to support deeper knowledge discovery or semantic analysis.

Applications of text mining can be classified as either direct applications, where the discovered knowledge would be directly consumed by users, or indirect applications, where the discovered knowledge isn't necessarily directly useful to a user, but can indirectly help a user through better support of information access. Knowledge acquisition can also be further classified based on what knowledge is to be discovered. However, due to the wide range of variations of the "knowledge," it is impossible to use a small number of categories to cover all the variations. Nevertheless, we can still identify a few common categories which we cover in this book. For example, one type of knowledge that a TIS can discover is a set of topics or subtopics buried in text data, which can serve as a concise summary of the major content in the text data. Another type of knowledge that can be acquired from opinionated text is the overall sentiment polarity of opinions about a topic.

1.2   Conceptual Framework for Text Information Systems

Conceptually, a text information system may consist of several modules, as illustrated in Figure 1.3.

First, there is a need for a module of *content analysis* based on natural language processing techniques. This module allows a TIS to transform raw text data into more meaningful representations that can be more effectively matched with a user's query in the case of a search engine, and more effectively processed in general in text analysis. Current NLP techniques mostly rely on *statistical machine learning* enhanced with limited linguistic knowledge with variable depth of understanding of text data; shallow techniques are robust, but deeper semantic analysis is only feasible for very limited domains. Some TIS capabilities (e.g., summarization) tend to require deeper NLP than others (e.g., search). Most text information systems use very shallow NLP, where text would simply be represented as a "*bag of words*," where words are basic units for representation and the order of words is ignored (although the counts of words are retained). However, a more sophisticated representation is
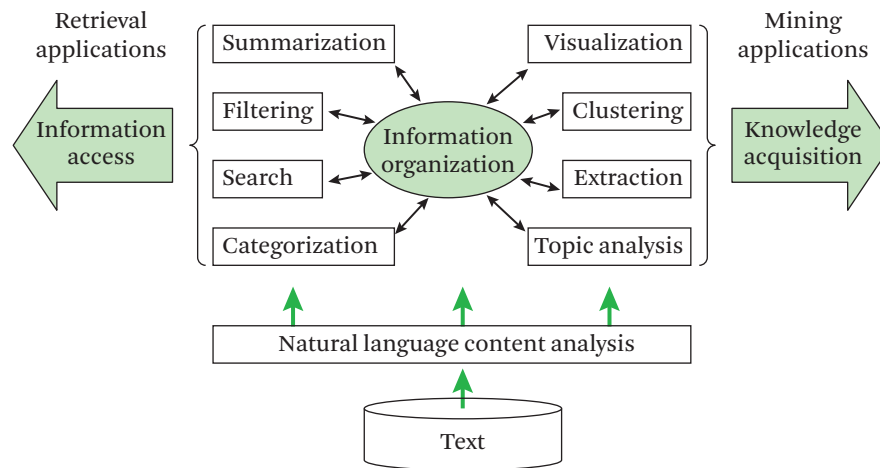
Figure 1.3 Conceptual framework of text information systems.

also possible, which may be based on recognized entities and relations or other techniques for more in-depth understanding of text.

With content analysis as the basis, there are multiple components in a TIS that are useful for users in different ways. The following are some commonly seen functions for managing and analyzing text information.

Search. Take a user's query and return relevant documents. The search component in a TIS is generally called a search engine. Web search engines are among the most useful search engines that enable users to effectively and efficiently deal with a huge amount of text data.

Filtering/Recommendation. Monitor an incoming stream, decide which items are relevant (or non-relevant) to a user's interest, and then recommend relevant items to the user (or filter out non-relevant items). Depending on whether the system focuses on recognizing relevant items or non-relevant items, this component in a TIS may be called a recommender system (whose goal is to recommend relevant items to users) or a filtering system (whose goal is to filter out non-relevant items to allow a user to keep only the relevant items). Literature recommender and spam email filter are examples of a recommender system and a filtering system, respectively.

Categorization. Classify a text object into one or several of the predefined categories where the categories can vary depending on applications. The categorization component in a TIS can annotate text objects with all kinds of meaningful categories, thus enriching the representation text data, which further enables more effective and deeper text analysis. The categories can also be used for organizing text data and facilitating text access. Subject categorizers that classify a text article into one or multiple subject categories and sentiment taggers that classify a sentence into positive, negative, or neutral in sentiment polarity are both specific examples of a text categorization system.

Summarization. Take one or multiple text documents, and generate a concise summary of the essential content. A summary reduces human effort in digesting text information and may also improve the efficiency in text mining. The summarization component of a TIS is called a summarizer. News summarizer and opinion summarizer are both examples of a summarizer.

Topic Analysis. Take a set of documents and extract and analyze topics in them. Topics directly facilitate digestion of text data by users and support browsing of text data. When combined with the companion non-textual data such as time, location, authors, and other meta data, topic analysis can generate many interesting patterns such as temporal trends of topics, spatiotemporal distributions of topics, and topic profiles of authors.

Information Extraction. Extract entities, relations of entities or other "knowledge nuggets" from text. The information extraction component of a TIS enables construction of entity-relation graphs. Such a knowledge graph is useful in multiple ways, including support of navigation (along edges and paths of the graph) and further application of graph mining algorithms to discover interesting entity-relation patterns.

Clustering. Discover groups of similar text objects (e.g., terms, sentences, documents, . . .). The clustering component of a TIS plays an important role in helping users explore an information space. It uses empirical data to create meaningful structures that can be useful for browsing text objects and obtaining a quick understanding of a large text data set. It is also useful for discovering outliers by identifying the items that do not form natural clusters with other items.

Visualization. Visually display patterns in text data. The visualization component is important for engaging humans in the process of discovering interesting patterns. Since humans are very good at recognizing visual patterns,

visualization of the results generated from various text mining algorithms is generally desirable.

This list also serves as an outline of the major topics to be covered later in this book. Specifically, search and filtering are covered first in Part II about text data access, whereas categorization, clustering, topic analysis, and summarization are covered later in Part III about text data analysis. Information extraction is not covered in this book since we want to focus on general approaches that can be readily applied to text data in *any* natural language, but information extraction often requires language-specific techniques. Visualization is also not covered due to the intended focus on algorithms in this book. However, it must be stressed that both information extraction and visualization are very important topics relevant to text data analysis and management. Readers interested in these techniques can find some useful references in the Bibliographic Notes at the end of this chapter.

1.3 Organization of the Book

The book is organized into four parts, as shown in Figure 1.4.

Part I. Overview and Background.   This part consists of the first four chapters and provides an overview of the book and background knowledge, including basic concepts needed for understanding the content of the book that some readers may not be familiar with, and an introduction to the META toolkit used for exercises in the book. This part also gives a brief overview of natural language processing techniques needed for understanding text data and obtaining informative representation of text needed in all text data analysis applications.

Part II. Text Data Access.   This part consists of Chapters 5–11, covering the major techniques for supporting text data access. This part provides a systematic discussion of the basic information retrieval techniques, including the formulation of retrieval tasks as a problem of ranking documents for a query (Chapter 5), retrieval models that form the foundation of the design of ranking functions in a search engine (Chapter 6), feedback techniques (Chapter 7), implementation of retrieval systems (Chapter 8), and evaluation of retrieval systems (Chapter 9). It then covers web search engines, the most important application of information retrieval so far (Chapter 10), where techniques for analyzing links in text data for improving ranking of text objects are introduced and application of supervised machine learning to combine multiple
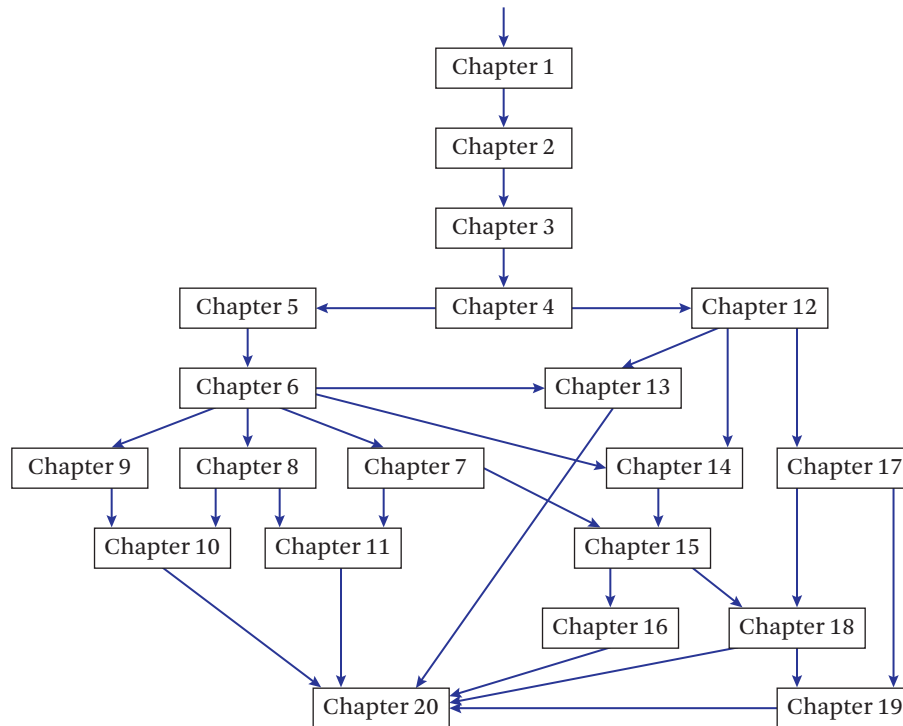
Figure 1.4   **Dependency relations among the chapters.**

features for ranking is briefly discussed. The last chapter in this part (Chapter 11) covers recommender systems which provide a "push" mode of information access, as opposed to the "pull" mode of information access supported by a typical search engine (i.e., querying by users).

Part III. Text Data Analysis. This part consists of Chapters 12–19, covering a variety of techniques for analyzing text data to facilitate user digestion of text data and discover useful topical or other semantic patterns in text data. Chapter 12 gives an overview of text analysis from the perspective of data mining, where we may view text data as data generated by humans as "subjective sensors" of the world; this view allows us to look at the text analysis problem in the more general context of data analysis and mining in general, and facilitates the discussion of joint analysis of text and non-text data. This is followed by multiple chapters covering a number of the most useful general techniques for analyzing text data without or with only minimum human effort. Specifically, Chapter 13 discusses techniques for discovering two fundamental se-

mantic relations between lexical units in text data, i.e., paradigmatic relations and syntagmatic relations, which can be regarded as an example of discovering knowledge about the natural language used to generate the text data (i.e., linguistic knowledge). Chapter 14 and Chapter 15 cover, respectively, two closely related techniques to generate and associate meaningful structures or annotations with otherwise unorganized text data, i.e., text clustering and text categorization. Chapter 16 discusses text summarization useful for facilitating human digestion of text information. Chapter 17 provides a detailed discussion of an important family of probabilistic approaches to discovery and analysis of topical patterns in text data (i.e., topic models). Chapter 18 discusses techniques for analyzing sentiment and opinions expressed in text data, which are key to discovery of knowledge about preferences, opinions, and behavior of people based on analyzing the text data produced by them. Finally, Chapter 19 discusses joint analysis of text and non-text data, which is often needed in many applications since it is in general beneficial to use as much data as possible for gaining knowledge and intelligence through (big) data analysis.

Part IV. **Unified Text Management and Analysis System.**   This last part consists of Chapter 20 where we attempt to discuss how all the techniques discussed in this book can be conceptually integrated in an operator-based unified framework, and thus potentially implemented in a general unified system for text management and analysis that can be useful for supporting a wide range of different applications. This part also serves as a roadmap for further extension of META to provide effective and general high-level support for various applications and provides guidance on how META may be integrated with many other related existing toolkits, including particularly search engine systems, database systems, natural language processing toolkits, machine learning toolkits, and data mining toolkits.

Due to our attempt to treat all the topics from a practical perspective, most of the discussions of the concepts and techniques in the book are informal and intuitive. To satisfy the needs of some readers that might be interested in deeper understanding of some topics, the book also includes an appendix with notes to provide a more detailed and rigorous explanation of a few important topics.

1.4 How to Use this Book

Due to the extremely broad scope of the topics that we would like to cover, we have to make many tradeoffs between breadth and depth in coverage. When making

such a tradeoff, we have chosen to emphasize the coverage of the basic concepts and practical techniques of text data mining at the cost of not being able to cover many advanced techniques in detail, and provide some references at the end of many chapters to help readers learn more about those advanced techniques if they wish to. Our hope is that with the foundation received from reading this book, you will be able to learn about more advanced techniques by yourself or via another resource. We have also chosen to cover more general techniques for text management and analysis and favor techniques that can be applicable to any text in any natural language. Most techniques we discuss can be implemented without any human effort or only requiring minimal human effort; this is in contrast to some more detailed analysis of text data, particularly using natural language processing techniques. Such "deep analysis" techniques are obviously very important and are indeed necessary for some applications where we would like to go in-depth to understand text in detail. However, at this point, these techniques are often not scalable and they tend to require a large amount of human effort. In practice, it would be beneficial to combine both kinds of techniques.

We envision three main (and potentially overlapping) categories of readers.

Students.   This book is specifically designed to give you hands-on experience in working with real text mining tools and applications. If used individually, we suggest first reading through Chapters 1–4 in order to get a good understanding of the prerequisite knowledge in this book. Chapters 1, 2, and 3 will familiarize you with the concepts and vocabulary necessary to understand the future chapters. Chapter 4 introduces you to the companion toolkit MᴇTA, which is used in exercises in each chapter. We hope the exercises and chapter descriptions provide inspiration to work on your own text mining project. The provided code in MᴇTA should give a large head start and allow you to focus more on your contribution.

   If used in class, there are several logical flows that an instructor may choose to take. As prerequisite knowledge, we assume some basic knowledge in probability and statistics as well as programming in a language such as C++ or Java. MᴇTA is written in modern C++, although some exercises may be accomplished only by modifying config files.

Instructors.   We have gathered a logical and cohesive collection of topics that may be combined together for various course curricula. For example, Part 1 and Part 2 of the book may be used as an undergraduate introduction to *Information Retrieval* with a focus on how search engines work. Exercises assume basic programming experience and a little mathematical background in probability and statistics. A different undergraduate course may choose to survey

the entire book as an *Introduction to Text Data Mining*, while skipping some chapters in Part 2 that are more specific to search engine implementation and applications specific to the Web. Another choice would be using all parts as a supplemental graduate textbook, where there is still some emphasis on practical programming knowledge that can be combined with reading referenced papers in each chapter. Exercises for graduate students could be implementing some methods they read in the references into META.

The exercises at the end of each chapter give students experience working with a powerful—yet easily understandable—text retrieval and mining toolkit in addition to written questions. In a programming-focused class, using the META exercises is strongly encouraged. Programming assignments can be created from selecting a subset of exercises in each chapter. Due to the modular nature of the toolkit, additional programming experiments may be created by extending the existing system or implementing other well-known algorithms that do not come with META by default. Finally, students may use components of META they learned through the exercises to complete a larger final programming project. Using different corpora with the toolkit can yield different project challenges, e.g., review summary vs. sentiment analysis.

Practitioners. Most readers in industry would most likely use this book as a reference, although we also hope that it may serve as some inspiration in your own work. As with the student user suggestion, we think you would get the most of this book by first reading the initial three chapters. Then, you may choose a chapter relevant to your current interests and delve deeper or refresh your knowledge.

Since many applications in META can be used simply via config files, we anticipate it as a quick way to get a handle on your dataset and provide some baseline results without any programming required.

The exercises at the end of each chapter can be thought of as default implementations for a particular task at hand. You may choose to include META in your work since it uses a permissive free software license. In fact, it is dual-licensed under MIT and University of Illinois/NCSA licenses. Of course, we still encourage and invite you to share any modifications, extensions, and improvements with META that are not proprietary for the benefit of all the readers.

No matter what your goal, we hope that you find this book useful and educational. We also appreciate your comments and suggestions for improvement of the book. Thanks for reading!

Bibliographic Notes and Further Reading

There are already multiple excellent text books in information retrieval (IR). Due to the long history of research in information retrieval and the fact that much foudational work has been done in 1960s, even some very old books such as van Rijsbergen [1979] and Salton and McGill [1983] and Salton [1989] remain very useful today. Another useful early book is Frakes and Baeza-Yates [1992]. More recent ones include Grossman and Frieder [2004], Witten et al. [1999], and Belew [2008]. The most recent ones are Manning et al. [2008], Croft et al. [2009], Büttcher et al. [2010], and Baeza-Yates and Ribeiro-Neto [2011]. Compared with these books, this book has a broader view of the topic of information retrieval and attempts to cover both text retrieval and text mining. While some existing books on IR have also touched some topics such as text categorization and text clustering, which we classify as text mining topics, no previous book has included an in-depth discussion of topic mining and analysis, an important family of techniques very useful for text mining. Recommender systems also seem to be missing in the existing books on IR, which we include as an alternative way to support users for text access complementary with search engines. More importantly, this book treats all these topics in a more systematic way than existing books by framing them in a unified coherent conceptual framework for managing and analyzing big text data; the book also attempts to minimize the gap between abstract explanation of algorithms and practical applications by providing a companion toolkit for many exercises. Readers who want to know more about the history of IR research and the major early milestones should take a look at the collection of readings in Sparck Jones and Willett [1997].

The topic of text mining has also been covered in multiple books (e.g., Feldman and Sanger [2007]). A major difference between this book and those is our emphasis on the integration of text mining and information retrieval with a belief that any text data application system must involve humans in the loop and search engines are essential components of any text mining systems to support two essential functions: (1) help convert a large raw text data set into a much smaller, but more relevant text data set which can be efficiently anlayzed by using a text mining algorithm (i.e., data reduction) and (2) help users verify the source text articles from which knowledge is discovered by a text mining algorithm (i.e., knowledge provenance). As a result, this book provides a more complete coverage of techniques required for developing big text data applications.

The focus of this book is on covering algorithms that are general and robust, which can be readily applied to any text data in any natural language, often with no or minimum human effort. An evitable cost of this focus is its lack of coverage

of some key techniques important for text mining, notably the information extraction (IE) techniques which are essential for text mining. We decided not to cover IE because the IE techniques tend to be language-specific and require non-trivial manual work by humans. Another reason is that many IE techniques rely on supervised machine learning approaches, which are well covered in many existing machine learning books (see, e.g., Bishop 2006, Mitchell 1997). Readers who are interested in knowing more about IE can start with the survey book [Sarawagi 2008] and review articles [Jiang 2012].

From an application perspective, another important topic missing in this book is information visualization, which is due to our focus on the coverage of models and algorithms. However, since every application system must have a user-friendly interface to allow users to optimally interact with a system, those readers who are interested in developing text data application systems will surely find it useful to learn more about user interface design. An excellent reference to start with is Hearst [2009], which also has a detailed coverage of information visualization.

Finally, due to our emphasis on breadth, the book does not cover any component algorithm in depth. To know more about some of the topics, readers can further read books in natural language processing (e.g., Jurafsky and Martin 2009, Manning and Schütze 1999), advanced books on IR (e.g., Baeza-Yates and Ribeiro-Neto [2011]), and books on machine learning (e.g., Bishop [2006]). You may find more specific recommendations of readings relevant to a particular topic in the Bibliographic Notes at the end of each chapter that covers the corresponding topic.

2 Background

This chapter contains background information that is necessary to know in order to get the most out of the rest of this book; readers who are already familiar with these basic concepts may safely skip the entire chapter or some of the sections. We first focus on some basic probability and statistics concepts required for most algorithms and models in this book. Next, we continue our mathematical background with an overview of some concepts in information theory that are often used in many text mining applications. The last section introduces the basic idea and problem setup of machine learning, particularly supervised machine learning, which is useful for *classification*, *categorization*, or *text-based prediction* in the text domain. In general, machine learning is very useful for many information retrieval and data mining tasks.

2.1 Basics of Probability and Statistics

As we will see later in this chapter and in many other chapters, probabilistic or statistical models play a very important role in text mining algorithms. This section gives every reader a sufficient background and vocabulary to understand these probabilistic and statistical approaches covered in the later chapters of the book.

A probability distribution is a way to assign likelihood to an event in some probability space Ω. As an example, let our probability space be a six-sided die. Each side has a different color. Thus, $\Omega = \{red, orange, yellow, green, blue, purple\}$ and an event is the act of rolling the die and observing a color.

We can quantify the uncertainty of rolling the die by declaring a **probability distribution** over all possible events. Assuming we have a fair die, the probability of rolling any specific color is $\frac{1}{6}$, or about 16%. We can represent our probability distribution as a collection of probabilities such as

$$\theta = \left\{ \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6} \right\},$$

where the first index corresponds to $p(\text{red}) = \frac{1}{6}$, the second index corresponds to $p(\text{orange}) = \frac{1}{6}$, and so on. But what if we had an unfair die? We could use a different probability distribution θ' to model events concerning it:

$$\theta' = \left\{ \frac{1}{3}, \frac{1}{3}, \frac{1}{12}, \frac{1}{12}, \frac{1}{12}, \frac{1}{12} \right\}.$$

In this case, red and orange are assumed to be rolled more often than the other colors. Be careful to note the difference between the sample space Ω and the defined probability model θ used to quantify its uncertainty. In our text mining tasks, we usually try to estimate θ given some knowledge about Ω. The different methods to estimate θ will determine how accurate or useful the probabilistic model is.

Consider the following notation:

$$x \sim \theta.$$

We read this as *x is drawn from theta*, or the **random variable** x is drawn from the probability distribution θ. The random variable x takes on each value from Ω with a certain probability defined by θ. For example, if we had $x \sim \theta'$, then there is a $\frac{2}{3}$ chance that x is either red or orange.

In our text application tasks, we usually have Ω as V, the vocabulary of some text corpus. For example, the vocabulary could be

$$V = \{a, and, apple, \ldots, zap, zirconium, zoo\}$$

and we could model the text data with a probability distribution θ. Thus, if we have some word w we can write $p(w \mid \theta)$ (read as *the probability of w given θ*). If w is the word *data*, we might have $p(w = \text{data} \mid \theta) = 0.003$ or equivalently $p_\theta(w = \text{data}) = 0.003$.

In our examples, we have only considered **discrete probability distributions**. That is, our models only assign probabilities for a finite (discrete) set of outcomes. In reality, there are also **continuous probability distributions**, where there are an infinite number of "events" that are not countable. For example, the normal (or Gaussian) distribution is a continuous probability distribution that assigns real-valued probabilities according to some parameters. We will discuss continuous distributions as necessary in later chapters. For now, though, it's sufficient to understand that we can use a discrete probability distribution to model the probability of observing a single word in a vocabulary V.

The Kolmogorov axioms describe facts about probability distributions in general (both discrete and continuous). We discuss them now, since they are a good sanity check when designing your own models. A valid probability distribution θ with probability space Ω must satisfy the following three axioms.

- Each event has a probability between zero and one:

$$0 \le p_\theta(\omega \in \Omega) \le 1. \tag{2.1}$$

- An event not in Ω has probability zero, and the probability of any event occurring from Ω is one:

$$p_\theta(\omega') = 0, \omega' \notin \Omega \quad \text{and} \quad p_\theta(\Omega) = 1. \tag{2.2}$$

- The probability of all (disjoint) events sums to one:

$$\sum_{\omega \in \Omega} p_\theta(\omega) = 1. \tag{2.3}$$

Note that, strictly speaking, an event is defined as a subset of the probability space Ω, and we say that an event happens if and only if the outcome from a random experiment (i.e., randomly drawing an outcome from Ω) is in the corresponding subset of outcomes defined by the event. Thus, it is easy to understand that the special event corresponding to the empty subset is an impossible event with a probability of zero, whereas the special event corresponding to the complete set Ω itself always happens and so has a probability of 1.0. As a special case, we can consider an event space with only those events that each have precisely one element of outcome, which is exactly what we assumed when talking about a distribution over all the words. Here each word corresponds to the event defined by the subset with the word as the only element; clearly, such an event happens if and only if the outcome is the corresponding word.

2.1.1 Joint and Conditional Probabilities

For this section, let's modify our original die rolling example. We will keep the original distribution as θ_C, indicating the color probabilities:

$$\theta_C = \left\{ \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6} \right\}.$$

Let's also assume that each color is represented by a particular shape. This makes our die look like

$$\{\ \blacksquare\ ,\ \blacksquare\ ,\ \bigcirc\ ,\ \bullet\ ,\ \bullet\ ,\ \blacktriangle\ \}$$

where the colors of the shapes are, red, orange, yellow, blue, green, and purple, respectively.

We can now create another distribution for the shape θ_S. Let each index in θ_S represent $p(\text{square})$, $p(\text{circle})$, $p(\text{triangle})$, respectively. That gives

$$\theta_S = \left\{ \frac{1}{3}, \frac{1}{2}, \frac{1}{6} \right\}.$$

Then we can let $x_C \sim \theta_C$ represent the color random variable and let $x_S \sim \theta_S$ represent the shape random variable. We now have two variables to work with.

A **joint probability** measures the likelihood that two events occur simultaneously. For example, what is the probability that $x_C = \text{red}$ and $x_S = \text{circle}$? Since there are no red circles, this has probability zero. How about $p(x_C = \text{green}, x_S = \text{circle})$? This notation signifies the joint probability of the two random variables. In this case, the joint probability is $\frac{1}{6}$ because there is only one green circle.

Consider a modified die:

$$\{\ \blacksquare\ ,\ \blacksquare\ ,\ \bigcirc\ ,\ \bullet\ ,\ \bullet\ ,\ \blacktriangle\ \}$$

where we changed the color of the blue circle (the fourth element in the set) to green. Thus, we now have two green circles instead of one green and one blue. What would $p(x_C = \text{green}, x_S = \text{circle})$ be? Since two out of the six elements satisfy both these criteria, the answer is $\frac{2}{6} = \frac{1}{3}$. As another example, if we had a 12-sided fair die with 5 green circles and 7 other combinations of shape and color, then $p(x_C = \text{green}, x_S = \text{circle}) = \frac{5}{12}$.

A **conditional probability** measures the likelihood that one event occurs given that another event has already occurred. Let's use the original die with six unique colors. Say we know that a square was rolled. With that information, what is the probability that the color is red? How about purple? We can write this as $p(x_C = \text{red} \mid x_S = \text{square})$. Since we know there are two squares, of which one is red, $p(\text{red} \mid \text{square}) = \frac{1}{2}$.

We can write the conditional probabilities for two random variables X and Y based on their joint probability with the following equation:

$$p(X = x \mid Y = y) = \frac{p(X = x, Y = y)}{p(Y = y)}. \tag{2.4}$$

The numerator $p(X = x, Y = y)$ is the probability of exactly the configuration we're looking for (i.e., both x and y have been observed), which is normalized by $p(Y = y)$, the probability that the condition is true (i.e., y has been observed). Using this knowledge, we can calculate $p(x_C = \text{green} \mid x_S = \text{circle})$:

$$p(x_C = \text{green} \mid x_S = \text{circle}) = \frac{p(x_C = \text{green}, x_S = \text{circle})}{p(x_S = \text{circle})} = \frac{1/6}{1/2} = \frac{1}{3}.$$

One other important concept to mention is **independence**. In the previous examples, the two random variables were **dependent**, meaning the value of one will influence the value of the other. Consider another situation where we have $c_1, c_2 \sim \theta_C$. That is, we draw two colors from the color distribution. Does the knowledge of c_1 inform the probability of c_2? No, since each draw is done "independently" of the other. In the case where two random variables X and Y are independent, $p(X, Y) = p(X)p(Y)$. Can you see why this is the case?

Both conditional and joint probabilities can be used to answer interesting questions about text. For example, given a document, what is the probability of observing the word *information* and *retrieval* in the same sentence? What is the probability of observing *retrieval* if we know *information* has occurred?

2.1.2 Bayes' Rule

Bayes' rule may be derived using the definition of conditional probability:

$$p(X \mid Y) = \frac{p(X, Y)}{p(Y)} \quad \text{and} \quad p(Y \mid X) = \frac{p(Y, X)}{p(X)}.$$

Therefore, setting the two joint probabilities equal,

$$p(X \mid Y)p(Y) = p(X, Y) = p(Y \mid X)p(X).$$

We can simplify them as

$$p(X \mid Y) = \frac{p(Y \mid X)p(X)}{p(Y)}. \tag{2.5}$$

The above formula is known as Bayes' rule, named after the Reverend Thomas Bayes (1701–1761). This rule has widespread applications. In this book, you will see heavy use of this formula in the text categorization chapter as well as the topic analysis chapter, among others. We will leave it up to the individual chapters to explain their use of this rule in their implementation. Essentially, though, Bayes' rule can be used to make inference about a hypothesis based on the observed evidence related to the hypothesis.

Specifically, we may view random variable X as denoting our hypothesis, and Y as denoting the observed evidence. $p(X)$ can thus be interpreted as our prior belief about which hypothesis is true; it is "prior" because it is our belief *before* we have any knowledge about evidence Y. In contrast, $p(X \mid Y)$ encodes our posterior belief about the hypothesis since it is our belief *after* knowing evidence Y. Bayes' rule is seen to connect the prior belief and posterior belief, and provide a way to update the prior belief $p(X)$ based on the likelihood of the observed evidence Y and obtain the posterior belief $p(X \mid Y)$. It is clear that if X and Y are independent, then no updating will happen as in this case, $p(X \mid Y) = p(X)$.

2.1.3   Coin Flips and the Binomial Distribution

In most discussions on probability, a good example to investigate is flipping a coin. For example, we may be interested in modeling the presence or absence of a particular word in a text document, which can be easily mapped to a coin flipping problem. There are two possible outcomes in coin flipping: heads or tails. The probability of heads is denoted as θ, which means the probability of tails is $1 - \theta$.

To model the probability of success (in our case, "heads"), we can use the Bernoulli distribution. The Bernoulli distribution gives the probability of success for a single event—flipping the coin once. If we want to model n throws and find the probability of k successes, we instead use the binomial distribution. The binomial distribution is a discrete distribution since k is an integer. We can write it as

$$p(k \text{ heads}) = \binom{n}{k}\theta^k(1 - \theta)^{n-k}. \tag{2.6}$$

We can also write it as follows:

$$p(k \text{ heads}) = \frac{n!}{k!(n - k)!}\theta^k(1 - \theta)^{n-k}. \tag{2.7}$$

But why is it this formula? Well, let's break it apart. If we have n total binary trials, and want to see k heads, that means we must have flipped k heads and $n - k$ tails. The probability of observing each of the k heads is θ, while the probability of observing each of the remaining $n - k$ tails is $1 - \theta$. Since we assume all these flips are independent, we simply multiply all the outcomes together. Since we don't care about the order of the outcomes, we additionally multiply by the number of possible ways to choose k items from a set of n items.

What if we do care about the order of the outcomes? For example, what is the probability of observing the particular sequence of outcomes (h, t, h, h, t) where h and t denote heads and tails, respectively? Well, it is easy to see that the probability

of observing this sequence is simply the product of observing each event, i.e., $\theta \times (1 - \theta) \times \theta \times \theta \times (1 - \theta) = \theta^3(1 - \theta)^2$ with no adjustment for different orders of observing three heads and two tails.

The more commonly used multinomial distribution in text analysis, which models the probability of seeing a word in a particular scenario (e.g., in a document), is very similar to this Bernoulli distribution, just with more than two outcomes.

2.1.4 Maximum Likelihood Parameter Estimation

Now that we have a model for our coin flipping, how can we estimate its parameters given some observed data? For example, maybe we observe the data D that we discussed above where $n = 5$:

$$D = \{h, t, h, h, t\}.$$

Now we would like to figure out what θ is based on the observed data. Using maximum likelihood estimation (MLE), we choose the θ that has the highest likelihood given our data, i.e., choose the θ such that the probability of observed data is maximized.

To compute the MLE, we would first write down the likelihood function, i.e., $p(D \mid \theta)$, which is $\theta^3(1 - \theta)^2$ as we explained earlier. The problem is thus reduced to find the θ that maximizes the function $f(\theta) = \theta^3(1 - \theta)^2$. Equivalently, we can attempt to maximize the log-likelihood: $\log f(\theta) = 3 \log \theta + 2 \log(1 - \theta)$, since logarithm transformation preserves the order of values. Using knowledge of calculus, we know that a necessary condition for a function to achieve a maximum value at a θ value is that the derivative at the same θ value is zero. Thus, we just need to solve the following equation:

$$\frac{d \log f(\theta)}{d\theta} = \frac{3}{\theta} - \frac{2}{1 - \theta} = 0,$$

and we easily find that the solution is $\theta = 3/5$.

More generally, let H be the number of heads and T be the number of tails. The MLE of the probability of heads is given by:

$$\theta_{MLE} = \arg \max_{\theta} p(D \mid \theta)$$

$$= \arg \max_{\theta} \theta^H (1 - \theta)^T$$

$$= \frac{H}{H + T}.$$

The notation arg max represents the argument (i.e., θ in this case) that makes the likelihood function (i.e., $p(D \mid \theta)$) reach its maximum. Thus, the value of an arg max expression stays the same if we perform any monotonic transformation of the function inside arg max. This is why we could use the logarithm transformation in the example above, which made it easier to compute the derivative.

The solution to MLE shown above should be intuitive: the θ that maximizes our data likelihood is just the ratio of heads. It is a general characteristic of the MLE that the estimated probability is the normalized counts of the corresponding events denoted by the probability. As an example, the MLE of a multinomial distribution (which will be further discussed in detail later in the book) gives each possible outcome a probability proportional to the observed counts of the outcome. Note that a consequence of this is that all unobserved outcomes would have a zero probability according to MLE. This is often not reasonable especially when the data sample is small, a problem that motivates Bayesian parameter estimation which we discuss below.

2.1.5   Bayesian Parameter Estimation

One potential problem of MLE is that it is often inaccurate when the size of the data sample is small since it always attempts to fit the data as well as possible. Consider an extreme example of observing just two data points of flipping a coin which happen to be all heads. The MLE would say that the probability of heads is 1.0 while the probability of tails is 0. Such an estimate is intuitively inaccurate even though it maximizes the probability of the observed two data points.

This problem of "overfitting" can be addressed and alleviated by considering the uncertainty on the parameter and using Bayesian parameter estimation instead of MLE. In Bayesian parameter estimation, we consider a distribution over all the possible values for the parameter; that is, we treat the parameter itself as a random variable.

Specifically, we may use $p(\theta)$ to represent a distribution over all possible values for θ, which encodes our prior belief about what value is the true value of θ, while the data D provide evidence for or against that belief. The prior belief $p(\theta)$ can then be updated based on the observed evidence. We'll use Bayes' rule to rewrite $p(\theta \mid D)$, or our belief of the parameters given data, as

$$p(\theta \mid D) = \frac{p(D \mid \theta)p(\theta)}{p(D)}, \tag{2.8}$$

where $p(D)$ can be calculated by summing over all configurations of θ. For a continuous distribution, that would be

$$p(D) = \int_{\theta'} p(\theta')p(D \mid \theta')d\theta' \tag{2.9}$$

which means the probability for a particular θ is

$$p(\theta \mid D) = \frac{p(D \mid \theta)p(\theta)}{\int_{\theta'} p(\theta')p(D \mid \theta')d\theta'}. \tag{2.10}$$

We have special names for these quantities:

- $p(\theta \mid D)$: the posterior probability of θ
- $p(\theta)$: the prior probability of θ
- $p(D \mid \theta)$: the likelihood of D
- $\int_{\theta'} p(\theta')p(D \mid \theta')d\theta'$: the marginal likelihood of D

The last one is called the marginal likelihood because the integration "marginalizes out" (removes) the parameter θ from the equation. Since the likelihood of the data remains constant, observing the constraint that $p(\theta \mid D)$ must sum to one over all possible values of θ, we usually just say

$$p(\theta \mid D) \propto p(\theta)p(D \mid \theta).$$

That is, the posterior is proportional to the prior times the likelihood.

The posterior distribution of the parameter θ fully characterizes the uncertainty of the parameter value and can be used to infer any quantity that depends on the parameter value, including computing a point estimate of the parameter (i.e., a single value of the parameter). There are multiple ways to compute a point estimate based on a posterior distribution. One possibility is to compute the mean of the posterior distribution, which is given by the weighted sum of probabilities and the parameter values. For a discrete distribution,

$$E[X] = \sum_x xp(x) \tag{2.11}$$

while in a continuous distribution,

$$E[X] = \int_x xf(x)dx \tag{2.12}$$

Sometimes, we are interested in using the mode of the posterior distribution as our estimate of the parameter, which is called Maximum a Posteriori (MAP) estimate, given by:

$$\theta_{MAP} = \arg\max_{\theta} p(\theta \mid D) = \arg\max_{\theta} p(D \mid \theta)p(\theta). \tag{2.13}$$

Here it is easy to see that the MAP estimate would deviate from the MLE with consideration of maximizing the probability of the parameter according to our prior belief encoded as $p(\theta)$. It is through the use of appropriate prior that we can address the overfitting problem of MLE since our prior can strongly prefer an estimate where neither heads, nor tails should have a zero probability.

For a continuation and more in-depth discussion of this material, consult Appendix A.

2.1.6 Probabilistic Models and Their Applications

With the statistical foundation from the previous sections, we can now start to see how we might apply a probabilistic model to text analysis.

In general, in text processing, we would be interested in a probabilistic model for text data, which defines distributions over sequences of words. Such a model is often called statistical language model, or a generative model for text data (i.e., a probabilistic model that can be used for sampling sequences of words).

As we started to explain previously, we usually treat the sample space Ω as V, the set of all observed words in our corpus. That is, we define probability distributions over words from our dataset, which are essentially multinomial distributions if we do not consider the order of words. While there are many more sophisticated models for text data (see, e.g., Jelinek [1997]), this simplest model (often called unigram language model) is already very useful for a number of tasks in text data management and analysis due to the fact that the words in our vocabulary are very well designed meaningful basic units for human communications.

For now, we can discuss the general framework in which statistical models are "learned." Learning a model means estimating its parameters. In the case of a distribution over words, we have one parameter for each element in V. The workflow looks like the following.

1. Define the model.
2. Learn its parameters.
3. Apply the model.

The first step has already been addressed. In our example, we wish to capture the probabilities of individual words occurring in our corpus. In the second step, we need to figure out actually how to set the probabilities for each word. One obvious way would be to calculate the probability of each individual word in the corpus itself. That is, the count of a unique word w_i divided by the total number of words

in the corpus could be the value of $p(w_i \mid \theta)$. This can be shown to be the solution of the MLE of the model. More sophisticated models and their parameter estimation will be discussed later in the book. Finally, once we have θ defined, what can we actually do with it? One use case would be analyzing the probability of a specific subset of words in the corpus, and another could be observing unseen data and calculating the probability of seeing the words in the new text. It is often possible to design the model such that the model parameters would encode the knowledge we hope to discover from text data. In such a case, the estimated model parameters can be directly used as the output (result) of text mining.

Please keep in mind that probabilistic models are a general tool and don't only have to be used for text analysis—that's just our main application!

2.2 Information Theory

Information theory deals with uncertainty and the transfer or storage of quantified information in the form of bits. It is applied in many fields, such as electrical engineering, computer science, mathematics, physics, and linguistics. A few concepts from information theory are very useful in text data management and analysis, which we introduce here briefly. The most important concept of information theory is **entropy**, which is a building block for many other measures.

The problem can be formally defined as the quantified uncertainty in predicting the value of a random variable. In the common example of a coin, the two values would be 1 or 0 (depicting heads or tails) and the random variable representing these outcomes is X. In other words,

$$X = \begin{cases} 1 & \text{if heads} \\ 0 & \text{if tails.} \end{cases}$$

The more random this random variable is, the more difficult the prediction of heads or tails will be. How does one quantitatively measure the randomness of a random variable like X? This is precisely what entropy does.

Roughly, the entropy of a random variable X, $H(X)$, is a measure of expected number of bits needed to represent the outcome of an event $x \sim X$. If the outcome is known (completely certain), we don't need to represent any information and $H(X) = 0$. If the outcome is unknown, we would like to represent the outcome in bits as efficiently as possible. That means using fewer bits for common occurrences and more bits when the event is less likely. Entropy gives us the expected number

of bits for any $x \sim X$ using the formula

$$H(X) = - \sum_{x \in X} p(x) \log_2 p(x). \tag{2.14}$$

In the cases where we have $\log_2 0$, we generally just define this to be 0 since $\log_2 0$ is undefined. We will get different $H(X)$ for different random variables X.

The exact theory and reasoning behind this formula are beyond the scope of this book, but it suffices to say that $H(X) = 0$ means there is no randomness, $H(X) = 1$ means there is complete randomness in that all events are equally likely. Thus, the amount of randomness varies from 0 to 1. For our coin example where the sample space is two events (heads or tails), the entropy function looks like

$$H(X) = -p(X = 0) \log_2 p(X = 0) - p(X = 1) \log_2 p(X = 1).$$

For a fair coin, we would have $p(X = 1) = p(X = 0) = \frac{1}{2}$. To calculate $H(X)$, we'd have the calculation

$$H(X) = -\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{2} \log_2 \frac{1}{2} = 1,$$

whereas for a completely biased coin with $p(X = 1) = 1$, $p(X = 0) = 0$ we would have

$$H(X) = -0 \log_2 0 - 1 \log_2 1 = 0.$$

For this example, we had only two possible outcomes (i.e., a binary random variable). As we can see from the formula, this idea of entropy easily generalizes to random variables with more than two outcomes; in those cases, the sum is over more than two elements.

If we plot $H(X)$ for our coin example against the probability of heads $p(X = 1)$, we receive a plot like the one shown in Figure 2.1. At the two ends of the x-axis, the probability of $X = 1$ is either very small or very large. In both these cases, the entropy function has a low value because the outcome is not very random. The most random is when $p(X = 1) = \frac{1}{2}$. In that case, $H(X) = 1$, the maximum value. Since the two probabilities are symmetric, we get a symmetric inverted U-shape as the plot of $H(X)$ as $p(X = 1)$ varies.

It's a good exercise to consider when a particular random variable (not just the coin example) has a maximum or minimal value. In particular, let's think about some special cases. For example, we might have a random variable Y that always takes a value of 1. Or, there's a random variable Z that is equally likely to take a value of 1, 2, or 3. In these cases, $H(Y) < H(Z)$ since the outcome of Y is much
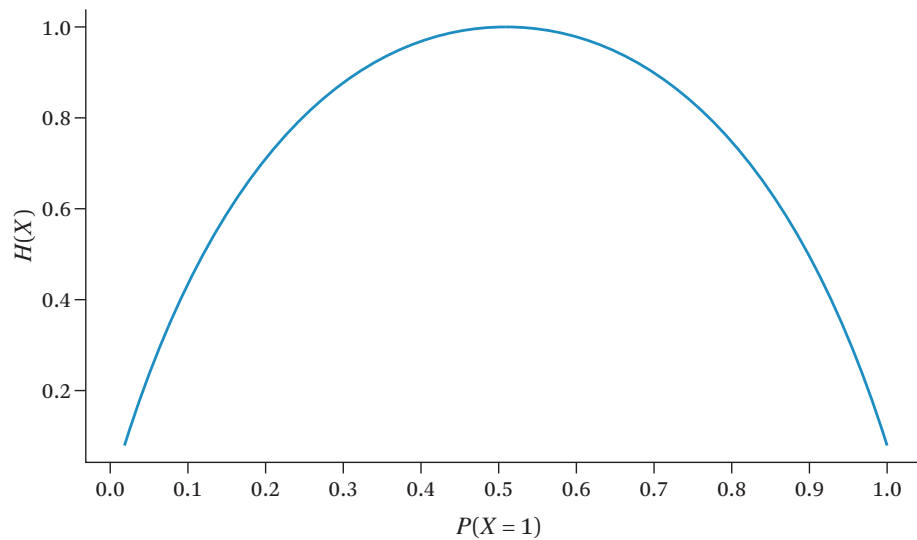
Figure 2.1   **Entropy as a measure of randomness of a random variable.**

easier to predict than the outcome of Z. This is precisely what entropy captures. You can calculate $H(Y)$ and $H(Z)$ to confirm this answer.

For our applications, it may be useful to consider the entropy of a word w in some context. Here, high-entropy words would be harder to predict. Let W be the random variable that denotes whether a word occurs in a document in our corpus. Say $W = 1$ if the word occurs and $W = 0$ otherwise. How do you think $H(W_{the})$ compares to $H(W_{computer})$? The entropy of the word *the* is close to zero since it occurs everywhere. It's not surprising to see this word in a document, thus it is easy to predict that $W_{the} = 1$. This case is just like the biased coin that always lands one way. The word *computer*, on the other hand, is a less common word and is harder to predict whether it occurs or not, so the entropy will be higher.

When we attempt to quantify uncertainties of conditional probabilities, we can also define conditional entropy $H(X \mid Y)$, which indicates the expected uncertainty of X given that we observe Y, where the expectation is taken under the distribution of all possible values of Y. Intuitively, if X is completely determined by Y, then $H(X \mid Y) = 0$ since once we know Y, there would be no uncertainty in X, whereas if X and Y are independent, then $H(X \mid Y)$ would be the same as the original entropy of X, i.e., $H(X \mid Y) = H(X)$ since knowing Y does not help at all in resolving the uncertainty of X.

Another useful concept is mutual information defined on two random variables, $I(X;Y)$, which is defined as the reduction of entropy of X due to knowledge about another random variable Y, i.e.,

$$I(X;Y) = H(X) - H(X \mid Y). \qquad (2.15)$$

It can be shown that mutual information can be equivalently written as

$$I(X;Y) = H(Y) - H(Y \mid X). \qquad (2.16)$$

It is easy to see that $I(X;Y)$ tends to be large if X and Y are correlated, whereas $I(X;Y)$ would be small if X and Y are not so related; indeed, in the extreme case when X and Y are completely independent, there would be no reduction of entropy, and thus $H(X) = H(X \mid Y)$, and $I(X;Y) = 0$. However, if X is completely determined by Y, then $H(X \mid Y) = 0$, thus $I(X;Y) = H(X)$. Intuitively, mutual information can measure the correlation of two random variables. Clearly as a correlation measure on X and Y, mutual information is symmetric.

Applications of these basic concepts, including entropy, conditional entropy, and mutual information will be further discussed later in this book.

2.3 Machine Learning

Machine learning is a very important technique for solving many problems, and has very broad applications. In text data management and analysis, it has also many uses. Any in-depth treatment of this topic would clearly be beyond the scope of this book, but here we introduce some basic concepts in machine learning that are needed to better understand the content later in the book.

Machine learning techniques can often be classified into two types: **supervised learning** and **unsupervised learning**. In supervised learning, a computer would learn how to compute a function $\hat{y} = f(x)$ based on a set of examples of the input value x (called training data) and the corresponding expected output value y. It is called "supervised" because typically the y values must be provided by humans for each x, and thus the computer receives a form of supervision from the humans. Once the function is learned, the computer would be able to take unseen values of x and compute the function $f(x)$.

When y takes a value from a finite set of values, which can be called labels, a function $f(\cdot)$ can serve as a classifier in that it can be used to map an instance x to the "right" label (or multiple correct labels when appropriate). Thus, the problem can be called a classification problem. The simplest case of the classification problem is when we have just two labels (known as binary classification). When y

takes a real value, the problem is often called a regression problem. Both forms of the problem can also be called prediction when our goal is mainly to infer the unknown y for a given x; the term "prediction" is especially meaningful when y is some property of a future event.

In text-based applications, both forms may occur, although the classification problem is far more common, in which case the problem is also called *text categorization* or *text classification*. We dedicate a chapter to this topic later in the book (Chapter 15). The regression problem may occur when we use text data to predict another non-text variable such as sentiment rating or stock prices; both cases are also discussed later.

In classification as well as regression, the (input) data instance x is often represented as a feature vector where each feature provides a potential clue about which y value is most likely the value of $f(x)$. What the computer learns from the training data is an optimal way to combine these features with weights on them to indicate their importance and their influence on the final function value y. "Optimal" here simply means that the prediction error on the training data is minimum, i.e., the predicted $\hat{y}$ values are maximally consistent with the true y values in the training data.

More formally, let our collection of objects be **X** such that $x_i \in \mathbf{X}$ is a feature vector that represents object i. A feature is an attribute of an object that describes it in some way. For example, if the objects are news articles, one feature could be whether the word *good* occurred in the article. All these different features are part of a document's feature vector, which is used to represent the document. In our cases, the feature vector will usually have to do with the words that appear in the document.

We also have **Y**, which is the set of possible labels for each object. Thus, y_i may be *sports* in our news article classification setup and y_j could be *politics*.

A classifier is a function $f(\cdot)$ that takes a feature vector as input and outputs a predicted label $\hat{y} \in \mathbf{Y}$. Thus, we could have $f(x_i) = sports$, meaning $\hat{y} = sports$. If the true y is also *sports*, the classifier was correct in its prediction.

Notice how we can only evaluate a classification algorithm if we know the true labels of the data. In fact, we will have to use the true labels in order to learn a good function $f(\cdot)$ to take unseen feature vectors and classify them. For this reason, when studying machine learning algorithms, we often split our corpus **X** into two parts: **training data** and **testing data**. The training portion is used to build the classifier, and the testing portion is used to evaluate the performance (e.g., determine how many correct labels were predicted). In applications, the training data are generally

all the labelled examples that we can generate, and the test cases are the data points, to which we would like to apply our machine learning program.

But what does the function $f(\cdot)$ actually do? Consider a very simple example that determines whether a news article has positive or negative sentiment, i.e., $\mathbf{Y} = \{positive, negative\}$:

$$f(x) = \begin{cases} positive & \text{if } x\text{'s count for the term } good \text{ is greater than 1} \\ negative & \text{otherwise.} \end{cases}$$

Of course, this example is overly simplified, but it does demonstrate the basic idea of a classifier: it takes a feature vector as input and outputs a class label. Based on the training data, the classifier may have determined that positive sentiment articles contain the term *good* more than once; therefore, this knowledge is encoded in the function. In Chapter 15, we will investigate some specific algorithms for creating the function $f(\cdot)$ based on the training data. Other topics such as feedback for information retrieval (Chapter 7) and sentiment analysis (Chapter 18) make use of classifiers, or resemble them. For this reason, it's good to know what machine learning is and what kinds of problems it can solve.

In contrast to supervised learning, in unsupervised learning we only have the data instances $\mathbf{X}$ without knowing $\mathbf{Y}$. In such a case, obviously we cannot really know how to compute y based on an x. However, we may still learn latent properties or structures of $\mathbf{X}$. Since there is no human effort involved, such an approach is called unsupervised. For example, the computer can learn that some data instances are very similar, and the whole dataset can be represented by three major clusters of data instances such that in each cluster, the data instances are all very similar. This is essentially the clustering technique that we will discuss in Chapter 14. Another form of unsupervised learning is to design probabilistic models to model the data (called "generative models") where we can embed interesting parameters that denote knowledge that we would like to discover from the data. By fitting the model to our data, we can estimate the parameter values that can best explain the data, and treat the obtained parameter values as the knowledge discovered from the data. Applications of such an approach in analyzing latent topics in text are discussed in detail in Chapter 17.

Bibliographic Notes and Further Reading

Detailed discussion of the basic concepts in probability and statistics can be found in many textbooks such as Hodges and Lehmann [1970]. An excellent introduction to the maximum likelihood estimation can be found in Myung [2003]. An accessi-

ble comprehensive introduction to Bayesian statistics is given in the book *Bayesian Data Analysis* [Gelman et al. 1995]. Cover and Thomas [1991] provide a comprehensive introduction to information theory. There are many books on machine learning where a more rigorous introduction to the basic concepts in machine learning as well as many specific machine learning approaches can be found (e.g., Bishop 2006, Mitchell 1997).

Exercises

2.1. What can you say about $p(X \mid Y)$ if we know X and Y are independent random variables? Prove it.

2.2. In an Information Retrieval course, there are 78 computer science majors, 21 electrical and computer engineering majors, and 10 library and information science majors. Two students are randomly selected from the course. What is the probability that they are from the same department? What is the probability that they are from different departments?

2.3. Use Bayes' rule to solve the following problem. One third of the time, Milo takes the bus to work and the other times he takes the train. The bus is less reliable, so he gets to work on time only 50% of the time. If taking the train, he is on time 90% of the time. Given that he was on time on a particular day, what is the probability that Milo took the bus?

2.4. In a game based on a deck of 52 cards, a single card is drawn. Depending on the type of card, a certain value is either won or lost. If the card is one of the four aces, $10 is won. If the card is one of the four kings, $5 is won. If the card is one of the eleven diamonds that is not a king or ace, $2 is won. Otherwise, $1 is lost. What are the expected winnings or losings after drawing a single card? (Would you play?)

2.5. Consider the game outlined in the previous question. Imagine that two aces were drawn, leaving 50 cards remaining. What is the expected value of the next draw?

2.6. In the information theory section, we defined three random variables X, Y, and Z when discussing entropy. We compared $H(Y)$ with $H(Z)$. How does $H(X)$ compare to the other two entropies?

2.7. In the information theory section, we compared the entropy of the word *the* to that of the word *unicorn*. In general, what types of words have a high entropy and what types of words have a low entropy? As an example, consider a corpus of ten

documents where *the* occurs in all documents, *unicorn* appears in five documents, and *Mercury* appears in one document. What would be the entropy value of each?

2.8. Brainstorm some different features that may be good for the sentiment classification task outlined in this chapter. What are the strengths and weaknesses of such features?

2.9. Consider the following scenario. You are writing facial recognition software that determines whether there is a face in a given image. You have a collection of $100,000$ images with the correct answer and need to determine if there are faces in new, unseen images.

Answer the following questions.

 (a) Is this supervised learning or unsupervised learning?
 (b) What are the labels or values we are predicting?
 (c) Is this binary classification or multiclass classification? (Or neither?)
 (d) Is this a regression problem?
 (e) What are the features that could be used?

2.10. Consider the following scenario. You are writing essay grading software that takes in a student essay and produces a score from 0–100%. To design this system, you are given essays from the past year which have been graded by humans. Your task is to use the system with the current year's essays as input.

Answer the same questions as in Exercise 2.9.

2.11. Consider the following scenario. You are writing a tool that determines whether a given web page is one of

- personal home page,
- links to a personal home page, or
- neither of the above.

To help you in your task, you are given $5,000,000$ pages that are already labeled. Answer the same questions as in Exercise 2.9.

Text Data Understanding

In this chapter, we introduce basic concepts in text data understanding through natural language processing (NLP). NLP is concerned with developing computational techniques to enable a computer to understand the meaning of natural language text. NLP is a foundation of text information systems because how effective a TIS is in helping users access and analyze text data is largely determined by how well the system can understand the content of text data. Content analysis is thus logically the first step in text data analysis and management.

While a human can instantly understand a sentence in their native language, it is quite challenging for a computer to make sense of one. In general, this may involve the following tasks.

Lexical analysis.   The purpose of lexical analysis is to figure out what the basic meaningful units in a language are (e.g., words in English) and determine the meaning of each word. In English, it is rather easy to determine the boundaries of words since they are separated by spaces, but it is non-trivial to find word boundaries in some other languages such as Chinese where there is no clear delimiter to separate words.

Syntactic analysis.   The purpose of syntactic analysis is to determine how words are related with each other in a sentence, thus revealing the syntactic structure of a sentence.

Semantic analysis.   The purpose of semantic analysis is to determine the meaning of a sentence. This typically involves the computation of meaning of a whole sentence or a larger unit based on the meanings of words and their syntactic structure.

Pragmatic analysis.   The purpose of pragmatic analysis is to determine meaning in context, e.g., to infer the speech acts of language. Natural language is used by humans to communicate with each other. A deeper understanding

of natural language than semantic analysis is thus to further understand the purpose in communication.

Discourse analysis.   Discourse analysis is needed when a large chunk of text with multiple sentences is to be analyzed; in such a case, the connections between these sentences must be considered and the analysis of an individual sentence must be placed in the appropriate context involving other sentences.

In Figure 3.1, we show what is involved in understanding a very simple English sentence *"A dog is chasing a boy on the playground."* The lexical analysis in this case involves determining the syntactic categories (parts of speech) of all the words (for example, *dog* is a noun and *chasing* is a verb). Syntactic analysis is to determine that *a* and *boy* form a noun phrase. So do *the* and *playground*, and *on the playground* is a prepositional phrase. Semantic analysis is to map noun phrases to entities and verb phrases to relations so as to obtain a formal representation of the meaning of the sentence. For example, the noun phrase *a boy* can be mapped to a semantic entity denoting a boy (i.e., b1), and *a dog* to an entity denoting a dog (i.e., d1). The verb phrase can be mapped to a relation predicate chasing(d1,b1,p1) as shown in the figure. Note that with this level of understanding, one may also infer additional information based on any relevant common sense knowledge. For example, if we assume that if someone is being chased, he or she may be scared, we could infer that the boy being chased (b1) may be scared. Finally, pragmatic analysis might further reveal that the person who said this sentence might intend to request an action, such as reminding the owner of the dog to take the dog back.

While it is possible to derive a clear semantic representation for a simple sentence like the one shown in Figure 3.1, it is in general very challenging to do this kind of analysis for unrestricted natural language text. The main reason for this difficulty is because natural language is designed to make human communication efficient; this is in contrast with a programming language which is designed to facilitate computer understanding. Specifically, there are two reasons why NLP is very difficult. (1) We omit a lot of "common sense" knowledge in natural language communication because we assume the hearer or reader possesses such knowledge (thus there's no need to explicitly communicate it). (2) We keep a lot of ambiguities, which we assume the hearer/reader knows how to resolve (thus there's no need to waste words to clarify them). As a result, natural language text is full of ambiguity, and resolving ambiguity would generally involve reasoning with a large amount of common-sense knowledge, which is a general difficult challenge in artificial intel-
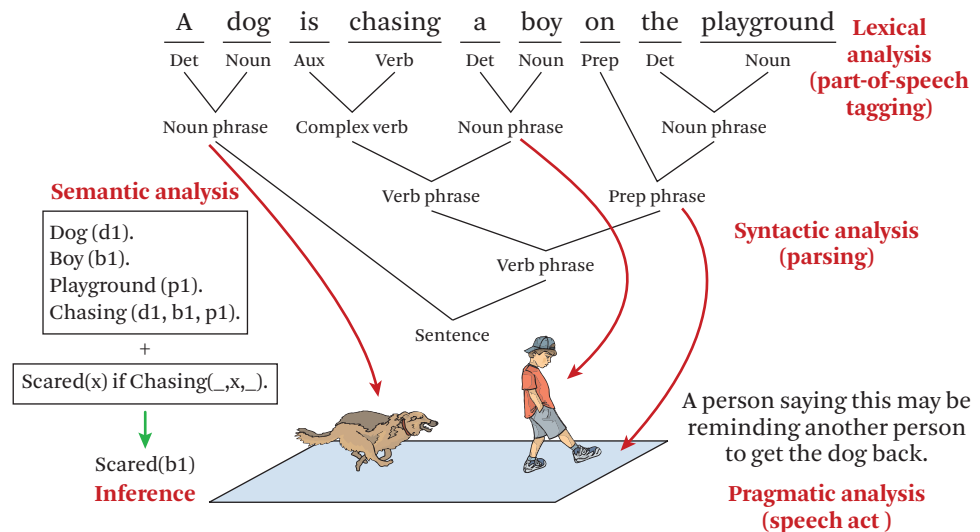
Figure 3.1 **An example of tasks in natural language understanding.**

ligence. In this sense, NLP is "AI complete", i.e., as difficult as any other difficult problems in artificial intelligence.

The following are a few examples of specific challenges in natural language understanding.

Word-level ambiguity. A word may have multiple syntactic categories and multiple senses. For example, *design* can be a noun or a verb (**ambiguous POS**); *root* has multiple meanings even as a noun (**ambiguous sense**).

Syntactic ambiguity. A phrase or a sentence may have multiple syntactic structures. For example, *natural language processing* can have two different interpretations: "processing of natural language" vs. "natural processing of language" (**ambiguous modification**). Another example: *A man saw a boy with a telescope* has two distinct syntactic structures, leading to a different result regarding who had the telescope (**ambiguous prepositional phrase (PP) attachment**).

Anaphora resolution. What exactly a pronoun refers to may be unclear. For example, in *John persuaded Bill to buy a TV for himself*, does *himself* refer to John or Bill?

Presupposition.   *He has quit smoking* implies that he smoked before; making such inferences in a general way is difficult.

3.1 History and State of the Art in NLP

Research in NLP dated back to at least the 1950s when researchers were very optimistic about having computers that understood human language, particularly for the purpose of machine translation. Soon however, it was clear, as stated in Bar-Hillel's report in 1960, that fully-automatic high-quality translation could not be accomplished without knowledge. That is, a dictionary is insufficient; instead, we would need an encyclopedia.

Realizing that machine translation may be too ambitious, researchers tackled less ambitious applications of NLP in the late 1960s and 1970s with some success, though the techniques developed failed to scale up, thus only having limited application impact. For example, people looked at speech recognition applications where the goal is to transcribe a speech. Such a task requires only limited understanding of natural language, thus more realistic; for example, figuring out the exact syntactic structure is probably not very crucial for speech recognition. Two interesting projects that demonstrated clear ability of computer understanding of natural language are worth mentioning. One is the Eliza project where shallow rules are used to enable a computer to play the role of a therapist to engage a natural language dialogue with a human. The other is the block world project which demonstrated feasibility of deep semantic understanding of natural language when the language is limited to a toy domain with only blocks as objects.

In the 1970s–1980s, attention was paid to process real-world natural-language text data, particularly story understanding. Many formalisms for knowledge representation and heuristic inference rules were developed. However, the general conclusion was that even simple stories are quite challenging to understand by a computer, confirming the need for large-scale knowledge representation and inferences under uncertainty.

After the 1980s, researchers started moving away from the traditional symbolic (logic-based) approaches to natural language processing, which mostly had proven to be not robust for real applications, and paying more attention to statistical approaches, which enjoyed more success, initially in speech recognition, but later also in virtually all other NLP tasks. In contrast to symbolic approaches, statistical approaches tend to be more robust because they have less reliance on human-generated rules; instead, they often take advantage of regularities and patterns in

empirical uses of language, and rely solely on labeled training data by humans and application of machine learning techniques.

While linguistic knowledge is always useful, today, the most advanced natural language processing techniques tend to rely on heavy use of statistical machine learning techniques with linguistic knowledge only playing a somewhat secondary role. These statistical NLP techniques are successful for some of the NLP tasks. Part of speech tagging is a relatively easy task, and state-of-the-art POS taggers may have a very high accuracy (above 97% on news data). Parsing is more difficult, though partial parsing can probably be done with reasonably high accuracy (e.g., above 90% for recognizing noun phrases)[1].

However, full structure parsing remains very difficult, mainly because of ambiguities. Semantic analysis is even more difficult, only successful for some aspects of analysis, notably information extraction (recognizing named entities such as names of people and organization, and relations between entities such as who works in which organization), word sense disambiguation (distinguishing different senses of a word in different contexts of usage), and sentiment analysis (recognizing positive opinions about a product in a product review). Inferences and speech act analysis are generally only feasible in very limited domains.

In summary, only "shallow" analysis of natural language processing can be done for arbitrary text and in a robust manner; "deep" analysis tends not to scale up well or be robust enough for analyzing unrestricted text. In many cases, a significant amount of training data (created by human labeling) must be available in order to achieve reasonable accuracy.

3.2 NLP and Text Information Systems

Because of the required robustness and efficiency in TIS applications, in general, robust shallow NLP techniques tend to be more useful than fragile deep analysis techniques, which may hurt application performance due to inevitable analysis errors caused by the general difficulty of NLP. The limited value of deep NLP for some TIS tasks is further due to various ways to bypass the more difficult task of precisely understanding the meaning of natural language text and directly optimize the task performance. Thus, while improved NLP techniques should in general enable improved TIS task performance, lack of NLP capability isn't necessarily a major barrier for some application tasks, notably text retrieval, which is a relatively

---

1. These performance numbers were based on a specific data set, so they may not generalize well even within the same domain.

Tasks   **Dependency on NLP**   **"Easier" and
more "workarounds"**

Classification/
retrieval

Summarization/
extraction/
topic mining

Translation/
dialogue

Question
answering
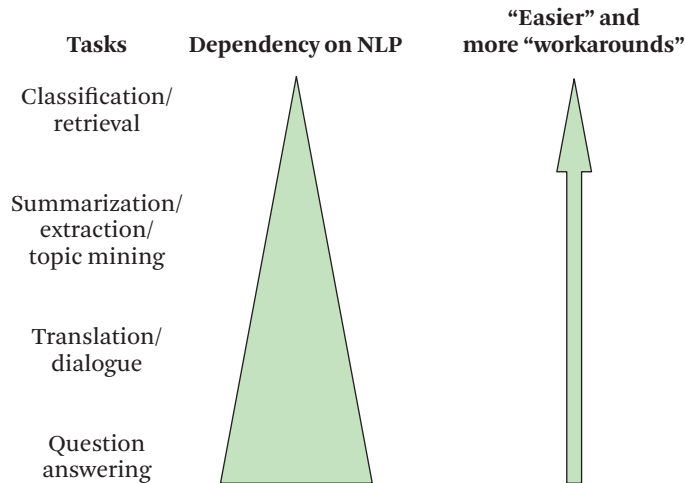
Figure 3.2   **"Easy" vs. "difficult" NLP applications.**

easy task as compared with a more difficult task such as machine translation where deep understanding of natural language is clearly required.

Figure 3.2 shows a number of TIS tasks that require somewhat different levels of NLP. At one end of the spectrum, tasks such as retrieval and classification are relatively easy, and in most of the cases, they don't require deep NLP; indeed, looking at the keywords mentioned in text is often sufficient to determine whether a document is relevant to a query or about a certain topic. At the other end, however, tasks such as machine translation and question answering would require much more precise understanding; for example, a wrong parse of a sentence generally would lead to a wrong translation unless the target language has a similar ambiguity structure, and similarly, a wrong understanding of the question would lead to wrong answers.

When it comes to a specific application task, it is often possible to bypass the difficulty in accurately understanding natural language and go directly to solve the application problem. A well-known example is the Eliza system,[2] which is supposed to play the role of a therapist and make a dialogue with a human user [Weizenbaum 1966]. The following is a sample dialogue.

   Person: Men are all alike.

_____

2. http://www.cs.nott.ac.uk/~pszgxk/courses/g5aiai/002history/eliza.htm

ELIZA :   In what way?

Person: They're always bugging us about something or other.

ELIZA: Can you think of a specific example?

Person: Well, my boyfriend made me come here.

ELIZA: Your boyfriend made you come here?

Person: He says I'm depressed much of the time.

ELIZA: I am sorry to hear that you are depressed.

Person: It's true. I am unhappy.

ELIZA: Do you think coming here will help you not to be unhappy?

Person: I need some help, that much seems certain.

ELIZA: What would it mean to you if you got some help?

Person: Perhaps I could learn to get along with my mother.

ELIZA: Tell me more about your family

On the surface, the dialogue appears to be quite natural, and indeed, such a dialogue might be useful to engage a depressed patient in a conversation. However, the system does not really understand the language, and solely relies on heuristic rules like the following to keep the dialogue going:

$$I \text{ remember } X \rightarrow \text{Do you often think of } X?$$

$$\text{always } \rightarrow \text{Can you think of a specific example?}$$

Such rules enable the system to directly perform the task, i.e., making a conversation, without necessarily trying to understand the real meaning of words and determining the meaning of the entire sentence.

Such a pattern-based way of solving a problem has turned out to be quite powerful. Indeed, modern machine learning approaches to natural language understanding are essentially based on this and in many ways are similar to the Eliza system, but with two important differences. The first is that the rules in a machine learning system would not be exact or strict; instead, they tend to be stochastic, and the probabilities of choosing which rule would be empirically set based on a training data set where the expected behavior of a function to be computed is known. Second, instead of having human to supply rules, the "soft" rules may be learned

automatically from the training data with only minimum help from users who can, e.g., specify the elements in a rule.

Even difficult tasks like machine translation can be done by such statistical approaches. The most useful NLP techniques for building a TIS are statistical approaches which tend to be much more robust than symbolic approaches. Statistical language models are especially useful because they can quantify the uncertainties associated with the use of natural language in a principled way.

3.3 Text Representation

Techniques from NLP allow us to design many different types of informative features for text objects. Let's take a look at the example sentence *A dog is chasing a boy on the playground* in Figure 3.3. We can represent this sentence in many different ways. First, we can always represent such a sentence as a string of characters. This is true for every language. This is perhaps the most general way of representing text since we can always use this approach to represent any text data. Unfortunately, the downside to this representation is that it can't allow us to perform semantic analysis, which is often needed for many applications of text mining. We're not even recognizing words, which are the basic units of meaning for any language. (Of course, there are some situations where characters are useful, but that is not the general case.)

The next version of text representation is performing word segmentation to obtain a sequence of words. In the example sentence, we get features like *dog* and *chasing*. With this level of representation, we suddenly have much more freedom. By identifying words, we can (for example), easily discover the most frequent words in this document or the whole collection. These words can then be used to form topics. Therefore, representing text data as a sequence of words opens up a lot of interesting analysis possibilities.

However, this level of representation is slightly less general than a string of characters. In some languages, such as Chinese, it's actually not that easy to identify all the word boundaries since in such a language text is a sequence of characters with no spaces in between words. To solve this problem, we have to rely on some special techniques to identify words and perform more advanced segmentation that isn't only based on whitespace (which isn't always 100% accurate). So, the sequence of words representation is not as robust as the string of characters representation. In English, it's very easy to obtain this level of representation so we can use this all the time.

If we go further in natural language processing, we can add part-of-speech (POS) tags to the words. This allows us to count, for example, the most frequent nouns; or,

we could determine what kind of nouns are associated with what kind of verbs. This opens up more interesting opportunities for further analysis. Note in Figure 3.3 that we use a plus sign on the additional features because by representing text as a sequence of part of speech tags, we don't necessarily replace the original word sequence. Instead, we add this as an additional way of representing text data.

Representing text as both words and POS tags enriches the representation of text data, enabling a deeper, more principled analysis. If we go further, then we'll be parsing the sentence to obtain a syntactic structure. Again, this further opens up more interesting analysis of, for example, the writing styles or grammatical error correction.

If we go further still into semantic analysis, then we might be able to recognize *dog* as an animal. We also can recognize *boy* as a person, and *playground* as a location and analyze their relations. One deduction could be that the dog was chasing the boy, and the boy is on the playground. This will add more entities and relations, through entity-relation recognition. Now, we can count the most frequent person that appears in this whole collection of news articles. Or, whenever you see a mention of this person you also tend to see mentions of another person or object. These types of repeated pattens can potentially make very good features.
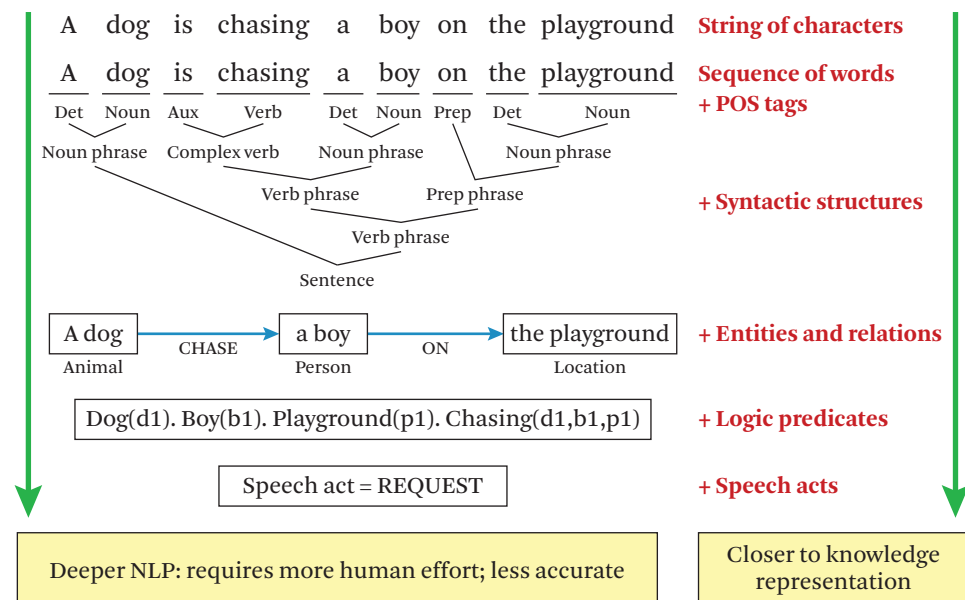
Figure 3.3   **Illustration of different levels of text representation.**

Such a high-level representation is even less robust than the sequence of words or POS tags. It's not always easy to identify all the entities with the right types and we might make mistakes. Relations are even harder to find; again, we might make mistakes. The level of representation is less robust, yet it's very useful. If we move further to a logic representation, then we have predicates and inference rules. With inference rules we can infer interesting derived facts from the text. As one would imagine, we can't do that all the time for all kinds of sentences since it may take significant computation time or a large amount of training data.

Finally, speech acts would add yet another level of representation of the intent of this sentence. In this example, it might be a request. Knowing that would allow us to analyze even more interesting things about the observer or the author of this sentence. What's the intention of saying that? What scenarios or what kinds of actions will occur?

Figure 3.3 shows that if we move downwards, we generally see more sophisticated NLP techniques. Unfortunately, such techniques would require more human effort as well, and they are generally less robust since they attempt to solve a much more difficult problem. If we analyze our text at levels that represent deeper analysis of language, then we have to tolerate potential errors. That also means it's still necessary to combine such deep analysis with shallow analysis based on (for example) sequences of words. On the right side, there is an arrow that points down to indicate that as we go down, our representation of text is closer to the knowledge representation in our mind. That's the purpose of text mining!

Clearly, there is a tradeoff here between doing deeper analysis that might have errors but would give us direct knowledge that can be extracted from text and doing shadow analysis that is more robust but wouldn't give us the necessary deeper representation of knowledge.

Text data are generated by humans and are meant to be consumed by humans. As a result, in text data analysis and text mining, humans play a very important role. They are always in the loop, meaning that we should optimize for a collaboration between humans and computers. In that sense, it's okay that computers may not be able to have a completely accurate representation of text data. Patterns that are extracted from text data can be interpreted by humans, and then humans can guide the computers to do more accurate analysis by annotating more data, guiding machine learning programs to make them work more effectively.

Different text representation tends to enable different analyses, as shown in Figure 3.4. In particular, we can gradually add more and more deeper analysis results to represent text data that would open up more interesting representation opportunities and analysis capabilities. The table summarizes what we have just

Text Rep	Generality	Enabled Analysis	Examples of Application
String		String processing	Compression
Words		Word relation analysis; topic analysis; sentiment analysis	Thesaurus discovery; topic- and opinion-related applications
+ Syntactic structures		Syntactic graph analysis	Stylistic analysis; structure-based feature extraction
+ Entities & relations		Knowledge graph analysis; information network analysis	Discovery of knowledge and opinions about specific entities
+ Logic predicates		Integrative analysis of scattered knowledge; logic inference	Knowledge assistant for biologists

Figure 3.4 **Text representation and enabled analysis.**

seen; the first column shows the type of text representation while the second visualizes the generality of such a representation. By generality, we mean whether we can do this kind of representation accurately for all the text data (very general) or only some of them (not very general). The third column shows the enabled analysis techniques and the final column shows some examples of applications that can be achieved with a particular level of representation.

As a sequence of characters, text can only be processed by string processing algorithms. They are very robust and general. In a compression application, we don't need to know word boundaries (although knowing word boundaries might actually help). Sequences of words (as opposed to characters) offer a very important level of representation; it's quite general and relatively robust, indicating that it supports many analysis techniques such as word relation analysis, topic analysis, and sentiment analysis. As you may expect, many applications can be enabled by these kinds of analysis. For example, thesaurus discovery has to do with discovering related words, and topic- and opinion-related applications can also be based on word-level representation. People might be interested in knowing major topics covered in the collection of text, where a topic is represented as a distribution over words.

Moving down, we'll see we can gradually add additional representations. By adding syntactic structures, we can enable syntactic graph analysis; we can use graph mining algorithms to analyze these syntactic graphs. For example, stylistic

analysis generally requires syntactical structure representation. We can also generate structure-based features that might help us classify the text objects into different categories by looking at their different syntactic structures. If you want to classify articles into different categories corresponding to different authors, then you generally need to look at syntactic structures. When we add entities and relations, then we can enable other techniques such as knowledge graphs or information networks. Using these more advanced feature representations allows applications that deal with entities.

Finally, when we add logical predicates, we can integrate analysis of scattered knowledge. For example, we can add an ontology on top of extracted information from text to make inferences. A good example of an application enabled by this level of representation is a knowledge assistant for biologists. This system is able to manage all the relevant knowledge from literature about a research problem such as understanding gene functions. The computer can make inferences about some of the hypotheses that a biologist might be interested in. For example, it could determine whether a gene has a certain function by reading literature to extract relevant facts. It could use a logic system to track answers to researchers' questions about what genes are related to what functions. In order to support this level of application, we need to go as far as logical representation.

This book covers techniques mainly focused on word-based representation. These techniques are general and robust and widely used in various applications. In fact, in virtually all text mining applications, you need this level of representation. Still, other levels can be combined in order to support more linguistically sophisticated applications as needed.

3.4 Statistical Language Models

A statistical language model (or just language model for short) is a probability distribution over word sequences. It thus gives any sequence of words a potentially different probability. For example, a language model may give the following three-word sequences different probabilities:

$$p(\text{Today is Wednesday}) = 0.001$$

$$p(\text{Today Wednesday is}) = 0.000000001$$

$$p(\text{The equation has a solution}) = 0.000001$$

Clearly, a language model can be context-dependent. In the language model shown above, the sequence *The equation has a solution* has a smaller probability than *Today is Wednesday.* This may be a reasonable language model for describ-

ing general conversations, but it may be inaccurate for describing conversations happening at a mathematics conference, where the sequence *The equation has a solution* may occur more frequently than *Today is Wednesday.*

Given a language model, we can sample word sequences according to the distribution to obtain a text sample. In this sense, we may use such a model to "generate" text. Thus, a language model is also often called a generative model for text.

Why is a language model useful? A general answer is that it provides a principled way to quantify the uncertainties associated with the use of natural language. More specifically, it allows us to answer many interesting questions related to text analysis and information retrieval. The following are some examples of questions that a language model can help answer.

- Given that we see *John* and *feels*, how likely will we see *happy* as opposed to *habit* as the next word? Answering this question can help speech recognition as *happy* and *habit* have very similar acoustic signals, but a language model can easily suggest that *John feels happy* is far more likely than *John feels habit.*

- Given that we observe *baseball* three times and *game* once in a news article, how likely is it about the topic "sports"? This will obviously directly help text categorization and information retrieval tasks.

- Given that a user is interested in sports news, how likely would it be for the user to use *baseball* in a query? This is directly related to information retrieval.

If we enumerate all the possible sequences of words and give a probability to each sequence, the model would be too complex to estimate because the number of parameters is potentially infinite since we have a potentially infinite number of word sequences. That is, we would never have enough data to estimate these parameters. Thus, we have to make assumptions to simplify the model.

The simplest language model is the unigram language model in which we assume that a word sequence results from generating each word independently. Thus, the probability of a sequence of words would be equal to the product of the probability of each word. Formally, let V be the set of words in the vocabulary, and $w_1, \ldots, w_n$ a word sequence, where $w_i \in V$ is a word. We have

$$p(w_1, \ldots, w_n) = \prod_{i=1}^{n} p(w_i). \tag{3.1}$$

Given a unigram language model θ, we have as many parameters as the words in the vocabulary, and they satisfy the constraint $\sum_{w \in V} p(w) = 1$. Such a model essentially specifies a multinomial distribution over all the words.

$p(w|\theta_1)$ $p(w|\theta_2)$

```
. . .              . . .
text 0.2           food 0.25
mining 0.1         nutrition 0.1
association 0.01   healthy 0.05
clustering 0.02    diet 0.02
. . .              . . .
food 0.00001       text 0.00001
. . .              . . .
```
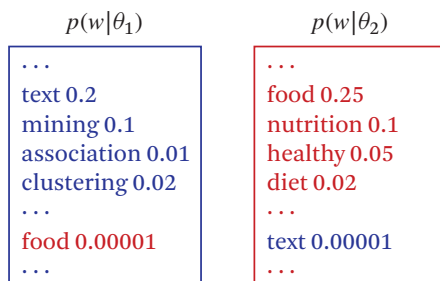
Figure 3.5 **Two examples of unigram language models, representing two different topics.**

Given a language model θ, in general, the probabilities of generating two different documents D_1 and D_2 would be different, i.e., $p(D_1 \,|\, \theta) \neq p(D_2 \,|\, \theta)$. What kind of documents would have higher probabilities? Intuitively it would be those documents that contain many occurrences of the high probability words according to $p(w \,|\, \theta)$. In this sense, the high probability words of θ can indicate the topic captured by θ.

For example, the two unigram language models illustrated in Figure 3.5 suggest a topic about "text mining" and a topic about "health", respectively. Intuitively, if D is a text mining paper, we would expect $p(D \,|\, \theta_1) > p(D \,|\, \theta_2)$, while if D' is a blog article discussing diet control, we would expect the opposite: $p(D' \,|\, \theta_1) < p(D' \,|\, \theta_2)$. We can also expect $p(D \,|\, \theta_1) > p(D' \,|\, \theta_1)$ and $p(D \,|\, \theta_2) < p(D' \,|\, \theta_2)$.

Now suppose we have observed a document D (e.g., a short abstract of a text mining paper) which is assumed to be generated using a unigram language model θ, and we would like to infer the underlying model θ (i.e., estimate the probabilities of each word w, $p(w \,|\, \theta)$) based on the observed D. This is a standard problem in statistics called parameter estimation and can be solved using many different methods.

One popular method is the maximum likelihood (ML) estimator, which seeks a model $\hat{\theta}$ that would give the observed data the highest likelihood (i.e., best explain the data):

$$\hat{\theta} = \arg\max_\theta p(D \,|\, \theta). \tag{3.2}$$

It is easy to show that the ML estimate of a unigram language model gives each word a probability equal to its relative frequency in D. That is,

$$p(w \,|\, \hat{\theta}) = \frac{c(w, D)}{|D|}, \tag{3.3}$$

where $c(w, D)$ is the count of word w in D and $|D|$ is the length of D, or total number of words in D.

Such an estimate is optimal in the sense that it would maximize the probability of the observed data, but whether it is really optimal for an application is still questionable. For example, if our goal is to estimate the language model in the mind of an author of a research article, and we use the maximum likelihood estimator to estimate the model based only on the abstract of a paper, then it is clearly non-optimal since the estimated model would assign zero probability to any unseen words in the abstract, which would make the whole article have a zero probability unless it only uses words in the abstract. Note that, in general, the maximum likelihood estimate would assign zero probability to any unseen token or event in the observed data; this is so because assigning a non-zero probability to such a token or event would take away probability mass that could have been assigned to an observed word (since all probabilities must sum to 1), thus reducing the likelihood of the observed data. We will discuss various techniques for improving the maximum likelihood estimator later by using techniques called **smoothing**.

Although extremely simple, a unigram language model is already very useful for text analysis. For example, Figure 3.6 shows three different unigram language models estimated on three different text data samples, i.e., a general English text database, a computer science research article database, and a text mining research paper. In general, the words with the highest probabilities in all the three models are those functional words in English because such words are frequently used in any text. After going further down on the list of words, one would see more content-carrying and topical words. Such content words would differ dramatically depending on the data to be used for the estimation, and thus can be used to discriminate the topics in different text samples.

Unigram language models can also be used to perform semantic analysis of word relations. For example, we can use them to find what words are semantically associated with a word like *computer*. The main idea for doing this is to see what other words tend to co-occur with the word *computer*. Specifically, we can first obtain a sample of documents (or sentences) where *computer* is mentioned. We can then estimate a language model based on this sample to obtain $p(w \mid \text{computer})$. This model tells us which words occur frequently in the context of "computer." However, the most frequent words according to this model would likely be functional words in English or words that are simply common in the data, but have no strong association with *computer*. To filter out such common words, we need a model for such words which can then tell us what words should be filtered. It is easy to see that the general English language model (i.e., a background language model) would serve

Background LM: $p(w|B)$ **Collection LM:** $p(w|C)$ **Document LM:** $p(w|D)$
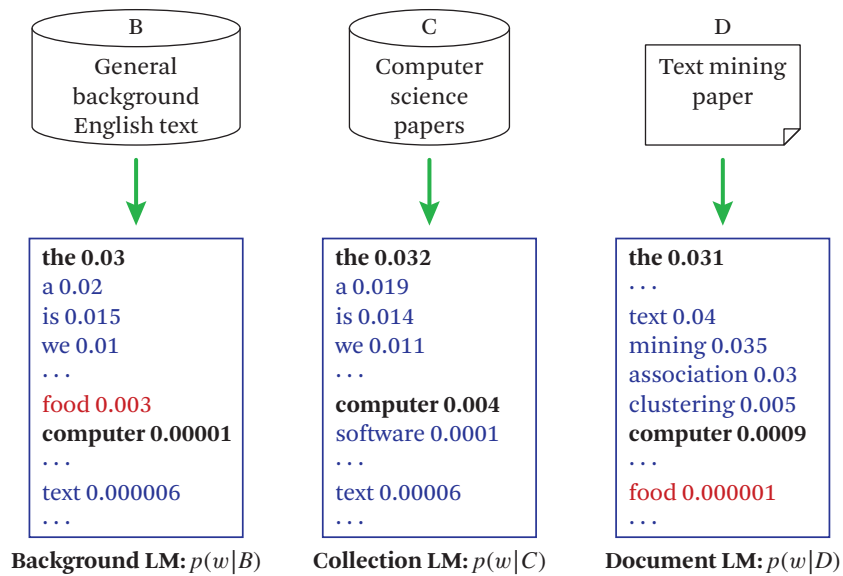
Figure 3.6 Three different language models representing three different topics.

the purpose well. So we can use the background language model to normalize the model $p(w \mid \text{computer})$ and obtain a probability ratio for each word. Words with high ratio values can then be assumed to be semantically associated with *computer* since they tend to occur frequently in its context, but not frequently in general. This is illustrated in Figure 3.7.

More applications of language models in text information systems will be further discussed as their specific applications appear in later chapters. For example, we can represent both documents and queries as being generated from some language model. Given this background however, the reader should have sufficient information to understand the future chapters dealing with this powerful statistical tool.

Bibliographic Notes and Further Reading

There are many textbooks on NLP, including, *Speech and Language Processing* [Jurafsky and Martin 2009], *Foundations of Statistical NLP* [Manning and Schütze 1999], and *Natural Language Understanding* [Allen 1995]. An in-depth coverage of statistical language models can be found in the book *Statistical Methods for Speech Recognition* [Jelinek 1997]. Rosenfeld [2000] provides a concise yet comprehensive

Topic LM: $p(w|\text{“computer”})$ **Normalized topic LM:**
                                                $p(w|\text{“computer”})/p(w|B)$

the 0.032
a 0.019
is 0.014
we 0.008
computer 0.004
software 0.0001
. . .
text 0.00006

All documents
containing word
"computer"

computer 400
software 150
program 104
. . .
text 3.0
. . .
the 1.1
a 0.99
is 0.9
we 0.8

Background LM: $p(w|B)$

B

General
background
English text

the 0.03
a 0.02
is 0.015
we 0.01
. . .
computer 0.00001
. . .
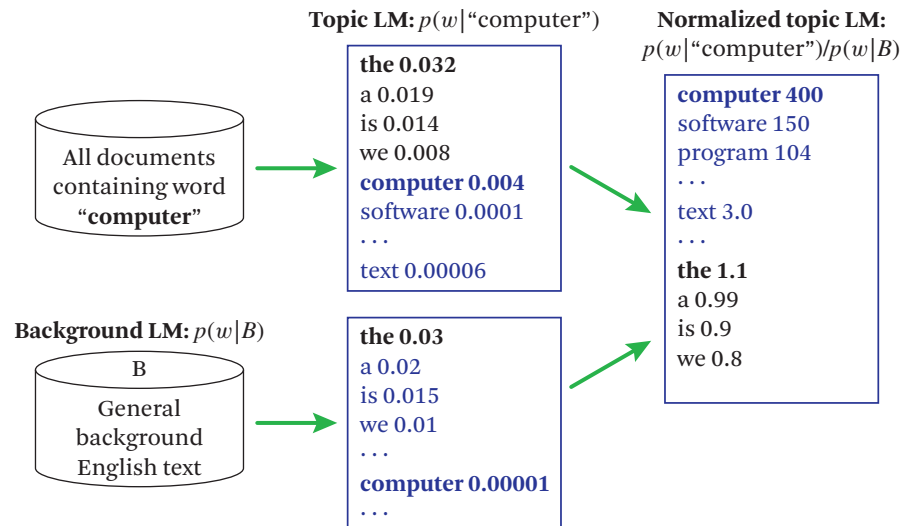
Figure 3.7   **Using topic language models and a background language model to find semantically related words.**

review of statistical language models. Zhai [2008] contains a detailed discussion of the use of statistical language models for information retrieval, some of which will be covered in later chapters of this book. An important topic in NLP that we have not covered much in this chapter is information extraction. A comprehensive introduction to this topic can be found in Sarawagi [2008], and a useful survey can be found in Jiang [2012]. For a discussion of this topic in the context of information retrieval, see the book Moens [2006].

Exercises

3.1. In what way is NLP related to text mining?

3.2. Does poor NLP performance mean poor retrieval performance? Explain.

3.3. Given a collection of documents for a specific topic, how can we use maximum likelihood estimation to create a topic unigram language model?

3.4. How might the size of a document collection affect the quality of a language model?

3.5. Why might maximum likelihood estimation not be the best guess of parameters for a topic language model?

3.6. Suppose we appended a duplicate copy of the topic collection to itself and re-estimated a maximum likelihood language model. Would θ change?

3.7. A unigram language model as defined in this chapter can take a sequence of words as input and output its probability. Explain how this calculation has strong independence assumptions.

3.8. Given a unigram language model θ estimated from this book, and two documents $d_1 = \texttt{information retrieval}$ and $d_2 = \texttt{retrieval information}$, then is $p(d_1 \mid \theta) > p(d_2 \mid \theta)$? Explain.

3.9. An n-gram language model records sequences of n words. How does the number of possible parameters change if we decided to use a 2-gram (bigram) language model instead of a unigram language model? How about a 3-gram (trigram) model? Give your answer in terms of V, the unigram vocabulary size.

3.10. Using your favorite programming language, estimate a unigram language model using maximum likelihood. Do this by reading a single text file and delimiting words by whitespace.

3.11. Sort the words by their probabilities from the previous exercise. If you used a different text file, how would your sorted list be different? How would it be the same?

4 META: A Unified Toolkit for Text Data Management and Analysis

This chapter introduces the accompanying software META, a free and open-source toolkit that can be used to analyze text data. Throughout this book, we give hands-on exercises with META to practice concepts and explore different text mining algorithms.

Most of the algorithms and methods discussed in this book can be found in some form in the META toolkit. If META doesn't include a technique discussed in this book, it's likely that a chapter exercise is to implement this feature yourself! Despite being a powerful toolkit, META's simplicity makes feature additions relatively straightforward, usually through extending a short class hierarchy.

Configuration files are an integral part of META's forward-facing infrastructure. They are designed such that exploratory analysis usually requires no programming effort from the user. By default, META is packaged with various executables that can be used to solve a particular task. For example, for a classification experiment the user would run the following command in their terminal[1]:

```
./classify config.toml
```

This is standard procedure for using the default executables; they take only one configuration file parameter. The configuration file format is explained in detail later in this chapter, but essentially it allows the user to select a dataset, a way

---

1. Running the default classification experiment requires a dataset to operate on. The 20news-groups dataset is specified in the default META config file and can be downloaded here: https://meta-toolkit.org/data/20newsgroups.tar.gz. Place it in the meta/data/ directory.

to tokenize the dataset, and a particular classification algorithm to run (for this example).

If more advanced functionality is desired, programming in C++ is required to make calls to META's API (applications programming interface). Both configuration file and API usage are documented on META's website, https://meta-toolkit.org as well as in this chapter. Additionally, a forum for META exists (https://forum.meta-toolkit.org), containing discussion surrounding the toolkit. It includes user support topics, community-written documentation, and developer discussions.

The sections that follow delve into a little more detail about particular aspects of META so the reader will be comfortable working with it in future chapters.

4.1 Design Philosophy

META's goal is to improve upon and complement the current body of open source machine learning and information retrieval software. The existing environment of this open source software tends to be quite fragmented.

There is rarely a single location for a wide variety of algorithms; a good example of this is the LIBLINEAR [Fan et al. 2008] software package for SVMs. While this is the most cited of the open source implementations of linear SVMs, it focuses solely on kernel-less methods. If presented with a nonlinear classification problem, one would be forced to find a different software package that supports kernels (such as the same authors' LIBSVM package [Chang and Lin 2011]). This places an undue burden on the researchers and students—not only are they required to have a detailed understanding of the research problem at hand, but they are now forced to understand this fragmented nature of the open-source software community, find the appropriate tools in this mishmash of implementations, and compile and configure the appropriate tool.

Even when this is all done, there is the problem of data formatting—it is unlikely that the tools have standardized upon a single input format, so a certain amount of data preprocessing is now required. This all detracts from the actual task at hand, which has a marked impact on the speed of discovery and education.

META addresses these issues. In particular, it provides a unifying framework for text indexing and analysis methods, allowing users to quickly run controlled experiments. It modularizes the feature generation, instance representation, data storage formats, and algorithm implementations; this allows for researchers and students to make seamless transitions along any of these dimensions with minimal effort.

META's modularity supports exploration, encourages contributions, and increases visibility to its inner workings. These facts make it a perfect companion toolkit for this book. As mentioned at the beginning of the chapter, readers will follow exercises that add real functionality to the toolkit. After reading this book and learning about text data management and analysis, it is envisioned readers continue to modify META to suit their text information needs, building upon their newfound knowledge.

Finally, since META will always be free and open-source, readers as a community can jointly contribute to its functionality, benefiting all those involved.

4.2 Setting up META

All future sections in this book will assume the reader has META downloaded and installed. Here, we'll show how to set up META.

META has both a website with tutorials and an online repository on GitHub. To actually download the toolkit, users will check it out with the version control software `git` in their command line terminal after installing any necessary prerequisites.

The META website contains instructions for downloading and setting up the software for a particular system configuration. At the time of writing this book, both Linux and Mac OS are supported. Visit https://meta-toolkit.org/setup-guide.html and follow the instructions for the desired platform. We will assume the reader has performed the steps listed in the setup guide and has a working version of META for all exercises and demonstrations in this book.

There are two steps that are not mentioned in the setup guide. The first is to make sure the reader has the version of META that was current when this book was published. To ensure that the commands and examples sync up with the software the reader has downloaded, we will ensure that META is checked out with version 2.2.0. Run the following command inside the `meta/` directory:

```
git reset --hard v2.2.0
```

The second step is to make sure that any necessary model files are also downloaded. These are available on the META releases page: https://github.com/meta-toolkit/meta/releases/tag/v2.2.0. By default, the model files should be placed in the `meta/build/` directory, but you can place them anywhere as long as the paths in the config file are updated.

Once these steps are complete, the reader should be able to complete any exercise or run any demo. If any additional files or information are needed, it will be provided in the accompanying section.

4.3   Architecture

All processed data in META is stored in an index. There are two index types: `forward_index` and `inverted_index`. The former is keyed by document IDs, and the latter is keyed by term IDs.

> `forward_index` is used for applications such as topic modeling and most classification tasks.

> `inverted_index` is used to create search engines, or do classification with k-nearest-neighbor or similar algorithms.

Since each META application takes an index as input, all processed data is interchangeable between all the components. This also gives a great advantage to classification: META supports out-of-core classification by default! If a dataset is small enough (like most other toolkits assume), a cache can be used such as `no_evict_cache` to keep it all in memory without sacrificing any speed. (Index usage is explained in much more detail in the search engine exercises.)

There are four corpus input formats.

> `line_corpus`. each dataset consists of one to three files:
>
>> `corpusname.dat`. each document appears on one line
>
>> `corpusname.dat.labels`. optional file that includes the class or label of the document on each line, again corresponding to the order in `corpusname.dat`. These are the labels that are used for the classification tasks.

> `file_corpus`. each document is its own file, and the name of the file becomes the name of the document. There is also a `corpusname-full-corpus.txt` which contains (on each line) a required class label for each document followed by the path to the file on disk. If there are no class labels, a placeholder label should be required, e.g., "[none]".

> `gz_corpus`. similar to `line_corpus`, but each of its files and metadata files are compressed using gzip compression:
>
>> `corpusname.dat.gz`. compressed version of `corpusname.dat`
>
>> `corpusname.dat.labels.gz`. compressed version of `corpusname.dat.labels`

`libsvm_corpus`. If only being used for classification, MeTA can also take LIBSVM-formatted input to create a `forward_index`. There are many machine learning datasets available in this format on the LIBSVM site.[2]

For more information on corpus storage and configuration settings, we suggest the reader consult https://meta-toolkit.org/overview-tutorial.html.

4.4 Tokenization with MeTA

The first step in creating an index over any sort of text data is the "tokenization" process. At a high level, this simply means converting individual text documents into sparse vectors of counts of terms—these sparse vectors are then typically consumed by an indexer to output an `inverted_index` over your corpus.

MeTA structures this text analysis process into several layers in order to give the user as much power and control over the way the text is analyzed as possible.

An analyzer, in most cases, will take a "filter chain" that is used to generate the final tokens for its tokenization process: the filter chains are always defined as a specific tokenizer class followed by a sequence of zero or more filter classes, each of which reads from the previous class's output. For example, here is a simple filter chain that lowercases all tokens and only keeps tokens with a certain length range:

`icu_tokenizer` → `lowercase_filter` → `length_filter`

Tokenizers always come first. They define how to split a document's string content into tokens. Some examples are as follows.

`icu_tokenizer`. converts documents into streams of tokens by following the Unicode standards for sentence and word segmentation.

`character_tokenizer`. converts documents into streams of single characters.

Filters come next, and can be chained together. They define ways that text can be modified or transformed. Here are some examples of filters.

`length_filter`. this filter accepts tokens that are within a certain length and rejects those that are not.

`icu_filter`. applies an ICU (International Components for Unicode)[3] transliteration to each token in the sequence. For example, an accented character like $\ddot{\imath}$ is instead written as i.

---

2. http://www.csie.ntu.edu.tw/˜cjlin/libsvmtools/datasets

3. http://site.icu-project.org/; note that different versions of ICU will tokenize text in slightly different ways!

list_filter. this filter either accepts or rejects tokens based on a list. For example, one could use a stop word list and reject stop words.

porter2_stemmer. this filter transforms each token according to the Porter2 English Stemmer rules.[4]

Analyzers operate on the output from the filter chain and produce token counts from documents. Here are some examples of analyzers.

ngram_word_analyzer. Collects and counts sequences of n words (tokens) that have been filtered by the filter chain.

ngram_pos_analyzer. Same as ngram_word_analyzer, but operates on part-of-speech tags from META's CRF implementation.

tree_analyzer. Collects and counts occurrences of parse tree features.

libsvm_analyzer. Converts a LIBSVM line_corpus into META format.

META defines a sane default filter chain that users are encouraged to use for general text analysis in the absence of any specific requirements. To use it, one should specify the following in the configuration file:

```
[[analyzers]]
method = "ngram-word"
ngram = 1
filter = "default-chain"
```

This configures the text analysis process to consider unigrams of words generated by running each document through the default filter chain. This filter chain should work well for most languages, as all of its operations (including but not limited to tokenization and sentence boundary detection) are defined in terms of the Unicode standard wherever possible.

To consider both unigrams and bigrams, the configuration file should look like the following:

```
[[analyzers]]
method = "ngram-word"
ngram = 1
filter = "default-chain"

[[analyzers]]
```

---

4. http://snowball.tartarus.org/algorithms/english/stemmer.html

```
method = "ngram-word"
ngram = 2
filter = "default-chain"
```

Each [[analyzers]] block defines a single analyzer and its corresponding filter chain: as many can be used as desired—the tokens generated by each analyzer specified will be counted and placed in a single sparse vector of counts. This is useful for combining multiple different kinds of features together into your document representation. For example, the following configuration would combine unigram words, bigram part-of-speech tags, tree skeleton features, and subtree features.

```
[[analyzers]]
method = "ngram-word"
ngram = 1
filter = "default-chain"

[[analyzers]]
method = "ngram-pos"
ngram = 2
filter = [{type = "icu-tokenizer"}, {type = "ptb-normalizer"}]
crf-prefix = "path/to/crf/model"

[[analyzers]]
method = "tree"
filter = [{type = "icu-tokenizer"}, {type = "ptb-normalizer"}]
features = ["skel", "subtree"]
tagger = "path/to/greedy-tagger/model"
parser = "path/to/sr-parser/model"
```

If an application requires specific text analysis operations, one can specify directly what the filter chain should look like by modifying the configuration file. Instead of filter being a string parameter as above, we will change filter to look very much like the [[analyzers]] blocks: each analyzer will have a series of [[analyzers.filter]] blocks, each of which defines a step in the filter chain. All filter chains must start with a tokenizer. Here is an example filter chain for unigram words like the one at the beginning of this section:

```
[[analyzers]]
method = "ngram-word"
ngram = 1
    [[analyzers.filter]]
    type = "icu-tokenizer"
```

```
[[analyzers.filter]]
type = "lowercase"

[[analyzers.filter]]
type = "length"
min = 2
max = 35
```

Meta provides many different classes to support building filter chains. Please look at the API documentation[5] for more information. In particular, the `analyzers::tokenizers` namespace and the `analyzers::filters` namespace should give a good idea of the capabilities. The static public attribute `id` for a given class is the string needed for the "type" in the configuration file.

4.5 Related Toolkits

Existing toolkits supporting text management and analysis tend to fall into two categories. The first is search engine toolkits, which are especially suitable for building a search engine application, but tend to have limited support for text analysis/mining functions. Examples include the following.

Lucene.   https://lucene.apache.org/

Terrier.   http://terrier.org/

Indri/Lemur.   http://www.lemurproject.org/

The second is text mining or general data mining and machine learning toolkits, which tend to selectively support some text analysis functions, but generally do not support search capability. Examples include the following.

Weka.   http://www.cs.waikato.ac.nz/ml/weka/

LIBSVM.   https://www.csie.ntu.edu.tw/~cjlin/libsvm/

Stanford NLP.   http://nlp.stanford.edu/software/corenlp.shtml

Illinois NLP Curator.   http://cogcomp.cs.illinois.edu/page/software_view/Curator

Scikit Learn.   http://scikit-learn.org/stable/

NLTK.   http://www.nltk.org/

---

5. Visit https://meta-toolkit.org/doxygen/namespaces.html

However, there is a lack of seamless integration of search engine capabilities with various text analysis functions, which is necessary for building a unified system for supporting text management and analysis. A main design philosophy of META, which also differentiates META from the existing toolkits, is its emphasis on the tight integration of search capabilities (indeed, text access capabilities in general) with text analysis functions, enabling it to provide full support for building a powerful text analysis application. To facilitate education and research, META is designed with an emphasis on modularity and extensibility achieved through object-oriented design. META can be used together with existing toolkits in multiple ways. For example, for very large-scale text applications, an existing search engine toolkit can be used to support search, while META can be used to further support analysis of the found search results or any subset of text data that are obtained from the original large data set. NLP toolkits can be used to preprocess text data and generate annotated text data for modules in META to use as input. META can also be used to generate a text representation that would be fed into a different data mining or machine learning toolkit.

Exercises

In its simplest form, text data could be a single document in `.txt` format. This exercise will get you familiar with various techniques that are used to analyze text. We'll use the novel *A Tale of Two Cities* by Charles Dickens as example text. The book is called `two-cities.txt`, and is located at http://sifaka.cs.uiuc.edu/ir/textdatabook/two-cities.txt. You can also use any of your own plaintext files that have multiple English sentences.

Like all future exercises, we will assume that the reader followed the META setup guide and successfully compiled the executables. In this exercise, we'll only be using the `profile` program. Running `./profile` from inside the `build/` directory will print out the following usage information:

```
Usage: ./profile config.toml file.txt [OPTION]
where [OPTION] is one or more of:
        --stem  perform stemming on each word
        --stop  remove stop words
        --pos   annotate words with POS tags
        --pos-replace  replace words with their POS tags
        --parse create grammatical parse trees from file content
        --freq-unigram  sort and count unigram words
        --freq-bigram   sort and count bigram words
        --freq-trigram  sort and count trigram words
        --all   run all options
```

If running `./profile` prints out this information, then everything has been set up correctly. We'll look into what each of these options mean in the following exercises.

4.1. Stop Word Removal. Consider the following words: *I, the, of, my, it, to, from*. If it was known that a document contained these words, would there be any idea what the document was about? Probably not. These types of words are called stop words. Specifically, they are very high frequency words that do not contain content information. They are used because they're grammatically required, such as when connecting sentences.

Since these words do not contain any topical information, they are often removed as a preprocessing step in text analysis. Not only are these (usually) useless words ignored, but having less data can mean that algorithms run faster!

```
./profile config.toml two-cities.txt --stop
```

Now, use the `profile` program to remove stop words from the document `two-cities.txt`. Can you still get an idea of what the book is about without these words present?

4.2. Stemming. Stemming is the process of reducing a word to a base form. This is especially useful for search engines. If a user wants to find books about *running*, documents containing the word *run* or *runs* would not match. If we apply a stemming algorithm to a word, it is more likely that other forms of the word will match it in an information retrieval task.

The most popular stemming algorithm is the Porter2 English Stemmer, developed by Martin Porter. It is a slightly improved version from the original Porter Stemmer from 1980. Some examples are:

$$\{run, \ runs, \ running\} \rightarrow run$$

$$\{argue, \ argued, \ argues, \ arguing\} \rightarrow argu$$

$$\{lies, \ lying, \ lie\} \rightarrow lie$$

META uses the Porter2 stemmer by default. You can read more about the Porter2 stemmer here: http://snowball.tartarus.org/algorithms/english/stemmer.html. An online demo of the stemmer is also available if you'd like to play around with it: http://web.engr.illinois.edu/~massung1/p2s_demo.html.

Now that you have an idea of what stemming is, run the stemmer on *A Tale of Two Cities*.

```
./profile config.toml two-cities.txt --stem
```

Like stop word removal, stemming tries to keep the basic meaning behind the original text. Can you still make sense of it after it's stemmed?

4.3. Part-of-Speech Tagging.   When learning English, students often encounter different grammatical labels for words, such as *noun, adjective, verb, etc*. In linguistics and computer science, there is a much larger dichotomy of these labels called part of speech (POS) tags. Each word can be assigned a tag based on surrounding words. Consider the following sentence: *All hotel rooms are pretty much the same, although the room number might change*. Here's a part-of-speech tagged version:

All$_{DT}$ hotel$_{NN}$ rooms$_{NNS}$ are$_{VBP}$ pretty$_{RB}$ much$_{RB}$ the$_{DT}$ same$_{JJ}$,,
although$_{IN}$ the$_{DT}$ room$_{NN}$ number$_{NN}$ might$_{MD}$ change$_{VB}$.

Above, VBP and VB are different types of verbs, NN and NNS are singular and plural nouns, and DT means determiner. This is just a subset of about 80 commonly used tags. Not every word has a unique part of speech tag. For instance, *flies* and *like* can have multiple tags depending on the context:

Time$_{NN}$ flies$_{VBZ}$ like$_{IN}$ an$_{DT}$ arrow$_{NN}$.
Fruit$_{NN}$ flies$_{NNS}$ like$_{VBP}$ a$_{DT}$ banana$_{NN}$.

Such situations can make POS-tagging challenging. Nevertheless, human agreement on POS tag labeling is about 97%, which is the ceiling for automatic taggers.

POS tags can be used in text analysis as an alternate (or additional) representation to words. Using these tags captures a slightly more grammatical sense of a document or corpus. The `profile` program has two options for POS tagging. The first annotates each word like the examples above, and the second replaces each word with its POS tag.

```
./profile config.toml two-cities.txt --pos
./profile config.toml two-cities.txt --pos-replace
```

Note that POS tagging the book may take up to one minute to complete. Does it look like META's POS tagger is accurate? Can you find any mistakes? When replacing the words with their tags, is it possible to determine what the original sentence was? Experiment with the book or any other text file.

4.4. Parsing.   Grammatical parse trees represent deeper syntactic knowledge from text sentences. They represent sentence phrase hierarchy as a tree structure. Consider the example in Figure 4.1.

The parse tree is rooted with S, denoting Sentence; the sentence is composed of a noun phrase (NP) followed by a verb phrase (VP) and period. The leaves of the
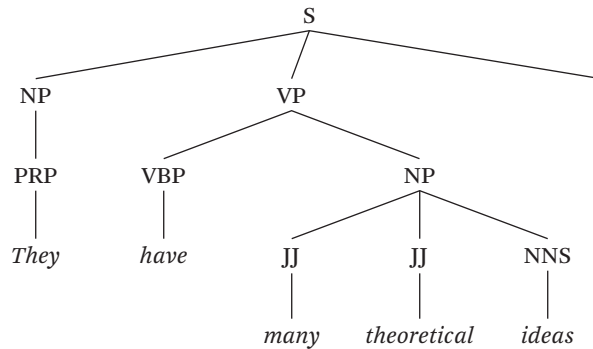
Figure 4.1   **An example of a parse tree.**

tree are the words in the sentence, and the preterminals (the direct parents of the leaves) are part-of-speech tags.

Some common features from a parse tree are production rules such as $S \rightarrow NP\ VP$, tree depth, and structural tree features. Syntactic categories (node labels) alone can also be used.

The following command runs the parser on each sentence in the input file:

```
./profile config.toml two-cities.txt --parse
```

Like POS-tagging, the parsing may also take a minute or two to complete.

4.5. Frequency Analysis.   Perhaps the most common text-processing technique is frequency counting. This simply counts how many times each unique word appears in a document (or corpus). Viewing a descending list of words sorted by frequency can give you an idea of what the document is about. Intuitively, similar documents should have some of the same high-frequency words . . . not including stop words.

Instead of single words, we can also look at strings of n words, called n-grams. Consider this sentence: *I took a vacation to go to a beach.*

- 1-grams (unigrams):

$$\{I : 1,\ \ took : 1,\ \ a : 2,\ \ vacation : 1,\ \ to : 2,\ \ go : 1,\ \ beach : 1\}$$

- 2-grams (bigrams):

$$\{I\ took : 1,\ \ took\ a : 1,\ \ a\ vacation : 1,\ \ vacation\ to : 1,$$
$$to\ go : 1,\ \ go\ to : 1,\ \ to\ a : 1,\ \ a\ beach : 1\}$$

- 3-grams (trigrams):

$$\{I\ took\ a : 1,\ \ took\ a\ vacation : 1,\ \ a\ vacation\ to : 1,\ \ \ldots\}$$

As we will see in this text, the unigram words document representation is of utmost importance for text representation. This vector of counts representation does have a downside though: we lost the order of the words. This representation is also known as "bag-of-words," since we only know the counts of each word, and no longer know the context or position. This unigram counting scheme can be used with POS tags or any other type of token derived from a document.

Use the following three commands to do an *n*-gram frequency analysis on a document, for $n \in [1, 3]$.

```
./profile config.toml two-cities.txt --freq-unigram
./profile config.toml two-cities.txt --freq-bigram
./profile config.toml two-cities.txt --freq-trigram
```

This will give the output file `two-cities.freq.1.txt` for the option `--freq-unigram` and so on.

What makes the output reasonably clear? Think back to stop words and stemming. Removing stop words gets rid of the noisy high-frequency words that don't give any information about the content of the document. Stemming will aggregate inflected words into a single count. This means the partial vector $\{run : 4, running : 2, runs : 3\}$ would instead be represented as $\{run : 9\}$. Not only does this make it easier for humans to interpret the frequency analysis, but it can improve text mining algorithms, too!

4.6. Zipf's Law.   In English, the top four most frequent words are about 10–15% of all word occurrences. The top 50 words are 35–40% of word occurrences. In fact, there is a similar trend in any human language. Think back to the stop words. These are the most frequent words, and make up a majority of text. At the same time, many words may only appear once in a given document.

We can plot the rank of a word on the x axis, and the frequency count on the y axis. Such a graph can give us an idea of the word distribution in a given document or collection. In Figure 4.2, we counted unigram words from another Dickens book, *Oliver Twist*. The plot on the left is a normal $x \sim y$ plot and the one on the right is a $\log x \sim \log y$ plot.
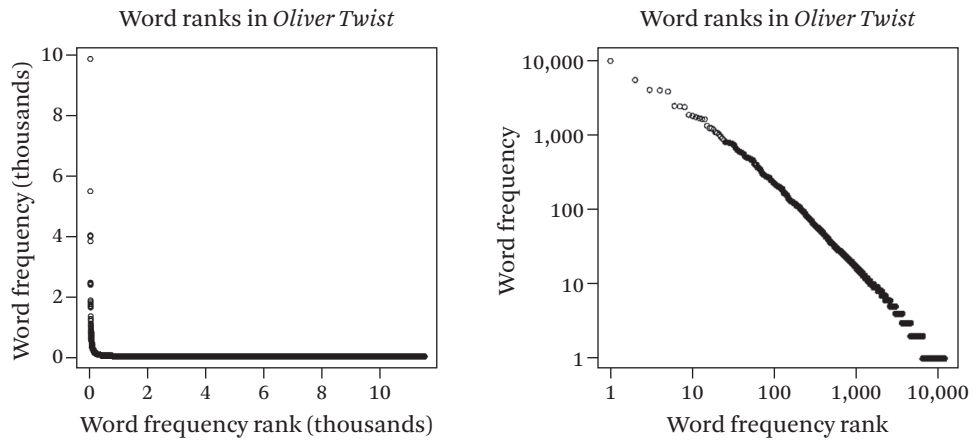
Figure 4.2   **Illustration of Zipf's law.**

Zipf's law describes the shape of these plots. What do you think Zipf's law states? The shape of these plots allows us to apply certain techniques to take advantage of the word distribution in natural language.

PART II

TEXT DATA ACCESS

5

Overview of Text Data Access

Text data access is the foundation for text analysis. Text access technology plays two important roles in text management and analysis applications. First, it enables retrieval of the most relevant text data to a particular analysis problem, thus avoiding unnecessary overhead from processing a large amount of non-relevant data. Second, it enables interpretation of any analysis results or discovered knowledge in appropriate context and provides data provenance (origin).

The general goal of text data access is to connect users with the right information at the right time. This connection can be done in two ways: **pull**, where the users take the initiative to fetch relevant information out from the system, and **push**, where the system takes the initiative to offer relevant information to users. In this chapter, we will give a high-level overview of these two modes of text data access. Then, we will formalize and motivate the problem of text retrieval. In the following chapters, we will cover specific techniques for supporting text access in both push and pull modes.

5.1 Access Mode: Pull vs. Push

Because text data are created for consumption by humans, humans play an important role in text data analysis and management applications. Specifically, humans can help select the most relevant data to a particular application problem, which is beneficial since it enables us to avoid processing the huge amount of raw text data (which would be inefficient) and focus on analyzing the most relevant part. Selecting relevant text data from a large collection is the basic task of text access. This selection is generally based on a specification of the information need of an analyst (a user), and can be done in two modes: pull and push. Figure 5.1 describes how these modes fit together along with querying and browsing.
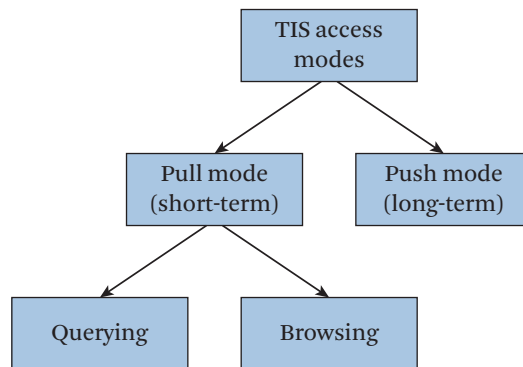
Figure 5.1   **The dichotomy of text information access modes.**

In pull mode, the user initiates the access process to find the relevant text data, typically by using a search engine. This mode of text access is essential when a user has an ad hoc information need, i.e., a temporary information need that might disappear once the need is satisfied. In such a case, the user can use a query to find relevant information with a search engine. For example, a user may have a need to buy a product and thus be interested in retrieving all the relevant opinions about candidate products; after the user has purchased the product, the user would generally no longer need such information. Another example is that during the process of analyzing social media data to understand opinions about an emerging event, the analyst may also decide to explore information about a particular entity related to the event (e.g., a person), which can also trigger a search activity.

While querying is the most common way of accessing text data in the pull mode, browsing is another complementary way of accessing text data in the pull mode, and can be very useful when a user does not know how to formulate an effective query, or finds it inconvenient to enter a keyword query (e.g., through a smartphone), or simply wants to explore a topic with no fixed goal. Indeed, when searching the Web, users tend to mix querying and browsing (e.g., while traversing through hyperlinks).

In general, we may regard querying and browsing as two complementary ways of finding relevant information in the information space. Their relation can be understood by making an analogy between information seeking and sightseeing in a physical world. When a tourist knows the exact address of an attraction, the tourist can simply take a taxi directly to the attraction; this is similar to when a user knows exactly what he or she is looking for and can formulate a query with the

"right keywords," which would bring to the user relevant pages directly. However, if a tourist doesn't know the exact address of an attraction, the tourist may want to take a taxi to an approximate location and then *walk* around to find the attraction. Similarly, if a user does not have a good knowledge about the target pages, he or she can also use an approximate query to reach some related pages and then *browse* into truly relevant information. Thus, when querying does not work well, browsing can be very useful.

In the push mode, the system initiates the process to recommend a set of relevant information items to the user. This mode of information access is generally more useful to satisfy a long-standing information need of a user or analyst. For example, a researcher's research interests can be regarded as relatively stable over time. In comparison, the information stream (i.e., published research articles) is dynamic. In such a scenario, although a user can regularly search for relevant literature information with queries, it is more desirable for a recommender (also called filtering) system to monitor the dynamic information stream and "push" any relevant articles to the user based on the matching of the articles with the user's interests (e.g., in the form of an email). In some long-term analytics applications, it would also be desirable to use the push mode to monitor any relevant text data (such as relevant social media) about a topic related to the application.

Another scenario of push mode is producer-initiated recommendation, which can be more appropriately called selective dissemination of information (SDI). In such a scenario, the producer of information has an interest in disseminating the information among relevant users, and would push an information item to such users. Advertising of product information on search result pages is such an example. The recommendation can be delivered through email notifications or recommended through a search engine result page.

There are broadly two kinds of information needs: short-term need and long-term need. Short-term needs are most often associated with pull mode, and long-term needs are most often associated with push mode. A short-term information need is temporary and usually satisfied through search or navigation in the information space, whereas a long-term information need can be better satisfied through filtering or recommendation where the system would take the initiative to push the relevant information to a user. Ad hoc retrieval is extremely important because ad hoc information needs show up far more frequently than long-term information needs. The techniques effective for ad hoc retrieval can usually be re-used for filtering and recommendation as well. Also, in the case of long-term information needs, it is possible to collect user feedback, which can be exploited. In this sense,

ad hoc retrieval is much harder, as we do not have much feedback information from a user (i.e., little training data for a particular query). Due to the availability of training data, the problem of filtering or recommendation can usually be solved by using supervised machine learning techniques, which are covered well in many existing books. Thus, we will cover ad hoc retrieval in much more detail than filtering and recommendation.

5.2 Multimode Interactive Access

Ideally, the system should provide support for users to have multimode interactive access to relevant text data so that the push and pull modes are integrated in the same information access environment, and querying and browsing are also seamlessly integrated to provide maximum flexibility to users and allow them to query and browse at will.

In Figure 5.2, we show a snapshot of a prototype system (http://timan.cs.uiuc .edu/proj/sosurf/) where a topic map automatically constructed based on a set of queries collected in a commercial search engine has been added to a regular search

Figure 5.2 Sample interface of browsing with a topic map where browsing and querying are naturally integrated.

engine interface to enable a user to browse the information space flexibly. With this interface, a user can do any of the following at any moment.

Querying (long-range jump).   When a user submits a new query through the search box the search results from a search engine will be shown in the right pane. At the same time, the relevant part of a topic map is also shown on the left pane to facilitate browsing should the user want to.

Navigating on the map (short-range walk).   The left pane in our interface is to let a user navigate on the map. When a user clicks on a map node, this pane will be refreshed and a local view with the clicked node as the current focus will be displayed. In the local view, we show the parents, the children, and the horizontal neighbors of the current node in focus (labelled as "center" in the interface). A user can thus zoom into a child node, zoom out to a parent node, or navigate into a horizontal neighbor node. The number attached to a node is a score for the node that we use for ranking the nodes. Such a map enables the user to "walk" in the information space to browse into relevant documents without needing to reformulate queries.

Viewing a topic region.   The user may double-click on a topic node on the map to view the documents covered in the topic region. The search result pane would be updated with new results corresponding to the documents in the selected topic region. From a user's perspective, the result pane always shows the documents in the current region that the user is focused on (either search results of the query or the documents corresponding to a current node on the map when browsing).

Viewing a document.   Within the result pane, a user can select any document to view as in a standard search interface.

In Figure 5.3, we further show an example trace of browsing in which the user started with a query *dining table*, zoomed into *asian dining table*, zoomed out back to *dining table*, browsed horizontally first to *dining chair* and then to *dining furniture*, and finally zoomed out to the general topic *furniture* where the user would have many options to explore different kinds of furniture. If this user feels that a "long-jump" is needed, he or she can use a new query to achieve it. Since the map can be hidden and only brought to display when the user needs it, such an interface is a very natural extension of the current search interface from a user's perspective. Thus, we can see how one text access system can combine multiple modes of information access to suit a user's current needs.
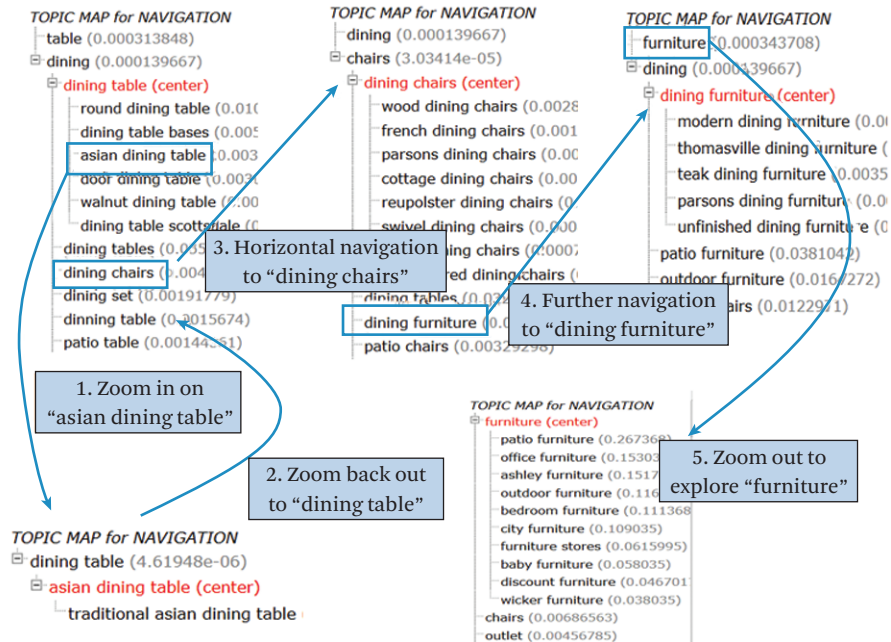
Figure 5.3   **A sample trace of browsing showing how a user can navigate in the information space without querying.**

5.3   Text Retrieval

The most important tool for supporting text data access is a search engine, which is why web search engines are used by many people on a daily basis. Search engines directly provide support for querying and can be easily extended to provide recommendation or browsing. Moreover, the techniques used to implement an effective search engine are often also useful for implementation of a recommender system as well as many text analysis functions. We thus devote a large portion of this book to discussing search engine techniques.

In this section, we discuss the problem of text retrieval (TR), which is solved by developing a search engine system. We specify the differences between unstructured TR and structured database retrieval. We then make an argument for document *ranking* as opposed to document *selection*. This provides a basis for us to discuss in the next chapter how to rank documents for a query.

From a user's perspective, the problem of TR is to use a query to find relevant documents in a collection of text documents. This is a frequently needed task because users often have temporary ad hoc information needs for various tasks,

and would like to find the relevant information immediately. The system to support TR is a text retrieval system, or a search engine.

Although TR is sometimes used interchangeably with the more general term "information retrieval" (IR), the latter also includes retrieval of other types of information such as images or videos. It is worth noting, though, that retrieval techniques for other non-textual data are less mature and, as a result, retrieval of other types of information tends to rely on using text retrieval techniques to match a keyword query with companion text data with a non-textual data element. For example, the current image search engines on the Web are essentially a TR system where each image is represented by a text document consisting of any associated text data with the image (e.g., title, caption, or simply textual context of the image such as the news article content where an image is included).

In industry, the problem of TR is generally referred to as the search problem, and the techniques for text retrieval are often called search technology or search engine technology.

The task of TR can be easy or hard, depending on specific queries and specific collections. For example, during a web search, finding homepages is generally easy, but finding out people's opinions about some topic (e.g., U.S. foreign policy) would be much harder. There are several reasons why TR is difficult:

- a query is usually quite short and incomplete (no formal language like SQL);

- the information need may be difficult to describe precisely, especially when the user isn't familiar with the topic, and

- precise understanding of the document content is difficult. In general, since what counts as the correct answer is subjective, even when human experts judge the relevance of documents, they may disagree with each other.

Due to the lack of clear semantic structures and difficulty in natural language understanding, it is often challenging to accurately retrieve relevant information to a user's query. Indeed, even though the current web search engines may appear to be sufficient sometimes, it may still be difficult for a user to quickly locate and harvest all the relevant information for a task. In general, the current search engines work very well for navigational queries and simple, popular informational queries, but in the case where a user has a complex information need such as analyzing opinions about products to buy, or researching medical information about some symptoms, they often work poorly. Moreover, the current search engines generally provide little or no support to help users digest and exploit the retrieved information. As a result, even if a search engine can retrieve the most relevant information,

a user would still have to sift through a long list of documents and read them in detail to fully digest the knowledge buried in text data in order to perform their task at hand. The techniques discussed later in this book can be exploited to help users digest the found information quickly or directly analyze a large amount of text data to reveal useful and actionable knowledge that can be used to optimize decision making or help a user finish a task.

5.4 Text Retrieval vs. Database Retrieval

It is useful to make a comparison of the problem of TR and the similar problem of database retrieval. Both retrieval tasks are to help users find relevant information, but due to the difference in the data managed by these two tasks, there are many important differences.

First, the data managed by a search engine and a database system are different. In databases, the data are *structured* where each field has a clearly defined meaning according to a schema. Thus, the data can be viewed as a table with well-specified columns. For example, in a bank database system, one field may be customer names, another may be the address, and yet another may be the balance of each type of account. In contrast, the data managed by a search engine are *unstructured* text which can be difficult for computers to understand.[1] Thus, even if a sentence says a person lives in a particular address, it remains difficult for the computer to answer a query about the address of a person in response to a keyword query since there is no simple defined structure to free text. Therefore structured data are often easier to manage and analyze since they conform to a clearly defined schema where the meaning of each field is well defined.

Second, a consequence of the difference in the data is that the queries that can be supported by the two are also different. A database query clearly specifies the constraints on the fields of the data table, and thus the expected retrieval results (answers to the query) are very well specified with no ambiguity. In a search engine, however, the queries are generally keyword queries, which are only a vague specification of what documents should be returned. Even if the computer can fully understand the semantics of natural language text, it is still often the case that the user's information need is vague due to the lack of complete knowledge about the information to be found (which is often the reason why the user wants to find the information in the first place!). For example, in the case of searching for relevant

---

1. Although common parlance refers to text as unstructured with a meaningful contrast with relational database structuring, it employs a narrow sense of "structure." For example, from a linguistics perspective, grammar provides well-defined structure. To study this matter further, see the 5S (societies, scenarios, spaces, structures, and streams) works by Fox et al. [2012]

literature to a research problem, the user is unlikely able to clearly and completely specify which documents should be returned.

Finally, the expected results in the two applications are also different. In database search, we can retrieve very specific data elements (e.g., specific columns); in TR, we are generally only able to retrieve a set of relevant documents. With passages or fields identified in a text document, a search engine can also retrieve passages, but it is generally difficult to retrieve specific entities or attribute values as we can in a database. This difference is not as essential as the difference in the vague specification of what exactly is the "correct" answer to a query, but is a direct consequence of the vague information need in TR.

Due to these differences, the challenges in building a useful database and a useful search engine are also somewhat different. In databases, since what items should be returned is clearly specified, there is no challenge in determining which data elements satisfy the user's query and thus should be returned; a major remaining challenge is how to find the answers as quickly as possible especially when there are many queries being issued at the same time. While the efficiency challenge also exists in a search engine, a more important challenge there is to first figure out which documents should be returned for a query before worrying about how to return the answers quickly. In database applications—at least traditional database applications—it is also very important to maintain the integrity of the data; that is, to ensure no inconsistency occurs due to power failure. In TR, modeling a user's information need and search tasks is important, again due to the difficulty for a user to clearly specify information needs and the difficulty in NLP.

Since what counts as the best answer to a query depends on the user, in TR, the user is actually part of our input (together with the query, and document set). Thus, there is no mathematical way to prove that one answer is better than another or prove one method is better than another. Instead, we always have to rely on empirical evaluation using some test collections and users. In contrast, in database research, since the main issue is efficiency, one can prove one algorithm is better than another by analyzing the computational complexity or do some simulation study. Note that, however, when doing simulation study (to determine which algorithm is faster), we also face the same problem as in text retrieval—the simulation may not accurately reflect the real applications. Thus, an algorithm shown to be faster with simulation may not be actually faster for a particular application. Similarly, a retrieval algorithm shown to be more effective with a test collection may turn out to be less effective for a particular application or even another test collection. How to reliably evaluate retrieval algorithms is itself a challenging research topic.

Because of the difference, the two fields have been traditionally studied in different communities with a different application basis. Databases have had widespread applications in virtually every domain with a well-established strong industry. The IR community that studies text retrieval has been an interdisciplinary community involving library and information science and computer science, but had not had a strong industry base until the Web was born in the early 1990s. Since then, the search engine industry has dominated, and as more and more online information is available, the search engine technologies (which include TR and other technical components such as machine learning and natural language processing) will continue to grow. Soon we will find search technologies to have widespread use just like databases. Furthermore, because of the inherent similarity between database search and TR, because both efficiency and effectiveness (accuracy) are important, and because most online data has text fields as well as some kind of structures, the two fields are now moving closer and closer to each other, leading to some common fundamental questions such as: "What should be the right query language?"; "How can we rank items accurately?"; "How do we find answers quickly?"; and "How do we support interactive search?"

Perhaps the most important conclusion from this comparison is that the problem of text retrieval is an *empirically defined problem*. This means that which method works better cannot be answered by pure analytical reasoning or mathematical proofs. Instead, it has to be empirically evaluated by users, making it a significant challenge in evaluating the effectiveness of a search engine. This is also the reason why a significant amount of effort has been spent in research of TR evaluation since it was initially studied in the 1960s. The evaluation methodology of TR remains an important open research topic today; we discuss it in detail in Chapter 9.

5.5 Document Selection vs. Document Ranking

Given a document collection (a set of unordered text documents), the task of text retrieval can be defined as using a user query (i.e., a description of the user's information need) to identify a subset of documents that can satisfy the user's information need. In order to computationally solve the problem of TR, we must first formally define it. Thus, in this section, we will provide a formal definition of TR and discuss high-level strategies for solving this problem.

Let $V = \{w_1, \ldots, w_N\}$ be a vocabulary set of all the words in a particular natural language where w_i is a word. A user's query $q = q_1, q_2, \ldots, q_m$ is a sequence of words, where $q_i \in V$. Similarly, a document $d_i = d_{i1}, \ldots, d_{im}$ is also a sequence of words where $d_{ij} \in V$. In general, a query is much shorter than a document since the query

is often typed in by a user using a search engine system, and users generally do not want to make much effort to type in many words. However, this is not always the case. For example, in a Twitter search, each document is a tweet which is very short, and a user may also cut and paste a text segment from an existing document as a query, which can be very long. Our text collection $C = \{d_1, \ldots, d_M\}$ is a set of text documents. In general, we may assume that there exists a subset of documents in the collection, i.e., $R(q) \subset C$, which are relevant to the user's query q; that is, they are relevant documents or documents useful to the user who typed in the query. Naturally, this relevant set depends on the query q. However, which documents are relevant is generally unknown; the user's query is only a "hint" at which documents should be in the set $R(q)$. Furthermore, different users may use the same query to intend to retrieve somewhat different sets of relevant documents (e.g., in an extreme case, a query word may be ambiguous). This means that it is unrealistic to expect a computer to return exactly the set $R(q)$, unlike the case in database search where this is feasible. Thus, the best a computer can do is to return an approximation of $R(q)$, which we will denote by $R'(q)$.

Now, how can a computer compute $R'(q)$? At a high level, there are two alternative strategies: document selection vs. document ranking. In document *selection*, we will implement a binary classifier to classify a document as either relevant or non-relevant with respect to a particular query. That is, we will design a binary classification function, or an indicator function, $f(q, d) \in \{0, 1\}$. If $f(q, d) = 1$, d would be assumed to be relevant, whereas if $f(q, d) = 0$, it would be non-relevant. Thus, $R'(q) = \{d \mid f(q, d) = 1, d \in C\}$. Using such a strategy, the system must estimate the *absolute relevance*, i.e., whether a document is relevant or not.

An alternative strategy is to rank documents and let the user decide a cutoff. That is, we will implement a ranking function $f(q, d) \in \mathbb{R}$ and rank all the documents in descending values of this ranking function. A user would then browse the ranked list and stop whenever they consider it appropriate. In this case, the set $R'(q)$ is actually defined partly by the system and partly by the user, since the user would implicitly choose a score threshold θ based on the rank position where he or she stopped. In this case, $R'(q) = \{d \mid f(q, d) \geq \theta\}$. Using this strategy, the system only needs to estimate the *relative relevance* of documents: which documents are more likely relevant.

Since estimation of relative relevance is intuitively easier than that of absolute relevance, we can expect it to be easier to implement the ranking strategy. Indeed, ranking is generally preferred to document selection for multiple reasons.

First, due to the difficulty for a user to prescribe the exact criteria for selecting relevant documents, the binary classifier is unlikely accurate. Often the query is

either over-constrained or under-constrained. In the case of an over-constrained query, there may be no relevant documents matching all the query words, so forcing a binary decision may result in no delivery of any search result. If the query is under-constrained (too general), there may be too many documents matching the query, resulting in over-delivery. Unfortunately, it is often very difficult for a user to know the "right" level of specificity in advance before exploring the document collection due to the knowledge gap in the user's mind (which can be the reason why the user wants to find information about the topic). Even if the classifier can be accurate, a user would still benefit from prioritization of the matched relevant documents for examination since a user can only examine one document at a time and some relevant documents may be more useful than others (relevance is a matter of degree). For all these reasons, ranking documents appropriately becomes a main technical challenge in designing an effective text retrieval system.

The strategy of ranking is further shown to be optimal theoretically under two assumptions based on the **probability ranking principle** [Robertson 1997], which states that returning a ranked list of documents in descending order of predicted relevance is the optimal strategy under the following two assumptions.

1. The utility of a document to a user is independent of the utility of any other document.

2. A user will browse the results sequentially.

So the problem is the following. We have a query that has a sequence of words, and a document that's also a sequence of words, and we hope to define the function $f(\cdot, \cdot)$ that can compute a score based on the query and document. The main challenge is designing a good ranking function that can rank all the relevant documents on top of all the non-relevant ones. Now clearly this means our function must be able to measure the likelihood that a document d is relevant to a query q. That also means we have to have some way to define relevance. In particular, in order to implement the program to do that, we have to have a computational definition of relevance, and we achieve this goal by designing a retrieval model which gives us a formalization of relevance. We introduce retrieval models in the Chapter 6.

Bibliographic Notes and Further Reading

A broader discussion of supporting information access via a digital library is available in Gonçalves et al. [2004]. The relation between retrieval ("pull") and filtering ("push") has been discussed in the article Belkin and Croft [1992]. The contrast between information retrieval and database search was discussed in the classic in-

formation retrieval book by van Rijsbergen [1979]. [Hearst 2009] has a systematic discussion of user interfaces of a search system, which is relevant to the design of interfaces for any information system in general; in particular, many visualization techniques that can facilitate browsing and querying are discussed in the book. Exploratory search is a particular type of search tasks that often requires multimodal information access including both querying and browsing. It was covered in a special issue of *Communications of ACM* [White et al. 2006], and White and Roth [2009]. The probability ranking principle [Robertson 1997] is generally regarded as the theoretical foundation for framing the retrieval problem as a ranking problem. More historical work related to this, as well as a set of important research papers in IR up to 1997, can be found in *Readings in Information Retrieval* [Sparck Jones and Willett 1997]. A brief survey of IR history can be found in Sanderson and Croft [2012].

Exercises

5.1. When might browsing be preferable to querying?

5.2. Given search engine user logs, how could you distinguish between browsing behavior and querying behavior?

5.3. Often, push and pull modes are combined in a single system. Give an example of such an application.

5.4. Imagine you have search engine session logs from users that you *know* are browsing for information. How can you use these logs to enhance search results of future users with ad hoc information needs?

5.5. In a Chapter 11, we will discuss recommender systems. These are systems in push mode that deliver information to users. What are some specific applications of recommender systems? Can you name some services available to you that fit into this access mode?

5.6. How could a recommender system (push mode) be coupled with a search engine (pull mode)? Can these two services mutually enhance one another?

5.7. Design a text information system used to explore musical artists. For example, you can search for an artist's name directly. The results are displayed as a graph, with edges to similar artists (as measured by some similarity algorithm). Use TIS access mode vocabulary to describe this system and any enhancements you could make to satisfy different information needs.

5.8. In the same way as the previous question, categorize "Google knowledge graph" (http://www.google.com/insidesearch/features/search/knowledge.html).

5.9. In the same way as the previous two questions, categorize "citation alerts." These are alerts that are based on previous search history in an academic search engine. When new papers are found that are potentially interesting to the user based on their browsing history, an alert is created.

5.10. One assumption of the probability ranking principle is that each document's usefulness to the user is independent of the usefulness of other documents in the index. What is a scenario where this assumption does not hold?

Retrieval Models

In this chapter, we introduce the two main information retrieval models: vector space and query likelihood, which are among the most effective and practically useful retrieval models. We begin with a brief overview of retrieval models in general and then discuss the two basic models, i.e., the vector space model and the query likelihood model afterward.

6.1 Overview

Over many decades, researchers have designed various different kinds of retrieval models which fall into different categories (see Zhai [2008] for a detailed review). First, one family of the models are based on the similarity idea. Basically, we assume that if a document is more similar to the query than another document is, we would say the first document is more relevant than the second one. So in this case, the ranking function is defined as the similarity between the query and the document. One well-known example of this case is the **vector space model** [Salton et al. 1975], which we will cover more in detail later in this chapter.

The second set of models are called **probabilistic retrieval models** [Lafferty and Zhai 2003]. In this family of models, we follow a very different strategy. We assume that queries and documents are all observations from random variables, and we assume there is a binary random variable called R (with a value of either 1 or 0) to indicate whether a document is relevant to a query. We then define the score of a document with respect to a query as the probability that this random variable R is equal to 1 given a particular document and query. There are different cases of such a general idea. One is the classic probabilistic model, which dates back to work done in the 1960s and 1970s [Maron and Kuhns 1960, Robertson and Sparck Jones 1976], another is the language modeling approach [Ponte and Croft 1998], and yet another is the divergence-from-randomness model [Amati and Van Rijsbergen 2002]. We will cover a particular language modeling approach called query likelihood retrieval model in detail later in this chapter. One of the most effective retrieval models

derived from the classic probabilistic retrieval framework is BM25 [Robertson and Zaragoza 2009], but since the retrieval function of BM25 is so similar to a vector space retrieval model, we have chosen to cover it as a variant of the vector space model.

The third kind of model is probabilistic inference [Turtle and Croft 1990]. Here the idea is to associate uncertainty to inference rules. We can then quantify the probability that we can show that the query follows from the document. This family of models is theoretically appealing, but in practice, they are often reduced to models essentially similar to vector-space model or a regular probabilistic retrieval model.

Finally, there is also a family of models that use axiomatic thinking [Fang et al. 2011]. The idea is to define a set of constraints that we hope a good retrieval function satisfies. In this case the problem is to find a good ranking function that can satisfy all the desired constraints. Interestingly, although all these models are based on different thinking, in the end the retrieval functions tend to be very similar and involve similar variables. The axiomatic retrieval framework has proven effective for diagnosing deficiencies of a retrieval model and developing improved retrieval models accordingly (e.g., BM25+ [Lv and Zhai 2011]).

Although many models have been proposed, very few have survived extensive experimentation to prove effective and robustness. In this book, we have chosen to cover four specific models (i.e., BM25, pivoted length normalization, query likelihood with JM smoothing, and query likelihood with Dirichlet prior smoothing) that are among the very few most effective and robust models.[1]

6.2   Common Form of a Retrieval Function

Before we introduce specific models, we first take a look at the common form of a state-of-the-art retrieval model and examine some of the common ideas used in all these models. This is illustrated in Figure 6.1. First, these models are all based on the assumption of using a bag-of-words representation of text. This was explained in detail in the natural language processing chapter. A bag-of-words representation remains the main representation used in all the search engines. With this assumption, the score of a query like *presidential campaign news*, with respect to a document d, would be based on scores computed on each individual query word. That means the score would depend on the score of each word, such as *presidential*, *campaign*, and *news*.

---

1. PL2 is another very effective model that the readers should also know of [Amati and Van Rijsbergen 2002].

$f(q =$ "presidential campaign news", d)   "Bag of Words"

$g($"presidential", d) $g($"campaign", d) $g($"news", d)

How many times does "presidential" occur in d?
Term frequency (TF): $c($"presidential", d)

How long is d?   **Document length:** $|d|$

How often do we see "presidential" in the entire collection?
Document frequency: DF("presidential")
P("presidential"|collection)
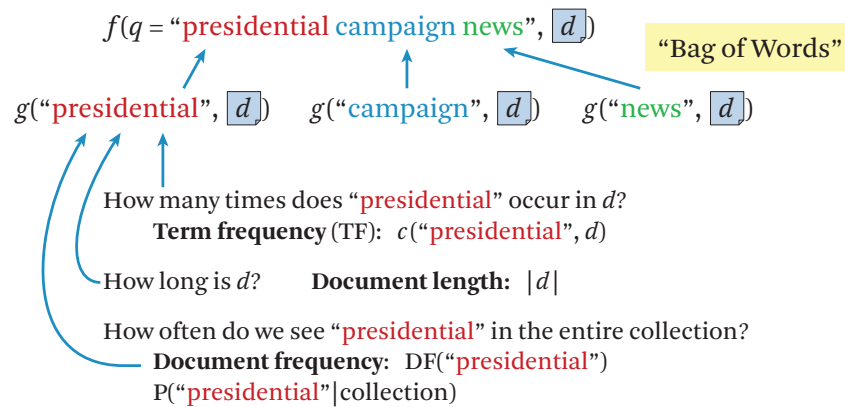
Figure 6.1   **Illustration of common ideas for scoring with a bag-of-words representation.**

We can see there are three different components, each corresponding to how well the document matches each of the query words. Inside of these functions, we see a number of heuristics. For example, one factor that affects the function g is how many times the word *presidential* occurs in each document. This is called a **term frequency** (TF). We might also denote this as $c(presidential, d)$. In general, if the word occurs more frequently in the document, the value of this function would be larger. Another factor is the **document length.** In general, if a term occurs in a long document many times, it is not as significant as if it occurred the same number of times in a short document (since any term is expected to occur more frequently in a long document). Finally, there is a factor called **document frequency**. This looks at how often *presidential* occurs at least once in any document in the entire collection. We call this the document frequency, or DF, of *presidential*. DF attempts to characterize the popularity of the term in the collection. In general, matching a rare term in the collection is contributing more to the overall score than matching a common term. TF, DF, and document length capture some of the main ideas used in pretty much all state-of-the-art retrieval models. In some other models we might also use a probability to characterize this information.

A natural question is: Which model works the best? It turns out that many models work equally well, so here we list the four major models that are generally regarded as state-of-the-art:

- pivoted length normalization [Singhal et al. 1996];
- Okapi BM25 [Robertson and Zaragoza 2009];

- query likelihood [Ponte and Croft 1998]; and

- PL2 [Amati and Van Rijsbergen 2002].

When optimized, these models tend to perform similarly as discussed in detail in Fang et al. [2011]. Among all these, BM25 is probably the most popular. It's most likely that this has been used in virtually all search engine implementations, and it is quite common to see this method discussed in research papers. We'll talk more about this method in a later section.

In summary, the main points are as follows. First, the design of a good ranking function requires a computational definition of relevance, and we achieve this goal by designing a proper retrieval model. Second, many models are equally effective but we don't have a single winner. Researchers are still actively working on this problem, trying to find a truly optimal retrieval model. Finally, the state-of-the-art ranking functions tend to rely on the following ideas: (1) bag of words representation; and (2) TF and the document frequency of words. Such information is used by a ranking function to determine the overall contribution of matching a word, with an adjustment for document length. These are often combined in interesting ways. We'll discuss how exactly they are combined to rank documents later in this book.

6.3 Vector Space Retrieval Models

The vector space (VS) retrieval model is a simple, yet effective method of designing ranking functions for information retrieval. It is a special case of similarity-based models that we discussed previously, where we assume relevance is roughly correlated to similarity between a document and a query. Whether this assumption is the best way to capture the notion of relevance formally remains an open question, but in order to solve our search problem we have to convert the vague notion of relevance into a more precise definition that can be implemented with a programming language in one way or another. In this process we inevitably have to make a number of assumptions. Here we assume that if a document is more similar to a query than another document, then the first document would be assumed to be more relevant than the second one. This is the basis for ranking documents in the vector space model. This is not the only way to formalize relevance; we will see later there are other ways to model relevance.

The basic idea of VS retrieval models is actually very easy to understand. Imagine a high dimensional space, where each dimension corresponds to a term; we can plot our documents in this space since they are represented as vectors of term magnitudes.
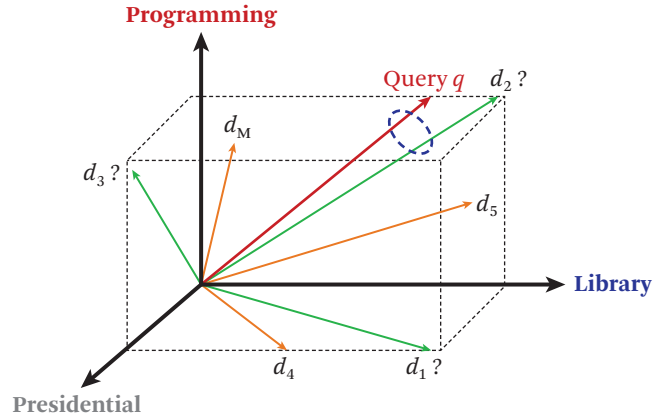
Figure 6.2   **Illustration of documents plotted in vector space.** (Courtesy of Marti Hearst)

In Figure 6.2, we show a three-dimensional space with three words: *programming*, *library*, and *presidential*. Each term defines one dimension. We can consider vectors in this three dimensional space, and we will assume all our documents and the query will all be placed in this vector space. For example, the vector d_1 represents a document that probably covers the terms *library* and *presidential* without really talking about *programming*. What does this mean in terms of representation of the document? It means that we will rely solely on this vector to represent the original document, and thus ignore everything else, including, e.g., the order of the words (which may sometimes be important to keep!). It is thus not an optimal representation, but it is often sufficient for many retrieval problems.

Intuitively, in this representation, d_1 seems to suggest a topic in either *presidential* or *library*. Now this is different from another document which might be represented as a different vector d_2. In this case, the document covers *programming* and *library*, but it doesn't talk about *presidential*. As you can probably guess, the topic is likely about programming language and the library is actually a software library. By using this vector space representation, we can intuitively capture the differences between topics of documents. Next, d_3 is pointing in a direction that might be about *presidential* and *programming*. We place all documents in our collection in this vector space and they will be pointing to all kinds of directions given by these three dimensions.

Similarly, we can place our query in this space as another vector. We can then measure the similarity between the query vector and every document vector. In this case, for example, we can easily see d_2 seems to be the closest to the query vector

and therefore d_2 will be ranked above the others. This is the main idea of the vector space model.

To be more precise, the VS model is a framework. In this framework, we make some assumptions. One assumption is that we represent each document and query by a term vector. Here, a term can be any basic concept such as a word or a phrase, or even n-grams of characters or any other feature representation. Each term is assumed to define one dimension. Therefore, since we have $|V|$ terms in our vocabulary, we define a $|V|$-dimensional space. A query vector would consist of a number of elements corresponding to the weights of different terms. Each document vector is also similar; it has a number of elements and each value of each element is indicating the weight of the corresponding term. The relevance in this case is measured based on the similarity between the two vectors. Therefore, our retrieval function is also defined as the similarity between the query vector and document vector.

Now, if you were asked to write a program to implement this approach for a search engine, you would realize that this explanation was far from complete. We haven't seen many things in detail, therefore it's impossible to actually write the program to implement this. That's why this is called the vector space retrieval framework. It has to be refined in order to actually suggest a particular function that can be implemented on a computer. First, it did not say *how to define or select the basic concepts* (terms). We clearly assume the concepts are orthogonal, otherwise there will be redundancy. For example, if two synonyms are somehow distinguished as two different concepts, they would be defined in two different dimensions, causing a redundancy or overemphasis of matching this concept (since it would be as if you matched two dimensions when you actually matched only one semantic concept). Second, it did not say *how to place documents and queries in this vector space*. We saw some examples of query and document vectors, but where exactly should the vector for a particular document point to? This is equivalent to how to define the term weights. This is a very important question because the term weight in the query vector indicates the importance of a term; depending on how you assign the weight, you might prefer some terms to be matched over others. Similarly, term weight in the document is also very meaningful—it indicates how well the term characterizes the document. If many nonrelevant documents are returned by a search engine using this model, then the chosen terms and weights must not represent the documents accurately. Finally, *how to define the similarity measure* is also unclear. These questions must be addressed before we can have an operational function that we can actually implement using a programming language. Solving these problems is the main topic of the next section.

6.3.1 Instantiation of the Vector Space Model

In this section, we will discuss how to instantiate a vector space model so that we can get a very specific ranking function. As mentioned previously, the vector space model is really a framework: it doesn't specify many things. For example, it did not say how we should define the dimensions of the vectors. It also did not say how we place a document vector or query vector into this space. That is, how should we define/calculate the values of all the elements in the query and document vectors? Finally, it did not say how we should compute similarity between the query vector and the document vector. As you can imagine, in order to implement this model, we have to determine specifically how we should compute and use these vectors.

In Figure 6.3, we illustrate the simplest instantiation of the vector space model. In this instantiation, we use each word in our vocabulary to define a dimension, thus giving $|V|$ dimensions—this is the bag-of-words instantiation. Now let's look at how we place vectors in this space. Here, the simplest strategy is to use a bit vector to represent both a query and a document, and that means each element x_i and y_i would take a value of either zero or one. When it's one, it means the corresponding word is present in the document or query. When it's zero, it's absent. If the user types in a few words for a query, then the query vector would have a few ones and many, many zeros. The document vector in general would have more ones than the query vector, but there will still be many zeros since the vocabulary is often very large. Many words in the vocabulary don't occur in a single document; many words will only occasionally occur in a given document. Most words in the vocabulary will be absent in any particular document.

Now that we have placed the documents and the query in the vector space, let's look at how we compute the similarity between them. A commonly used similarity measure is the dot product; the dot product of two vectors is simply defined as the sum of the products of the corresponding elements of the two vectors. In Figure 6.3 we see that it's the product of x_1 and y_1 plus the product of x_2 and y_2, and so on.

$$q = (x_1, ..., x_N) \qquad x_i, y_i \in \{0, 1\}$$
$$\textbf{1}\text{: word } W_i \text{ is } \textbf{present}$$
$$d = (y_1, ..., y_N) \qquad \textbf{0}\text{: word } W_i \text{ is } \textbf{absent}$$

$$Sim(q, d) = q.d = x_1 y_1 + \cdots + x_N y_N = \sum_{i=1}^{N} x_i y_i$$

Figure 6.3 Computing the similarity between a query and document vector using a bit vector representation and dot product similarity.

This is only one of the many different ways of computing the similarity. So, we've defined the dimensions, the vector space, and the similarity function; we finally have the simplest instantiation of the vector space model! It's based on the bit vector representation, dot product similarity, and bag of words instantiation. Now we can finally implement this ranking function using a programming language and then rank documents in our corpus given a particular query.

We've gone through the process of modeling the retrieval problem using a vector space model. Then, we made assumptions about how we place vectors in the vector space and how we define the similarity. In the end, we've got a specific retrieval function shown in Figure 6.3. The next step is to think about whether this individual function actually makes sense. Can we expect this function will actually perform well? It's worth thinking about the value that we are calculating; in the end, we've got a number, but what does this number mean? Please take a few minutes to think about that before proceeding to the next section.

6.3.2 Behavior of the Bit Vector Representation

In order to assess whether this simplest vector space model actually works well, let's look at the example in Figure 6.4.

This figure shows some sample documents and a simple query. The query is *news about presidential campaign*. For this example, we will examine five documents from the corpus that cover different terms in the query. You may realize that some documents are probably relevant and others probably not relevant. If we ask you to rank these documents, how would you rank them? Your answer (as the user) is the ideal ranking, $R'(q)$. Most users would agree that d_4 and d_3 are probably better than

Query = "news about presidential campaign"   Ideal ranking?

d_1 ... news about ... $d_4 +$
                                                                         $d_3 +$
d_2 ... news about organic food campaign ...

d_3 ... news of presidential campaign ...

d_4 ... news of presidential campaign ... $d_1 -$
          ... presidential candidate ... $d_2 -$
                                                                         $d_5 -$
d_5 ... news of organic food campaign ...
          campaign ... campaign ... campaign ...

Figure 6.4   Application of the bit vector VS model in a simple example.

Query = "news about presidential campaign"

d_1 | ... news about ... |

d_3 | ... news of presidential campaign ... |

$V = \{$news, about, presidential, campaign, food ...$\}$
$q = (1, \quad 1, \quad 1, \quad\quad\quad 1, \quad\quad\quad 0, \quad ...)$
$d_1 = (1, \quad 1, \quad 0, \quad\quad\quad 0, \quad\quad\quad 0, \quad ...)$
$\quad\quad f(q, d_1) = 1 * 1 + 1 * 1 + 1 * 0 + 1 * 0 + 0 * 0 + ... = 2$
$d_3 = (1, \quad\quad 0, \quad 1, \quad\quad\quad 1, \quad\quad\quad 0, \quad ...)$
$\quad\quad f(q, d_3) = 1 * 1 + 1 * 0 + 1 * 1 + 1 * 1 + 0 * 0 + ... = 3$

Figure 6.5   **Computation of the bit vector retrieval model on a sample query and corpus.**

the others since these two really cover the query well. They match *news*, *presidential*, and *campaign*, so they should be ranked on top. The other three, d_1, d_2, and d_5, are non-relevant.

Let's see if our vector space model could do the same or could do something close to our ideal ranking. First, think about how we actually use this model to score documents. In Figure 6.5, we show two documents, d_1 and d_3, and we have the query here also. In the vector space model, we want to first compute the vectors for these documents and the query. The query has four words, so for these four words, there would be a one and for the rest there will be zeros. Document d_1 has two ones, *news* and *about*, while the rest of the dimensions are zeros. Now that we have the two vectors, we can compute the similarity with the dot product by multiplying the corresponding elements in each vector. Each pair of vectors forms a product, which represents the similarity between the two items. We actually don't have to care about the zeroes in each vector since any product with one will be zero. So, when we take a sum over all these pairs, we're just counting how many pairs of ones there are. In this case, we have seen two, so the result will be two. That means this number is the value of this scoring function; it's simply the count of how many unique query terms are matched in the document. This is how we interpret the score. Now we can also take a look at d_3. In this case, you can see the result is three because d_3 matched the three distinct query words *news*, *presidential*, and *campaign*, whereas d_1 only matched two. Based on this, d_3 is ranked above d_1. That looks pretty good. However, if we examine this model in detail, we will find some problems.

Query = "news about presidential campaign"

d_1 | ... news about ... | $f(q, d_1) = 2$

d_2 | ... news about organic food campaign ... | $f(q, d_2) = 3$

d_3 | ... news of presidential campaign ... | $f(q, d_3) = 3$

d_4 | ... news of presidential campaign ...
... presidential candidate ... | $f(q, d_4) = 3$

d_5 | ... news of organic food campaign ...
campaign ... campaign ... campaign ... | $f(q, d_5) = 2$

Figure 6.6   **Ranking of example documents using the simple vector space model.**

In Figure 6.6, we show all the scores for these five documents. The bit vector scoring function counts the number of unique query terms matched in each document. If a document matches more unique query terms, then the document will be assumed to be more relevant; that seems to make sense. The only problem is that there are three documents, d_2, d_3, and d_4, that are tied with a score of three. Upon closer inspection, it seems that d_4 should be right above d_3 since d_3 only mentioned *presidential* once while d_4 mentioned it many more times. Another problem is that d_2 and d_3 also have the same score since for d_2, *news*, *about*, and *campaign* were matched. In d_3, it matched *news*, *presidential*, and *campaign*. Intuitively, d_3 is more relevant and should be scored higher than d_2. Matching *presidential* is more important than matching *about* even though *about* and *presidential* are both in the query. But this model doesn't do that, and that means we have to solve these problems.

To summarize, we talked about how to instantiate a vector space model. We need to do three things:

1. define the dimensions (the concept of what a document is);

2. decide how to place documents and queries as vectors in the vector space; and

3. define the similarity between two vectors.

Based on this idea, we discussed a very simple way to instantiate the vector space model. Indeed, it's probably the simplest vector space model that we can derive. We used each word to define a dimension, with a zero-one bit vector to represent a document or a query. In this case, we only care about word presence or absence, ignoring the frequency. For a similarity measure, we used the dot product

and showed that this scoring function scores a document based on the number of distinct query words matched in it. We also showed that such a simple vector space model still doesn't work well, and we need to improve it. This is the topic for the next section.

6.3.3 Improved Instantiation

In this section, we will improve the representation of this model from the bit vector model. We saw the bit vector representation essentially counts how many unique query terms match the document. From Figure 6.6 we would like d_4 to be ranked above d_3, and d_2 is really not relevant. The problem here is that this function couldn't capture the following characteristics.

- First, we would like to give more credit to d_4 because it matches *presidential* more times than d_3.

- Second, matching *presidential* should be more important than matching *about*, because *about* is a very common word that occurs everywhere; it doesn't carry that much content.

It's worth thinking at this point about why we have these issues. If we look back at the assumptions we made while instantiating the VS model, we will realize that the problem is really coming from some of those assumptions. In particular, it has to do with how we place the vectors in the vector space. Naturally, in order to fix these problems, we have to revisit those assumptions. A natural thought is to consider multiple occurrences of a term in a document as opposed to binary representation; we should consider the TF instead of just the absence or presence. In order to consider the difference between a document where a query term occurred multiple times and one where the query term occurred just once, we have to consider the **term frequency**—the count of a term in a document. The simplest way to express the TF of a word w in a document d is

$$TF(w, d) = \text{count}(w, d). \tag{6.1}$$

With the bit vector, we only captured the presence or absence of a term, ignoring the actual number of times that a term occurred. Let's add the count information back: we will represent a document by a vector with as each dimension's weight. That is, the elements of both the query vector and the document vector will not be zeroes and ones, but instead they will be the counts of a word in the query or the document, as illustrated in Figure 6.7.

$$q = (x_1, ..., x_N) \quad \boxed{x_i = \text{count of word } W_i \text{ in query}}$$

$$d = (y_1, ..., y_N) \quad \boxed{y_i = \text{count of word } W_i \text{ in doc}}$$

$$Sim(q, d) = q.d = x_1 y_1 + ... + x_N y_N = \Sigma_{i=1}^{N} x_i y_i$$

Figure 6.7   Frequency vector representation and dot product similarity.

Figure 6.8   Frequency vector representation rewards multiple occurrences of a query term.

Now, let's see what the formula would look like if we change this representation. The formula looks identical since we are still using the dot product similarity. The difference is inside of the sum since x_i and y_i are now different—they're now the counts of words in the query and the document. Because of the change in document representation, the new score has a different interpretation. We can see whether this would fix the problems of the bit vector VS model.

Look at the three documents again in Figure 6.8. The query vector is the same because all these words occurred exactly once in the query. The same goes for d_2 and d_3 since none of these words has been repeated. As a result, the score is also the same for both these documents. But, d_4 would be different; here, *presidential* occurred twice. Thus, the corresponding dimension would be weighted as two instead of one, and the score for d_4 is higher. This means, by using TF, we can now rank d_4 above d_2 and d_3 as we had hoped to.

Unfortunately, d_2 and d_3 still have identical scores. We would like to give more credit for matching *presidential* than matching *about*. How can we solve this problem in a general way? Is there any way to determine which word should be treated more importantly and which word can be essentially ignored? *About* doesn't carry that much content, so we should be able to ignore it. We call such a word a **stop word**. They are generally very frequent and they occur everywhere such that matching it doesn't have any significance. Can we come up with any statistical approaches to somehow distinguish a content word like *presidential* from a stop word like *about*? One difference is that a word like *about* occurs everywhere. If you count the occurrence of the word in the whole collection of M documents (where $M \gg 5$), then we would see that *about* has a much higher count than *presidential*. This idea suggests that we could somehow use the global statistics of terms or some other information to try to decrease the weight of the *about* dimension in the vector representation of d_2. At the same time, we hope to somehow increase the weight of *presidential* in the d_3 vector. If we can do that, then we can expect that d_2 will get an overall score of less than three, while d_3 will get a score of about three. That way, we'll be able to rank d_3 on top of d_2.

This particular idea is called the **inverse document frequency** (IDF). It is a very important signal used in modern retrieval functions. The **document frequency** is the count of documents that contain a particular term. Here, we say *inverse document frequency* because we actually want to reward a word that doesn't occur in many documents. The way to incorporate this into our vector is to modify the frequency count by multiplying it by the IDF of the corresponding word, as shown in Figure 6.9.

We can now penalize common words which generally have a low IDF and reward informative words that have a higher IDF. IDF can be defined as

$$\text{IDF}(w) = \left(\frac{M + 1}{\text{df}(w)} \right), \tag{6.2}$$

where M is the total number of documents in the collection and $\text{df}(\cdot)$ counts the document frequency (the total number of documents containing w).

Let's compare the terms *campaign* and *about*. Intuitively, *about* should have a lower IDF score than *campaign* since *about* is a less informative word. For clarity, let's assume $M = 10,000$, $\text{df}(\text{about}) = 5000$, $\text{df}(\text{campaign}) = 1166$, and we use a base two logarithm. Then,

$$\text{IDF}(\text{about}) = \log \left(\frac{10,001}{\text{df}(\text{about})} \right) = \log \left(\frac{10,001}{5000} \right) \approx 1.0$$
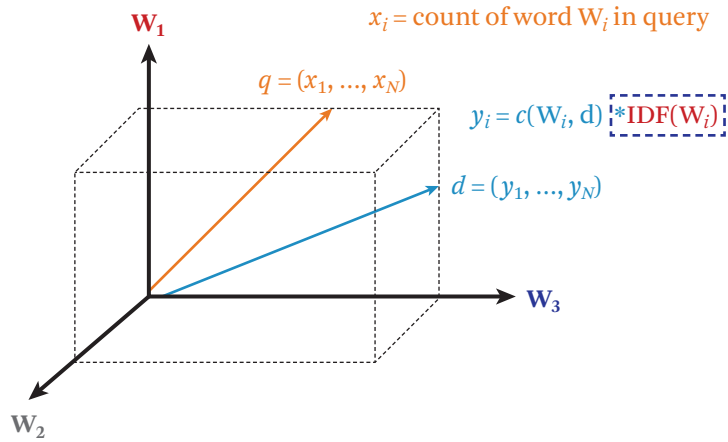
Figure 6.9   Representation of a blue document vector and red query vector with TF-IDF weighting.
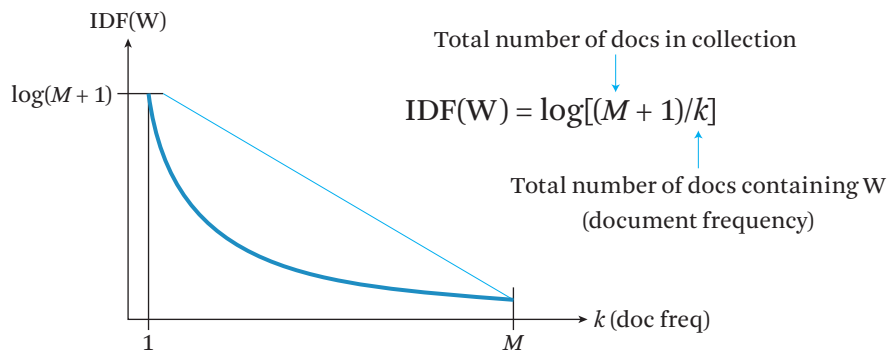
Figure 6.10   Illustration of the IDF function as the document frequency varies.

and

$$IDF(campaign) = \log\left(\frac{10,001}{df(campaign)}\right) = \log\left(\frac{10,001}{1166}\right) \approx 3.1.$$

Let k represent $df(w)$; if you plot the IDF function by varying k, then you will see a curve like the one illustrated in Figure 6.10. In general, you can see it would give a higher value for a low df, indicating a rare word. You can also see the maximum value of this function is $\log(M+1)$. The specific function is not as important as the heuristic it captures: penalizing popular terms. Whether there is a better form of

the IDF function is an open research question. With the evaluation skills you will learn in Chapter 9, you can test your different instantiations.

If we use a linear function like the diagonal line (as shown in the figure), it may not be as reasonable as the IDF function we just defined. In the standard IDF, we have a dropping off point where we say "these terms are essentially not very useful." This makes sense when the term occurs so frequently that it's unlikely to differentiate two documents' relevance (since the term is so common). But, if you look at the linear representation, there is no dropping off point. Intuitively, we want to focus more on the discrimination of low df words rather than these common words. Of course, which one works better still has to be validated by running experiments on a data set.

Let's look at the two documents again in Figure 6.11. Without IDF weighting, we just had bit vectors. With IDF weighting, we now can adjust the TF (term frequency) weight by multiplying it with the IDF weight. With this scheme, there is an adjustment by using the IDF value of *about* which is smaller than the IDF value of *presidential*. Thus, the IDF will distinguish these two words based on how informative they are. Including the IDF weighting causes d_3 to be ranked above d_2, since it matched a rare (informative) word, whereas d_2 matched a common (uninformative) word. This shows that the idea of weighting can solve our second problem. How effective is this model in general when we use this **TF-IDF weighting**? Well, let's take a look at all the documents that we have seen before.

In Figure 6.12, we show all the five documents that we have seen before and their new scores using TF-IDF weighting. We see the scores for the first four documents

$$
\begin{aligned}
d_2 \quad &\boxed{\text{... news about organic food campaign ...}} \\
d_3 \quad &\boxed{\text{... news of presidential campaign ...}}
\end{aligned}
$$

$V =$	{news,	about,	presidential,	campaign,	food	...}
IDF(W) =	1.5	1.0	2.5	3.1	1.8	
$q =$	(1,	1,	1,	1,	0,	...)
$d_2 =$	(1 * 1.5,	1 * 1.0,	0,	1 * 3.1,	0,	...)
$q =$	(1,	1,	1,	1,	0,	...)
$d_3 =$	(1 * 1.5,	0,	1 * 2.5,	1 * 3.1,	0,	...)

$$f(q, d_2) = 5.6 \quad < \quad f(q, d_3) = 7.1$$

Figure 6.11 The impact of IDF weighting on document ranking.

Query = "news about presidential campaign"

d_1 | ... news about ... | $f(q, d_1) = 2.5$

d_2 | ... news about organic food campaign ... | $f(q, d_2) = 5.6$

d_3 | ... news of presidential campaign ... | $f(q, d_3) = 7.1$

d_4 | ... news of presidential campaign ... ... presidential candidate ... | $f(q, d_4) = 9.6$

d_5 | ... news of organic food campaign ... campaign ... campaign ... campaign ... | $f(q, d_5) = 13.9!$

Figure 6.12   **Scores of all five documents using TF-IDF weighting.**

seem to be quite reasonable. But again, we also see a new problem since d_5 did not even have a very high score with our simplest vector space model, but now d_5 has the highest score. This is actually a common phenomenon when designing retrieval functions; when you try to fix one problem, you tend to introduce other problems! That's why it's very tricky to design an effective ranking function, and why finding a "best" ranking function is an open research question. In the next few sections, we'll continue to discuss some additional ideas to further improve this model and try to fix this problem.

6.3.4   TF Transformation

In the previous section, we derived a TF-IDF weighting formula using the vector space model and showed that this model actually works pretty well for the examples shown in the figures—except for d_5, which has a very high score. This document is intuitively non-relevant, so its position is not desirable. Now, we're going to discuss how to use a TF transformation to solve this problem. Before we discuss the details, let's take a look at the formula for the TF-IDF weighting and ranking function we previously derived. It is shown in Figure 6.13.

If you look at the formula carefully, you will see it involves a sum over all the matched query terms. Inside the sum, each matched query term has a particular weight; this weight is TF-IDF weighting. It has an IDF component where we see two variables: one is the total number of documents in the collection, M. The other is the document frequency, $\mathrm{df}(w)$, which is the number of documents that contain w. The other variables involved in the formula include the count of the query term

$$f(q, d) = \sum_{i=1}^{N} x_i y_i = \sum_{w \in q \cap d} c(w, q) c(w, d) \log \frac{M+1}{df(w)}$$

Total number of documents in collection

All matched query words in document

Document frequency

Figure 6.13   **A ranking function using a TF-IDF weighting scheme.**

w in the query, and the count of w in the document, represented as $c(w, q)$ and $c(w, d)$, respectively.

Looking at d_5 again, it's not hard to realize that the reason why it has received a high score is because it has a very high count of the term *campaign*. Its count in d_5 is four, which is much higher than the other documents, and has contributed to the high score of this document. Intriguingly, in order to lower the score for this document, we need to somehow restrict the contribution of matching this term in the document. Essentially, we shouldn't reward multiple occurrences so generously. The first occurrence of a term says a lot about matching of this term because it goes from a zero count to a count of one, and that increase is very informative. Once we see a word in the document, it's very likely that the document is talking about this word. If we see an extra occurrence on top of the first occurrence, that is to go from one to two, then we also can say the second occurrence confirmed that it's not an accidental mention of the word. But imagine we have seen, let's say, 50 occurrences of the word in the document. Then, adding one extra occurrence is not going to bring new evidence about the term because we are already sure that this document is about this word. Thus, we should restrict the contribution of a high-count term. That is exactly the idea of TF transformation, illustrated in Figure 6.14.

This transformation function is going to turn the raw count of word into a TF weight for the word in the document. On the x-axis is the raw count, and on the y-axis is the TF weight. In the previous ranking functions, we actually have implicitly used some kind of transformation. For example, in the zero-one bit vector representation, we actually used the binary transformation function as shown here. If the count is zero then it has zero weight. Otherwise it would have a weight of one. Then, we considered term count as a TF weight, which is a linear function. We just saw that this is not desirable. With a logarithm, we can have a sublinear

TF transformation: $c(w, d) \rightarrow \text{TF}(w, d)$

Term frequency weight
$y = \text{TF}(w, d)$

$y = x$

$y = \log(1 + x)$

$y = \log(1 + \log(1 + x))$

0/1 bit vector

$x = c(w, d)$

2

1

0   1   2   3   ...
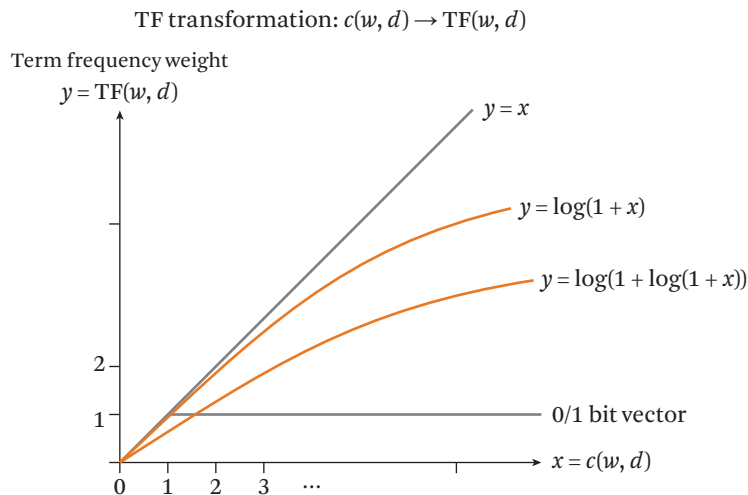
Figure 6.14   **Illustration of different ways to transform TF.**

transformation that looks like the red lines in the figure. This will control the influence of a very high weight because it's going to lower its influence, yet it will retain the influence of a small count. We might even want to bend the curve more by applying a logarithm twice. Researchers have tried all these methods and they are indeed working better than the linear transformation, but so far what works the best seems to be this special transformation called **BM25 TF**, illustrated in Figure 6.15, where BM stands for best matching.

In this transformation, there is a parameter k which controls the upper bound of this function. It's easy to see this function has a upper bound, because if you look at the $\frac{x}{x+k}$ as being multiplied by $(k + 1)$, the fraction will never exceed one, since the numerator is always less than the denominator. Thus, it's upper-bounded by $(k + 1)$. This is also the difference between the BM25 TF function and the logarithm transformation, which doesn't have an upper bound. Furthermore, one interesting property of this function is that as we vary k, we can actually simulate different transformation functions including the two extremes that are shown in the figure. When $k = 0$, we have a zero one bit transformation. If we set k to a very large number, on the other hand, it's going to look more like the linear transformation function. In this sense, this transformation is very flexible since it allows us to control the shape of the TF curve quite easily. It also has the nice property of a simple upper bound. This upper bound is useful to control the influence of a particular term. For example, we can prevent a spammer from just increasing the count of one term to spam all queries that might match this term. In other words, this upper bound

Term frequency weight
$y = \text{TF}(w, d)$
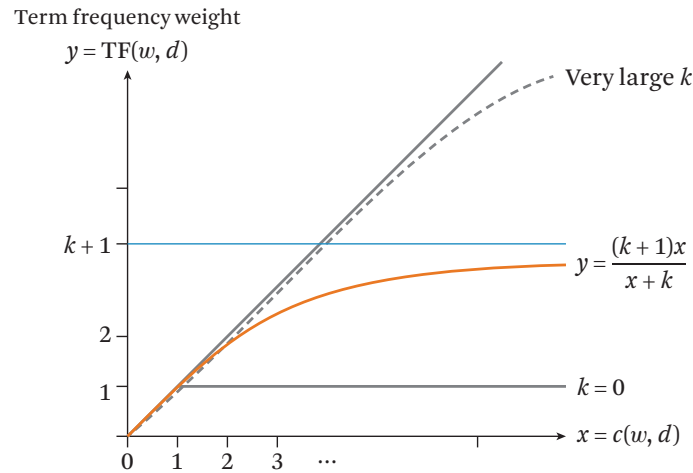
Figure 6.15   Illustration of BM25 TF transformation.

ensures that all terms will be counted when we aggregate the weights to compute a score.

To summarize, we need to capture some sublinearity in the TF function. This ensures that we represent the intuition of diminishing return from high term counts. It also avoids a dominance by one single term over all others. The BM25 TF formula we discussed has an upper bound while being robust and effective. If we plug this function into our TF-IDF vector space model, then we would end up having a ranking function with a BM25 TF component. This is very close to a state-of-the-art ranking function called BM25. We'll see the entire BM25 formula soon.

6.3.5   Document Length Normalization

In this section, we will discuss the issue of document length normalization. So far in our exploration of the vector space model we considered the TF or the count of a term in a document. We have also considered the global statistic IDF. However, we have not considered the document length.

In Figure 6.16, we show two example documents. Document d_4 is very short with only one hundred words. Conversely, d_6 has five thousand words. If you look at the matching of these query words we see that d_6 has many more matchings of the query words; one might reason that d_6 may have matched these query words in a scattered manner. Perhaps d_6's topic is not really the same as the query's topic. In the beginning of d_6, there is discussion of a campaign. This discussion may have

Query = "news about presidential campaign"
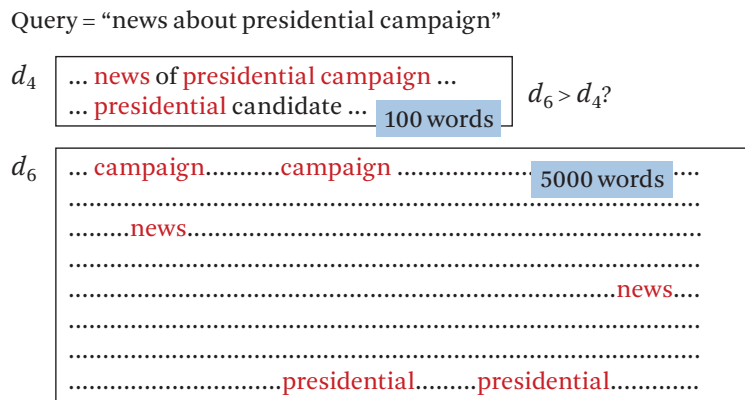
Figure 6.16   Two documents with very different document lengths.

nothing to do with the mention of *presidential* at the end. In general, if you think about long documents, they would have a higher chance to match any query since they contain more words. In fact, if you generate a long document by randomly sampling words from the distribution of all words, then eventually you probably will match any query! In this sense, we should penalize long documents because they naturally have a better chance to match any query. This is our idea of document length normalization. On the one hand, we want to penalize a long document, but on the other hand, we also don't want to over-penalize them. The reason is that a document may be long because of different reason: in one case the document may be longer because it uses more words. For example, think about a research paper article. It would use more words than the corresponding abstract. This is the case where we probably should penalize the matching of a long document such as a full paper. When we compare matching words in such long document with matching words in the short abstract, the long papers generally have a higher chance of matching query words. Therefore, we should penalize the long documents.

However, there is another case when the document is long—that is when the document simply has more content. Consider a case of a long document, where we simply concatenated abstracts of different papers. In such a case, we don't want to penalize this long document. That's why we need to be careful about using the right degree of length penalization, and an understanding of the discourse structure of documents is needed for optimal document length normalization.

A method that has worked well is called pivoted length normalization, illustrated in Figure 6.17 and described originally in Singhal et al. [1996]. Here, the idea is to

Pivoted length normalization

$$\text{normalizer} = 1 - b + b\,\frac{|d|}{avdl} \qquad\qquad b \in [0, 1]$$

$b \gg 0$

$b > 0$

Reward

1.0

$b = 0$

Penalization

$|d|$

0   1   2   ... $avdl$ ...

Shorter than $avdl$ Longer than $avdl$
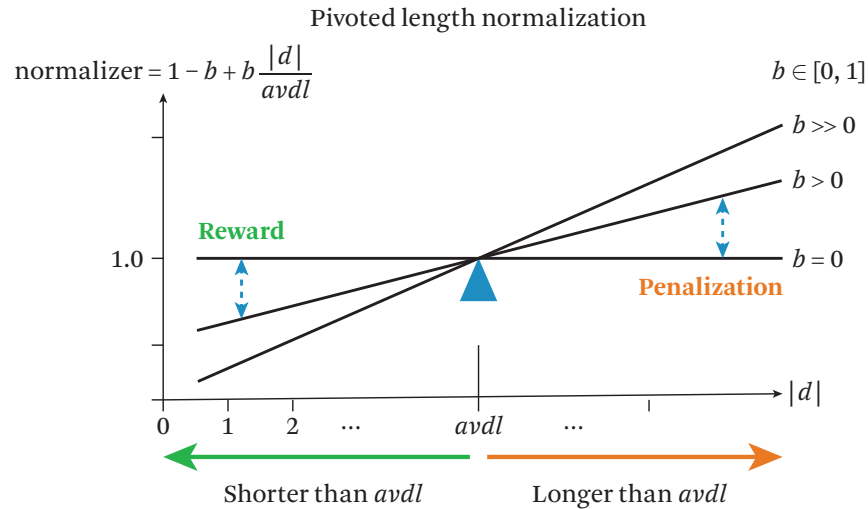
Figure 6.17   Illustration of pivoted document length normalization.

use the average document length as a pivot, or reference point. That means we will assume that for the average length documents, the score is about right (a normalizer would be one). If a document is longer than the average document length, then there will be some penalization. If it's shorter than the average document length, there's even some reward. The x-axis represents the length of a document. On the y-axis we show the normalizer, i.e., the pivoted length normalization. The formula for the normalizer is an interpolation of one and the normalized document lengths, controlled by a parameter b. When we first divide the length of the document by the average document length, this not only gives us some sense about how this document is compared with the average document length, but also gives us the benefit of not worrying about the unit of length. This normalizer has an interesting property; first, we see that if we set the parameter b to zero, then the normalizer value would be one, indicating no length normalization at all. If we set b to a nonzero value, then the value would be higher for documents that are longer than the average document length, whereas the value of the normalizer will be smaller for shorter documents. In this sense we see there's a penalization for long documents and a reward for short documents. The degree of penalization is controlled by b. By adjusting b (which varies from zero to one), we can control the degree of length normalization. If we plug this length normalization factor into the

Pivoted length normalization VSM

$$f(q,d) = \sum_{w \in q \cap d} c(w,q) \frac{\ln(1 + \ln(1 + c(w,d)))}{1 - b + b\frac{|d|}{avdl}} \log \frac{M+1}{\mathrm{df}(w)} \quad b \in [0, 1]$$

BM25/Okapi

$$f(q,d) = \sum_{w \in q \cap d} c(w,q) \frac{(k+1)c(w,d)}{c(w,d) + k(1 - b + b\frac{|d|}{avdl})} \log \frac{M+1}{\mathrm{df}(w)} \quad b \in [0, 1], \;\; k \in [0, +\infty)$$

Figure 6.18   **State-of-the-art vector space models: pivoted length normalization and Okapi BM25.**

vector space model ranking functions that we have already examined, we will end up with state-of-the-art retrieval models, some of which are shown in Figure 6.18.

Let's take a look at each of them. The first one is called **pivoted length normalization**. We see that it's basically the TF-IDF weighting model that we have discussed. The IDF component appears in the last term. There is also a query TF component, and in the middle there is normalized TF. For this, we have the double logarithm as we discussed before; this is to achieve a sublinear transformation. We also put a document length normalizer in the denominator of the TF formula, which causes a penalty for long documents, since the larger the denominator is, the smaller the TF weight is. The document length normalization is controlled by the parameter b.

The next formula is called **Okapi BM25**, or just BM25. It's similar to the pivoted length normalization formula in that it has an IDF component and a query TF component. In the middle, the normalization is a little bit different; we have a sublinear transformation with an upper bound. There is a length normalization factor here as well. It achieves a similar effect as discussed before, since we put the normalizer in the denominator. Thus, again, if a document is longer, the term weight will be smaller.

We have now reached one of the best-known retrieval functions by thinking logically about how to represent a document and by slowly tweaking formulas and considering our initial assumptions.

6.3.6   Further Improvement of Basic VS Models

So far, we have talked mainly about how to place the document vector in vector space. This has played an important role in determining the performance of the ranking function. However, there are also other considerations that we did not really examine in detail. We've assumed that we can represent a document as a bag of words. Obviously, we can see there are many other choices. For example,

stemmed words (words that have been transformed into a basic root form) are a viable option so all forms of the same word are treated as one, and can be matched as one term. We also need to perform stop word removal; this removes some very common words that don't carry any content such as *the*, *a*, or *of*. We could use phrases or even latent semantic analysis, which characterizes documents by which cluster words belong to. We can also use smaller units, like character *n*-grams, which are sequences of *n* characters, as dimensions. In practice, researchers have found that the bag-of-words representation with phrases (or "bag-of-phrases") is the most effective representation. It's also efficient so this is still by far the most popular document representation method and it's used in all the major search engines.

Sometimes we need to employ language-specific and domain-specific representation. This is actually very important as we might have variations of the terms that prevent us from matching them with each other even though they mean the same thing. Take Chinese, for example. We first need to segment text to obtain word boundaries because it's originally just a sequence of characters. A word might correspond to one character or two characters or even three characters. It's easier in English when we have a space to separate the words, but in some other languages we may need to do some natural language processing to determine word boundaries.

There is also possibility to improve the similarity function. So far, we've used the dot product, but there are other measures. We could compute the cosine of the angle between two vectors, or we can use a Euclidean distance measure. The dot product still seems the best and one of the reasons is because it's very general; in fact, it's sufficiently general. If you consider the possibilities of doing weighting in different ways, cosine measure can be regarded as the dot product of two normalized vectors. That means we first normalize each vector, and then we take the dot product. That would be equivalent to the cosine measure.

We mentioned that BM25 seems to be one of the most effective formulas—but there has also been further development in improving BM25, although none of these works have changed the BM25 fundamentally. In one line of work, people have derived BM25-F. Here, *F* stands for field, and this is BM25 for documents with structure. For example, you might consider the title field, the abstract field, the body of the research article, or even anchor text (on web pages). These can all be combined with an appropriate weight on different fields to help improve scoring for each document. Essentially, this formulation applies BM25 on each field, and then combines the scores, but keeps global (i.e., across all fields) frequency counts. This has the advantage of avoiding over-counting the first occurrence of the term. Recall that in the sublinear transformation of TF, the first occurrence is very important

and contributes a large weight. If we do that for all the fields, then the same term might have gained a large advantage in every field. When we just combine counts on each separate field, the extra occurrences will not be counted as fresh first occurrences. This method has worked very well for scoring structured documents. More details can be found in Robertson et al. [2004].

Another line of extension is called BM25+. Here, researchers have addressed the problem of over-penalization of long documents by BM25. To address this problem, the fix is actually quite simple. We can simply add a small constant to the TF normalization formula. But what's interesting is that we can analytically prove that by doing such a small modification, we will fix the problem of over-penalization of long documents by the original BM25. Thus, the new formula called BM25+ is empirically and analytically shown to be better than BM25 [Lv and Zhai 2011].

6.3.7   Summary

In vector space retrieval models, we use similarity as a notion of relevance, assuming that the relevance of a document with respect to a query is correlated with the similarity between the query and the document. Naturally, that implies that the query and document must be represented in the same way, and in this case, we represent them as vectors in a high dimensional vector space. The dimensions are defined by words, concepts, or terms. We generally need to use multiple heuristics to design a ranking function; we gave some examples which show the need for several heuristics, which include:

- TF (term frequency) weighting and sublinear transformation;
- IDF (inverse document frequency) weighting; and
- document length normalization.

These three are the most important heuristics to ensure such a general ranking function works well for all kinds of tasks. Finally, BM25 and pivoted length normalization seem to be the most effective VS formulas. While there has been some work done in improving these two powerful measures, their main idea remains the same. In the next section, we will discuss an alternative approach to the vector space representation.

6.4   Probabilistic Retrieval Models

In this section, we will look at a very different way to design ranking functions than the vector space model that we discussed before. In probabilistic models, we define the ranking function based on the probability that a given document d is relevant

to a query q, or $p(R = 1 \mid d, q)$ where $R \in \{0, 1\}$ is a binary random variable denoting relevance. In other words, we introduce a binary random variable R and we model the query and the documents as observations from random variables.

Note that in the vector space model, we assume that documents are all equal-length vectors. Here, we assumed they are the data observed from random variables. Thus, the problem is to estimate the probability of relevance.

In this category of models, there are many different variants. The classic probabilistic model has led to the BM25 retrieval function, which we discussed in the vector space model section because its form is quite similar to these types of models. We will discuss another special case of probabilistic retrieval functions called *language modeling* approaches to retrieval. In particular, we're going to discuss the query likelihood retrieval model, which is one of the most effective models in probabilistic models. There is also another line of functions called *divergence-from-randomness models* (such as the PL2 function [Amati and Van Rijsbergen 2002]). It's also one of the most effective state-of-the-art retrieval models.

In query likelihood, our assumption is that this probability of relevance can be approximated by the probability of a query given a document and relevance, $p(q \mid d, R = 1)$. Intuitively, this probability just captures the following probability: if a user likes document d, how likely would the user enter query q in order to retrieve document d? The condition part contains document d and $R = 1$, which can be interpreted as the condition that the user likes document d. To understand this idea, let's first take a look at the basic idea of probabilistic retrieval models.

Figure 6.19 lists some imagined relevance status values (or *relevance judgments*) of queries and documents. It shows that q_1 is a query that the user typed in and d_1 is a document the user has seen. A "1" in the far right column means the user thinks d_1 is relevant to q_1. The R here can be also approximated by the clickthrough data that the search engine can collect by watching how users interact with the search results. In this case, let's say the user clicked on document d_1, so there's a one associated with the pair (q_1, d_1). Similarly, the user clicked on d_2, so there's a one associated with (q_1, d_2). Thus, d_2 is assumed to be relevant to q_1 while d_3 is non-relevant, d_4 is non-relevant, d_5 is again relevant, and so on and so forth. Perhaps the second half of the table (after the ellipses) is from a different user issuing the same queries. This other user typed in q_1 and then found that d_1 is actually not useful, which is in contrast to the first user's judgement.

We can imagine that we have a large amount of search data and are able to ask the question, "how can we estimate the probability of relevance?" Simply, if we look at all the entries where we see a particular d and a particular q, we can calculate how likely we will see a one in the third column. We can first count how many times

Query q	Document d	Relevant? R
q1	d1	1
q1	d2	1
q1	d3	0
d1	d4	0
q1	d5	1
$\vdots$		
q1	d1	0
d1	d2	1
q1	d3	0
q2	d3	1
q3	d1	1
q4	d2	1
q4	d3	0

$$f(q,d) = p(R = 1 \mid d, q) = \frac{\text{count}(q, d, R = 1)}{\text{count}(q, d)}$$

$P(R = 1 \mid \text{q1}, \text{d1}) = 1/2$

$P(R = 1 \mid \text{q1}, \text{d2}) = 2/2$

$P(R = 1 \mid \text{q1}, \text{d3}) = 0/2$

Figure 6.19   **Basic idea of probabilistic models for information retrieval.**

we see q and d as a pair in this table and then count how many times we actually have also seen a one in the third column and compute the ratio:

$$p(R = 1 \mid d, q) = \frac{\text{count}(R = 1, d, q)}{\text{count}(d, q)}. \tag{6.3}$$

Clearly, $p(R = 1 \mid d, q) + p(R = 0 \mid d, q) = 1$.

Let's take a look at some specific examples. Suppose we are trying to compute this probability for d_1, d_2, and d_3 for q_1. What is the estimated probability? If we are interested in q_1 and d_1, we consider the two pairs containing q_1 and d_1; only in one of the two cases has the user said that the document is relevant. So R is equal to 1 in only one of the two cases, which gives our probability a value of 0.5. What about d_2 and d_3? For d_2, R is equal to 1 in both cases. For d_3, R is equal to 0 in both cases. We now have a score for d_1, d_2, and d_3 for q_1. We can simply rank them based on these probabilities—that's the basic idea of probabilistic retrieval model. In our example,

it's going to rank d_2 above all the other documents because in all the cases, given q_1 and d_2, $R = 1$.
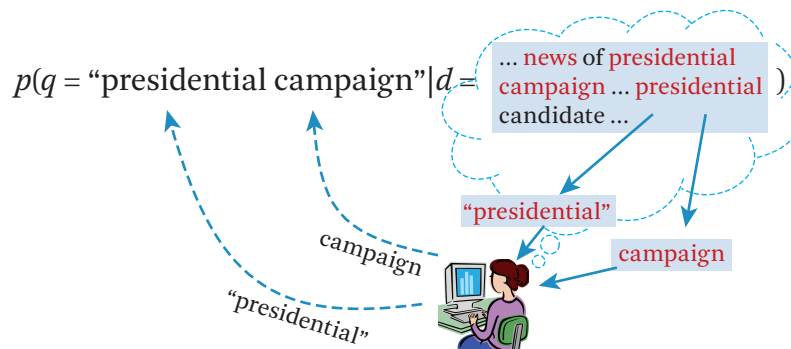
With volumes of clickthrough data, a search engine can learn to improve its results. This is a simple example that shows that with even a small number of entries, we can already estimate some probabilities. These probabilities would give us some sense about which document might be more useful to a user for this query. Of course, the problem is that we don't observe all the queries and all of the documents and all the relevance values; there will be many unseen documents. In general, we can only collect data from the documents that we have shown to the users. In fact, there are even more unseen queries because you cannot predict what queries will be typed in by users. Obviously, this approach won't work if we apply it to unseen queries or unseen documents. Nevertheless, this shows the basic idea of the probabilistic retrieval model.

What do we do in such a case when we have a lot of unseen documents and unseen queries? The solution is that we have to approximate in some way. In the particular case called the query likelihood retrieval model, we just approximate this by another conditional probability, $p(q \mid d, R = 1)$ [Lafferty and Zhai 2003].

We assume that the user likes the document because we have seen that the user clicked on this document, and we are interested in all these cases when a user liked this particular document and want to see what kind of queries they have used. Note that we have made an interesting assumption here: we assume that a user formulates the query based on an imaginary relevant document. If you just look at this as a conditional probability, it's not obvious we are making this assumption. We have to somehow be able to estimate this conditional probability without relying on the big table from Figure 6.19. Otherwise, we would have similar problems as before. By making this assumption, we have some way to bypass the big table.

Let's look at how this new model works for our example. We ask the following question: which of these documents is most likely the imaginary relevant document in the user's mind when the user formulates this query? We quantify this probability as a conditional probability of observing this query if a particular document is in fact the imaginary relevant document in the user's mind. We compute all these query likelihood probabilities—that is, the likelihood of the query given each document. Once we have these values, we can then rank these documents.

To summarize, the general idea of modeling relevance in the probabilistic retrieval model is to assume that we introduce a binary random variable R and let the scoring function be defined based on the conditional probability $p(R = 1 \mid d, q)$. We also talked about approximating this by using query likelihood. This means we have a ranking function that's based on a probability of a query given the document.

$p(q = \text{``presidential campaign''}|d =$

... news of presidential campaign ... presidential candidate ...

"presidential"

campaign

campaign

"presidential"

If the user is **thinking of this doc**,
how likely would she **pose this query**?

Figure 6.20 **Generating a query by sampling words from a document.**

This probability should be interpreted as the probability that a user who likes document d would pose query q. Now the question, of course, is how do we compute this conditional probability? We will discuss this in detail in the next section.

6.4.1 The Query Likelihood Retrieval Model

In the query likelihood retrieval model, we quantify how likely a user would pose a particular query in order to find a particular document.

Figure 6.20 shows how the query likelihood model assumes a user imagines some ideal document and generates a query based on that ideal document's content. In this example, the ideal document is about "presidential campaign news." Under this model, the user would use this ideal document as a basis to compose a query to try and retrieve a desired document. More concretely, we assume that the query is generated by sampling words from the document. For example, a user might pick a word like *presidential* from this imaginary document, and then use this as a query word. The user would then pick another word like *campaign*, and that would be the second query word. Of course, this is only an assumption we have made about how users pose queries. Whether a user actually follows this process is a different question. Importantly, though, this assumption has allowed us to formally characterize the conditional probability of a query given a document without relying on the big table that was presented earlier. This is why we can use this fundamental idea to further derive retrieval functions that we can implement with language models.

$$p(q = \text{``presidential campaign''}|d) = \frac{c(\text{``presidential''}, d)}{|d|} * \frac{c(\text{``campaign''}, d)}{|d|}$$

$$p(q|d_4) = \boxed{\begin{array}{l} \text{... news of presidential campaign} \\ \text{... presidential candidate ...} \end{array}}\ ) = \frac{2}{|d_4|} * \frac{1}{|d_4|}$$

$$p(q|d_3) = \boxed{\text{... news of presidential campaign ...}}\ ) = \frac{1}{|d_3|} * \frac{1}{|d_3|}$$

$$p(q|d_2) = \boxed{\begin{array}{l} \text{... news about organic food} \\ \text{campaign ...} \end{array}}\ ) = \frac{0}{|d_2|} * \frac{1}{|d_2|} = 0$$

Figure 6.21 **Computing the probability of a query given a document using the query likelihood formulation.**

We've made the assumption that each query word is independent and that each word is obtained from the imagined ideal document satisfying the user's information need. Let's see how this works exactly. Since we are computing a query likelihood, then the total probability is the probability of this particular query, which is a sequence of words. Since we make the assumption that each word is generated independently, the probability of the query is just a product of the probability of each query word, where the probability of each word is just the relative frequency of the word in the document. For example, the probability of *presidential* given the document would be just the count of *presidential* in the document divided by the total number of words in the document (i.e., the document length). We now have an actual formula for retrieval that we can use to rank documents.

Let's take a look at some example documents from Figure 6.21. Suppose now the query is *presidential campaign*. To score these documents, we just count how many times we have seen *presidential* and how many times we have seen *campaign*. We've seen *presidential* two times in d_4, so that's $\frac{2}{|d_4|}$. We also multiply by $\frac{1}{|d_4|}$ for the probability of *campaign*. Similarly, we can calculate probabilities for the other two documents d_3 and d_2. If we assume d_3 and d_4 have about the same length, then it looks like we will rank d_4 above d_3, which is above d_2. As we would expect, it looks like this formulation captures the TF heuristic from the vector space models.

However, if we try a different query like this one, *presidential campaign update*, then we might see a problem. Consider the word *update*: none of the documents contain this word. According to our assumption that a user would pick a word from a document to generate a query, the probability of obtaining a word like *update*

would be zero. Clearly, this causes a problem because it would cause all these documents to have zero probability of generating this query.

While it's fine to have a zero probability for d_2 which is not relevant, it's not okay to have zero probability for d_3 and d_4 because now we no longer can distinguish them. In fact, we can't even distinguish them from d_2. Clearly, that's not desirable. When one has such a result, we should think about what has caused this problem, examining what assumptions have been made as we derive this ranking function. We have made an assumption that every query word must be drawn from the document in the user's mind—in order to fix this, we have to assume that the user could have drawn a word not necessarily from the document. So let's consider an improved model.

Instead of drawing a word from the document, let's imagine that the user would actually draw a word from a document language model as depicted in Figure 6.22. Here, we assume that this document is generated by using this unigram language model, which doesn't necessarily assign zero probability for the word *update*. In fact, we assume this model does not assign zero probability for *any* word. If we're thinking this way, then the generative process is a bit different: the user has this model (distribution of words) in mind instead of a particular ideal document, although the model still has to be estimated based on the documents in our corpus.

The user can generate the query using a similar process. They may pick a word such as *presidential* and another word such as *campaign*. The difference is that now we can pick a word like *update* even though it doesn't occur in the document. This
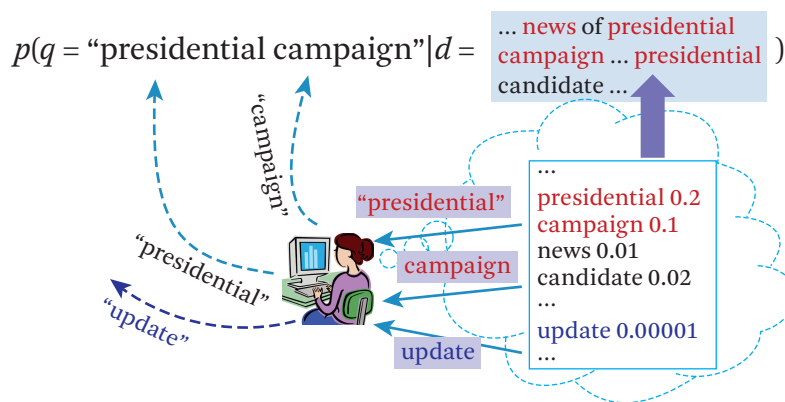

Figure 6.22   Computing the probability of a query given a document using a document language model.
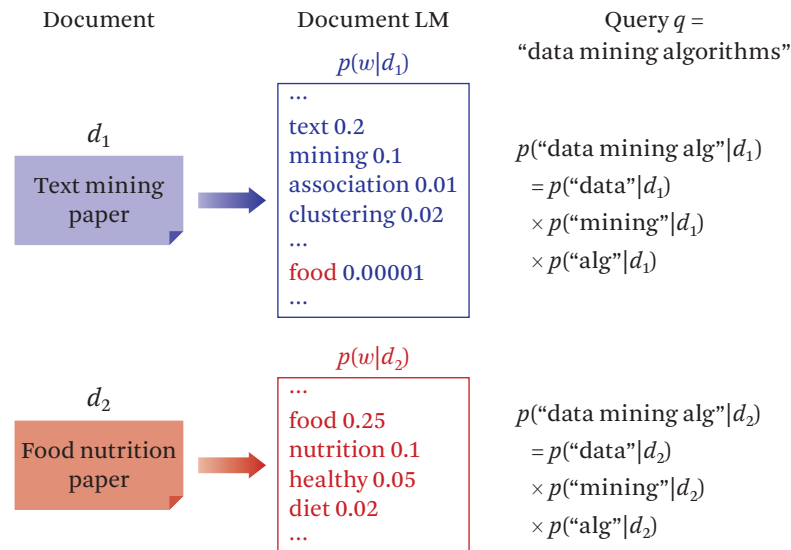
Document   Document LM   Query q = "data mining algorithms"

$p(w|d_1)$

...
text 0.2
mining 0.1
association 0.01
clustering 0.02
...
food 0.00001
...

d_1

Text mining paper

$p(\text{"data mining alg"}|d_1)$
$= p(\text{"data"}|d_1)$
$\quad \times p(\text{"mining"}|d_1)$
$\quad \times p(\text{"alg"}|d_1)$

$p(w|d_2)$

...
food 0.25
nutrition 0.1
healthy 0.05
diet 0.02
...

d_2

Food nutrition paper

$p(\text{"data mining alg"}|d_2)$
$= p(\text{"data"}|d_2)$
$\quad \times p(\text{"mining"}|d_2)$
$\quad \times p(\text{"alg"}|d_2)$

Figure 6.23   **Scoring a query on two documents based on their language models.**

would fix our problem with zero probabilities and it's also reasonable because we're now thinking of what the user is looking for in a more general way, via a unigram language model instead of a single fixed document.

In Figure 6.23, we show two possible language models based on documents d_1 and d_2, and a query *data mining algorithms*. By making an independence assumption, we could have $p(q \mid d)$ as a product of the probability of each query word in each document's language model. We score these two documents and then rank them based on the probabilities we calculate.

Let's formally state our scoring process for query likelihood. A query q contains the words

$$q = w_1, w_2, \ldots, w_n$$

such that $|q| = n$. The scoring or ranking function is then the probability that we observe q given that a user is thinking of a particular document d. This is the product of probabilities of all individual words, which is based on the independence assumption mentioned before:

$$p(q \mid d) = p(w_1 \mid d) \times p(w_2 \mid d) \times \ldots \times p(w_n \mid d). \qquad (6.4)$$

In practice, we score the document for this query by using a logarithm of the query likelihood:

$$\text{score}(q, d) = \log p(q \mid d) = \sum_{i=1}^{n} \log p(w_i \mid d) = \sum_{w \in V} c(w, q) \log p(w \mid d). \quad (6.5)$$

We do this to avoid having numerous small probabilities multiplied together, which could cause underflow and precision loss. By transforming using a logarithm, we maintain the order of these documents while simultaneously avoiding the underflow problem. Note the last term in the equation above; in this sum, we have a sum over all the possible words in the vocabulary V and iterate through each word in the query. Essentially, we are only considering the words in the query because if a word is not in the query, its contribution to the sum would be zero.

The only part we don't know is this document language model, $p(w \mid d)$. Therefore, we can convert the retrieval problem into the problem of estimating this document language model so that we can compute the probability of a query being generated by each document. Different estimation methods for $p(w \mid d)$ lead to different ranking functions, and this is just like the different ways to place a document into a vector in the vector space model. Here, there are different ways to estimate parameters in the language model, which lead to different ranking functions for query likelihood.

6.4.2 Smoothing the Document Language Model

When calculating the query likelihood retrieval score, recall that we take a sum of log probabilities over all of the query words, using the probability of a word in the query given the document (i.e., the document language model). The main task now is to estimate this document language model. In this section we look into this task in more detail.

First of all, how do we estimate this language model? The obvious choice would be the maximum likelihood estimation (MLE) that we have seen before in Chapter 2. In MLE, we normalize the word frequencies in the document by the document length. Thus, all the words that have the same frequency count will have an equal probability under this estimation method. Note that words that have not occurred in the document will have zero probability. In other words, we assume the user will sample a word from the document to formulate the query, and there is no chance of sampling any word that is not in the document. But we know that's not good, so how would we improve this? In order to assign a non-zero probability to words that have not been observed in the document, we would have to take away some

probability mass from seen words because we need some extra probability mass for the unseen words—otherwise, they won't sum to one.

To make this transformation and to improve the MLE, we will assign nonzero probabilities to words that are not observed in the data. This is called *smoothing*, and smoothing has to do with improving the estimate by including the probabilities of unseen words. Considering this factor, a smoothed language model would be a more accurate representation of the actual document. Imagine you have seen the abstract of a research paper; or, imagine a document is just an abstract. If we assume words that don't appear in the abstract have a probability of zero, that means sampling a word outside the abstract is impossible. Imagine the user who is interested in the topic of this abstract; the user might actually choose a word that is not in the abstract to use as query. In other words, if we had asked this author to write more, the author would have written the full text of the article, which contains words that don't appear in the abstract. So, smoothing the language model is attempting to try to recover the model for the whole article. Of course, we don't usually have knowledge about the words not observed in the abstract, so that's why smoothing is actually a tricky problem.

The key question here is what probability should be assigned to those unseen words. As one would imagine, there are many different approaches to solve this issue. One idea that's very useful for retrieval is to let the probability of an unseen word be proportional to its probability as given by a reference language model. That means if you don't observe the word in the corpus, we're going to assume that its probability is governed by another reference language model that we construct. It will tell us which unseen words have a higher probability than other unseen words. In the case of retrieval, a natural choice would be to take the collection LM as the reference LM. That is to say if you don't observe a word in the document, we're going to assume that the probability of this word would be proportional to the probability of the word in the whole collection.

More formally, we'll be estimating the probability of a word given a document as follows:

$$p(w \mid d) = \begin{cases} p_{\text{seen}}(w \mid d) & \text{if } w \text{ seen in } d \\ \alpha_d \cdot p(w \mid C) & \text{otherwise.} \end{cases} \qquad (6.6)$$

If the word is seen in the document, then the probability would be a discounted MLE estimate p_{seen}. Otherwise, if the word is not seen in the document, we'll let the probability be proportional to the probability of the word in the collection $p(w \mid C)$, with the coefficient α_d controlling the amount of probability mass that we assign

$$\log p(q|d) = \sum_{w \in V} c(w, q) \log p(w|d)$$

$$= \sum_{w \in V, c(w,d)>0} c(w, q) \log p_{\text{Seen}}(w|d) \; + \; \boxed{\sum_{w \in V, c(w,d)=0} c(w, q) \log \alpha_d \, p(w|C)}$$

Query words **matched** in d Query words **not matched** in d

$$= \sum_{w \in V} c(w, q) \log \alpha_d \, p(w|C) \; - \; \sum_{w \in V, c(w,d)>0} c(w, q) \log \alpha_d \, p(w|C)$$

All query words   Query words **matched** in d

$$= \sum_{w \in V, c(w,d)>0} c(w, q) \log \frac{p_{\text{Seen}}(w|d)}{\alpha_d \, p(w|C)} \; + \; |q| \log \alpha_d \; + \; \sum_{w \in V} c(w, q) \log p(w|C)$$

Figure 6.24   Substituting smoothed probabilities into the query likelihood retrieval formula.

to unseen words. Regardless of whether the word w is seen in the document or not, all these probabilities must sum to one, so α_d is constrained.

Now that we have this smoothing formula, we can plug it into our query likelihood ranking function, illustrated in Figure 6.24. In this formula, we have a sum over all the query words, written in the form of a sum over the corpus vocabulary. Although we sum over words in the vocabulary, in effect we are just taking a sum of query words since each word is weighted by its frequency in the query. Such a way to write this sum is convenient in some transformations. In our smoothing method, we're assuming the words that are not observed in the document have a somewhat different form of probability. Using this form we can decompose this sum into two parts: one over all the query words that are matched in the document and the other over all the words that are *not* matched. These unmatched words have a different form of probability because of our assumption about smoothing.

We can then rewrite the second sum (of query words not matched in d) as a difference between the scores of all words in the vocabulary minus all the query words matched in d. This is actually quite useful, since part of the sum over all $w \in V$ can now be written as $|q| \log \alpha_d$. Additionally, the sum of query words matched in d

$$\log p(q \mid d) = \sum_{\substack{w \in d \\ w \in q}} c(w, q)[\log \frac{p_{\text{Seen}}(w \mid d)}{\alpha_d p(w \mid C)}] + n \log \alpha_d + \boxed{\sum_{i=1}^{N} \log p(w_i \mid C)}$$

TF weighting — $p_{\text{Seen}}(w \mid d)$

Doc length normalization — α_d

Matched query terms — $c(w, q)$

IDF weighting — $\alpha_d p(w \mid C)$

Ignore for ranking — $\sum_{i=1}^{N} \log p(w_i \mid C)$

Figure 6.25   **The query likelihood retrieval formula captures the three heuristics from the vector space models.**

is in terms of words that we observe in the query. Just like in the vector space model, we are now able to take a sum of terms in the intersection of the query vector and the document vector.

If we look at this rewriting further as shown in Figure 6.25, we can see how it actually would give us two benefits. The first benefit is that it helps us better understand the ranking function. In particular, we're going to show that from this formula we can see the connection of smoothing using a collection language model with weighting heuristics similar to TF-IDF weighting and length normalization. The second benefit is that it also allows us to compute the query likelihood more efficiently, since we only need to consider terms matched in the query.

We see that the main part of the formula is a sum over the matching query terms. This is much better than if we take the sum over all the words. After we smooth the document using the collection language model, we would have nonzero probabilities for all the words $w \in V$. This new form of the formula is much easier to compute. It's also interesting to note that the last term is independent of the document being scored, so it can be ignored for ranking. Ignoring this term won't affect the order of the documents since it would just be the same value added onto each document's final score.

Inside the sum, we also see that each matched query term would contribute a weight. This weight looks like TF-IDF weighting from the vector space models. First, we can already see it has a frequency of the word in the query, just like in the vector space model. When we take the dot product, the word frequency in the query appears in the sum as a vector element from the query vector. The corresponding term from the document vector encodes a weight that has an effect similar to TF-IDF weighting. p_{seen} is related to the term frequency in the sense that if a word occurs very frequently in the document, then the seen probability will tend to be

larger. This term is really doing something like TF weighting. In the denominator, we achieve the IDF effect through $p(w \mid C)$, or the popularity of the term in the collection. Because it's in the denominator, a larger collection probability actually makes the weight of the entire term smaller. This means a popular term carries a smaller weight—this is precisely what IDF weighting is doing! Only now, we have a different form of TF and IDF. Remember, IDF has a logarithm of document frequency, but here we have something different. Intuitively, however, it achieves a similar effect to the VS interpretation.

We also have something related to the length normalization. In particular, α_d might be related to document length. It encodes how much probability mass we want to give to unseen words, or how much smoothing we are allowed to do. Intuitively, if a document is long then we need to do less smoothing because we can assume that it is large enough that we have probably observed all of the words that the author could have written. If the document is short, the number of unseen words is expected to be large, and we need to do more smoothing in this case. Thus, α_d penalizes long documents since it tends to be smaller for long documents. The variable α_d actually occurs in two places. Thus its overall effect may not necessarily be penalizing long documents, but as we will see later when we consider smoothing methods, α_d would always penalize long documents in a specific way.

This formulation is quite convenient since it means we don't have to think about the specific way of doing smoothing. We just need to assume that if we smooth with the collection language model, then we would have a formula that looks like TF-IDF weighting and document length normalization. It's also interesting that we have a very fixed form of the ranking function. Note that we have not heuristically put a logarithm here, but have used a logarithm of query likelihood for scoring and turned the product into a sum of logarithms of probabilities. If we only want to heuristically implement TF-IDF weighting, we don't necessarily have to have a logarithm. Imagine if we drop this logarithm; we would still have TF and IDF weighting. But, what's nice with probabilistic modeling is that we are automatically given a logarithm function which achieves sublinear scaling of our term "weights."

In summary, a nice property of probabilistic models is that by following some assumptions and probabilistic rules, we'll get a formula by derivation. If we heuristically design the formula, we may not necessarily end up having such a specific form. Additionally, we talked about the need for smoothing a document language model. Otherwise, it would give zero probability for unseen words in the document, which is not good for scoring a query with an unseen word. It's also necessary to improve the accuracy of estimating the model representing the topic of this document. The general idea of smoothing in retrieval is to use the collection language model to give us some clue about which unseen word would have a higher proba-

bility. That is, the probability of the unseen word is assumed to be proportional to its probability in the entire collection. With this assumption, we've shown that we can derive a general ranking formula for query likelihood retrieval models that automatically contains the vector space heuristics of TF-IDF weighting and document length normalization.

We also saw that through some rewriting, the scoring of such a ranking function is primarily based on a sum of weights on matched query terms, also just like in the vector space model. The actual ranking function is given to us automatically by the probabilistic derivation and assumptions we have made, unlike in the vector space model where we have to heuristically think about the forms of each function. However, we still need to address the question: how exactly should we smooth a document language model? How exactly should we use the reference language model based on the collection to adjust the probability of the MLE of seen terms? This is the topic of the next section.

6.4.3 Specific smoothing methods

From the last section, we showed how to smooth the query likelihood retrieval model with the collection language model. We end up having a retrieval function that looks like the following:

$$\sum_{w \in d, q} c(w, q) \log \left(\frac{p_{\text{seen}}(w \mid d)}{\alpha_d \cdot p(w \mid C)} \right) + |q| \log \alpha_d. \tag{6.7}$$

We can see it's a sum of all the matched query terms, and inside the sum it's a count of terms in the query with some weight for the term in the document. We saw in the previous section how TF and IDF are captured in this sum. We also mentioned how the second term α_d can be used for document length normalization. If we wanted to implement this function using a programming language, we'd still need to figure out a few variables; in particular, we're going to need to know how to estimate the probability of a word and how to set α_d. In order to answer these questions, we have to think about specific smoothing methods, where we define p_{seen} and α_d.

We're going to talk about two different smoothing methods. The first is a linear interpolation with a fixed mixing coefficient. This is also called **Jelinek-Mercer smoothing**. The idea is actually quite simple. Figure 6.26 shows how we estimate the document language model by using MLE. That gives us word counts normalized by the total number of words in the document. The idea of using this method is to maximize the probability of the observed text. As a result, if a word like *network* is not observed in the text, it's going to get zero probability. The idea of smoothing is to rely on the collection reference model where this word is not going to have a

zero probability, helping us decide what non-zero probability should be assigned to such a word. In Jelinek-Mercer smoothing, we do a linear interpolation between the maximum likelihood estimate and the collection language model. This is controlled by the smoothing parameter $\lambda \in [0, 1]$. Thus, λ is a smoothing parameter for this particular smoothing method. The larger λ is, the more smoothing we have, putting more weight on the background probabilities. By mixing the two distributions together, we achieve the goal of assigning non-zero probability to unseen words in the document that we're currently scoring.

So let's see how it works for some of the words here. For example, if we compute the smoothed probability for the word *text*, we get the MLE estimate in the document interpolated with the background probability. Since *text* appears ten times in d and $|d| = 100$, our MLE estimate is $\frac{10}{100}$. In the background, we have $p(\text{text} \mid C) = 0.001$, giving our smoothed probability of

$$p_{\text{seen}}(w \mid d) = (1 - \lambda) \cdot p_{MLE}(w \mid d) + \lambda \cdot p(w \mid C)$$

$$= (1 - \lambda) \cdot \frac{10}{100} + \lambda \cdot 0.001.$$

In Figure 6.26 we also consider the word *network*, which does not appear in d. In this case, the MLE estimate is zero, and its smoothed probability is $0 + \lambda \cdot p(w \mid C) = \lambda \cdot 0.001$. You can see now that α_d in this smoothing method is just λ

$$p(w|d) = (1 - \lambda)\frac{c(w, d)}{|d|} + \lambda\, p(w|C) \qquad \lambda \in [0, 1]$$

| | Document d | Collection LM |
| Unigram LM $p(w|\theta) = ?$ | Total #words $= 100$ | $P(w|C)$ |

...	text 10	the 0.1
10/100 → text ?	mining 5	a 0.08
5/100 → mining ?	association 3	...
3/100 → association ?	database 3	computer 0.02
3/100 → database ?	algorithm 2	database 0.01
...	...	...
1/100 → query ?	query 1	**text 0.001**
0/100 → **network** ?	efficient 1	**network 0.001**
		mining 0.0009
		...

$$p(\text{``text''}|d) = (1 - \lambda)\frac{10}{100} + \lambda * 0.001 \qquad p(\text{``network''}|d) = \lambda * 0.001$$
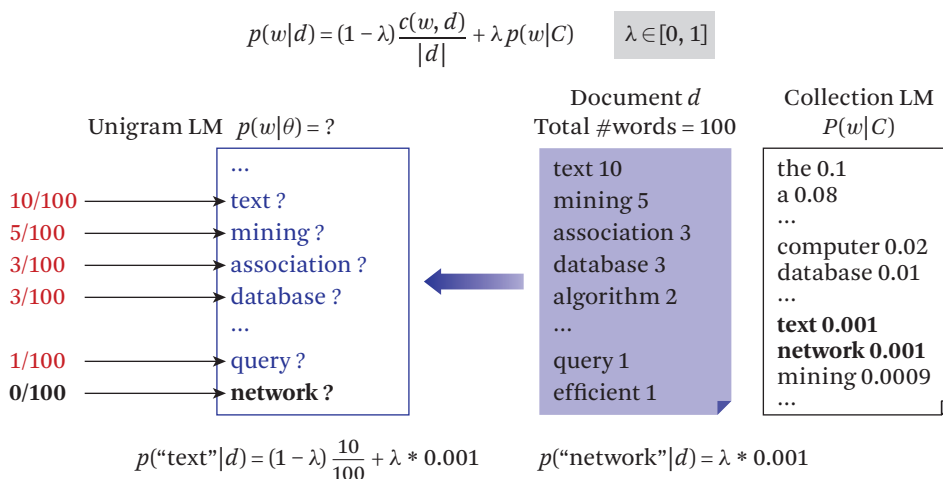
Figure 6.26   Smoothing the query likelihood retrieval function with linear interpolation: Jelinek-Mercer smoothing.

because that's the coefficient in front of the probability of the word given by the collection language model.

The second smoothing method we will discuss is called **Dirichlet prior smoothing**, or *Bayesian smoothing*. Again, we face the problem of zero probability for words like *network*. Just like Jelinek-Mercer smoothing, we'll use the collection language model, but in this case we're going to combine it with the MLE esimate in a somewhat different way. The formula first can be seen as an interpolation of the MLE probability and the collection language model as before. Instead, however, α_d is not simply a fixed λ, but a dynamic coefficient which takes $\mu > 0$ as a parameter.

Based on Figure 6.27, we can see if we set μ to a constant, the effect is that a long document would actually get a smaller coefficient here. Thus, a long document would have less smoothing as we would expect, so this seems to make more sense than fixed-coefficient smoothing. The two coefficients $\frac{|d|}{|d|+\mu}$ and $\frac{\mu}{|d|+\mu}$ would still sum to one, giving us a valid probability model. This smoothing can be understood as a dynamic coefficient interpolation. Another way to understand this formula—which is even easier to remember—is to rewrite this smoothing method in this form:

$$p(w \mid d) = \frac{c(w,d) + \mu \cdot p(w \mid C)}{|d| + \mu}. \qquad (6.8)$$

$$p(w|d) = \frac{c(w,d) + \mu\, p(w|C)}{|d| + \mu} = \frac{|d|}{|d|+\mu}\frac{c(w,d)}{|d|} + \frac{\mu}{|d|+\mu}p(w|C) \qquad \mu \in [0, +\infty)$$

Unigram LM $\;p(w|\theta) = ?$

Document d
Total #words = 100

Collection LM $P(w|C)$

...	
10/100	text ?
5/100	mining ?
3/100	association ?
3/100	database ?
...	
1/100	query ?
0/100	**network ?**

Document d
text 10
mining 5
association 3
database 3
algorithm 2
...
query 1
efficient 1

Collection LM
the 0.1
a 0.08
...
computer 0.02
database 0.01
...
text 0.001
network 0.001
mining 0.0009
...

$$p(\text{``text''}|d) = \frac{10 + \mu * 0.001}{100 + \mu} \qquad p(\text{``network''}|d) = \frac{\mu}{100 + \mu} * 0.001$$
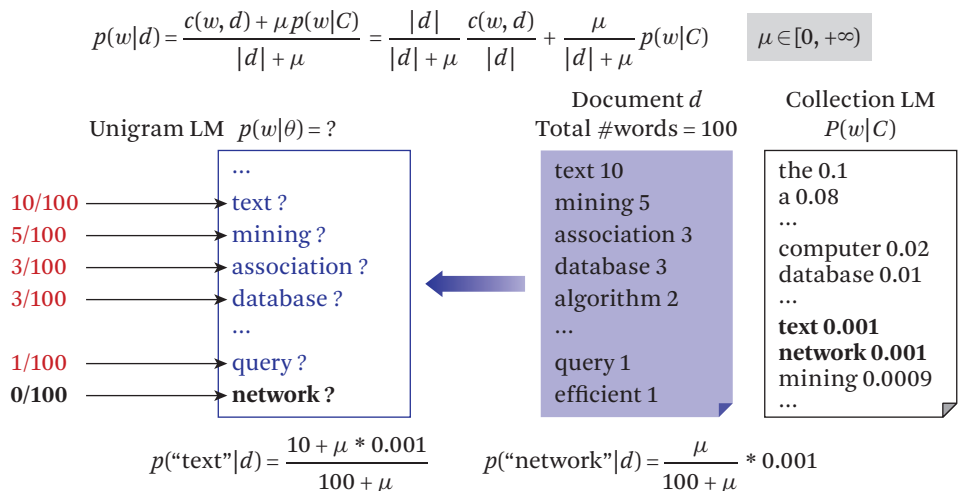
Figure 6.27 Smoothing the query likelihood retrieval function with linear interpolation: Dirichlet prior smoothing.

Here, we can easily see what change we have made to the MLE. In this form, we see that we add a count of $\mu \cdot p(w \mid C)$ to every word, which is proportional to the probability of w in the entire corpus. We pretend every word w has $\mu \cdot p(w \mid C)$ additional pseudocounts. Since we add this extra probability mass in the numerator, we have to re-normalize in order to have a valid probability distribution. Since $\sum_{w \in V} p(w \mid C) = 1$, we can add a μ in the denominator, which is the total number of pseudocounts we added for each w in the numerator.

Let's also take a look at this specific example again. For the word *text*, we will have ten counts that we actually observe but we also added some pseudocounts which are proportional to the probability of *text* in the entire corpus. Say we set $\mu = 3000$, meaning we will add 3000 extra word counts into our smoothed model. We want some portion of the 3000 counts to be allocated to *text*; since $p(\text{text} \mid C) = 0.001$, we'll assign $0.001 \cdot 3000$ counts to that word. The same goes for the word *network*; for d, we observe zero counts, but also add $\mu \cdot p(\text{network} \mid C)$ extra pseudocounts for our smoothed probability.

In Dirichlet prior smoothing, α_d will actually depend on the current document being scored, since $|d|$ is used in the smoothed probability. In the Jelinek-Mercer linear interpolation, $\alpha_d = \lambda$, which is a constant. For Dirichlet prior, we have $\alpha_d = \frac{\mu}{|d|+\mu}$, which is the interpolation coefficient applied to the collection language model. For a slightly more detailed derivation of these variables, the reader may consult Appendix A.

Now that we have defined p_{seen} and α_d for both smoothing methods, let's plug these variables in the original smoothed query likelihood retrieval function. Let's start with Jelinek-Mercer smoothing:

$$\frac{p_{\text{seen}}(w \mid d)}{\alpha_d \cdot p(w \mid C)} = \frac{(1 - \lambda) \cdot p_{MLE}(w \mid d) + \lambda \cdot p(w \mid C)}{\lambda \cdot p(w \mid C)} = 1 + \frac{1 - \lambda}{\lambda} \cdot \frac{c(w, d)}{|d| \cdot p(w \mid C)}. \quad (6.9)$$

Then, plugging this into the entire query likelihood retrieval formula, we get

$$\text{score}_{JM}(q, d) = \sum_{w \in q, d} c(w, q) \log \left(1 + \frac{1 - \lambda}{\lambda} \cdot \frac{c(w, d)}{|d| \cdot p(w \mid C)} \right). \quad (6.10)$$

We ignore the $|q| \log \alpha_d$ additive term (derived in the previous section) since $\alpha_d = \lambda$ does not depend on the current document being scored. We'll end up having a ranking function that is strikingly similar to a vector space model since it is a sum over all the matched query terms. The value of the logarithm term is non-negative. We see very clearly the TF weighting in the numerator, which is scaled sublinearly. We also see the IDF-like weighting, which is the $p(w \mid C)$ term in the denominator; the more frequent the term is in the entire collection, the more

discounted the numerator will be. Finally, we can see the $|d|$ in the denominator is a form of document length normalization, since as $|d|$ grows, the overall term weight would decrease, suggesting that the impact of α_d in this case is clearly to penalize a long document. The second fraction can also be considered as the ratio of two probabilities; if the ratio is greater than one, it means the probability of w in d is greater than appearing by chance in the background. If the ratio is less than one, the chance of seeing w in d is actually less likely than observing it in the collection.

What's also important to note is that we received this weighting function automatically by making various assumptions, whereas in the vector space model, we had to go through those heuristic design choices in order to get this. These are the advantages of using this kind of probabilistic reasoning where we have made explicit assumptions. We know precisely why we have a logarithm here, and precisely why we have these probabilities. We have a formula that makes sense and does TF-IDF weighting and document length normalization.

Let's look at the complete function for Dirichlet prior smoothing now. We know what p_{seen} is and we know that $\alpha_d = \frac{\mu}{|d|+\mu}$:

$$p_{\text{seen}}(w \mid d) = \frac{c(w,d) + \mu \cdot p(w \mid C)}{|d|+\mu} = \frac{|d|}{|d|+\mu} \cdot \frac{c(w,d)}{|d|} + \frac{\mu}{|d|+\mu} \cdot p(w \mid C), \text{ (6.11)}$$

therefore,

$$\frac{p_{\text{seen}}(w \mid d)}{\alpha_d \cdot p(w \mid C)} = \frac{\frac{c(w,d)+\mu \cdot p(w|C)}{|d|+\mu}}{\frac{\mu \cdot p(w|C)}{|d|+\mu}} = 1 + \frac{c(w,d)}{\mu \cdot p(w \mid C)}. \tag{6.12}$$

We can now substitute this into the complete formula:

$$\text{score}_{DIR}(q,d) = \sum_{w \in q,d} c(w,q) \log\left(1 + \frac{c(w,d)}{\mu \cdot p(w \mid C)}\right) + |q| \log \frac{\mu}{\mu + |d|}. \tag{6.13}$$

The form of the function looks very similar to the Jelinek-Mercer scoring function. We compute a ratio that is sublinearly scaled by a non-negative logarithm. Both TF and IDF are computed in almost the exact same way. The difference here is that Dirichlet prior smoothing can capture document length normalization differently than Jelinek-Mercer smoothing. Here, we have retained the $|q| \log \alpha_d$ term since α_d depends on the document, namely $|d|$. If $|d|$ is large, then less extra mass is added onto the final score; if $|d|$ is small, more extra mass is added to the score, effectively rewarding a short document.

To summarize this section, we've talked about two smoothing methods: Jelinek-Mercer, which is doing the fixed coefficient linear interpolation, and Dirichlet prior,

which adds pseudo counts proportional to the probability of the current word in the background collection. In most cases we can see, by using these smoothing methods, we will be able to reach a retrieval function where the assumptions are clearly articulated, making them less heuristic than some of the vector space models. Even though we didn't explicitly set out to define the popular VS heuristics, in the end we naturally arrived at TF-IDF weighting and document length normalization, perhaps justifying their inclusion in the VS models. Each of these functions also has a smoothing parameter (λ or μ) with an intuitive meaning. Still, we need to set these smoothing parameters or estimate them in some way. Overall, this shows that by using a probabilistic model, we follow very different strategies than the vector space model. Yet in the end, we end up with retrieval functions that look very similar to the vector space model. Some advantages here are having assumptions clearly stated and a final form dictated by a probabilistic model.

This section also concludes our discussion of the query likelihood probabilistic retrieval models. Let's recall what assumptions we have made in order to derive the functions that we have seen the following.

1. The relevance can be modeled by the query likelihood, i.e., $p(R \mid d, q) \approx p(q \mid d)$.

2. Query words are generated independently, allowing us to decompose the probability of the whole query into a product of probabilities of observed words in the query.

3. If a word is not seen in the document, its probability is proportional to its probability in the collection (smoothing with the background collection).

4. Finally, we made one of two assumptions about the smoothing, using either Jelinek-Mercer smoothing or Dirichlet prior smoothing.

If we make these four assumptions, then we have no choice but to take the form of the retrieval function that we have seen earlier. Fortunately, the function has a nice property in that it implements TF-IDF weighting and document length normalization. In practice, these functions also work very well. In that sense, these functions are less heuristic compared with the vector space model.

Bibliographic Notes and Further Reading

A brief review of many different kinds of retrieval models can be found in Chapter 2 Zhai [2008]. The vector space model with pivoted length normalization was proposed and discussed in detail in Singhal et al. [1996]. The query likelihood retrieval model was initially proposed in Ponte and Croft [1998]. A useful reference for the

BM25 retrieval function is Robertson and Zaragoza [2009]. A comprehensive survey of language models for information retrieval can be found in Zhai [2008]. A formal treatment of retrieval heuristics is given in Fang et al. [2004], and a diagnostic evaluation method for assessing deficiencies of a retrieval model is proposed in Fang et al. [2011], where multiple improved basic retrieval functions are also derived.

Exercises

6.1. Here's a query and document vector. What is the score for the given document using dot product similarity?

$$d = \{1, 0, 0, 0, 1, 4\} \qquad q = \{2, 1, 0, 1, 1, 1\}$$

6.2. In what kinds of queries do we probably not care about query term frequency?

6.3. Let d be a document in a corpus. Suppose we add another copy of d to collection. How does this affect the IDF of all words in the corpus?

6.4. Given a fixed vocabulary size, the length of a document is the same as the length of the vector used to represent it. True or false? Why?

6.5. Consider Euclidean distance as our similarity measure for text documents:

$$d(q, d) = \sqrt{\sum_{i=1}^{|V|} (q_i - d_i)^2}.$$

What does this measure capture compared to the cosine measure discussed in this chapter? Would you prefer one over the other?

6.6. If you perform stemming on words in V to create V' then $|V'| > |V|$. True or false? Why?

6.7. Which of the following ways is best to reduce the size of the vocabulary in a large corpus?

- Remove top 10 words
- Remove words that occur 10 times or fewer

6.8. Why might using raw term frequency counts with dot product similarity not give the best possible ranking?

6.9. How can you apply the VS model to a domain other than text documents? For example, how do you find similar movies in IMDB or similar music to a specific song? Hint: first define your concept space; what is your "term" vector?

6.10. In Okapi BM25, how can we remove document length normalization by setting a parameter? What value should it have?

6.11. Examine the Okapi BM25 retrieval function in META. You should see that it is slightly different than the formula discussed in this chapter. What are the differences and what do you suppose their effect is?

6.12. How are query likelihood and language models related?

6.13. In a unigram document LM, how many parameters are needed? (That is, how many probabilities must be known in order to describe the LM?)

6.14. In a bigram document LM, how many parameters are needed?

6.15. Given a unigram language model θ estimated from this book's content, and two documents $d_1 =$ "information retrieval" and $d_2 =$ "retrieval information", then $p(d_1 \mid \theta) > p(d_2 \mid \theta)$. True or false? Why?

6.16. For this and the next question, refer to this probabilistic retrieval method called **absolute discounting**:

$$p_s(w \mid d) = \frac{\max(c(w, d) - \delta, 0)}{|d|} + \frac{\delta |d|_u}{|d|} \cdot p(w \mid C)$$

and

$$\alpha_d = \frac{\delta |d|_u}{|d|},$$

where $\delta \in [0, 1]$ or $|d|_u$ is the total number of unique terms in a particular document d.

  What happens in the extreme cases where $\delta = 0$ and $\delta = 1$?

6.17. Does absolute discounting capture document length normalization? How?

6.18. Give two reasons why Dirichlet Prior smoothing is better than Add-1 smoothing, which is defined as

$$p_s(w \mid d) = \frac{c(w, d) + 1}{|d| + |V|}.$$

6.19. Which heuristics from the vector space models are captured in the general smoothed query likelihood formula?

6.20. Is the following formula an acceptable scoring function? Why or why not?

$$\text{score}(q, d) = \sum_{w \in q, d} \frac{k \cdot c(w, C)}{c(w, d)} \cdot \ln\left(\frac{N + 1}{\text{df}(w)}\right) \cdot \frac{n}{n_{avg}},$$

where:

- $k > 0$ is some parameter;
- $c(w, C)$ and $c(w, d)$ are the count of the current word in the collection and current document, respectively;
- N is the total number of documents;
- $\mathrm{df}(w)$ is the number of documents that the current word w appears in;
- n is the document length of the current document;
- n_{avg} is the average document length of the corpus.

7 Feedback

In this chapter, we will discuss feedback in a TR system. Feedback takes the results of a user's actions or previous search results to improve retrieval results. This is illustrated in Figure 7.1. As shown, feedback is often implemented as updates to a query, which alters the list of returned documents. We can see the user would type in a query and then the query would be sent to a standard search engine, which returns a ranked list of results (we discussed this in depth in Chapter 6). These search results would be shown to the user. The user can make judgements about whether each returned document is useful or not. For example, the user may say one document is good or one document is not very useful. Each decision on a document is called a **relevance judgment**. This overall process is a type of relevance feedback, because we've got some feedback information from the user based on the judgements of the search results.

As one would expect, this can be very useful to the retrieval system since we should be able to learn what exactly is interesting to a particular user or users. The feedback module would then take these judgements as input and also use the document collection to try to improve future rankings. As mentioned, it would typically involve updating the query so the system can now rank the results more accurately for the user; this is the main idea behind relevance feedback.

These types of relevance judgements are reliable, but the users generally don't want to make extra effort unless they have to. There is another form of feedback called **pseudo relevance feedback**, or *blind feedback*. In this case, we don't have to involve users since we simply assume that the top k ranked documents are relevant. Let's say we assume the top $k = 10$ documents are relevant. Then, we will use these documents to learn and to improve the query. But how could this help if the top-ranked documents are random? In fact, the top documents are actually similar to relevant documents, even if they are not relevant. Otherwise, how would they have appeared high in the ranked list? So, it's possible to learn some related terms to the query from this set anyway regardless whether the user says that a document is relevant or not.
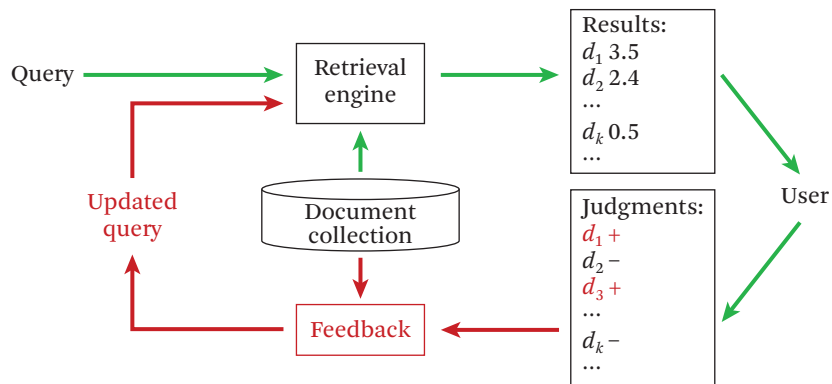
Figure 7.1   **How feedback is part of an information retrieval system.**

You may recall that we talked about using language models to analyze word associations by learning related words to the word *computer* (see Chapter 3). First, we used *computer* to retrieve all the documents that contain that word. That is, imagine the query is *computer*. Then, the results will be those documents that contain *computer*. We take the top k results that match *computer* well and we estimate term probabilities (by counting them) in this set for our topic language model. Lastly, we use the background language model to choose the terms that are frequent in this retrieved set but not frequent in the whole collection. If we contrast these two ideas, what we can find is that we'll learn some related terms to *computer*. These related words can then be added to the original query to expand the query, which helps find documents that don't necessarily match *computer*, but match other words like *program* and *software* that may not have been in the original query.

Unfortunately, pseudo relevance feedback is not completely reliable; we have to arbitrarily set a cutoff and hope that the ranking function is good enough to get at least some useful documents. There is also another feedback method called **implicit feedback**. In this case, we still involve users, but we don't have to explicitly ask them to make judgements. Instead, we are going to observe how the users interact with the search results by observing their clickthroughs. If a user clicked on one document and skipped another, this gives a clue about whether a document is useful or not. We can even assume that we're going to use only the snippet here in a document that is displayed on the search engine results page (the text that's actually seen by the user). We can assume this displayed text is probably relevant or interesting to the user since they clicked on it. This is the idea behind implicit

feedback and we can again use this information to update the query. This is a very important technique used in modern search engines—think about how Google and Bing can collect user activity to improve their search results.

To summarize, we talked about three types of feedback. In relevance feedback, we use explicit relevance judgements, which require some user effort, but this method is the most reliable. We talked about pseudo feedback, where we simply assumed the top k document are relevant without involving the user at all. In this case, we can actually do this automatically for each query before showing the user the final results page. Lastly, we mentioned implicit feedback where we use clickthrough data. While this method does involve users, the user doesn't have to make explicit effort to make judgements on the results.

Next, we will discuss how to apply feedback techniques to both the vector space and query likelihood retrieval models. The future sections do not make any note of how the feedback documents are obtained since no matter how they are obtained, they would be dealt with the same way by each of the following two feedback methods.

7.1 Feedback in the Vector Space Model

This section is about feedback in the vector space retrieval model. As we have discussed, feedback in a TR system is based on learning from previous queries to improve retrieval accuracy in future queries. We will have positive examples, which are the documents that we assume to be relevant to a particular query, and we have negative examples, which are non-relevant to a specific query. The way the system gets these judged documents depends on the particular feedback strategy that is employed (which was discussed in the previous section).

The general method in the vector space model for feedback is to modify our query vector. We want to place the query vector in a better position in the high-dimensional term space, plotting it closer to relevant documents. We might adjust weights of old terms or assign weights to new terms in the query vector. As a result, the query will usually have more terms, which is why this is often called **query expansion**. The most effective method for the vector space model feedback was proposed several decades ago and is called **Rocchio feedback**.

We illustrate this idea in Figure 7.2 by using a two-dimensional display of all the documents in the collection in addition to the query vector q. The query vector is in the center and the $+$ (positive) or $-$ (negative) represent documents. When we have a query vector and use a similarity function to find the most similar documents, we are drawing this dotted circle, denoting the top-ranked documents. Of course, not
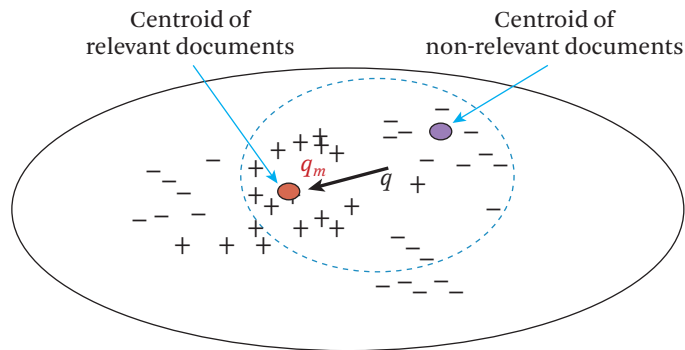
Figure 7.2   Illustration of Rocchio Feedback: adjusting weights in the query vector to move it closer to a cluster of relevant documents.

all the top-ranked documents will be positive, and this is the motivation behind feedback in the first place. Our goal is to move the query vector to some position to improve the retrieval accuracy, shifting the dotted circle of similarity. By looking at this diagram, we see that we should move the query vector so that the dotted circle encompasses more + documents than − documents. This is the basic idea behind Rocchio feedback.

Geometrically, we're talking about moving a vector closer to some vectors and away from other vectors. Algebraically, it means we have the following formula (using the arrow vector notation for clarity):

$$\vec{q}_m = \alpha \cdot \vec{q} + \frac{\beta}{|D_r|} \cdot \sum_{\vec{d}_j \in D_r} \vec{d}_j - \frac{\gamma}{|D_n|} \cdot \sum_{\vec{d}_j \in D_n} \vec{d}_j, \tag{7.1}$$

where $\vec{q}$ is the original query vector that is transformed into $\vec{q}_m$, the modified (i.e., expanded) vector. D_r is the set of relevant feedback documents and D_n is the set of non-relevant feedback documents. Additionally, we have the parameters α, β, and γ which are weights that control the amount of movement of the original vector. In terms of movement, we see that the terms in the original query are boosted by a factor of α, and terms from positive documents are boosted by a factor of β, while terms from negative documents are shrunk by a factor of γ.

Another interpretation of the second term (the sum over positive documents) is the centroid vector of relevant feedback documents while the third term is the centroid vector of the negative feedback documents. In this sense, we shift the original query towards the relevant centroid and away from the negative centroid. Thus, the average over these two terms computes one dimension's weight in the centroid of these vectors.

After we have performed these operations, we will get a new query vector which can be used again to score documents in the index. This new query vector will then reflect the move of the original query vector toward the relevant centroid vector and away from the non-relevant centroid vector.

Let's take a look at a detailed example depicted below. Imagine we have a small vocabulary,

$$V = \{news, about, presidential, campaign, food, text\}$$

and a query

$$\vec{q} = \{1, 1, 1, 1, 0, 0\}.$$

Recall from Chapter 6 that our vocabulary V is a fixed-length term vector. It's not necessary to know what type of weighting scheme this search engine is using, since in Rocchio feedback, we will only be adding and subtracting term weights from the query vector.

Say we are given five feedback documents whose term vectors are denoted as relevant with a $+$ prefix. The negative feedback documents are prefixed with $-$.

		{	news	about	pres.	campaign	food	text	}
$-$	d_1	{	1.5	0.1	0.0	0.0	0.0	0.0	}
$-$	d_2	{	1.5	0.1	0.0	2.0	2.0	0.0	}
$+$	d_3	{	1.5	0.0	3.0	2.0	0.0	0.0	}
$+$	d_4	{	1.5	0.0	4.0	2.0	0.0	0.0	}
$-$	d_5	{	1.5	0.0	0.0	6.0	2.0	0.0	}

For Rocchio feedback, we first compute the centroid of the positive and negative feedback documents. The centroid of the positive documents would have the average of each dimension, and the case is the same for the negative centroid:

		{	news	about	pres.	campaign	food	text	}
$+$	C_r	{	$\frac{1.5+1.5}{2}$	0.0	$\frac{3.0+4.0}{2}$	$\frac{2.0+2.0}{2}$	0.0	0.0	}
$-$	C_n	{	$\frac{1.5+1.5+1.5}{3}$	$\frac{0.1+0.1+0.0}{3}$	0.0	$\frac{0.0+2.0+6.0}{3}$	$\frac{0.0+2.0+2.0}{3}$	0.0	}

Now that we have the two centroids, we modify the original query to create the expanded query $\vec{q}_m$:

$$\vec{q}_m = \alpha \cdot \vec{q} + \beta \cdot C_r - \gamma \cdot C_n$$
$$= \{\alpha + 1.5\beta - 1.5\gamma, \alpha - 0.067\gamma, \alpha + 3.5\beta, \alpha + 2\beta - 2.67\gamma, -1.33\gamma, 0\}.$$

We have the parameter α controlling the original query term weight, which all happened to be one. We have β to control the influence of the relevant centroid vector C_r. Finally, we have γ, which is the non-relevant centroid C_n weight. Shifting the original query vector $\vec{q}$ by these amounts yields our modified query $\vec{q}_m$. We rerun the search with this new query. Due to the movement of the query vector, we should match the relevant documents much better, since we moved $\vec{q}$ closer to them and away from the non-relevant documents—this is precisely what we want from feedback.

If we apply this method in practice we will see one potential problem: we would be performing a somewhat large computation to calculate the centroids and modify all the weights in the new query. Therefore, we often truncate this vector and only retain the terms which contain the highest weights, considering only a small number of words. This is for efficiency. Additionally, negative examples or non-relevant examples tend not to be very useful, especially compared with positive examples. One reason is because negative documents distract the query in all directions, so taking the average doesn't really tell us where exactly it should be moving to. On the other hand, positive documents tend to be clustered together and they are often in a consistent direction with respect to the query. Because of this effect, we sometimes don't use the negative examples or set the parameter γ to be small.

It's also important to avoid over-fitting, which means we have to keep relatively high weight α on the original query terms. We don't want to overly trust a small sample of documents and completely reformulate the query without regard to its original meaning. Those original terms are typed in by the user because the user decided that those terms were important! Thus, we bias the modified vector towards the original query direction. This is especially true for pseudo relevance feedback, since the feedback documents are less trustworthy. Despite these issues, the Rocchio method is usually robust and effective, making it a very popular method for feedback.

7.2 Feedback in Language Models

This section is about feedback for language modeling in the query likelihood model of information retrieval. Recall that we derive the query likelihood ranking function by making various assumptions, such as term independence. As a basic retrieval function, that family of functions worked well. However, if we think about incorporating feedback information, it is not immediately obvious how to modify

query likelihood to perform feedback. Many times, the feedback information is additional information about the query, but since we assumed that the query is generated by assembling words from an ideal document language model, we don't have an easy way to add this additional information.

However, we have a way to generalize the query likelihood function that will allow us to include feedback documents more easily: it's called a **Kullback-Leibler divergence retrieval model**, or *KL-divergence retrieval model* for short. This model actually makes the query likelihood retrieval function much closer to the vector space model. Despite this, the new form of the language model retrieval can still be regarded as a generalization of query likelihood (in that it covers query likelihood without feedback as a special case). Here, the feedback can be achieved through query model estimation or updating. This is very similar to Rocchio feedback which updates the query vector; in this case, we update the query language model instead.

Figure 7.3 shows the difference between our original query likelihood formula and the generalized KL-divergence model. On top, we have the query likelihood retrieval function. The KL-divergence retrieval model generalizes the query term frequency into a probabilistic distribution. This distribution is the only difference, which is able to characterize the user's query in a more general way. This query language model can be estimated in many different ways—including using feedback information. This method is called KL-divergence because this can be interpreted as measuring the divergence (i.e., difference) between two distributions; one is the query model $p(w \mid \hat{\theta}_Q)$ and the other is the document language model from before. We won't go into detail on KL-divergence, but there is a more detailed explanation in appendix C.

$$\text{Query likelihood} \quad f(q, d) = \sum_{\substack{w \in d \\ w \in q}} \boxed{c(w, q)} [\log \frac{p_{\text{seen}}(w|d)}{\alpha_d p(w|C)}] + n \log \alpha_d$$

$$\begin{array}{l}\text{KL-divergence} \\ \text{(cross entropy)}\end{array} \quad f(q, d) = \sum_{w \in d, p(w|\theta_Q) > 0} \boxed{[p(w|\hat{\theta}_Q)]} \log \frac{p_{\text{seen}}(w|d)}{\alpha_d p(w|C)}] + \log \alpha_d$$

$$\text{Query LM} \quad p(w|\hat{\theta}_Q) = \frac{c(w, Q)}{|Q|}$$

Figure 7.3   The KL-divergence retrieval model changes the way we represent the query. This enables feedback information to be incorporated into the query more easily.
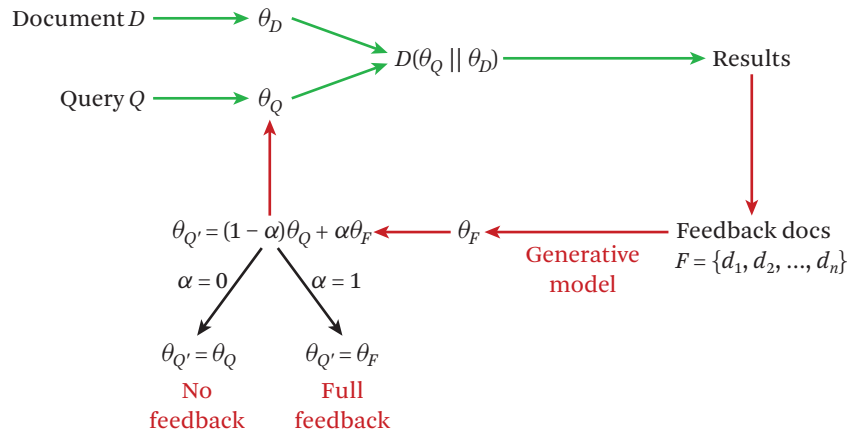
Figure 7.4   **Model-based feedback.**

So, the two formulas look almost identical except that in the generalized formula we have a probability of a word given by a query language model. Still, we add all the words that are in the document and have non-zero probability for the query language model. Again, this becomes a generalization of summing over all the matching query words. We can recover the original query likelihood formula by simply setting the query language model to be the relative frequency of a word in the query, which eliminates the query length term $n = |q|$ which is a constant.

Figure 7.4 shows that we first estimate a document language model, then we estimate a query language model and we compute the KL-divergence, often denoted by $D(\cdot||\cdot)$. We compute a language model from the documents containing the query terms called the feedback language model θ_F. This feedback language model is similar to the positive centroid C_r in Rocchio feedback. This model can be combined with the original query language model using a linear interpolation, which produces an updated model, again just like Rocchio.

We have a parameter $\alpha \in [0, 1]$ that controls the strength of the feedback documents. If $\alpha = 0$, there is no feedback; if $\alpha = 1$, we receive full feedback and ignore the original query. Of course, these extremes are generally not desirable. The main question is how to compute this θ_F.

Now, we'll discuss one of the approaches to estimate θ_F. This approach is based on a generative model shown in Figure 7.5. Let's say we are observing the positive documents, which are collected by users' judgements, the top k documents from a search, clickthrough logs, or some other means. One approach to estimate a language model over these documents is to assume these documents are gen-
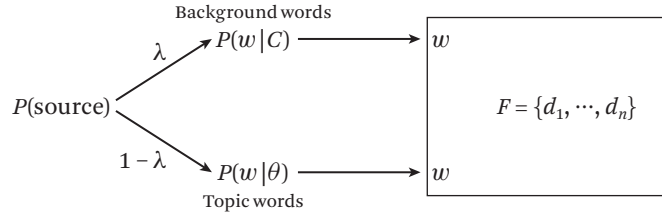
$$\log p(F|\theta) = \sum_i \sum_w c(w; d_i) \log[(1 - \lambda)p(w|\theta) + \lambda p(w|C)]$$

Maximum likelihood $\theta_F = \underset{\theta}{\operatorname{argmax}}\ \log p(F|\theta)$

λ = noise in feedback documents

Figure 7.5   **Mixture model for feedback.**

erated from some ideal feedback language model as we did before; this entails normalizing all the frequency counts from all the feedback documents. But is this distribution good for feedback? What would the top-ranked words in θ_F be?

As depicted in the language model on the right in Figure 7.6, the high-scoring words are actually common words like *the*. This isn't very good for feedback, because we will be adding many such words to our query when we interpolate with our original query language model. Clearly, we need to get rid of these stop words. In fact, we have already seen one way to do that, by using a background language model while learning word associations in Chapter 2. Instead, we're going to talk about another approach which is more principled. What we can do is to assume that those unwanted words are from the background language model. If we use a maximum likelihood estimate, a single model would have been forced to assign high probabilities to a word like *the* because it occurs so frequently. In order to reduce its probability in this model, we have to have another model to explain such a common word. It is appropriate to use the background language model to achieve this goal because this model will assign high probabilities to these common words.

We assume the machine that generated these words would work as follows. Imagine we flip a coin to decide what distribution to use (topic words or background words). With the probability of $\lambda \in [0, 1]$ the coin shows up as heads and then we're going to use the background language model. Once we know we will use the background LM, we can then sample a word from that model. Alternatively, with probability $1 - \lambda$, we decide to use an unknown topic model to generate a word. This is a mixture model because there are two distributions that are mixed together, and we actually don't know when each distribution is used. We can treat this feedback

Query: "airport security"

Mixture model approach

λ = 0.9

w	$P(w|\theta_F)$
security	0.0558
airport	0.0546
beverage	0.0488
alcohol	0.0474
bomb	0.0236
terrorist	0.0217
author	0.0206
license	0.0188
bond	0.0186
counter-terror	0.0173
terror	0.0142
newsnet	0.0129
attack	0.0124
operation	0.0121
headline	0.0121

Web database

Top 10 docs

λ = 0.7

w	$P(w|\theta_F)$
the	0.0405
security	0.0377
airport	0.0342
beverage	0.0305
alcohol	0.0304
to	0.0268
of	0.0241
and	0.0214
author	0.0156
bomb	0.0150
terrorist	0.0137
in	0.0135
license	0.0127
state	0.0127
by	0.0125

Figure 7.6   **Example of query models learned via pseudo-relevance feedback.**

mixture model as a single distribution in that we can still ask it to generate words, and it will still give us a word in a random way (according to the underlying models). Which word will show up depends on both the topic distribution and background distribution. In addition, it would also depend on the mixing parameter λ; if λ is high, it's going to prefer the background distribution. Conversely, if λ is very small, we're going to use only our topic words.

Once we're thinking this way, we can do exactly the same as what we did before by using MLE to adjust this model and set the parameters to best explain the data. The difference, however, is that we are not asking the unknown topic model alone to explain all the words; rather, we're going to ask the whole mixture model to explain the data. As a result, it doesn't have to assign high probabilities to words like *the*, which is exactly what we want. It would then assign high probabilities to other words that are common in the topic distribution but not having high probability in the background distribution. As a result, this topic model must assign high probabilities to the words common in the feedback documents yet not common across the whole collection.

Mathematically, we have to compute the log likelihood of the feedback documents F with another parameter λ, which denotes noise in the feedback documents. We assume it will be fixed to some value. Assuming it's fixed, then

we only have word probabilities θ as parameters, just like in the simplest unigram language model. This gives us the following formula to estimate the feedback language model:

$$\theta_F = \arg\max_{\theta} \log p(F \mid \theta)$$

$$= \arg\max_{\theta} \sum_{d \in F} \sum_{w} c(w, d) \cdot \log\left[(1 - \lambda) \cdot p(w \mid \theta) + \lambda \cdot p(w \mid C)\right] \quad (7.2)$$

We choose this probability distribution θ_F to maximize the log likelihood of the feedback documents under our model. This is the same idea as the maximum likelihood estimator. Here though, the mathematical problem is to solve this optimization problem. We could try all possible θ values and select the one that gives the whole expression the maximum probability. Once we have done that, we obtain this θ_F that can be interpolated with the original query model to do feedback. Of course, in practice it isn't feasible to try all values of θ, so we use the EM algorithm to estimate its parameters [Zhai and Lafferty 2001]. Such a model involving multiple component models combined together is called a mixture model, and we will further discuss such models in more detail in the topic analysis chapter (Chapter 17).

Figure 7.6 shows some examples of the feedback model learned from a web document collection for performing pseudo feedback. We just use the top 10 documents, and we use the mixture model with parameters $\lambda = 0.9$ and $\lambda = 0.7$. The query is *airport security*. We select the top ten documents returned by the search engine for this query and feed them to the mixture model. The words in the two tables are learned using the approach we described. For example, the words *airport* and *security* still show up as high probabilities in each case naturally because they occur frequently in the top-ranked documents. But we also see *beverage*, *alcohol*, *bomb*, and *terrorist*. Clearly, these are relevant to this topic, and if combined with the original query can help us match other documents in the index more accurately.

If we compare the two tables, we see that when λ is set to a smaller value, we'll still see some common words when we don't use the background model often. Remember that λ can "choose" the probability of using the background model to generate to the text. If we don't rely much on the background model, we still have to use the topic model to account for the common words. Setting λ to a very high value uses the background model more often to explain these words and there is no burden on explaining the common words in the feedback documents. As a result, the topic model is very discriminative—it contains all the relevant words without common words.

To summarize, this section discussed feedback in the language model approach; we transform our original query likelihood retrieval function to a more general KL-divergence model. This generalization allows us to use a language model for the query, which can be manipulated to include feedback documents. We described a method for estimating the parameters in this feedback model that discriminates between topic words (relevant to the query) and background words (useless stop words).

In this chapter, we talked about the three major feedback scenarios: relevance feedback, pseudo feedback, and implicit feedback. We talked about how to use Rocchio to do feedback in the vector-space model and how to use query model estimation for feedback in language models. We briefly talked about the mixture model for its estimation, although there are other methods to estimate these parameters that we mention later on in the book.

Bibliographic Notes and Further Reading

An early empirical comparison of various relevance feedback techniques can be found in Salton and Buckley [1990]. Pseudo-relevance feedback has become popular after positive results being observed in TREC experiments (e.g., Buckley 1994, Xu and Croft 1996). A comparison of feedback approaches in language models is available in Lv and Zhai [2009]. The positional relevance model proposed in Lv and Zhai [2010] appears to be one of the most effective methods for estimating a query language model for pseudo feedback. Due to the availability of a large amount of search engine log data, implicit feedback based on users' interaction behavior has become a very important and very effective technique to enable web search engines to improve their accuracy over time as more and more users are using the systems, though the interpretation of user clickthroughs must take position bias into consideration, which is discussed in detail in Joachims et al. [2007]. A bibliography on implicit feedback can be found in Kelly and Teevan [2003]. In the web search era, implicit feedback is often implemented in the form of using feedback features in a ranking function using machine learning, i.e., learning to rank techniques; they are discussed briefly in Chapter 10 of the book. For a more thorough discussion of mining query logs, see the tutorial [Silvestri 2010].

Exercises

7.1. How should you set the Rocchio parameters α, β, and γ depending on what type of feedback you are using? That is, should the parameters be set differently if

you are using pseudo feedback compared to user-supplied relevance judgements? What about implicit feedback through clickthrough data?

7.2. Imagine you are in charge of a large search-engine company. What other strategies could you devise to get relevance judgments from users?

7.3. Say one of your new strategies is to measure the amount of time t a user spends on each search result document. How can you incorporate this t for each document into a feedback measure for a particular query?

7.4. Implement Rocchio pseudo feedback in META.

7.5. Implement mixture model feedback for language models in META. Use whichever method is most convenient to estimate θ_F. Or, compare different estimation methods for θ_F.

7.6. After implementing Rocchio pseudo feedback, index a dataset with relevance judgements. Plot MAP (see Chapter 9) across different values of k. Do you see any trends?

7.7. After implementing mixture model feedback, index a dataset with relevance judgements. Plot MAP (see Chapter 9) across different values of the mixing parameter α. Do you see any trends?

7.8. Design a heuristic to automatically determine the best k for pseudo feedback on a query-by-query basis. You could look at the query itself, the number of matching documents, or the distribution of ranking scores in the original results. Test your heuristic by doing experiments.
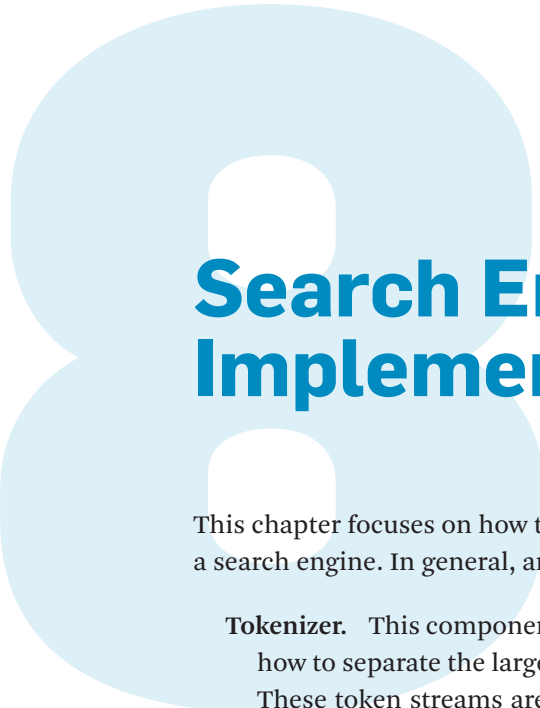
7.9. Design a heuristic to automatically determine the best α for mixture model feedback on a query-by-query basis. You could look at the query itself, the number of matching documents, or the distribution of ranking scores in the original results. Test your heuristic by doing experiments.

7.10. In mixture model feedback, we discussed how to incorporate positive feedback documents via a language model θ_F. Design a formula that also incorporates a set of negative feedback documents. Ensure that your new query language model is still a valid probability distribution.

7.11. In mixture model feedback, we estimated the feedback LM with a probabilistic model that ensures stop words do not affect the reformulated query. Do we need to do anything like this for Rocchio feedback?

7.12. In the feedback methods we discussed in this chapter, we assumed we only had sets of relevant and non-relevant documents. In reality, we actually have two ranked lists of relevant and non-relevant documents. How can we take advantage of these ranked lists for feedback? In other words, how can we treat feedback documents differently depending on how similar they are to the original query? Consider the vector space model, the query likelihood model, or both.

7.13. In a real search system, storing modified query vectors for all observed queries will take up a large amount of space. How could you optimize the amount of space required? What kind of solutions provide a tradeoff between space and query time? How about an online system that benefits the majority of users or the majority of queries?

Search Engine Implementation

This chapter focuses on how to implement an information retrieval (IR) system or a search engine. In general, an IR system consists of four components.

Tokenizer. This component takes in documents as raw strings and determines how to separate the large document string into separate tokens (or features). These token streams are then passed on to the indexer. This is perhaps the most vital part of the system as a whole, since a poor tokenization method will affect all other parts of the indexing, and propagate downstream to the end user.

Indexer. This is the module that processes documents and indexes them with appropriate data structures. An Indexer can be run offline. The main challenges are to index large amounts of documents quickly with a limited amount of memory. Other challenges include supporting addition and deletion of documents.

Scorer/Ranker. This is the module that takes a query and returns a ranked list of documents. Here the challenge is to implement a retrieval model efficiently so that we can score documents efficiently.

Feedback/Learner. This is the module that is responsible for relevance feedback or pseudo feedback. When there is a lot of implicit feedback information such as user clickthroughs available (as in a modern web search engine), this learning module can be fairly sophisticated. It was discussed in detail in the previous chapter, so in this chapter we will just outline how it may be added to an existing system.

For the first three items, there are fairly standard techniques that are essentially used in all current search engines. The techniques for implementing feedback,

however, highly depend on the learning approaches and applications. Despite this, we did discuss some common methods for feedback in the previous chapter.

We will additionally investigate two additional optimizations. These are not required to ensure the correctness of an information retrieval system, but they will enable such a system to be much more efficient in both speed and disk usage.

Compression.   The documents we index could consume hundreds of gigabytes or terabytes. We can simultaneously save disk space and increase disk read efficiency by losslessly compressing the data in our index, which is usually just integers.[1]

Caching.   Even after designing and compressing an efficient data structure for document retrieval storage, the system will still be at the mercy of the hard disk speed. Thus, it is common practice to add a cache between the front-facing API and the document index on disk. The cache will be able to save frequently-accessed term information so the number of slow disk seeks during query-time is reduced.

The following sections in this chapter discuss each of the above components in turn.

8.1   Tokenizer

Document tokenization is the first step in any text mining task. This determines how we represent a document. We saw in the previous chapter that we often represent documents as document vectors, where each index corresponds to a single word. The value stored in the index is then a raw count of the number of occurrences of that word in a particular document.

When running information retrieval scoring functions on these vectors, we usually prefer some alternate representation of term count, such as smoothed term count, or TF-IDF weighting. In real search engine systems, we often leave the term scoring up to the index scorer module. Thus, in tokenization we will simply use the raw count of features (words), since the raw count can be used by the scorer to calculate some weighted term representation. Additionally, calculating something like TF-IDF is more involved than a simple scanning of a single document (since we need to calculate IDF). Furthermore, we'd like our scorer to be able to use different

---

1. As we will discuss, the string terms themselves are almost always represented as term IDs, and most of the processing on "words" is done on integer IDs instead of strings for efficiency.

scoring functions as necessary; storing only TF-IDF weight would then require us to *always* use TF-IDF weighting.

Therefore, a tokenizer's job is to segment the document into countable features or tokens. A document is then represented by how many and what kind of tokens appear in it. The raw counts of these tokens are used by the scorer to formulate the retrieval scoring functions that we discussed in the previous chapter.

The most basic tokenizer we will consider is a *whitespace tokenizer*. This tokenizer simply delimits words by their whitespace. Thus,

`whitespace_tokenizer`(Mr. Quill's book is very very long.)

could result in

$$\{Mr.: 1, \quad Quill's: 1, \quad book: 1, \quad is: 1, \quad very: 2, \quad long.: 1\}.$$

A slightly more advanced unigram words tokenizer could first lowercase the sentence and split the words based on punctuation. There is a special case here where the period after *Mr.* is not split (since it forms a unique word):

$$\{mr.: 1, \quad quill: 1, \quad 's: 1, \quad book: 1, \quad is: 1, \quad very: 2, \quad long: 1, \quad .: 1\}.$$

Of course, we aren't restricted to using a unigram words representation. Look back to the exercises from Chapter 4 to see some different ways in which we can represent text. We could use bigram words, POS-tags, grammatical parse tree features, or any combination. Common words (stop words) could be removed and words could also be reduced to their common stem (stemming). Again, the exercises in Chapter 4 give good examples of these transformations using META. In essence, the indexer and scorer shouldn't care how the term IDs were generated; this is solely the job of the tokenizer.

Another common task of the tokenizer is to assign document IDs. It is much more efficient to refer to documents as unique numbers as opposed to strings such as `/home/jeremy/docs/file473.txt`. It's much faster to do integer comparisons than string comparisons, in addition to integers taking up much less space. The same argument may be made for string terms vs. term IDs. Finally, it will almost always be necessary to map terms to counts or documents to counts. In C++, we could of course use some structure internally such as `std::unordered_map<std::string, uint64_t>`. As you know, using a hash table like this gives amortized $O(1)$ lookup time to find a `uint64_t` corresponding to a particular `std::string`.

However, using term IDs, we can instead write `std::vector<uint64_t>`. This data structure takes up less space, and allows true $O(1)$ access to each `uint64_t` using a term ID integer as the index into the `std::vector`. Thus, for term ID 57, we would look up index 57 in the array.

Using term IDs and the second tokenizer example, we could set *mr.*→ term id 0, *quill*→ term id 1 and so on, then our document vector looks like

$$\{1, 1, 1, 1, 1, 2, 1, 1\}.$$

Of course, a real document vector would be much larger and much sparser—that is, most of the dimensions will have a count of zero.

This process is also called **feature generation**. It defines the building blocks of our document objects and gives us meaningful ways to compare them. Once we define how to conceptualize documents, we can index them, cluster them, and classify them, among many other text mining tasks. As mentioned in the Introduction, tokenization is perhaps the most critical component of our indexer, since all downstream operations depend on its output.

8.2 Indexer

Modern search engines are designed to be able to index data that is much larger than the amount of system memory. For example, a Wikipedia database dump is about 40 GB of uncompressed text. At the time of writing this book, this is much larger than the amount of memory in common personal systems, although it is quite a common dataset for computer science researchers. TREC research datasets may even be as large as several terabytes. This doesn't even take into account real-world production systems such as Google that index the entire Web.

This requires us to design indexing systems that only load portions of the raw corpus in memory at one time. Furthermore, when running queries on our indexed files, we want to ensure that we can return the necessary term statistics fast enough to ensure a usable search engine. Scanning over every document in the corpus to match terms in the query will not be sufficient, even for relatively small corpora.

An **inverted index** is the main data structure used in a search engine. It allows for quick lookup of documents that contain any given term. The relevant data structures include (1) the **lexicon** (a lookup table of term-specific information, such as document frequency and where in the postings file to access the per-document term counts) and (2) the **postings file** (mapping from any term integer ID to a list of document IDs and frequency information of the term in those documents).
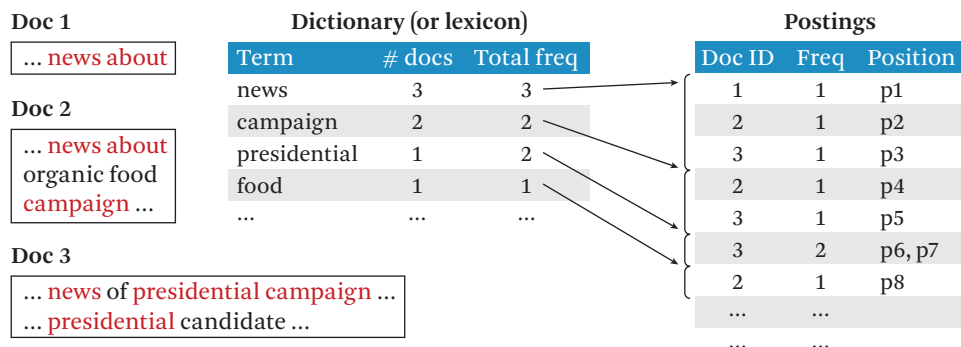
Figure 8.1 Inverted index postings and lexicon files.

In order to support "proximity heuristics" (rewarding matching terms that are together), it is also common to store the position of each term occurrence. Such position information can be used to check whether all the query terms are matched within a certain window of text, e.g., it can be used to check whether a phrase is matched. This information is stored in the postings file since it is document-specific.

Figure 8.1 shows a representation of the lexicon and postings files. The arrows in the image are actually integer offsets that represent bit or byte indices into the postings file.

For example, if we want to score the term *computer*, which is term ID 56, we look up 56 in the lexicon. The information we receive could be:

```
Term ID: 56
Document frequency: 78
Total number of occurrences: 443
Offset into postings file: 8923754
```

Of course, the actual lexicon would just store $56 \rightarrow \{78, 443, 8923754\}$. Since the tokenizer assigned term IDs sequentially, we could represent the lexicon as a large array indexed by term ID. Each element in the large array would store tuples of (document frequency, total count, offset) information. If we seek to position 8,923,754 in the large postings file, we could see something like

```
Term ID: 56
Doc ID: 4, count: 1, position 56
Doc ID: 7, count: 9, position 4, position 89, position...
Doc ID: 24, count: 19, position 1, position 67, position...
```

```
Doc ID: 90, count: 4, position 90, position 93, position...
Doc ID: 141, count: 1, position 100
Doc ID: 144, count: 2, position 34, position 89
.
.
.
```

which is the counts and position information for the 78 documents that term ID 56 appears in. Notice how the doc IDs (and positions) are stored in increasing order; this is a fact we will take advantage of when compressing the postings file. Also make note of the large difference in size of the lexicon and postings file. For each entry in the lexicon, we know we will only store three values per term. In the postings file, we store at *least* three values (doc ID, count, positions) for each document that the term appears in. If the term appears in all documents, we'd have a list of the length of the number of documents in the corpus. This is true for all unique terms. For this reason, we often assume that the lexicon can fit into main memory and the postings file resides on disk, and is seeked into based on pointers from the lexicon.

Indexing is the process of creating these data structures based on a set of tokenized documents. A popular approach for indexing is the following sorting-based approach.

- Scan the raw document stream sequentially. In tokenization, assign each document an ID. Tokenize each document to obtain term IDs, creating new term IDs as needed.

- While scanning documents, collect term counts for each term-document pair and build an inverted index for a subset of documents in memory. When we reach the limit of memory, write the incomplete inverted index into the disk. (It will be the same format as the resulting postings file, just smaller.)

- Continue this process to generate many incomplete inverted indices (called "runs") all written on disk.

- Merge all these runs in a pair-wise manner to produce a single sorted (by term ID) postings file. This algorithm is essentially the `merge` function from mergesort.

- Once the postings file is created, create the lexicon by scanning through the postings file and assigning the offset values for each term ID.

Figure 8.2 shows how documents produce terms originally in document ID order. The terms from multiple documents are then sorted by term ID in small postings chunks that fit in memory before they are flushed to the disk.
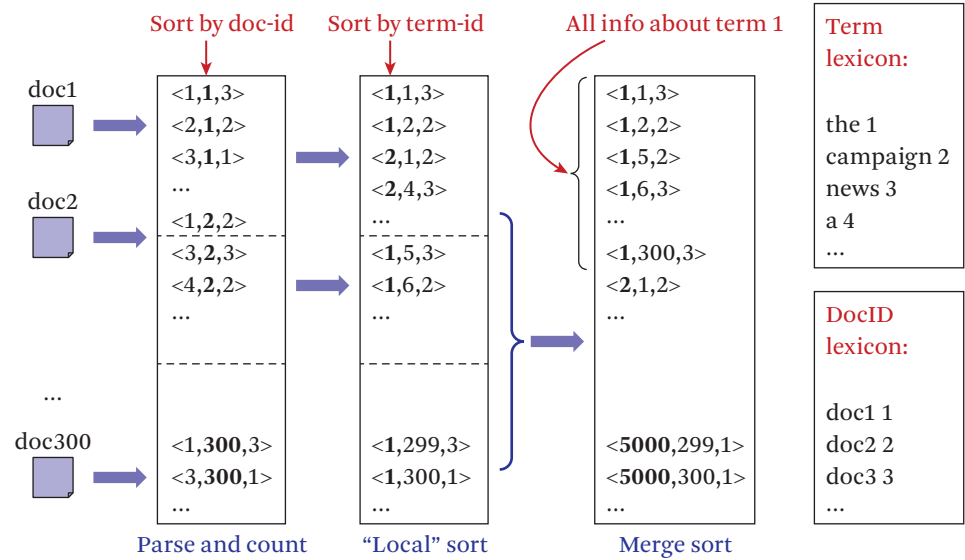
Figure 8.2   Sort-based inversion of inverted index chunks.

A **forward index** may be created in a very similar way to the inverted index. Instead of mapping terms to documents, a forward index maps documents to a list of terms that occur in them. This type of setup is useful when doing other operations aside from search. For example, clustering or classification would need to access an entire document's content at once. Using an inverted index to do this is not efficient at all, since we'd have to scan the entire postings file to find all the terms that occur in a specific document. Thus, we have the forward index structure that records a term vector for each document ID.

In the next section, we'll see how using the *inverted* term-to-document mapping can greatly decrease query scoring time. There are other efficiency aspects that are relevant to the forward index as well, such as compression and caching.

8.3 Scorer

Now that we have our inverted index, how can we use it to efficiently score queries? Imagine for a moment that we don't have an inverted index; we only have the forward index, which maps document IDs to a list of terms that occur in them. To score a query vector, we'd need to iterate through every single entry (i.e., document) in the forward index and run a scoring function on the each (document, query) pair.

Algorithm 8.1   **Term-at-a-time Ranking**

$scores = \{\}$ // score accumulator maps doc IDs to scores
for $w \in q$ **do**
   for d, $count \in Idx.fetch\_docs(w)$ **do**
      $scores[d] = scores[d] + score\_term(count)$
   end for
end for
return top k documents from $scores$

Most likely, many documents do not contain any of the query terms (especially if stop word removal is performed), which means that their ranking score will be zero. Why should we bother scoring these documents anyway? This is exactly how we can benefit from an inverted index: we can only score documents that match at least one query term—that is, *we will only score documents that will have nonzero scores*. We assume (and verify in practice) that scoring only documents containing terms that appear in the query results in much less scoring computation. This leads us to our first scoring algorithm using the inverted index.

8.3.1   Term-at-a-time Ranking

Once an inverted index is built, scoring a query term-by-term can be done efficiently on an inverted index Idx using Algorithm 8.1 with query q.

For each term, fetch the corresponding entries (frequency counts) in the inverted index. Create document score accumulators as needed (variables that hold the accumulated score for each document). Scan the inverted index entries for the current term and for each entry (corresponding to a document containing the term), update its score accumulator based on some term weighting method (the *score_term* function). This could be (for example) Okapi BM25. As we finish processing all the query terms, the score accumulators should have the final scores for all the documents that contain at least one query term. Note that we don't need to create a score accumulator if the document doesn't match any query term.

In reality, the *fetch_docs* function would return some object that contains information about the current term in the document, such as count, background probability, or any other necessary information that the *score_term* function would need to operate.

Once we've iterated through all the query terms, the score accumulators have been finalized. We just need to sort these documents by their accumulated scores

and return (usually) the top k. Again, we save time in this sorting operation by only sorting documents that contained a query term as opposed to sorting every single document in the index, even if its score is zero.

8.3.2 Document-at-a-time Ranking

One disadvantage to term-at-a-time ranking is that the size of the score accumulators *scores* will be the size of the number of documents matching at least one term. While this is a huge improvement over all documents in the index, we can still make this data structure smaller.

Instead of iterating through each document multiple times for each matched query term occurrence, we can instead score an entire document at once. Since most (if not all) searches are top-k searches, we can only keep the top k documents at any one time. This is only possible if we have the complete score for each document in our structure that holds scored documents. Otherwise, as with term-at-a-time scoring, a document may start out with a lower score than another, only to surpass it as more terms are scored.

We can hold the k best completely scored documents with a priority queue. Using the inverted index, we can get a list of document IDs and postings data that need to be scored. As we score a complete document, it is added on the priority queue. We assign high priorities to documents with low scores; this is so that after adding the $(k + 1)^{st}$ document, we can (in $O(\log k)$ time) remove the lowest-score document and only hold onto the top k. Once we've iterated through all the document IDs, we can easily sort the k documents and return them. See Algorithm 8.2.

We can use a similar priority queue approach while extracting the top k documents from the term-at-a-time score accumulators, but we would still need to store all the scores before finding the top k.

8.3.3 Filtering Documents

Another common task is only returning documents that meet a certain criteria. For example, our index may store newspaper articles with dates as metadata. In our top-k search, suppose we want to only return documents that were written within the past year. This is a common **document filtering** problem.

With term-at-a-time ranking, we can ignore documents that are not in the correct date range by not updating their scores in the score accumulator (thus not inserting those document IDs into the structure). In document-at-a-time ranking, we can simply skip the document if it doesn't pass the filter when creating the context for that particular document.

Algorithm 8.2 **Document-at-a-time Ranking**

$context = \{\}$ // maps a document to a list of matching terms
for $w \in q$ **do**
    for d, $count \in Idx.fetch\_docs(w)$ **do**
        $context[d].append(count)$
    end for
end for
$priority\_queue = \{\}$ // low score is treated as high priority
for d, $term\_counts \in context$ **do**
    $score = 0$
    for $count \in term\_counts$ **do**
        $score = score + score\_term(count)$
    end for
    $priority\_queue.push(d, score)$
    if $priority\_queue.size() > k$ **then**
        $priority\_queue.pop()$ // removes lowest score so far
    end if
end for
Return sorted documents from $priority\_queue$

Filters can be as complex as desired, since a filter is essentially just a Boolean function that takes a document and returns whether or not it should be returned in the list of scored documents. The filtering function can then be an optional parameter to the scoring function which has access to the document metadata store (usually a database) and a forward index (in order to filter documents that contain certain terms).

8.3.4 Index Sharding

Index sharding is the concept of keeping more than one inverted index for a particular search engine. This is easily achieved by stopping the postings chunk merging process when the number of chunks is equal to the number of desired shards. All the same data is stored as one final chunk, but it's just broken down into several pieces.

But why would we want multiple inverted index chunks? Consider when we have the number of shards equal to the number of threads (or cluster nodes) in our search system. You can probably imagine an algorithm where each thread searches for matching terms in its respective shard, and the final search results

are then merged together. This type of algorithm design—distributing the work and then merging the results—is a very common paradigm called **Map Reduce**. We will discuss its generality and many other applications in future chapters.

8.4 Feedback Implementation

Chapter 7 discussed feedback in a standard information retrieval system. We saw two implementations of feedback: the vector space Rocchio feedback and the query likelihood mixture model for feedback.

Both can be implemented with the inverted index and document metadata we've described in the previous sections. For Rocchio feedback, we can use the forward index to obtain the vectors of both the query and feedback documents, running the Rocchio algorithm on that set of vectors. The mixture model feedback method requires a language model to be learned over the feedback documents; again, this can be achieved efficiently by using the term counts from the forward index. The only other information needed is the corpus background probabilities for each term, which can be stored in the term lexicon.

With this information, it is now possible to create an online (or "in-memory") pseudo-feedback method. Recall that pseudo-feedback looks at the top k returned documents from search and assumes they are relevant. The following process could be used to enable online feedback.

- Run the user's original query.
- Use the top k documents and the forward index to either modify the query vector (Rocchio) or estimate a language model and interpolate with the feedback model (query likelihood).
- Rerun the search with the modified query and return the new results to the user.

There are both advantages and disadvantages to this simple feedback model. For one, it requires very little memory and disk storage to implement since each modified query is "forgotten" as soon as the new results are returned. Thus, we don't need to create any additional storage structures for the index.

The downside is that all the processing is done at query time, which could be quite computationally expensive, especially when using a search engine with many users. The completely opposite tradeoff is to store every modified query in a database, and look up its expanded form, running the search function only once. Of course, this is infeasible as the number of queries would quickly make the database explode in size, not to mention that adding more documents to the index would

invalidate the stored query vectors (since the new documents might also match the original query).

In practice, we can have some compromise between these two extremes, e.g., only storing the very frequently expanded queries, or using query similarity to search for a similar query that has been saved. The caching techniques discussed in a later section are also applicable to feedback methods, so consider how to adopt them from caching terms to caching expanded queries.

Of course, this only touches on the pseudo-feedback method. There is also clickthrough data, which can be stored in a database, and relevance judgements, which can be stored the same way. Once we know which documents we'd like to include in the chosen feedback method, all implementations are the same since they deal with a list of feedback documents.

8.5 Compression

Another technical component in a retrieval system is integer compression, which is applied to compress the very large postings file. A compressed index is not only smaller, but also faster when it's loaded into main memory. The general idea of compressing integers (and compression in general) is to exploit the non-uniform distribution of values. Intuitively, we will assign a short code to values that are frequent at the price of using longer codes for rare values. The optimal compression rate is related to the entropy of the random variable taking the values that we consider—skewed distributions would have lower entropy and are thus easier to compress.

It is important that all of our compression methods need to support random access decoding; that is, we could like to seek to a particular position in the postings file and start decompressing without having to decompress all the previous data.

Because inverted index entries are stored sequentially, we may exploit this fact to compress document IDs (and position information) based on their gaps. The document IDs would otherwise be distributed relatively uniformly, but the distribution of their gaps would be skewed since when a term is frequent, its inverted list would have many document IDs leading to many small gaps. Consider the following example of a list of document IDs:

$$\{23, 25, 34, 35, 39, 43, 49, 51, 57, 59, \ldots\}.$$

Instead of storing these exact numbers, we can store the offsets between them; this creates more smaller numbers, which are easier to compress since they take

up less space and are more frequent:

$$\{23, 2, 9, 1, 4, 4, 6, 2, 6, 2, \ldots\}.$$

To get the actual document ID values, simply add the offset to the previous value. So the first ID is 23 and the second is $23 + 2 = 25$. The third is $25 + 9 = 34$, and so on.

In this section, we will discuss the following types of compression, which may or may not operate on gap-encoded values:

- unary encoding (bitwise);
- γ-encoding (bitwise);
- δ-encoding (bitwise);
- vByte (block); and
- frame of reference (block).

8.5.1   Bitwise compression

With bitwise compression, instead of writing out strings representing numbers (like "1624"), or fixed byte-width chunks (like a 4-byte integer as "00000658"), we are writing raw binary numbers. When the representation ends, the next number begins. There is no fixed width, or length, of the number representations. Using bitwise compression means performing some bit operations for every bit that is encoded in order to "build" the compressed integer back into its original form.

Unary. Unary encoding is the simplest method. To write the integer k, we simply write $k - 1$ zeros followed by a one. The one acts as a delimiter and lets us know when to stop reading:

$$1 \rightarrow 1$$
$$2 \rightarrow 01$$
$$3 \rightarrow 001$$
$$4 \rightarrow 0001$$
$$5 \rightarrow 00001$$
$$19 \rightarrow 0000000000000000001$$

Note that we can't encode the number zero—this is true of most other methods as well. An example of a unary-encoded sequence is

$$0001001000100000101000100001 = 4, 3, 4, 8, 2, 4, 5.$$

As long as the lexicon has a pointer to the beginning of a compressed integer, we can easily support random access decoding. We also have the property that small numbers take less space, while larger numbers take up more space. The next two compression methods are built on the concept of unary encoding.

Gamma. To encode a number with γ-encoding, first simply write the number in binary. Let k be the number of bits in your binary string. Then, prepend $k-1$ zeros to the binary number:

$$1 \rightarrow 1$$
$$2 \rightarrow 010$$
$$3 \rightarrow 011$$
$$4 \rightarrow 00100$$
$$5 \rightarrow 00101$$
$$19 \rightarrow 000010011$$
$$47 \rightarrow 00000101111$$

To decode, read and count k zeros until you hit a one. Read the one and additional k bits in binary. Note that all γ codes will have an odd number of bits.

Delta. In short, δ-encoding is γ-encoding a number and then γ-encoding the unary prefix (including the one):

$$1 \rightarrow 1 \rightarrow 1$$
$$2 \rightarrow 010 \rightarrow 0100$$
$$3 \rightarrow 011 \rightarrow 0101$$
$$4 \rightarrow 00100 \rightarrow 01100$$
$$5 \rightarrow 00101 \rightarrow 01101$$
$$19 \rightarrow 000010011 \rightarrow 001010011$$
$$47 \rightarrow 00000101111 \rightarrow 0011001111$$

To decode, decode the γ code at your start position to get an integer k. Write a one, and then read the next $k+1$ bits in binary (including the one you wrote). As you can see, the δ compression at first starts off to have more bits than the γ encoding, but eventually becomes more efficient as the numbers get larger. It probably depends on the particular dataset (the distribution of integers) as to which compression method would be better in terms of **compression ratio**. A compression ratio is simply the uncompressed size divided by the compressed size. Thus, a compression ratio of 3 is better (in that the compressed files are smaller) than a compression ratio of 2.

8.5.2 Block compression

While bitwise encoding can achieve a very high compression ratio due to its fine-grained distribution model, its downside is the amount of processing that is re-

quired to encode and decode. Every single bit needs to be read in order to read one integer. Block compression attempts to alleviate this issue by reading bytes at a time instead of bits. In block compression schemes, only one bitwise operation per byte is usually required as opposed to at *least* one operation per bit in the previous three schemes (e.g., count how many bit operations are required to δ-encode the integer 47).

Block compression seeks to reduce the number of CPU instructions required in decoding at the expense of using more storage. The two block compression methods we will investigate deal mainly with bytes instead of bits.

vByte stands for **variable byte** encoding. It uses the lower seven bits of a byte to store a binary number and the most significant bit as a flag. The flag signals whether the decoder should keep reading the binary number. The parentheses below are added for emphasis.

$$1 \to (0)0000001$$
$$2 \to (0)0000010$$
$$19 \to (0)0010011$$
$$47 \to (0)0101111$$
$$127 \to (0)1111111$$
$$128 \to (1)0000000(0)0000001$$
$$194 \to (1)1000010(0)0000001$$
$$678 \to (1)0100110(0)0000101$$
$$20,000 \to (1)0100000(1)0011100(0)0000001$$

The decoder works by keeping a sum (which starts at zero) and adding each byte into the sum as it is processed. Notice how the bytes are "chained" together backwards. For every "link" we follow, we left shift the byte to add by $7 \times k$, where k is the number of bytes read so far. Therefore, the sum we have to decode the integer 20,000 is

$$(0100000 << 0) + (0011100 << 7) + (0000001 << 14)$$

which is the same as

$$0100000_b$$
$$+ 00111000000000_b$$
$$+ 000000100000000000000_b$$
$$= 00000100111000100000_b$$
$$= 20{,}000_d$$

In this method, the "blocks" that we are encoding are bytes.

Frame of reference encoding takes a block size k and encodes k integers at a time, so a block in this method is actually a sequence of numbers. Take the following block of size $k = 8$ as an example:

$$\{45, 47, 51, 59, 63, 64, 70, 72\}.$$

We transform this block by subtracting the minimum value in the list from each element:

$$\{0, 2, 6, 14, 18, 19, 25, 27\}, min = 45.$$

Up to this point, this is similar to the gap encoding discussed previously. However, instead of encoding each value with a bitwise compression such as γ- or δ-encoding, we will simply use binary with the smallest number of bits possible. Since the maximum number in this chunk is 27, we will store each of the integers in 5 bits:

$$\{00000, 00010, 00110, 01110, 10010, 10011, 11001, 11011\}, min = 45, bytes = 5.$$

This might look like

$$45, 5, 0000000010001100111010010100111100111011,$$

where 45 and 5 could be stored however is convenient (e.g., fixed binary length). This method also reduces the number of bitwise operations, since we read chunks of 5 bits and add them to the base value 45 to recreate the sequence. Another nice side effect is that we know the minimum value and maximum *possible* value stored in this chunk; therefore, if we are looking for a particular integer (say for a document ID), we know we can skip this chunk and not decompress it if $ID < 45$ or $ID > 45 + 2^5$.

8.6 Caching

While we designed our inverted index structure to be very efficient, we still have the issue of disk latency. For this reason, it is very common for a real-world search engine to also employ some sort of caching structure stored in memory for postings data objects. In this section, we'll overview two different caching strategies.

The basic idea of a cache is to make frequently accessed objects fast to acquire. To accomplish this, we attempt to keep the frequently accessed objects in memory so we don't need to seek to the disk. In our case, the objects we access are postings lists. Due to Zipf's law [Zipf 1949], we expect that a relatively small number of
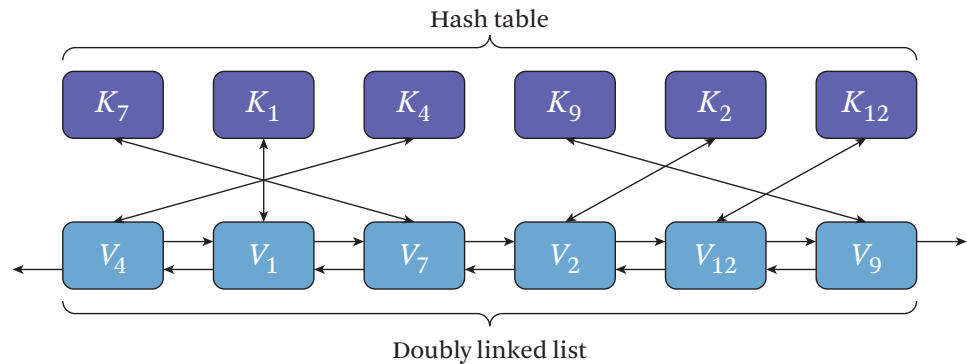
Figure 8.3   **An implementation of a Least-Recently Used (LRU) cache.**

postings lists (keyed by term ID) will be accessed the majority of the time. But how do we know which terms to store? Even if we knew which terms are most queried, how do we set a cutoff for memory consumption?

The two cache designs we describe address both these issues. That is: (1) the most-frequently used items are fast to access, and (2) we can set a maximum size for our cache so it doesn't get too large.

8.6.1   LRU cache

We first consider the least recently used (LRU) cache as displayed in Figure 8.3. The LRU algorithm is as follows, assuming we want to retrieve the postings list for term ID x.

- First, search the cache for term ID x.

- If it exists in the cache, return the postings list corresponding to x.

- If it doesn't exist in the cache, retrieve it from disk and insert it into the cache. If this causes the cache to exceed its maximum size, remove the least-recently used postings list.

We want searching the cache to be fast, so we use a hash table to store term IDs as keys and postings lists as values. This enables $O(1)$ amortized insert, find, and delete operations. The interesting part of the LRU cache is how to determine the access order of the objects. To do this, we link together the values as a doubly linked list. Once an element is inserted or accessed, it is moved to the head (front) of the

Primary map

Secondary map
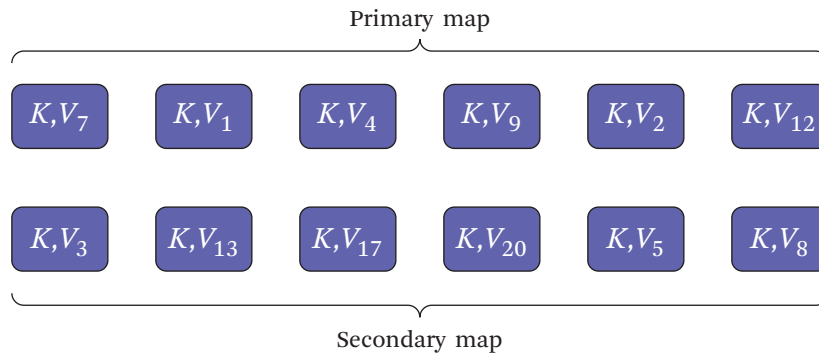
Figure 8.4 **A full Double Barrel Least-Recently Used (DBLRU) cache.**

list in $O(1)$ time. When we need to remove the LRU item, we look at the element at the tail (end) of the linked list and delete it, also in constant time.

8.6.2 DBLRU cache

The double barrel LRU cache was originally used in the popular Lucene search engine.[2] It is a simplified approximation of the LRU cache. Figure 8.4 shows a DBLRU cache with size six.

The DBLRU cache is just two hash tables named `primary` and `secondary`. The algorithm is as follows, assuming we want to retrieve the postings list for term ID x.

- First, search `primary` for term ID x; if it exists, return it.

- If it's not in `primary`, search `secondary`.

- If it's in `secondary`, delete it. Insert it in `primary` and return it. If this causes `primary` to reach the maximum size, clear the entire contents of `secondary`. Then, swap the two hash tables.

- If it's not in `secondary`, retrieve it from disk and insert it into `secondary`.

This cache has a rough hierarchy of usage: `primary` contains elements that are more frequently accessed than `secondary`. So when the cache fills, the `secondary` table is emptied to free memory. While the temporal accuracy of the DBLRU cache is not as precise as the LRU cache, it is a much simpler setup, which translates to faster access times. As usual, there is a tradeoff between the speed and accuracy of these two caches.

_____

2. https://lucene.apache.org/

Bibliographic Notes and Further Reading

A classic reference book for the implementation of search engines is the book Managing Gigabytes [Witten et al. 1999]. Other books on information retrieval, such as *Introduction to Information Retrieval* [Manning et al. 2008], and *Search Engines: Information Retrieval in Practice* [Croft et al. 2009], and *Information Retrieval: Implementing and Evaluating Search Engines* [Büttcher et al. 2010] also have an excellent coverage of implementations of search engines.

Exercises

8.1. Why is using an inverted index to score documents preferred over a more naive solution?

8.2. For the following values, explain whether we can *efficiently* get the value from a default (standard) inverted index postings file. By efficiently, we mean with one lookup—not scanning the entire index.

Symbol	Value
$|d|$	total number of terms in a given doc
$|d|_u$	number of unique terms in a given doc
df	number of documents a given term appears in
$c(w, C)$	number of times w occurs in the corpus
$c(w, d)$	count of w in d
$p(w, d)$	probability of w occurring in d

8.3. For values in the previous question that were *not* able to be extracted efficiently from the postings file, either explain what auxiliary structures need to be searched or why it isn't possible to retrieve the value efficiently.

8.4. Imagine that each document is tagged with some sentiment score (either positive or negative). Outline a method on how you could score documents if you only wanted to return documents of a certain sentiment on a per-query basis.

8.5. Compress the following sequence of integers using each of the compression methods discussed:

$$\{34, 36, 39, 42, 47, 48, 49, 51\}$$

8.6. According to Zipf's law, which of the following strategies is more effective for reducing the size of an inverted index? (1) Remove all words that appear 10 times or less or (2) remove the top 10 most frequent words.

8.7. What type of integer compression is supported in MᴇTA? Write one of the implementations discussed in this chapter that does not exist in the toolkit.

8.8. Which type of scoring (document- vs. term-at-a-time) does MᴇTA use? Hint: see the file `meta/src/index/ranker/ranker.cpp`.

8.9. Implement the *other* type of query scoring and compare its runtime to the existing method in MᴇTA.

8.10. MᴇTA has an implementation of a DBLRU cache. Experiment with the cache size parameter and plot the retrieval speed against different cache sizes while holding any other variables constant.

8.11. MᴇTA does not currently have an implementation of an LRU cache as discussed in this chapter. Can you write one and compare its performance to the DBLRU cache in terms of memory usage and speed?

Search Engine Evaluation

This chapter focuses on the evaluation of text retrieval (TR) systems. In the previous chapter, we talked about a number of different TR methods and ranking functions, but how do we know which one works the best? In order to answer this question, we have to compare them, and that means we'll have to *evaluate* these retrieval methods. This is the main focus of this chapter. We start out with the methodology behind evaluation. Then, we compare the retrieval of sets with the retrieval of ranked lists as well as judgements with multiple levels of relevance. We end with practical issues in evaluation, followed by exercises.

9.1 Introduction

Let's think about why we have to do evaluation. There are two main reasons; the first is that we have to use evaluation to figure out which retrieval method works the best. This is very important for advancing our knowledge, otherwise we wouldn't know whether a new idea works better than an old idea. Previously in this book (Chapter 6), we discussed the problem of text retrieval and compared it with database retrieval. Search engine evaluation must rely on users, so this becomes a very challenging problem. Because of this, we must determine how we can get users involved and draw a fair comparison of different methods.

The second reason to perform evaluation is to assess the actual utility of an overall TR system (as opposed to specific methods). Imagine you're building your own applications; you would be interested in knowing how well your search engine works for your users. In this case, measures must reflect the utility to the actual users in the real application as opposed to measures on each individual retrieval result. Typically, this has been done via user studies—where human users interact with the corpus via the system. In the case of comparing different methods, the measures we use all need to be correlated with the utility to the users. The measures only need to be good enough to determine which method works better. This is usually done by using a test collection, which is a main idea that we'll be talking about

in this chapter. This has been very important for comparing different algorithms and for improving search engines systems in general.

9.1.1   What to Measure?

There are many aspects of a search engine we can measure—here are the three major ones.

Effectiveness or accuracy.   How accurate are the search results? In this case, we're measuring a system's capability of ranking relevant documents on top of non-relevant ones.

Efficiency.   How quickly can a user get the results? How large are the computing resources that are needed to answer a query? In this case, we need to measure the space and time overhead of the system.

Usability.   How useful is the system for real user tasks? Here, interfaces and many other things are also important and we typically have to do user studies.

In this book, we're going to talk mainly about the effectiveness and accuracy measures because the efficiency and usability dimensions are not unique to search engines (they are needed for evaluating other software systems). There is also very good coverage of such material in other books, so we suggest the reader consult Harman [2011] for further reading in this area. Additional readings are Sanderson [2010] and Kelly [2009], which cover user studies and A-B testing (concepts that are discussed later in this chapter).

9.1.2   Cranfield Evaluation Methodology

The **Cranfield evaluation methodology** was developed in the 1960s and is a strategy for laboratory testing of system components. It's actually a methodology that has been very useful not only for search engine evaluation, but also for evaluating virtually all kinds of empirical tasks. For example, in image processing or other fields where the problem is empirically defined we typically would need to use a method such as this.

The basic idea of this approach is to build reusable test collections and define measures using these collections. Once such a test collection is built, it can be used again and again to test different algorithms or ideas. Using these test collections, we will define measures that allow us to quantify the performance of a system or algorithm. The assembled test collection of documents is similar to a real document collection in a search application. We can also have a sample set of queries or topics that simulate the user's information need. Then, we need to have **relevance judg-**

ments. These are judgments of which documents should be returned for which queries. Ideally, they have to be made by users who formulated the queries because those are the people that know exactly what the documents (search results) would be used for. Finally, we have to have measures to quantify how well a system's result matches the ideal ranked list that would be constructed based on users' relevance judgements.

This methodology is very useful for evaluating retrieval algorithms because the test set can be reused many times. It also provides a fair comparison for all the methods, since the evaluation is exactly the same for each one. That is, we have the same criteria, same corpus, and same relevance judgements to compare the different algorithms. This allows us to compare a new algorithm with an old algorithm that was invented many years ago by using the same approach.

In Figure 9.1, we illustrate how the Cranfield evaluation methodology works. As mentioned, we need a set of queries that are shown here. We have Q_1, Q_2, and so on. We also need the document collection, D_1, D_2, . . ., and on the far right side of the figure, we have the relevance judgments which are plus or minus annotations on each document specifying whether it is relevant (plus) or not relevant (minus). Essentially, these are binary judgments of documents with respect to a specific query since there are only two levels of relevance. For example, D_1 and D_2 are judged as being relevant to Q_1. D_3 is judged as non-relevant with respect to Q_1.
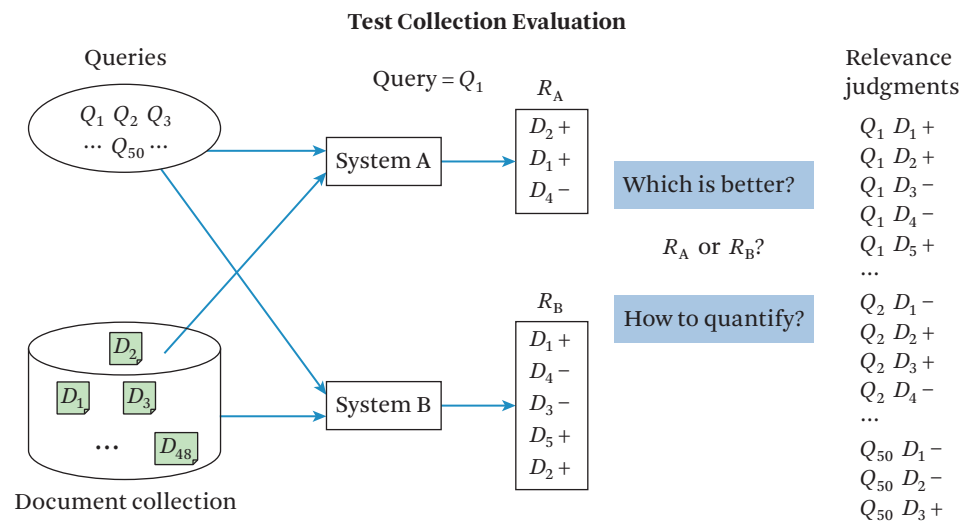
Figure 9.1   **Illustration of Cranfield evaluation methodology.**

These Q_i judgements are created by users that interact with each system. Once we have these judgements, we can compare two or more systems. Each query is run on each system, and we investigate the documents that each system returns.

Let's say the query is Q_1. In the figure we have R_A as ranked results from system A and R_B as the ranked results from system B. Thus, R_A is system A's approximation of relevant documents and R_B is system B's approximation. Let's take a look at these results—which is better? As a user, which one would you like? There are some differences and there are some documents that are returned by both systems. But if you look at the results you will feel that maybe system A is better in the sense that we don't have that many documents returned, and among the three documents returned two of them are relevant. That's good; system A is precise. On the other hand, we can also say system B is better because it returned *more* relevant documents; it returned three instead of two. So which one is better and how do we quantify this? This question highly depends on a user's task, and it depends on the individual users as well! For some users, maybe system A is better if the user is not interested in getting all the relevant documents so he or she doesn't have to read too much. On the other hand, imagine a user might need to have as many relevant documents as possible, for example, in writing a literature survey. You might be in the second category, and then you might find that system B is better. In either case, we'll have to also define measures that would quantify the information need of a user. We will need to define multiple measures because users have different perspectives when looking at the results.

9.2 Evaluation of Set Retrieval

In this section, we examine the basic measures for evaluation of text retrieval systems. We discuss how to design these basic measures and how to quantitatively compare two systems. Although the systems return a ranked list of documents, the evaluation metrics discussed in this section deal with *sets* of returned documents; that is, the order of the returned results is not taken into account. These measures and their intuition are used to design other more sophisticated methods, but are also quite valuable on their own.

9.2.1 Precision and Recall

Let's return to Figure 9.1. Which set of results is better—system A's or system B's? We can now discuss how to actually quantify their performance. Suppose we have a total of ten relevant documents in the corpus for the current query, Q_1. Of course, the relevance judgements shown on the right did not include all the ten.

We have only seen three relevant documents there, but we can imagine there are other documents judged for this query. Intuitively we thought that system A is better because it did not have much noise. In particular we have seen, out of three results, two are relevant. On the other hand, in system B we have five results and only three of them are relevant. Based on this, it looks like system A is more *accurate*. This can be captured by the measure of **precision**, where we simply compute to what extent all the retrieval results are relevant. 100% precision would mean all the retrieved documents are relevant. Thus in this case, system A has a precision of $\frac{2}{3} = 0.6\overline{6}$. System B has $\frac{3}{5} = 0.60$. This shows that system A is better according to precision.

But we also mentioned that system B might be preferred by some other users who wish to retrieve as many relevant documents as possible. So, in that case we have to compare the number of total relevant documents to the number that is actually retrieved. This is captured in another measure called **recall**, which measures the completeness of coverage of relevant documents in your retrieval result. We assume that there are ten relevant documents in the collection. Here we've got two of them from system A, so the recall is $\frac{2}{10} = 0.20$. System B has $\frac{3}{10} = 0.30$. Therefore, system B is better according to recall.

These two measures are the very basic foundation for evaluating search engines. They are very important because they are also widely used in many other evaluation problems. For example, if you look at the applications of machine learning you tend to see precision and recall numbers being reported for all kinds of tasks. Now, let's define these two measures more precisely and how these measures are used to evaluate a set of retrieved documents. That means we are considering that approximation of a set of relevant documents. We can distinguish the results in four cases, depending on the situation of a document, as shown in Figure 9.2.

A document is either retrieved or not retrieved since we're talking about the *set* of results. The document can be also relevant or not relevant, depending on whether the user thinks this is a useful document. We now have counts of documents in each of the four categories. We can have a represent the number of documents that are retrieved and relevant, b for documents that are not retrieved but relevant, c for documents that are retrieved but not relevant, and d for documents that are both not retrieved and not relevant. With this table, we have defined precision as the ratio of the relevant retrieved documents a to the total number of retrieved documents a and c: $\frac{a}{a+c}$. In this case, the denominator is all the retrieved documents. Recall is defined by dividing a by the sum of a and b, where $a + b$ is the total number of relevant documents. Precision and recall are focused on looking at a, the number of retrieved relevant documents. The two measures differ based on the denominator of the formula.

Action

		Retrieved	Not retrieved
Doc	Relevant	*a*	*b*
	Not relevant	*c*	*d*

$$\text{Precision} = \frac{a}{a+c} \qquad \text{Ideal results: precision} = \text{recall} = 1.0$$

$$\text{Recall} = \frac{a}{a+b} \qquad \begin{array}{l}\text{In reality, high recall tends to be} \\ \text{associated with low precision}\end{array}$$

Set can be defined by a cutoff (e.g., precision @ 10 docs)

Figure 9.2   **Basic measures: precision and recall.**

So what would be an ideal result? If precision and recall are both 1.0, that means all the results that we returned are relevant, and we didn't miss any relevant documents; there's no single non-relevant document returned. In reality, however, high recall tends to be associated with low precision; as you go down the list to try to get as many relevant documents as possible, you tend to include many non-relevant documents, which decreases the precision. We often are interested in the precision up to ten documents for web search. This means we look at the top ten results and see how many documents among them are actually relevant. This is a very meaningful measure, because it tells us how many relevant documents a user can expect to see on the first page of search results (where there are typically ten results).

In the next section, we'll see how to combine precision and recall into one score that captures both measures.

9.2.2   The F Measure: Combining Precision and Recall

There tends to be a tradeoff between precision and recall, so it is natural to combine them. One metric that is often used is called the F_β measure, displayed in Figure 9.3. In the case where $\beta = 1$, it's a harmonic mean of precision and recall.

Considering the parameter β and after some simplification, we can see the F measure may be written in the form on the right-hand side of the figure. Often, β set to one, which indicates an equal preference towards precision and recall. In the case where $\beta = 1$, we have a special case of the F measure, often called F_1. This is a popular measure that is often used as a combined precision and recall score. If β is not equal to one, it controls the emphasis between precision and recall. It's easy

$$F_\beta = \frac{1}{\frac{\beta^2}{\beta^2+1}\frac{1}{R}\frac{1}{\beta^2+1}\frac{1}{P}} = \frac{(\beta^2+1)P*R}{\beta^2 P + R}$$

$$F_1 = \frac{2PR}{P+R}$$

where P = precision, R = recall, β = parameter (often set to 1)

Figure 9.3   The F measure.

to see that if you have a large precision or large recall then the F_1 measure would be high. But what's interesting is how the tradeoff between precision and recall is captured in the F_1 score.

In order to understand the formulation, we can first ask the natural question: *Why not combine them using a simple arithmetic mean?* That would be likely the most natural way of combining them. Why is this not as good as F_1, i.e., what's the problem with an arithmetic mean?

The arithmetic mean tends to be dominated by large values. That means if you have a very high P or a very high R, then you really don't care about whether the other value is low since the whole sum would be high. This is not the desirable effect because one can easily have a perfect recall by returning all the documents! Then we have a perfect recall and a low precision. This will still give a relatively high average. Such search results are clearly not very useful for users even though the average using this formula would be relatively high. In contrast, the F_1 score will reward a case where precision and recall are roughly similar. So, it would penalize a case with an extremely high result for only one of them. This means F_1 encodes a different tradeoff between them than a simple arithmetic mean. This example shows a very important methodology: when we try to solve a problem, you might naturally think of one solution (e.g., the arithmetic mean), but it's important not to settle on this solution; rather, think whether there are other ways to approach it. Once you have multiple ideas, it's important to analyze their differences and then think about which one makes more sense in a real scenario.

To summarize, we talked about precision, which addresses the question: are the retrieval results all relevant? We also talked about recall, which addresses the question: have all the relevant documents been retrieved? These two are the two basic measures in information retrieval evaluation. They are used for many other tasks as well. We talked about F measure as a way to combine precision and recall. We also talked about the tradeoff between precision and recall, and it turns out to depend on the users' search tasks and preferences.

9.3 Evaluation of a Ranked List

In the previous section, we only considered whether a relevant document appeared in the results or not—a binary measure. In this section, we will see how we can take each document's position into account when assigning an evaluation score.

We saw that precision and recall are the two basic ways to quantitatively measure the performance of a search result. But, as we talked about in depth in Chapter 5, the text retrieval problem is a *ranking* problem, not a classification one. Thus, we need to evaluate the quality of a ranked list as opposed to whether a relevant document was returned anywhere in the results.

How can we use precision and recall to evaluate a ranked list? Naturally, we will have to look at precision and recall at different cutoffs since a ranked list of relevant documents is determined by where the user stops browsing. If we assume the user sequentially browses the list of results, the user would stop at some point. That point would determine the size of the set. Therefore, that's the most important cutoff that we have to consider when we compute the precision-recall.

Without knowing where exactly the user would stop, we have to consider all the possible positions where they might stop. A **precision-recall** curve does exactly this, as illustrated in Figure 9.4

What if the user stops at the first document? What's the precision-recall at this point? Since D_1 is relevant, the precision is one out of one since we have one

Evaluating Ranking: Precision–Recall (PR) Curve

Total number of relevant documents in collection: 10

	Precision	Recall
D_1 +	1/1	1/10
D_2 +	2/2	2/10
D_3 −	2/3	2/10
D_4 −		
D_5 +	3/5	3/10
D_6 −		
D_7 −		
D_8 +	4/8	4/10
D_9 −		
D_{10} −	?	10/10
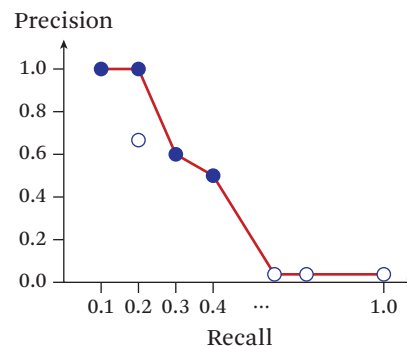
Figure 9.4   Computing a precision-recall curve.

document and it is relevant. What about the recall? Note that we are assuming that there are ten relevant documents for this query in the collection so it's one out of ten.

What if the user stops at the second position? The precision is the same since both D_1 and D_2 are relevant: 100%, or two out of two. The recall is two out of ten, or 20%. If the user stops at the third position, we have an interesting case because we don't have any additional relevant documents, so the recall does not change. However, the precision is lower because we have two out of three relevant documents. The recall won't change until we see another relevant document. In this case, that point is at D_5. There, the recall has increased to three out of ten and the precision is three out of five. As you can see, if we keep doing this, we can also get to D_8 and have a precision of four out of eight, because there are eight documents and four of them are relevant. There, the recall is four out of ten.

When can we get a recall of five out of ten? In this list, we don't have it. For convenience, we often assume that the precision is zero in a situation like this. This is a pessimistic assumption since the actual precision would be higher, but we make this assumption in order to have an easy way to compute another measure called *average precision*, that we will discuss soon.

Note that we've made some assumptions that are clearly not accurate. But, this is okay for the relative comparison of two text retrieval methods. As long as the deviation is not biased toward any particular retrieval method, the measure is acceptable since we can still accurately tell which method works better. This is the most important point to keep in mind: when you compare different algorithms, the key is to avoid any bias toward a particular method. As long as you can avoid that, it's perfectly fine to do a transformation of these measures that preserves the order.

Since we can get a lot of precision-recall numbers at different positions, we can plot a curve; this is what's shown on the right side of Figure 9.4. On the *x*-axis are the recall values. On the *y*-axis are the precision values. We plot precision-recall numbers so that we display at what recall we can obtain a certain precision. Furthermore, we can link these points to form a curve. As you see in the figure, we assumed all the precision values at the high-level recalls are zero. Although the real curves will not be exactly like this, it doesn't matter that much for comparing two methods whether we get the exact precision values here or not.

In Figure 9.5, we compare two systems by plotting their PR-curves on the same graph. System A is shown in red and system B is shown in blue. Which one is better? On the left, system A is clearly better since for the same level of recall, the precision value by system A is better than system B. In general, the higher the curve is, the
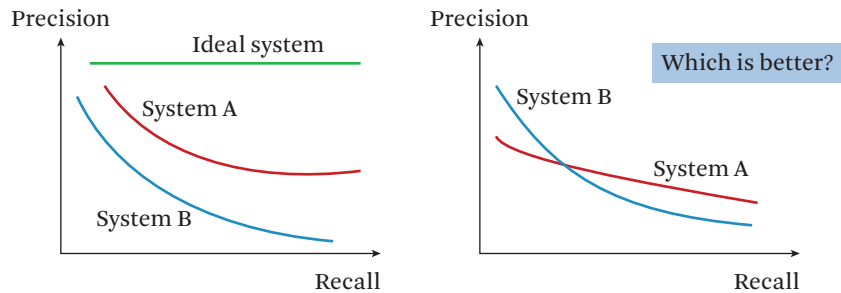
Figure 9.5 **Comparison of two PR curves.** (Courtesy of Marti Hearst)

better. The problem is that we might see a case like the right graph—this actually happens quite often where the two curves cross each other. In this case, which one is better? This is a real problem that you might actually have to face. Suppose you build a search engine and you have an old algorithm that's shown here in blue as system B. Then, you have come up with a new idea and test it with results shown in red as system A.

The question is, is your new method better than the old method? Or, more practically, do you have to replace the algorithm that you're already using in your search engine with another new algorithm? If you make the replacement, the search engine would behave like system A here, whereas if you don't do that, it will be like system B. Now, some users might like system A, while other users might like system B. So what's the difference here? Well, the difference is just that—in the low level of recall, system B is better, but in the high recall region, system A is better. That means it depends on whether the user cares about high recall, or low recall with high precision. Imagine someone is just going to check out what's happening today and wants to find out something relevant in the news. Which system is better for that task? In this case, system B is better because the user is unlikely to examine many results, i.e., the user doesn't care about high recall, only the first few results being useful. On the other hand, if a user wants to determine whether an idea has been thought of before, they will want to emphasize high recall so that they see as many relevant documents as possible and don't miss the chance to find the idea. Therefore, those users would favor system A.

But this brings us back to the original question: *which one is better?* Again, this actually depends on the users, or more precisely, the users' task. You may not necessarily be able to come up with one number that would accurately depict the performance. You have to look at the overall picture. Despite this, it can be

beneficial to have one number to compare the systems so that we can easily make a lot of different system comparisons; we need a number to summarize the range of precision-recall values. One way is to look at the area underneath the curve—the average precision. Basically, we're going to take a look at every different recall point and consider the precision.

The precisions we add up correspond to retrieving the first relevant document, the second, and so on. In the example in the figure, we missed many relevant documents so in all of these cases we assume that they have zero precision. Finally, we take the average and divide it by ten, which is the total number of relevant documents in the collection. Note that we're *not* dividing this sum by four (which is the number of retrieved relevant documents). Dividing by four is a common mistake; this favors a system that would retrieve very few documents, making the denominator very small. In the correct formula, the denominator is ten, the total number of relevant documents. This will allow us to compute the area under the PR-curve, combining recall and precision.

Mathematically, we can define **average precision** on a ranked list L where $|L| = n$ as

$$\text{avp}(L) = \frac{1}{|Rel|} \sum_{i=1}^{n} p(i), \tag{9.1}$$

where $p(i)$ denotes the precision at rank i of the documents in L, and Rel is the set of all relevant documents in the collection. If D_i is not relevant, we would ignore the contribution from this rank by setting $p(i) = 0$. If D_i is relevant, to obtain $p(i)$ we divide the number of relevant documents we've seen so far by the current position in the list (which is i). If the first relevant document is at the second rank, then $p(2) = \frac{1}{2}$. If the third relevant document is at the seventh rank, then $p(7) = \frac{3}{7}$. Let's use this formula to calculate the average precision of the documents returned in Figure 9.4. Figure 9.6 shows the calculation.

This measure is sensitive to a small change in position of a relevant document. If we move the third or fourth relevant document up, it would increase the averages. Conversely, if we move any relevant document down, then it would decrease. Therefore, this is a good measure because it's sensitive to the ranking of each individual relevant document. It can distinguish small differences between two ranked lists, and that's exactly what we want.

In contrast, if we look at the *precision* at ten documents it's easy to see that it's four out of ten. That precision is very meaningful because it tells us what a user would see from their perspective. But, if we use this measure to compare two or more systems, it wouldn't be as effective since precision alone is not sensitive

i	Rel	$p(i)$
1	+	$\frac{1}{1} = 1.0$
2	+	$\frac{2}{2} = 1.0$
3	−	0.0
4	−	0.0
5	+	$\frac{3}{5} = 0.6$
6	−	0.0
7	−	0.0
8	+	$\frac{4}{8} = 0.5$
$\vdots$	−	0.0
sum		3.1
avp		$\frac{3.1}{10} = 0.31$

Figure 9.6   Calculating average precision for a ranked list of results.

to where these four relevant documents are ranked in the list. If they are moved around the top ten spots, the precision at ten remains the same. In contrast, *average precision* is a much better measure since subtle differences in rank affect the overall score.

9.3.1   Evaluating Ranked Lists from Multiple Queries

Average precision is computed for just one query. Generally, though, we experiment with many different queries in order to capture the variance across them. For example, one system may perform very well with one query on which another system happens to perform poorly; using only this query would not give an accurate assessment of each systems' capability. Using more queries then requires the researcher to take an average of the average precision over all these queries. Naturally, we can simply calculate an arithmetic mean. In fact, this would give us what's called **mean average precision** (MAP). In this case, we take arithmetic mean of all the average precisions over several queries or topics. Let $\mathcal{L} = L_1, L_2, \ldots, L_m$ be the ranked lists returned from running m different queries. Then we have

$$MAP(\mathcal{L}) = \frac{1}{m} \sum_{i=1}^{m} \text{avp}(\mathcal{L}_i). \tag{9.2}$$

Recall our discussion about the F_1 score. In this situation, is an arithmetic mean of average precisions acceptable? We concluded before that the arithmetic mean of precision and recall was not as good as the harmonic mean. Here, we have a similar situation: we should think about the alternative ways of aggregating the average precisions. Another way is the geometric mean; using the geometric mean to consolidate the average precisions is called **geometric mean average precision**, or gMAP for short. We define it below mathematically as

$$\text{gMAP}(\mathcal{L}) = \left(\prod_{i=1}^{m} \text{avp}(\mathcal{L}_i) \right)^{\frac{1}{m}}, \tag{9.3}$$

or in log space as

$$\text{gMAP}(\mathcal{L}) = \exp \left\{ \frac{1}{m} \sum_{i=1}^{m} \ln \text{avp}(\mathcal{L}_i) \right\}. \tag{9.4}$$

Imagine you are again testing a new algorithm. You've tested multiple topics (queries) and have the average precision for each topic. You wish to consider the overall performance, but which strategy would you use? Can you think of scenarios where using one of them would make a difference? That is, is there a situation where one measure would give different rankings of the two methods? Similar to our argument about F_1, we realize in the arithmetic mean the sum is dominated by large values. Here, a large value means that the query is relatively easy. On the other hand, gMAP tends to be affected more by *low* values—those are the queries that don't have good performance (the average precision is low). If you wish to improve the search engine for those difficult queries, then gMAP would be preferred. If you just want to improve over all kinds of queries, then perhaps MAP would be preferred. So again, the answer depends on your users' tasks and preferences. Which measure is most likely going to represent your users' needs?

As a special case of the mean average precision, we can also think about the case where there is precisely one relevant document in the entire collection. This actually happens quite often, for example, in what's called a *known item search*, where you know a target page such as Amazon or Facebook. Or in another application such as question answering, there is only one answer. In this scenario, if you rank the answers, then your goal is to rank that one particular answer on top. In this case, the average precision will boil down to the **reciprocal rank**. That is, $\frac{1}{r}$ where r is the position (rank) of the single relevant document. If that document is ranked on the very top, then the reciprocal rank would be $\frac{1}{1} = 1$. If it's ranked at the second

position, then it's $\frac{1}{2}$ and so on. This means we can also take an average of all the reciprocal ranks over a set of topics, which gives us the **mean reciprocal rank** (MRR). It's a very popular measure for known item search or any problem where you have just one relevant item.

We can see this r is quite meaningful; it indicates how much effort a user would have to make in order to find that one relevant document. If it's ranked on the top it's low effort; if it's ranked at 100 then you actually have to (presumably) sift through 100 documents in order to find it. Thus, r is also a meaningful measure and the reciprocal rank will take the reciprocal of r instead of using it directly.

The usual question also applies here: Why not just simply use r? If you were to design a ratio to measure the performance of a system where there is only one relevant item, you might have thought about using r directly as the measure. After all, that measures the user's effort, right? But, think about if you take an average of this over a large number of topics. Again, it would make a difference. For one single topic, using r or using $\frac{1}{r}$ wouldn't make any difference. A larger r corresponds to a small $\frac{1}{r}$. The difference appears when there are many topics. Just like MAP, this sum will be dominated by large values of r. So what are those values? Those are basically large values that indicate lower ranked results. That means the relevant items rank very low down on the list. The average would then be dominated by the relevant documents that are ranked in the lower portion of the list. From a user's perspective we care more about the highly ranked documents, so by taking this transformation by using reciprocal rank we emphasize more on the difference on the top. Think about the difference between rank one and rank two and the difference between rank 100 and 1000 using each method. Is one more preferable than the other?

In summary, we showed that the precision-recall curve can characterize the overall accuracy of a ranked list. We emphasized that the actual utility of a ranked list depends on how many top ranked results a user would examine; some users will examine more than others. Average precision is a standard measure for comparing two ranking methods; it combines precision and recall while being sensitive to the rank of every relevant document. We concluded this section with three methods to summarize multiple average precision values: MAP, gMAP, and MRR.

9.4 Evaluation with Multi-level Judgements

In this section, we will explain how to evaluate text retrieval systems when there are multiple levels of relevance judgments. So far we discussed about binary judgements—that means a documents is judged as being relevant or non-relevant.

	Gain	Cumulative gain	Discounted cumulative gain
D_1	3	3	3
D_2	2	$3 + 2$	$3 + 2/\log 2$
D_3	1	$3 + 2 + 1$	$3 + 2/\log 2 + 1/\log 3$
D_4	1	$3 + 2 + 1 + 1$	$\cdots$
D_5	3	$\cdots$	
D_6	1		
D_7	1		
D_8	2		
D_9	1		
D_{10}	1		

$$\text{Normalized DCG} = \frac{\text{DCG@10}}{\text{IdealDCG@10}}$$

$$\text{DCG@10} = 3 + 2/\log 2 + 1/\log 3 + \cdots + 1/\log 10$$

$$\text{IdealDCG@10} = 3 + 3/\log 2 + 3/\log 3 + \cdots + 3/\log 9 + 2/\log 10$$

Relevance level: $r = 1$ (non-relevant), 2 (marginally relevant), 3 (very relevant)

Assume: there are 9 documents rated "3" in total in the collection

Figure 9.7   **Computation of NDCG.**

Earlier we made the point that relevance is a matter of degree. We often are able to distinguish very highly relevant documents from documents that are still useful, but with a lower relevance.

In Figure 9.7, we show an example of three relevance levels: level three for highly relevant, two for marginally relevant, and one for non-relevant. How do we evaluate a new system using these judgements? We can't use average precision since it only operates on binary relevance values; if we treat level two and three as only one level, then we lose the information gained from comparing these two categories. MAP, gMAP, and MRR depend on average precision, so we can't use them either.

Let's look at the top relevant results when using these judgments. We imagine the user would mostly care about the top ten results. We call these multi-level judgements "gains," since they roughly correspond to how much information a user *gains* when viewing a document. Looking at the first document, the user can gain three points; looking at the non-relevant documents, the user would only gain one point. This gain usually matches the utility of a document from a user's perspective. If we assume the user stops at ten documents we can cumulatively sum the information gain from traversing the list of returned documents. Let r_i be the gain of result i, and let i range from one to n, where we set n to ten in our example. We then have the **cumulative gain** (CG) as

$$CG(L) = \sum_{i=1}^{n} r_i. \tag{9.5}$$

If the user looks at more documents, the cumulative gain is more. This is at the cost of spending more time to examine the list. Thus, cumulative gain gives us some idea about how much total gain the user would have if the user examines all these documents.

There is one deficiency which is not considering the rank or position of each document. Looking at the CG sum of the top four documents, we know there is only one highly relevant document, one marginally relevant document, two non-relevant documents; we don't know where they are ranked in the list. Ideally, we want those with gains of three to be ranked on the top. But how can we capture that intuition? The second three is not as good as the first three at the top. That means the contribution of gain from different documents has to be weighted by their position. The document at position one doesn't need to be discounted because you can assume that the user always sees this document, but the second one will be discounted a little bit because there's a small possibility that the user wouldn't notice it. We divide this gain by a weight based on the position in order to capture this position-based penalty. The **discounted cumulative gain** does exactly this:

$$DCG(L) = r_1 + \sum_{i=2}^{n} \frac{r_i}{\log_2 i}. \tag{9.6}$$

Each document's gain is discounted by dividing by a logarithm of its position in the list. Thus, a lowly ranked document would not contribute as much gain as a highly ranked document. That means if, for example, you switch the position of D_5 and D_2, then the overall DCG score would increase since D_5's relevance score of three is discounted less by being close to the top.

At this point, we have a discounted cumulative gain for measuring the utility of a ranked list with multiple levels of judgment. We still need to do a little bit more in order to make this measure comparable across different queries. The idea here is **normalized discounted cumulative gain** (NDCG):

$$NDCG(L) = \frac{DCG(L)}{IDCG}. \tag{9.7}$$

It is simply DCG normalized by the ideal DCG (IDCG) for a particular query. The IDCG is the DCG of an ideal ranked list with the most relevant documents at the top, sorted in decreasing order of relevance. For example, imagine that we have nine documents in the whole collection rated three. Then, our ideal ranked list would have put all these nine documents on the very top. All this would be followed by a two, because that's the best we could do after we have run out of threes. Then, we can compute the DCG for this ideal ranked list. This becomes the denominator

for NDCG in order to normalize our own DCG in the range [0, 1]. Essentially, we compare the actual DCG with the best result you can possibly get for this query. This doesn't affect the relative comparison of systems for just one topic because this ideal DCG is the same for all the systems. The difference is when we have multiple topics—if we don't do normalization, different topics will have different scales of DCG. For a query like this one, we have nine highly relevant documents, but of course that will not always be the case.

Thus, NDCG is used for measuring relevance based on much more than one relevance level. In a more general way, this is basically a measure that can be applied through any ranked task with a large range of judgments. Furthermore, the scale of the judgments can be dependant on the application at hand. The main idea of this measure is to summarize the total utility of the top k documents; you always choose a cutoff, and then you measure the total utility. It discounts the contribution from lowly ranked documents, and finally, it performs normalization to ensure comparability across queries.

9.5 Practical Issues in Evaluation

In order to create a test collection, we have to create a set of queries, a set of documents, and a set of relevance judgments. It turns out that each requirement has its own challenges.

First, the documents and queries must be representative. They must represent real queries and real documents that users interact with. We also have to use many queries and many documents in order to avoid biased conclusions. In order to evaluate a high-recall retrieval task, we must ensure there exist many relevant documents for each query. If a query has only one relevant document in the collection, then it's not very informative to compare different methods using such a query because there is not much room to see a difference.

In terms of relevance judgements, the challenge is to ensure complete judgements of all the documents for all the queries while simultaneously minimizing human effort. Because we have to use human effort to label these documents, it's a very labor-intensive task. As a result, it's usually impossible to actually label all of the documents for all the queries, especially considering a data set like the Web.

It's also challenging to correlate the evaluation measures with the perceived utility of users. We have to consider carefully what the users care about and then design measures to capture their preferences.

With a certain probability, we can mathematically quantify whether the evaluation scores of two systems are indeed different. The way we do this is with a

	Experiment I			Experiment II	
Query	System A	System B	Query	System A	System B
1	0.20	0.40	1	0.02	0.76
2	0.21	0.41	2	0.39	0.07
3	0.22	0.42	3	0.16	0.37
4	0.19	0.39	5	0.58	0.21
5	0.17	0.37	6	0.04	0.02
6	0.20	0.40	6	0.09	0.91
7	0.21	0.41	7	0.12	0.46
Average	0.20	0.40	Average	0.20	0.40

Figure 9.8   **Statistical significance: two sets of experiments with an identical MAP.** (Courtesy of Douglas W. Oard and Philip Resnik)

statistical significance test. The significance test gives us an idea as to how likely a difference in evaluation scores is due to random chance. This is the reason why we have to use a lot of queries; the more data points we have, the more confident we can be in our measure.

Figure 9.8 displays some sample average precision results from system A and system B in two different experiments. As you can see in the bottom of the figure, we have the MAP for each system in each experiment. They happen to be identical in experiment one and two. Yet if you look at the exact average precisions for different queries, you will realize that in one case you might feel that you can trust the conclusion here given by the average. In the other case, you might not feel as confident. Based on only the MAP score, we can easily say that system B is better. After all, it's 0.4, which is twice as much as 0.2. Clearly, that's better performance. But if you look at these two experiments and look at the detailed results, you will see that we'll be more confident to say that in experiment 1 that system B is in fact better since the average precisions are *consistently* better than system A's. In experiment 2, we're not sure that system B is better since the scores fluctuate so wildly.

How can we quantitatively answer this question? This is why we need to do a statistical significance test. The idea behind these tests is to assess the variance in average precision scores (or any other score) across these different queries. If there's a big variance, that means that the results could fluctuate according to different queries, which makes the result unreliable.

Query	System A	System B	Sign Test	Wilcoxon
1	0.02	0.76	+	+0.74
2	0.39	0.07	−	−0.32
3	0.16	0.37	+	+0.21
4	0.58	0.21	−	−0.37
5	0.04	0.02	−	−0.02
6	0.09	0.91	+	+0.82
7	0.12	0.46	+	+0.34
Average	0.20	0.40	$p = 1.0$	$p = 0.9375$

Figure 9.9 **Statistical significance tests.** (Courtesy Douglas W. Oard and Philip Resnik)

So let's look at these results again in the second case. In Figure 9.9, we show two different ways to compare them. One is a **sign test**. If system B is better than system A, then we have a plus sign. If system A is better, we have a minus sign. Using this, we have four cases where system B is better and three cases where system A is better. Intuitively, these results appear random. If you flip seven coins, using plus to denote heads and minus to denote tails, then these could easily be the results of just randomly flipping the seven coins. The fact that the average is larger doesn't tell us anything! This intuition can be quantified by the concept of a p value. A p value is the probability that this result is in fact from random fluctuation. In this case, the probability is one; it means it *surely* is a random fluctuation.

There are many different significance tests that we can use to quantify the likelihood that the observed difference in the results is simply due to random fluctuations. A particularly interesting test is the **Wilcoxon signed-rank test**. It's a nonparametric test, and we not only look at the signs, but also consider the magnitude of the difference in scores. Another is the (parametric) t-test where a normal distribution is assumed. In any event, we would draw a similar conclusion in our example case: the outcome is very likely to be random. For further study on this and other statistical significance tests, we suggest the reader start with Smucker et al. [2007].

To illustrate the concept of p-values, consider the distribution in Figure 9.10. This is a normal distribution, with a mean of zero in the center. Say we started with the assumption that there's no difference between the two systems. But, we assume that because of random fluctuations depending on the queries we might observe a difference; thus, the actual difference might be on the left side or right side. This
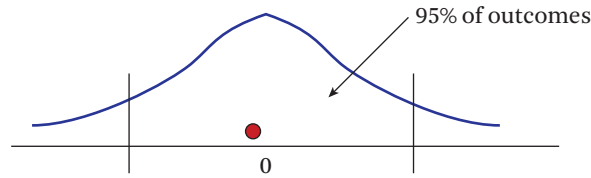
Figure 9.10   **The null hypothesis.** (Courtesy of Douglas Oard and Philip Resnik)

curve shows the probability that we would observe values that are deviating from zero here when we subtract system A's MAP from system B's MAP (or even vice versa). Based on the picture, we see that if a difference is observed at the dot shown in the figure, then the chance is very high that this is in fact a random observation. We can define the region of likely observation due to random fluctuation; we usually use the value 95% of all outcomes. Inside this interval the observed values are from random fluctuation with 95% chance. If you observe a value in the tails on the side, then the difference is unlikely from random fluctuation (only 5% likely). This 95% value determines where the lines are drawn on the x axis. If we are only confident in believing a 1% chance is due to random fluctuations, then the vertical lines are redrawn farther from the mean; determining the exact x values where the lines are drawn depends on the specific significance test used.

The takeaway message here is that we need to use many queries to avoid jumping to an incorrect conclusion that one system is better than another. There are many different ways of doing this statistical significance test, which is essentially determining where to place the boundary lines between random chance and an actual difference in systems.

Now, let's discuss the problem of making relevance judgements. As mentioned earlier, it's very hard to judge all the documents completely unless it is a very small data set. The question is, if we can't afford judging all the documents in the collection, which subset should we judge? The solution here is **pooling**. This is a strategy that has been used in many cases to solve this problem. First, choose a diverse set of ranking methods; these are different types of retrieval systems. We hope these methods can help us nominate likely relevant documents. The goal is to pick out the relevant documents so the users can make judgements on them. That way, we would have each system return the top k documents according to its ranking function. The k value can vary between systems, but the point is to ask them to suggest the most likely relevant documents. We then simply combine all these top k sets to form a pool of documents for human assessors to judge. Of course, there will be many duplicated documents since many systems might have retrieved

the same documents. There are also unique documents that are only returned by one system, so the idea of having a diverse set of result ranking methods is to ensure the pool is broad. We can include as many possible random documents as possible. Then, the human assessors would make complete judgements on this data set, or pool. The remaining unjudged documents are assumed to be non-relevant and the human annotators do not need to spend time and effort manually judging them. If the pool is large enough, this assumption is perfectly fine. That means if your system participates in contributing to the pool then it's unlikely that it will be penalized since the top-ranked documents have all been judged. However, this is problematic for evaluating a new system that may not have contributed to the pool, since the documents it returns may not have been judged and are assumed to be non-relevant.

What we haven't covered are some other evaluation strategies such as A-B testing; this is where an evaluating system would mix the results of two methods randomly, showing the mix of results to users. Of course, the users don't see which result is from which method, so the users would judge those results or click on those documents in a search engine application. In this case, then, the system can keep track of the clicked documents, and see if one method has contributed more to the clicked documents. If the user tends to click on one of the results from one method, then that method may be better. A-B testing can also be used to compare two different retrieval interfaces.

Text retrieval evaluation is extremely important since the task is empirically defined. If we don't rely on users, there's no way to tell whether one method works better. If we have an inappropriate experiment design, we might misguide our research or applications, drawing the wrong conclusions. The main strategy is the Cranfield evaluation methodology for all kinds of empirical evaluation tasks (not just for search engines). MAP and NDCG are the two main measures that you should definitely know about since you will see them often in research papers. Finally, retrieving up to ten documents (or some small number) is easier to interpret from a user's perspective since this is the number of documents they would likely see in a real application.

Bibliographic Notes and Further Reading

Evaluation has always been an important research problem in information retrieval, and in empirical AI problems in general. The Cranfield evaluation methodology was established in 1960s by early pioneers of information retrieval researchers; important early papers on the topic can be found in Sparck Jones and

Willett [1997]. The book *Information Retrieval Evaluation* by Harman [2011] is an excellent comprehensive introduction to this topic particularly in providing a historical view of the development of the IR evaluation methodology and initiatives of evaluation such as TREC. Sanderson's book on test collection evaluation [Sanderson 2010] is another very useful survey of research work on evaluation. The book on interactive IR evaluation by Kelly is yet another excellent introduction to interactive IR evaluation via user studies [Kelly 2009].

Exercises

9.1. In reality, high recall tends to be associated with low precision. Why?

9.2. What is the range of values (minimum and maximum scores) for the following measures?

  (a) Precision

  (b) Recall

  (c) F_1 score

  (d) Average precision

  (e) MAP

  (f) gMAP

  (g) MRR

  (h) DCG

  (i) NDCG

9.3. Assume there are 16 total relevant documents in a collection. Consider the following result, where plus indicates relevant and minus indicates non-relevant:

$$\{+, +, -, +, +, -, -, +, -, -\}$$

Calculate the following evaluation measures on the ranked list.

  (a) Precision

  (b) Recall

  (c) F_1 score

  (d) Average precision

9.4. Consider how recall and precision interact with each other.

  (a) Can precision at five documents (P@5) ever be lower than P@10 for the same ranked list?

(b) Can recall at five documents (R@5) ever be lower than R@10 for the same ranked list?

(c) Assume there are 100 relevant documents. Can R@5 be higher than P@5 for the same ranked list?

(d) Assume there is only one relevant document. Can R@5 be higher than P@5 for the same ranked list?

9.5. When using the NDCG evaluation metric, researchers sometimes consider NDCG@k. This is the NDCG score of the top k documents returned by the search engine. When calculating the ideal DCG, should we consider only k results or should we consider all relevant documents for the query?

9.6. The breakeven point precision is the precision at the cutoff in the ranked list where precision and recall are equal. Can this value be used to compare two or more retrieval systems? How does this measure compare to other single-point measures such as F_1 and average precision?

9.7. A researcher wishes to show that his method is better than an existing method. He used a statistical significance test and found that with 95% confidence, the results were random (i.e., his method was not arguably better). If he changes the confidence level to 90%, he can show that his method is better than the baseline using the same significance test. Is there anything wrong with this?

9.8. How does stemming words affect retrieval performance? Design an experiment to find an answer. What is your dataset? What other information do you need in order to perform this test? How can you quantify the results?

9.9. Use META to perform the stemming experiment described above. A corpus with relevance judgements can be found on the META site.

9.10. Find the best unigram words tokenization method for retrieval. For example, consider lowercasing, keeping only alpha characters, stemming, stopword removal, or a combination of these. Quantify your results by using IR evaluation techniques.

9.11. Find a publicly available dataset and create your own queries and relevance judgements for it using the META format (described on the site).

9.12. Modify META's `index::ir_eval` class to add support for mean reciprocal rank (MRR) evaluation.

Web Search

In this chapter, we discuss one of the most important applications of text retrieval: web search engines. Although many information retrieval algorithms had been developed before the web was born, it created the best opportunity to apply those algorithms to a major application problem that everyone cares about. Naturally, there had to be some further extensions of the classical search algorithms to fully address new challenges encountered in web search.

First, this is a scalability challenge. How can we handle the size of the web and ensure completeness of coverage of all its information (be it textual or not)? How can we serve many users quickly by answering all their queries? Before the web was born, the scale of search was relatively small, usually focused on libraries, so these questions were not serious.

The second problem is that there is much low quality information known as spam. *Search engine optimization* is the attempt to heighten a particular page's rank by taking advantage of how pages are scored, e.g., adding many words that are not necessarily relevant to the actual content or creating many fake links to a particular page to make it seem more popular than it really is. Many different approaches have been designed to detect and prevent such spamming practices [Spirin and Han 2012].

The third challenge is the dynamic nature of the web. New pages are constantly created and updated very quickly. This makes it harder to keep the index fresh with the most recent content. These are just some of the challenges that we have to solve in order to build a high quality web search engine. Despite these challenges, there are also some interesting opportunities that we can leverage to improve search results. For example, we can imagine that using links between pages can improve scoring.

The algorithms that we talked about such as the vector space model are general—they can be applied to any search application. On the other hand, they also don't take advantage of special characteristics of pages or documents in specific applications such as web search. Due to these challenges and opportunities, there are new

techniques that have been developed specifically for web search. One such technique is parallel indexing and searching. This addresses the issue of scalability; in particular, Google's MapReduce framework is very influential.

There are also techniques that have been developed for addressing the spam problem. We'll have to prevent those spam pages from being ranked high. There are also techniques to achieve robust ranking in the light of search engine optimizers. We're going to use a wide variety of signals to rank pages so that it's not easy to spam the search engine with one particular trick.

The third line of techniques is link analysis; these are techniques that can allow us to improve search results by leveraging extra information about the networked nature of the web. Of course, we will use multiple features for ranking—not just link analysis. We can also exploit all kinds of features like the layout of web pages or anchor text that describes a link to another page.

The first component of a web search engine is the *crawler*. This is a program that downloads web page content that we wish to search. The second component is the indexer, which will take these downloaded pages and create an inverted index. The third component is retrieval, which answers a user's query by talking to the user's browser. The browser will show the search results and allow the user to interact with the web. These interactions with the user allow opportunities for feedback (discussed in Chapter 7) and evaluation (discussed in Chapter 9). In the next section, we will discuss crawling. We've already described all indexing steps except crawling in detail in Chapter 8.

After our crawling discussion, we move onto the particular challenges of web indexing. Then, we discuss how we can take advantage of links between pages in link analysis. The last technique we discuss is *learning to rank*, which is a way to combine many different features for ranking.

10.1   Web Crawling

The crawler is also called a *spider* or a software robot that crawls (traverses, parses, and downloads) pages on the web. Building a toy crawler is relatively easy because you just need to start with a set of seed pages, fetch pages from the web, and parse these pages' new links. We then add them to a queue and then explore those page's links in a breadth-first search until we are satisfied.

Building a real crawler is quite tricky and there are some complicated issues that we inevitably deal with. One issue is robustness: What if the server doesn't respond or returns unparseable garbage? What if there's a trap that generates dynamically generated pages that attract your crawler to keep crawling the same

site in circles? Yet another issue is that we don't want to overload one particular server with too many crawling requests. Those may cause the site to experience a denial of service; some sites will also block IP addresses that they believe to be crawling them or creating too many requests. In a similar vein, a crawler should respect the robot exclusion protocol. A file called `robots.txt` at the root of the site tells crawlers which paths they are not allowed to crawl. You also need to handle different types of files such as images, PDFs, or any other kinds of formats on the web that contain useful information for your search application. Ideally, the crawler should recognize duplicate pages so it doesn't repeat itself or get stuck in a loop. Finally, it may be useful to discover hidden URLs; these are URLs that may not be linked from any page yet still contain content that you'd like to index.

So, what are the major crawling strategies? In general, breadth-first search is the most common because it naturally balances server load. Parallel crawling is also very natural because this task is very easy to parallelize. One interesting variation is called **focused crawling**. Here, we're going to crawl some pages about a particular topic, e.g., all pages about automobiles. This is typically going to start with a query that you use to get some results. Then, you gradually crawl more. An even more extreme version of focused crawling is (for example) downloading and indexing all forum posts on a particular forum. In this case, we might have a URL such as

```
http://www.text-data-book-forum.com/boards?id=3
```

which refers to the third post on the forum. By changing the `id` parameter, we can iterate through all forum posts and index them quite easily. In this scenario, it's especially important to add a delay between requests so that the server is not overwhelmed.

Another challenge in crawling is to find new pages that have been created since the crawler last ran. This is very challenging if the new pages have not been linked to any old page. If they are, then you can probably find them by following links from existing pages in your index.

Finally, we might face the scenario of incremental crawling or repeated crawling. Let's say you want to be able to create a web search engine. Clearly, you first crawl data from the web. In the future we just need to crawl the updated pages. This is a very interesting research question: how can we determine when a page needs to be recrawled (or even when a new page has been created)? There are two major factors to consider here, the first of which is whether a particular page would be updated frequently. If the page is a static page that hasn't been changed for months, it's probably not necessary to re-crawl it every day since it's unlikely that it will be changed frequently. On the other hand, if it's (for example) a sports score page

that gets updated very frequently, you may need to re-crawl even multiple times on the same day. The second factor to consider is how frequently a particular page is accessed by users of the search engine system. If it's a high-utility page, it's more important to ensure it is fresh. Compare it with another page that has never been fetched by any users for a year; even though that unpopular page has been changed a lot, it's probably not necessary to crawl that page—or at least it's not as urgent—to maintain its freshness.

10.2 Web Indexing

In this section, we will discuss how to create a web-scale index. After our crawler delivers gigabytes or terabytes of data, the next step is to use the indexer to create the inverted index. In general, we can use the standard information retrieval techniques for creating the index, but there are new challenges that we have to solve for web scale indexing. The two main challenges are scalability and efficiency.

The index will be so large that it cannot actually fit into any single machine or single disk, so we have to store the data on multiple machines. Also, because the data is so large, it's beneficial to process the data in parallel so that we can produce the index quickly. To address these challenges, Google has made a number of innovations. One is the **Google File System**, which is a general distributed file system that can help programmers manage files stored on a cluster of machines. The second is **MapReduce**, which is a general software framework for supporting parallel computation. Hadoop is the most well known open source implementation of MapReduce, now used in many applications.

Figure 10.1 shows the architecture of the Google File System (GFS). It uses a very simple centralized management mechanism to manage all the specific locations of files. That is, it maintains a file namespace and lookup table to know where exactly each file is actually stored. The application client talks to the GFS master node, which obtains specific locations of the files to process. This filesystem stores its files on machines in fixed-size chunks; each data file is separated into many 64 MB chunks. These chunks are replicated to ensure reliability. All of these details are something that the programmer doesn't have to worry about, and it's all taken care of by this filesystem. From the application perspective, the programmer would see a normal file. The program doesn't have to know where exactly it's stored, and can just invoke high level operators to process the file. Another feature is that the data transfer is directly between application and chunk servers, so it's efficient in this sense as well.

On top of the GFS, Google proposed MapReduce as a general framework for parallel programming. This supports tasks like building an inverted index. Like GFS,

Simple centralized
management

Fixed chunk size
(64 MB)

Chunk is replicated
to ensure reliability
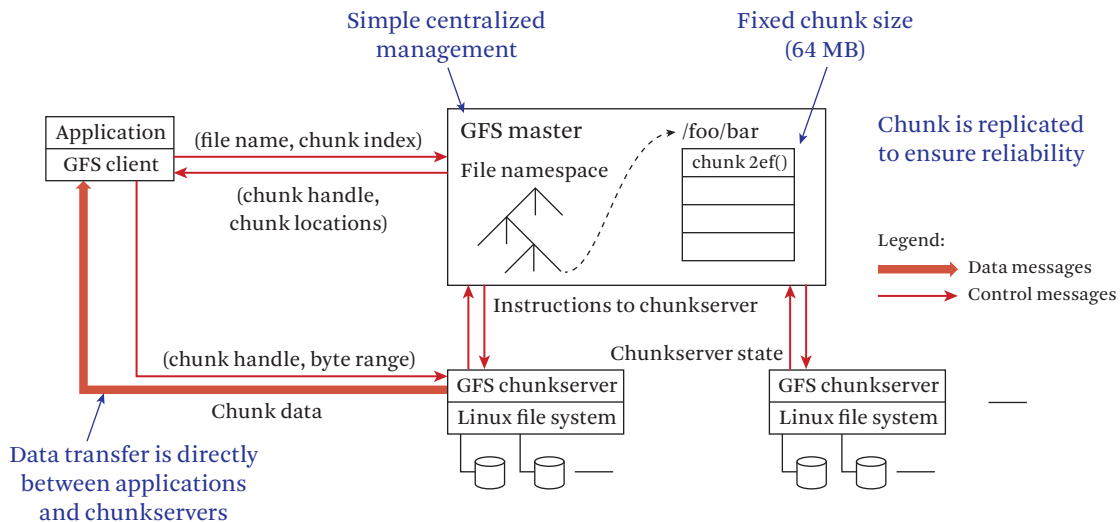
Figure 10.1   Illustration of Google File System. (Based on Ghemawat et al. [2003])

this framework hides low level features from the programmer. As a result, the programmer can make minimum effort to create an application that can be run on a large cluster in parallel. Some of the low-level details hidden in the framework are communications, load balancing, and task execution. Fault tolerance is also built in; if one server goes down, some tasks may not be finished. Here, the MapReduce mechanism would know that the task has not been completed and would automatically dispatch the task on other servers that can do the job. Again, the programmer doesn't have to worry about this.

In MapReduce, the input data are separated into a number of (*key*, *value*) pairs. What exactly the value is will depend on the data. Each pair will be then sent to a map function which the programmer writes. The map function will then process these (*key*, *value*) pairs and generate a number of other (*key*, *value*) pairs. Of course, the new key is usually different from the old key that's given to map as input. All the outputs of all the calls to map are collected and sorted based on the key. The result is that all the values that are associated with the same key will be grouped together.

For each unique key we now have a set of values that are attached to this key. This is the data that is sent to the reduce function. Each reduce instance will handle a different key. This function processes its input, which is a key and a set of values, to produce another set of (*key*, *value*) pairs as the output. This is the general framework of MapReduce. Now, the programmer only needs to write the map function and the reduce function. Everything else is taken care of by the
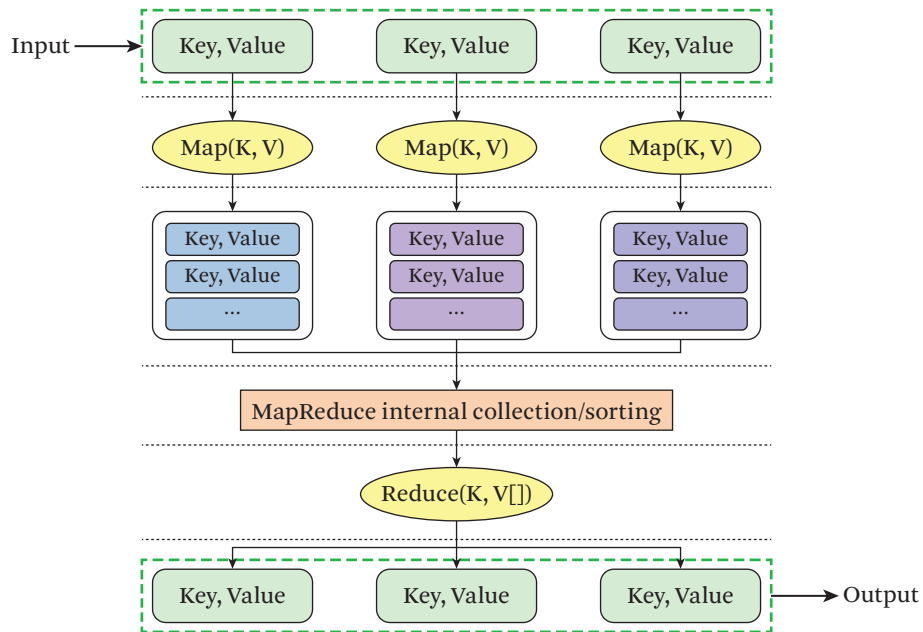
Figure 10.2 Computation flow of MapReduce.

MapReduce framework. With such a framework, the input data can be partitioned into multiple parts which are processed in parallel first by map, and then processed again in parallel once we reach the reduce stage.

Figure 10.3 shows an example of word counting. The input is files containing tokenized words and the output that we want to generate is the number of occurrences of each word. This kind of counting would be useful to assess the popularity of a word in a large collection or achieving an effect of IDF weighting for search. So, how can we solve this problem? One natural thought is that this task can be done in parallel by simply counting different parts of the file in parallel and combining all the counts. That's precisely the idea of what we can do with MapReduce: we can parallelize lines in this input file. More specifically, we can assume the input to each map function is a $(key, value)$ pair that represents the line number and the string on that line.

The first line is the pair (1, Hello World Bye World). This pair will be sent to a map function that counts the words in this line. In this case, there are only four words and each word gets a count of one. The map pseudocode shown at the bottom of the figure is quite simple. It simply needs to iterate over all the words in this line,
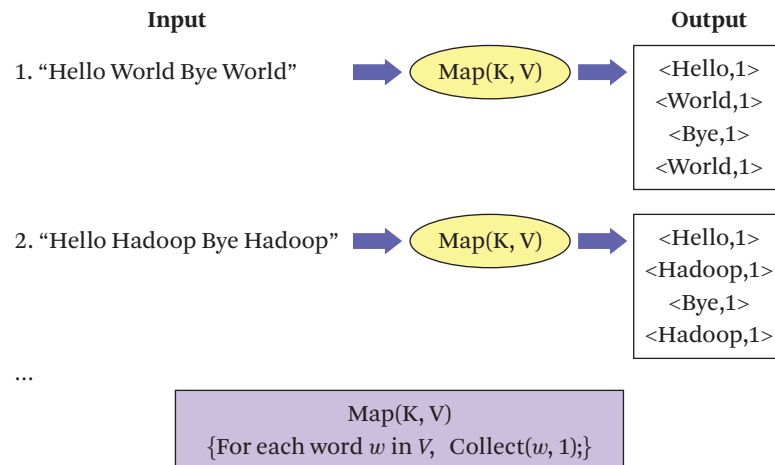
Figure 10.3   The `map` **function for word counting.**

and then just call a `Collect` function, which means it would then send the word and the counter to the collector. The collector would then try to sort all these key value pairs from different map functions. The programmer specifies this function as a way to process each part of the data. Of course, the second line will be handled by a different instance of the map function, which will produce a similar output.

As mentioned, the collector will do the internal grouping or sorting. At this stage, you can see we have collected multiple pairs. Each pair is a word and its count in the line. Once we see all these pairs, then we can sort them based on the key, which is the word. Each word now is attached to a number of values, i.e., a number of counts. These counts represent the occurrences of this word in different lines. These new $(key, value)$ pairs will then be fed into a `reduce` function.

Figure 10.4 shows how the `reduce` function finishes the job of counting the total occurrences of this word. It already has these partial counts, so all it needs to do is simply add them up. We have a counter and then iterate over all the words that we see in this array, shown in pseudocode at the bottom of the figure. Finally, we output the key and the total count, which is precisely what we want as the output of this whole program. As we can see, this is already very similar to building an inverted index; the output here is indexed by a word, and we have a dictionary of the vocabulary. What's missing is the document IDs and the specific frequency counts of words in each particular document. We can modify this slightly to actually build an inverted index in parallel.
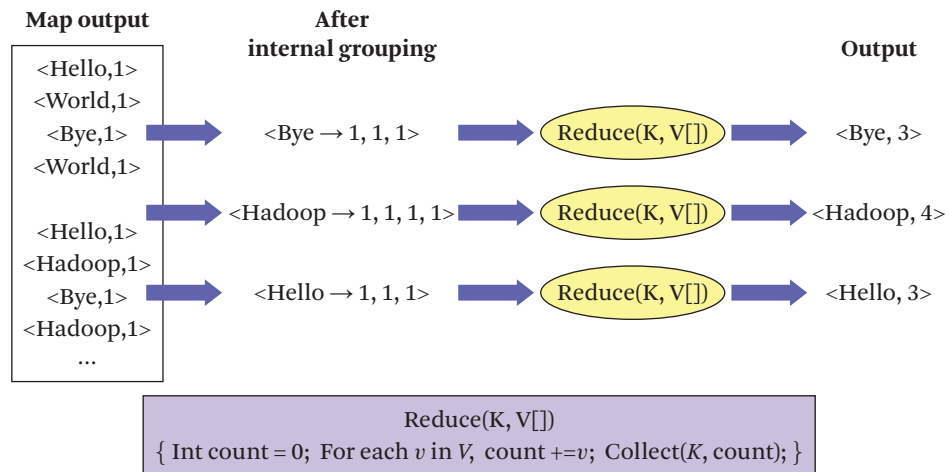
Figure 10.4 The `reduce` **function for word counting.**

Let's modify our word-counting example to create an inverted index. Figure 10.5 illustrates this example. Now, we assume the input to `map` function is a (key, $value$) pair where the key is a document ID and the value denotes the string content of all the words in that document. The `map` function will do something very similar to what we have seen in the previous example: it simply groups all the counts of this word in this document together, generating new pairs. In the new pairs, each key is a word and the value is the count of this word in this document followed by the document ID. Later, in the inverted index, we would like to keep this document ID information, so the `map` function keeps track of it.

After the `map` function, there is a sorting mechanism that would group the same words together and feed this data into the `reduce` function. We see the `reduce` function's input looks like an inverted index entry. It's just the word and all the documents that contain the word and the frequency of the word in those documents. All we need to do is simply to concatenate them into a continuous chunk of data, and this can be then stored on the filesystem. The `reduce` function is going to do very minimal work. Algorithm 10.1 can be used for this inverted index construction.

Algorithm 10.1, adapted from Lin and Dyer [2010], describes the `map` and `reduce` functions. A programmer would specify these two functions to run on top of a MapReduce cluster. As described before, `map` counts the occurrences of a word using an associative array (dictionary), and outputs all the counts together with the document ID. The `reduce` function simply concatenates all the input that it
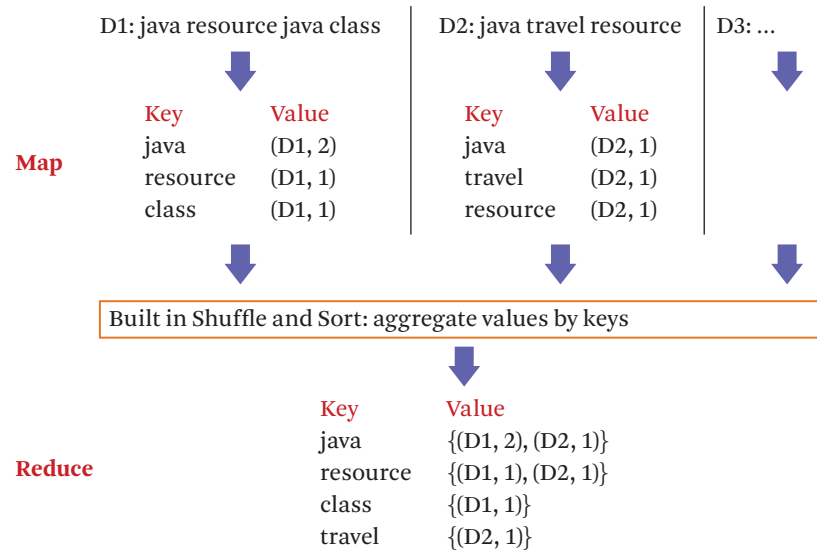
Figure 10.5   **Using MapReduce to create an inverted index.**

Algorithm 10.1   **Pseudocode for inverted index construction**

function MAP(docid n, doc d)
    $H \leftarrow$ new ASSOCIATIVEARRAY
    for term $t \in d$ **do**
        $H[t] \leftarrow H[t] + 1$
    end for
    for $t \in H$ **do**
        EMIT(t, $[n, H[t]]$)
    end for
end function

function REDUCE(term t, postings $[(a_1, f_1), (a_2, f_2), \ldots]$)
    $P \leftarrow$ new LIST
    for all (a, f) **do**
        APPEND(P, (a, f))
    end for
    SORT(P)
    EMIT(t, P)
end function

has been given and as single entry for this document ID key. Despite its simplicity, this MapReduce function allows us to construct an inverted index at a very large scale. Data can be processed by different machines and the programmer doesn't have to take care of the details. This is how we can do parallel index construction for web search.

To summarize, web scale indexing requires some new techniques that go beyond the standard traditional indexing techniques. Mainly, we have to store the index on multiple machines, and this is usually done by using a distributed file system like the GFS. Second, it requires creating the index in parallel because it's so large. This is done by using the MapReduce framework. It's important to note that the both the GFS and MapReduce framework are very general, so they can also support many other applications aside from indexing.

10.3   Link Analysis

In this section, we're going to continue our discussion of web search, particularly focusing on how to utilize links between pages to improve search. In the previous section, we talked about how to create a large index on using MapReduce on GFS. Now that we have our index, we want to see how we can improve ranking of pages on the web. Of course, standard IR models can be applied here; in fact, they are important building blocks for supporting web search, but they aren't sufficient for the following reasons.

First, on the web we tend to have very different information needs. For example, people might search for a web page or entry page—this is different from the traditional library search where people are primarily interested in collecting literature information. These types of queries are often called *navigational queries*, where the purpose is to navigate into a particular targeted page. For such queries, we might benefit from using link information. For example, navigational queries could be *facebook* or *yahoo finance*. The user is simply trying to get to those pages without explicitly typing in the URL in the address bar of the browser.

Secondly, web documents have much more information than pure text; there is hierarchical organization and annotations such as the page layout, title, or hyperlinks to other pages. These features provide an opportunity to use extra context information of the document to improve scoring. Finally, information quality greatly varies. All this means we have to consider many factors to improve the standard ranking algorithm, giving us a more robust way to rank the pages and making it more difficult for spammers to manipulate one signal to improve a single page's ranking.

"Extra text"/summary for a doc

Description
("anchor text")

Links indicate the utility of a doc

Hub   What does a link tell us?   Authority
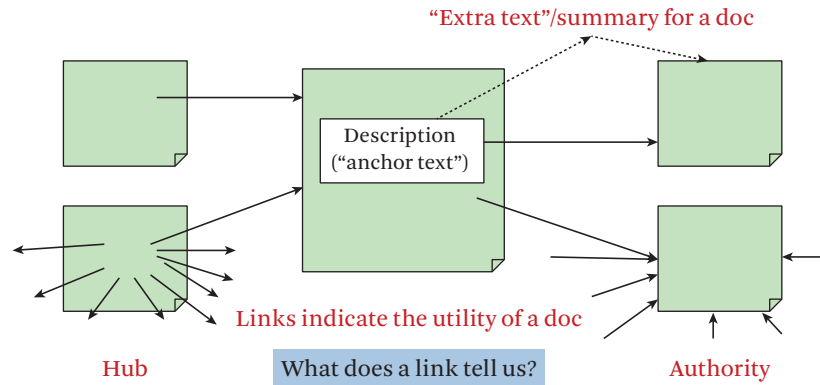
Figure 10.6 **Links provide useful information about pages.**

As a result of all these concerns, researchers have made a number of major extensions to the standard ranking algorithms. One is to exploit links to improve scoring, which is the main topic of this section. There are also algorithms to exploit large scale implicit feedback information in the form of clickthroughs. Of course, that belongs in the category of feedback techniques, and machine learning techniques are often used there. In general, web search ranking algorithms are based on machine learning algorithms to combine all kinds of features. Many of them are based on standard models such as BM25 that we talked about in Chapter 6. Link information is one of the important features used in combined scoring functions in modern web search systems.

Figure 10.6 shows a snapshot of a part of the web. We can see there are many links that connect different pages, and in the center, there is a description of a link that's pointing to the document on the right side. This description text is called *anchor text*. It is actually incredibly useful for search engines because it provides some extra description of the page being pointed to. For example, if someone wants to bookmark the Amazon.com front page, the person might make a link called *the big online bookstore* pointing to Amazon. The description is very similar to what the user would type in the query box when they are looking for such a page. Suppose someone types in a query like *online bookstore* or *big online bookstore*. The query would match this anchor text in the page. This actually provides evidence for matching the page that's been pointed to—the Amazon entry page. Thus, if you match the anchor text that describes the link to a page, it provides good evidence for the relevance of the page being pointed to.

On the bottom of Figure 10.6, there are some patterns of links which may indicate the utility of a document. For example, on the right side you can see a page has received many inlinks, meaning many other pages are pointing to this page. This shows that this page is quite useful. On the left side you can see a page that points to many other pages. This is a central page that would allow you to see many other pages. We call the first case an *authority page* and the second case a *hub page*. This means the link information can help in two ways; one is to provide extra text for matching (in the case of anchors) and the other is to provide some additional scores for the web pages to characterize how likely a page is a hub or an authority.

10.3.1   PageRank

Google's PageRank, a main technique that was used originally for link analysis, is a good example of leveraging page link information. PageRank captures page popularity, which is another word for authority. The intuition is that links are just like citations in literature. Think about one page pointing to another page; this is very similar to one paper citing another paper. Thus, if a page is cited often, we can assume this page is more useful. PageRank takes advantage of this intuition and implements it in a principled approach. In its simplest sense, PageRank is essentially doing citation counting or inlink counting.

It improves this simple idea in two ways. One is to consider indirect citations. This means you don't just look at the number of inlinks, rather you also look at the inlinks of your inlinks, recursively. If your inlinks themselves have many inlinks, your page gets credit from that. In short, if important pages are pointing to you, you must also be important. On the other hand, if those pages that are pointing to you are not pointed to by many other pages, then you don't get that much credit. This is the concept of indirect citations, or cascading citations.

Again, we can understand this idea by considering research papers. If you are cited by ten papers that are not very influential, that's not as good as if you're cited by ten papers that themselves have attracted a lot of other citations. Clearly, this is a case where we would like to consider indirect links, which is exactly what PageRank does. The other idea is that it's good to smooth the citations to accommodate potential citations that have not yet been observed. Assume that every page has a non-zero pseudo citation count. Essentially, you are trying to imagine there are many virtual links that will link all the pages together so that you actually get pseudo citations from everyone.

Another way to understand PageRank is the concept of a random surfer visiting every web page. Let's take a look at this example in detail, illustrated in Figure 10.7. On the left, there is a small graph, where each document d_1, d_2, d_3, and d_4 is a web

page, and the edges between documents are hyperlinks connecting them to each other. Let's assume that a random surfer or random walker can be on any of these pages. When the random surfer decides to move to a different page, they can either randomly follow a link from the current page or randomly choose a document to jump to from the entire collection. So, if the random surfer is at d_1, with some probability that random surfer will follow the links to either d_3 or d_4. The random surfing model also assumes that the surfer might get bored sometimes and decide to ignore the actual links, randomly jumping to any page on the web. If the surfer takes that option, they would be able to reach any of the other pages even though there is no link directly to that page. Based on this model, we can ask the question, "How likely, on average, would the surfer reach a particular page?" This probability is precisely what PageRank computes.

The PageRank score of a document d_i is the average probability that the surfer visits d_i. Intuitively, this should be proportional to the inlink count. If a page has a high number of inlinks then it would have a higher chance of being visited since there will be more opportunities of having the surfer follow a link there. This is how the random surfing model captures the idea of counting the inlinks. But, it also considers the indirect inlinks; if the pages that point to d_i have themselves a lot of inlinks, that would mean the random surfer would very likely reach one of them. This increases the chance of visiting d_i. This is a nice way to capture both indirect and direct links.

Mathematically, we can represent this document network as a matrix M, displayed in the center of Figure 10.7. Each row stands for a starting page. For example, row one would indicate the probability of going to any of the four pages from d_1. We see there are only two non-zero entries. Each is one half since d_1 is pointing to only two other pages; thus if we can randomly choose to visit either of them from d_1, they'd each have a probability of $\frac{1}{2}$. We have zeros for the first two columns for

$$M = \begin{bmatrix} 0 & 0 & 1/2 & 1/2 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \end{bmatrix}$$

Transition matrix

$M_{ij} =$ probability of going from d_i to d_j

$$\sum_{j=1}^{N} M_{ij} = 1$$
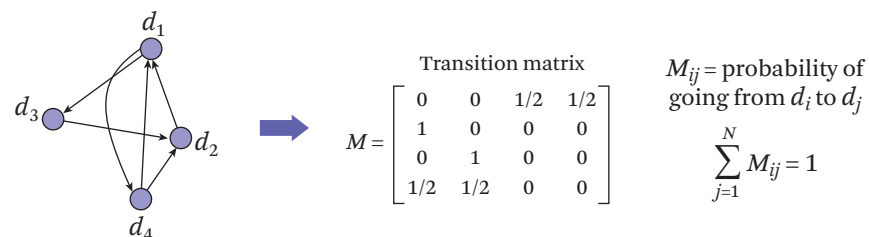
Figure 10.7 Example of a web graph and the corresponding transition matrix.

d_1 since d_1 doesn't link to itself and it doesn't link to d_2. Thus, M_{ij} is the probability of going from d_i to d_j. Each row's values should sum to one, because the surfer will have to go to precisely one of these pages. Now, how can we compute the probability of a surfer visiting a particular page?

We can compute the probability of reaching a page as follows:

$$p_{t+1}(d_j) = \underbrace{(1-\alpha)\sum_{i=1}^{N} M_{ij}\,p_t(d_i)}_{\text{reach } d_j \text{ by following a link}} + \underbrace{\alpha\sum_{i=1}^{N}\frac{1}{N}p_t(d_i)}_{\text{reach } d_j \text{ by random jumping}} \tag{10.1}$$

On the left-hand side is the probability of visiting page d_j at time $t + 1$, the next time count. On the right-hand side, we can see the equation involves the probability at page d_i at time t, the current time step. The equation captures the two possibilities of reaching a page d_j at time $t + 1$: through random surfing or following a link. The first part of the equation captures the probability that the random surfer would reach this page by following a link. The random surfer chooses this strategy with probability $1 - \alpha$; thus, there is a factor of $1 - \alpha$ before this term. This term sums over all the possible N pages that the surfer could have been at time t. Inside the sum is the product of two probabilities. One is the probability that the surfer was at d_i at time t. That's $p_t(d_i)$. The other is the transition probability from d_i to d_j, which we know is represented as M_{ij}. So, in order to reach this d_j page, the surfer must first be at d_i at time t and would have to follow the link to go from d_i to d_j. The second part is a similar sum. The only difference is that now the transition probability is uniform: $\frac{1}{N}$. This part captures the probability of reaching this page through random jumping, where α is the probability of random jumping.

This also allows us to see why PageRank captures a smoothing of the transition matrix. You can think this $\frac{1}{N}$ comes from another transition matrix that has all the elements as $\frac{1}{N}$. It is then clear that we can merge the two parts. Because they are of the same form, we can imagine there's a different matrix that's a combination of this M and the uniform matrix I. In this sense, PageRank uses this idea of smoothing to ensure that there's no 0 entry in the transition matrix.

Now, we can imagine that if we want to compute average probabilities, they would satisfy this equation without considering the time index. So let's drop the time index and assume that they would be equal; this would give us N equations, since each page has its own equation. Similarly, there are also precisely N variables. This means we now have a system of N linear equations with N variables. The problem boils down to solving this system of equations, which we can write in the following form:

$$p(d_j) = \sum_{i=1}^{N} \left[\frac{1}{N}\alpha + (1-\alpha)M_{ij} \right] \cdot p(d_i) \rightarrow \vec{p} = (\alpha I + (1-\alpha)M)^{\mathsf{T}}\vec{p}, \quad (10.2)$$

where

$$I_{ij} = \frac{1}{N} \quad \forall i, j.$$

The vector $\vec{p}$ equals the transpose of a matrix multiplied by $\vec{p}$ again. The transposed matrix is in fact the sum from 1 to N written in matrix form. Recall from linear algebra that this is precisely the equation for an eigenvector. Thus, this equation can be solved by using an iterative algorithm. In this iterative algorithm, called *power iteration*, we simply start with a random $\vec{p}$. We then repeatedly update $\vec{p}$ by multiplying the transposed matrix expression by $\vec{p}$.

Let's look at a concrete example: set $\alpha = 0.2$. This means that there is a 20% chance of randomly jumping to a page on the entire web and an 80% chance of randomly following a link from the current page. We have the original transition matrix M as before that encodes the actual links in the graph. Then, we have this uniform smoothing transition matrix I representing random jumping. We combine them together with interpolation via α to form another matrix we call A:

$$A = (1 - 0.2)M + 0.2I = 0.8 \begin{bmatrix} 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \end{bmatrix} + 0.2 \begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{bmatrix}. \quad (10.3)$$

The PageRank algorithm will randomly initialize $\vec{p}$ first, and then iteratively update it by using matrix multiplication. If we rewrite this matrix multiplication in terms of just A, we'll get the following:

$$\begin{bmatrix} p_{t+1}(d_1) \\ p_{t+1}(d_2) \\ p_{t+1}(d_3) \\ p_{t+1}(d_4) \end{bmatrix} = A^{\mathsf{T}} \begin{bmatrix} p_t(d_1) \\ p_t(d_2) \\ p_t(d_3) \\ p_t(d_4) \end{bmatrix} = \begin{bmatrix} 0.05 & 0.85 & 0.05 & 0.45 \\ 0.05 & 0.05 & 0.85 & 0.45 \\ 0.45 & 0.05 & 0.05 & 0.05 \\ 0.45 & 0.05 & 0.05 & 0.05 \end{bmatrix} \begin{bmatrix} p_t(d_1) \\ p_t(d_2) \\ p_t(d_3) \\ p_t(d_4) \end{bmatrix}. \quad (10.4)$$

If you want to compute the updated value for d_1, you multiply the top row in A by the column vector of PageRank scores from the previous iteration. This is how we update the vector; we started with some initial values and iteratively multiply the matrices together, which generates a new set of scores. We repeat this multiplication until the values in $\vec{p}$ converge. From linear algebra, we know

that since there are no zero values in the matrix, such iteration is guaranteed to converge. At that point we will have the PageRank scores for all the pages.

Interestingly, this update formula can be interpreted as propagating scores across the graph. We can imagine we have values initialized on each of these pages, and if you look at the equation, we combine the scores of the pages that would lead to reaching a page. That is, we'll look at all the pages that are pointing to a page and combine their scores with the propagated score in order to get the next score for the current document. We repeat this for all documents, which transfers probability mass across the network.

In practice, the calculation of the PageRank score is actually quite efficient because the matrices are sparse—that means that if there isn't a link into the current page, we don't have to worry about it in the calculation. It's also possible to normalize the equation, and that will give a somewhat different form, although the relative ranking of pages will not change. The normalization is to address the potential problem of zero outlinks. In that case, the probabilities of reaching the next page from the current page will not sum to 1 because we have lost some probability mass when we assume that there's some probability that the surfer will try to follow links (although in this case there are no links to follow!).

There are many extensions to PageRank. One extension is to do query-specific PageRank, also called **Personalized PageRank**. For example, in this topic-specific PageRank, we can simply assume when the surfer gets bored, they won't randomly jump into any page on the web. Instead, they jump to only those pages that are relevant to the query. For example, if the query is about sports, then we could assume that when we do random jumping, we randomly jump to a sports page. By doing this, our PageRank scores align with sports. Therefore, if you know the current query is about sports, we can use this specialized PageRank score to rank the results. Clearly, this would be better than using a generic PageRank score for the entire web.

PageRank is a general algorithm that can be used in many other applications such as network analysis, particularly in social networks. We can imagine if you compute a person's PageRank score on a social network (where a link indicates a friendship relation), you'll get some meaningful scores for people.

10.3.2 HITS

We've talked about PageRank as a way to capture authority pages. In the beginning of this section, we also mentioned that hub pages are useful. There is another algorithm we will discuss called HITS that is designed to compute both these scores for each page.

Adjacency matrix

$$A = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

Initial values: $a(d_i) = h(d_i) = 1$

$$h(d_i) = \sum_{d_j \in OUT(d_i)} a(d_j)$$

$$a(d_i) = \sum_{d_j \in IN(d_i)} h(d_j)$$

Iterate

$$\bar{h} = A\bar{a}; \quad \bar{a} = A^T\bar{h}$$
$$\bar{h} = AA^T\bar{h}; \quad \bar{a} = A^TA\bar{a}$$

Normalize:

$$\sum_i a(d_i)^2 = \sum_i h(d_i)^2 = 1$$
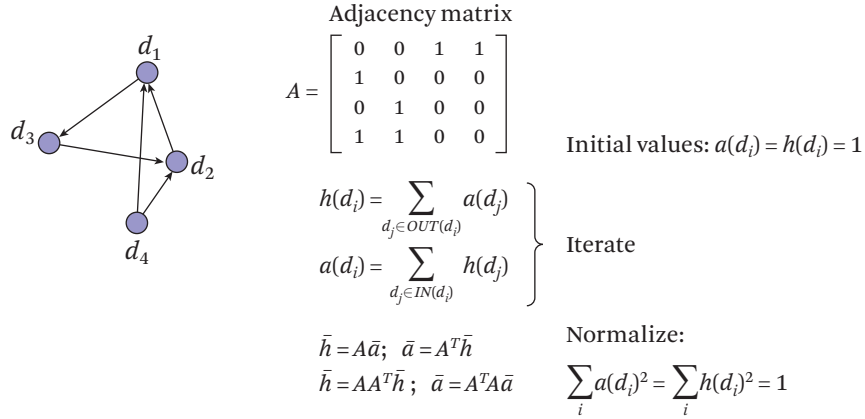
Figure 10.8   **Running the HITS algorithm on a small graph.**

Authority pages capture the intuition of widely-cited pages. Hub pages are those that point to good, authority pages, or contain some collection of knowledge in the form of links. The main idea of the HITS algorithm is a reinforcement mechanism to help improve the scoring for both hubs and authorities. It will assume that good authorities are cited by good hubs. That means if you're cited by many pages with good hub scores, then that increases your authority score. Similarly, good hubs are those that point to good authorities. So if you are pointing to many good authority pages, then your hub score would be increased. Like PageRank, HITS is also quite general and has many applications in graph and network analysis. Briefly, we'll describe how it works.

Figure 10.8 shows the following. First, we construct the adjacency matrix A; it contains a 1 at position A_{ij} if d_i links to d_j and a zero otherwise. We define the hub score of a page $h(d_i)$ as a sum of the authority scores of all the pages that it points to. In the second equation, we define the authority score of a page $a(d_i)$ as a sum of the hub scores of all pages that point to it. This forms an iterative reinforcement mechanism.

These two equations can be also written in matrix form. The hub vector is equal to the product of the adjacency matrix and the authority vector. Similarly, the second equation can be written as the authority vector is equal to the product of A^T multiplied by the hub vector. These are just different ways of expressing these equations. What's interesting is that if you look at the matrix forms, you can plug the authority equation into the first one. That is, you can actually eliminate the authority vector completely, and you get the equation of only hub scores. We can

do the same trick for the hub formula. Thus, although we framed the problem as computing hubs and authorities, we can actually eliminate one of them to obtain the equation for the other.

The difference between this and PageRank is that now the matrix is actually a multiplication of the adjacency matrix and its transpose. Mathematically then, we would be computing a very similar problem. In HITS, we would initialize the values to one and apply the matrix equations $A^\mathsf{T}A$ and AA^T. We still need to normalize the adjacency matrix after each iteration. This would allow us to control the growth of the values; otherwise they would grow larger and larger.

To summarize, this section has shown that link information is very useful. In particular, the anchor text is an important feature in the text representation of a page. We also talked about the PageRank and HITS algorithms as two major link analysis algorithms in web search. Both can generate scores for pages that can be used in addition to standard IR ranking functions. PageRank and HITS are very general algorithms with useful variants, so they have many applications in analyzing other graphs or networks aside from the web.

10.4 Learning to Rank

In this section, we discuss using machine learning to combine many different features into a single ranking function to optimize search results. Previously, we've discussed a number of ways to rank documents. We talked about some retrieval models like BM25 or query likelihood; these can generate a content-based score for matching document text with a query. We also talked about the link-based approaches like PageRank that can give additional scores to help us improve ranking. The question now is how can we combine all these features (and potentially many other features) to do ranking? This will be very useful for ranking web pages not only just to improve accuracy, but also to improve the robustness of the ranking function so that's it not easy for a spammer to just perturb one or a few features to promote a page.

The general idea of learning to rank is to use machine learning to combine these features, optimizing the weight on different features to generate the best ranking function. We assume that given a query-document pair (q, d), we can define a number of features. These features don't necessarily have to be content-based features. They could be a score of the document with respect to the query according to a retrieval function such as BM25, query likelihood, pivoted length normalization, PL2, etc. There also can be a link-based score like PageRank or HITS, or an application of retrieval models to the anchor text of the page, which

are the descriptions of links that point to d. These can all be clues about whether this document is relevant or not to the query. We can even include a feature such as whether the URL has a tilde because this might indicate a home page.

The question is, of course, how can we combine these features into a single score? In this approach, we simply hypothesize that the probability that this document is relevant to this query is a function of all these features. We hypothesize that the probability of relevance is related to these features through a particular function that has some parameters. These parameters control the influence of different features on the final relevance. This is, of course, just an assumption. Whether this assumption really makes sense is still an open question.

Naturally, the next question is how to estimate those parameters. How do we know which features should have high weight and which features should have low weight? This is a task of training or learning.

In this approach, we use training data. This is data that have been judged by users, so we already know the relevance judgments. We know which documents should be highly ranked for which queries, and this information can be based on real judgments by users or can be approximated by just using clickthrough information as we discussed in Chapter 7. We will try to optimize our search engine's retrieval accuracy (using, e.g., MAP or NDCG) on the training data by adjusting these parameters. The training data would look like a table of tuples. Each tuple has three elements: the query, the document, and the judgment. Let's take a look at a specific method that's based on logistic regression:

$$
\log \frac{P(R = 1 \mid q, d)}{1 - P(R = 1 \mid q, d)} = \beta_0 + \sum_{i=1}^{n} \beta_i X_i. \tag{10.5}
$$

This is one of many different methods, and actually one of the simpler ones. In this approach, we simply assume the relevance of a document with respect to the query is related to a linear combination of all the features. Here we have X_i to denote the ith feature value, and we can have as many features as we would like. We assume that these features can be combined in a linear manner. The weight of feature X_i is controlled by a parameter β_i. A larger β_i would mean the feature would have a higher weight and it would contribute more to the scoring function.

The specific form of the function also gives the following probability of relevance:

$$
P(R = 1 \mid d, q) = \frac{1}{1 + \exp\left\{ -\beta_0 - \sum_{i=1}^{n} \beta_i X_i \right\}}. \tag{10.6}
$$

	$X_1(q,d)$	$X_2(q,d)$	$X_3(q,d)$
$d_1(R=1)$	0.7	0.11	0.65
$d_2(R=0)$	0.3	0.05	0.4

Figure 10.9   **Example of a combination of multiple features in ranking.**

We know that the probability of relevance is within the range $[0, 1]$ and we assume that the scoring function is a transformed form of the linear combination of features. We could have had a scoring function directly based on the linear combination of β and X, but then the value of this linear combination could easily go beyond 1. Thus the reason why we use the logistic regression instead of linear regression is to map this combination onto the range $[0, 1]$. This allows us to connect the probability of relevance (which is between 0 and 1) to a linear combination of arbitrary coefficients. If we rewrite this combination of weights into a probability function, we will get the predicted score.

If this combination of features and weights gives us a high value, then the document is more likely relevant. This isn't necessarily the best hypothesis, but it is a simple way to connect these features with the probability of relevance.

The next task is to see how we estimate the parameters so that the function can truly be applied; that is, we need to estimate the β values. Let's take a look at a simple example shown in Figure 10.9.

In this example, we have three features. One is the BM25 score of the document for the query. One is the PageRank score of the document, which might or might not depend on the query. We might also have a topic-sensitive PageRank score that would depend on the query. Lastly, we have a BM25 score on the anchor text of the document. These are then the three feature values for a particular (*document*, *query*) pair. In this case the document is d_1 and the judgment says that it's relevant. The document d_2 is another training instance with different feature values, but in this case it's non-relevant. Of course, this is an overly-simplified example where we just have two instances, but it's sufficient to illustrate the point.

We use the maximum likelihood estimator to estimate the parameters. That is, we're going to predict the relevance status of the document based on the feature values. The likelihood of observing the relevance status of these two documents using our model is

$$p(\{q, d_1, R = 1\}, \{q, d_2, R = 0\}) = \frac{1}{1 + \exp\left\{-\beta_0 - 0.7\beta_1 - 0.11\beta_2 - 0.65\beta_3\right\}}$$
$$\times \left(1 - \frac{1}{1 + \exp\left\{-\beta_0 - 0.3\beta_1 - 0.05\beta_2 - 0.4\beta_3\right\}}\right).$$

We hypothesize that the probability of relevance is related to the features in this way. We're going to see for what values of β we can predict the relevance effectively. The expression for d_1 should give a higher value than the expression for d_2; in fact, we hope d_1's value is close to one since it's a relevant document.

Let's see how this can be mathematically expressed. It's similar to expressing the probability of a document, only we are not talking about the probability of words, but the probability of relevance. We need to plug in the X values. The β values are still unknown, but this expression gives us the probability that this document is relevant if we assume such a model. We want to maximize this probability for d_1 since this is a relevant document. For the second document, we want to predict the probability that the document is non-relevant. This means we have to compute 1 minus the probability of relevance. That's the reasoning behind this whole expression then; it's our probability of predicting these two relevance values. The whole equation is our probability of observing a $R = 1$ and $R = 0$ for d_1 and d_2 respectively. Our goal is then to adjust the β values to make the whole expression reach its maximum value. In other words, we will look at the function and choose β values to make this expression as large as possible.

After we learn the regression parameters, we can use this expression for any new query and new document once we have their features. This formula is then applied to generate a ranking score for a particular query.

There are many more advanced learning algorithms than the regression-based reproaches. They generally attempt to theoretically optimize a retrieval measure such as MAP or NDCG. Note that the optimization objective we just discussed is not directly related to a retrieval measure. By maximizing the prediction of one or zero, we don't necessarily optimize the ranking of those documents. One can imagine that while our prediction may not be too bad, the ranking can be wrong. We might have a larger probability of relevance for d_2 than d_1. So, that won't be good from a retrieval perspective, even though by likelihood the function is not bad. More advanced approaches will try to correct this problem. Of course, then the challenge is that the optimization problem will be harder to solve. In contrast, we might have another case where we predicted probabilities of relevance around

0.9 for non-relevant documents. Even though the predicted score is very high, as long as the truly relevant documents receive scores that are greater than 0.9, the ranking will still be acceptable to a user.

These learning to rank approaches are actually quite general. They can be applied to many other ranking problems aside from retrieval problems. For example, recommender systems, computational advertising, summarization, and many other relevant applications can all be solved using this approach.

To summarize, we talked about using machine learning to combine features to predict a ranking result. Actually, the use of machine learning in information retrieval began many decades ago. Rocchio feedback, discussed in Chapter 7, was a machine learning approach applied to learn the optimal feedback. Many algorithms are driven by the availability of massive amounts of training data in the form of clickthroughs. This data provides much useful knowledge about relevance, and so machine learning methods are applied to leverage this. The need for machine learning is also driven by the desire to combine many different feature types to predict an accurate ranking. web search especially drives this need since there are more features available on the web that can be taken advantage of for search. Using many different features also increases the robustness of the scoring function, which is useful in combating spam. Modern search engines all use some kind of machine learning techniques to combine many features to optimize ranking, and this is a major feature of current engines such as Google and Bing.

10.5   The Future of Web Search

Since this chapter concludes our coverage of search engines, we briefly talk about some possible future trends of web search and intelligent information retrieval systems in general. To further improve the accuracy of a search engine, it's important to consider special cases of information need. One particular trend is to have more and more specialized and customized search engines, which can be called *vertical search engines*. These vertical search engines can be expected to be more effective than the current general search engines because they could assume that a particular user belongs to a special group that might have a common information need.

Due to this customization, it's also possible to do personalization. The search can be personalized because we have a better understanding of the users. Restricting the domain of the search engine can also have some advantages in handling the documents, because we would have a better understanding of these documents. For example, particular words may not be ambiguous in such a domain, so we can bypass the problem of ambiguity.

Another trend we can expect to see is search engines that are able to learn over time, a form of lifetime learning or lifelong learning. This is very attractive because that means the search engine will be able to self-improve. As more people use it, the search engine will become better and better. This is already happening, because the search engines can learn from the relevance feedback. More users use it, and the quality of the search engine allows for the popular queries that are typed in by many users to retrieve better results.

A third trend might be the integration of information access. Search, navigation, and recommendation might be combined to form a full-fledged information management system. In the beginning of this book, we talked about push access versus pull access; these modes can be combined. For example, if a search engine detects that a user is unsatisfied with search results, a "note" may be made. In the future, if a new document is crawled that matches the user's information need recorded in the note, this new document could be pushed to the user. Currently, most of the cases of information recommendation are advertising, but in the future, you can imagine recommendation is seamlessly integrated into the system with multi-mode information access.

Another trend is that we might see systems that try to go beyond search to support user tasks. After all, the reason why people want to search is to solve a problem or to make a decision to perform a task. For example, consumers might search for opinions about products in order to purchase a product, so it would be beneficial to support the whole shopping workflow. For example, you can sometimes look at the review displayed directly in search results; if the user decides to buy the product, they can simply click a button to go to the shopping site directly and make the purchase. While there is good support for shopping, current search engines do not provide good task support for many other tasks. Researchers might want to find related work or suggested citations. Currently, there's not much support for a task such as writing a paper.

We can think about any intelligent system—especially intelligent information systems—specified by three nodes. If we connect these nodes into a triangle, then we'll able to specify an information system. We can call this triangle the **Data-User-Service Triangle**. The three questions you ask are as follows.

- Who are you serving?
- What kind of data are you managing?
- What kind of service are you providing?
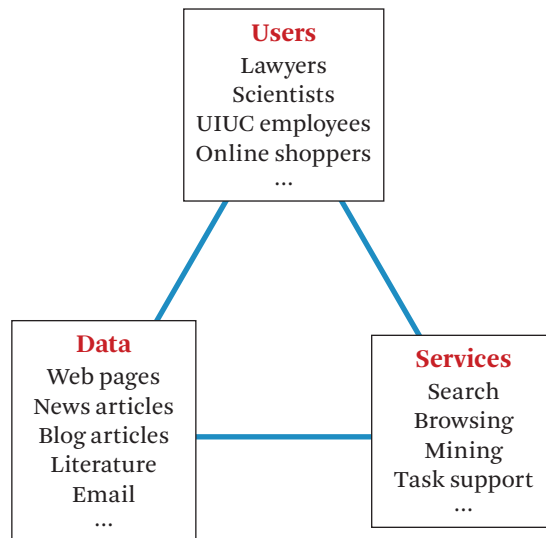
Figure 10.10   **The Data-User-Service triangle.**

These questions specify an information system; there are many different ways to connect them. Depending on how they are connected, we can specify all different types of systems. Let's consider some examples.

On the top of Figure 10.10 there are different kinds of users. On the left side, there are different types of data or information, and on the bottom, there are different service functions. Now imagine you can connect all these in different ways. For example, if you connect everyone with web pages, and support search and browsing, you get web search. If we connect university employees with organization documents or enterprise documents and support the search and browsing, we get enterprise search.

We could connect scientists with literature information to provide all kinds of services including search, browsing, alert to new relevant documents, mining or analyzing research trends, or task and decision support. For example, we might be able to provide support for automatically generating a related works section for a research paper; this would be closer to task support. Then, we can imagine this intelligent information system would be a type of literature assistant.

If we connect online shoppers with blog articles or product reviews, then we can help these people improve their shopping experience. We can provide data mining capabilities to analyze reviews, compare products and product sentiment, and provide task or decision support on choosing which product to buy. Or, we can connect

customer service people with emails from the customers. Imagine a system that can provide an analysis of these emails to find that the major complaints of the customers. We can imagine a system that could provide task support by automatically generating a response to a customer email by intelligently attaching a promotion message if appropriate. If they detect a positive message (not a complaint) then they might take this opportunity to attach some promotion information. If it's a complaint, then you might be able to automatically generate some generic response first and tell the customer that he or she can expect a detailed response later. All of these aim to help people to improve their productivity.

Figure 10.10 shows the trend of technology and characterizes intelligent information systems with three angles. In the center of the figure there's a triangle that connects keyword queries to search a bag-of-words representation. That means the current search engines basically provides search support to users and mostly model users based on keyword queries, seeing the data through a bag-of-words representation. Current search engines don't really "understand" information in the indexed documents. Consider some trends to push each node toward a more advanced function, away from the center. Imagine if we can go beyond keyword queries, look at the user search history, and then further model the user to completely understand the user's task environment, context, or other information. Clearly, this is pushing for personalization and a more complete user model, which is a major direction in order to build intelligent information systems. On the document side, we can also go beyond a bag-of-words implementation to have an entity-relation representation. This means we'll recognize people's names, their relations, locations, and any other potentially useful information. This is already feasible with today's natural language processing techniques.

Google has initiated some of this work via its Knowledge Graph. Once we can get to that level without much manual human effort, the search engine can provide a much better service. In the future, we would like to have a knowledge representation where we can perhaps add some inference rules, making the search engine more intelligent. This calls for large-scale semantic analysis, and perhaps this is initially more feasible for vertical search engines. That is, it's easier to make progress in one particular domain.

On the service side, we see we need to go beyond search to support information access in general; search is only one way to get access to information. Going beyond access, we also need to help people digest information once it is found, and this step has to do with analysis of information or data mining. We have to find patterns or convert the text information into real knowledge that can be used in application or actionable knowledge that can be used for decision making. Furthermore, the

knowledge will be used to help a user improve productivity in finishing a task. In this dimension, we anticipate that future intelligent information systems will provide interactive task support.

We should emphasize *interactive* here, because it's important to optimize the combined intelligence of users and the system. We can get some help from users in a natural way without assuming the system has to do everything. That is, the user and the machine can collaborate in an intelligent and efficient way. This combined intelligence will be high and in general, we can minimize the user's overall effort in solving their current problem.

This is the big picture of future intelligent information systems, and this hopefully can provide us with some insights about how to make further innovations on top of what we have today and also motivate the additional techniques to be covered in the later chapters of the book.

Bibliographic Notes and Further Reading

The classic reference for PageRank is Page et al. [1999], and that for HITS is Kleinberg [1999]. Lin and Dyer [2010] provides an excellent introduction to using MapReduce for text processing applications, including particularly a detailed treatment of how to use MapReduce for constructing an inverted index. Liu [2009] gives an excellent survey of research work on learning to rank.

Exercises

10.1. Examine the `robots.txt` file for several common sites. Can you figure out the format of this file? What type of data do these sites not want you to crawl? What is a user agent?

10.2. Simple Web Crawlers.   On Linux or Mac, try using `wget` to download a web page:

```
wget http://www.[insert-domain-here].com/
```

The file is probably saved with the extension `.html`. Open it up in your favorite text editor. It's just a bunch of HTML! This is the same thing you'd see if you right click the page in a browser and select "View Source".

10.3. Parsing Web Content.   An important step of web crawling is parsing the HTML into plaintext. There are many libraries available that do this. These libraries can also provide all the outgoing links (the `a href= . . .` tags). It's the crawler's job

to schedule crawling these links and to not get stuck in a cycle. This is easily avoided by keeping a list of already visited sites.

Using `wget` is not the only way to build a simple web crawler. You can make your own in Ruby or Python. These languages have many useful libraries for crawling. It's also possible to use an existing web crawler. For Python, Scrapy and Beautiful Soup are two popular crawling tools.

The last important point is to have a short wait period in between page requests. Not only is it considered polite to wait a few seconds between requests to avoid hammering the server, you may get blocked if you attempt to crawl too fast!

Use one of the above-mentioned tools to create a simple web crawler. Limit your crawling to 100 pages initially.

10.4. JavaScript Crawlers. For a simple page, an easy call to `wget` works very well. But nowadays, most web pages have a large amount of dynamically generated content that isn't part of the downloadable source. Try crawling a page that generates dynamic content. (Hint: you can find a page that generates dynamic content by using `wget` and searching for text you know is on the page. If you can't find it in the downloaded HTML file, it must have been dynamically generated!)

Most modern sites are composed almost entirely of dynamically generated content. Simply downloading the basic HTML source will not retrieve all the necessary content for indexing. We need to actually load the page and run the JavaScript that is called to populate the content.

Open up the URL of a dynamic page you'd like to crawl in your browser, and start the JavaScript console. (In Chrome, it's `CTRL-SHIFT-I`.) You can now interact with the page via JavaScript. Try typing this in the console:

```
alert('I'm a popup!')
```

Then try

```
for(var i = 0; i < 5; ++i) { console.log('Hello ' + i); }
```

Besides making annoying popups and useless counters, we can access the page title:

```
document.title
```

or we can access the text content:

```
document.body.innerText
```

This is what we want for indexing! Experiment with a JavaScript crawler using a technology such as PhantomJS. We've provided a simple script to download a

page, located at: http://sifaka.cs.uiuc.edu/ir/textdatabook/files/text-scraper.js. Try experimenting with the script to make it a true crawler instead of downloading only a single page.

10.5. Crawling a Domain.   Now that you know how to download a single HTML file, you can start crawling a domain. This means starting with some base URL and having the crawler follow and download links up to a certain depth. `wget` has some nice options that make this very easy.

```
wget --recursive --level=3 --wait=2 --accept html [url]
```

This command tells `wget` to traverse the site recursively by following links up to a depth of 3. It waits 2 between requests (which is considered polite and will help you from getting blocked). Finally, it is told to only download HTML pages since those are the ones with text that we want to index.

The output is kept in the same structure as on the web, downloaded into our working directory.

If you don't stop it manually (with CTRL-C), it will continue to crawl until all pages at the specified depth have been downloaded. Start off with a conservative depth, since you will not be sure how many pages are under a certain domain.

10.6. Cleaning HTML Files.   Before we add a page to the search engine's index, we will probably want to "clean" the HTML page. This means converting the HTML file into a plaintext file so that the keywords given to the search engine more easily match the words in our crawled document.

There are many tools available to convert HTML to text, some of which are even online. For our use though, we want to have a command-line based tool so we can automate it. We suggest using Python or Ruby. Below are two simple programs that both do the same thing.

- Python:

```
from bs4 import BeautifulSoup

html = open('filename.html').read()
soup = BeautifulSoup(html)
print soup.get_text() # or save to another file
```

   For this method, you will have to install the BeautifulSoup library.

- Ruby:

```
require 'nokogiri'
```

```
html = File.open('filename.html').read
noko = Nokogiri::HTML(html)
puts noko.text # or save to another file
```

For this method, you will have to install the Nokogiri library.

Experiment with one or both of these cleaners to parse your crawled files. Or, write your own cleaner with different technology!

10.7. Create a dynamically updating web search engine by combining your crawler, cleaner, and META. You can schedule the crawler to run during a specified time interval. Once new files are downloaded and cleaned, recreate the index.

10.8. Give a suggestion on how to improve a ranking function for web search to take in additional page information such as the title field or page layout.

10.9. Write MapReduce pseudocode that creates an inverted index that contains all the necessary information to rank documents using Dirichlet prior smoothing or Jelinek-Mercer smoothing.

10.10. If we add a new page to the web, what happens to other existing PageRank scores? Explain.

10.11. Compare PageRank with Personalized PageRank. Can one or both be pre-computed to save query processing time? Why or why not?

10.12. Give a query where a high-scoring authority page could be a desired document and a query where a high-scoring hub page could be a desired document.

10.13. After reading Chapter 15, you may have some alternative ideas of how to design a learning to rank algorithm. For example, can you outline an idea of how we can optimize (e.g.) MAP for a set of training queries?

10.14. Thinking back to Chapter 9, what is a good objective function to optimize for learning to rank? Is MAP the best choice? Why or why not?

10.15. Outline a method for combining user feedback with a learning to rank approach.

Recommender Systems

In our many discussions of search engine systems, we have addressed the issue of short-term (ad hoc) information need. This is a temporary need from a static information source, where the user pulls relevant information. Examples are library or web search. Conversely, most users also have long-term information needs, such as filtering or recommending documents (or any other item type) from a dynamic information source; here, the user is pushed information by a system. Examples include a news filter, email filter, movie/book recommender, or literature recommender. Although there is some distinction between a recommender system (emphasizing delivery of useful items to users) and a filtering system (emphasizing exclusion of useless items), the techniques used are similar, so we will use "recommender" and "filtering" interchangeably for convenience.

Unlike ad hoc search where we may not get much feedback from a user, in filtering, we can expect to collect a lot of feedback information from the user, making it important to learn from the feedback information to improve filtering performance.

In filtering, documents are delivered from some dynamic information source. A system must make a binary decision regarding the relevance of a document to a user as soon as it "arrives." This is more difficult than search where we can simply provide a ranked list and rely on a user to flexibly set the cutoff. On the other hand, since we can collect feedback information, we can expect to get more and more information about what the user likes, making it easier to distinguish relevant documents from non-relevant ones.

The essential filtering question is: will user u like item x? Our approach to answering this question defines which of the following two strategies we apply.

- Content-based filtering: look at what u likes and characterize x
- Collaborative filtering: look at who likes x and characterize u

Content-based filtering is to learn what kind of content a user likes and then match the content of a current article with a "content prototype" that we believe describes well what the user likes. **Collaborative filtering** is to look at what other

similar users like and assume that if those other users who are similar to you like an item, you may also like it. Note that if we can get user ratings of items, collaborative filtering can be applied to recommend any item. Content-based filtering, however, can only be applied to a case where we know how to measure similarity of items. In any specific application, we will want to combine the two approaches to optimize the filtering performance.

Content-based filtering can usually be done by extending a retrieval system to add a thresholding component. There are two challenges associated with threshold setting. First, at the beginning, we must set an initial threshold without requiring much information from a user. Second, over time, we need to learn from feedback to optimize the threshold. Many threshold learning methods have been proposed. In practice, a simple thresholding strategy such as the beta-gamma threshold setting method we will discuss is often good enough.

The basic idea behind collaborative filtering is to predict the rating of a current active user u for object x based on a weighted average of the ratings of x given by similar users to u. Thus, we can think of this approach involving two steps: In the first step, we simply "retrieve" similar users to the current user u where similarity is often defined as the similarity between the two vectors for two users. Each user can be represented by a rating vector (i.e., all the ratings given by this user). The similarity of two vectors can be measured based on the cosine similarity or Pearson correlation of the two vectors, which tends to perform very well empirically. In the second step, we compute a weighted average of the ratings of x given by all these retrieved similar users where the weight is the correlation between the active user and the corresponding user (to the weight).

As we will see in this chapter, many recommender systems are extensions of the information retrieval systems and techniques we have discussed previously. Therefore, it may be beneficial to read Chapter 6 before this one if the reader is unfamiliar with the basic concepts or terminology. We continue this chapter with content-based recommendation, followed by a section on user-based recommendation (collaborative filtering).

11.1 Content-based Recommendation

Figure 11.1 shows a generic information filtering system where a stream of content is absorbed by a filtering system. Based on either the content of the item or the other users that liked the item, the system decides whether or not to pass the item along to the user. In this section, the filtering system will inspect the content of the item and compare it to both the user's preferences and feedback without considering information from other users.
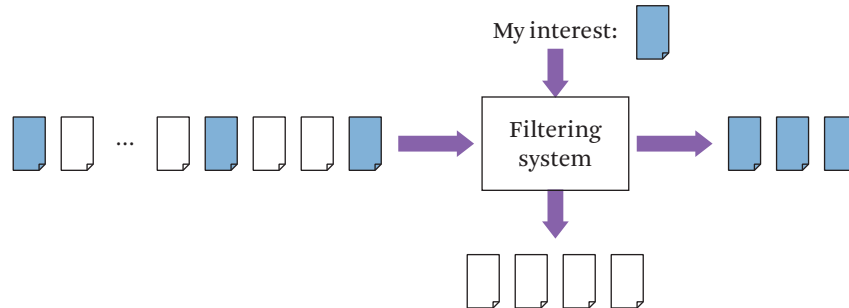
Figure 11.1 Diagram of a generic information filtering system; blue documents should be delivered to the user based on the user's preferences.
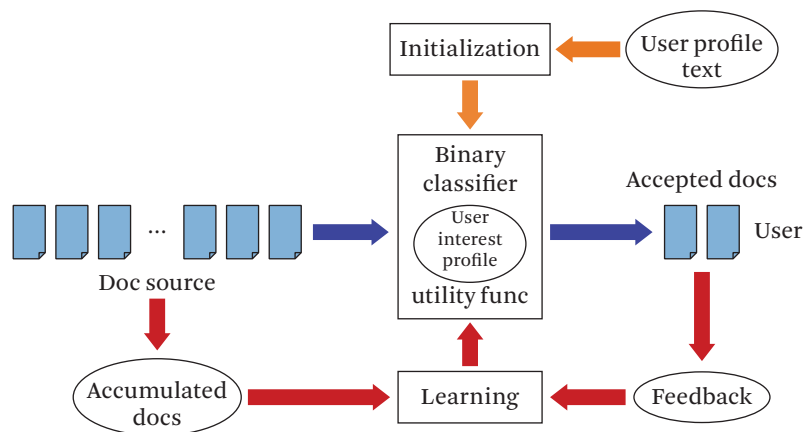
Figure 11.2 A content-based information filtering system with more details filled in for each component.

As shown in Figure 11.2, in an information filtering system there would be a binary classifier[1] that would have some knowledge of the user's interests, called the user interest profile. Originally, the user profile could be a text summary or keywords of what the user is interested in for the case of text document recommendation. This information is set in an initialization module that would take a user's

---

1. A binary classifier is an algorithm that can take an item and determine whether it belongs to one of two categories. In this case, the categories are relevant or non-relevant. For more information on classification, see Chapter 15.

input, perhaps from the user's specified keywords or chosen categories. This is fed into the system as the initial user profile.

Next, there is a utility function to help the system make decisions; it helps the system decide where to set an acceptance threshold θ determining whether or not the current item should be shown to the user. The learning module adjusts its parameters based on the user's feedback over time. Typically in information filtering applications, the users' information need is stable. Due to this, the system would have many opportunities to observe the user if the user views a recommended item, since the user can indicate whether the recommended item was relevant or not. Thus, such feedback can be long-term, allowing the system to collect much information about this user's interests, which is then used to improve the classifier.

How do we know this filtering system actually performs well? In this case, we cannot use the ranking evaluation measures such as MAP or NDCG because we can't afford waiting for a significant number of documents to rank them to make a decision for the user; the system must make a decision in real time. In general, this decision is whether the item is above the acceptance threshold θ or not. In other words, we're trying to decide absolute relevance. One common strategy is to use a utility function, and below is an example of a linear utility function:

$$\mathcal{U} = 3 \cdot |R| - 2 \cdot |R'|, \tag{11.1}$$

where R is the set of relevant documents delivered to the user and R' is the set of non-relevant documents delivered to the user (that the user rejected). In a way, we can treat this as a gambling game. If the system delivers one good item, let's say you win \$3, or you gain \$3. If you deliver a bad document, you would lose \$2. This utility function measures how much money you would accumulate (or lose) by considering this kind of game. It's clear that if you want to maximize this utility function, your strategy should be to deliver as many good items as possible while simultaneously minimizing the delivery of bad items.

One interesting question here is how to set these coefficients. We just showed a 3 and a -2 as the possible coefficients, but we can ask the question "are they reasonable?" What about other choices? We could have 10 and -1, or 1 and -10. How would these utility functions affect the system's output? If we use 10 and -1, you will see that while we get a big reward for delivering a good document, we incur only a small penalty for delivering a bad one. Intuitively, the system would be encouraged to deliver more documents, since delivering more documents gives a better chance of obtaining a high reward. If we choose 1 and -10, it is the opposite case: we don't really get such a big prize if a good document is delivered, while a

large loss is incurred if we deliver a bad one. The system in this case would be very reluctant to deliver many documents, and has to be absolutely sure that it's a relevant one. In short, the utility function has to be designed based on a specific application preference, potentially different for different users.

The three basic components in content-based filtering are the following.

Initialization module.   Gets the system started based only on a very limited text description, or very few examples, from the user.

Decision module.   Given a text document and a profile description of the user, decide whether the document should be delivered or not.

Learning module.   Learn from limited user relevance judgments on the delivered documents. (If we don't deliver a document to the user, we'd never know whether the user likes it or not.)

All these modules would have to be optimized to maximize the utility function $\mathcal{U}$. To solve these problems, we will talk about how to extend a retrieval system for information filtering. First, we can reuse retrieval techniques to do scoring; we know how to score documents against queries and measure the similarity between a profile text description and a document. We can use a score threshold θ for the filtering decision. If $\mathrm{score}(d) > \theta$, we say document d is relevant and we are going to deliver it to the user. Of course, we still need to learn from the history, and for this we can use the traditional feedback techniques to learn to improve scoring, such as Rocchio. What we don't know how to do yet is learn how to set θ. We need to set it initially and then we have to learn how to update it over time as more documents are delivered to the user and we have more information.

Figure 11.3 shows what the system might look like if we generalized a vector-space model for filtering problems. The document vector could be fed into a scoring module, which already exists in a search engine that implements the vector-space model, where the profile will be treated as a query. The profile vector can be matched with the document vector to generate the score. This score will be fed into a thresholding module that would say *yes* or *no* depending on the current value of θ. The evaluation would be based on the utility for the filtering results. If it says *yes*, the document will be sent to the user, and then the user could give some feedback. The feedback information would be used to both adjust the threshold and change the vector representation. In this sense, *vector* learning is essentially the same as query modification or feedback in search. The *threshold* learning is a new component that we need to talk a little bit more about.
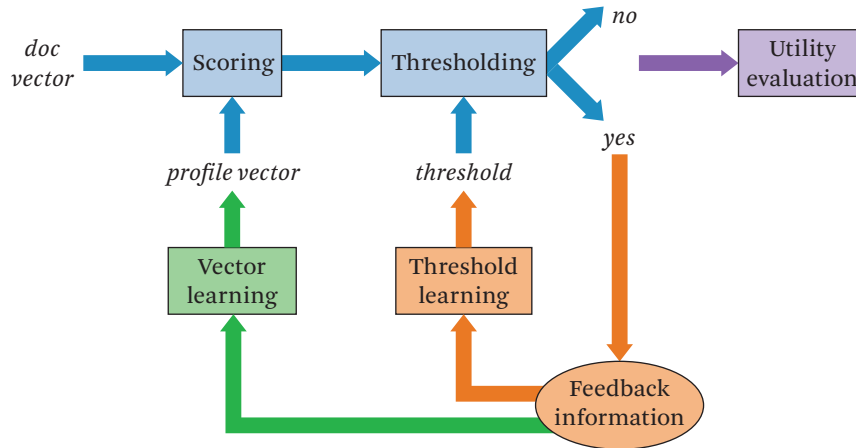
Figure 11.3   The three modules in content-based recommendation and how they fit into its framework.

Figure 11.4   Information available to the content-based recommender system.

There are some interesting challenges in threshold learning. Figure 11.4 depicts the type of data that you can collect in the filtering system. We have the scores and the status of relevance. The first document has a score 36.5 and it's relevant. The second one is not relevant. We have many documents for which we don't know the status, since their scores are less than θ and are not shown to the user for judging. Thus, the judged documents are not a random sample; it's biased or censored data, which creates some difficulty for learning an optimal θ. Secondly, there are in general very little labeled data and very few relevant data, which make it challenging

for machine learning approaches, which require a large amount of training data. In the extreme case at the beginning, we don't even have any labeled data at all, but the system still has to make a decision.

This issue is called the **exploration-exploitation** tradeoff. This means we want to explore the document space to see if the user might be interested in the documents that we have not yet labeled, but we don't want to show the user too many non-relevant documents or they will be unsatisfied with the system. So how do we do that? We could lower the threshold a little bit and deliver some near misses to the user to see what their response to this extra document is. This is a tradeoff because on one hand, you want to explore, but on the other hand, you don't want to explore too much since you would over-deliver non-relevant information. Exploitation means you would take advantage of the information learned about the user. Say you know the user is interested in this particular topic, so you don't want to deviate that much. However, if you don't deviate at all, then you don't explore at all, and you might miss the opportunity to learn another interest of the user. Clearly, this is a dilemma and a difficult problem to solve.

Why don't we just use the empirical utility optimization strategy to optimize $\mathcal{U}$? The problem is that this strategy is used to optimize the threshold based on historical data. That is, you can compute the utility on the training data for each candidate score threshold, keeping track of the highest utility observed given a θ. This doesn't account for the exploration that we just mentioned, and there is also the difficulty of biased training samples. In general, we can only get an upper bound for the true optimal threshold because the threshold might be lower than we found; it's possible that some of the discarded items might actually be interesting to the user. So how do we solve this problem? We can lower the threshold to explore a little bit. We'll discuss one particular approach called **beta-gamma threshold learning** [Zhai et al. 1998].

The basic idea of the beta-gamma threshold learning algorithm is as follows. Given a ranked list of all the documents in the training database sorted by their scores on the x-axis, their relevance, and a specific utility $\mathcal{U}$, we can plot the utility value at each different cutoff position θ. Each cutoff position corresponds to a score threshold. Figure 11.5 shows this configuration and how a choice of α determines a cutoff point between the optimal and the zero utility points, and how β and γ help us to adjust α dynamically according to the number of judged examples in the training database. The optimal point θ_{opt} is the point when we would achieve the maximum utility if we had chosen this threshold. The θ_{zero} threshold is the zero utility threshold. Between these two θ values give us a safe point to explore the potential cutoff values. As one can see from the formula, the threshold will be
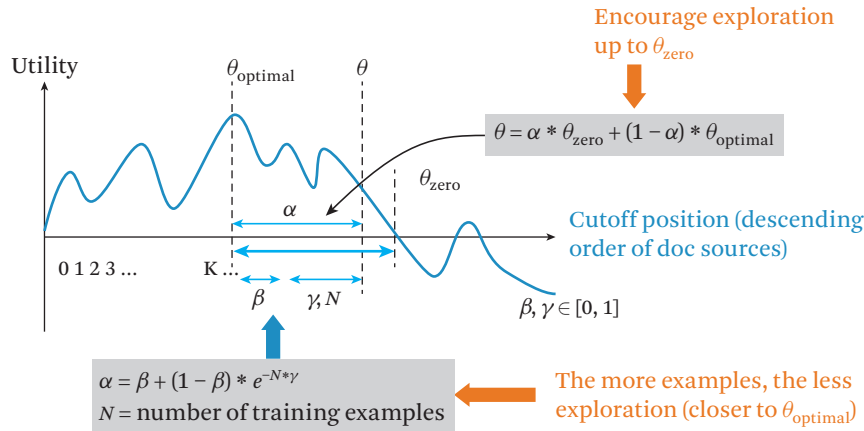
Figure 11.5   Beta-gamma threshold learning to set the optimal value of θ.

just the interpolation of the zero utility threshold and the optimal threshold via the interpolation parameter α.

Now the question is how we should set α and deviate from the optimal utility point. This can depend on multiple factors and one way to solve the problem is to encourage this threshold mechanism to explore only up to the θ_{zero} point (which is still a safe point), but not necessarily reach all the way to it. Rather, we're going to use other parameters to further define α's value given some additional information.

The β parameter controls the deviation from θ_{opt}, which can be based on our previously observed documents (i.e., the training data). What's more interesting is the γ parameter which controls the influence of the number of examples in the training data set, N. As N becomes greater, it encourages *less* exploration. In other words, when N is very small, the algorithm will try to explore more, meaning that if we have seen only a few examples, we're not sure whether we have exhausted the space of interest. But, as we observe many data points from the user, we feel that we probably don't have to explore as much. This gives us a dynamic strategy for exploration: the more examples we have seen, the less exploration we are going to do, so the threshold will be closer to θ_{opt}.

This approach has worked well in some empirical studies, particularly on the TREC filtering tasks. It's also convenient that it welcomes any arbitrary utility function with an appropriate lower bound. It explicitly addresses the exploration-exploration tradeoff, and uses θ_{zero} as a safeguard. That is, we're never going to explore further than the zero utility point. If you take the analogy of gambling, you

don't want to risk losing money, so it's a "safe" strategy in that sense. The problem is, of course, that this approach is purely heuristic and the zero utility lower bound is often too conservative in practice. There are more advanced machine learning projects that have been proposed for solving these problems; it is actually a very active research area.

11.2 Collaborative Filtering

In collaborative filtering, a system makes decisions for an individual user based on judgements of other users (hence it is "collaborative"). The basic idea is to infer individual interests or preferences based solely on similar users. Given a user, collaborative filtering finds a set of similar users. Based on the set of similar users, it predicts the current user's preferences.

This method makes some assumptions.

- Users with a common interest will have similar preferences
- Users with similar preferences share the same interest

For example, if a user has an interest in information retrieval, they might favor papers published in SIGIR. If users favor SIGIR papers, then they might have an interest in IR. The text content of items doesn't matter! This is in sharp contrast to the previous section, where we looked at item similarity through content-based filtering.

Here, we will infer an individual's interest based on other similar users. The general idea is displayed in Figure 11.6: given a user u, we will rank other users based on similarity, $u_1, \ldots, u_m$. We then predict user preferences based on the preferences of these m other users. The preference is on a common set of items $o_1, \ldots, o_n$. If we arrange the users and objects into a matrix X, we can consider the user u_i and the object o_j as the point (u_i, o_j) in the matrix. If we have a judgment by that user for that object, the element in that position would be the user rating X_{ij}.

Again, note that the exact content of each item doesn't matter at all. We only consider the *relationship* between the users and the items. This makes this approach very general since it can be applied to any items—not just text documents. Those items could be movies or products and the users could give ratings (e.g.) one through five. Some users have watched movies and rated them, but most movies for a given user are unrated (since it's unlikely that a user has examined all items in the object space). Thus, many item entries have unknown values and it is the job of
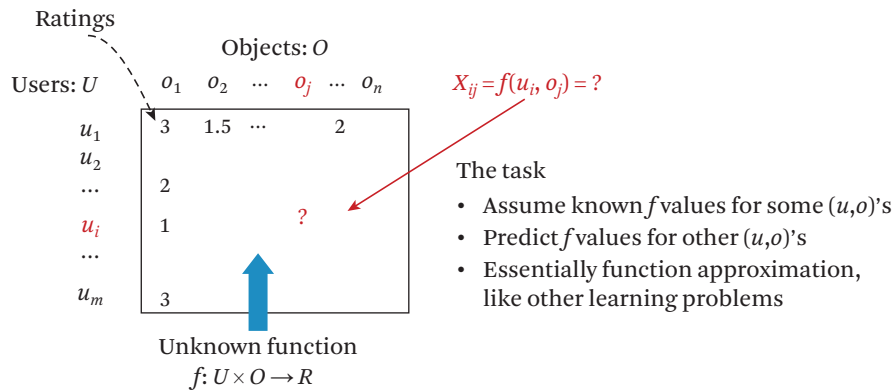
Figure 11.6   **Collaborative filtering viewed as an $m \times n$ matrix with partially filled indices, representing user judgements.**

collaborative filtering to infer the value of a element in this matrix based on other known values.

One other assumption we have to make is that there are a sufficiently large number of user preferences available to us. For example, we need an appreciable number of ratings by users for movies that indicate their preferences for those particular movies. If we don't have sufficient data, there will be a data sparsity problem and that's often called the *cold start problem*.

We assume an unknown function $f(\cdot, \cdot)$, that maps a user and object to a rating. In the matrix X, we have observed there are some output values of this function and we want to infer the value of this function for other pairs that don't have values. This is very similar to other machine learning problems, where we would know the values of the function on some training data and we hope to predict the values of this function on some unseen test data. As usual, there are many approaches to solving this problem. In fact, there is a major conference specifically dedicated to this problem.

We will discuss what is called a *memory-based approach*. When we consider a particular user, we're going to try to retrieve the relevant (i.e., similar) users to the current user. Then, we use those users to predict the preference of the current user. Let n_i be the average rating of all objects by user u_i. We need n_i so we can normalize the ratings of objects by this user by subtracting the average rating from all the ratings. This is necessary so that the ratings from different users will be comparable; some users might be more generous and generally give higher ratings

while others might be more critical and have a lower average rating. So, their ratings can not be directly compared with each other or aggregated together, which is why we first normalize.

Let u_a be the user that we are interested in recommending items to (the "active" user). In particular, we are interested in recommending o_j to u_a. The idea here is to look at whether similar users to this user have liked this object or not. Mathematically, the predicted rating of this user on this object is a combination of the normalized ratings of different users. We're picking a sum of all the users, but not all users contribute equally to the average; each user's weight controls the influence of a user on the prediction. Naturally, the weight is related to the similarity between u_a and a particular user, u_i. The more similar they are, the more contribution we would like user u_i to make in predicting the preference of u_a.

We have the following formulas. First, using the normalized ratings

$$V_{ij} = X_{ij} - n_i \tag{11.2}$$

we can write the predicted normalized rating

$$\hat{V}_{aj} = k \cdot \sum_{i=1}^{m} w(u_a, u_i) \cdot V_{ij}, \tag{11.3}$$

where $w(\cdot, \cdot)$ is the similarity function and k is the normalizer

$$k = \frac{1}{\sum_{i=1}^{m} w(u_a, u_i)} \tag{11.4}$$

that ensures $\hat{V}_{aj} \in [0, 1]$. Once we have the predicted normalized rating, we transform it into the rating range that u_a uses:

$$\hat{X}_{aj} = \hat{V}_{aj} + n_a. \tag{11.5}$$

If we want to write a program to implement this collaborative filtering, we still face the problem of determining the weighting function. Once we know this, then the formula is very easy to implement. Specific definitions of the weighting function define the different interpretations of the collaborative filtering rating estimate. As you may imagine, there are many possibilities of similarity functions. One popular approach is the Pearson Correlation Coefficient:

$$w_p(u_a, u_i) = \frac{\sum_j (X_{aj} - n_a)(X_{ij} - n_i)}{\sqrt{\sum_j (X_{aj} - n_a)^2 \sum_j (X_{ij} - n_i)^2}}.$$

This is a sum of a common range of judged items and measures whether the two users tended to all give higher ratings to similar items, or lower ratings to similar items. Another measure is the cosine measure, which treats the rating vectors as vectors in the vector space, and measures the cosine of the angle between the two vectors:

$$w_c(u_a, u_i) = \frac{\sum_j x_{aj} x_{ij}}{\sqrt{\sum_j x_{aj}^2 \sum_j x_{ij}^2}}.$$

As we've discussed previously in this book, this measure has been used in the vector space model for retrieval. We'll also see how it is used in clustering in Chapter 14.

In all these cases, note that the user similarity is based on their preferences on items, and we did not actually use any content information of these items. It didn't matter what these items are; they can be movies, books, products, or text documents. This allows such an approach to be applied to a wide range of problems.

There are some ways to improve this approach, most of which consider the user similarity measure. There are some practical issues to deal with here as well; for example, there will be many missing values. We could set them to default values or the average ratings of other users. That will be a simple solution, but there are advantages to approaches that can actually try to predict those missing values and then use the predicted values to improve the similarity measure. In fact, in memory-based collaborative filtering, we can predict judgements with missing values. As you can imagine, we could apply an iterative approach where we first do some preliminary prediction and then use the predicted values to further improve the similarity function.

Another idea which is quite similar to the idea of IDF that we have seen in text research, is called the *inverse user frequency* or IUF. Here, the idea is to look at where the two users share similar ratings. If the item is a popular item that has been viewed by many people, it's not as informative. Conversely, if it's a rare item that has not been viewed by many users, then it says more about their similarity, emphasizing more on similarity of items that are not viewed by many users.

Let's summarize our discussion of recommender systems. In some sense, the filtering task of recommendation is easy and in another sense the task is rather difficult. It's easy because the user expectation is low. That is, any recommendation is better than none. The system takes initiative to push the information to the user, so the user doesn't really make an effort. Unless you recommend only noisy items or useless documents, any information would be appreciated. Thus in that sense, it's easy.

Filtering can also be considered a much harder task because you have to make a binary decision and can't afford waiting for many items to enhance your belief that one is better than others. Let's think about news filtering as soon as the system detects the news articles: you have to decide whether the news would be interesting to a user. If you wait for a few days, even an accurate recommendation of the most relevant news is not interesting. Another reason why it's hard is due to data sparseness. If you think of this as a learning problem in collaborative filtering, for example, it's purely based on learning from the past ratings. If you don't have many ratings, there's really not much you can do. As we mentioned, there are strategies that have been proposed to solve the problem. For example, we can use more user information to assess their similarity instead of just using the item preferences.

We also talked about the two strategies for a filtering task; one is content-based where we look at item content similarity. The other is user similarity, which is collaborative filtering. We talked about push vs. pull as two strategies for getting access to the text data. Recommender systems aid users in push mode whereas search engines assist users in pull mode.

11.3 Evaluation of Recommender Systems

In evaluation setups for collaborative filtering, we have a set P of pairs of predicted ratings $\hat{r}$ and actual ratings r across all user-item pairs. A very common measure called root-mean squared error (RMSE) has a mathematical formula that follows its name:

$$RMSE(P) = \sqrt{\frac{1}{|P|} \sum_{(\hat{r},r) \in P} (\hat{r} - r)^2}. \tag{11.6}$$

A similar metric is mean absolute error (MAE), defined as

$$MAE(P) = \sqrt{\frac{1}{|P|} \sum_{(\hat{r},r) \in P} |\hat{r} - r|}. \tag{11.7}$$

Due to the square in RMSE, RMSE is more affected by larger errors. The similarity between RMSE and MAE should remind the reader of the differences between gMAP and MAP. Both these measures quantify the difference in values between $\hat{r}$ and the true rating r. Using such a measure is natural when we have ratings on some ordinal scale.

One important note is that these measures only capture the accuracy of predicted ratings; in an actual recommender system, the top k elements are recommended to the user. Despite having a low RMSE or MAE, it may be possible that the top

documents may not actually be relevant to the user. In the extreme case where $k = 1$, we may recommend the one element that received an erroneous high score. To combat this issue, we can rank all ratings for a particular user and then use an information retrieval metric such as NDCG to view the list as a whole when compared to the true rating r.

In information filtering tasks, a system pushes items to a user if it thinks the user would like the item. In this case, there are no explicit ratings for each item, but rather a relevant *vs* not-relevant judgement. Once the user sees the item, the user then determines if the suggestion was a good one. For a single user, it's easy to see that we can use some evaluation metrics from information retrieval, as discussed in Chapter 9.

Since items are pushed to users as soon as they become available, we can't use any rank-sensitive measures. Still, we can examine the set of all documents pushed to the user in some time period and compute statistics like precision, recall, F_1 score, and other similar measures.

In a true information filtering system, there will be many users who receive all pushed items. A simple way to aggregate scores would be to take an average of the individual user metrics, e.g., average F_1 score across all users. However, this may not be the best measure if some users have more recommendations than others. Since θ is set on a per-user basis, different users will aggregate different numbers of seen documents. Furthermore, in a real system users may not all join at the same time, or some users may have more training data available than others if they have more complete user profiles or if they have been in the system longer. For these reasons, it could be advantageous to instead take a weighted average of user metrics as an overall metric. The weight may be assigned such that all weights sum to one, and each user's weight is determined by that user's total number of judgements.

It may also be interesting to compute the precision or recall over time, where time is measured as the number of documents that the filtering system has seen. A variant of this is to measure time based on the number of elements judged by the user (which are only those elements that are shown to the user). Ideally, as the number of documents increases, the overall precision (or precision of the last k documents) should increase. Once the precision has reached a flat line, we are most likely at θ_{opt} given the current system setup. Of course, this learning-over-time evaluation can also be generalized to multiple users in the same way as previously discussed.

As a final note for both recommender system types, it is valuable to find those users affecting the evaluation metric the most. That is, are there any outliers that cause the evaluation metric to be significantly lower than expected? If so, these

users can be further investigated and the overall system may be adjusted to ensure their satisfaction. Of course, these outliers depend heavily on the evaluation metric used, so using multiple metrics will give the most complete view of user satisfaction.

Bibliographic Notes and Further Reading

For further reading, we suggest that the reader consult Herlocker et al. [2004], Shani and Gunawardana [2011]. Ricci et al. [2010] is a comprehensive resource for learning more about recommender systems in general. More information about the beta-gamma threshold setting algorithm can be found in Zhai et al. [2000]. A description of memory-based collaborative filtering algorithm can be found in Breese et al. [1998], which also provides a comparison of different collaborative filtering algorithms. A more recent comparison of multiple collaborative filtering algorithms can be found in the Cacheda et al. [2011].

Exercises

11.1. When delivering content to a user, it's important to not deliver duplicate information. Describe a strategy that doesn't deliver a document to a user if it is a duplicate. Then, describe a strategy that doesn't deliver a document if it is too similar to a previously delivered document. Ensure that your methods are space efficient. That is, don't store the full text of every document seen!

11.2. A user may be interested in a few diverse information needs; a user may enjoy both romance movies and action movies. Does this pose any problem to a recommendation system? If so, suggest how this issue may be addressed. If not, explain how the existing structure of the filtering system handles this.

11.3. In the introduction to this chapter, we noted that combining content-based filtering and collaborative filtering could yield an optimal recommender application. Suggest some ways to combine these two methods, explaining your intuition.

11.4. Beta-gamma thresholding compares search engine scores to a cutoff parameter θ. Consider the following argument: "Using a single fixed point θ does not allow query comparability since query scores are not normalized." Defend or refute this statement.

11.5. Use META's search engine to implement beta-gamma thresholding. Use a dataset with relevance judgements as the user preferences, and a small number of

related queries as the initial user profile. Shuffle document IDs and examine them sequentially to simulate a data stream.

11.6. We described a linear utility function $\mathcal{U}$ for scoring the usefulness of the filtering system's output. Give an example of a nonlinear utility function. How does this compare to the version introduced in this chapter?

11.7. In our description of beta-gamma thresholding, we showed the user a document if $\text{score}(d) > \theta$. Can you think of an alternate scoring strategy? Your strategy may include additional features about the document or user if they are available. If possible, evaluate your alternate scoring function in META.

11.8. Given n users and m objects, determine the running time of recommending one item to each user using collaborative filtering. Keep in mind the running time of a particular similarity algorithm—first, assume we are using cosine similarity. Would using the Pearson Correlation Coefficient instead change the running time?

11.9. As a method to improve the running time of collaborative filtering, we can consider only the top-k most active users, where $k \ll n$. An active user is one that has given a large amount of ratings. What is a potential problem with this method?

11.10. Imagine we run a collaborative filtering system on a database of movies. One movie producer creates many accounts on the collaborative filtering system and only gives high ratings to movies produced by their own company and low ratings to all others. Is the collaborative filtering system described in this chapter susceptible to such spam? If so, brainstorm some anti-spamming measures.

11.11. A collaborative filtering system treats each item independently, even if they are almost identical. For example, movies in a trilogy would be treated separately. Give an example where this is desired and give an example where this is problemental.

11.12. What steps need to be taken when new elements are added to the collaborative filtering dataset (e.g., new books are released)? Do we have the same cold start problem as before?

11.13. Is there some relation of RMSE and MAE to L_1 and L_2 error? Recall that L_1 error is defined as

$$\sum_{i=1}^{n} |y_i - f(x_i)|$$

and L_2 error is defined as

$$\sum_{i=1}^{n}(y_i - f(x_i))^2,$$

where y_i represents the true value of a prediction, $f(x_i)$ is the prediction itself, and n is the total number of predictions.

PART

III

TEXT DATA ANALYSIS

Overview of Text Data Analysis

In the previous chapters that we grouped under Part II, we have covered techniques for text data access, which is logically an initial step for processing text data for the purpose of both significantly reducing the size of the data set to be further processed (either by humans or machines) and filtering out any obvious noise in the text data so as to focus on the truly relevant data to a particular application problem. In Part III of the book, starting from this chapter, we will cover techniques for further processing relevant text data so as to extract and discover useful actionable knowledge that can be directly used for decision making or supporting a user's task. One difference between Part II and Part III is the extent to which the information need of a user, or equivalently, a specific application, is emphasized. Specifically, since the purpose of text data access is, in general, to connect users with the right information at the right time so that they can further digest and exploit the relevant text data (with or without the help of additional text analysis techniques), the concept of information need and the closely related concept of relevance play an important role in all the techniques covered in Part II. For example, queries play an essential role in any search engines, and accurate modeling of a user's interest and information need plays an equally important role in any recommender systems. In Part III, however, we generally can assume that the text data to be considered are all relevant, so we will see that we no longer emphasize the information need so much, but instead will emphasize the goal toward understand the content of text in more detail and find any interesting patterns in text data such as topical trends or sentiment polarity so as to eventually extract and discover actionable knowledge directly useful for finishing a user task. As such, we will attempt to view the process of text data analysis as a special case of the general process of data mining, where users would use various data mining operators to probe and analyze the data in an interactive manner. Multiple operators may be combined. Specifically, we will view humans as "subjective sensors" of our world, and text data as data generated

by such subjective sensors, making text data more similar to other kinds of data generated by objective machine sensors and enabling us to naturally discuss how to jointly analyze text and non-text data together.

However, we must point out that the separation of the text data access stage and text data analysis stage, thus also the separation of Part II and Part III in the book, is somewhat artificial since in a sophisticated application, these two stages are often interleaved, and an iterative process involving both stages is often followed. For example, after a user has zoomed into a set of relevant documents and performed an analysis task (such as clustering of documents into topical clusters), the user may also choose to further search inside a particular cluster to further zoom into a specific subset of documents, and additional analysis operators such as sentiment analysis may then be applied to this newly obtained smaller subset. Moreover, techniques from both Part II and Part III can often be combined to provide more useful functions to users (e.g., summarization can be naturally combined with a search engine or recommender system), and they may enhance each other (e.g., the term weighitng methods discussed in Part II are also very useful for many tasks such as clustering, categorization, and summarization in Part III, and clustering in Part III can be useful for improving retrieval algorthm covered in Part II). Nevertheless, we have chosen to separate them so as to allow the readers to see a meaningful overall picture of all the techniques we covered and their high-level relations.

12.1   Motivation: Applications of Text Data Analysis

The importance of text data to our lives can be easily seen from the fact that we all process a lot of text data on a daily basis. In most cases, however, the computers only play a minor role in the entire process of making use of text data; for example, we use search engines frequently, but once we find relevant documents, the further processing of the found documents is generally done manually. Such a manual process is acceptable when the amount of text data to be processed is small, the application task does not demand a fast response, and when we have the time to digest text data. However, as the amount of text data increases, the manual processing of text data would not be feasible or acceptable, especially for time-critical applications. Thus, it becomes increasingly essential to develop advanced text analysis tools to help us digest and make use of text data effectively and efficiently.

In general, we may distinguish two kinds of text analysis applications. One kind is those that can replace our current manual labor in digesting text content; they help improve our productivity, but do not do anything beyond what we humans can

do. For example, automatic sorting of emails would save us a lot of time. The other kind is those that can discover knowledge that we humans may not be able to do even if we have "sufficient" time to read all the text data. For example, an intelligent biomedical literature anlayzer may reveal a chain of associations of genes and diseases by synthesizing gene-gene relations and gene-disease relations scattered in many different research articles, thus suggesting a potential opportunity to design drugs targeting some of the genes for treatment of a disease.

Due to the broad coverage of knowledge in text data and our reliance on text data for communications, it is possible to imagine text analysis applications in virtually any domain. Below are just a few specific examples that may provide some application contexts for understanding the text analysis techniques covered in the subsequent chapters.

One important application domain of text analysis is business intelligence. For example, product managers may be interested in hearing customer feedback about their products, knowing how well their products are being received as compared to the products of competitors. This can be a good opportunity for leveraging text data in the form of product reviews on the Web. If we can develop and master text mining techniques to tap into such an information source to extract the knowledge and opinions of people about these products, then we can help these product managers gain business intelligence or gain feedback from their customers.

Another important application domain is scientific research, where timely digestion of knowledge encoded in literature articles is essential. Scientists are also interested in knowing the trends of research topics or learning about discoveries in fields related to their own. This problem is especially important in biology research—different communities tend to use different terminologies, yet they're stating very similar problems. How can we integrate the knowledge that is covered in different communities (using different vocabularies) to help study a particular problem? Answering such a question speeds up scientific discovery. There are many more such examples where we can leverage text data to discover usable knowledge to optimize our decision-making processes.

Yet another broad category of applications is to leverage social media to optimize decision making. In general, we can imagine building an intelligent sensor system to "listen" to all the text data produced in real time, especially social media data such as tweets which report real-world events almost in real time, and monitor interesting patterns relevant to an application. For example, performing sentiment analysis on people's opinions about policies can help better understand society's response to a policy and thus potentially improve the policy if needed. Disaster response and management would benefit early discovery of any

warning signs of a natural disaster, which is possible through analyzing tweets in real time.

In general, "big data" can enhance our perception. Just as a microscope allows us to see things in the "micro world," and a telescope allows us to see things far away, in the era of big data, we may envision a "datascope" would allow us to "see" useful hidden knowledge buried in large amounts of data. As a special kind of data, text data presents unique opportunities to help us "see" virtually all kinds of knowledge we encode in text, especially knowledge about people's opinions and thoughts, which may not be easy to see in other kinds of data.

12.2   Text vs. Non-text Data: Humans as Subjective Sensors

For the purpose of data mining, it is useful to view text data as data generated by humans as subjective sensors. We can compare humans as subjective sensors to physical sensors, such as a network sensor or a thermometer. Any sensor monitors the real world in some way; it senses some signal from the real world and then reports the signal as various forms of data. For example, a thermometer would sense the temperature of the real world and then report the temperature as data in a format like Fahrenheit or Celsius. Similarly, a geo-sensor would sense its geographical location and then report it as GPS coordinates. Interestingly, we can also think of humans as subjective sensors that observe the real world from their own perspective. Humans express what they have observed in the form of text data. In this sense, a human is actually a subjective sensor of what is happening in the world, who then expresses what's observed in the form of data—text data. This idea is illustrated in Figure 12.1.
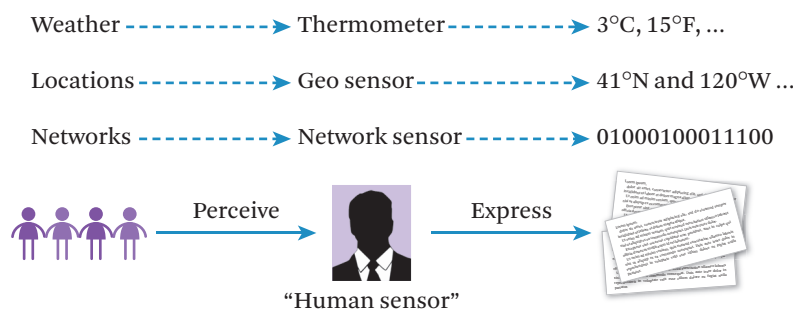
Figure 12.1   **Humans as subjective sensors.**
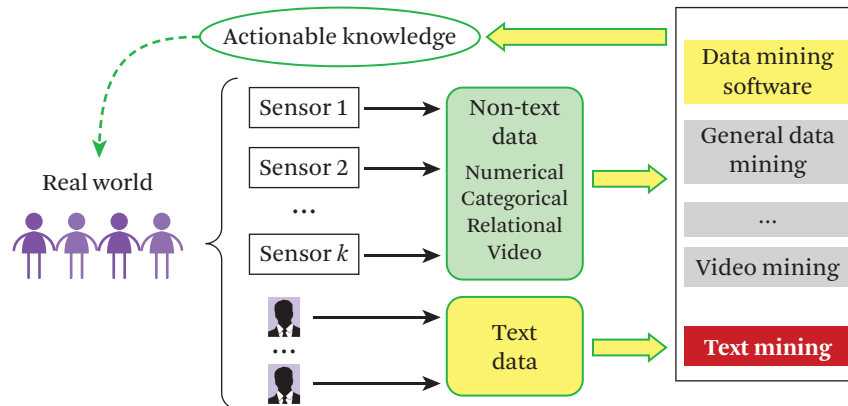
Figure 12.2   **The general problem of data mining.**

Looking at text data in this way has an advantage of being able to integrate all types of data together, which is instrumental in almost all data mining problems. In a data mining scenario, we would be dealing with data about our world that are related to a particular problem. Most problems would be dealing with both non-text data and text data. Of course, the non-text data are usually produced by physical sensors and can exist in many different formats such as numerical, categorical, or relational. It could even be multimedia data like video or speech. Text data is also very important because they contain knowledge about users, especially preferences and opinions.

By treating text data as data observed from human sensors, we can examine all this data together in the same framework. The data mining problem can then be defined as to turn all such data into actionable knowledge that we can take advantage of to change the world for the better. This is illustrated in Figure 12.2.

Inside of the data mining module, you can also see we have a number of different kinds of mining algorithms. Of course, for different kinds of data, we generally need different algorithms, each suitable for mining a particular kind of data. For example, video data would require computer vision to understand video content, which would facilitate more effective general mining. We also have many general algorithms that are applicable to all kinds of data; those algorithms, of course, are very useful, but for a particular kind of data, in order to achieve the best mining results, we generally would still need to develop a specialized algorithm. This part of the book will cover specialized algorithms that are particularly useful for mining and analyzing text data.
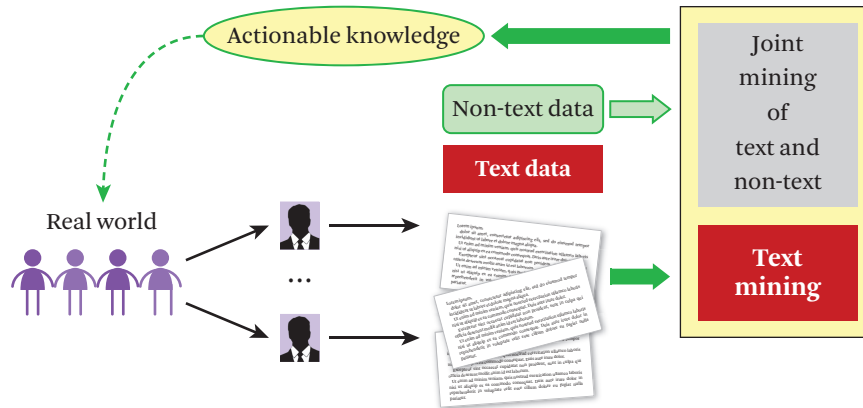
Figure 12.3 **Text mining as a special case of data mining.**

Looking at the text mining problem more closely in Figure 12.3, we see that the problem is similar to general data mining except that we'll be focusing more on text. We need text mining algorithms to help us turn text data into actionable knowledge that we can use in the real world, especially for decision making or for completing whatever tasks require text data support. Many real-world problems of data mining also tend to have other kinds of data that are non-textual. So, a more general picture would be to include non-text data as well. For this reason, we might be concerned with joint mining of text and non-text data. With this problem definition we can now look at the landscape of the topics in text mining and analytics.

12.3 Landscape of text mining tasks

In this section, we provide a high-level description of the landscape of various text mining tasks, which also serves as a roadmap for the topics to be covered in the subsequent chapters in Part III.

Figure 12.4 shows the process of generating text data in more detail. Specifically, a human sensor or human observer would look at the world from some perspective. Different people would be looking at the world from different angles and they'll pay attention to different things. The same person at different times might also pay attention to different aspects of the observed world. Each human—a sensor— would then form their own view of the world. This would be different from the real world because the perspective that the person has taken can often be biased. The observed world can be represented as (for example) entity-relation graphs or using
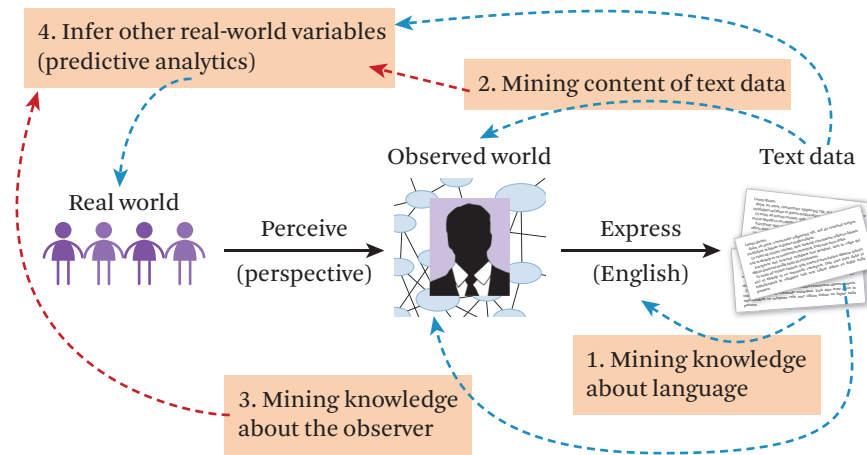
Figure 12.4   **Mining different types of knowledge from text data.**

a knowledge representation language. This is basically what a person has in mind about the world.

As the users of human-generated data, we will never exactly know what the real world actually looked like at the moment when the author made the observation. The human expresses what is observed using a natural language such as English: the result is text data. In some cases, we might have text data of mixed languages or different languages.

The main goal of text mining is to reverse this process of generating text data and uncover various knowledge about the real world as it was observed by the human sensor. As illustrated in Figure 12.4, we can distinguish four types of text mining tasks.

Mining knowledge about natural language.   Since the observed text is written in a particular language, by mining the text data, we can potentially mine knowledge about the usage of the natural language itself. For example, if the text is written in English, we may be able to discover knowledge about English, such as usages, collocations, synonyms, and colloquialisms.

Mining knowledge about the observed world.   This has much to do with mining the content of text data, focusing on extracting the major statements in the text data and turn text data into high quality information about a particular aspect of the world that we're interested in. For example, we can discover everything that has been said about a particular person or a particular entity.

This can be regarded as mining content to describe the observed world in the author's mind.

Mining knowledge about the observers (text producers).   Since humans are subjective sensors, the text data expressed by humans often contain subjective statements and opinions that may be unique to the particular human observer (text producers). Thus, we can potentially mine text data to infer some properties of the authors that produced the text data, such as the mood or sentiment of the person toward an issue. Note that we distinguish mining knowledge about the observed world from mining knowledge about the text producer because text data is generally a mixture of objective statements about the world observed and subjective statements or comments reflecting the text producer's opinions and beliefs, and it is possible and useful to extract each separately.

Inferring knowledge about properties of the real world. On the left side of the figure, we illustrate that text mining can also allow us to infer values of interesting real world variables by leveraging the correlation of the values of such variables and the content in text data. For example, there may be some correlation between the stock price changes on the stock market and the events reported in the news data (e.g., a positive earnings report of a company may be correlated with the increase of the stock price of the company). Such correlations can be leveraged to perform text-based forecasting, where we use text data as a basis for prediction of other variables that may only be remotely related to text data (e.g., prediction of stock prices). Inference about unknown factors that affect decision making can have many applications, especially if we can make predictions about future events (i.e., text-based predictive analytics).

   Note that when we infer other real-world variables, it is often possible and beneficial to leverage the results of all kinds of text mining algorithms to generate more effective features for use in a predictive model than the basic features we can generate directly from the original text data. For example, if we can mine text data to discover topics, we would be able to use topics (i.e., a set of semantically related words), rather than individual words, as features. Since topics can address the issue of word sense ambiguity and variations of word usages when discussing a topic, such high-level semantic features can be expected to be more effective than word-level features for prediction. Another example is to predict what products may be liked by a user based on

what the user has said in text data (e.g., reviews), in which case, the results from mining knowledge about the observer would clearly be very useful for prediction.

Futhermore, non-text data can be very important in predictive analysis. For example, if you want to predict stock prices or changes of stock prices, the historical stock price data are presumably the best data to use for prediction even though online discussions, news articles, or social media, may also be useful for further improvement of prediction accuracy by contributing additional effective features computed based on text data (which would be combined with non-text features).

Non-text data can also be used for analyzing text by supplying context, thus opening up many interesting opportunities to mine context-sensitive knowledge from text data, i.e., associating the knowledge discovered from text data with the non-text data (e.g., associating topics discovered from text with time would generate temporal trends of topics). When we look at the text data alone, we'll be mostly looking at the content or opinions expressed in the text. However, text data generally also has context associated with it. For example, the time and the location of the production of the text data are both useful "metadata" values of a text document. This context can provide interesting angles for analyzing text data; we might partition text data into different time periods because of the availability of the time. Now, we can analyze text data in each time period and make a comparison. Similarly, we can partition text data based on location or any other metadata that's associated with it to form interesting comparisons in those areas. In this sense, non-text data can provide interesting angles or perspectives for text data analysis. It can help us make context-sensitive analysis of content, language usage, or opinions about the observer or the authors of text data. We discuss joint analysis of text and non-text data in detail in Chapter 19.

This is a fairly general landscape of the topics in text mining and analytics. In this book, we will selectively cover some of those topics that are representative of the different kinds of text mining tasks. Chapters 2 and 3 already covered natural language processing and the basics of machine learning, which allow us to understand, represent, and classify text data—important steps in any text mining task. In the remaining chapters of Part III of the book, we will start to enumerate different text mining tasks that build upon the NLP and IR techniques discussed earlier.

First, we will discuss how to mine word associations from text data (Chapter 13), revealing lexical knowledge about language. After word association mining, we will

look at clustering text objects (Chapter 14). This groups similar objects together, allowing exploratory analysis, among many other applications. Chapter 15 covers text categorization, which expands on the introduction to machine learning given in Chapter 2. We also explore different methods of text summarization (Chapter 16). Next, we'll discuss topic mining and analysis (Chapter 17). This is only one way to analyze content of text, but it's very useful and used in a wide array of applications. Then, we will introduce opinion mining and sentiment analysis. This can be regarded as one example of mining knowledge about the observer, and will be covered in Chapter 18. Finally, we will briefly discuss text-based prediction problems where we try to predict some real-world variable based on text data and present a number of cutting-edge research results on how to perform joint analysis of text and non-text data (Chapter 19).

Word Association Mining

In this chapter, we're going to talk about how to mine associations of words from text. This is an example of knowledge about the natural language that we can mine from text data. We'll first talk about what word association is and then explain why discovering such relations is useful. Then, we'll discuss some general ideas about how to mine word associations.

In general, there are two types of word relations; one is called a **paradigmatic relation** and the other is a **syntagmatic relation**. Word w_a and w_b have a paradigmatic relation if they can be substituted for each other. That means the two words that have paradigmatic relation would be in the same semantic class, or syntactic class. We can replace one with another without affecting the understanding of the sentence. Chapter 14 gives some additional ideas not discussed in this chapter about how to group similar terms together.

As an example, the words *cat* and *dog* have a paradigmatic relation because they are in the same word class: animal. If you replace *cat* with *dog* in a sentence, the sentence would still be (mostly) comprehensible. Similarly, *Monday* and *Tuesday* have a paradigmatic relation.

The second kind of relation is called a syntagmatic relation. In this case, the two words that have this relation can be combined with each other. Thus, w_a and w_b have a syntagmatic relation if they can be combined with each other in a grammatical sentence—meaning that these two words are semantically related. For example, *cat* and *sit* are related because a cat can sit somewhere (usually anywhere they please). Similarly, *car* and *drive* are related semantically because they can be combined with each other to convey some meaning. However, we cannot replace *cat* with *sit* in a sentence or *car* with *drive* in the sentence and still have a valid sentence. Therefore, the previous pairs of words have a syntagmatic relation and not a paradigmatic relation.

These two relations are in fact so fundamental that they can be generalized to capture basic relations between units in arbitrary sequences. They can be generalized to describe relations of any items in a language; that is, w_a and w_b don't

have to be words. They could be phrases or entities. If you think about the general problem of sequence mining, then we can think about any units being words. We think of paradigmatic relations as relations that are applied to units that tend to occur in a similar location in a sentence (or a sequence of data elements in general). Syntagmatic relations capture co-occurring elements that tend to show up in the same sequence. So, these two measures are complimentary and we're interested in discovering them automatically from text data.

Discovering such word relations has many applications. First, such relations can be directly useful for improving accuracy of many NLP tasks, and this is because these relations capture some knowledge about language. If you know two words are synonyms, for example, that would help with many different tasks. Grammar learning can be also done by using such techniques; if we can learn paradigmatic relations, then we can form classes of words. If we learn syntagmatic relations, then we would be able to know the rules for putting together a larger expression based on component expressions by learning the sentence structure. Word relations can be also very useful for many applications in text retrieval and mining.

In search and text retrieval, we can use word associations to modify a query for feedback, making search more effective. As we saw in Chapter 7, this is often called query expansion. We can also use related words to suggest related queries to a user to explore the information space. Yet another application is to use word associations to automatically construct a hierarchy for browsing. We can have words as nodes and associations as edges, allowing a user to navigate from one word to another to find information. Finally, such word associations can also be used to compare and summarize opinions. We might be interested in understanding positive and negative opinions about a new smartphone. In order to do that, we can look at what words are most strongly associated with a feature word like *battery* in positive vs. negative reviews. Such syntagmatic relations would help us show the detailed opinions about the product.

13.1   General idea of word association mining

So, how can we discover such associations automatically? Let's first look at the paradigmatic relation. Here, we essentially can take advantage of similar context. Figure 13.1 shows a simple example using the words *dog* and *cat*.

Generally, we see the two words occur in similar context. After all, that is the definition of a paradigmatic relation. On the right side of the figure, we extracted the context of *cat* and *dog* from this small sample of text data. We can have different perspectives to look at the context. For example, we can look at what words occur

Paradigmatic: similar context

My **cat** eats fish on Saturday
His **cat** eats turkey on Tuesday
My **dog** eats meat on Sunday
His **dog** eats turkey on Tuesday
...

cat:

My ___ eats fish on Saturday
His ___ eats turkey on Tuesday
...

dog:

My ___ eats meat on Sunday
His ___ eats turkey on Tuesday
...

Similar **left** context   Similar **right** context   Similar **general** context
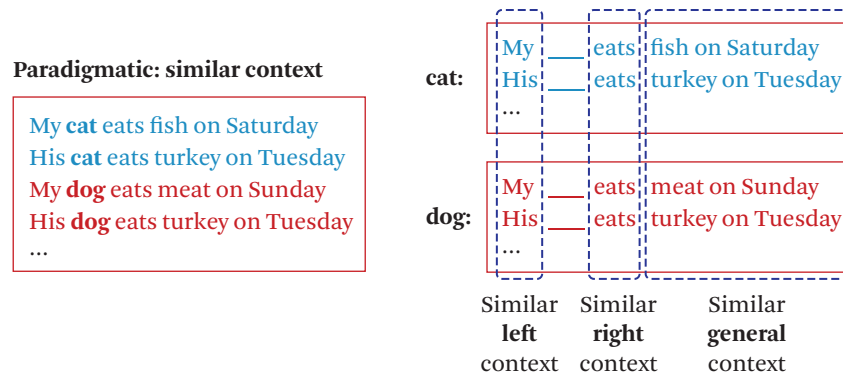
How similar are context ("**cat**") and context ("**dog**")?
How similar are context ("**cat**") and context ("**computer**")?

Figure 13.1   **Intuition for paradigmatic relation discovery.**

in the left part of this context. That is, what words occur before we see *cat* or *dog*? Clearly, these two words have a similar left context. In the same sense, if you look at the words that occur after *cat* and *dog* (the right context), we see that they are also very similar in this case. In general, we'll see many other words that can follow both cat and dog. We can even look at the general context; this includes all the words in the sentence or in sentences around this word. Even in the general context, there is also similarity between the two words. Examining context is a general way of discovering paradigmatic words.

Let's consider the following questions. How similar is the context of *cat* and *dog*? In contrast, how similar are the contexts of *cat* and *computer*? Intuitively, the context of *cat* and the context of *dog* would be more similar than the context of *cat* and *computer*. That means in the first case the similarity value would be high, and in the second, the similarity would be low. This is the basic idea of what paradigmatic relations capture.

For syntagmatic relations, we're going to explore correlated occurrences, again based on the definition of syntagmatic relations. Figure 13.2 shows the same sample of text as the example before. Here, however, we're interested in knowing what other words are correlated with the verb *eats*. On the right side of the figure we've taken away the two words around *eats*. Then, we ask the question, what words tend to occur to the left of *eats*? What words tend to occur to the right of *eats*? Therefore, the question here has to do with whether there are some other words that tend to co-occur with *eats*. For example, knowing whether *eats* occurs in a sentence would

Syntagmatic: correlated occurrences

My **cat** **eats** **fish** on Saturday
His **cat** **eats** **turkey** on Tuesday
My **dog** **eats** **meat** on Sunday
His **dog** **eats** **turkey** on Tuesday
...

My ___ **eats** ___ on Saturday
His ___ **eats** ___ on Tuesday
My ___ **eats** ___ on Sunday
His ___ **eats** ___ on Tuesday
...

What words tend to occur
to the **left** of "**eats**"?

What words
to the **right**?
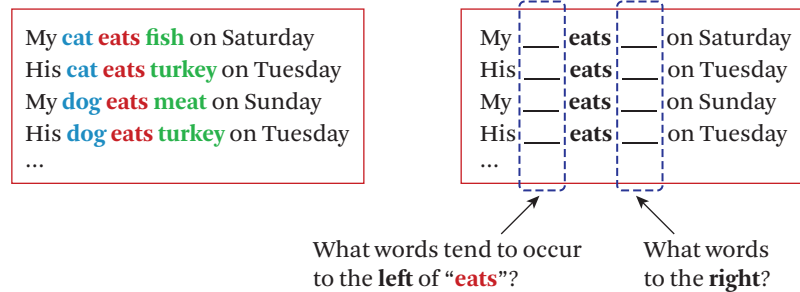
Whenever "**eats**" occurs, what **other words** also tend to occur?
How helpful is the occurrence of "**eats**" for predicting occurrence of "**meat**"?
How helpful is the occurrence of "**eats**" for predicting occurrence of "**text**"?

Figure 13.2   Intuition for syntagmatic relation discovery.

generally help us predict whether *meat* also occurs. This is the intuition we would like to capture. In other words, if we see *eats* occur in the sentence, that should increase the chance that *meat* would also occur.

In contrast, if you look at the question at the bottom, how helpful is the occurrence of *eats* for predicting an occurrence of *text*? Because *eats* and *text* are not really related, knowing whether *eats* occurred in the sentence doesn't really help us predict whether *text* also occurs in the sentence. Essentially, we need to capture the correlation between the occurrences of two words.

In summary, paradigmatic relations consider each word by its context and we can compute the context similarity. We assume the words that have high context similarity will have a high paradigmatic relation. For syntagmatic relations, we will count how many times two words occur together in a context, which can be a sentence, a paragraph, or even a document. We compare their co-occurrences with their individual occurrences. We assume words with high co-occurrences but relatively low individual occurrences will have a syntagmatic relation because they tend to occur together and they don't usually occur alone.

Note that paradigmatic relations and syntagmatic relations are closely related in that paradigmatically related words tend to have a syntagmatic relation with the same word. This fact suggests that we can perform a joint discovery of the two relations.

Left1("cat") = {"my", "his", "big", "a", "the", ...}

cat:

My	___	eats	fish on Saturday
His	___	eats	turkey on Tuesday
...			

Window8("cat") =
{"my", "his", "big", "eats", "fish", ...}

Right1("cat") = {"eats", "ate", "is", "has", ...}

Context = pseudo document = "bag of words"
Context may contain adjacent or non-adjacent words
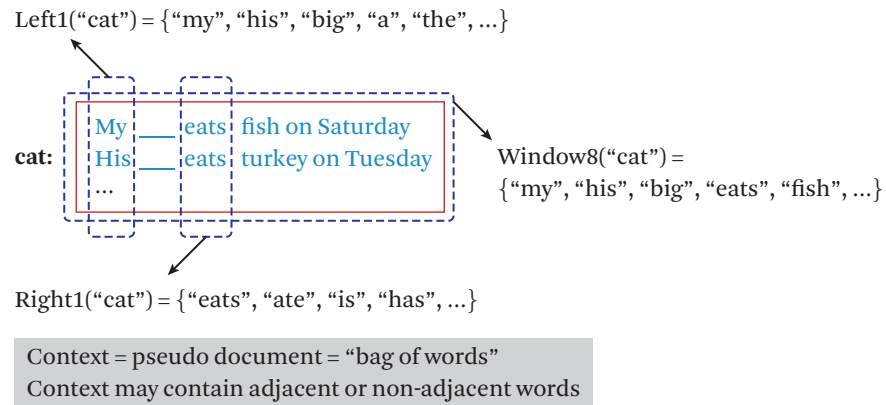
Figure 13.3   **Context of words convey semantics.**

13.2 Discovery of paradigmatic relations

By definition, two words are paradigmatically related if they share a similar context. Naturally, our idea of discovering such a relation is to look at the context of each word and then try to compute the similarity of those contexts.

In Figure 13.3, we have taken the word *cat* out of its context. The remaining words in the sentences that contain *cat* are the words that tend to co-occur with it. We can do the same thing for another word like *dog*. In general, we would like to capture such contexts and then try to assess the similarity of the context of *cat* and the context of a word like *dog*.

The question is how to formally represent the context and define the similarity function between contexts. First, we note that the context contains many words. These words can be regarded as a pseudo document, but there are also different ways of looking at the context. For example, we can look at the word that occurs before the word *cat*. We call this context the left context L. In this case, we will see words like *my*, *his*, *big*, *a*, *the*, and so on. Similarly, we can also collect the words that occur after the word *cat*, which is called the right context R. Here, we see words like *eats*, *ate*, *is*, and *has*. More generally, we can look at all the words in the window of text around the target word. For example, we can take a window of eight words around the target word.

These word contexts from the left or from the right form a bag of words representation. Such a word-based representation would actually give us a useful way to define the perspective of measuring context similarity. For example, we can compare only the L context, the R context, or both. A context may contain adjacent

Sim("cat", "dog") =
    Sim(**Left1**("cat"), **Left1**("dog"))
        + Sim(**Right1**("cat"), **Right1**("dog")) +
        ...
            + Sim(**Window8**("cat"), **Window8**("dog")) = ?

> **High** sim(word1, word2)
>   → word1 and word2 are **paradigmatically related**

Figure 13.4   **Multiple views of the context of a word can be used to compute similarity.**

words like *eats* and *my* or non-adjacent words like *Saturday* or *Tuesday*. This flexibility allows us to match the similarity in somewhat different ways. We might want to capture similarity based on general content, which yields loosely related paradigmatic relations. If we only used words immediately to the left and right, we would likely capture words that are very much related by their syntactic categories. Thus, the general idea of discovering paradigmatic relations is to compute the similarity of context of two words. For example, we can measure the similarity of *cat* and *dog* based on the similarity of their context, as shown in Figure 13.4.

The similarity function can be a combination of similarities on different contexts, and we can assign weights to these different similarities to allow us to focus more on a particular kind of context. Naturally, this would be application-specific, but again, the main idea for discovering pardigmatically related words is to compute the similarity of their contexts.

Let's see how we exactly compute these similarity functions. Unsurprisingly, we can use the vector space model on bag-of-words context data to model the context of a word for paradigmatic relation discovery. In general, we can represent a pseudo document or context of *cat* as one frequency vector d_1 and another word *dog* would give us a different context, d_2. We can then measure the similarity of these two vectors. By viewing context in the vector space model, we convert the problem of paradigmatic relation discovery into the problem of computing the vectors and their similarity.

The two questions that we have to address are how to compute each vector and how to compute their similarity. There are many approaches that can be used to solve the problem, and most of them are developed for information retrieval. They have been shown to work well for matching a query vector and a document vector. We can adapt many of the ideas to compute a similarity of context documents for our purpose.

Probability that a randomly
picked word from d_1 is w_i

Count of word w_i in d_1

$$d_1 = (x_1, \dots x_N) \qquad x_i = \frac{c(w_i, d_1)}{|d_1|}$$

Total counts of words in d_1

$$d_2 = (y_1, \dots y_N) \qquad y_i = \frac{c(w_i, d_2)}{|d_2|}$$

$$\mathrm{Sim}(d_1, d_2) = d_1.d_2 = x_1 y_2 + \dots + x_N y_N = \sum_{i=1}^{N} x_i y_i$$

Probability that two randomly picked words
from d_1 and d_2, respectively, are identical
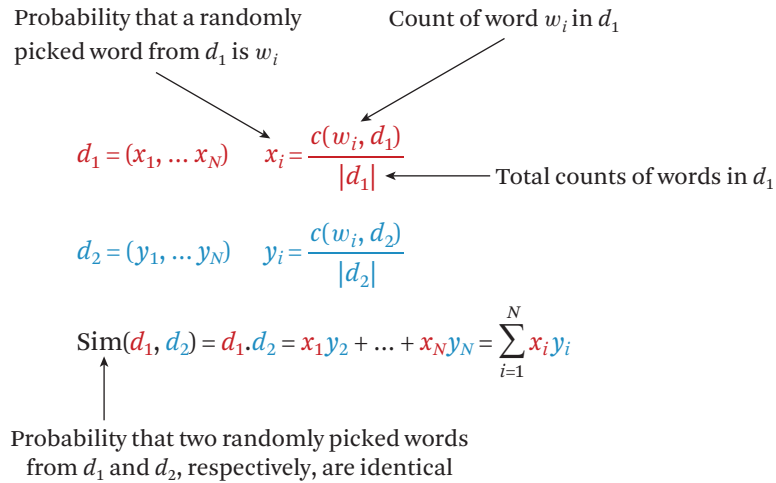
Figure 13.5   **A similarity function for word contexts.**

Figure 13.5 shows one plausible approach, where we match the similarity of context based on the expected overlap of words, and we call this EOW. We represent a context by a word vector where each word has a weight that's equal to the probability that a randomly picked word from this document vector is the current word. Equivalently, given a document vector x, x_i is defined as the normalized account of word w_i in the context, and this can be interpreted as the probability that you would randomly pick this word from d_1. The x_i's would sum to one because they are normalized frequencies, which means the vector is a probability distribution over words. The vector d_2 can be computed in the same way, and this would give us then two probability distributions representing two contexts. This addresses the problem of how to compute the vectors.

For similarity, we simply use a dot product of two vectors. The dot product, in fact, gives us the probability that two randomly picked words from the two contexts are identical. That means if we try to pick a word from one context and try to pick another word from another context, we can then ask the question, are they identical? If the two contexts are very similar, then we should expect we frequently will see the two words picked from the two contexts are identical. If they are very different, then the chance of seeing identical words being picked from the two contexts would be small. This is quite intuitive for measuring similarity of contexts.

Let's look at the exact formulas and see why this can be interpreted as the probability that two randomly picked words are identical. Each term in the sum

gives us the probability that we will see an overlap on a particular word w_i, where x_i gives us a probability that we will pick this particular word from d_1, and y_i gives us the probability of picking this word from d_2. This is how expected overlap of words in context similarity works.

As always, we would like to assess whether this approach would work well. Ultimately, we have to test the approach with real data and see if it gives us really semantically related words. Analytically, we can also analyze this formula. Initially, it does make sense because this formula will give a higher score if there is more overlap between the two contexts. However, if you analyze the formula more carefully, then you also see there might be some potential problems.

The first problem is that it might favor matching one frequent term very well over matching more distinct terms. That is because in the dot product, if one element has a high value and this element is shared by both contexts, it contributes a lot to the overall sum. It might indeed make the score higher than in another case where the two vectors actually have much overlap in different terms. In our case, we should intuitively prefer a case where we match more different terms in the context, so that we have more confidence in saying that the two words indeed occur in similar context. If you only rely on one high-scoring term, it may not be robust. The second problem is that it treats every word equally. If we match a word like *the*, it will be the same as matching a word like *eats*, although we know matching *the* isn't really surprising because it occurs everywhere. This is another problem of this approach.

We can introduce some heuristics used in text retrieval that solve these problems, since problems like these also occur when we match a query with a document. To tackle the first problem, we can use a sublinear transformation of term frequency. That is, we don't have to use the raw frequency count of the term to represent the context. To address this problem, we can transform it into some form that wouldn't emphasize the raw frequency so much. To address the second problem, we can reward matching a rare word. A sublinear transformation of term frequency and inverse document frequency (IDF) weighting are exactly what we'd like here; we discussed these types of weighting schemes in Chapter 6.

In order to achieve this desired weighting, we will use BM25 weighting, which is of course based on the BM25 retrieval function. It is able to solve the above two problems by sublinearly transforming the count of w_i in d_1 and including the IDF weighting heuristic in the similarity measure.

For this similarity scheme, we define the document vector as containing elements representing normalized BM25 TF values, as shown in Figure 13.6. The normalization function takes a sum over all the words in order to normalize the

$$d_1 = (x_1, \dots x_N) \quad \text{BM25}(w_i, d_1) = \frac{(k+1)c(w_i, d_1)}{c(w_i, d_1) + k(1 - b + b * |d_1|/\text{avdl})}$$

$$x_i = \frac{\text{BM25}(w_i, d_1)}{\sum_{j=1}^{N} \text{BM25}(w_j, d_1)} \qquad \begin{array}{l} b \in [0, 1] \\ k \in [0, +\infty) \end{array}$$

$$d_2 = (y_1, \dots y_N) \quad y_i \text{ is defined similarly}$$

$$\text{Sim}(d_1, d_2) = \sum_{i=1}^{N} \text{IDF}(w_i) x_i y_i$$

Figure 13.6 **A different similarity function based on BM25.**

weight of each word by the sum of the weights of all the words. This is to ensure all the x_i's will sum to one in this vector. This would be very similar to what we had before, in that this vector approximates a word distribution (since the x_i's will sum to one). For the IDF factor, the similarity function multiplies the IDF of word w_i by $x_i y_i$, which is the similarity in the ith dimension. Thus, the first problem (sublinear scaling) is addressed in the vector representation and the second problem (lack of IDF) is addressed in the similarity function itself.

We can also use this approach to discover syntagmatic relations. When we represent a term vector to represent a context with a term vector we would likely see some terms have higher weights and other terms have lower weights. Depending on how we assign weights to these terms, we might be able to use these weights to discover the words that are strongly associated with a candidate word in the context. The idea is to use the converted representation of the context to see which terms are scored high. If a term has high weight, then that term might be more strongly related to the candidate word.

We have each x_i defined as a normalized weight of BM25. This weight alone reflects how frequently the w_i occurs in the context. We can't simply say a frequent term in the context would be correlated with the candidate word because many common words like *the* will occur frequently in the context. However, if we apply IDF weighting, we can then re-weight these terms based on IDF. That means the words that are common, like *the*, will get penalized. Now, the highest-weighted terms will not be those common terms because they have lower IDFs. Instead, the highly weighted terms would be the terms that are frequently in the context but not frequent in the collection. Clearly, these are the words that tend to occur in the context of the candidate word. For this reason, the highly weighted terms in this

idea of a weighted vector can also be assumed to be candidates for syntagmatic relations.

Of course, this is only a byproduct of our approach for discovering paradigmatic relations. In the next section, we'll talk more about how to discover syntagmatic relations in particular. This discussion clearly shows the relation between discovering the two relations. Indeed, these two word relations may be discovered in a joint manner by leveraging such associations. This also shows some interesting connections between the discovery of syntagmatic relations and paradigmatic relations. Specifically, words that are paradigmatically related tend to have a syntagmatic relation with the same word.

To summarize, the main idea of computing paradigmatic relations is to collect the context of a candidate word to form a pseudo document which is typically represented as a bag of words. We then compute the similarity of the corresponding context documents of two candidate words; highly similar word pairs have the highest paradigmatic relations, i.e., the words that share similar contexts. There are many different ways to implement this general idea, but we just talked about a few of the approaches. Specifically, we talked about using text retrieval models to help us design an effective similarity function to compute the paradigmatic relations. More specifically, we used BM25 TF and IDF weighting to discover paradigmatic relations. Finally, syntagmatic relations can also be discovered as a byproduct when we discover paradigmatic relations.

13.3 Discovery of Syntagmatic Relations

There are strong syntagmatic relations between words that have correlated co-occurrences. That means when we see one word occur in some context, we tend to see the other word.

Consider a more specific example shown in Figure 13.7. We can ask the question, whenever *eats* occurs, what other words also tend to occur? Looking at the sentences on the left, we see some words that might occur together with *eats*, like *cat*, *dog*, or *fish*. If we remove them and look at where we only show *eats* surrounded by two blanks, can we predict what words occur to the left or to the right?

If these words are associated with *eats*, they tend to occur in the context of *eats*. More specifically, our prediction problem is to take any text segment (which can be a sentence, paragraph, or document) and determine what words are most likely to co-occur in a specific context.

Let's consider a particular word w. Is w present or absent in the segment from Figure 13.8? Some words are actually easier to predict than other words—if you

Whenever "**eats**" occurs, what **other words** also tend to occur?

My **cat eats fish** on Saturday
His **cat eats turkey** on Tuesday
My **dog eats meat** on Sunday
His **dog eats turkey** on Tuesday
...

My ___ **eats** ___ on Saturday
His ___ **eats** ___ on Tuesday
My ___ **eats** ___ on Sunday
His ___ **eats** ___ on Tuesday
...

What words tend to occur
to the **left** of "**eats**"?

What words are
to the **right**?
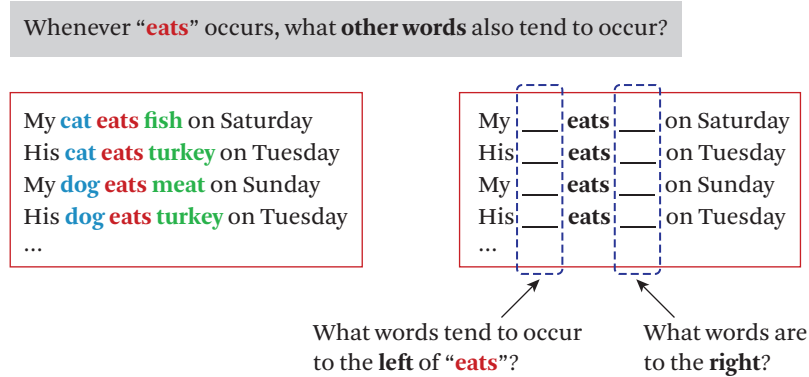
Figure 13.7   **Prediction of words in a context of another word.**

Prediction question: Is word w present (or absent) in this segment?

Text segment (any unit, e.g., sentence, paragraph, document)

Are some words easier to predict than others?
(1) w = "meat"   (2) w = "the"   (3) w = "unicorn"

Figure 13.8   **Prediction of absence and presence of a word.**

take a look at the three words shown in the figure (*meat*, *the*, and *unicorn*), which one do you think is easier to predict? If you think about it for a moment you might conclude that *the* is easier to predict because it tends to occur everywhere. The word *unicorn* is also relatively easy to predict because *unicorn* is rare. However, *meat* is somewhere in between in terms of frequency, making it harder to predict (since it's possible that it occurs in the segment).

Recall our discussion of entropy from Chapter 2. Earlier, we talked about using entropy to capture how easy it is to predict the presence or absence of a word. We can create a random variable X_w for a particular word w that depicts whether w occurs. Clearly, this is related to the previous question. Here we will further talk about **conditional entropy,** which is useful for discovering syntagmatic relations.

Know nothing about the segment   Know "eats" is present ($X_{\text{eats}} = 1$)

$$p(X_{\text{meat}} = 1) \quad \text{---------->} \quad p(X_{\text{meat}} = 1 | X_{\text{eats}} = 1)$$
$$p(X_{\text{meat}} = 0) \quad \text{---------->} \quad p(X_{\text{meat}} = 0 | X_{\text{eats}} = 1)$$

$$H(X_{\text{meat}}) = -p(X_{\text{meat}} = 0) \log_2 p(X_{\text{meat}} = 0) - p(X_{\text{meat}} = 1) \log_2 p(X_{\text{meat}} = 1)$$

$$H(X_{\text{meat}} | X_{\text{eats}} = 1) = -p(X_{\text{meat}} = 0 | X_{\text{eats}} = 1) \log_2 p(X_{\text{meat}} = 0 | X_{\text{eats}} = 1)$$
$$-p(X_{\text{meat}} = 1 | X_{\text{eats}} = 1) \log_2 p(X_{\text{meat}} = 1 | X_{\text{eats}} = 1)$$

$H(X_{\text{meat}} | X_{\text{eats}} = 0)$ can be defined similarly

Figure 13.9   **Illustration of conditional entropy.**

Now, we'll address a different scenario where we assume that we know something about the random variable. That is, suppose we know that *eats* occurred in the segment. How would that help us predict the presence or absence of a word like *meat*? If we frame this question using entropy, that would mean we are interested in knowing whether knowing the presence of *eats* could reduce uncertainty about *meat*. In other words, can we reduce the entropy of the random variable corresponding to the presence or absence of *meat*? What if we know of the absence of *eats*? Would that also help us predict the presence or absence of *meat*? These questions can be addressed by using conditional entropy.

To explain this concept, let's first look at the scenario we had before, when we know nothing about the segment. We have probabilities indicating whether a word occurs or doesn't occur in the segment. We have an entropy function that looks like the one in Figure 13.9.

Suppose we know *eats* is present, which means we know the value of X_{eats}. That fact changes all these probabilities to conditional probabilities. We look at the presence or absence of *meat*, given that we know *eats* occurred in the context. That is, we have $p(X_{meat} | X_{eats} = 1)$. If we replace these probabilities with their corresponding conditional probabilities in the entropy function, we'll get the conditional entropy (conditioned on the presence of *eats*). This is essentially the same entropy function as before, except that all the probabilities now have a condition. This then tells us the entropy of *meat* after we have known *eats* occurs in the segment. Of course, we can also define this conditional entropy for the scenario where we don't see *eats*. Now, putting these different scenarios together, we have the complete definition of conditional entropy:

$$H(X_{meat} \mid X_{eats}) = \sum_{u \in \{0,1\}} p(X_{eats} = u) H(X_{meat} \mid X_{eats} = u)$$

$$= \sum_{u \in \{0,1\}} p(X_{eats} = u)$$

$$\cdot \sum_{v \in \{0,1\}} \left(-p(X_{meat} = v \mid X_{eats} = u) \log_2 p(X_{meat} = v \mid X_{eats} = u) \right)$$

This formula considers both scenarios of the value of *eats* and captures the conditional entropy regardless of whether *eats* is equal to 1 or 0 (present or absent). We define the conditional entropy of *meat* given *eats* as the following expected entropy of *meat* for both values of *eat:*

$$H(X_{meat} \mid X_{eats}) = \sum_{u \in \{0,1\}} p(X_{eats} = u) H(X_{meat} \mid X_{eats} = u). \qquad (13.1)$$

In general, for any discrete random variables X and Y, we have the conditional entropy is no larger than the entropy of the variable X; that is,

$$H(X) \geq H(X \mid Y). \qquad (13.2)$$

This is an upper bound for the conditional entropy. The inequality states that we can only reduce uncertainty by adding more information, which makes sense. As we know more information, it should always help us make the prediction and can't hurt the prediction in any case.

This conditional entropy gives us one way to measure the association of two words because it tells us to what extent we can predict one word given that we know the presence or absence of another word.

Before we look at the intuition of conditional entropy in capturing syntagmatic relations, it's useful to think of a very special case of the conditional entropy of a word given itself: $H(X_{meat} \mid X_{meat})$. This means we know where *meat* occurs in the sentence, and we hope to predict whether the *meat* occurs in the sentence. This is zero because once we know whether the word occurs in the segment, we'll already know the answer of the prediction! That also happens to be when this conditional entropy reaches the minimum.

Let's look at some other cases. One is knowing *the* and trying to predict *meat*. Another is the case of knowing *eats* and trying to predict *meat*. We can ask the question: which is smaller, $H(X_{meat} \mid X_{the})$ or $H(X_{meat} \mid X_{eats})$? We know that smaller entropy means it is easier to predict.

In the first case, *the* doesn't really tell us much about *meat*; knowing the occurrence of *the* doesn't really help us reduce entropy that much, so it stays fairly

close to the original entropy of *meat*. In the case of *eats*, since *eats* is related to *meat*, knowing presence or absence of *eats* would help us predict whether *meat* occurs. Thus, it reduces the entropy of *meat*. For this reason, we expect the second term $H(X_{meat} \mid X_{eats})$ to have a smaller entropy, which means there is a stronger association between these two words.

This suggests that when you use conditional entropy for mining syntagmatic relations, the algorithm would look as follows.

1. For each word w_1, enumerate all other words w_2 from the corpus.

2. Compute $H(X_{w1} \mid X_{w2})$. Sort all candidates in ascending order of the conditional entropy.

3. Take the top-ranked candidate words as words that have potential syntagmatic relations with w_1.

Note that we need to use a threshold to extract the top words; this can be the number of top candidates to take or a value cutoff for the conditional entropy.

This would allow us to mine the most strongly correlated words with a particular word w_1. But, this algorithm does not help us mine the strongest k syntagmatic relations from the entire collection. In order to do that, we have to ensure that these conditional entropies are comparable across different words. In this case of discovering the syntagmatic relations for a target word like w_1, we only need to compare the conditional entropies for w_1 given different words.

The conditional entropy of w_1 given w_2 and the conditional entropy of w_1 given w_3 are comparable because they all measure how hard it is to predict the w_1. However, if we try to predict a different word other than w_1, we will get a different upper bound for the entropy calculation. This means we cannot really compare conditional entropies across words. The next section shows how we can use mutual information to solve this problem.

13.3.1 Mining syntagmatic relations using mutual information

The main issue with conditional entropy is that its values are not comparable across different words, making it difficult to find the most highly correlated words in an entire corpus. To address this problem, we can use **mutual information**.

In particular, the mutual information of X and Y, denoted $I(X; Y)$, is the reduction in entropy of X obtained from knowing Y. Specifically, the question we are interested in here is how much of a reduction in entropy of X can we obtain by knowing Y. Mathematically, mutual information can be defined as

$$I(X; Y) = H(X) - H(X \mid Y) = H(Y) - H(Y \mid X). \tag{13.3}$$

Mutual information is always non-negative. This is easy to understand because the original entropy is always not going to be lower than the (possibly) reduced conditional entropy. In other words, the conditional entropy will never exceed the original entropy; knowing some information can always help us potentially, but will not hurt us in predicting X. Another property is that mutual information is symmetric: $I(X;Y) = I(Y;X)$. A third property is that it reaches its minimum, zero, if and only if the two random variables are completely independent. That means knowing one of them does not tell us anything about the other.

When we fix X to rank different Ys using conditional entropy, we would get the same order as ranking based on mutual information. Thus, ranking based on mutual entropy is exactly the same as ranking based on the conditional entropy of X given Y, but the mutual information allows us to compare different pairs of X and Y. That is why mutual information is more general and more useful.

Let's examine the intuition of using mutual information for syntagmatic relation mining in Figure 13.10.

The question we ask is: whenever *eats* occurs, what other words also tend to occur? This question can be framed as a mutual information question; that is, which words have high mutual information with *eats*? So, we need to compute the mutual information between *eats* and other words.

For example, we know the mutual information between *eats* and *meat*, which is the same as between *meat* and *eats* because the mutual information is symmetric. This is expected to be higher than the mutual information between *eats* and *the*, because knowing *the* does not really help us predict the other word. You also can easily see that the mutual information between a word and itself is the largest, which is equal to the entropy of the word. In that case, the reduction is maximum because knowing one allows us to predict the other completely. In other words, the conditional entropy is zero which means mutual information reaches its maximum.

Mutual information: $I(X; Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$

Whenever "**eats**" occurs, what **other words** also tend to occur?

Which **words** have high mutual information with "**eats**"?

$$I(X_{\textbf{eats}}; X_{\textbf{meats}}) = I(X_{\textbf{meats}}; X_{\textbf{eats}}) \; > \; I(X_{\textbf{eats}}; X_{\textbf{the}}) = I(X_{\textbf{the}}; X_{\textbf{eats}})$$
$$I(X_{\textbf{eats}}; X_{\textbf{eats}}) = H(X_{\textbf{eats}}) \; \geq \; I(X_{\textbf{eats}}; X_{\textbf{w}})$$

Figure 13.10   **Mutual information for discovering syntagmatic relations.**

In order to compute mutual information, we often use a different form of mutual information that we can mathematically rewrite as

$$
\begin{aligned}
I(X_{w1}; X_{w2}) \\
= \sum_{u \in \{0,1\}} \sum_{v \in \{0,1\}} p(X_{w1} = u, X_{w2} = v) \log_2 \frac{p(X_{w1} = u, X_{w2} = v)}{p(X_{w1} = u) p(X_{w2} = v)},
\end{aligned}
\quad (13.4)
$$

which is in the context of KL-divergence (see Appendix C). The numerator of the fraction is the observed joint distribution and the denominator is the expected joint distribution if they are independent. KL-divergence quantifies the difference between these two distributions. That is, it measures the divergence of the actual joint distribution from the expected distribution under an independence assumption. The larger the divergence is, the higher the mutual information would be.

Continuing to inspect this formulation of mutual information, we see that it is also summed over many combinations of different values of the two random variables. Inside the sum, we are doing a comparison between the two joint distributions. Again, the numerator has the actually observed joint distribution of the two random variables while the denominator can be interpreted as the expected joint distribution of the two random variables. If the two random variables are independent, their joint distribution is equal to the product of the two probabilities, so this comparison will tell us whether the two variables are indeed independent.

If they are indeed independent, then we would expect that the numerator and denominator are the same. If the numerator is different from the denominator, that would mean the two variables are not independent and their difference can measure the strength of their association. The sum is simply to take all of the combinations of the values of these two random variables into consideration. In our case, each random variable can choose one of the two values, zero or one, so we have four combinations. If we look at this form of mutual information, it shows that the mutual information measures the divergence of the actual joint distribution from the expected distribution under the independence assumption. The larger this divergence is, the higher the mutual information would be.

Let's further look at exactly what probabilities are involved in the mutual information formula displayed in Figure 13.11.

First, we have to calculate the probabilities corresponding to the presence or absence of each word. For w_1, we have two probabilities shown here. They should sum to one, because a word can either be present or absent in the segment, and similarly for the second word, we also have two probabilities representing presence

$$I(X_{w1}; X_{w2}) = \sum_{u \in \{0,1\}} \sum_{v \in \{0,1\}} p(X_{w1} = u, X_{w2} = v) \log_2 \frac{p(X_{w1} = u, X_{w2} = v)}{p(X_{w1} = u)p(X_{w2} = v)}$$

Presence and absence of $w1$: $p(X_{w1} = 1) + p(X_{w1} = 0) = 1$

Presence and absence of $w2$: $p(X_{w2} = 1) + p(X_{w2} = 0) = 1$

Co-occurrences of $w1$ and $w2$:

$$\underline{p(X_{w1} = 1, X_{w2} = 1)} + \underline{p(X_{w1} = 1, X_{w2} = 0)} + \underline{p(X_{w1} = 0, X_{w2} = 1)} + \underline{p(X_{w1} = 0, X_{w2} = 0)} = 1$$

Both $w1$ and $w2$ occur   Only $w1$ occurs   Only $w2$ occurs   None of them occurs

Figure 13.11   **Probabilities involved in the definition of mutual information.**

Presence and absence of $w1$: $p(X_{w1} = 1) + p(X_{w1} = 0) = 1$

Presence and absence of $w2$: $p(X_{w2} = 1) + p(X_{w2} = 0) = 1$

Co-occurrences of $w1$ and $w2$:

$$p(X_{w1} = 1, X_{w2} = 1) + p(X_{w1} = 1, X_{w2} = 0) + p(X_{w1} = 0, X_{w2} = 1) + p(X_{w1} = 0, X_{w2} = 0) = 1$$

Constraints:

$$p(X_{w1} = 1, X_{w2} = 1) + p(X_{w1} = 1, X_{w2} = 0) = p(X_{w1} = 1)$$
$$p(X_{w1} = 0, X_{w2} = 1) + p(X_{w1} = 0, X_{w2} = 0) = p(X_{w1} = 0)$$
$$p(X_{w1} = 1, X_{w2} = 1) + p(X_{w1} = 0, X_{w2} = 1) = p(X_{w2} = 1)$$
$$p(X_{w1} = 1, X_{w2} = 0) + p(X_{w1} = 0, X_{w2} = 0) = p(X_{w2} = 0)$$

Figure 13.12   **Constraints on probabilities in the mutual information function.**

or absence of this word. These all sum to one as well. Finally, we have a lot of joint probabilities that represent the scenarios of co-occurrences of the two words. They also sum to one because the two words can only have the four shown possible scenarios. Once we know how to calculate these probabilities, we can easily calculate the mutual information.

It's important to note that there are some constraints among these probabilities. The first was that the marginal probabilities of these words sum to one. The second was that the two words have these four scenarios of co-occurrence. The additional constraints are listed at the bottom of Figure 13.12.

Presence and absence of $w1$: $\boxed{p(X_{w1} = 1)} + p(X_{w1} = 0) = 1$

Presence and absence of $w2$: $\boxed{p(X_{w2} = 1)} + p(X_{w2} = 0) = 1$

Co-occurrences of $w1$ and $w2$:

$\boxed{p(X_{w1} = 1, X_{w2} = 1)} + p(X_{w1} = 1, X_{w2} = 0) + p(X_{w1} = 0, X_{w2} = 1) + p(X_{w1} = 0, X_{w2} = 0) = 1$

$\boxed{p(X_{w1} = 1, X_{w2} = 1)} + p(X_{w1} = 1, X_{w2} = 0) = \boxed{p(X_{w1} = 1)}$

$p(X_{w1} = 0, X_{w2} = 1) + p(X_{w1} = 0, X_{w2} = 0) = p(X_{w1} = 0)$

$\boxed{p(X_{w1} = 1, X_{w2} = 1)} + p(X_{w1} = 0, X_{w2} = 1) = \boxed{p(X_{w2} = 1)}$

$p(X_{w1} = 1, X_{w2} = 0) + p(X_{w1} = 0, X_{w2} = 0) = p(X_{w2} = 0)$

We only need to know $p(X_{w1} = 1), p(X_{w2} = 1)$, and $p(X_{w1} = 1, X_{w2} = 1)$.

Figure 13.13   **Computation of mutual information.**

The first new constraint means if we add up the probabilities of two words co-occurring and the probabilities when the first word occurs and the second word does not occur, we get exactly the probability that the first word is observed. The other three new constraints have a similar interpretation. These equations allow us to compute some probabilities based on other probabilities, and this can simplify the computation. More specifically, if we know the probability that a word is present, then we can easily compute the absence probability. It is very easy to use these equations to compute the probabilities of the presence and absence of each word.

Now let's look at the joint distribution. Assume that we also have available the probability that they occurred together. It's easy to see that we can actually compute all the rest of these probabilities based on these, as shown in Figure 13.13. Using the first of the four equations, we can compute the probability that the first word occurred and the second word did not because we know the two probabilities in the boxes. Similarly, using the third equation we can compute the probability that we observe only the second word. The figure shows that we only need to know how to compute the three boxed probabilities, namely the presence of each word and the co-occurrence of both words in a segment. All others can be computed based on them.

In general, we can use the empirical count of events in the observed data to estimate the probabilities, as shown in Figure 13.14. A commonly used technique is

$$p(X_{w1} = 1) = \frac{\text{count}(w1)}{N}$$

$$p(X_{w2} = 1) = \frac{\text{count}(w2)}{N}$$

$$p(X_{w1} = 1, X_{w2} = 1) = \frac{\text{count}(w1, w2)}{N}$$

	$w1$	$w2$	
Segment_1	1	0	Only $w1$ occurred
Segment_2	1	1	Both occurred
Segment_3	1	1	Both occurred
Segment_4	0	0	Neither occurred
...			
Segment_N	0	1	Only $w2$ occurred

count($w1$) = total number segments that contain $w1$
count($w2$) = total number segments that contain $w2$
count($w1$, $w2$) = total number segments that contain both $w1$ and $w2$

Figure 13.14   **Estimation of probabilities involved in the definition of mutual information.**

the maximum likelihood estimate (MLE), where we simply normalize the observed counts. Using MLE, we can compute these probabilities as follows. For estimating the probability that we see a word occuring in a segment, we simply normalize the count of segments that contain this word. On the right side of Figure 13.14, you see a list of some segments of data. In some segments you see both words occur, which is indicated as ones for both columns. In some other cases only one will occur, so only that column has a one and the other column has a zero.

To estimate these probabilities, we simply need to collect the three counts: the count of w_1 (the total number of segments that contain w_1), the segment count for w_2, and the count when both words occur (both columns have ones). Once we have these counts, we can just normalize these counts by N, which is the total number of segments, giving us the probabilities that we need to compute mutual information.

There is a small problem when we have zero counts sometimes. In this case, we don't want a zero probability, so we use smoothing, as discussed previously in this book.

To smooth, we will add a small constant to these counts so that we don't get zero probability in any case. Smoothing for this application is displayed in Figure 13.15. We pretend to observe pseudo-segments that would contribute additional counts of these words so that no event will have zero probability. In particular for this example, we introduce four pseudo-segments. Each is weighted at 1/4. These represent the four different combinations of occurrences of the two words.

Each combination will have at least a non-zero count from a pseudo-segment; thus, in the actual segments that we'll observe, it's okay if we haven't observed all of

$$p(X_{w1} = 1) = \frac{\text{count}(w1) + 0.5}{N + 1}$$

$$p(X_{w2} = 1) = \frac{\text{count}(w2) + 0.5}{N + 1}$$

$$p(X_{w1} = 1, X_{w2} = 1) = \frac{\text{count}(w1, w2) + 0.25}{N + 1}$$

Smoothing: Add pseudo data
so that no event has zero counts
(pretend we observed extra data)

		$w1$	$w2$
¼	PseudoSeg_1	0	0
¼	PseudoSeg_2	1	0
¼	PseudoSeg_3	0	1
¼	PseudoSeg_4	1	1
Segment_1		1	0
...			
Segment_N		0	1

Actually observed data
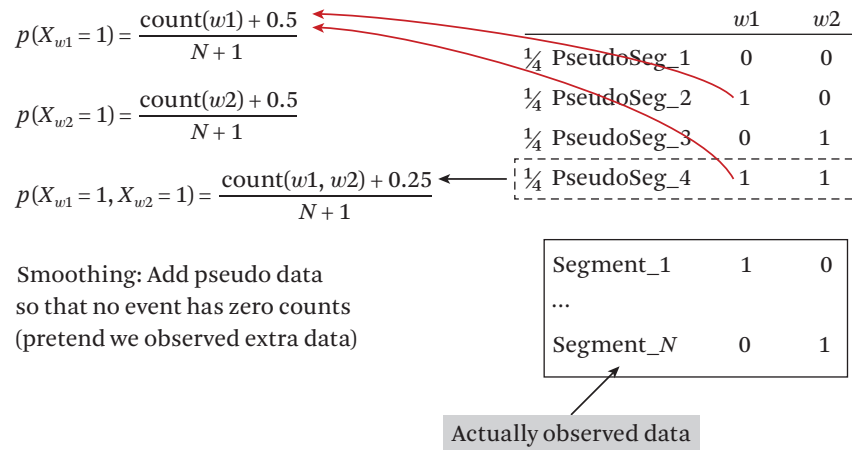
Figure 13.15   **Smoothing in estimation of probabilities for computing mutual information.**

the combinations. More specifically, you can see the 0.5 coming from the two ones in the two pseudo-segments, because each is weighted at one quarter. If we add them up, we get 0.5. Similar to this, 0.25 comes from one single pseudo-segment that indicates the two words occur together. In the denominator, we add the total number of pseudo-segments, which in this case is four pseudo-segments.

To summarize, syntagmatic relations can generally be discovered by measuring correlations between occurrences of two words. We've used three concepts from information theory: entropy, which measures the uncertainty of a random variable X; conditional entropy, which measures the entropy of X given we know Y; and mutual information of X and Y, which matches the entropy reduction of X due to knowing Y, or entropy reduction of Y due to knowing X. These three concepts are actually very useful for other applications as well. Mutual information allows us to have values computed on different pairs of words that are comparable, allowing us to rank these pairs and discover the strongest syntagmatic relations from a collection of documents.

Note that there is some relation between syntagmatic relation discovery and paradigmatic relation discovery. We already discussed the possibility of using BM25 to weight terms in the context and suggest candidates that have syntagmatic relations with the target word. Here, once we use mutual information to discover syntagmatic relations, we can also represent the context with this mutual information as weights. This would give us another way to represent the context of a word. And if we do the same for all the words, then we can cluster these words or compute

the similarity between these words based on their context similarity. This provides yet another way to do term weighting for paradigmatic relation discovery.

To summarize this chapter about word association mining, we introduced two basic associations called paradigmatic and a syntagmatic relations. These are fairly general since they apply to any items in any language; that is, the units don't have to be words (they can be phrases or entities). We introduced multiple statistical approaches for discovering them, mainly showing that pure statistical approaches are viable for discovering both kinds of relations. And they can be combined to perform joint analysis as well. These approaches can be applied to any text with no human effort, mostly because they are based on simple word counting, yet they can actually discover interesting word relations. We can also use different ways to define context and segment, and this would lead us to some interesting variations of applications. For example, the context can be very narrow like a few words, around a word, a sentence, or maybe paragraphs. Using differing contexts would allow discovery of different flavors of paradigmatic relations. Of course, these associations can support many other applications in both information retrieval and text data mining.

Discovery of word associations is closely related to term clustering, a topic that will be discussed in detail in Chapter 14, where some advanced techniques that can be potentially used for word association discovery will also be briefly discussed.

13.4 Evaluation of Word Association Mining

Word association mining is a fundamental technique, in that it is often used as a first step in many other tasks. In this chapter, we gave one example of using word association mining for query expansion.

The best way to convince an application developer that they should use word association mining is to show how it can improve their application. If the application is search, the question becomes: Does adding query expansion via word association mining improve MAP at a statistically-significant level? We know how to perform this type of evaluation from Chapter 9. The variable we control here between the two experiments is whether we perform query expansion or not. To be more thorough, we can compare query expansion with word association mining to query expansion with (for example) Rocchio feedback as a baseline model.

To evaluate word association mining in isolation, we would need some set of gold standard data. If we don't have such data, we would need to use human manual effort to judge whether the associations found are acceptable. Let's first consider the case where we have gold-standard data.

Without loss of generality, assume we wish to evaluate syntagmatic association mining. Given a word, the task may be to rank all other words in the vocabulary according to how similar they are to the target word. Thus, we could compute average precision for each word, and use MAP as a summary metric over each word that we evaluate. Of course, if such ranked lists contained numerical relevance scores we could instead use NDCG and average NDCG.

A human-based evaluation metric would be **intrusion detection**. In fact, this is one measure described in the evaluation of topic models [Chang et al. 2009], which we discuss further in Chapter 17. If the word associations are good, it should be fairly easy to find an "intruder" that has been added into the top k similar words. For example, consider the following two examples of intrusion detection presented in Chang et al. [2009]. We have two lists with $k + 1 = 6$ items. The top $k = 5$ items are chosen for some word in the vocabulary and an additional random word from the vocabulary is also added.

$$L_1 = \{dog, cat, horse, apple, pig, cow\},$$
$$L_2 = \{car, teacher, platypus, agile, blue, Zaire\}$$

The idea here is that if it's easy to spot the intruder, the top k words form a coherent group, meaning that they are a very good word association group. In L_1, it's quite obvious that *apple* is the intruder since it doesn't fit in with the other words in the list. Thus, the remaining words form a good word association list. In L_2, we can't really tell which word is the intruder, meaning the word association algorithm used to create the k candidates in L_2 is not as good as the one used to generate L_1. Performing this type of experiment over many different words in the vocabulary is a good (yet expensive) way to strictly evaluate the word associations. We say this method is expensive since it requires many human judgements.

Finally, it's important to consider the time-accuracy tradeoff of using such a tool in word association mining. Imagine the scenario where we have a baseline system with a MAP of 0.89 on some dataset. If we use query expansion via word association mining, we can get a statistically significantly higher MAP of 0.90. However, this doesn't take into account the preprocessing time of mining the word associations. In this example, the query time is not affected because the word association mining takes place beforehand offline, but it still is a non-negligible cost. The application manager would have to decide whether an increase in MAP of 0.01 is worth the effort of implementing, running, and maintaining the query expansion program. This is actually quite a realistic and general issue whenever new technology is proposed to replace or extend an existing one. As a data scientist, it is often part of the job to

convince others that such modifications are useful and worthwhile to the overall system.

Bibliographic Notes and Further Reading

Manning and Schütze [1999] has two useful relevant chapters on the discovery of word associations: Chapter 5 (Collocations) and Chapter 8 (Lexical Acquisition). An early reference on the use of mutual information for discovering word associations is Church and Hanks [1990]. Both paradigmatic and syntagmatic relations can also be discovered using random walks defined on word adjacency graphs, and a unified framework for modeling both kinds of word associations was proposed in Jiang and Zhai [2014]. Non-compositional phrases (also called lexical atoms) such as *hot dog* can also be discovered using similar heuristics to what we have discussed in this chapter (see Zhai 1997, Lin 1999). Another approach to word association discovery is the *n*-gram class language model [Brown et al. 1992]. Recently, word embedding techniques (e.g., `word2vec`; Mikolov et al. 2013) have shown great promise for learning a vector representation of a word that can further enable computation of similarity between two words, thus directly supporting paradigmatic relation discovery. Both the *n*-gram class language model and `word2vec` are briefly discussed in the context of term clustering in Chapter 14.

Exercises

13.1. What are the minimum and maximum possible values of the conditional entropy $H(X \mid Y)$? Under what situations do they occur?

13.2. In the mutual information section, we applied a simple smoothing technique. Based on your knowledge from Chapter 6, define a more robust smoothing method for calculating syntagmatic relations.

13.3. Feature selection is the process of reducing the dimensionality of the feature space to increase performance and decrease running time (since there are fewer features). Outline a feature selection method for the unigram words feature representation using word relations.

13.4. Do you think using the syntagmatic and paradigmatic word association mining methods would work for other feature types? Give some examples of other features where it may work and others where it may not.

13.5. Use META to implement one or both of the word association mining methods. Use the default unigram tokenization chain to read over a corpus and create

feature vectors for each term ID. Then, given a query term, return the most similar terms.

13.6. Outline a method to group together groups of words (i.e., more than two) that share similar meaning. For example, *plane*, *car*, and *train* may all be related.

13.7. Outline a method to determine synonyms based on search engine logs. That is, you are given many queries, and for each query is a list of clicked (assumed relevant) documents.

13.8. Outline a method to disambiguate homographs (two words that are spelled the same) based on search engine logs. For example, how can we distinguish a financial institution *bank* and a river *bank* based on these logs?

13.9. In what scenario (if any) is word association mining a generalization of an *n*-gram language model?

13.10. Depending on the context size, we get many different types of semantic meanings from word associations. Give two extremes of the types of relations we get when the context is very large and when it is very small.

13.11. Is it possible to adjust the word association mining algorithms to find antonyms instead of synonyms? If so, explain how; if not, explain why it is not possible. That is, we would like to assign a high score to the pair (*hot*, *cold*) since they are opposites and a low score to (*freezing*, *cold*) since they are synonyms.

Text Clustering

Clustering is a natural problem in exploratory text analysis. In its most basic sense, clustering (i.e., grouping) objects together lets us discover some inherent structure in our corpus by collecting similar objects. These objects could be documents, sentences, or words. We could cluster search engine queries, search engine results, and even users themselves.

Clustering is a general data mining technique very useful for exploring large data sets. When facing a large collection of text data, clustering can reveal natural semantic structures in the data in the form of multiple clusters (i.e., groups) of data objects. The clustering results can sometimes be regarded as knowledge directly useful in an application. For example, clustering customer emails can reveal major customer complaints about a product. In general, the clustering results are useful for providing an overview of the data, which is often very useful for understanding a data set at a high-level before zooming into any specific subset of the data for focused analysis. The clustering results can also support navigation into the relevant subsets of the data since the structures can facilitate linking of objects inside a cluster and linking of related clusters. In general, clustering methods are very useful for text mining and exploratory text analysis with widespread applications especially due to the fact that clustering algorithms are mostly **unsupervised** without requiring any manual effort, and can thus be applied to any text data set.

The object types that we cluster necessitate different tasks, and this variation leads to many interesting applications. For example, clustering of retrieval results can be used as a result summary or as a way to remove redundant documents. Clustering the documents in our entire corpus lets us find common underlying themes and can give us a better understanding of the type of data it contains. Term clustering is a powerful way to find concepts or create a thesaurus.

However, how do we formally define the problem of clustering? In particular, what does it actually mean for an object to be in a particular cluster? Intuitively, we
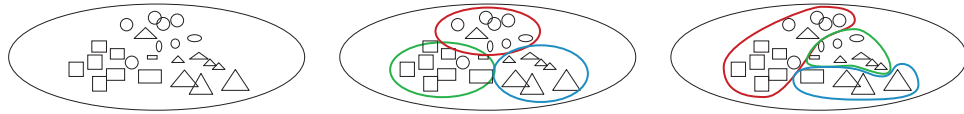
Figure 14.1   **Illustration of clustering bias. The figure on the left shows a set of objects that can be potentially clustered in different ways depending on the definition of similarity (or clustering bias). The figure in the middle shows the clustering results when similarity is defined based on the shape of an object. The figure on the right shows the clustering results of the same set of objects when similarity is defined based on size.**

imagine objects inside the same cluster are similar in some way—more so than objects that appear in two different clusters. However, such a definition of clustering is strictly speaking not well defined as we did not make it clear how exactly we should measure similarity. Indeed, an appropriate definition of similarity is quite crucial for clustering as a different definition would clearly lead to a different clustering result.

Consider the illustration in Figure 14.1. How should we cluster the objects shown in the figure on the left side? What would an ideal clustering result look like? Clearly these questions cannot be answered until we define the perspective for measuring similarity very clearly, i.e., inject a particular "**clustering bias**." If we define similarity based on the shape of an object, we will obtain a clustering result as shown in the picture in the middle of the figure. However, if we define the similarity based on the size of an object, then we would have very different results as shown in the figure on the right side. Thus, when we define a clustering task, it is important to state the desired perspective of measuring similarity, which we refer to as a "clustering bias." This bias will also be the basis for evaluating clustering results. The ambiguity of perspective for similarity not only exists in such an artificial example, but also exists everywhere. Take words for example: are "car" and "horse" similar? A car and a horse are clearly not similar physically. However, if we look at them from the perspective of their functions, we may say that they are similar since they can both be used as transportation tools. The "right" clustering bias clearly has to be determined by the specific application.

In different algorithms, the clustering bias is injected in different ways. For some clustering algorithms, it is up to the user to define or select explicitly a **similarity algorithm** for the clustering method to use. It will put (for example) documents that are all similar according to the chosen similarity algorithm in the same cluster. Other clustering algorithms are **model-based** (typically based on

generative probabilistic models), where the objective function of the model for the data (e.g., the likelihood function in the case of a generative probabilistic model) is to create an indirect bias on how similarity is defined. With these model-based methods, it's often the case that an object is assigned a probability distribution over all the clusters, meaning there is no "hard cluster assignment" as in the similarity-based methods. We explore both similarity-based clustering and model-based clustering in this book. This particular chapter focuses on similarity-based clustering, and the topic analysis chapter (Chapter 17) is a fine example of model-based clustering.

In this chapter, we examine clustering techniques for both words and documents. Clustering sentences can be viewed as a case of clustering small documents. We first start with an overview of clustering techniques, where we categorize the different approaches. Next, we discuss similarity-based clustering via two common methods (hierarchical and divisive methods). Then, we introduce term clustering via both semantic-relatedness and pointwise mutual information before mentioning two more advanced topics. We end with clustering evaluation.

14.1   Overview of Clustering Techniques

As mentioned previously, **document clustering** groups documents together into clusters. We can categorize document clustering methods into two categories.

Similarity-based clustering . These clustering algorithms need a similarity function to work. Any two objects have the potential to be similar, depending on how they are viewed. Therefore, a user must define the similarity in some way.

Agglomerative clustering is a "bottom up" approach, also called hierarchical clustering. In this approach, we gradually merge similar objects to generate clusters.

Divisive clustering is a "top down" approach. In this approach, we gradually divide the whole set of objects into smaller clusters.

For both above methods, each document can only belong to one cluster. This is a "hard assignment," unlike the clusters we receive from a model-based method.

Model-based techniques design a probabilistic model to capture the latent structure of data (i.e., features in documents), and fit the model to data to obtain clusters. Typically, this is an example of **soft clustering**, since one object can be in multiple clusters (with a certain probability). There will be much more discussion on this in the topic analysis chapter.

Term clustering has applications in query expansion. It allows similar terms to be added to the query, increasing the possible number of documents matched from the index. It also allows humans to understand advanced features more easily if there are many hundreds or thousands of them, or if they are hard to conceptualize. Later, we will first discuss term clustering using semantic relatedness via topic language models mentioned in Chapter 2. Then, we explore a simple probabilistic technique called **pointwise mutual information**. We briefly mention a hierarchical technique for term clustering called **Brown clustering**. This technique has similarity to the agglomerative document clustering along with the probabilistic nature of model-based methods. We finish term clustering with an explanation of **word vectors**, a context-based word representation that should remind you of the information retrieval problem setup.

As we will see in Chapter 17, output from a model-based topic analysis additionally gives us groups of similar words (in fact, these are the "topics"). Thus, topic analysis delivers both term and document clusters to the user.

Although with an unsupervised clustering algorithm, we generally do not provide any prior expectations as to what our clusters may contain, it is also possible to provide some supervision in the form of requiring two objects to be in the same cluster or not to be in the same cluster. Such supervision is useful when we have some knowledge about the clustering problem that we would like to incorporate into a clustering algorithm and allows users to "steer" the clustering in a flexible way. A user can also control the number of clusters by setting the number of clusters desired beforehand, or the user may leave it to the algorithm to determine what a natural breakdown of our objects is, in which case the number of clusters is usually optimized based on some statistical measures such as how well the data can be explained by a certain number of clusters.

Most clustering output does not give labels for the clusters found; it's up to the user to examine the groups of terms or documents and mentally assign a label such as "biology" or "architecture." However, there are also approaches to automate assignment of a label to a text cluster where a label is often a phrase or multiple phrases [Mei et al. 2007b]. This labeling task can be regarded as a form of text summarization which we will further discuss in Chapter 16.

Finally, a brief note on the implementation of clustering algorithms. As with the rest of the chapters in this part of the book, we will see that the information retrieval techniques that we discussed in Part II are often also very useful for implementing many other algorithms for text analysis, including clustering. For example, in the case of document clustering, we may assume we already have a forward index of tokenized documents according to some feature representation. Leveraging the

data structures already in place for supporting search is especially desirable in a unified software system for supporting both text data access and text analysis. The clustering techniques we discuss are general, so they can be potentially used for clustering many different types of objects, including, e.g., unigram words, bigram words, trigram POS-tags, or syntactic tree features. All the clustering algorithms need are a term vocabulary represented as term IDs. The clustering algorithms only care about term occurrences and probabilities, not what they actually represent. Thus—with the same clustering algorithm—we can cluster documents by their word usage or by similar stylistic patterns represented as grammatical parse tree segments. For term clustering, we may not use an index, but we do also assume that each sentence or document is tokenized and term IDs are assigned.

14.2 Document Clustering

In this section, we examine similarity-based document clustering through two methods: agglomerative clustering and divisive clustering. As these are both similarity-based clustering methods, a similarity measure is required. In case a refresh of similarity measures is required, we suggest the reader consult Chapter 6.

In particular, the similarity algorithms we use for clustering need to be **symmetric**; that is, $\text{sim}(d_1, d_2)$ must be equal to $\text{sim}(d_2, d_1)$. Furthermore, our similarity algorithm must be **normalized** on some range. Usually, this range is [0, 1]. These constraints ensure that we can fairly compare similarity scores of different pairs of objects. Most retrieval formulas we have seen—such as BM25, pivoted length normalization, and query likelihood methods—are asymmetric since they treat the query differently from the current document being scored. Whissell and Clarke [2013] explore symmetric versions of popular retrieval formulas and they show that they are quite effective.

Despite the fact that default query-document similarity measures are not used for clustering, it is possible to use (for example) Okapi BM25 term weighting in document vectors which are then scored with a simple symmetric similarity algorithm like **cosine similarity**. Recall that cosine similarity is defined as

$$\text{sim}_{\text{cosine}}(x, y) = \frac{x \cdot y}{||x|| \cdot ||y||} = \frac{\sum_i x_i y_i}{\sqrt{\sum_i (x_i)^2}\sqrt{\sum_i (y_i)^2}}. \tag{14.1}$$

Since all term weights in our document vector representation are positive, the cosine similarity score ranges from [0, 1]. As mentioned, the term weights may be raw counts, TF-IDF, or anything else the user could imagine. The cosine similarity captures the cosine of the angle between the two document vectors plotted in

their high-dimensional space; the larger the angle, the more dissimilar the documents are.

Another common similarity metric is **Jaccard similarity**. This metric is a set similarity; that is, it only captures the presence and absence of terms with no regard to magnitude. It is defined as follows:

$$\text{sim}_{\text{Jaccard}}(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}, \tag{14.2}$$

where X and Y represent the set of elements in the document vector x and y, respectively. In plain English, it captures the ratio of shared objects and total objects in both sets.

For a more in-depth study of similarity measures and their effectiveness, we suggest that the reader consult Huang [2008]. For the rest of this chapter, it is sufficient to assume that the base document-document similarity measure is cosine or Jaccard similarity. In any event, the goal of a particular similarity algorithm is to find an optimal partitioning of data to simultaneously maximize intra-group similarity and minimize inter-group similarity.

14.2.1 Agglomerative Hierarchical Clustering

We are now ready to discuss our first general clustering strategy. This method progressively constructs clusters to generate a hierarchy of merged groups. This bottom-up (agglomerative) approach gradually groups similar objects (single documents or groups of documents) into larger and larger clusters until there is only one cluster left. The tree may then be segmented as needed. Alternatively, the merging may be stopped when the desired number of clusters is found. This series of merges forms a **dendrogram**, represented in Figure 14.2.

In the figure, the original documents are numbered one through eleven and comprise the bottom row of the dendrogram. Circles represent clusters of more than one document, and lines represent which documents or clusters were merged together to form the next, larger cluster.

The clustering algorithm is straightforward: while there is more than one cluster, find the two most similar clusters and merge them. This does present an issue though when we need to compare the similarity of a cluster with a cluster, or a cluster with a single document. Until now, we have only defined similarity measures that take two documents as input. To simplify this problem, we will treat individual documents as clusters; thus we only need to compare clusters for similarity. The cluster similarity measures we define make use of the document-document similarity measures presented previously.
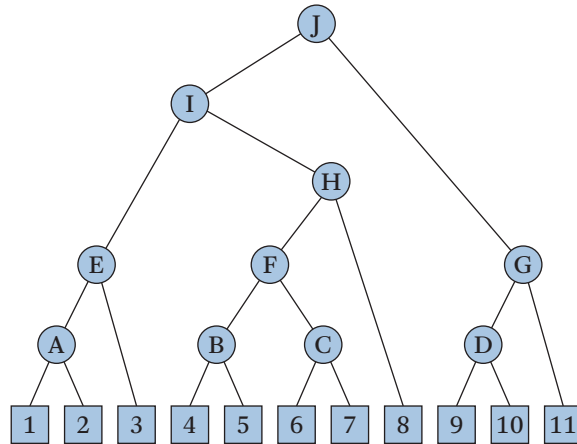
Figure 14.2   Hierarchical clustering represented as a dendrogram.
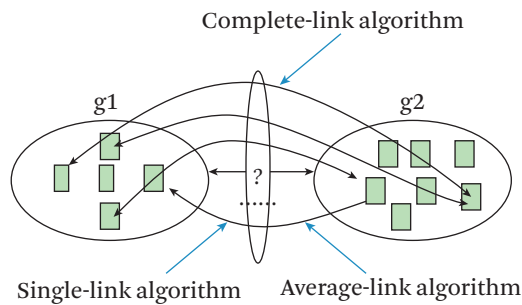
Figure 14.3   Three different cluster-cluster similarity metrics.

Below, we outline three cluster similarity measures and illustrate them in Figure 14.3.

Single-link merges the two clusters with the *smallest minimum* distance. This results in "looser" clusters, since we only need to find two individually close elements in each cluster in order to perform the merge.

Complete-link merges the two clusters with the *smallest maximum* distance between elements. This results in very "tight" and "compact" clusters since the cluster diameter is kept small (i.e., the distance between all elements low).

Algorithm 14.1 *K*-means clustering algorithm

Initialize K randomly selected centroids
while not converged **do**
    Assign each document to the cluster whose centroid is closest to
        it using sim($\cdot$) (Ex.)
    Recompute centroids of the new clusters found from previous step (Max.)
end while

Average-link is a compromise between the two previous measures. As its name implies, it takes the *smallest average* distance between two clusters.

Both single-link and complete-link are sensitive to outliers since they rely on only the similarity of one pair of documents. Average-link is essentially a group decision, making it less sensitive to outliers. Of course, as with most methods discussed in this book, the specific application will determine which method is preferred. In fact, it may even be useful to try out different document-document similarity measures combined with different cluster-cluster similarity measures to see how the dataset is partitioned.

14.2.2 *K*-means

A complementary clustering method to our hierarchical algorithm is a top-down, divisive approach. In this approach, we repeatedly apply a flat clustering algorithm to partition the data into smaller and smaller clusters. In flat clustering, We will start with an initial tentative clustering and iteratively improve it until we reach some stopping criterion. Here, we represent a cluster with a **centroid**; a centroid is a special document that represents all other documents in its cluster, usually as an average of all its members' values.

The K-means algorithm[1] sets K centroids and iteratively reassigns documents to each one until the change in cluster assignment is small or nonexistent. This technique is described in the algorithm below. Let sim($\cdot$) be the chosen document-document similarity measure.

The two steps in K-means are marked as the expectation step (Ex.) and the maximization step (Max.); this algorithm is one instantiation of the widely found **Expectation-Maximization algorithm**, commonly called just EM. We will return to this powerful algorithmic paradigm in much more detail in Chapter 17 on topic

---

1. K-means is not at all related to the classification algorithm k-NN (see Chapter 15).
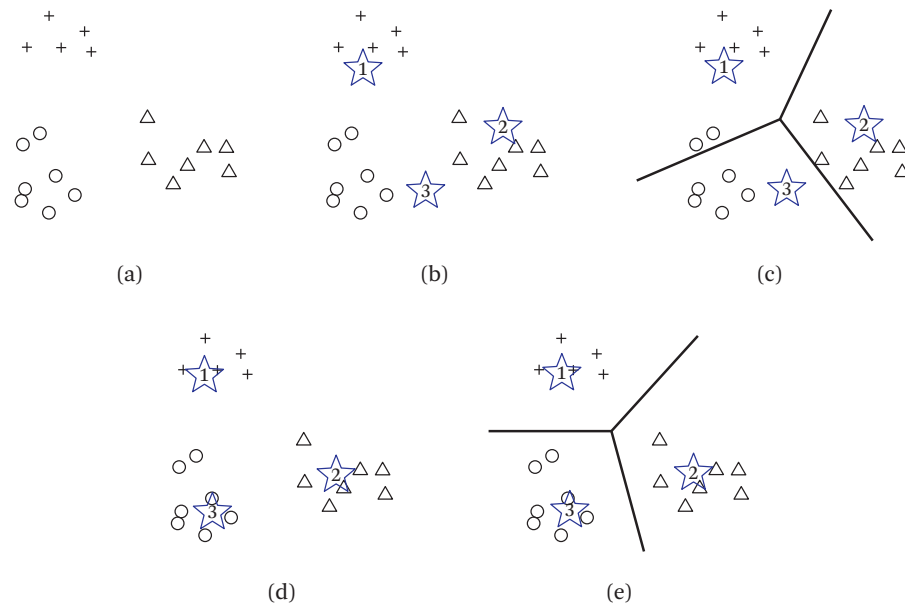
Figure 14.4   Steps in the K-means clustering algorithm for a small set of data points to be clustered (shown in (a)). First, three initial (starting) centroids are randomly chosen (shown in (b)). Then, all the data points are each assigned to one of the three clusters based on their distances to each centroid; the decision boundaries are shown as lines in (c). The assignments lead to three tentative clusters, each of which can then be used to compute a new centroid to better represent the cluster (shown as three stars in new locations in (d)). The algorithm continues to iterate with the new centroids as the starting centroids to re-assign all the data points. The new boundaries are shown in (e), which are easily seen to be already very close to the optimal centroids for generating three clusters from this data set.

analysis through the PLSA algorithm. For this chapter, it is sufficient to realize that K-means is a particular manifestation of hard cluster assignment via EM.

Figure 14.4 shows the K-means algorithm in action. Frame (a) shows our initial setup with the data points to be clustered. Here we visualize the data points with different shapes to suggest that there are three distinct clusters, corresponding to three shapes (crosses, circles, and triangles). Frame (b) shows how three random centroids are chosen ($K = 3$). In frame (c), the black lines show the partition of documents in their respective centroid. These lines can be found by first drawing a line to connect each pair of centroids and then finding the perpendicular bisectors of the segments connecting two centroids. This step is marked (Ex.) in the pseudocode. Then, once the cluster assignments are determined, frame (d) shows how

the centroids are recomputed to improve the centroids' positions. This centroid reassignment step is marked as (Max.) in the pseudocode. Thus, frames (c) and (d) represent one iteration of the algorithm which leads to improved centroids. Frames (e) further shows how the algorithm can continue to obtain improved boundaries, which in turn would lead to further improved centroids.

When a document is represented as a term vector (as discussed in Chapter 6), and a Euclidean distance function is used, the K-means algorithm can be shown to minimize an objective function that computes the average distances of all the data points in a cluster to the centroid of the cluster. The algorithm is also known to converge to a local minimum, but not guaranteed to converge to a global minimum. Thus, multiple trials are generally needed in order to obtain a good local minimum.

The K-means algorithm can be repeatedly applied to divide the data set gradually into smaller and smaller clusters, thus creating a hierarchy of clusters similar to what we can achieve with the agglomerative hierarchical clustering algorithm. Thus both agglomerative hierarchical clustering and K-means can be used for hierarchical clustering; they complement each other in the sense that K-means constructs the hierarchy by incrementally dividing the whole set of data (a top-down strategy), while agglomerative hierarchical clustering constructs the hierarchy by incrementally merging data points (a bottom-up strategy). Note that although in its basic form, agglomerative hierarchical clustering generates a binary tree, it can easily adapted to generate more than two branches by merging more than two groups into a cluster at each iteration. Similarly, if we only allow a binary tree, then we also do not have to set K in the K-means algorithm for creating a hierarchy.

14.3 Term Clustering

The goal of term clustering is quite similar to document clustering; we wish to find related terms. By "related," we usually mean words that have a similar semantic meaning. For example, *soccer* and *basketball* are related in the sense that they are both sports. Similarly, *evaluation* and *assessment* are related since they are synonyms.

In this section we will refer to "terms" and "words" interchangeably, though keep in mind we don't necessarily only have to cluster words. We commonly use this example since it is quite straightforward to imagine. The techniques we describe in this section will generally work for any sequence of features, whether they are words or parse tree fragments. It is important to keep in mind, however, that the algorithms we discuss were designed for use on words in most cases. It's also important to note that in some forms of term "clustering," we only receive

a pairwise score between two words w_1 and w_2. If these scores are normalized, we can still cluster the entire set of terms by using the pairwise scores.

As in all clustering problems, definition of similarity is important. In the case of term clustering, the question is how we should define the similarity between two terms. It is easy to see that the paradigmatic relations and syntagmatic relations between words (or terms) are both natural candidates for serving as a basis to define similarity. The paradigmatic relation similarity would lead to clusters of terms that tend to occur in very similar contexts with the same relative "location" in the context, whereas the syntagmatic relations would lead to clusters of terms that are semantically related and also tend to co-occur in similar contexts but in different "locations."

In the rest, we will first revisit a method for finding semantically related words from earlier in this book. Then, we introduce the concept of pointwise mutual information and show how it can also be used to find related terms. We end with an introduction to more advanced term clustering methods.

14.3.1 Semantically Related Terms

Recall from Section 3.4 where we found which words were semantically related with the term *computer*. Figure 14.5 is reproduced here from Section 3.4.

Topic LM: $p(w|\text{"computer"})$

the 0.032
a 0.019
is 0.014
we 0.008
computer 0.004
software 0.0001
. . .
text 0.00006

All documents containing word "**computer**"

Background LM: $p(w|B)$

B

General background English text

the 0.03
a 0.02
is 0.015
we 0.01
. . .
computer 0.00001
. . .

Normalized topic LM:
$p(w|\text{"computer"})/p(w|B)$

computer 400
software 150
program 104
. . .
text 3.0
. . .
the 1.1
a 0.99
is 0.9
we 0.8
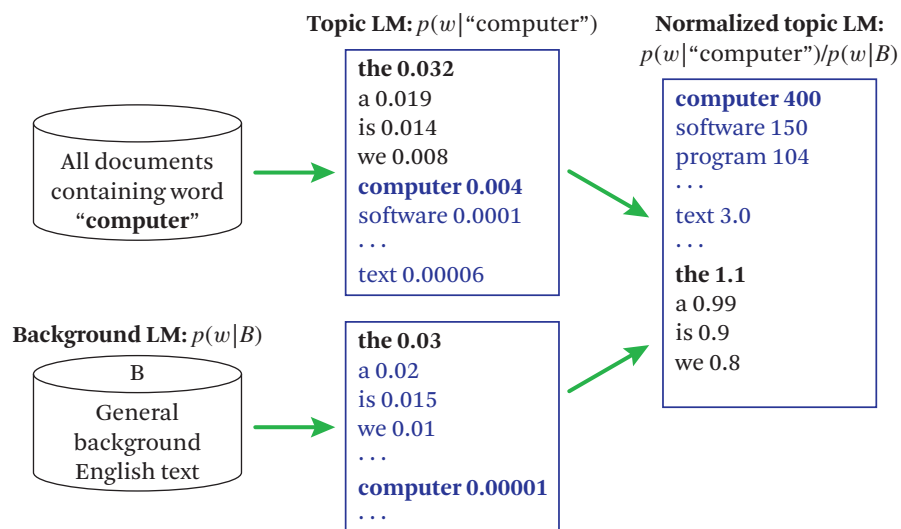
Figure 14.5 Using topic language models and a background language model to find semantically related words.

Also recall that we used the maximum likelihood estimate of a unigram language model to find $p(w \mid \hat{\theta})$, where $\hat{\theta}$ in our case is the topic language model associated with documents containing the term *computer*. That is,

$$p(w \mid \hat{\theta}) = \frac{c(w, D)}{|D|}. \tag{14.3}$$

After we estimated the topic and background language models, we used the following formula to assign scores to words in our vocabulary:

$$\text{score}(w) = \frac{p(w \mid computer)}{p(w \mid C)}. \tag{14.4}$$

The score indicates how related a word is to our topic language model term *computer*. Using maximum likelihood estimation, this becomes

$$\text{score}(w) = \frac{p(w \mid computer)}{p(w \mid C)} = \frac{\frac{c(w, D)}{|D|}}{\frac{c(w)}{|C|}} = \frac{c(w, D) \cdot |C|}{c(w) \cdot |D|}, \tag{14.5}$$

where D is the set of documents containing the term *computer* and C is the entire collection of documents.

We see that words that are more likely to appear in the context of *computer* will have a greater numerator than denominator, thus increasing the score. Words (such as *the*) that appear about equally regardless of the context will have a score close to one. Words that usually do not occur in the context of *computer* will have a denominator less than the numerator, resulting in a score less than one.

As mentioned in Section 3.4, there is a slight issue with this normalization formula. For example, assume the word *artichoke* appears only once in the corpus, and it happens to be in a document where *computer* is mentioned. Using the above formula will have *artichoke* and *computer* very highly related, even though we know this is not true.

One way to solve this problem is to smooth a maximum likelihood estimator by pretending that we have observed an extra pseudo count of every word, including unseen words. Thus, the formula for computing a smoothed background language model would be

$$p(w \mid C) = \frac{c(w, C) + 1}{|C| + |V|}, \tag{14.6}$$

where $|C|$ is the total count of all the words in collection C, and $|V|$ is the size of the complete vocabulary set. Note that the variable $|V|$ in the denominator is the total number of pseudo counts we have added to all the words in the vocabulary.

With such a formula, an unseen word would not have a zero probability, and the estimated probability is, in general, more accurate. We can replace $p(w \mid C)$ in the previous scoring function with our smoothed version. In the example, this brings the score for *artichoke* much lower since we "pretend" to have seen a count of it in the background. Words that actually are semantically related (i.e., that occur much more frequently in the context of *computer*) would not be affected by this smoothing and instead would "rise up" as the unrelated words are shifted downwards in the list of sorted scores.

From Chapter 6 we learned that this Add-1 smoothing may not be the best smoothing method as it applies too much probability mass to unseen words. In an even more improved scoring function, we could use other smoothing methods such as Dirichlet prior or Jelinek-Mercer interpolation.

In any event, this semantic relatedness is what we wish to capture in our term clustering applications. However, you can probably see that it would be infeasible to run this calculation for every term in our vocabulary. Thus, in the next section, we will examine a more efficient method to cluster terms together. The basic idea of the problem is exactly the same.

14.3.2 Pointwise Mutual Information

Pointwise Mutual Information (PMI) treats word occurrences as random variables and quantifies the probability of their co-occurrence within some context of a window of words. For example, to find words that co-occur with w_i using a window of size n, we look at the words

$$w_{i-n}, \ldots, w_{i-1}, w_i, w_{i+1}, \ldots, w_{i+n}.$$

This allows us to calculate the probability of w_i and w_j co-occurring, which is represented as the joint probability $p(w_i, w_j)$. Along with the individual probabilities $p(w_i)$ and $p(w_j)$, we can write the formula for PMI:

$$\text{pmi}(x_i, x_j) = \log\left(\frac{p(w_i, w_j)}{p(w_i)p(w_j)}\right). \tag{14.7}$$

Note that if w_i and w_j are independent, then $p(w_i)p(w_j) = p(w_i, w_j)$. This forces us to take a logarithm of 1, which yields a PMI of zero; there is no measure of information transferred by two independent words. If, however, the probability of observing the two words occurring together, i.e., $p(w_i, w_j)$ is substantially larger than their expected probability of co-occurrence if there were independent, i.e., $p(w_i)p(w_j)$, then the PMI would be high as we would expect.

Depending on our application, we can define the context as the aforementioned window of size n, a sentence, a document, and so on. Changing the context modifies the interpretation of PMI—for example, if we only considered a context to be of size $n = 1$, we will get significantly different results than if we set the context to be an entire document from the corpus. In order to have comparable PMI scores, we also need to ensure that our PMI measure is symmetric; this again depends on our definition of context. If we define context to be "w_j follows w_i", then $\text{pmi}(w_i, w_j) \neq \text{pmi}(w_j, w_i)$, which is required to cluster terms.

It is possible to normalize the PMI score in the range $[0, 1]$:

$$\text{npmi}(w_i, w_j) = \frac{\text{pmi}(w_i, w_j)}{-\log p(w_i, w_j)}, \tag{14.8}$$

making comparisons between different word pairs possible. However, this normalization doesn't fix a major issue in the PMI formula itself. Imagine that we have a rare word that always occurs in the context of another (perhaps very common) word. It would seem that this word pair is very highly related, but in fact our data is just too sparse to model the connection appropriately. This problem can be alleviated by using the mutual information measure introduced in Chapter 13 which considers not just the case when the rare word is observed, but also the case when it is *not* observed. Indeed, since mutual information is bounded by the entropy of one of the two variables, and a rare word has very low entropy, it generally wouldn't have a high mutual information with any other word.

Despite their drawbacks, however, PMI and nPMI are often used in practice and are also useful building blocks for more advanced methods as well as allowing us to understand the basic idea behind information capture in word co-occurrence. We thus included a discussion in this book.

Below we will briefly introduce two advanced methods for term clustering. The windowing technique employed here is critical in both of the following advanced methods.

14.3.3   Advanced Methods

In this section, we introduce two advanced methods for term clustering.

14.3.3.1   *N*-gram Class Language Models

Brown clustering [Brown et al. 1992] is a model-based term clustering algorithm that constructs term clusters (called word classes) to maximize the likelihood of an n-gram class language model. However, since the optimization problem is intractable to solve computationally, the actual process of constructing term clusters is actually similar to hierarchical agglomerative clustering where single words are

merged gradually, but the criterion for merging in Brown clustering is based on a similarity function derived from the likelihood function. Specifically, the maximization of the likelihood function is shown to be equivalent to maximization of the mutual information of adjacent word classes, thus when merging two words, the algorithm would favor merging two words that are distributed very similarly since when such words are replaced by their respective classes, it would minimize the decrease of mutual information between adjacent classes.

Mathematically, assuming that we partition all the words in the vocabulary into C classes, the n-gram class language model defines the probability of observing a word w_n given that we have already $n - 1$ words preceeding w_n, i.e., $w_{n-1}, \ldots, w_1$ as

$$p(w_n \mid w_{n-1}, \ldots, w_1) = p(w_n \mid c_n) p(c_n \mid c_{n-1}, \ldots, c_1),$$

where c_i is the class of word w_i. It essentially assumes that the probability of observing w_n only depends on the *classes* of the previous words, but does not depend on the specific words, thus unless C is the same as vocabulary size (i.e., every word is in its own class), the n-gram class language model always has fewer parameters than the regular n-gram language model.

As a generative model, we would generate a word by first looking up the classes of the previous words, i.e., $c_{n-1}, \ldots, c_1$, then sample a class for the n-th position c_n using $p(c_n \mid c_{n-1}, \ldots, c_1)$, and finally sample a word at the n-th position by using $p(w \mid c_n)$. The distribution $p(w \mid c_n)$ captures how frequently we will observe word w when the latent class c_n is used.

If we are given the partitioning of words into C classses, then the maximum likelihood estimation is not hard as we can simply replace the words with their corresponding classes to estimate $p(c_n \mid c_{n-1}, \ldots, c_1)$ in the same way as we would for estimating a regular n-gram language model, and the probability of a word given a particular class $p(w \mid c)$ can also be easily estimated by pooling together all the observations of words in the data belonging to the class c and normalizing their counts, which gives an estimate of $p(w \mid c)$ essentially based on the count of word w in the whole data set.

However, finding the best partitioning of words is computationally intractable. Fortunately, we can use a greedy algorithm to construct word classes in very much the same way as agglomerative hierarchical clustering, i.e., gradually merging words to form classes by keeping track of the objective of maximizing the likelihood. A neat theoretical result is that the maximization of the likelihood is equivalent to maximization of the mutual information between all the adjacent classes in the case of bigram model. Thus, the best pairs of words to merge would tend to

be those that are distributed in very similar contexts (e.g., *Tuesday* and *Wednesday*) since by putting such words in the same class, the prediction power of the class would be about the same as that of the original word, allowing to minimize the loss of mutual information. Computation-wise, we simply do agglomerative hierarchical clustering and measure the "distance" of two words based on a derived function based on the likelihood function that can capture the loss of mutual information due to merging the two words. Due to the complexity of the model, only bigrams ($n = 2$) were originally investigated [Brown et al. 1992].

Empirically, the bigram class language model has been shown to work very well and can generate very high-quality paradigmatic word associations directly by treating words in the same class as having paradigmatic relation. Figure 14.6 shows some sample word clusters taken from Brown et al. [1992]; they clearly capture paradigmatic relations well.

The model can also be used to generate syntagmatic associations by essentially computing the pointwise mutual information between words that occur in different
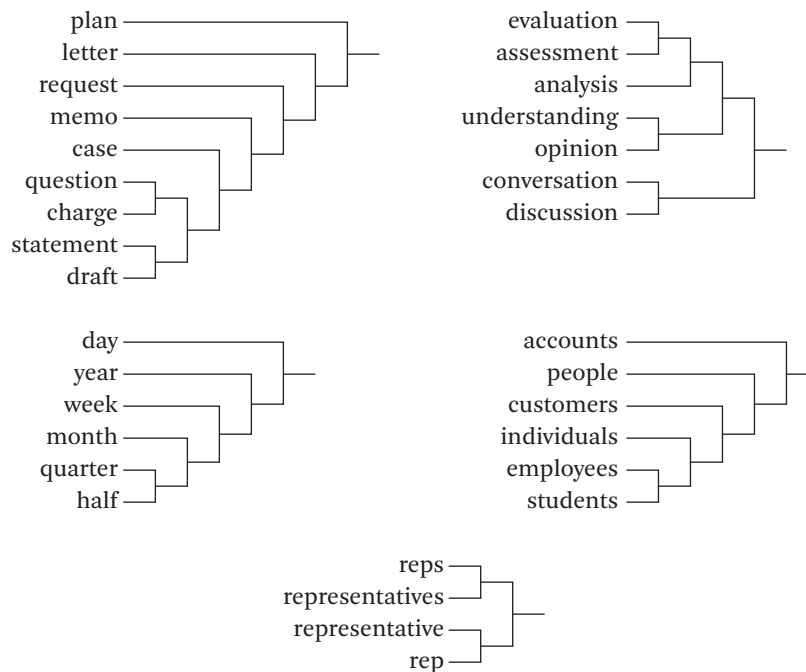
Figure 14.6 **Sample word classes constructed hierarchically using n-gram class language model.** (From Brown et al. [1992])

Word Pair	Mutual Information
Humpty Dumpty	22.5
Klux Klan	22.2
Ku Klux	22.2
Chah NuIth	22.2
Lao Bao	22.2
Nuu Chah	22.1
Tse Tung	22.1
avant garde	22.1
Carena Bancorp	22.0
gizzard shad	22.0
Bobby Orr	22.0
Warnok Hersey	22.0
mutatis murtandis	21.9
Taj Mahal	21.8

Figure 14.7   **Sample non-compositional phrases discovered using *n*-gram class language model.** (From Brown et al. [1992])

positions. When the window of co-occurrences is restricted to two words (i.e., adjacent co-occurrences), the model can discover "sticky phrases" (see Figure 14.7 for sample results), which are non-compositional phrases whose meaning is not a direct composition of the meanings of individual words. Such non-compositional phrases can also be discovered using some other statistical methods (see, e.g., Zhai 1997, Lin 1999).

14.3.3.2   Neural language model (word embedding)

In Chapter 13, we discussed in length how to represent a term as a term vector based on the words in the context where the term occurs, and compute term similarity based on the similarity of their vector representations. Such a contextual view of term representation can not only be used for discovering paradigmatic relations, but also support term clustering in general since we can use any document clustering algorithm by viewing a term as a "document" represented by a vector. It can also help word sense disambiguation since when an ambiguous word takes a different

sense, it tends to "attract" different words in its surrounding text, thus would have a different context representation.

This technique is not limited to unigram words, and we can think of other representations for the vector such as part-of-speech tags or even elements like sentiment. Adding these additional features means expanding the word vector from $|V|$ to whatever size we require. Additionally, aside from finding semantically-related terms, using this richer word representation has the ability to improve downstream tasks such as grammatical parsing or statistical machine translation.

However, the heuristic way to obtain vector representation discussed in Chapter 13 has the disadvantage that we need to make many ad hoc choices, especially in how to obtain the term weights. Another deficiency is that the vector spans the entire space of words in the vocabulary, increasing the complexity of any further processing applied to the vectors.

As an alternative, we can use a neural language model [Mikolov et al. 2010] to systematically learn a vector representation for each word by optimizing a meaningful objective function. Such an approach is also called *word embedding*, which refers to the mapping of a word into a vector representation in a low-dimensional space. The general idea of these methods is to assume that each word corresponds to a vector in an unknown (latent) low-dimensional space and define a language model solely based on the vector representations of the involved words so that the parameters for such a language model would be the vector representations of words. As a result, by fitting the model to a specific data set, we can learn the vector representations for all the words. These language models are called neural language models because they can be represented as a neural network. For example, to model an n-gram language model $p(w_n \mid w_{n-1}, \ldots, w_1)$, the neural network would have $w_{n-1}, \ldots, w_1$ as input and w_n as the output. In some neural language models, the hidden layer in the neural network connected to a word can be interpreted as a vector representation of the word with the elements being the weights on the edges connected to the word.

For example, in the skip-gram neural language model [Mikolov et al. 2013], the objective function is to use each word to predict all other words in its context as defined by a window around the word, and the probability of predicting word w_1 given word w_2 is given by

$$p(w_1 \mid w_2) = \frac{exp(\vec{v_1}.\vec{v_2})}{\sum_{w_i \in V} exp(\vec{v_i}.\vec{v_2})}$$

where v_i is the corresponding vector representation of word w_i. In words, such a model says that the probability $p(w_1 \mid w_2)$ is proportional to the dot product of the

Term	Cosine similarity to "france"
spain	0.678515
belgium	0.665923
netherlands	0.652428
italy	0.633130
switzerland	0.622323
luxembourg	0.610033
portugal	0.577154
russia	0.571507
germany	0.563291
catalonia	0.534176

Figure 14.8 **Using `word2vec` to find the most similar terms to the query "france".** (From Mikolov et al. [2013])

vectors corresponding to the two words, w_1 and w_2. With such a model, we can then try to find the vector representation for all the words that would maximize the probability of using each word to predict all other words in a small window of words surrounding the word. In effect, we would want the vectors representing two semantically related words, which tend to co-occur together in a window, to be more similar so as to generate a higher value when taking their dot product.

Google's implementation of skip-gram, called `word2vec` [Mikolov et al. 2013] is perhaps the most well-known software in this area. They showed that performing vector addition on terms in vector space yielded interesting results. For example, adding the vectors for *Germany* and *capital* resulted in a vector very close to the vector *Berlin*. Figure 14.8 shows example output from using this tool. Although similar results can also be obtained by using heuristic paradigmatic relation discovery (e.g., using the methods we described in Chapter 13) and the *n*-gram class language model, word embedding provides a very promising new alternative that can potentially open up many interesting new applications of text mining due to its flexibility in formulating the objective functions to be optimized and the fact that the vector representation is systematically learned through optimizing an explicitly defined objective function. One disadvantage of word embedding, at least in its current form, is that the elements in the vector representation of a word are not meaningful and cannot be easily interpreted intuitively. As a result, the utility of these word vectors has so far been mostly limited to computation of word similarities, which can also obtained by using many other methods.

In summary, we have shown several methods to measure term similarity, which can then be used for term clustering. We started with a unigram language modeling approach, followed by pointwise mutual information. We then briefly introduced two model-based approaches, one based on *n*-gram language models and one based on neural language models for word embedding. These term clustering methods can be leveraged to improve the computation of similarity between documents or other text objects by allowing inexact matching of terms (e.g., allowing words in the same cluster or with high similarity to "match" with each other).

14.4 Evaluation of Text Clustering

All clustering methods attempt to maximize the following measures.

Coherence.   How similar are objects in the same cluster?

Separation.   How far away are objects in different clusters?

Utility.   How useful are the discovered clusters for an application?

As with most text mining (and many other) tasks, we can evaluate in one of two broad strategies: **manual evaluation** (using humans) or **automatic evaluation** (using predefined measures). Of the three criteria mentioned above, coherence and separation can be measured automatically with measures such as vector similarity, purity, or mutual information. There is a slight challenge when evaluating term clustering, since word-to-word similarity algorithms may not be as obvious as document-to-document similarities. We may choose to encode terms as word vectors and use the document similarity measures, or we may wish to use some other concept of semantic similarity as defined by preexisting ontologies like WordNet.[2] Although slightly more challenging, the concept of utility can also be captured if the final system output can be measured quantitatively. For example, if clustering is used as a component in search, we can see if using a different clustering algorithm improves F_1, MAP, or NCDG (see Chapter 9).

All clustering methods need some notion of similarity (or bias). After all, we wish to find groups of objects that are *similar* to one another in some way. We mainly discussed unigram words representations, though in this book we have elaborated on many different feature types. Indeed, *feature engineering* is an important component of implementing a clustering algorithm, and in fact any text mining algorithm in general. Choosing the right representation for your text allows you to quantify the important differences between items that cause them to end up in either the

---

2. https://wordnet.princeton.edu/

same or different clusters. Even if your clustering algorithm performs spectacularly in terms of (for example) intra-cluster similarity, the clusters may not be acceptable from a human viewpoint unless an adequate feature representation was used; it's possible that the feature representation is not able to capture a crucial concept and needs to be reexamined. Chapter 4 gives a good overview of many different textual features supported by META. In the next chapter on text categorization (Chapter 15), we also discuss how choosing the right features plays an important role in the overall classification accuracy.

As we saw in this chapter, similarity-based algorithms explicitly encode a similarity function in their implementation. Ideally, this similarity between objects is optimized to maximize intra-cluster coherence and minimize intra-cluster separation. In model-based methods (which will be discussed in Chapter 17), similarity functions are not inherently part of the model; instead, the notion of object similarity is most often captured by probabilistically high co-occurring terms within "similar" objects.

Measuring coherence and separation automatically can potentially be accomplished by leveraging a categorization data set; such a corpus has predefined clusters where each document belongs to a particular category. For example, a text categorization corpus could be product descriptions from an online retailer, and each product belongs in a product category, such as *kitchenware*, *books*, *grocery*, and so on. A clustering algorithm would be effective if it was able to partition the products based on their text into categories that roughly matched the predefined ones. A simple measure to evaluate this application would be to consider each output cluster and see if one of the predefined categories dominates the cluster population. In other words, take each cluster C_i and calculate the percentage of each predefined class in it. The clustering algorithm would be effective if, for each C_i, one predefined category dominates and scarcely appears in other clusters. Effectively, the clustering algorithm recreated the class assignments in the original dataset without any supervision. Of course, however, we have to be careful (if this is a parameter), to set the final number of clusters to match the number of classes.

In fact, deciding the optimal number of clusters is a hard problem for all methods! For example, in K-means, the final clusters depend on the initial random starting positions. Thus it's quite common to run the algorithm several times and manually inspect the results. The algorithm G-means [Hamerly and Elkan 2003] reruns K-means in a more principled way, splitting clusters if the data assigned to each cluster is not normally-distributed. Model-based methods may have some advantages in terms of deciding the optimal number of clusters, but the model itself

is often inaccurate. In practice, we may empirically set the number of clusters to a fixed number based on application needs or domain knowledge.

Which method works the best highly depends on whether the bias (definition of similarity) reflects our perspective for clustering accurately and whether the assumptions made by an approach hold for the problem and applications. In general, model-based approaches have more potential for doing "complex clustering" by encoding more constraints into the probabilistic model.

Bibliographic Notes and Further Reading

Clustering is a general technique in data mining and is usually covered in detail in any book on data mining [Han 2005], [Aggarwal 2015]. There is also a chapter on text clustering in Aggarwal and Zhai [2012], where many text clustering methods are reviewed. An empirical comparison of some document clustering techniques can be found in Steinbach et al. [2000]. Term clustering is related to word association discovery, a topic covered in Chapter 13. An interesting theoretical work on clustering is Kleinberg [2002], where it is shown that there does not exist any clustering that can satisfy a small number of desirable properties (i.e., an impossibility theorem about clustering).

Exercises

14.1. Clustering search results to allow browsing was one application of clustering given in this chapter. What clustering method would you choose to implement for your search engine, assuming simplicity, effectiveness, and running time were all concerns?

14.2. What type of clustering algorithm would you use to support browsing a corpus? Imagine users start with a small set of clusters and continually refine (or backtrack) their path in a search for interesting information.

14.3. The number of clusters plays an important role in the output of a clustering algorithm. For which clustering algorithms does the number of clusters play a large role in the overall running time, if any?

14.4. Cluster labeling is an active research field. Brainstorm some ideas how to assign cluster labels when clustering documents and when clustering terms. A good cluster-labeling algorithm would probably include some formulas based on term or cluster statistics.

14.5. Consider all three cluster-cluster similarity metrics discussed in this chapter. Which of them is most likely to form "chains" of documents as opposed to a tighter group? Why?

14.6. Implement K-means or hierarchical agglomerative clustering in META. Make your algorithm general to any bag-of-words tokenization method by clustering already-analyzed documents.

14.7. Design a heuristic to set the number of clusters (and their contents) given a dendrogram of hierarchically clustered data. Try to make your algorithm run in only one traversal of the tree.

14.8. Using the topic language models for semantic relatedness, rewrite the scoring function using each of Dirichlet prior and Jelinek-Mercer smoothing. That is, smooth $p(w \mid C)$ in the following function:

$$\text{score}(w) = \frac{p(w \mid computer)}{p(w \mid C)}$$

14.9. What are the maximum and minimum values of (unnormalized) PMI?

14.10. We discussed one potential drawback to PMI and nPMI. Is there any sort of preprocessing you can do that helps with this issue? For example, can we set a threshold on the type of words in the window, the minimum or maximum frequency of each word, or the implementation of the window itself?

14.11. What type of index structure could we use to efficiently store word vectors? Assume that the distribution of values is sparse in each vector.

14.12. Design a way to cluster documents based on multiple feature types. As a first case, consider clustering on both unigram words and unigram POS tags. As a more advanced case, consider clustering documents via unigram words, sentence lengths, and structural parse tree features. Hint: how would we do this in META?

14.13. In Chapter 11 we described collaborative filtering. One issue is that it does not scale well when the number of users or items is large. Suggest a solution using clustering that is able to provide a faster running time when many different users need recommendations.

Text Categorization

15.1 Introduction

In the previous chapter, we discussed how to perform text clustering—grouping documents together based on similar features. Clustering techniques are unsupervised, which has the advantage of not requiring any manual effort from humans and being applicable to any text data. However, we often want to group text objects in a particular way according to a set of pre-defined categories. For example, a news agency may be interested in classifying news articles into one or more topical categories such as *technology*, *sports*, *politics*, or *entertainment*, etc. If we are to use clustering techniques to solve this problem, we may obtain coherent topical clusters, but these clusters do not necessarily correspond to the categories the news agency has designed (for their application purpose). To solve such a problem, we can use text categorization techniques, which have widespread applications.

In general, the text categorization problem is as follows. Given a set of predefined categories, possibly forming a hierarchy, and often also a training set of labeled text objects (i.e., text objects with known labels of categories), the task of text categorization is to label (unseen) text objects with one or more categories. This is illustrated in Figure 15.1.

At the very high level, text categorization is usually to help achieve two goals of applications.

1. To **enrich text representation** (i.e., achieving more understanding of text): with text categorization, we would be able to represent text in multiple levels (keywords + categories). In such an application, we also call text categorization *text annotation*. For example, semantic categories assigned to text can be directly useful for an application as in the case of spam detection. Semantic categories assigned to text data can also facilitate aggregation of text content in a more meaningful way; for example, sentiment classification would enable aggregation of all positive/negative opinions about a product so as to give a more meaningful overall assessment of opinions.
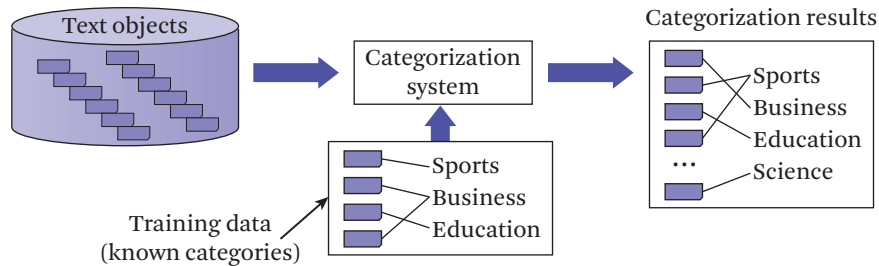
Figure 15.1 The task of text categorization (with training examples available).

2. To **infer properties of entities** associated with text data (i.e., discovery of knowledge about the world): as long as an entity can be associated with text data in some way, it is always potentially possible to use the text data to help categorize the associated entities. For example, we can use the English text data written by a person to predict whether the person is a non-native speaker of English. Prediction of party affiliation based on a political speech is another example. Naturally, in such a case, the task of text categorization is much harder as the "gap" between the category and text content is large. Indeed, in such an application, text categorization should really be called *text-based prediction*.

These two somewhat different goals can also be distinguished based on the difference in the categories in each case. For the purpose of enriching text representation, the categories tend to be "internal" categories that characterize a text object (e.g., topical categories, sentiment categories). For the purpose of inferring properties of associated entities with text data, the categories tend to be "external" categories that characterize an entity associated with the text object (e.g., author attribution or any other meaningful categories associated with text data, potentially through indirect links). Computationally, however, these variations are all similar in that the input is a text object and the output is one or multiple categories. We thus do not further distinguish these different variations.

The landscape of applications of text categorization is further enriched due to the variation we have in the text objects to be classified, which can include, e.g., documents, sentences, passages, or collections of text.

15.2 Overview of Text Categorization Methods

When there is no training data available (i.e., text data with known categories explicitly labelled), we often have to manually create heuristic rules to solve the problem

of categorization. For example, the rule *if the word "governor" occurs → assign politics label*. Obviously, designing effective rules requires a significant amount of knowledge about the specific problem of categorization.

Such a rule-based manual approach would work well if: (1) the categories are very clearly defined (usually means that the categories are relatively simple); (2) the categories are easily distinguished based on surface features in text (e.g., particular words only occur in a particular category of documents); and (3) sufficient domain knowledge is available to suggest many effective rules.

However, the manual approach has some significant disadvantages. The first is that it is labor-intensive, thus it does not scale up well both to the number of categories (since a new category requires new rules) and to the growth of data (since new data may also need new rules). The second is that it may not be possible to come up with completely reliable rules and it is hard to handle the uncertainty in the rules. Finally, the rules may not be all consistent with each other. As a result, the categorization results may depend on the order of application of different rules.

These problems with the rule-based manual approach can mostly be addressed by using machine learning where humans would help the machine by labeling some examples with the correct categories (i.e., creating training examples), and the machine will learn from these examples to somewhat automatically construct rules for categorization, only that the rules are somewhat "soft" and weighted, and how the rules should be combined is also learned based on the training data. Note that although in such a supervised machine learning approach, cateorization appears to be "automatic," it does require human effort in creating the training data, unless the training data is naturally available to us (which sometimes does happen). The human-created rules, if any, can also be used as features in such a learning-based approach, and they will be combined in a weighted manner to minimize the classification errors on the training data with the weights automatically learned. The machine may also automatically construct soft rules based on primitive features provided by humans as in the case of decision trees [Quinlan 1986], which can be easily interpreted as a "rule-based" classifier, but the paths from the root to the leaves (i.e., the rules) are inducted automatically by using machine learning. Once a classifier (categorizer) is trained, it can be used to categorize any unseen text data.

In general, all these learning-based categorization methods rely on discriminative features of text objects to distinguish categories, and they would combine multiple features in a weighted manner where the weights are automatically learned (i.e., adjusted to minimize errors of categorization on the training data). Different methods tend to vary in their way of measuring the errors on the training data, i.e.,

they may optimize a different objective function (also called a loss/cost function), and their way of combining features (e.g., linear vs. non-linear).

In the rest of the chapter, we will further discuss learning-based approaches in more detail. These automatic categorization methods generally fall into three categories. **Lazy learners** or **instance-based classifiers** do not model the class labels explicitly, but compare the new instances with instances seen before, usually with a similarity measure. These models are called "lazy" due to their lack of explicit generalization or training step; most calculation is performed at testing time. **Generative classifiers** model the data distribution in each category (e.g., unigram language model for each category). They classify an object based on the likelihood that the object would be observed according to each distribution. **Discriminative classifiers** compute features of a text object that can provide a clue about which category the object should be in, and combine them with parameters to control their weights. Parameters are optimized by minimizing categorization errors on training data.

As with clustering, we will be able to leverage many of the techniques we've discussed in previous chapters to create *classifiers*, the algorithms that assign labels to unseen data based on seen, labeled data. This chapter starts out with an explanation of the categorization problem definition. Next, we examine what types of features (text representation) are often used for classification. Then, we investigate a few common learning algorithms that we can implement with our forward and inverted indexes. After that, we see how evaluation for classification is performed, since the problem is inherently different from search engine evaluation.

15.3   Text Categorization Problem

Let's take our intuitive understanding of categorizing documents and rewrite the example from Chapter 2 into a more mathematical form. Let our collection of documents be $\mathbf{X}$; perhaps they are stored in a forward index (see Chapter 8). Therefore, one $x_i \in \mathbf{X}$ is a term vector of features that represent document i. As with our retrieval setup, each x_i has $|x_i| = |V|$ (one dimension for each feature, as assigned by the tokenizer). Our vector from Chapter 8 is an example of such an x_i with a very small vocabulary of size $|V| = 8$.

$$\{mr.: 1,\ \ quill: 1,\ \ 's: 1,\ \ book: 1,\ \ is: 1,\ \ very: 2,\ \ long: 1,\ \ .: 1\}.$$

Recall that if a document x_j consisted of the text *long long book*, it would be

$$\{mr.: 0,\ \ quill: 0,\ \ 's: 0,\ \ book: 1,\ \ is: 0,\ \ very: 0,\ \ long: 2,\ \ .: 0\}.$$

In our forward index, we'd store

$$x_i = \{1, 1, 1, 1, 1, 2, 1, 1\} \qquad x_j = \{0, 0, 0, 1, 0, 0, 2, 0\},$$

so x_{ik} is the kth term in the ith document.

We also have **Y**, which is a vector of labels for each document. Thus y_i may be *sports* in our news article classification setup and y_j could be *politics*.

A classifier is a function $f(\cdot)$ that takes a document vector as input and outputs a predicted label $\hat{y} \in \mathbf{Y}$. Thus we could have $f(x_i) = sports$. In this case, $\hat{y} = sports$ and the true y is also *sports*; the classifier was correct in its prediction.

Notice how we can only evaluate a classification algorithm if we know the true labels of the data. In fact, we will have to use the true labels in order to learn a good function $f(\cdot)$ to take unseen document vectors and classify them. For this reason, we often split our corpus **X** into two parts: **training data** and **testing data**. The training portion is used to build the classifier, and the testing portion is used to evaluate the performance (e.g., seeing how many correct labels were predicted).

But what does the function $f(\cdot)$ actually do? Consider this very simple example that determines whether a news article has positive or negative sentiment, i.e., $\mathbf{Y} = \{positive, negative\}$:

$$f(x) = \begin{cases} positive & \text{if } x\text{'s count for the term } good \text{ is greater than 1} \\ negative & \text{otherwise.} \end{cases}$$

Of course, this example is overly simplified, but it does demonstrate the basic idea of a classifier: it takes a document vector as input and outputs a class label. Based on the training data, the classifier may have determined that positive sentiment articles contain the term *good* more than once; therefore, this knowledge is encoded in the function. Later in this chapter, we will investigate some specific algorithms for creating the function $f(\cdot)$ based on the training data.

It's also important to note that these learning algorithms come in several different flavors. In **binary classification** there are only two categories. Depending on the type of classifier, it may only support distinguishing between two different classes. **Multiclass classification** can support an arbitrary number of labels. As we will see, it's possible to combine multiple binary classifiers to create a multiclass classifier. **Regression** is a very related problem to classification; it assigns real-valued scores on some range as opposed to discrete labels. For example, a regression problem could be to predict the amount of rainfall for a particular day given rainfall data for previous years. The output $\hat{y}$ would be a number ≥ 0, perhaps representing rainfall

in inches. On the other hand, the classification variant could predict whether there would be rainfall or not, $\mathbf{Y} = \{yes, no\}$.

15.4   Features for Text Categorization

In Chapter 6 we emphasized the importance of the document representation in retrieval performance. In Chapter 2, we emphasized the importance of the feature representation in general. The case is the same—if not greater—in text categorization. Suppose we wish to determine whether a document has positive or negative sentiment. Clearly, a bad text representation method could be the average sentence length. That is, the document term vector is a histogram of sentence lengths for each document. Intuitively, sentence length would not be a good indicator of sentiment. Even the best learning algorithm would not be able to distinguish between positive and negative documents based only on sentence lengths.[1]

On the other hand, suppose our task is basic essay scoring, where $\mathbf{Y} = \{fail, pass\}$. In this case, sentence length may indeed be some indicator of essay quality. While not perfect, we can imagine that a classifier trained on documents represented as sentence lengths would get a higher accuracy than a similar classification setup predicting sentiment.[2]

As a slightly more realistic example, we return to the sentiment analysis problem. Instead of using sentence length, we decide to use the standard unigram words representation. That is, each feature can be used to distinguish between positive or negative sentiment. Usually, most features are not useful, and the bulk of the decision is based on a smaller subset of features. Determining this smaller subset is the definition of **feature selection**, but we do not discuss this in depth at this point.

Although most likely effective, even unigram words may not be the best representation. Consider the terms *good* and *bad*, as mentioned in the classifier example in the previous section. In this scenario, context is very important:

> *I thought the movie was good.*
> *I thought the movie was not bad.*

Alternatively,

> *I thought the movie was not good.*
> *I thought the movie was bad.*

---

1. This, we assume. As an exercise, create a document tokenizer for META that uses sentence length as a feature. Can you get a decent sentiment classification accuracy?

2. Again, try this experiment in META using the same sentence-length tokenizer.

Clearly, a bigram words representation would most likely give better performance since we can capture *not good* and *not bad* as well as *was good* and *was bad*.

As a counterexample, using only bigram words leads us to miss out on rarer informative single words such as *overhyped*. This term is now captured in bigrams such as *overhyped (period)* and *very overhyped*. If we see the same rarer informative word in a different context—such as *was overhyped*—this is now an out-of-vocabulary term and can't be used in determining the sentence polarity. Due to this phenomenon, it is very common to combine multiple feature sets together. In this case, we can tokenize documents with both unigram and bigram words.

A well-known strategy discussed in Stamatatos [2009] shows that low-level lexical features combined with high-level syntactic features give the best performance in a classifier. These two types of features are more orthogonal, thus capturing different perspectives of the text to enrich the feature space. Having many different types of features allows the classifier a wide range of space on which to create a decision boundary between different class labels.

An example of very high-level features can be found in Massung et al. [2013]. Consider the grammatical parse tree discussed in Chapter 4 reproduced in Figure 15.2.
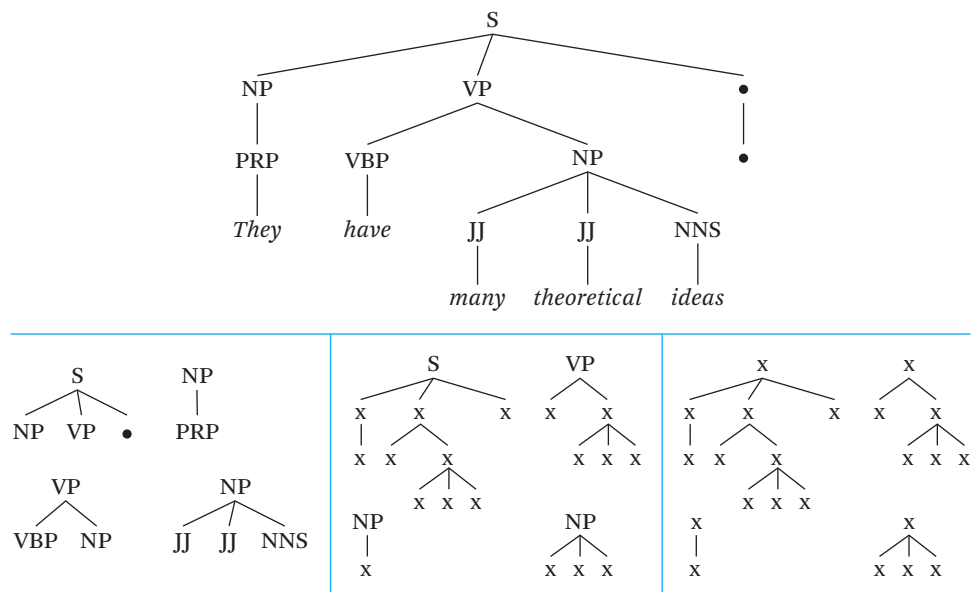
Figure 15.2 A grammatical parse tree and different feature representations derived from it. For each feature type, each dimension in a feature vector would correspond to a weight of a particular parse tree structure.

Here, we see three versions of increasingly "high-level" syntactic features. The bottom left square shows rewrite rules; these are the grammatical productions found in the sentence containing the syntactic node categories. For example, the *S* represents *sentence*, which is composed of a noun phrase (NP) followed by a verb phrase (VP), ending with a period. The middle square in Figure 15.2 omits all node labels except the roots of all subtrees. This captures a more abstract view of the production rules, focused more on structure. Lastly, the right square is a fully abstracted structural feature set, with no syntactic category labels at all. The authors found that these structural features combined with low-level lexical features (i.e., unigram words) improved the classification accuracy over using only one feature type.

Another interesting feature generation method is described in Massung and Zhai [2015] and called SYNTACTICDIFF. The idea of SYNTACTICDIFF is to define three basic (and therefore general) edit operations: `insert` a word, `remove` a word, and `substitute` one word for another. These edits are used to transform a given sentence. With a source sentence *S* and a reference text collection *R*, it applied edits that make *S* appear to come from *R*. In the non-native text analysis scenario [Massung and Zhai 2016], we operate on text from writers who are not native English speakers. Thus, transforming *S* with respect to *R* is a form of grammatical error correction.

While this itself is a specific application task, the series of edits performed on each sentence can also be used to represent the sentences themselves. For example,

$$\{insert(the) : 3, substitute(a \rightarrow an) : 1, \dots, remove(of) : 2\}$$

could be a feature vector for a particular sentence. This is just one example of a feature representation that goes beyond bag-of-words. The effectiveness of these "edit features" determines how effective the classifier can be in learning a model to separate different classes. In this example, the features can be used to distinguish between different native languages of essay writers. Again, it's important to emphasize that almost all machine learning algorithms are not affected by the type of features employed (in terms of operation; of course, the accuracy may be affected). Since internally the machine learning algorithms will simply refer to each feature as an ID, the algorithm may never even know if it's operating on a parse tree, a word, bigram POS tags, or edit features.

The NLP pipeline discussed in Chapter 3 and the tokenization schemes discussed in Chapter 4 give good examples of the options for effective feature representations. Usually, unigram words will be the default method, and more advanced techniques are added as necessary in order to improve accuracy. With these more

advanced techniques comes a requirement for more processing time. When using features from grammatical parse trees, a parser must first be run across the dataset, which is often at least an order of magnitude slower than simple whitespace-delimited unigram words processing. Running a parser requires the sentence to be part-of-speech tagged, and running coreference resolution requires grammatical parse trees. The level of sophistication in the syntactic or semantic features usually depends on the practitioner's tolerance for processing time and memory usage.

15.5 Classification Algorithms

In this section, we will look into how the function $f(\cdot)$ is actually able to distinguish between class labels. We examine three different algorithms, all of which are available in META. We will continue to use the sentiment analysis example, classifying new text into either the *positive* or *negative* label. Let's also assume we split our corpus into a training partition and testing partition. The training documents will be used to build $f(\cdot)$, and we will be able to evaluate its performance on each testing document.

Remember that we additionally have the metadata information **Y** for all documents, so we know the true labels of all the testing and training data. When used in production, we will not know the true label (unless a human assigns one), but we can have some confidence of the algorithm's prediction based on its performance on the testing data, which mimics the unseen data from the real world. The closer the testing data is to the data you expect to see in the wild, the greater is your belief in the classifier's accuracy.

15.5.1 *k*-Nearest Neighbors

k-NN is a learning algorithm that directly uses our inverted index and search engine. Unlike the next two algorithms we will discuss, there is no explicit training step; all we need to do is index the training documents. This makes k-NN a lazy learner or instance-based classifier.

As shown in the training and testing algorithms, the basic idea behind k-NN is to find the most similar documents to the query document, and use the most common class label of the similar documents. The assumption is that similar documents will have the same class label. Figure 15.3 shows an example of k-NN in action in the document vector space. Here there are three different classes represented as different colors plotted in the vector space. If $k = 1$, we would assign the red label to the query; if $k = 4$, we would assign the blue label, since three out of the top four

Algorithm 15.1 *k*-**NN Training**

Create an inverted index over the training documents

Algorithm 15.2 *k*-**NN Testing**

Let R be the results from searching the index with the unseen document as the query
Select the top k results from R
return the label that is most common in the k documents via majority voting

---
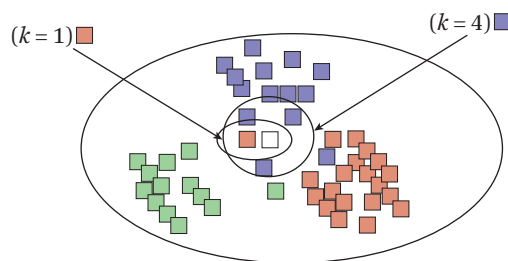
Figure 15.3 An example of k-NN with three classes where $k = 1, 4$. The query is represented as the white square.

similar documents are blue. In the case of a tie, the highest ranking document of the class with a tie would be chosen.

k-NN can be applied to any distance measure and any document representation. With only some slight modifications, we can directly use this classification method with an existing inverted index. A forward index is not required. Despite these advantages, there are some downsides as well. For one, finding the nearest neighbors requires performing a search engine query for each testing instance. While this is a heavily optimized operation, it will still be significantly slower than other machine learning algorithms in test time. As we'll see, the other two algorithms perform simple vector operations on the query vector as opposed to querying the inverted index. However, these other algorithms have a much longer training time than k-NN—this is the tradeoff. One more important point is the chosen label for k-NN is highly dependant on only the k neighbors; on the other hand, the other two algorithms take *all* training examples in account. In this way, k-NN is sensitive to the local structure of the feature space that the top k documents occupy. If it so hap-

pens that there are a few outliers from a different class close to our query, it will be classified incorrectly.

There are several variations on the basic k-NN framework. For one, we can weight the votes of the neighbors based on distance to the query in **weighted k-nearest neighbors**. That is, a closer neighbor to the query would have more influence, or a higher-weighted vote. A simple weighting scheme would be to multiply each neighbor's vote by $\frac{1}{d}$, where d is the distance to the query. Thus, more distant neighbors have less of an impact on the predicted label.

Another variant is the **nearest-centroid classifier**. In this algorithm, instead of using individual documents as neighbors, we consider the centroid of each class label (see Chapter 14 for more information on centroids and clustering). Here, if we have n classes, we simply see which of the n is closest to the query. The centroid of each class label may be thought of as a prototype, or ideal representation of a document from that class. We also receive a performance benefit, since we only need to do n similarity comparisons as opposed to a full search engine query over all the training documents.

15.5.2   Naive Bayes

Naive Bayes is an example of a generative classifier. It creates a probability distribution of features over each class label in addition to a distribution of the class labels themselves. This is very similar to language model topic probability calculation. With the language model, we create a distribution for each topic. When we see a new text object, we use our existing topics to find topic language model $\hat{\theta}$ that is most likely to have generated it. Recall from Chapter 2 that

$$\hat{\theta} = \arg\max_{\theta} p(w_1, \ldots, w_n \mid \theta) = \arg\max_{\theta} \prod_{i=1}^{n} p(w_i \mid \theta). \qquad (15.1)$$

Algorithm 15.3   **Naive Bayes Training**

Calculate $p(y)$ for each class label in the training data
Calculate $p(x_i \mid y)$ for each feature for each class label in the training data

Algorithm 15.4   **Naive Bayes Testing**

return the $y \in \mathbf{Y}$ that maximizes $p(y) \cdot \prod_{i=1}^{n} p(x_i \mid y)$

Our Naive Bayes classifier will look very similar. Essentially, we will have a feature distribution $p(x_i \mid y)$ for each class label y where x_i is a feature. Given an unseen document, we will calculate the most likely class distribution that it is generated from. That is, we wish to calculate $p(y \mid x)$ for each label $y \in \mathbf{Y}$. Let's use our knowledge of Bayes' rule from Chapter 2 to rewrite this into a form we can use programmatically given a document x.

$$
\begin{aligned}
\hat{y} &= \arg \max_{y \in \mathbf{Y}} p(y \mid x_1, \ldots, x_n) \\
&= \arg \max_{y \in \mathbf{Y}} \frac{p(y)p(x_1, \ldots, x_n \mid y)}{p(x_1, \ldots, x_n)} \\
&= \arg \max_{y \in \mathbf{Y}} p(y)p(x_1, \ldots, x_n \mid y) \\
&= \arg \max_{y \in \mathbf{Y}} p(y) \prod_{i=1}^{n} p(x_i \mid y)
\end{aligned}
\tag{15.2}
$$

Notice that we eliminate the denominator produced by Bayes' Rule since it does not change the arg max result. The final simplification is the independence assumption that none of the features depend on one another, letting us simply multiply all the probabilities together when finding the joint probability. It is for this reason that Naive Bayes is called *naive*.

This means we need to estimate the following distributions: $p(y)$ for all classes and $p(x_i \mid y)$ for each feature in each class. This estimation is done in the exact same way as our unigram language model estimation. That is, an easy inference method is maximum likelihood estimation, where we count the number of times a feature occurs in a class divided by its total number of occurrences. As discussed in Chapter 2 this may lead to some issues with unseen words or sparse data. In this case, we can smooth the estimated probabilities using any smoothing method we'd like as discussed in Chapter 6. We've covered Dirichlet prior smoothing and Jelinek-Mercer interpolation, among others. Finally, we need to calculate $p(y)$, which is just the probability of each class label. This parameter is essential when the class labels are unbalanced; that is, we don't want to predict a label that occurs only a few times in the training data at the same rate that we predict the majority label.

Whereas k-NN spent most of its calculation time in testing, Naive Bayes spends its time in training while estimating the model parameters. In testing, $|\mathbf{Y}|$ calculations are performed to find the most likely label. When learning the parameters, a forward index is used so it is known which class label to attribute features to; that is, look up the counts in each document, and update the parameter for that

document's true class label. An inverted index is not necessary for this usage. Memory aside from the forward index is required to store the parameters, which can be represented as $O(|\mathbf{Y}| + |V| \cdot |\mathbf{Y}|)$ floating point numbers.

Due to its simplicity and strong independence assumptions, Naive Bayes is often outperformed by more sophisticated classification methods, many of which are based on the linear classifiers discussed in the next section.

15.5.3   Linear Classifiers

Linear classifiers are inherently binary classifiers. Consider the following linear classifier:

$$f(x) = \begin{cases} +1 & \text{if } w \cdot x > 0 \\ -1 & \text{otherwise} \end{cases} \tag{15.3}$$

It takes the dot product between the unseen document vector and the weights vector w (where $|w| = |x| = |V|$). Training a linear classifier is learning (setting) the values in the weights vector w such that dotting it with a document vector produces a value greater than 0 for a positive instance and less than zero for a negative instance. There are many such algorithms, including the state-of-the-art support vector machines (SVM) classifier [Campbell and Ying 2011].

We call this group of learning algorithms *linear* classifiers because their decision is based on a linear combination of feature weights (w and x). Figure 15.4 shows how the dot product combination creates a decision boundary between two label groups plotted in a simple two-dimensional example. Two possible decision boundaries
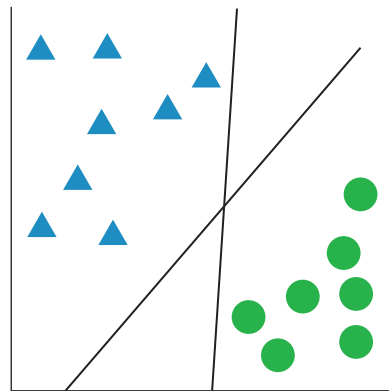
Figure 15.4   Two decision boundaries created by linear classifiers on two classes in two dimensions.

Algorithm 15.5   **Perceptron Training**

$w \leftarrow \{0, \ldots, 0\}$
for iteration $t \in T$ **do**
    for each training element i **do**
        $\hat{y} = w \cdot x_i$
        $w_j = w_j + \alpha(y_i - \hat{y}) \cdot x_{ij}, \forall j \in [0, |V|]$
    end for
    break if change in w is small
end for

---

are shown; the almost vertical line barely separates the two classes while the other line has a wide margin between the two classes. The SVM algorithm mentioned in the previous paragraph attempts to maximize the margin between the decision boundary and the two classes, thus leaving more "room" for new examples to be classified correctly, as they will fall on the side of the decision boundary close to the examples of the same class.

Of course, it's possible that not all data points are **linearly separable**, so the decision boundary will be created such that it splits the two classes as accurately as possible. Naive Bayes can also be shown to be a linear classifier. This is in contrast to k-NN—since it only considers a local subspace in which the query is plotted, it ignores the rest of the corpus and no lines are drawn. Some more advanced methods such as the *kernel trick* may change linear classifiers into nonlinear classifiers, but we refer the reader to a text more focused on machine learning for the details [Bishop 2006].

In this book, we choose to examine the relatively simple **perceptron classifier**, on which many other linear classifiers are based. We need to specify several parameters which are used in the training algorithm. Let T represent the maximum number of iterations to run training for. Let $\alpha > 0$ be the learning rate. The learning rate controls by how much we adjust the weights at each step. We may terminate training early if the change in w is small; this is usually measured by comparing the norm of the current iteration's weights to the norm of the previous iteration's weights.

There are many discussions about the choice of learning rate, convergence criteria and more, but we do not discuss these in this book. Instead, we hope to familiarize the reader with the general spirit of the algorithm, and again refer the reader to Bishop [2006] for many more details on algorithm implementation and theoretical properties.

Algorithm 15.6 **Perceptron Testing**

$\hat{y} \leftarrow w \cdot x$
return $+1$ **if** $\hat{y} > 0,$ **else return** -1

As the algorithm shows, training of the perceptron classifier consists of continuously updating the weights vector based on its performance in classifying known examples. In the case where y_i and $\hat{y}$ have the same sign (classified correctly), the weights are unchanged. In the case where $y_i < \hat{y}$, the object should have been classified as -1 so weight is subtracted from each active feature index in w. In the opposite case ($y_i > \hat{y}$), weight is added to each active feature in w. By "active feature," we mean features that are present in the current example x; only features $x_{ij} > 0$ will contribute to the update in w.

Eventually, the change in w will be small after some number of iterations, signifying that the algorithm has found the best accuracy it could. The final w vector is saved as the model, and it can be used to classify unseen documents.

What if we need to support multiclass classification? Not all classification problems fit nicely into two categories. Fortunately, there are two common methods for using multiple binary classifiers to create one multiclass categorization method on k classes.

One-vs-all (OVA) trains one classifier per class (for k total classifiers). Each classifier is trained to predict $+1$ for its respective class and -1 for all other classes. With this scheme, there may be ambiguities if multiple classifiers predict $+1$ at test time. Because of this, linear classifiers that are able to give a confidence score as a prediction are used. A confidence score such as $+0.588$ or $+1.045$ represents the $+1$ label, but the latter is "more confident" than the former, so the class that the algorithm predicting $+1.045$ would be chosen.

All-vs-all (AVA) trains $\frac{k(k-1)}{2}$ classifiers to distinguish between all pairs of k classes. The class with the most $+1$ predictions is chosen as the final answer. Again, confidence-based scoring may be used to add votes into totals for each class label.

15.6 Evaluation of Text Categorization

As with information retrieval evaluation, we can use precision, recall, and F_1 score by considering true positives, false positives, true negatives, and false negatives. We are also usually more concerned about accuracy (the number of correct predictions divided by the number of total predictions).

Training and testing splits were mentioned in the previous sections, but another partition of the total corpus is also sometimes used; this is the **development set,** used for parameter tuning. Typically, a corpus is split into about 80% training, 10% development, and 10% testing. For example, consider the problem of determining a good k value for k-NN. An index is created over the training documents, for (e.g.) $k = 5$. The accuracy is determined using the development documents. This is repeated for $k = 10, 15, 20, 25$. The best-performing k-value is then finally run on the testing set to find the overall accuracy. The purpose of the development set is to prevent **overfitting**, or tailoring the learning algorithm too much to a particular corpus subset and losing generality. A trained model is **robust** if it is not prone to overfitting.

Another evaluation paradigm is n-fold **cross validation**. This splits the corpus into n partitions. In n rounds, one partition is selected as the testing set and the remaining $n - 1$ are used for training. The final accuracy, F_1 score, or any other evaluation metric is then averaged over the n folds. The variance in scores between the folds can be a hint at the overfitting potential of your algorithm. If the variance is high, it means that the accuracies are not very similar between folds. Having one fold with a very high accuracy suggests that your learning algorithm may have overfit during that training stage; when using that trained algorithm on a separate corpus, it's likely that the accuracy would be very low since it modeled noise or other uninformative features from that particular split (i.e., it overfit).

Another important concept is **baseline accuracy**. This represents the minimum score to "beat" when using your classifier. Say there are 3,000 documents consisting of three classes, each with 1,000 documents. In this case, random guessing would give you about 33% accuracy, since you'd be correct approximately $\frac{1}{3}$ of the time. Your classifier would have to do better than 33% accuracy in order to make it useful! In another example, consider the 3,000 documents and three classes, but with an uneven class distribution: one class has 2,000 documents and the other two classes have 500 each. In this case, the baseline accuracy is 66%, since picking the majority class label will result in correct predictions $\frac{2}{3}$ of the time. Thus, it's important to take class imbalances into consideration when evaluating a classifier.

A **confusion matrix** is a way to examine a classifier's performance at a per-label level. Consider Figure 15.5, the output from running META on a three-class classification problem to determine the native language of the author of English text.

Each $(row, column)$ index in the table shows the fraction of times that row was classified as $column$. Therefore, the rows all sum to one. The diagonal represents the true positive rate, and hopefully most of the probability mass lies here, indicating a good classifier. Based on the matrix, we see that predicting Chinese was 80.2%

```
          chinese    english    japanese
         -----------------------------
 chinese | 0.802      0.011      0.187
 english | 0.0069     0.807      0.186
japanese | 0.0052     0.0039     0.991
```

Figure 15.5   META's confusion matrix output on a three-class classification problem.

accurate, with native English and Japanese as 80.7% and 99.1%, respectively. This shows that while English and Chinese had relatively the same difficulty, Japanese was very easy for the classifier to distinguish. We also see that if the classifier was wrong in a prediction on a Chinese or English true label, it almost always chose Japanese as the answer. Based on the matrix, the classifier seems to default to the label "Japanese". The table doesn't tell us why this is, but we can make some hypotheses based on our dataset. Based on this observation, we may want to tweak our classifier's parameters or do a more thorough feature analysis.

Bibliographic Notes and Further Reading

Text categorization has been extensively studied and is covered in Manning et al. [2008]. An early survey of the topic can be found in Sebastiani [2002]; a more recent one can be found in Aggarwal and Zhai [2012] where one chapter is devoted to this topic. Yang [1999] includes a systematic empirical evaluation of multiple commonly used text categorization methods and a discussion of text categorization evaluation. Moreover, since text categorization is often performed by using supervised machine learning, any book on machine learning is relevant (e.g., Mitchell 1997).

Exercises

15.1. In Section 15.4 we have two footnotes about sentence length feature generation. As they suggest, implement this tokenizer in META and see if one particular dataset type benefits from this method or not.

15.2. Use META to experiment with document classification. Which of the k-NN variants seems to perform the best? How dependent on the ranking function is the k-NN accuracy?

15.3. In META, SVM is called SGD with hinge loss (the default classifier). Does SVM always outperform Naive Bayes and k-NN? How do the runtimes in META compare for these three learners?

15.4. In text categorization, there are often thousands if not millions of features. This makes it very likely to be able to create a hyperplane separating class objects plotted in this high-dimensional space. Based on this information, make a suggestion for a default classifier to use in text categorization and explain your reasoning.

15.5. In an application setting, you must choose between using Naive Bayes or k-NN in order to do classification on text documents. Your application demands a very high performance and needs to classify documents very quickly. Explain your choice of classifier in this setting.

15.6. Give one similarity and one difference between k-NN and Naive Bayes.

15.7. Why is Naive Bayes "naive", and why is "Bayes" in the name?

15.8. Say we have a dataset and a classifier. We evaluate the classifier with 5-fold cross validation and 10-fold cross validation. Which do you think gives a higher accuracy? Why?

15.9. What is a difference between text categorization evaluation and information retrieval evaluation?

15.10. How can we determine if we have "enough" training data for a classifier? Make an argument using a plot, where the x axis is training data size and the y axis is classification accuracy on unseen test data.

15.11. Explain how to read a confusion matrix in order to determine two classes that are often mistaken for each other by the classifier.

15.12. Can a confusion matrix give us any clue to class imbalances? Explain.

Text Summarization

Text summarization refers to the task of compressing a relatively large amount of text data or a long text article into a more concise form for easy digestion. It is obviously very important for text data access, where it can help users see the main content or points in the text data without having to read all the text. Summarization of search engine results is a good example of such an application. However, summarization can also be useful for text data analysis as it can help reduce the amount of text to be processed, thus improving the efficiency of any analysis algorithm.

However, summarization is a non-trivial task. Given a large document, how can we convey the important points in only a few sentences? And what do we mean by "document" and "important"? Although it is easy for a human to recognize a good summary, it is not as straightforward to define the process. In short, for any text summarization application, we'd like a semantic compression of text; that is, we would like to convey essentially the same amount of information in less space. The output should be fluent, readable language. In general, we need a purpose for summarization although it is often hard to define one. Once we know a purpose, we can start to formulate how to approach the task, and the problem itself becomes a little easier to evaluate.

In one concrete example, consider a news summary. If our input is a collection of news articles from one day, a potentially valid output is a list of headlines. Of course, this wouldn't be the entire list of headlines, but only those headlines that would interest a user. For a different angle, consider a news summarization task where the input is one text news article and the output should be one paragraph explaining what the article talks about in a readable format. Each task will require a different solution.

Summarizing retrieval results is also of particular interest. On a search engine result page, how can we help the user click on a relevant link? A common strategy is to highlight words matching the query in a short snippet. An alternative approach would be to take a few sentences to summarize each result and display the short

summaries on the results page. Using summaries in this way could give the user a better idea of what information the document contains before he or she decides to read it.

Opinion summarization is useful for both businesses and shoppers. Summarizing all reviews of a product lets the business know whether the buyers are satisfied (and why). The review summaries also let the shoppers make comparisons between different products when searching online. Reviews can be further broken down into summaries of positive reviews and summaries of negative reviews. An even more granular approach described in Wang et al. [2010] and Wang et al. [2011] and further discussed in Chapter 18 uses topic models to summarize product reviews relating to different aspects. For hotel reviews, this could correspond to service, location, price, and value. Although the output in these two works is not a human-readable summary, we could imagine a system that is able to summarize all the hotel reviews in English (or any other language) for the user.

In this chapter, we overview two main paradigms of summarization techniques and investigate their different applications.

16.1   Overview of Text Summarization Techniques

There are two main methods in text summarization. The first is **selection-based** or **extractive summarization**. With this method, a summary consists of a sequence of sentences selected from the original documents. No new sentences are written, hence the summary is *extracted*. The second method is **generation-based** or **abstractive summarization**. Here, a summary may contain new sentences not in any of the original documents. One method that we explore here is using a language model. Previously in this book, we've used language models to calculate the likelihood of some text; in this chapter, we will show how to use a language model in reverse to generate sentences. We also briefly touch on the field of **natural language generation** in our discussion of abstractive techniques.

Following the pattern of previous chapters, we then move on to evaluation of text summarization. The two methods each have evaluation metrics that are particularly focused towards their respective implementation, but it is possible to use (e.g.) an abstractive evaluation metric on a summary generated by an extractive algorithm. Finally, we look into some applications of text summarization and see how they are implemented in real-world systems.

Text summarization is a broad field and we only touch on the core concepts in this chapter. For further reading, we recommend that the reader start with Das and Martins [2007], which provides a systematic overview of the field and contains much of the content from this chapter in an expanded form.

16.2 Extractive Text Summarization

Information retrieval-based techniques use the notion of sentence vectors and similarity functions in order to create a summarization text. A sentence vector is equivalent in structure to a document vector, albeit based on a smaller number of words. Below, we will outline a basic information retrieval-based summarization system.

1. Split the document to be summarized into sections or passages.

2. For each passage, "compress" its sentences into a smaller number of relevant (yet not redundant) sentences.

This strategy retains coherency since the sentences in the summary are mostly in the same order as they were in the original document.

Step one is portrayed in Figure 16.1. The sentences in the document are traversed in order and a normalized, symmetric similarity measure (see Chapter 14) is applied on adjacent pairs of sentences. The plot on the right-hand side of the figure shows the change in similarity between the sentences. We can inspect these changes to segment the document into passages when the similarity is low, i.e., a shift in topic occurs. An alternative approach to this segmentation is to simply use paragraphs if the document being operated on contains that information, although most of the time this is not the case. This rudimentary partitioning strategy is a task in
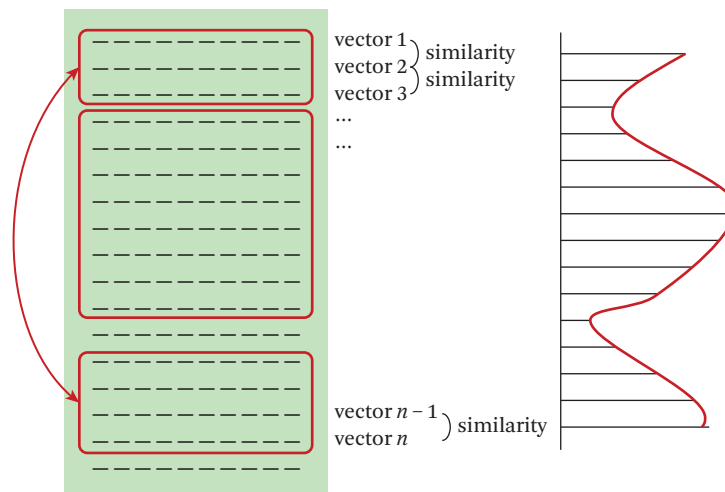
Figure 16.1   Segmenting a document into passages with a similarity-based discourse analysis.
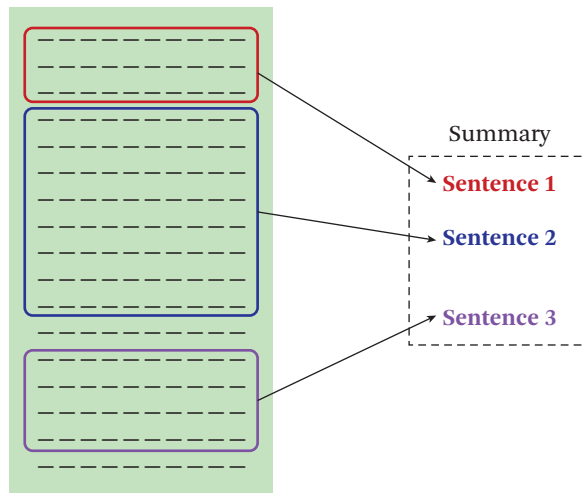
Figure 16.2   **Text summarization using maximum marginal relevance to select one sentence from each passage as a summary.**

discourse analysis (a subfield of NLP). Discourse analysis deals with sequences of sentences as opposed to only one sentence.

Now that we have our passages, how can we remove redundancy and increase diversity in the resulting summarization during step two? The technique **maximal marginal relevance** (MMR) reranking can be applied to our problem. Essentially, this algorithm greedily reranks each sentence in the current passage, outputting only the top few as a summary. Figure 16.2 shows the output of the algorithm when we only select one sentence from each passage.

The MMR algorithm is as follows. Assume we are given an original list R and a profile p to construct the set of selected sentences S (where $|S| \ll |R|$). R is a partitioned chunk of sentences in the document we wish to summarize. The profile p determines what is exactly meant by "relevance." Originally, the MMR formula was applied to documents returned from an information retrieval system (hence the term *reranking*). Documents were selected based on their marginal relevance to a query (which is our variable p) in addition to non-redundancy to already-selected documents. Since our task deals with sentence retrieval, p can be a user profile (text about the user), the entire document itself, or it could even be a query formulated by the user.

According to marginal relevance, the next sentence s_i to be added into the selected list S is defined as

$$s_i = \arg \max_{s \in R \setminus S} \left\{ (1 - \lambda) \cdot \text{sim}_1(s, p) - \lambda \cdot \arg \max_{s_j \in S} \text{sim}_2(s, s_j) \right\}. \qquad (16.1)$$

The $R \setminus S$ notation may be read as "R set minus S", i.e., all the elements in R that are not in S. The MMR formulation uses $\lambda \in [0, 1]$ to control relevance versus redundancy; the positive relevance score is discounted by the amount of redundancy (similarity) to the already-selected sentences. Again, the two similarity metrics may be any normalized, symmetric measures. The simplest instantiation for the similarity metric would be cosine similarity, and this is in fact the measure used in Carbonell and Goldstein [1998].

The algorithm may be terminated once an appropriate number of words or sentences is in S, or if the score $\text{sim}_1(s, p)$ is below some threshold. Furthermore, the similarity functions may be tweaked as well. Could you think of a way to include sentence position in the similarity function? That is, if a sentence is far away (dissimilar) from the candidate sentence, we could subtract from the similarity score. Even better, we could interpolate the two values into a new similarity score such as

$$\text{sim}(s, s') = \alpha \cdot \text{sim}_{\text{cosine}}(s, s') + (1 - \alpha) \cdot \left(1 - \frac{d(s, s')}{\max d(s, \cdot)} \right), \qquad (16.2)$$

where $\alpha \in [0, 1]$ controls the weight between the regular cosine similarity and the distance measure, and $d(\cdot, \cdot)$ is the number of sentences between the two parameters. Note the "one minus" in front of the distance calculation, since a smaller distance implies a greater similarity.

Of course, λ in the MMR formula is also able to be set. In fact, for multi-document summarization, Das and Martins [2007] suggests starting out with $\lambda = 0.3$ and then slowly increasing to $\lambda = 0.7$. The reasoning behind this is to first emphasize novelty and then default to relevance. This should remind you of the exploration-exploitation tradeoff discussed in Chapter 11.

16.3 Abstractive Text Summarization

An abstractive summary creates sentences that did not exist in the original document or documents. Instead of a document vector, we will use a language model to represent the original text. Unlike the document vector, our language model gives us a principled way in which to generate text. Imagine we tokenized our document with unigram words. In our language model, we would have a parameter representing the probability of each word occurring. To create our own text, we will draw words from this probability distribution.
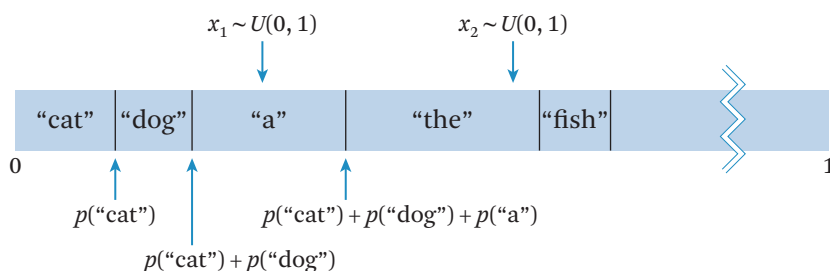
Figure 16.3   **Drawing words from a unigram language model.**

Say we have the unigram language model θ estimated on a document we wish to summarize. We wish to draw words $w_1, w_2, w_3, \ldots$ from θ that will comprise our summary. We want the word w_i to occur in our summary with about the same probability it occurred in the original document—this is how our generated text will approximate the longer document. Figure 16.3 depicts how we can accomplish this task. First, we create a list of all our parameters and incrementally sum their probabilities; this will allow us to use a random number on $[0, 1]$ to choose a word w_i. Simply, we get a uniform random floating point number between zero and one. Then, we iterate through the words in our vocabulary, summing their probabilities until we get to the random number. We output the term and repeat the process.

In the example, imagine we have the following values:

$p(\text{cat})$	0.010
$p(\text{cat}) + p(\text{dog})$	0.018
$p(\text{cat}) + p(\text{dog}) + p(\text{a})$	0.038
$\vdots$	$\vdots$
$p(\text{cat}) + p(\text{dog}) + p(\text{a}) + \cdots + p(\text{zap})$	1.0

Say we generate a random number x_1 using a uniform distribution on $[0, 1]$. This is denoted as $x_1 \sim \mathcal{U}(0, 1)$. Now imagine that $x_1 = 0.032$. We go to the cumulative point 0.032 in our distribution and output "a". We can repeat this process until our summary is of a certain length or until we generate an end-of-sentence token `</s>`.

At this point, you may be thinking that the text we generate will not make any sense—that is certainly true if we use a unigram language model since each word is generated independently without regard to its context. If more fluent language is required, we can use an n-gram language model, where $n > 1$. Instead of each word being independently generated, the new word will depend on the previous $n - 1$

words. The generation will work the same way as in the unigram case: say we have the word w_i and wish to generate w_{i+1} with a bigram language model. Our bigram language model gives us a distribution of words that occur after w_i and we draw the next word from there in the same way depicted in Figure 16.3.

The sentence generation from a bigram language model proceeds as follows: start with (e.g.) *The*. Then, pick from the distribution $p(w \mid \text{The})$ using the cumulative sum technique. The next selected word could be *cat*. Then, we use the distribution $p(w \mid \text{cat})$ to find the next w, and so on. While the unigram model only had one "sum table" (Figure 16.3), the bigram case needs V tables, one for each w' in $p(w \mid w')$.

Typically, the n-value will be around three to five depending on how much original data there is. We saw what happened when n is too small; we get a jumble of words that don't make sense together. But we have another problem if n is too large. Consider the extreme case where $n = 20$. Then, given 19 words, we wish to generate the next one using our 20-gram language model. It's very unlikely that those 19 words occurred more than once in our original document. That means there would only be one choice for the 20th word. Because of this, we would just be reproducing the original document, which is not a very good summary. In practice, we would like to choose an n-gram value that is large enough to produce coherent text yet small enough to not simply reproduce the corpus.

There is one major disadvantage to this abstractive summarization method. Due to its nature, a given word only depends on the n surrounding words. That is, there will be no long-range dependencies in our generated text. For example, consider the following sentence generated from a trigram language model:

> *They imposed a gradual smoking ban on virtually all corn seeds planted are hybrids*.

All groups of three words make sense, but as a whole the sentence is incomprehensible; it seems the writer changed the topic from a smoking ban to hybrid crops mid-sentence. In special cases, when we restrict the length of a summary to a few words when summarizing highly redundant text, such a strategy appears to be effective as shown in the micropinion summarization method described in Ganesan et al. [2012].

16.3.1 Advanced Abstractive Methods

Some advanced abstractive methods rely more heavily on natural language processing to build a model of the document to summarize. **Named entity recognition** can be used to extract people, places, or businesses from the text. **Dependency parsers** and other syntactic techniques can be used to find the relation between the entities

and the actions they perform. Once these actors and roles are discovered, they are stored in some internal representation. To generate the actual text, some representations are chosen from the parsed collection, and English sentences are created based on them; this is called **realization**.

Such realization systems have much more fine-grained control over the generated text than the basic abstractive language model generator described above. A templated document structure may exist (such as intro→paragraph 1→paragraph 2→conclusion), and the structures are chosen to fill each spot. This control over text summarization and layout enables an easily-readable summary since it has a natural topical flow. In this environment, it would be possible to merge similar sentences with conjunctions such as *and* or *but*, depending on the context. To make the summary sound even more natural, pronouns can be used instead of entity names if the entity name has already been mentioned. Below are examples of these two operations:

> *Gold prices fell today. Silver prices fell today.* → *Gold and silver prices fell today.*
> *Company A lost* 9.43% *today. Company A was the biggest mover.* → *Company A lost* 9.43% *today. It was the biggest mover.*

Even better would be

> *Company A was today's biggest mover, losing* 9.43%.

These operations are possible since the entities are stored in a structured format. For more on advanced natural language generation, we suggest Reiter and Dale [2000], which has a focus on practicality and implementation.

16.4 Evaluation of Text Summarization

In extractive summarization, representative sentences were selected from passages in the text and output as a summary. This solution is modeled as an information retrieval problem, and we can evaluate it as such. Redundancy is a critical issue, and the MMR technique we discussed attempts to alleviate it. When doing our evaluation, we should consider redundant sentences to be irrelevant, since the user does not want to read the same information twice. For a more detailed explanation of IR evaluation measures, please consult Chapter 9.

For full output scoring, we should prefer IR evaluation metrics that do not take into account result position. Although our summary is generated by ranked sentences per passage, the entire output is not a ranked list since the original document is composed of multiple passages. Therefore we can use precision, recall, and F_1 score.

It is possible to rank the passage scoring retrieval function using position-dependent metrics such as average precision or NDCG, but with the final output this is not feasible. Thus we need to decide whether to evaluate the passage scoring or the entire output (or both). Entire output scoring is likely more useful for actual users, while passage scoring could be useful for researchers to fine-tune their methods.

In abstractive summarization, we can't use the IR measures since we don't have a fixed set of candidate sentences. How can we compute recall if we don't know the total number of relevant sentences? There is also no intermediate ranking stage, so we also can't use average precision or NDCG (and again, we don't even know the complete set of correct sentences).

A laborious yet accurate evaluation would have human annotators create a gold standard summary. This "perfect" summary would be compared with the generated one, and some measure (e.g., ROUGE) would be used to quantify the difference. For the comparison measure, we have many possibilities—any measure that can compare two groups of text would be potentially applicable. For example, we can use the cosine similarity between the gold standard and generated summary. Of course, this has the downside that fluency is completed ignored (using unigram words). An alternative means would be to learn an n-gram language model over the gold standard summary, and then calculate the log-likelihood of the generated summary. This can ensure a basic level of fluency at the n-gram level, while also producing an interpretable result. Other comparisons between two probability distributions would also be applicable, such as KL-divergence.

The overall effectiveness of a summary can be tested if users read a summary and then answer questions about the original text. Was the summary able to capture the important information that the evaluator needs? If the original text was an entire textbook chapter, could the user read a three-paragraph summary and obtain sufficient information to answer the provided exercises? This is the only metric that can be used for both extractive and abstractive measures. Using a language model to score an extractive summary vs. an abstractive one would likely be biased towards the extractive one since this method contains phrases directly from the original text, giving it a very high likelihood.

16.5 Applications of Text Summarization

At the beginning of the chapter, we've already touched on a few summarization applications; we mentioned news articles, retrieval results, and opinion summarization. Summarization saves users time from manually reading the entire corpus while simultaneously enhancing preexisting data with summary "annotations."

The aspect opinion analysis mentioned earlier segments portions of user reviews into speaking about a particular topic. We can use this topic analysis to collect passages of text into a large group of comments on one aspect. Instead of describing this aspect with sorted unigram words, we could run a summarizer on each topic, generating readable text as output. These two methods complement each other, since the first step finds what aspects the users are interested in, while the second step conveys the information.

A theme in this book is the union of both structured and unstructured data, mentioned much more in detail in Chapter 19. Summarization is an excellent example of this application. For example, consider a financial summarizer with text reports from the Securities and Exchange Commission (SEC) as well as raw stock market data. Summarizing both these data sources in one location would be very valuable for (e.g.) mutual fund managers or other financial workers. Being able to summarize (in text) a huge amount of structured trading data could reveal patterns that humans would otherwise be unaware of—this is an example of **knowledge discovery**.

E-discovery (electronic discovery) is the process of finding relevant information in litigation (lawsuits and court cases). Lawyers rely on e-discovery to sift through vast amounts of textual information to build their case. The Enron email dataset[1] is a well-known corpus in this field. Summarizing email correspondence between two people or a department lets investigators quickly decide whether they'd like to dig deeper in a particular area or try another approach. In this way, summarization and search are coupled; search allows a subset of data to be selected that is relevant to a query, and the summarization can take the search results and quickly explain them to the user. Finally, linking email correspondence together (from sender to receivers) is a structured complement to the unstructured text content of the email itself.

Perhaps of more interest to those reading this book is the ability to summarize research from a given field. Given proceedings from a conference, could we have a summarizer explain the main trends and common approaches? What was most novel compared to previous conferences? When writing your own paper, can you write everything except the introduction and related work? The introduction is an overview summary of your paper. Related work is mostly a summary of papers similar to yours.

---

1. https://www.cs.cmu.edu/~./enron/

Bibliographic Notes and Further Reading

As mentioned in this chapter, Das and Martins [2007] is a comprehensive survey on summarization techniques. Additionally, Nenkova and McKeown [2012] is a valuable read. For applications, latent aspect rating analysis [Wang et al. 2010], [Wang et al. 2011] is a form of summarization applied to product reviews. We mention this particular application in more detail in Chapter 18. A typical extractive summarizer is presented in Radev et al. [2004], a typical abstractive summarizer is presented in Ganesan et al. [2010], and evaluation suggestions are presented in Steinberger and Jezek [2009]. The MMR algorithm was originally described in Carbonell and Goldstein [1998]. For advanced NLG (natural language generation) techniques, a good starting point is Reiter and Dale [2000].

Exercises

16.1. Do you think one summarization method (extractive or abstractive) would perform better on a small dataset? How about a large dataset? Justify your reasoning.

16.2. Explain how you can improve the passage detection by looking beyond only the adjacent sentences. How would you implement this?

16.3. Write a basic passage segmenter in META. As input, take a document and extract the sentences into a vector with a built-in tokenizer. Segment the vector into passages using a similarity algorithm.

16.4. Now that you have a document segmented into passages, use META to set up a search engine over each passage, where you treat passages as individual documents. Ensure that you have enough sentences per passage. You many need to tweak your previous answer to achieve this.

16.5. With your passage search engine, find a representative sentence from each passage to create a summary for the original document.

16.6. Use META's language model to learn a distribution of words over a document you wish to summarize.

16.7. Add a `generate` function to the language model. It should take a context ($n - 1$ terms) and generate the nth term. Use the calculation described in this chapter to generate the next word.

16.8. Summarize the input document using the generator. Experiment with different stopping criteria. Which seems to work the best?

16.9. Create some simple post-processing rules for natural language generation realization. The examples we gave in the text were sentence joining and pronoun insertion. What else can you think of?

16.10. Explain how we can combine text summarization and topic modeling to create a powerful exploratory text mining application.

16.11. What can we accomplish by interpolating a language model distribution for an abstractive summarizer with another probability distribution, perhaps from existing summaries?

Topic Analysis

This chapter is about topic mining and analysis, covering a family of unsupervised text mining techniques called probabilistic topic models that can be used to discover latent topics in text data.

A topic is something that we all understand intuitively, but it's actually not easy to formally define it. Roughly speaking, a topic is the main idea discussed in text data, which may also be regarded as a theme or subject of a discussion or conversation. A topic can have different granularities. For example, we can talk about the topic of a sentence, the topic of an article, the topic of a paragraph, or the topic of all the research articles in a library. Different granularities of topics have different applications.

There are many applications that require discovery and analysis of topics in text. For example, we might be interested in knowing about what Twitter users are talking about today. Are they talking about NBA sports, international events, or another topic? We may also be interested in knowing about research topics; one might be interested in knowing the current research topics in data mining, and how they are different from those five years ago. To answer such questions, we need to discover topics in the data mining literature, including specifically topics in today's literature and those in the past so that we can make a comparison.

We might also be interested in knowing what people like about some products, such as smartphones. This requires discovering topics in both positive reviews and negative reviews. Or, perhaps we're interested in knowing what the major topics debated in a presidential election are. All these have to do with discovering topics in text and analyzing them. How to do this is a main topic of this chapter.

We can view a topic as describing some knowledge about the world as shown in Figure 17.1. From text data, we want to discover a number of topics which can provide a description about the world. That is, a topic tells us something about the world (e.g., about a product or a person).

Besides text data, we often also have some non-text data which can be used as additional context for analyzing the topics. We might know the time associated
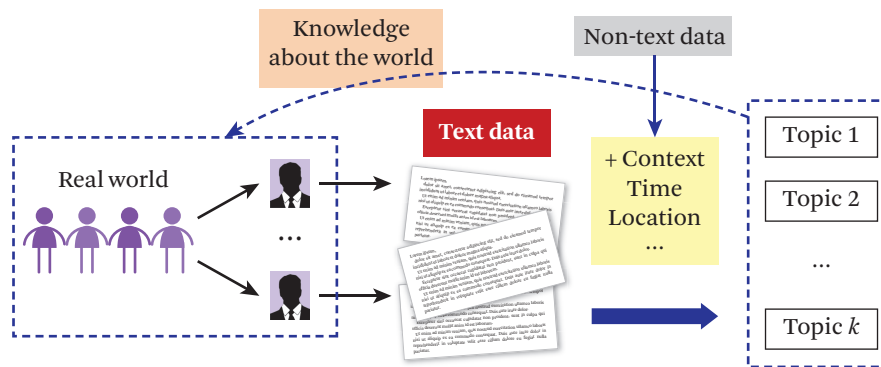
Figure 17.1   **Mining topics as knowledge about the world.**

with the text data, or locations where the text data were produced, or the authors of the text, or the sources of the text. All such metadata (or context variables) can be associated with the topics that we discover, and we can then use these context variables to analyze topic patterns.

For example, looking at topics over time, we would be able to discover whether there's a trending topic or some topics might be fading away. Similarly, looking at topics in different locations might help reveal insights about people's opinions in different locations.

Let's look at the tasks of topic mining and analysis. As shown in Figure 17.2, topic analysis first involves discovering a number of topics. In this case, there are k topics. We also would like to know which topics are covered in which documents, and to what extent. For example, in Doc 1, the visualization shows that Topic 1 is well covered while Topic 2 and Topic k are covered with a small portion. Doc 2, on the other hand, covered Topic 2 very well but it did not cover Topic 1 at all. It also covers Topic k to some extent. Thus, there are generally two different tasks or subtasks: the first is to discover the k topics from a collection of text; the second task is to figure out which documents cover which topics to what extent.

More formally, we can define the problem as shown in Figure 17.3. First, we have as input a collection of N text documents. Here we can denote the text collection as C, and denote a text article as d_i. We also need to have as input the number of topics, k, though this number may be potentially set automatically based on data characteristics. However, in the techniques that we will discuss in this chapter, we need to specify a number of topics.
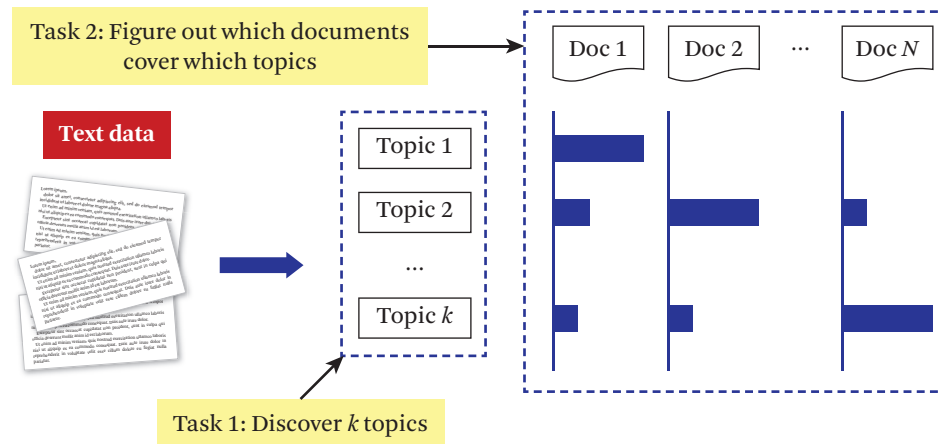
Figure 17.2 The task of topic mining.

- Input
  - A collection of N text documents $C = \{d_1, \ldots, d_N\}$
  - Number of topics: k
- Output
  - k topics: $\{\theta_1, \ldots, \theta_k\}$
  - Coverage of topics in each d_i: $\{\pi_{i1}, \ldots, \pi_{ik}\}$ $\quad \sum_{j=1}^{k} \pi_{ij} = 1$
  - π_{ij} = prob of d_i covering topic θ_j

How to define θ_j?

Figure 17.3 Formal definition of topic mining tasks

The output includes the k topics that we would like to discover denoted by $\theta_1, \ldots, \theta_k$, and the coverage of topics in each document of d_i, which is denoted by π_{ij}. π_{ij} is the probability of document d_i covering topic θ_j. For each document, we have a set of such π values to indicate to what extent the document covers each topic. We assume that these probabilities sum to one, which means that we assume a document won't be able to cover other topics outside of the topics we discovered.

Now, the next question is, how do we define a topic θ_i? Our task has not been completely defined until we define exactly what θ is. In the next section we will first discuss the simplest way to define a topic (as a term).
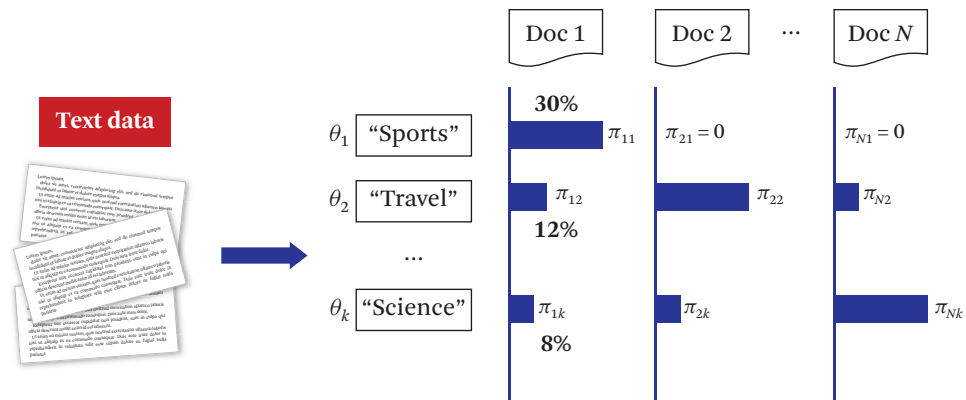
Figure 17.4 A term as a topic.

17.1 Topics as Terms

The simplest, natural way to define a topic is just as a term. A term can be a word or a phrase. For example, we may have terms like *sports*, *travel*, or *science* to denote three separate topics covered in text data, as shown in Figure 17.4.

If we define a topic in this way, we can then analyze the coverage of such topics in each document based on the occurrences of these topical terms. A possible scenario may look like what's shown in Figure 17.4: 30% of the content of Doc 1 is about sports, and 12% is about travel, etc. We might also discover Doc 2 does not cover sports at all. So the coverage π_{21} is zero.

Recall that we have two tasks. One is to discover the topics and the other is to analyze coverage. To solve the first problem, we need to mine k topical terms from a collection. There are many different ways to do that. One natural way is to first parse the text data in the collection to obtain candidate terms. Here candidate terms can be words or phrases. The simplest case is to just take each word as a term. These words then become candidate topics.

Next, we will need to design a scoring function to quantify how good each term is as a topic. There are many things that we can consider when designing such a function with a main basis being the statistics of terms. Intuitively, we would like to favor representative terms, meaning terms that can represent a lot of content in the collection. That would mean we want to favor a frequent term. However, if we simply use the frequency to design the scoring function, then the highest scored terms would be general terms or function words like *the* or *a*. Those terms occur

very frequently in English, so we also want to avoid having such words on the top. That is, we would like to favor terms that are *fairly* frequent but not *too* frequent.

A specific approach to achieving our goal is to use TF-IDF weighting discussed in some previous chapters of the book on retrieval models and word association discovery. An advantage of using such a statistical approach to define a scoring function is that the scoring function would be very general and can be applied to any natural language, any text. Of course, when we apply such an approach to a particular problem, we should always try to leverage some domain-specific heuristics. For example, in news we might favor title words because the authors tend to use the title to describe the topic of an article. If we're dealing with tweets, we could also favor hashtags, which are invented to denote topics.

After we have designed the scoring function, we can discover the k topical terms by simply picking the k terms with the highest scores. We might encounter a situation where the highest scored terms are all very similar. That is, they are semantically similar, or closely related, or even synonyms. This is not desirable since we also want to have a good coverage over *all* the content in the collection, meaning that we would like to remove redundancy. One way to do that is to use a greedy algorithm, called Maximal Marginal Relevance (MMR) re-ranking.

The idea is to go down the list based on our scoring function and select k topical terms. The first term, of course, will be picked. When we pick the next term, we will look at what terms have already been picked and try to avoid picking a term that's too similar. So while we are considering the ranking of a term in the list, we are also considering the redundancy of the candidate term with respect to the terms that we already picked. With appropriate thresholding, we can then get a balance of redundancy removal and picking terms with high scores. The MMR technique is described in more detail in Chapter 16.

After we obtain k topical terms to denote our topics, the next question is how to compute the coverage of each topic in each document, π_{ij}. One solution is to simply count occurrences of each topical term as shown in Figure 17.5. So, for example, *sports* might have occurred four times in document d_i, and *travel* occurred twice. We can then just normalize these counts as our estimate of the coverage probability for each topic. The normalization is to ensure that the coverage of each topic in the document would add to one, thus forming a distribution over the topics for each document to characterize coverage.

As always, when we think about an idea for solving a problem, we have to ask the following questions: how effective is the solution? Is this the best way of solving problem? In general, we have to do some empirical evaluation by using actual data sets and to see how well it works. However, it is often also instructive to analyze
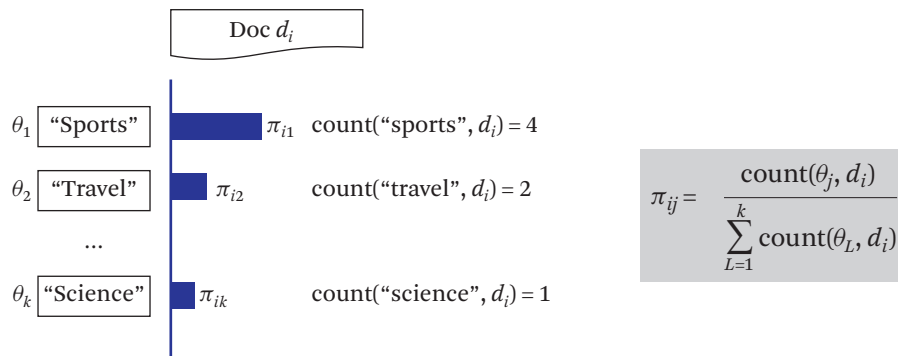
Figure 17.5   Computing topic coverage when a topic is a term.
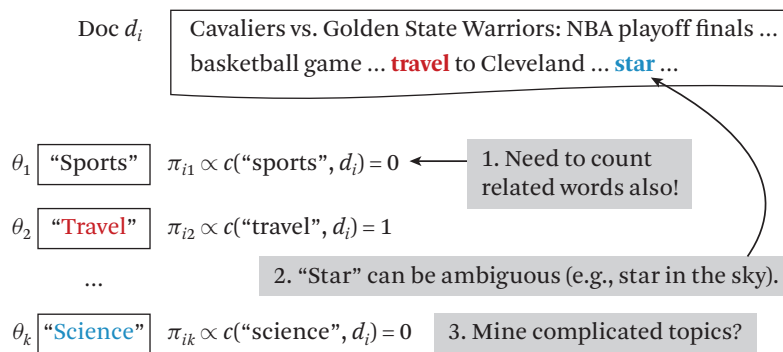
Figure 17.6   Problems in representing a topic as a term.

some specific examples. So now let's examine the simple approach we have been discussing with a sample document in Figure 17.6.

Here we have a text document that's about an NBA basketball game. In terms of the content, it's about sports, but if we simply count these words that represent our topics, we will find that the word *sports* actually did not occur in the article, even though the content is about sports. Since the count of *sports* is zero, the coverage of sports would be estimated as zero. We may note that the term *science* also did not occur in the document, and so its estimate is also zero, which is intuitively what we want since the document is not about science. However, giving a zero probability to *sports* certainly is a problem because we know the content is about sports. What's worse, the term *travel* actually occurred in the document, so when we estimate

the coverage of the topic *travel*, we would have a non-zero count, higher than the estimated coverage of *sports*. This obviously is also not desirable.

Our analysis of this simple example thus reveals a few problems of this approach. First, when we count what words belong to a topic, we also need to consider related words. We cannot simply just count the extracted term denoting a topic (e.g., *sports*), which may not occur at all in a document about the topic. On the other hand, there are many words related to the topic like *basketball* and *game*, which should presumably also be counted when estimating the coverage of a topic.

The second problem is that a word like *star* is ambiguous. While in this article it means a basketball star, it might also mean a star on the sky in another context, so we need to consider the uncertainty of an ambiguous word.

Finally, a main restriction of this approach is that we have only one term to describe the topic, so it cannot really describe complicated topics. For example, a very specialized topic in sports would be harder to describe by using just a word or one phrase. We need to use more words.

A key take-away point from analyzing this simple example is that there are three general problems with our simple approach of defining a topic as a single term: first, it lacks expressive power. It can only represent the simple general topics, but cannot represent the complicated topics that might require more words to describe. Second, it's incomplete in vocabulary coverage, meaning that the topic itself is only represented as one term. It does not suggest what other terms are related to the topic, making it impossible to estimate the contribution of related words to the coverage of a topic. Finally, there is a problem due to ambiguity of words. A topical term or related term can be ambiguous. In the next section, we will discuss an improved representation of a topic (as a distribution over words) that can address these problems.

17.2 Topics as Word Distributions

A natural idea to address the problems of using one single term to denote a topic is to use more words to describe the topic, which would immediately address the first problem of lack of expressive power. When we have more words that we can use to describe the topic, we would be able to describe complicated topics. To address the second problem (of how to involve related words), we need to introduce weights on words. This is what allows us to distinguish subtle differences in topics, and to introduce semantically related words in a quantitative manner. Finally, to solve the problem of word ambiguity, we need to "split" ambiguous words to allow them to be used to (potentially) describe multiple topics.

θ_1 "Sports" θ_2 "Travel"   ... θ_k "Science"

$P(w\,|\,\theta_1)$ $P(w\,|\,\theta_2)$ $P(w\,|\,\theta_k)$

sports 0.02	travel 0.05	science 0.04
game 0.01	attraction 0.03	scientist 0.03
basketball 0.005	trip 0.01	spaceship 0.006
football 0.004	flight 0.004	telescope 0.004
play 0.003	hotel 0.003	genomics 0.004
star 0.003	island 0.003	star 0.002
...	...	...
nba 0.001	culture 0.001	genetics 0.001
...	...	...
travel 0.0005	play 0.0002	travel 0.00001
...	...	...

$$\sum_{w\in V} p(w\,|\,\theta_i) = 1 \quad \text{Vocabulary set: } V = \{w_1, w_2, ... \}$$

Figure 17.7   **Topic as a word distribution.**

It turns out that all these can be elegantly achieved by using a probability distribution over words (i.e., a unigram language model) to denote a topic, as shown in Figure 17.7. Here, you see that for every topic, we have a word distribution over all the words in the vocabulary.

For example, the high probability words for the topic "sports" are *sports*, *game*, *basketball*, *football*, *play*, and *star*. These are all intuitively sports-related terms whose occurrences should contribute to the likelihood of covering the topic "sports" in an article. Note that, in general, the distribution may give all the words a non-zero probability since there is always a very very small chance that even a word not so related to the topic would be mentioned in an article about the topic. Note also that these probabilities for all the words always sum to one for each topic, thus forming a probability distribution over all the words.

Such a word distribution represents a topic in that if we sample words from the distribution, we tend to see words that are related to the topic. It is also interesting to note that as a very special case, if the probability of the mass is concentrated entirely on just one word, e.g., *sports*, then the word distribution representation of a topic would degenerate to the simplest representation of a topic as just one single word discussed before. In this sense, the word distribution representation is a natural generalization and extension of the single-term representation.

However, representing a topic by a distribution over words can involve many words to describe a topic and model subtle differences of topics. Through adjusting probabilities of different words, we may model variations of the general "sports" topic to focus more on a particular kind of sports such as basketball (where we would expect *basketball* to have a very high probability) or football (where *football* would have a much higher probability than *basketball*).

Similarly, in the distribution for "travel," we see top words like *attraction*, *trip*, *flight*, and so on. In "science," we see *scientist*, *spaceship*, and *genomics*, which are all intuitively related to the corresponding topic. It is important to note that it doesn't mean sports-related terms will necessarily have zero probabilities in a distribution representing the topic "science," but they generally have much lower probabilities.

Note that there are some words that are shared by these topics, meaning that they have reasonably high probabilities for all these topics. For example, the word *travel* occurred in the top-word lists for all the three topics, but with different probabilities. It has the highest probability for the "travel" topic, 0.05, but with much smaller probabilities for "sports" and "science," which makes sense. Similarly, you can see *star* also occurred in "sports" and "science" with reasonably high probabilities because the word is actually related to both topics due to its ambiguous nature. We have thus seen that representing a topic by a word distribution effectively addresses all the three problems of a topic as a single term mentioned earlier.

- It now uses multiple words to describe a topic, allowing us to describe fairly complicated topics.

- It assigns weights to terms, enabling the modeling of subtle differences of semantics in related topics. We can also easily bring in related words together to model a topic and estimate the coverage of the topic.

- Because we have probabilities for the same word in different topics, we can accommodate multiple senses of a word, addressing the issue of word ambiguity.

Next, we examine the task of discovering topics represented in this way. Since the representation is a probability distribution, it is natural to use probabilistic models for discovering such word distributions, which is referred to as probabilistic topic modeling.

When using a word distribution to denote a topic, our task of topic analysis can be further refined based on the formal definition in Figure 17.3 by making each topic a word distribution. That is, each θ_i is now a word distribution, and we have

$$\sum_{w \in V} p(w \mid \theta_i) = 1. \tag{17.1}$$

Naturally, we still have the same constraint on the topic coverage, i.e.,

$$\sum_{j=1}^{k} \pi_{ij} = 1, \forall i. \tag{17.2}$$

As a computation problem, our input is text data, a collection of documents C, and we assume that we know the number of topics, k, or hypothesize that there are k topics in the text data. As part of our input, we also know the vocabulary V, which determines what units would be treated as the basic units (i.e., words) for analysis. In most cases, we will use words as the basis for analysis simply because they are the most natural units, but it is easy to generalize such an approach to use phrases or any other units that we can identify in text, as the basic units and treat them as if they were words. Our output consists of two families of probability distributions. The first is a set of topics represented by a set of θ_i's, each of which is a word distribution. The second is a topic coverage distribution for each document d_i, $\{\pi_{i1}, \ldots, \pi_{ik}\}$.

The question now is how to generate such output from our input. There are potentially many different ways to do this, but here we introduce a general way of solving this problem called a generative model. This is, in fact, a very general idea and a principled way of using statistical modeling to solve text mining problems.

The basic idea of this approach is to first design a generative model for our data, i.e., a probabilistic model to model how the data are generated, or a model that can allow us to compute the probability of how likely we will observe the data we have. The actual data aren't necessarily (indeed often unlikely) generated this way, but by assuming the data to be generated in a particular way according to a particular model, we can have a formal way to characterize our data which further facilitates topic discovery.

In general, our model will have some parameters (which can be denoted by Λ); they control the behavior of the model by controlling what kind of data would have high (or low) probabilities. If you set these parameters to different values, the model would behave differently; that is, it would tend to give different data points high (or low) probabilities.

We design the model in such a way that its parameters would encode the knowledge we would like to discover. Then, we attempt to estimate these parameters based on the data (or infer the values of parameters based on the observed data) so as to generate the desired output in the form of parameter values, which we have

designed to denote the knowledge we would like to discover. How exactly we should
fit the model to the data or infer the parameter values based on the data is often a
standard problem in statistics, and there are many different ways to do this as we
discussed briefly in Chapter 2.

Following the idea of using a generative model to solve the specific problem of
discovering topics and topic coverages from text data, we see that our generative
model needs to contain all the k word distributions representing the topics and
the topic coverage distributions for all the documents, which is all the output we
intend to compute in our problem setup. Thus, there will be many parameters in
the model. First, we have $|V|$ parameters for the probabilities of words in each
word distribution, so we have in total $|V|k$ word probability parameters. Second,
for each document, we have k values of π, so we have in total Nk topic coverage
probability parameters. Thus, we have in total $|V|k + Nk$ parameters. Given that
we have constraints on both θ and π, however, the number of free parameters
is smaller at $(|V| - 1)k + N(k - 1)$; in each word distribution, we only need to
specify $|V| - 1$ probabilities and for each document, we only need to specify $k - 1$
probabilities.

Once we set up the model, we can fit its parameters to our data. That means we
can estimate the parameters or infer the parameters based on the data. In other
words, we would like to adjust these parameter values until we give our data set
maximum probability. Like we just mentioned, depending on the parameter values,
some data points will have higher probabilities than others. What we're interested
in is what parameter values will give our data the highest probability.

In Figure 17.8, we illustrate Λ, the parameters, as a one-dimensional variable.
It's oversimplification, obviously, but it suffices to show the idea. The y axis shows
the probability of the data. This probability obviously depends on this setting of Λ,
so that's why it varies as you change Λ's value in order to find Λ^*, the parameter
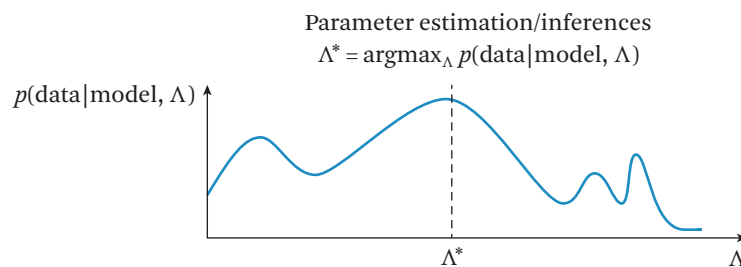
Figure 17.8   Maximum likelihood estimate of a generative model.

settings that maximize the probability of the observed data. Such a search yields our estimate of the model parameters. These parameters are precisely what we hoped to discover from the text data, so we view them as the output of our data mining or topic analysis algorithm.

This is the general idea of using a generative model for text mining. We design a model with some parameter values to describe the data as well as we can. After we have fit the data, we learn parameter values. We treat the learned parameters as the discovered knowledge from text data.

17.3   Mining One Topic from Text

In this section, we discuss the simplest instantiation of a generative model for modeling text data, where we assume that there is just one single topic covered in the text and our goal is to discover this topic.

More specifically, as illustrated in Figure 17.9, we are interested in analyzing each document and discovering a single topic covered in the document. This is the simplest case of a topic model. Our input now no longer has k topics because we know (or rather, specify) that there is only one topic. Since each document can be mined independently, without loss of generality, we further assume that the collection has only one document. In the output, we also no longer have coverage because we assumed that the document has a 100% coverage of the topic we would like to discover. Thus, the output to compute is the word distribution representing this single topic, or probabilities of all words in the vocabulary given by this distribution, as illustrated in Figure 17.9.
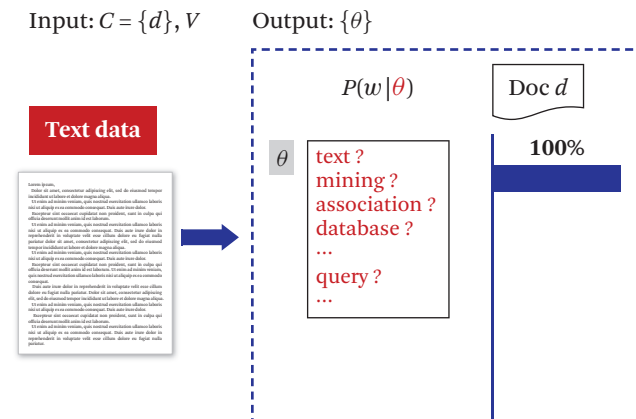
Figure 17.9   The simplest topic model with one topic.

17.3.1   The Simplest Topic Model: Unigram Language Model

When we use a generative model to solve a problem, we start with thinking about what kind of data we need to model and from what perspective. Our data would "look" differently if we use a different perspective. For example, we may view a document simply as a set of words without considering the frequencies of words, which would lead to a bit vector representation as we discussed in the context of the vector space retrieval model. Such a representation would need a different generative model than if we view the document as a bag of words where we care about the different frequencies of words. In topic analysis, the frequencies of words can help distinguish subtle semantic variations, so we generally should retain the word frequencies.

Once we decide on a perspective to view the data, we will design a specific model for generating the data from the desired perspective, i.e., model the data based on the representation of the data reflecting the desired perspective. The choice of a particular model partly depends on our domain knowledge about the data and partly depends on what kind of knowledge we would like to discover. The target knowledge would determine what parameters we would include in the model since we want our parameters to denote the knowledge interesting to us (after we estimate the values of these parameters).

Here we are interested in discovering a topic represented as a word distribution, so a natural choice of model would be a unigram language model, as in Section 3.4. After we specify the model, we can formally write down the likelihood function, i.e., the probability of the data given the assumed model. This is generally a function of the (unknown) parameters, and the value of the function would vary according to the values of the parameters. Thus, we can attempt to find the parameter values that would maximize the value of this function (the likelihood) given data from the model. Such a way of obtaining parameters is called the Maximum Likelihood Estimate (MLE) as we've discussed previously. Sometimes, it is desirable to also incorporate some additional knowledge (a prior belief) about the parameters that we may have about a particular application. We can do this by using Bayesian estimation of parameters, which seeks a compromise of maximizing the probability of the observed data (maximum likelihood) and being consistent with the prior belief that we impose.

In any case, once we have a generative model, we would be able to fit such a model to our data and obtain the parameter values that can best explain the data. These parameter values can then be taken as the output of our mining process.

Let's follow these steps to design the simplest topic model for discovering a topic from one document; we will examine many more complicated cases later. The model is shown in Figure 17.10 where we see that we have decided to view a

- **Data**: Document $d = x_1 x_2 \ldots x_{|d|}$, $x_i \in V = \{w_1, \ldots, w_M\}$ is a word
- **Model**: Unigram $LM\theta (= \text{topic}) : \{\theta_i = p(w_i \mid \theta)\}, i = 1, \ldots, M; \theta_1 + \cdots + \theta_M = 1$
- **Likelihood function**:

$$p(d \mid \theta) = p(x_1 \mid \theta) \times \cdots \times p(x_{|d|} \mid \theta)$$

$$= p(w_1 \mid \theta)^{c(w_1, d)} \times \cdots \times p(w_M \mid \theta)^{c(w_M, d)}$$

$$= \prod_{i=1}^{M} p(w_i \mid \theta)^{c(w_i, d)} = \prod_{i=1}^{M} \theta_i^{c(w_i, d)}$$

- **ML estimate**: $(\hat{\theta}_1, \ldots, \hat{\theta}_M) \arg\max_{\theta_1, \ldots, \theta_M} p(d \mid \theta) = \arg\max_{\theta_1, \ldots, \theta_M} \prod_{i=1}^{M} \theta_i^{c(w_i, d)}$

Figure 17.10 Unigram language model for discovering one topic.

document as a sequence of words. Each word here is denoted by x_i. Our model is a unigram language model, i.e., a word distribution that denotes the latent topic that we hope to discover. Clearly, the model has as many parameters as the number of words in our vocabulary, which is M in this case. For convenience, we will use θ_i to denote the probability of word w_i. According to our model, the probabilities of all the words must sum to one: $\sum_{i=1}^{M} \theta_i = 1$.

Next, we see that our likelihood function is the probability of generating this whole document according to our model. In a unigram language model, we assume independence in generating each word so the probability of the document equals the product of the probability of each word in the document (the first line of the equation for the likelihood function). We can rewrite this product into a slightly different form by grouping the terms corresponding to the same word together so that the product would be over all the *distinct* words in the vocabulary (instead of over all the positions of words in the document), which is shown in the second line of the equation for the likelihood function.

Since some words might have repeated occurrences, when we use a product over the unique words we must also incorporate the count of a word w_i in document d, which is denoted by $c(w_i, d)$. Although the product is taken over the entire vocabulary, it is clear that if a word did not occur in the document, it would have a zero count ($c(w_i, d) = 0$), and that corresponding term would be essentially absent in the formula, thus the product is still essentially over the words that actually occurred in the document. We often prefer such a form of the likelihood function where the product is over the entire vocabulary because it is convenient for deriving formulas for parameter estimation.

Maximize $p(d \mid \theta)$: $(\hat{\theta}_1, \ldots, \hat{\theta}_M) = \arg\max_{\theta_1, \ldots, \theta_M} p(d \mid \theta) = \arg\max_{\theta_1, \ldots, \theta_M} \prod_{i=1}^{M} \theta_i^{c(w_i, d)}$

Max. Log-Likelihood: $(\hat{\theta}_1, \ldots, \hat{\theta}_M) = \arg\max_{\theta_1, \ldots, \theta_M} \log[p(d \mid \theta)] = \arg\max_{\theta_1, \ldots, \theta_M} \sum_{i=1}^{M} c(w_i, d) \log \theta_i$

Subject to constraint: $\sum_{i=1}^{M} \theta_i = 1$ Use Lagrange multiplier approach

Lagrange function: $f(\theta \mid d) = \sum_{i=1}^{M} c(w_i, d) \log \theta_i + \lambda \left(\sum_{i=1}^{M} \theta_i - 1 \right)$

$\frac{\partial f(\theta \mid d)}{\partial \theta_i} = \frac{c(w_i, d)}{\theta_i} + \lambda = 0 \rightarrow \theta_i = -\frac{c(w_i, d)}{\lambda}$

$\sum_{i=1}^{M} -\frac{c(w_i, d)}{\lambda} = 1 \rightarrow \lambda = -\sum_{i=1}^{M} c(w_i, d) \rightarrow \hat{\theta}_t = p(w_t \mid \hat{\theta}) = \frac{x(w_t, d)}{\sum_{i=1}^{M} c(w_i, d)} = \frac{c(w_t, d)}{|d|}$

Figure 17.11   **Computation of a maximum likelihood estimate for a unigram language model.**

Now that we have a well defined likelihood function, we will attempt to find the parameter values (i.e., word probabilities) that maximize this likelihood function. Let's take a look at the maximum likelihood estimation problem more closely in Figure 17.11. The first line is the original optimization problem of finding the maximum likelihood estimate. The next line shows an equivalent optimization problem with the log-likelihood. The equivalence is due to the fact that the logarithm function results in a monotonic transformation of the original likelihood function and thus does not affect the solution of the optimization problem. Such a transformation is purely for mathematical convenience because after the logarithm transformation our function will become a sum instead of product; the sum makes it easier to take the derivative, which is often needed for finding the optimal solution of this function.

Although simple, this log-likelihood function reflects some general characteristics of a log-likelihood function of some more complex generative models.

- The sum is over all the unique data points (the words in the vocabulary).
- Inside the sum, there's a count of each unique data point, i.e., the count of each word in the observed data, which is multiplied by the logarithm of the probability of the particular unique data point.

At this point, our problem is a well-defined mathematical optimization problem where the goal is to find the optimal solution of a constrained maximization problem. The objective function is the log-likelihood function and the constraint is that all the word probabilities must sum to one. How to solve such an optimization problem is beyond the scope of this book, but in this case, we can obtain a simple analytical solution by using the Lagrange multiplier approach. This is a commonly used approach, so we provide some detail on how it works in Figure 17.11.

We will first construct a Lagrange function, which combines our original objective function with another term that encodes our constraint with the Lagrange multiplier, denoted by λ, introducing an additional parameter. It can be shown that the solution to the original constrained optimization problem is the same as the solution to the new (unconstrained) Lagrange function.

Since there is no constraint involved any more, it is straightforward to solve this optimization problem by taking partial derivatives with respect to all the parameters and setting all of them to zero, obtaining an equation for each parameter.[1] We thus have, in total, $M + 1$ linear equations, corresponding to the M word probability parameters and λ. Note that the equation for the Lagrange multiplier λ is precisely our original constraint. We can easily solve this system of linear equations to obtain the Maximum Likelihood estimate of the unigram language model as

$$p(w_i \mid \hat{\theta}) = \frac{c(w_i, d)}{\sum_{j=1}^{M} c(w_j, d)} = \frac{c(w_i, d)}{|d|}. \tag{17.3}$$

This has a very meaningful interpretation: the estimated probability of a word is the count of each word normalized by the document length, which is also a sum of all the counts of words in the document. This estimate mostly matches our intuition in order to maximize the likelihood: words observed more often "deserve" higher probabilities, and only words observed are "allowed" to have non-zero probabilities (unseen words should have a zero probability). In general, maximum likelihood estimation tends to result in a probability estimated as normalized counts of the corresponding event so that the events observed often would have a higher probability and the events not observed would have zero probability.

While we have obtained an analytical solution to the maximum likelihood estimate in this simple case, such an analytical solution is not always possible; indeed, it is often impossible. The optimization problem of the MLE can often be very

---

1. Zero derivatives are a necessary condition for the function to reach an optimum, but not sufficient. However, in this case, we have only one local optimum, thus the condition is also sufficient.

$p(w|\theta)$
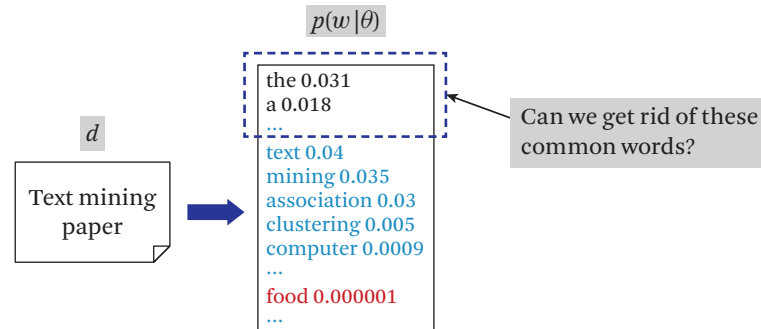
Figure 17.12   Common words dominate the estimated unigram language model.

complicated, and numerical optimization algorithms would generally be needed
to solve the problem.

What would the topic discovered from a document look like? Let's imagine the
document is a text mining paper. In such a case, the estimated unigram language
model (word distribution) may look like the distribution shown in Figure 17.12. On
the top, you will see the high probability words tend to be those very common words,
often function words in English. This will be followed by some content words that
really characterize the topic well like *text* and *mining*. In the end, you also see there
is a small probability of words that are not really related to the topic but might
happen to be mentioned in the document.

As a topic representation, such a distribution is not ideal because the high prob-
ability words are function words, which do not characterize the topic. Giving com-
mon words high probabilities is a direct consequence of the assumed generative
model, which uses one distribution to generate *all* the words in the text. How can
we improve our generative model to down-weight such common words in the es-
timated word distribution for our topic? The answer is that we can introduce a
second background word distribution into the generative model so that the com-
mon words can be generated from this background model, and thus the topic word
distribution would only need to generate the content-carrying topical words. Such a
model is called a mixture model because multiple component models are "mixed"
together to generate data. We discuss it in detail in the next section.

17.3.2   Adding a Background Language Model

In order to solve the problem of assigning highest probabilities to common words
in the estimated unigram language model based on one document, it would be

useful to think about why we end up having this problem. It is not hard to see that the problem is due to two reasons. First, these common words are very frequent in our data, thus any maximum likelihood estimator would tend to give them high probabilities. Second, our generative model assumes all the words are generated from one single unigram language model. The ML estimate thus has no choice but to assign high probabilities to such common words in order to maximize the likelihood.

Thus, in order to get rid of the common words, we must design a different generative model where the unigram language model representing the topic doesn't have to explain all the words in the text data. Specifically, our target topic unigram language model should not have to generate the common words. This further suggests that we must introduce another distribution to generate these common words so that we can have a complete generative model for all the words in the document. Since we intend for this second distribution to explain the common words, a natural choice for this distribution is the background unigram language model. We thus have a mixture model with two component unigram language models, one being the unknown topic that we would like to discover, and one being a background language model that is fixed to assign high probabilities to common words.

In Figure 17.13, we see that the two distributions can be mixed together to generate the text data, with the background model generates common words while the topic language model to generate content-bearing words in the document. Thus, we can expect the discovered (learned) topic unigram language model to
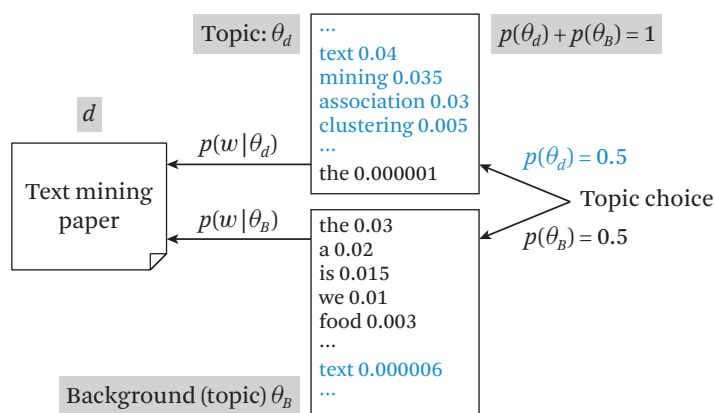
Figure 17.13   A two-component mixture model with a background component model.

assign high probabilities to such content-bearing words rather than the common function words in English.

The assumed process for generating a word with such a mixture model is just slightly different from the generation process of our simplest unigram language. Since we now have two distributions, we have to decide which distribution to use when we generate the word, but each word will still be a sample from one of the two distributions. The text data are still generated in the same way, by generating one word at a time. More specifically, when we generate a word, we first decide which of the two distributions to use. This is controlled by a new probability distribution over the choices of the component models to use (two choices in our case), including specifically the probability of θ_d (using the unknown topic model) and the probability of θ_B (using the known background model). Thus, $p(\theta_d) + p(\theta_B) = 1$.

In the figure, we see that both $p(\theta_d)$ and $p(\theta_B)$ are set to 0.5. This means that we can imagine flipping a fair coin to decide which distribution to use, although in general these probabilities don't have to be equal; one topic could be more likely than another.

The process of generating a word from such a mixture model is as follows. First, we flip a biased coin which would show up as heads with probability $p(\theta_d)$ (and thus as tails with probability $p(\theta_B) = 1 - p(\theta_d)$) to decide which word distribution to use. If the coin shows up as heads, we would use θ_d; otherwise, θ_B. We will use the chosen word distribution to generate a word. This means if we are to use θ_d, we would sample a word using $p(w \mid \theta_d)$, otherwise using $p(w \mid \theta_B)$, as illustrated in Figure 17.13.

We now have a generative model that has some uncertainty associated with the use of which word distribution to generate a word. If we treat the whole generative model as a black box, the model would behave very similarly to our simplest topic model where we only use one word distribution in that the model would specify a distribution over sequences of words. We can thus examine the probability of observing any particular word from such a mixture model, and compute the probability of observing a sequence of words.

Let's assume that we have a mixture model as shown in Figure 17.13 and consider two specific words, *the* and *text*. What's the probability of observing a word like *the* from the mixture model? Note that there are two ways to generate *the*, so the probability is intuitively a sum of the probability of observing *the* in each case. What's the probability of observing *the* being generated using the background model? In order for *the* to be generated in this way, we must have first chosen to use the background model, and then obtained the word *the* when sampling

$$P(\text{"the"}) = p(\theta_d)p(\text{"the"} \mid \theta_d) + p(\theta_B)p(\text{"the"} \mid \theta_B)$$
$$= 0.5 \times 0.000001 + 0.5 \times 0.03$$

$$P(\text{"text"}) = p(\theta_d)p(\text{"text"} \mid \theta_d) + p(\theta_B)p(\text{"text"} \mid \theta_B)$$
$$= 0.5 \times 0.04 + 0.5 \times 0.000006$$

Figure 17.14   **Probability of *the* and *text*.**

a word from the background language model $p(w \mid \theta_B)$. Thus, the probability of observing *the* from the background model is $p(\theta_B)p(\text{the} \mid \theta_B)$, and the probability of observing *the* from the mixture model regardless of which distribution we use would be $p(\theta_B)p(\text{the} \mid \theta_B) + p(\theta_d)p(\text{the} \mid \theta_d)$, as shown in Figure 17.14, where we also show how to compute the probability of *text*.

It is not hard to generalize the calculation to compute the probability of observing any word w from such a mixture model, which would be

$$p(w) = p(\theta_B)p(w \mid \theta_B) + p(w \mid \theta_d)p(\theta_d). \tag{17.4}$$

The sum is over the two different ways to generate the word, corresponding to using each of the two distributions. Each term in the sum captures the probability of observing the word from one of the two distributions. For example, $p(\theta_B)p(w \mid \theta_B)$ gives the probability of observing word w from the background language model. The product is due to the fact that in order to observe word w, we must have (1) decided to use the background distribution (which has the probability of $p(\theta_B)$), and (2) obtained word w from the distribution θ_B (which has the probability of $p(w \mid \theta_B)$). Both events must happen in order to observe word w from the background distribution, thus we multiply their probabilities to obtain the probability of observing w from the background distribution. Similarly, $p(\theta_d)p(w \mid \theta_d)$ gives the probability of observing word w from the topic word distribution. Adding them together gives us the total probability of observing w regardless which distribution has actually been used to generate the word.

Such a form of likelihood actually reflects some general characteristics of the likelihood function of any mixture model. First, the probability of observing a data point from a mixture model is a sum over different ways of generating the word, each corresponding to using a different component model in the mixture model. Second, each term in the sum is a product of two probabilities: one is the probability of selecting the component model corresponding to the term, while

the other is the probability of actually observing the data point from that selected component model. Their product gives the probability of observing the data point when it is generated using the corresponding component model, which is why the sum would give the total probability of observing the data point regardless which component model has been used to generate the data point. As will be seen later, more sophisticated topic models tend to use more than two components, and their probability of generating a word would be of the same form as we see here except that there are more than two products in the sum (more precisely as many products as the number of component models).

Once we write down the likelihood function for one word, it is very easy to see that as a whole, the mixture model can be regarded as a *single* word distribution defined in a somewhat complicated way. That is, it also gives us a probability distribution over words as defined above. Thus, conceptually the mixture model is yet another generative model that also generates a sequence of words by generating each word independently. This is the same as the case of a simple unigram language model, which defines a distribution over words by explicitly specifying the probability of each word.

The main idea of a mixture model is to group multiple distributions together as one model, as shown in Figure 17.15, where we draw a box to "encapsulate" the two distributions to form a single generative model. When viewing the whole box as one model, we can easily see that it's just like any other generative model that would give us the probability of each word. However, how this probability is determined in such a mixture model is quite different from when we have just one unigram language model.
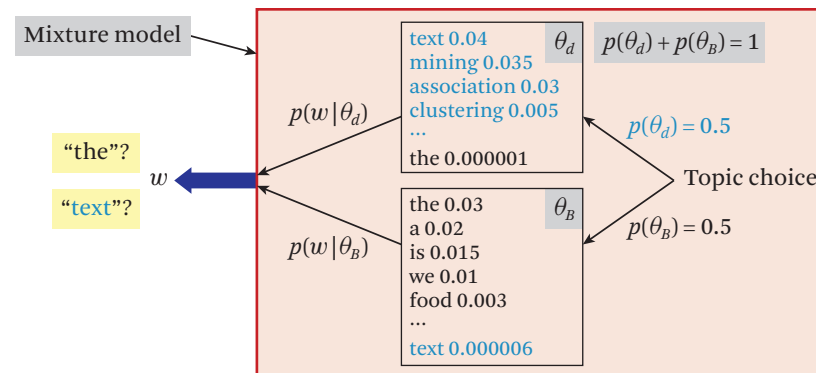
Figure 17.15   The idea of a mixture language model.

It's often useful to examine some special cases of a model as such an exercise can help interpret the model intuitively and reveal relations between simpler models and a more complicated model. In this case, we can examine what would happen if we set the probability of choosing the background component model to zero. It is easy to see that in such a case, the term corresponding to the background model would disappear from the sum, and the mixture model would degenerate to the special case of just one distribution characterizing the topic to be discovered. In this sense, the mixture model is more general than the previous model where we have just one distribution, which can be covered as a special case. Naturally, our reason for using a mixture model is to enforce a non-zero probability of choosing the background language model so that it can help explain the common words in the data and allow our topic word distribution to be more concentrated on content words.

Once we write down the likelihood function, the next question is how to estimate the parameters. As in the case of the single unigram language model, we can use any method (e.g., the maximum likelihood estimator) to estimate the parameters, which can then be regarded as the knowledge that we discover from the text.

- **Data**: Document d
- **Mixture Model**: parameters $\Lambda = (\{p(w \mid \theta_d)\}, \{p(w \mid \theta_B)\}, p(\theta_B), p(\theta_d))$
    - Two unigram LMs: θ_d (the topic of d); θ_B (background topic)
    - Mixing weight (topic choice): $p(\theta_d) + p(\theta_B) = 1$
- **Likelihood function**:

$$p(d \mid \Lambda) = \prod_{i=1}^{|d|} p(x_i \mid \Lambda) = \prod_{i=1}^{|d|} [p(\theta_d)p(x_i \mid \theta_d) + p(\theta_B)p(x_i \mid \theta_B)]$$

$$= \prod_{i=1}^{M} [p(\theta_d)p(w_i \mid \theta_d) + p(\theta_B)p(w_i \mid \theta_B)]^{c(w,d)}$$

- **ML Estimate**: $\Lambda^* = \arg\max_\Lambda p(d \mid \Lambda)$

$$\text{Subject to } \sum_{i=1}^{M} p(w_i \mid \theta_d) = \sum_{i=1}^{M} p(w_i \mid \theta_B) = 1 \quad p(\theta_d) + p(\theta_B) = 1$$

Figure 17.16   **Summary of a two-component mixture model.**

What parameters do we have in such a two-component mixture model? In Figure 17.16, we summarize the mixture of two unigram language models, list all the parameters, and illustrate the parameter estimation problem. First, our data is just one document d, and the model is a mixture model with two components. Second, the parameters include two unigram language models and a distribution (mixing weight) over the two language models. Mathematically, θ_d denotes the topic of document d while θ_B represents the background word distribution, which we can set to a fixed word distribution with high probabilities on common words. We denote all the parameters collectively by Λ. (Can you see how many parameters exactly we have in total?)

The figure also shows the derivation of the likelihood function. The likelihood function is seen to be a product over all the words in the document, which is exactly the same as in the case of a simple unigram language model. The only difference is that inside the product, it's now a sum instead of just one probability as in the simple unigram language model. We have this sum due to the mixture model where we have an uncertainty in using which model to generate a data point. Because of this uncertainty, our likelihood function also contains a parameter to denote the probability of choosing each particular component distribution. The second line of the equation for the likelihood function is just another way of writing the product, which is now a product over all the *unique* words in our vocabulary instead of over all the positions in the document as in the first line of the equation.

We have two types of constraints: one is that all the word distributions must sum to one, and the other constraint is that the probabilities of choosing each topic must sum to one. The maximum likelihood estimation problem can now be seen as a constrained optimization problem where we seek parameter values that can maximize the likelihood function and satisfy all the constraints.

17.3.3 Estimation of a mixture model

In this section, we will discuss how to estimate the parameters of a mixture model. We will start with the simplest scenario where one component (the background) is already completely known, and the topic choice distribution has an equal probability of choosing either the background or the topic word distribution. Our goal is to estimate the unknown topic word distribution where we hope to not see common words with high probabilities. A main assumption is that those common words are generated using the background model, while the more discriminative content-bearing words are generated using the (unknown) topic word distribution,
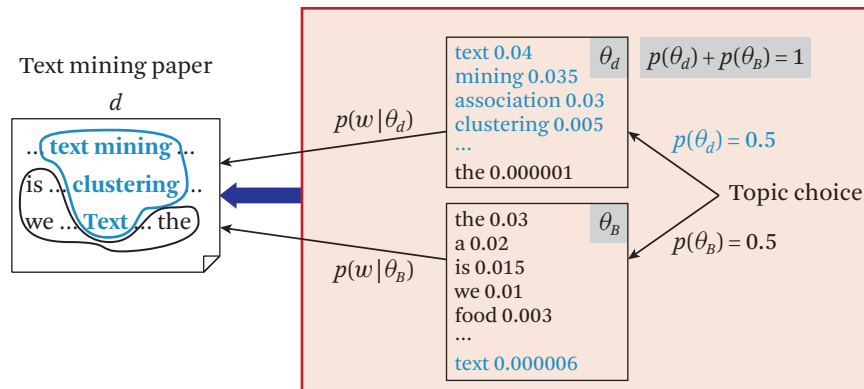
Figure 17.17   **A mixture language model to factor out background words.**

as illustrated in Figure 17.17. This is also the scenario that we used to motivate the use of the mixture model.

Figure 17.18 illustrates such a scenario. In this scenario, the only parameters unknown would be the topic word distribution $p(w \mid \theta_d)$. Thus, we have exactly the same number of parameters to estimate as in the case of a single unigram language model. Note that this is an example of customizing a general probabilistic model so that we can embed an unknown variable that we are interested in computing, while simplifying other parts of the model based on certain assumptions that we can make about them. That is, we assume that we have knowledge about other variables. Setting the background model to a fixed word distribution based on the maximum likelihood estimate of a unigram language model of a large sample of English text is not only feasible, but also desirable since our goal of designing such a generative model is to factor out the common words from the topic word distribution to be estimated. Feeding the model with a known background word distribution is a powerful technique to inject our knowledge about what words are counted as noise (stop words in this case). Similarly, the parameter $p(\theta_B)$ can also be set based on our desired percentage of common words to factor out; the larger $p(\theta_B)$ is set, the more common words would be removed from the topic word distribution. It's easy to see that if $p(\theta_B) = 0$, then we would not be able to remove any common words as the model degenerates to the simple case of using just one distribution (to explain all the words).

Note that we could have assumed that both θ_B and θ_d are unknown, and we can also estimate both by using the maximum likelihood estimation, but in such a case, we would no longer be able to guarantee that we will obtain a distribution θ_B that

Adjust θ_d to maximize $p(d|\Lambda)$
(all other parameters are known)

Would the ML estimate demote background words in θ_d?

d

... **text mining** ...
is ... **clustering** ...
we ... **Text** ... the

text ?
mining ?
association ?
clustering ?
...
the ?

θ_d $p(\theta_d) + p(\theta_B) = 1$

$p(\theta_d) = 0.5$

Topic choice

the 0.03
a 0.02
is 0.015
we 0.01
food 0.003
...
text 0.000006

θ_B $p(\theta_B) = 0.5$
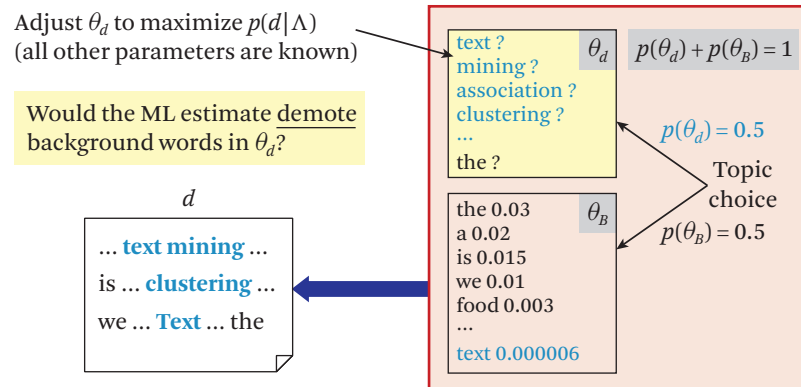
Figure 17.18   Estimation of one topic language model.

assigns high probabilities to common words. For our application scenario (i.e., factoring out common words), it is more appropriate to pre-set the background word distribution to bias the model toward allocating the common words to the background word distribution, and thus allow the topic word distribution to focus more on the content words as we will further explain.

If we view the mixture model in Figure 17.18 as a black box, we would notice that it actually now has exactly the same number of parameters (indeed, the same parameters) as the simplest single unigram language model. However, the mixture model gives us a different likelihood function which intuitively requires θ_d to work together optimally with the fixed background model θ_B to best explain the observed document. It might not be obvious why the constraint of "working together" with the given background model would have the effect of factoring out the common words from θ_d as it would require understanding the behavior of parameter estimation in the case of a mixture model, which we explain in the next section.

17.3.4   Behavior of a Mixture Model

In order to understand some interesting behaviors of mixture models, we take a look at a very simple case, as illustrated in Figure 17.19. Although the example is very simple, the observed patterns here actually are applicable to mixture models in general.

Let's assume that the probability of choosing each of the two models is exactly the same. That is, we will flip a fair coin to decide which model to use. Furthermore, we will assume that there are precisely two words in our vocabulary: *the* and *text*.

$d =$ | **text** the

Likelihood:

$$P(\text{“text”}) = p(\theta_d)p(\text{“text”}|\theta_d) + p(\theta_B)p(\text{“text”}|\theta_B)$$
$$= 0.5 * p(\text{“text”}|\theta_d) + 0.5 * 0.1$$

$$P(\text{“the”}) = 0.5 * p(\text{“the”}|\theta_d) + 0.5 * 0.9$$

$$p(d|\Lambda) = p(\text{“text”}|\Lambda)p(\text{“the”}|\Lambda)$$
$$= [0.5 * p(\text{“text”}|\theta_d) + 0.5 * 0.1] \times$$
$$[0.5 * p(\text{“the”}|\theta_d) + 0.5 * 0.9]$$

| text ? | θ_d |
| the ? | |

$p(\theta_d) = 0.5$

| the 0.9 | θ_B |
| text 0.1 | |

$p(\theta_B) = 0.5$
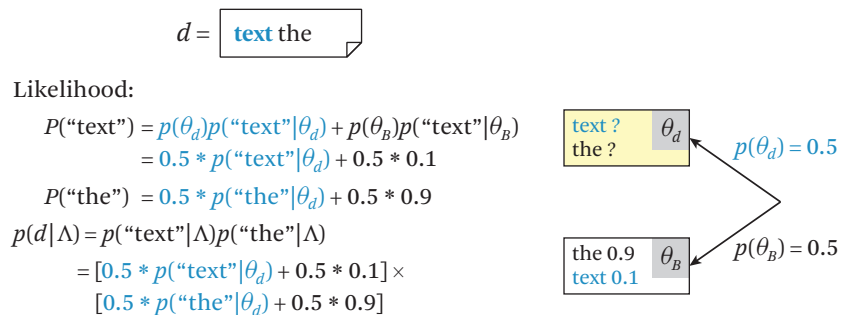
How can we set $p(\text{“text”}|\theta_d)$ and $p(\text{“the”}|\theta_d)$ to maximize it?

Note that $p(\text{“text”}|\theta_d) + p(\text{“the”}|\theta_d) = 1$

Figure 17.19   **Illustration of behavior of mixture model.**

Obviously this is a naive oversimplification of the actual text, but it's useful to ex-
amine the behavior in such a special case. We further assume that the background
model gives probability of 0.9 to *the* and 0.1 to *text*.

We can write down the likelihood function in such a case as shown in Fig-
ure 17.19. The probability of the two-word document is simply the product of the
probability of each word, which is itself a sum of the probability of generating the
word with each of the two distributions. Since we already know all the parameters
except for the θ_d, the likelihood function has just two unknown variables, $p(\text{the} \mid \theta_d)$
and $p(\text{text} \mid \theta_d)$. Our goal of computing the maximum likelihood estimate is to find
out for what values of these two probabilities the likelihood function would reach
its maximum.

Now the problem has become one to optimize a very simple expression with two
variables as shown in Figure 17.20. Note that the two probabilities must sum to
one, so we have to respect this constraint. If there were no constraint, we would
have been able to set both probabilities to their maximum value (which would be
1.0) to maximize the likelihood expression. However, we can't do this because we
can't give both words a probability of one, or otherwise they would sum to 2.0.

How should we allocate the probability between the two words? As we shift
probability mass from one word to the other, it would clearly affect the value of the
likelihood function. Imagine we start with an even allocation between *the* and *text*,
i.e., each would have a probability of 0.5. We can then imagine we could gradually
move some probability mass from *the* to *text* or vice versa. How would such a change
affect the likelihood function value?

$p(d|\Lambda) = p(\text{"text"}|\Lambda)p(\text{"the"}|\Lambda)$

$\quad = [0.5 * p(\text{"text"}|\theta_d) + 0.5 * 0.1] \times$

$\quad\quad [0.5 * p(\text{"the"}|\theta_d) + 0.5 * 0.9]$

Note that $p(\text{"text"}|\theta_d) + p(\text{"the"}|\theta_d) = 1$

If $x + y = $ constant, then xy reaches maximum when $x = y$

$0.5 * p(\text{"text"}|\theta_d) + 0.5 * 0.1 = 0.5 * p(\text{"the"}|\theta_d) + 0.5 * 0.9$

$\rightarrow p(\text{"text"}|\theta_d) = 0.9 \;>>\; p(\text{"the"}|\theta_d) = 0.1!$

$d = $ | **text** the

text ? θ_d
the ?

$p(\theta_d) = 0.5$

$p(\theta_B) = 0.5$

the 0.9 θ_B
text 0.1

Behavior 1: if $p(w_1|\theta_B) > p(w_2|\theta_B)$, then $p(w_1|\theta_d) < p(w_2|\theta_d)$
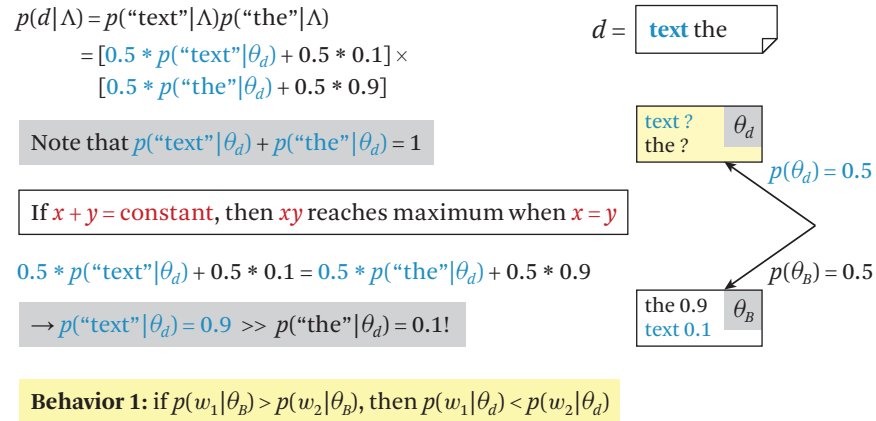
Figure 17.20 Behavior of a mixture model: competition of the two component models.

If you examine the formula carefully, you might intuitively feel that we want to set the probability of *text* to be somewhat larger than *the*, and this intuition can indeed be supported by a mathematical fact: when the sum of two variables is a constant, their product would reach the maximum when the two variables have the same value.

In our case, the sum of the two terms in the product is

$$0.5 \cdot p(\text{text} \mid \theta_d) + 0.5 \cdot 0.1 + 0.5 \cdot p(\text{the} \mid \theta_d) + 0.5 \cdot 0.9 = 1.0,$$

so their product reaches maximum when

$$0.5 \cdot p(\text{text} \mid \theta_d) + 0.5 \cdot 0.1 = 0.5 \cdot p(\text{the} \mid \theta_d) + 0.5 \cdot 0.9.$$

Plugging in the constraint $p(\text{text} \mid \theta_d) + p(\text{the} \mid \theta_d) = 1$, we can easily obtain the solution $p(\text{text} \mid \theta_d) = 0.9$ and $p(\text{the} \mid \theta_d) = 0.1$.

Therefore, the probability of *text* is indeed much larger than the probability of *the*, effectively factoring out this common word. Note that this is not the case when we have just one distribution where *the* has a much higher probability than *text*. The effect of reducing the estimated probability of *the* is clearly due to the use of the background model, which assigned very high probability to *the* and low probability to *text*.

Looking into the process of reaching this solution, we see that the reason why *text* has a higher probability than *the* is because its corresponding probability by the background model $p(\text{text} \mid \theta_B)$ is smaller than that of *the*; had the background model given *the* a smaller probability than *text*, our solution would give *the* a

higher probability than *text* in order to ensure that the overall probability given by the two models working together is the same for *text* and *the*. Thus, the ML estimate tends to give a word a higher probability if the background model gives it a smaller probability, or more generally, if one distribution has given word w_1 a higher probability than w_2, then the other distribution would give word w_2 a higher probability than word w_1 so that the combined probability of w_1 given by the two distributions working together would be the same as that of w_2. In other words, the two distributions tend to give high probabilities to different words as if they try to avoid giving the high probability to the same word.

In such a two-component mixture model, we see that the two distributions will be collaborating to maximize the probability of the observed data, but they are also competing on the words in the sense that they would tend to "bet" high probabilities on different words to gain advantages in this competition. In order to make their combined probability equal (so as to maximize the product in the likelihood function), the probability assigned by θ_d must be higher for a word that has a smaller probability given by the background model θ_B.

The general behavior we have observed here about a mixture model is that if one distribution assigns a higher probability to one word than another, the other distribution would tend to do the opposite; it would discourage other distributions to do the same. This also means that by using a background model that is fixed to assigning high probabilities to common (stop) words, we can indeed encourage the unknown topical word distribution to assign smaller probabilities for such common words so as to put more probability mass on those content words that cannot be explained well by the background model.

Let's look at another behavior of the mixture model in Figure 17.21 by examining the response of the estimated probabilities to the data frequencies. In Figure 17.21, we have shown a scenario where we've added more words to the document, specifically, more *the*'s to the document. What would happen to the estimated $p(w \mid \theta)$ if we keep adding more and more *the*'s to the document?

As we add more words to the document, we would need to multiply the likelihood function by additional terms to account for the additional occurrences. In this case, since all the additional terms are *the*, we simply need to multiply by the term representing the probability of *the*. This obviously changes the likelihood function, and thus also the solution of the ML estimation. How exactly would the additional terms accounting for multiple occurrences of *the* change the ML estimate? The solution we derived earlier, $p(\text{text} \mid \theta_d) = 0.9$, is no longer optimal. How should we modify this solution to make it optimal for the new likelihood function?

$d = $ | **text** the

$$p(d|\Lambda) = [0.5 * p(\text{``text''}|\theta_d) + 0.5 * 0.1]$$
$$\times [0.5 * p(\text{``the''}|\theta_d) + 0.5 * 0.9]$$

$$\rightarrow p(\text{``text''}|\theta_d) = 0.9 \gg p(\text{``the''}|\theta_d) = 0.1!$$

$d' = $ | **text** the
the the the
the ... the

$$p(d'|\Lambda) = [0.5 * p(\text{``text''}|\theta_d) + 0.5 * 0.1]$$
$$\times [0.5 * p(\text{``the''}|\theta_d) + 0.5 * 0.9]$$
$$\times [0.5 * p(\text{``the''}|\theta_d) + 0.5 * 0.9]$$
$$\times [0.5 * p(\text{``the''}|\theta_d) + 0.5 * 0.9]$$
$$...$$
$$\times [0.5 * p(\text{``the''}|\theta_d) + 0.5 * 0.9]$$

What if we increase $p(\theta_B)$?

What is the optimal solution now? $p(\text{``text''}|\theta_d) > 0.1$? or $p(\text{``the''}|\theta_d) < 0.1$?

Behavior 2: high frequency words get higher $p(w|\theta_d)$
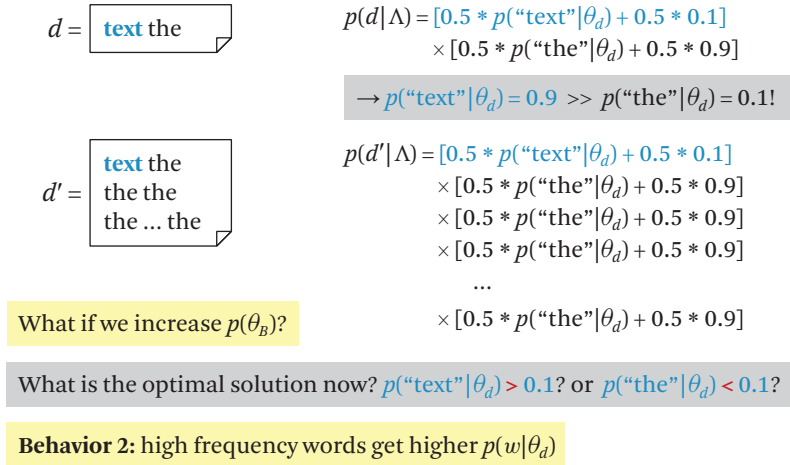
Figure 17.21   Behavior of a mixture model: maximizing data likelihood.

One way to address this question is to take away some probability mass from one word and add the probability mass to the other word, which would ensure that they sum to one. The question is, of course, which word to have a reduced probability and which word to have a larger probability. Should we make the probability of *the* larger or that of *text* larger?

If you look at the formula for a moment, you might notice that the new likelihood function (which is our objective function for optimization) is influenced more by *the* than *text*, so any reduction of probability of *the* would cause more decrease of the likelihood than the reduction of probability of *text*. Indeed, it would make sense to take away some probability from *text*, which only affects one term, and add the extra probability to *the*, which would benefit more terms in the likelihood function (since *the* occurred many times), thus generating an overall effect of increasing the value of the likelihood function. In other words, because *the* is repeated many times in the likelihood function, if we increase its probability a little bit, it will have substantial positive impact on the likelihood function, whereas a slight decrease of probability of *text* will have a relatively small negative impact because it occurred just once.

The analysis above reveals another behavior of the ML estimate of a mixture model: high frequency words in the observed text would tend to have high probabilities in all the distributions. Such a behavior should not be surprising at all because—after all—we are maximizing the likelihood of the data, so the more a

word occurs, the higher its overall probability should be. This is, in fact, a very general phenomenon of all the maximum likelihood estimators. In our special case, if a word occurs more frequently in the observed text data, it would also encourage the unknown distribution θ_d to assign a somewhat higher probability to this word.

We can also use this example to examine the impact of $p(\theta_B)$, the probability of choosing the background model. We've been so far assuming that each model is equally likely, i.e., $p(\theta_B) = 0.5$. But, you can again look at this likelihood function shown in Figure 17.21 and try to picture what would happen to the likelihood function if we increase the probability of choosing the background model.

It is not hard to notice that if $p(\theta_B) > 0.5$ is set to a very large value, then all the terms representing the probability of *the* would be even larger because the background has a very high probability for *the* (0.9), and the coefficient in front of 0.9, which was 0.5, would be even larger.

The consequence is that it is now less important for θ_d to increase the probability mass for *the* even when we add more and more occurrences of *the* to the document. This is because the overall probability of *the* is already very large (due to the large $p(\theta_B)$ and large $p(\text{the} \mid \theta_B)$), and the impact of increasing $p(\text{the} \mid \theta_d)$ is regulated by the coefficient $p(\theta_d)$ which would be small if we make $p(\theta_B)$ very large. It would be more beneficial for θ_d to ensure $p(\text{text} \mid \theta_d)$ to be high since *text* does not get any help from the background model, and it must rely on θ_d to assign a high probability. While high frequency words tend to get higher probabilities in the estimated $p(w \mid \theta_d)$, the degree of increase of probability due to the increased counts of a word observed in the document is regularized by $p(\theta_d)$ (or equivalently $p(\theta_B)$). The smaller $p(\theta_d)$ is, the less important for θ_d to respond to the increase of counts of a word in the data. In general, the more likely a component is being chosen in a mixture model, the more important it is for the component model to assign higher probability values to these frequent words.

To summarize, we discussed the mixture model, the estimation problem of the mixture model, and some general behaviors of the maximum likelihood estimator. First, every component model attempts to assign high probabilities to high frequent words in the data so as to collaboratively maximize the likelihood. Second, different component models tend to bet high probabilities on different words in order to avoid the "competition," or waste of probability. This would allow them to collaborate more efficiently to maximize the likelihood. Third, the probability of choosing each component regulates the collaboration and the competition between component models. It would allow some component models to respond more to the change of frequency of a word in the data. We also discussed the special case of fixing one component to a background word distribution, which can be estimated based on a large collection of English documents using the simplest single

unigram language model to model the data. The behaviors of the ML estimate of such a mixture model ensure that the use of a fixed background model in such a specialized mixture model can effectively factor out common words such as *the* in the other topic word distribution, making the discovered topic more discriminative.

We may view our specialized mixture model as one where we have imposed a very strong prior on the model parameter and we use Bayesian parameter estimation. Our prior is on one of the two unigram language models and it requires that this particular unigram LM must be exactly the same as a pre-defined background language model. In general, Bayesian estimation would seek for a compromise between our prior and the data likelihood, but in this case, we can assume that our prior is infinitely strong, and thus there is essentially no compromise, holding one component model as constant (the same as the provided background model). It is useful to point out that this mixture model is precisely the mixture model for feedback in information retrieval that we introduced earlier in the book.

17.3.5 Expectation-Maximization

The discussion of the behaviors of the ML estimate of the mixture model provides an intuition about why we can use a mixture model to mine one topic from a document with common words factored out through the use of a background model. In this section, we further discuss how we can compute such an ML estimate. Unlike the simplest unigram language model, whose ML estimate has an analytical solution, there is no analytical solution to the ML estimation problem for the two-component mixture model even though we have exactly the same number of parameters to estimate as a single unigram language model after we fix the background model and the choice probability of the component models (i.e., $p(\theta_d)$). We must use a numerical optimization algorithm to compute the ML estimate.

In this section, we introduce a specific algorithm for computing the ML estimate of the two-component mixture model, called the Expectation-Maximization (EM) algorithm. EM is a family of useful algorithms for computing the maximum likelihood estimate of mixture models in general.

Recall that we have assumed both $p(w \mid \theta_B)$ and $p(\theta_B)$ are already given, so the only "free" parameters in our model are $p(w \mid \theta_d)$ for all the words subject to the constraint that they sum to one. This is illustrated in Figure 17.22. Intuitively, when we compute the ML estimate, we would be exploring the space of all possible values for the word distribution θ_d until we find a set of values that would maximize the probability of the observed documents.

According to our mixture model, we can imagine that the words in the text data can be partitioned into two groups. One group will be explained (generated) by

If we know which word is from which distribution ...
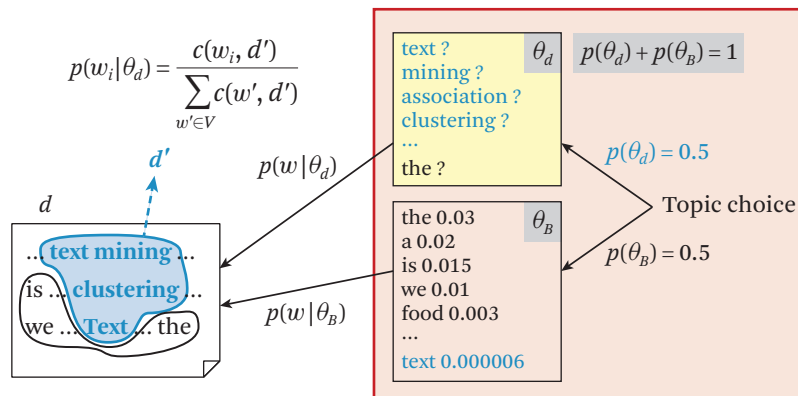
Figure 17.22 Estimation of a topic when each word is known to be from a particular distribution.

the background model. The other group will be explained by the unknown topical model (the topic word distribution). The challenge in computing the ML estimate is that we do not know this partition due to the possibility of generating a word using either of the two distributions in the mixture model.

If we actually know which word is from which distribution, computation of the ML estimate would be trivial as illustrated in Figure 17.22, where d' is used to denote the pseudo document that is composed of all the words in document d that are known to be generated by θ_d, and the ML estimate of θ_d is seen to be simply the normalized word frequency in this pseudo document d'. That is, we can simply pool together all the words generated from θ_d, compute the count of each word, and then normalize the count by the total counts of all the words in such a pseudo document. In such a case, our mixture model is really just two independent unigram language models, which can thus be estimated separately based on the data points generated by each of them.

Unfortunately, the real situation is such that we don't really know which word is from which distribution. The main idea of the EM algorithm is to guess (infer) which word is from which distribution based on a tentative estimate of parameters, and then use the inferred partitioning of words to improve the estimate of parameters, which, in turn, enables improved inference of the partitioning, leading to an iterative hill-climbing algorithm to improve the estimate of the parameters until hitting a local maximum. In each iteration, it would invoke an E-step followed by an M-step, which will be explained in more detail.

For now, let's assume we have a tentative estimate of all the parameters. How can we infer which of the two distributions a word has been generated from? Consider a specific word such as *text*. Is it more likely from θ_d or θ_B? To answer this question, we compute the conditional probability $p(\theta_d \mid \text{text})$. The value of $p(\theta_d \mid \text{text})$ would depend on two factors.

- How often is θ_d (as opposed to θ_B) used to generate a word in general? This probability is given by $p(\theta_d)$. If $p(\theta_d)$ is high, then we'd expect $p(\theta_d \mid \text{text})$ to be high.

- If θ_d is indeed chosen to generate a word, how likely would we observe *text*? This probability is given by $p(w \mid \theta_d)$. If $p(\text{text} \mid \theta_d)$ is high, then we'd also expect $p(\theta_d \mid \text{text})$ to be high.

Our intuition can be rigorously captured by using Bayes' rule to infer $p(\theta_d \mid \text{text})$, where we essentially compare the product $p(\theta_d) p(\text{text} \mid \theta_d)$ with the product $p(\theta_B) p(\text{text} \mid \theta_B)$ to see whether *text* is more likely generated from θ_d or from θ_B. This is illustrated in Figure 17.23.

The Bayesian inference involved here is a typical one where we have some prior about how likely each of these two distributions is used to generate any word (i.e., $p(\theta_d)$ and $p(\theta_B)$). These are *prior* because they encode our belief about which distribution *before* we even observe the word *text*; a prior that has very high $p(\theta_d)$ would encourage us to lean toward guessing θ_d for any word. Such a prior is then

Given all the parameters, infer the distribution a word is from ...
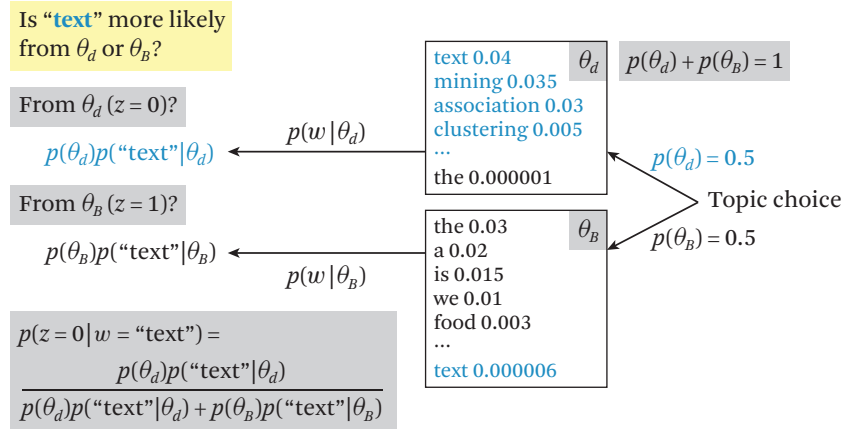
Figure 17.23   Inference of which distribution a word is from.

updated by incorporating the data likelihood $p(\text{text} \mid \theta_d)$ and $p(\text{text} \mid \theta_B)$ so that we would favor a distribution that gives *text* a higher probability.

In the example shown in Figure 17.23, our prior says that each of the two models is equally likely; thus, it is a non-informative prior (one with no bias). As a result, our inference of which distribution has been used to generate a word would solely be based on $p(w \mid \theta_d)$ and $p(w \mid \theta_B)$. Since $p(\text{text} \mid \theta_d)$ is much larger than $p(\text{text} \mid \theta_B)$, we can conclude that θ_d is much more likely the distribution that has been used to generate *text*. In general, our prior may be biased toward a particular distribution. Indeed, a heavily biased prior can even dominate over the data likelihood to essentially dictate the decision. For example, imagine our prior says $p(\theta_B) = 0.99999999$, then our inference result would say that *text* is more likely generated by θ_B than by θ_d even though $p(\text{text} \mid \theta_d)$ is much higher than $p(\text{text} \mid \theta_B)$, due to the very strong prior. Bayes' Rule provides us a principled way of combining the prior and data likelihood.

In Figure 17.23, we introduced a binary latent variable z here to denote whether the word is from the background or the topic. When z is 0, it means it's from the topic, θ_d; when it's 1, it means it's from the background, θ_B. The posterior probability $p(z = 0 \mid w = \text{text})$ formally captures our guess about which distribution has been used to generate the word *text*, and it is seen to be proportional to the product of the prior $p(\theta_d)$ and the likelihood $p(\text{text} \mid \theta_d)$, which is intuitively very meaningful since in order to generate *text* from θ_d, we must first choose θ_d (as opposed to θ_B), which is captured by $p(\theta_d)$, and then obtain word *text* from the selected θ_d, which is captured by $p(w \mid \theta_d)$.

Understanding how to make such a Bayesian inference of which distribution has been used to generate a word based on a set of tentative parameter values is very crucial for understanding the EM algorithm. This is essentially the E-step of the EM algorithm where we use Bayes' rule to partition data and allocate all the data points among all the component models in the mixture model.

Note that the E-step essentially helped us figure out which words have been generated from θ_d (and equivalently, which words have been generated from θ_B) except that it does not completely allocate a word to θ_d (or θ_B), but splits a word in between the two distributions. That is, $p(z = 0 \mid \text{text})$ tells us what percent of the count of *text* should be allocated to θ_d, and thus contribute to the estimate of θ_d. This way, we will be able to collect all the counts allocated to θ_d, and renormalize them to obtain a potentially improved estimate of $p(w \mid \theta_d)$, which is our goal. This step of re-estimating parameters based on the results from the E-step is called the M-step.

With the E-step and M-step as the basis, the EM algorithm works as follows. First, we initialize all the (unknown) parameters values randomly. This allows us to have a complete specification of the mixture model, which further enables us to use Bayes' rule to infer which distribution is more likely to generate each word. This prediction (i.e., E-step) essentially helps us (probabilistically) separate words generated by the two distributions. Finally, we will collect all the probabilistically allocated counts of words belonging to our topic word distribution and normalize them into probabilities, which serve as an improved estimate of the parameters. The process can then be repeated to gradually improve the parameter estimate until the likelihood function reaches a local maximum. The EM algorithm can guarantee reaching such a local maximum, but it cannot guarantee reaching a global maximum when there are multiple local maxima. Due to this, we usually repeat the algorithm multiple times with different initializations in practice, using the run that gives the highest likelihood value to obtain the estimated parameter values.

The EM algorithm is illustrated in Figure 17.24 where we see that a binary hidden variable z has been introduced to indicate whether a word has been generated from the background model ($z = 1$) or the topic model ($z = 0$). For example, the illustration shows that *the* is generated from background, and thus the z value is 1.0, while *text* is from the topic, so its z value is 0. Note that we simply assumed (imagined) the existence of such a binary latent variable associated with each word

The Expectation-Maximization (EM) Algorithm

Hidden variable:
$z \in [0, 1]$

Initialize $p(w|\theta_d)$ with random values.
Then iteratively improve it using E-step and M-step.
Stop when likelihood doesn't change.

	z
the	1
paper	1
presents	1
a	1
text	0
mining	0
algorithm	0
for	1
clustering	0
...	...

$$p^{(n)}(z = 0|w) = \frac{p(\theta_d)p^{(n)}(w|\theta_d)}{p(\theta_d)p^{(n)}(w|\theta_d) + p(\theta_B)p(w|\theta_B)}$$

E-step

How likely w is from θ_d

$$p^{(n+1)}(w|\theta_d) = \frac{c(w, d)p^{(n)}(z = 0|w)}{\sum_{w' \in V} c(w', d)p^{(n)}(z = 0|w')}$$

M-step
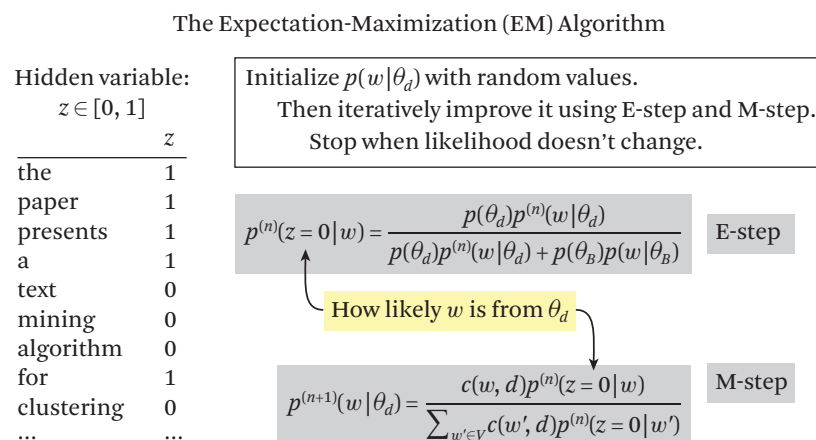
Figure 17.24   The EM algorithm.

token, but we don't really observe these z values. This is why we referred to such a variable as a hidden variable.

A main idea of EM is to leverage such hidden variables to simplify the computation of the ML estimate since knowing the values of these hidden variables makes the ML estimate trivial to compute; we can pool together all the words whose z values are 0 and normalize their counts. Knowing z values can potentially help simplify the task of computing the ML estimate, and EM exploits this fact by alternating the E-step and M-step in each iteration so as to improve the parameter estimate in a hill-climbing manner.

Specifically, the E-step is to infer the value of z for all the words, while the M-step is to use the inferred z values to split word counts between the two distributions, and use the allocated counts for θ_d to improve its estimation, leading to a new generation of improved parameter values, which can then be used to perform a new iteration of E-step and M-step to further improve the parameter estimation.

In the M-step, we adjust the count $c(w, d)$ based on $p(z = 0 \mid w)$ (i.e., probability that the word w is indeed from θ_d) so as to obtain a discounted count $c(w, d) p(z = 0 \mid w)$ which can be interpreted as the expected count of the event that word w is generated from θ_d. Similarly, θ_B has its own share of the count, which is $c(w, d) p(z = 1 \mid w) = c(w, d)[1 - p(z = 0 \mid w)]$, and we have

$$c(w, d) p(z = 0 \mid w) + c(w, d) p(z = 1 \mid w) = c(w, d), \qquad (17.5)$$

showing that all the counts of word w have been split between the two distributions.

Thus, the M-step is simply to normalize these discounted counts for all the words to obtain a probability distribution over all the words, which can then be regarded as our improved estimate of $p(w \mid \theta_d)$. Note that in the M-step, if $p(z = 0 \mid w) = 1$ for all words, we would simply compute the simple single unigram language model based on all the observed words (which makes sense since the E-step would have told us that there is no chance that any word has been generated from the background).

In Figure 17.25, we further illustrate in detail what happens in each iteration of the EM algorithm. First, note that we used superscripts in the formulas of the E-step and M-step to indicate the generation of parameters. Thus, the M-step is seen to use the n-th generation of parameters together with the newly inferred z values to obtain a new $(n + 1)$th generation of parameters (i.e., $p^{n+1}(w \mid \theta_d)$). Second, we assume the two component models (θ_d and θ_B) have equal probabilities; we also assume that the background model word distribution is known (fixed as shown in the third column of the table).

The computation of EM starts with preparation of relevant word counts. Here we assume that we have just four words, and their counts in the observed text data

$$\text{E-step:} \quad p^{(n)}(z=0 \mid w) = \frac{p(\theta_d)p^{(n)}(w \mid \theta_d)}{p(\theta_d)p^{(n)}(w \mid \theta_d) + p(\theta_B)p(w \mid \theta_B)}$$

$$\text{M-step:} \quad p^{(n+1)}(w \mid \theta_d) = \frac{c(w,d)p^{(n)}(z=0 \mid w)}{\sum_{w' \in V} c(w',d)p^{(n)}(z=0 \mid w')}$$

Assume $p(\theta_d) = p(\theta_B) = 0.5$ and $p(w \mid \theta_B)$ is known.

Word	No.	$p(w \mid \theta_B)$	Iteration 1		Iteration 2		Iteration 3	
			$P(w \mid \theta)$	$p(z=0 \mid w)$	$P(w \mid \theta)$	$P(z=0 \mid w)$	$P(w \mid \theta)$	$P(z=0 \mid w)$
The	4	0.5	0.25	0.33	0.20	0.29	0.18	0.26
Paper	2	0.3	0.25	0.45	0.14	0.32	0.10	0.25
Text	4	0.1	0.25	0.71	0.44	0.81	0.50	0.93
Mining	2	0.1	0.25	0.71	0.22	0.69	0.22	0.69
Log-Likelihood				-16.96		-16.13		-16.02

Likelihood increasing $\longrightarrow$

Figure 17.25   **An example of EM computation.**

are shown in the second column of the table. The EM algorithm then initializes all the parameters to be estimated. In our case, we set all the probabilities to 0.25 in the fourth column of the table.

In the first iteration of the EM algorithm, we will apply the E-step to infer which of the two distributions has been used to generate each word, i.e., to compute $p(z=0 \mid w)$ and $p(z=1 \mid w)$. We only showed $p(z=0 \mid w)$, which is needed in our M-step ($p(z=1 \mid w) = 1 - p(z=0 \mid w)$). Clearly, $p(z=0 \mid w)$ has different values for different words, and this is because these words have different probabilities in the background model and the initialized θ_d. Thus, even though the two distributions are equally likely (by our prior) and our initial values for $p(w \mid \theta_d)$ form a uniform distribution, the inferred $p(z=0 \mid w)$ would tend to give words with smaller probabilities if $p(w \mid \theta_B)$ give them a higher probability. For example, $p(z=0 \mid \text{text}) > p(z=0 \mid \text{the})$.

Once we have the probabilities of all these z values, we can perform the M-step, where these probabilities would be used to adjust the counts of the corresponding words. For example, the count of *the* is 4, but since $p(z=0 \mid \text{the}) = 0.33$, we would obtain a discounted count of *the*, 4×0.33, when estimating $p(\text{the} \mid \theta_d)$ in the M-step. Similarly, the adjusted count for *text* would be 4×0.71. After the M-step, $p(\text{text} \mid \theta_d)$ would be much higher than $p(\text{the} \mid \theta_d)$ as shown in the table (shown in the first

column under Iteration 2). Those words that are believed to have come from the topic word distribution θ_d according to the E-step would have a higher probability.

This new generation of parameters would allow us to further adjust the inferred latent variable or hidden variable values, leading to a new generation of probabilities for the z values, which can be fed into another M-step to generate yet another generation of potentially improved estimate of θ_d.

In the last row of the table, we show the log-likelihood after each iteration. Since each iteration would lead to a different generation of parameter estimates, it would also give a different value for the log-likelihood function. These log-likelihood values are all negative because the probability is between 0 and 1, which becomes a negative value after the logarithm transformation. We see that after each iteration, the log-likelihood value is increasing, showing that the EM algorithm is iteratively improving the estimated parameter values in a hill-climbing manner. We will provide an intuitive explanation of why it converges to a local maximum later.

For now, it is worth pointing out that while the main goal of our EM algorithm is to obtain a more discriminative word distribution to represent the topic that we hope to discover, i.e., $p(w \mid \theta_d)$, the inferred $p(z = 0 \mid w)$ after convergence is also meaningful and may sometimes be a useful byproduct. Specifically, these are the probabilities that a word is believed to have come from the topic distribution, and we can add them up to obtain an estimate of to what extent the document has covered background vs. content, or to what extent the content of the document deviates from a "typical" background document. This would give us a single numerical score for each document, so we can then use the score to compare different documents or different subsets of documents (e.g., those associated with different authors or from different sources). Thus, our simple two-component mixture model can not only help us discover a single topic from the document, but also provide a useful measure of "typicality" of a document which may be useful in some applications.

Next, we provide some intuitive explanation why the EM algorithm will converge to a local maximum in Figure 17.26. Here we show the parameter θ_d on the X-axis, and the Y-axis denotes the likelihood function value. This is an over-simplification since θ_d is an M-dimensional vector, but the one-dimensional view makes it much easier to understand the EM algorithm. We see that, in general, the original likelihood function (as a function of θ_d) may have multiple local maxima. The goal of computing the ML estimate is to find the global maximum, i.e., the θ_d value that makes the likelihood function reach it global maximum.

The EM algorithm is a hill-climbing algorithm. It starts with an initial (random) guess of the optimal parameter value, and then iteratively improves it. The picture shows the scenario of going from iteration n to iteration $n + 1$. At iteration n, the

EM as Hill-Climbing → Converse to Local Maximum

Likelihood $p(d|\theta)$

Original likelihood

E-step = computing the lower bound

Lower bound of likelihood function

Next guess $p^{(n+1)}(w|\theta_d)$

Current guess $p^{(n)}(w|\theta_d)$

M-step = maximizing the lower bound

θ
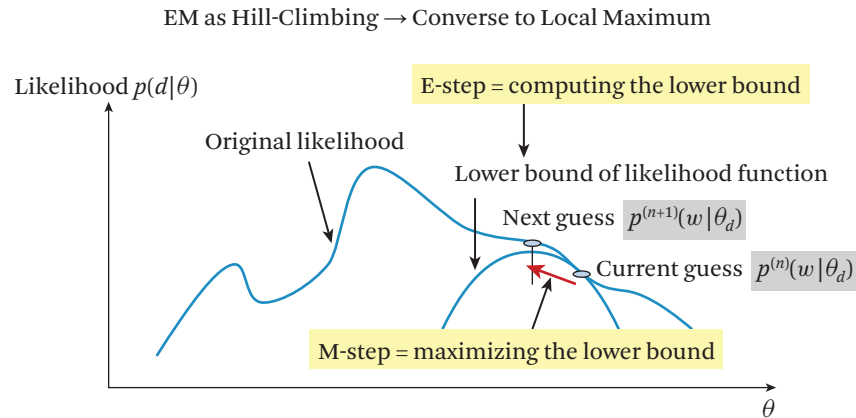
Figure 17.26   **EM as hill-climbing for optimizing likelihood.**

current guess of the parameter value is $p^{(n)}(w \mid \theta_d)$, and it is seen to be non-optimal in the picture. In the E-step, the EM algorithm (conceptually) computes an auxiliary function which lower bounds the original likelihood function. Lower bounding means that for any given value of θ_d, the value of this auxiliary function would be no larger than that of the original likelihood function. In the M-step, the EM algorithm finds an optimal parameter value that would maximize the auxiliary function and treat this parameter value as our improved estimate, $p^{(n+1)}(w \mid \theta_d)$. Since the auxiliary function is a lower bound of the original likelihood function, maximizing the auxiliary function ensures the new parameter to also have a higher value according to the original likelihood function unless it has already reached a local maximum, in which case, the optimal value maximizing the auxiliary function is also a local maximum of the original likelihood function. This explains why the EM algorithm is guaranteed to converge to a local maximum. You might wonder why we don't work on finding an improved parameter value directly on the original likelihood function. Indeed, it is possible to do that, but in the EM algorithm, the auxiliary function is usually much easier to optimize than the original likelihood function, so in this sense, it reduces the problem into a somewhat simpler one. Although the auxiliary function is generally easier to optimize, it does not always have an analytical solution, which means that the maximization of the auxiliary function may itself require another iterative process, which would be embedded in the overall iterative process of the EM algorithm.

In our case of the simple mixture model, we did not explicitly compute this auxiliary function in the E-step because the auxiliary function is very simple and

as a result, our M-step has an analytical solution, thus we were able to bypass the explicit computation of this auxiliary function and go directly to find a re-estimate of the parameters. Thus in the E-step, we only computed a key component in the auxiliary function, which is the probability that a word has been generated from each of the two distributions, and our M-step directly gives us an analytical solution to the problem of optimizing the auxiliary function, and the solution directly uses the values obtained from the E-step.

The EM algorithm has many applications. For example, in general, parameter estimation of all mixture models can be done by using the EM algorithm. The hidden variables introduced in a mixture model often indicate which component model has been used to generate a data point. Thus, once we know the values of these hidden variables, we would be able to partition data and identify the data points that are likely generated from any particular distribution, thus facilitating estimation of component model parameters. In general, when we apply the EM algorithm, we would augment our data with supplementary unobserved hidden variables to simplify the estimation problem. The EM algorithm would then work as follows. First, it would randomly initialize all the parameters to be estimated. Second, in the E-step, it would attempt to infer the values of the hidden variables based on the current generation of parameters, and obtain a probability distribution of hidden variables over all possible values of these hidden variables. Intuitively, this is to take a good guess of the values of the hidden variables. Third, in the M-step, it would use the inferred hidden variable values to compute an improved estimate of the parameter values. This process is repeated until convergence to a local maximum of the likelihood function. Note that although the likelihood function is guaranteed to converge to a local maximum, there is no guarantee that the parameters to be estimated always have a stable convergence to a particular set of values. That is, the parameters may oscillate even though the likelihood is increasing. Only if some conditions are satisfied would the parameters be guaranteed to converge (see Wu 1983).

17.4   Probabilistic Latent Semantic Analysis

In this section, we introduce probabilistic latent semantic analysis (PLSA), the most basic topic model, with many applications. In short, PLSA is simply a generalization of the two-component mixture model that we discussed earlier in this chapter to discover more than one topic from text data. Thus, if you have understood the two-component mixture model, it would be straightforward to understand how PLSA works.

Topic θ_1	government 0.3 response 0.2 ...	**Blog article about "Hurricane Katrina"**
Topic θ_2 ...	city 0.2 new 0.1 orleans 0.05 ...	[**Criticisms** of **government response** to the **hurricane primarily consisted** of **criticism** of its **response** to the **approach** of the **storm** and its **aftermath, specifically** in the **delayed response**] to the [**flooding of New Orleans. ... 80%** of the **1.3 million residents** of the **greater New Orleans metropolitan area evacuated**] ... [**Over seventy countries pledged monetary donations** or other **assistance.**] ...
Topic θ_k	donate 0.1 relief 0.05 help 0.02 ...	
Background θ_B	the 0.04 a 0.03 ...	Many applications are possible if we can "decode" the topics in text ...

Figure 17.27   **A document as a sample of words from mixed topics.**

As we mentioned earlier, the general task of topic analysis is to mine multiple topics from text documents and compute the coverage of each topic in each document. PLSA is precisely designed to perform this task. As in all topic models, we make two key assumptions. First, we assume that a topic can be represented as a word distribution (or more generally a term distribution). Second, we assume that a text document is a sample of words drawn from a probabilistic model. We illustrate these two assumptions in Figure 17.27, where we see a blog article about Hurricane Katrina and some imagined topics, each represented by a word distribution, including, e.g., a topic on government response (θ_1), a topic on the flood of the city of New Orleans (θ_2), a topic on donation (θ_k), and a background topic θ_B. The article is seen to contain words from all these distributions. Specifically, we see there is a criticism of government response at the beginning of this excerpt, which is followed by discussion of flooding of the city, and then a sentence about donation. We also see background words mixed in throughout the article.

The main goal of topic analysis is to try to decode these topics behind the text (by segmenting them), and figure out which words are from which distribution so that we can obtain both characterizations of all the topics in the text data and the coverage of topics in each document. Once we can do these, they can be directly used in many applications such as summarization, segmentation, and clustering.

Input: C, k, V Output: $\{\theta_1, ..., \theta_k\}, \{\pi_{i1}, ..., \pi_{ik}\}$
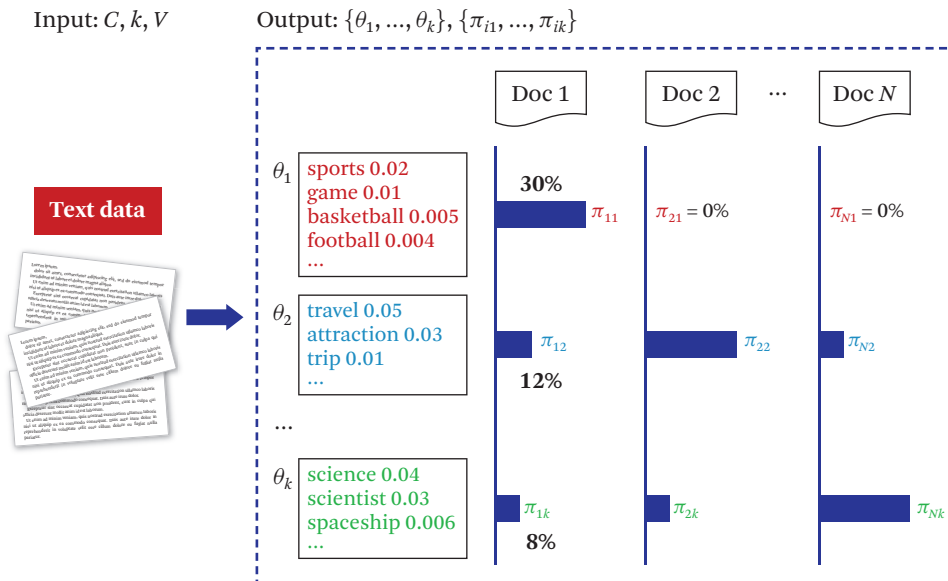
Figure 17.28   **Task of mining multiple topics in text.**

The formal definition of mining multiple topics from text is illustrated in Figure 17.28. The input is a collection of text data, the number of topics, and a vocabulary set. The output is of two types. One is topic characterization where each topic is represented by θ_i, which is a word distribution. The other is the topic coverage for each document π_{ij} which refers to the probability that document d_i covers topic θ_j.

Such a problem can be solved by using PLSA, a generalization of the simple two-component mixture model to more than two components. Such a more generative model is illustrated in Figure 17.29, where we also retain the background model used in the two-component mixture model (which, if you recall, was designed to discover just one topic). Different from the simple mixture model discussed earlier, the model here includes k component models, each of which represents a distinct topic and can be used to generate a word in the observed text data. Adding the background model θ_B, we thus have a total of $k + 1$ component unigram language models in PLSA.[2]

---

2. The original PLSA [Hofmann 1999] did not include a background language model, thus it gives common words high probabilities in the learned topics if such common words are not removed in the preprocessing stage.

Generating Text with Multiple Topics: $p(w) = ?$
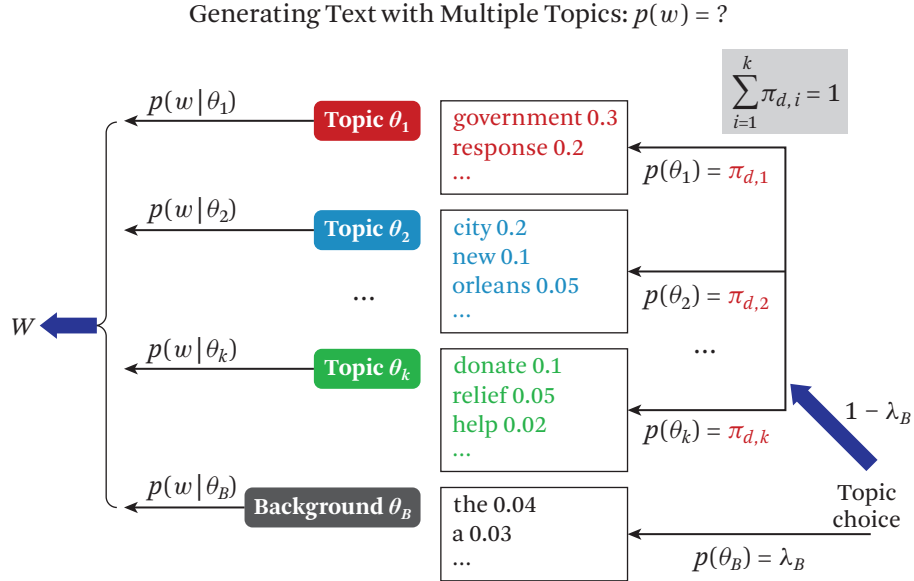
Figure 17.29 **Generating words from a mixture of multiple topics.**

As in the case of the simple mixture model, the process of generating a word still consists of two steps. The first is to choose a component model to use; this decision is controlled by both a parameter λ_B (denoting the probability of choosing the background model) and a set of $\pi_{d,i}$ (denoting the probability of choosing topic θ_i *if* we decided not to use the background model). If we do not use the background model, we must choose one from the k topics, which has the constraint $\sum_{i=1}^{k} \pi_{d,i} = 1$. Thus, the probability of choosing the background model is λ_B while the probability of choosing topic θ_i is $(1 - \lambda_B)\pi_{d,i}$.

Once we decide which component word distribution to use, the second step in the generation process is simply to draw a word from the selected distribution, exactly the same as in the simple mixture model.

As usual, once we design the generative model, the next step is to write down the likelihood function. We ask the question: what's the probability of observing a word from such a mixture model? As in the simple mixture model, this probability is a sum over all the different ways to generate the word; we have a total of $k + 1$ different component models, thus it is a sum of $k + 1$ terms, where each term captures the probability of observing the word from the corresponding component word distribution, which can be further written as the product of the probability

Probabilistic Latent Semantic Analysis (PLSA)

Percentage of
background
words (known)

Background
LM (known)

Coverage of topic θ_j in doc d

Probability of word w in topic θ_j

$$p_d(w) = \lambda_B p(w|\theta_B) + (1 - \lambda_B)\sum_{j=1}^{k} \pi_{d,j} p(w|\theta_j)$$

$$\log p(d) = \sum_{w \in V} c(w, d) \log \left[\lambda_B p(w|\theta_B) + (1 - \lambda_B)\sum_{j=1}^{k} \pi_{d,j} p(w|\theta_j)\right]$$

$$\log p(C|\Lambda) = \sum_{d \in C} \sum_{w \in V} c(w, d) \log \left[\lambda_B p(w|\theta_B) + (1 - \lambda_B)\sum_{j=1}^{k} \pi_{d,j} p(w|\theta_j)\right]$$

Unknown parameters: $\Lambda = (\{\pi_{d,j}\}, \{\theta_j\}), j = 1, ..., k$

Figure 17.30   **The likelihood function of PLSA.**

of selecting the particular component model and the probability of observing the particular word from the particular selected word distribution.

The likelihood function is as illustrated in Figure 17.30. Specifically, the probability of observing a word from the background distribution is $\lambda_B p(w|\theta_B)$, while the probability of observing a word from a topic θ_j is $(1 - \lambda_B)\pi_{d,j} p(w|\theta_j)$. The probability of observing the word regardless of which distribution is used, $p_d(w)$, is just a sum of all these cases.

Assuming that the words in a document are generated independently, it follows that the likelihood function for document d is the second equation in Figure 17.30, and that the likelihood function for the entire collection C is given by the third equation.

What are the parameters in PLSA? First, we see λ_B, which represents the percentage of background words that we believe exist in the text data (and that we would like to factor out). This parameter can be set empirically to control the desired discrimination of the discovered topic models. Second, we see the background language model $p(w|\theta_B)$, which we also assume is known. We can use any large collection of text, or use all the text that we have available in collection C to estimate $p(w|\theta_B)$ (e.g., assuming all the text data are generated from θ_B, we can use the ML estimate to set $p(w|\theta_B)$ to the normalized count of word w in the data). Third, we see $\pi_{d,j}$, which indicates the coverage of topic θ_j in document d. This parameter encodes the knowledge we hope to discover from text. Finally, we see the k word distributions,

Constrained Optimization: $\Lambda^* = \arg\max_\Lambda p(C \mid \Lambda)$

$$\forall j \in [1, k], \sum_{i=1}^{M} p(w_i \mid \theta_j) = 1 \qquad \forall d \in C, \sum_{j=1}^{k} \pi_{d,j} = 1$$

Figure 17.31 **ML estimate of PLSA.**

each representing a topic $p(w \mid \theta_j)$. This parameter also encodes the knowledge we would like to discover from the text data. Can you figure out how many unknown parameters are there in such a PLSA model? This would be a useful exercise to do, which helps us understand what exactly are the outputs that we would generate by using PLSA to analyze text data.

After we have obtained the likelihood function, the next question is how to perform parameter estimation. As usual, we can use the Maximum Likelihood estimator as shown in Figure 17.31, where we see that the problem is essentially a constrained optimization problem, as in the case of the simple mixture model, except that:

- we now have a collection of text articles instead of just one document;
- we have more parameters to estimate; and
- we have more constraint equations (which is a consequence of having more parameters).

Despite the third point, the kinds of constraints are essentially the same as before; namely, there are two. One ensures the topic coverage probabilities sum to one for each document over all the possible topics, and the other ensures that the probabilities of all the words in each topic sum to one.

As in the case of simple mixture models, we can also use the EM algorithm to compute the ML estimate for PLSA. In the E-step, we have to introduce more hidden variables because we have more topics. Our hidden variable z, which is a topic indicator for a word, now would take $k + 1$ values $\{1, 2, \ldots, k, B\}$, corresponding to the k topics and the extra background topic. The E-step uses Bayes' Rule to infer the probability of each value for z, as shown in Figure 17.32. A comparison between these equations as the E-step for the simple two-component mixture model would reveal immediately that the equations are essentially similar, only now we have more topics. Indeed, if we assume there is just one topic, $k = 1$, then we would recover the E-step equation of the simple mixture model with just one small difference: $p(z_{d,w} = j)$ is not quite the probability that the word is generated

EM Algorithm for PLSA: E-Step

Hidden variable (= topic indicator): $z_{d,w} \in \{B, 1, 2, ..., k\}$

Probability that w in doc d is generated from topic θ_j Use of Bayes Rule

$$p(z_{d,w} = j) = \frac{\pi_{d,j}^{(n)} p^{(n)}(w|\theta_j)}{\sum_{j'=1}^{k} \pi_{d,j'}^{(n)} p^{(n)}(w|\theta_{j'})}$$

$$p(z_{d,w} = B) = \frac{\lambda_B p(w|\theta_B)}{\lambda_B p(w|\theta_B) + (1 - \lambda_B)\sum_{j=1}^{k} \pi_{d,j}^{(n)} p^{(n)}(w|\theta_j)}$$

Probability that w in doc d is generated from background θ_B
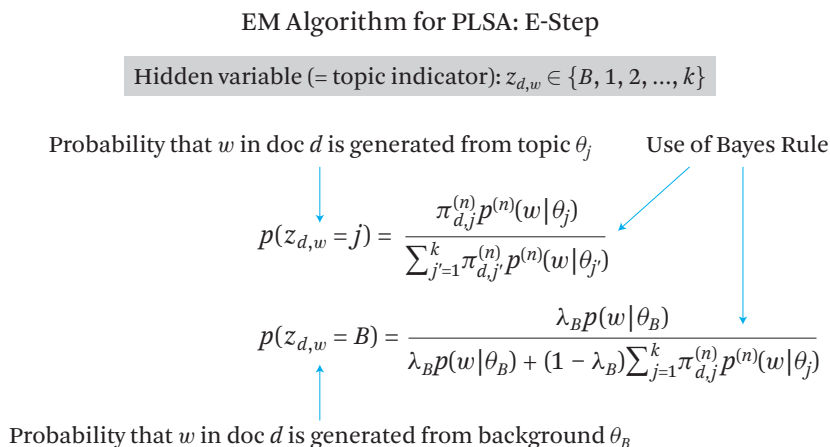
Figure 17.32   **E-Step of the EM Algorithm for estimating PLSA.**

from topic θ_j, but rather this probability conditioned on having not chosen the background model. In other words, the probability of generating a word using θ_j is $(1 - p(z_{d,w} = B))p(z_{d,w} = j)$. In the case of having just one topic other than the background model, we would have $p(z_{d,w} = j) = 1$ only for θ_j.

Note that we use document d here to index the word w. In our model, whether w has been generated from a particular topic actually depends on the document! Indeed, the parameter $\pi_{d,j}$ is tied to each document, and thus each document can have a potentially different topic coverage distribution. Such an assumption is reasonable as different documents generally have a different emphasis on specific topics. This means that in the E-step, the inferred probability of topics for the same word can be potentially very different for different documents since different documents generally have different $\pi_{d,j}$ values.

The M-step is also similar to that in the simple mixture model. We show the equations in Figure 17.33. We see that a key component in the two equations, for re-estimating π and $p(w \mid \theta)$ respectively, is $c(w, d)(1 - p(z_{d,w} = B))p(z_{d,w} = j)$, which can be interpreted as the allocated counts of w to topic θ_j. Intuitively, we use the inferred distribution of z values from the E-step to split the counts of w among all the distributions. The amount of split counts of w that θ_j can get is determined based on the inferred likelihood that w is generated by topic θ_j.

Once we have such a split count of each word for each distribution, we can easily pool together these split counts to re-estimate both π and $p(w \mid \theta)$, as shown in Figure 17.33. To re-estimate $\pi_{d,j}$, the probability that document d covers topic θ_j,

EM Algorithm for PLSA: M-Step

Hidden variable (= topic indicator): $z_{d,w} \in \{B, 1, 2, ..., k\}$

ML estimate based on "allocated" word counts to topic θ_j

Re-estimated probability of doc d covering topic θ_j

$$\pi_{d,j}^{(n+1)} = \frac{\sum_{w \in V} c(w,d)(1 - p(z_{d,w} = B))p(z_{d,w} = j)}{\sum_{j'} \sum_{w \in V} c(w,d)(1 - p(z_{d,w} = B))p(z_{d,w} = j')}$$

$$p^{(n+1)}(w|\theta_j) = \frac{\sum_{d \in C} c(w,d)(1 - p(z_{d,w} = B))p(z_{d,w} = j)}{\sum_{w' \in V} \sum_{d \in C} c(w',d)(1 - p(z_{d,w'} = B))p(z_{d,w'} = j)}$$

Re-estimated probability of word w for topic θ_j
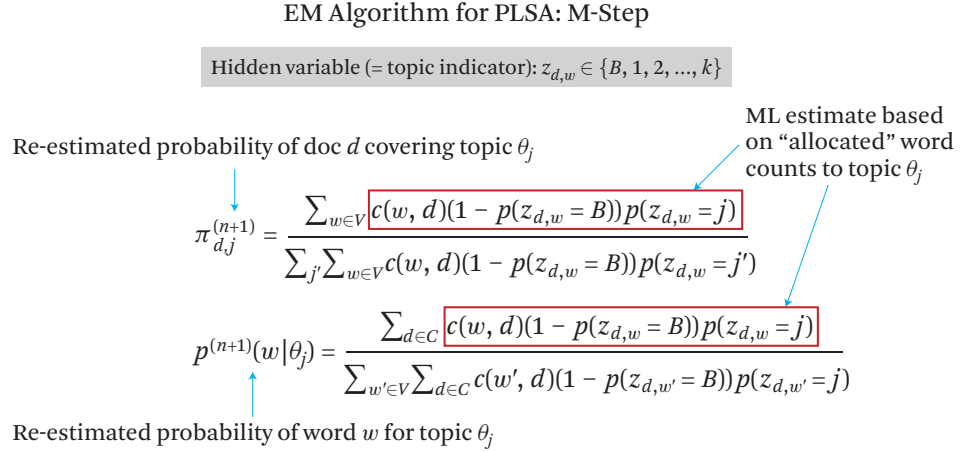
Figure 17.33 **M-Step of the EM Algorithm for estimating PLSA.**

we would simply collect all the split counts of words in document d that belong to each θ_j, and then normalize these counts among all the k topics. Similarly, to re-estimate $p(w | \theta_j)$, we would collect the split counts of a word toward θ_j from all the documents in the collection, and then normalize these counts among all the words. Note that the normalizers are very different in these two cases, which are directly related to the constraints we have on these parameters. In the case of re-estimation of π, the constraint is that the π values must sum to one for each document, thus our normalizer has been chosen to ensure that the re-estimated values of π indeed sum to one for each document. The same is true for the re-estimation of $p(w | \theta)$, where our normalizer allows us to obtain a word distribution for each topic.

What we observed here is actually generally true when using the EM algorithm. That is, the distribution of the hidden variables computed in the E-step can be used to compute the expected counts of an event, which can then be aggregated and normalized appropriately to obtain a re-estimate of the parameters. In the implementation of the EM algorithm, we can thus just keep the counts of various events and then normalize them appropriately to obtain re-estimates for various parameters.

In Figure 17.34, we show the computation of the EM algorithm for PLSA in more detail. We first initialize all the unknown parameters randomly, including the coverage distribution $\pi_{d,j}$ for each document d, and the word distribution for each topic $p(w | \theta_j)$. After the initialization step, the EM algorithm would go through

- Initialize all unknown parameters randomly
- Repeat until likelihood converges

– E-step $p(z_{d,w} = j) \propto \pi_{d,j}^{(n)} p^{(n)}(w|\theta_j)$ $\sum_{j=1}^{k} p(z_{d,w} = j) = 1$

$p(z_{d,w} = B) \propto \lambda_B p(w|\theta_B)$ ⟵ What's the normalizer for this one?

– M-step $\pi_{d,j}^{(n+1)} \propto \sum_{w \in V} c(w, d)(1 - p(z_{d,w} = B))p(z_{d,w} = j)$ $\forall d \in C, \sum_{j=1}^{k} \pi_{d,j} = 1$

$p^{(n+1)}(w|\theta_j) \propto \sum_{d \in C} c(w, d)(1 - p(z_{d,w} = B))p(z_{d,w} = j)$ $\forall j \in [1,k], \sum_{w \in V} p(w|\theta_j) = 1$

Figure 17.34   Computation of the EM Algorithm for estimating PLSA.

a loop until the likelihood converges. How do we know when the likelihood converges? We can keep track of the likelihood values in each iteration and compare the current likelihood with the likelihood from the previous iteration or the average of the likelihood from a few previous iterations. If the current likelihood is very similar to the previous one (judged by a threshold), we can assume that the likelihood has converged and can stop the algorithm.

In each iteration, the EM algorithm would first invoke the E-step followed by the M-step. In the E-step, it would augment the data by predicting the hidden variables. In this case, the hidden variable, $z_{d,w}$ indicates whether word w in d is from a "real" topic or the background. If it's from a real topic, it determines which of the k topics it is from.

From Figure 17.34, we see that in the E-step we need to compute the probability of z values for every unique word in each document. Thus, we can iterate over all the documents, and for each document, iterate over all the unique words in the document to compute the corresponding $p(z_{d,w})$. This computation involves computing the product of the probability of selecting a topic and the probability of word w given by the selected distribution. We can then normalize these products based on the constraints we have, to ensure $\sum_{j=1}^{k} p(z_{d,w} = j) = 1$. In this case, the normalization is among all the topics.

In the M-step, we will also collect the relevant counts and then normalize appropriately to obtain re-estimates of various parameters. We would use the estimated probability distribution $p(z_{d,w})$ to split the count of word w in document d among all the topics. Note that the same word would generally be split in different ways in different documents. Once we split the counts for all the words in this way, we can aggregate the split counts and normalize them. For example, to re-estimate $\pi_{d,j}$

(coverage of topic θ_j in document d), the relevant counts would be the counts of words in d that have been allocated to topic θ_j, and the normalizer would be the sum of all such counts over all the topics so that after normalization, we would obtain a probability distribution over all the topics. Similarly, to re-estimate $p(w \mid \theta_j)$, the relevant counts are the sum of all the split counts of word w in all the documents. These aggregated counts would then be normalized by the sum of such aggregated counts over all the words in the vocabulary so that after normalization, we again would obtain a distribution, this time over all the words rather than all the topics.

If we complete all the computation of the E-step before starting the M-step, we would have to allocate a lot of memory to keep track of all the results from the E-step. However, it is possible to interleave the E-step and M-step so that we can collect and aggregate relevant counts needed for the M-step while we compute the E-step. This would eliminate the need for storing many intermediate values unnecessarily.

17.5 Extension of PLSA and Latent Dirichlet Allocation

PLSA works well as a completely unsupervised method for analyzing topics in text data, thus it does not require any manual effort. While this is an advantage in the sense of minimizing human effort, the discovery of topics is solely driven by the data characteristics with no consideration of any extra knowledge about the topics and their coverage in the data set. Since we often have such extra knowledge or our application imposes a particular preference for the topics to be analyzed, it is beneficial or even necessary to impose some prior knowledge about the parameters to be estimated so that the estimated parameters would not only explain the text data well, but also be consistent with our prior knowledge. Prior knowledge or preferences may be available for all the parameters.

First, a user may have some expectations about which topics to analyze in the text data, and such knowledge can be used to define a prior on the topic word distributions. For example, an analyst may expect to see "retrieval models" as a topic in a data set with research articles about information retrieval, thus we would like to tell the model to allocate one topic to capture the retrieval models topic. Similarly, a user may be interested in analyzing review data about a laptop with a focus on specific aspects such as battery life and screen size, thus we again want the model to allocate two topics for battery life and screen size, respectively.

Second, users may have knowledge about what topics are (or are *not*) covered in a document. For example, if we have (topical) tags assigned to documents by users, we may regard the tags assigned to a document as knowledge about what topics

are covered in the document. Thus, we can define a prior on the topic coverage to ensure that a document can only be generated using topics corresponding to the tags assigned to it. This essentially gives us a constraint on what topics can be used to generate words in a document, which can be useful for learning co-occuring words in the context of a topic when the data are sparse and pure co-occurrence statistics are insufficient to induce a meaningful topic.

All such prior knowledge can be incorporated into PLSA by using Maximum A Posteriori Estimation (MAP) instead of Maximum Likelihood estimation. Specifically, we denote all the parameters by Λ and introduce a prior distribution $p(\Lambda)$ over all the possible values of Λ to encode our preferences. Such a prior distribution would technically include a distribution over all possible word distributions (for topic characterization) and all possible coverage distributions of topics in a document (for topic coverage), and can be defined based on whatever knowledge or preferences we would like to inject into the model. With such a prior, we can then estimate parameters by using MAP as follows:

$$\Lambda^* = \arg \max_\Lambda p(\Lambda)p(Data \mid \Lambda), \tag{17.6}$$

where $p(Data \mid \Lambda)$ is the likelihood function, which would be the sole term to maximize in the case of ML estimation. Adding the prior $p(\Lambda)$ would encourage the model to seek a compromise of the ML estimate (which maximizes $p(Data \mid \Lambda)$) and the mode of the prior (which maximizes $p(\Lambda)$).

There are potentially many different ways to define $p(\Lambda)$. However, it is particularly convenient to use a conjugate prior distribution, in which the prior density function $p(\Lambda)$ is of the same form as the likelihood function $p(Data \mid \Lambda)$ as a function of the parameter Λ. Due to the same form of the two functions, we can generally merge the two to derive a single function (again, of the same form). In other words, our posterior distribution is written as a function of the parameter, so the maximization of the posterior probability would be similar to the maximization of the likelihood function. Since the posterior distribution is of the same form as the likelihood function of the original data, we can interpret the posterior distribution as the likelihood function for an imagined pseudo data set that is formed by augmenting the original data with additional "pseudo data" such that the influence of the prior is entirely captured by the addition of such pseudo data to the original data.

When using such a conjugate prior, the computation of MAP can be done by using a slightly modified version of the EM algorithm that we introduced earlier for PLSA where appropriate counts of pseudo data are added to incorporate the prior. As a specific example, if we define a conjugate prior on the word distributions

EM Algorithm with Conjugate on $p(w|\theta_j)$

$$p(z_{d,w} = j) = \frac{\pi_{d,j}^{(n)} p^{(n)}(w|\theta_j)}{\sum_{j'=1}^{k} \pi_{d,j'}^{(n)} p^{(n)}(w|\theta_{j'})}$$

Prior: $p(w|\theta'_j)$

battery 0.5
life 0.5

$$p(z_{d,w} = B) = \frac{\lambda_B p(w|\theta_B)}{\lambda_B p(w|\theta_B) + (1 - \lambda_B)\sum_{j=1}^{k} \pi_{d,j}^{(n)} p^{(n)}(w|\theta_j)}$$

$$\pi_{d,j}^{(n+1)} = \frac{\sum_{w \in V} c(w, d)(1 - p(z_{d,w} = B)) p(z_{d,w} = j)}{\sum_{j'} \sum_{w \in V} c(w, d)(1 - p(z_{d,w} = B)) p(z_{d,w} = j')}$$

Pseudo counts
of w from
prior θ'

$$p^{(n+1)}(w|\theta_j) = \frac{\sum_{d \in C} c(w, d)(1 - p(z_{d,w} = B)) p(z_{d,w} = j) + \mu p(w|\theta'_j)}{\sum_{w' \in V} \sum_{d \in C} c(w', d)(1 - p(z_{d,w'} = B)) p(z_{d,w'} = j) + \mu}$$

What if $\mu = 0$? What if $\mu = +\infty$?

Sum of all pseudo counts

Figure 17.35 Maximum a posteriori estimation of PLSA with prior.

representing the topics $p(w \mid \theta_j)$, then the EM algorithm for computing the MAP is shown in Figure 17.35. We see that the difference is adding an additional pseudo count for word w in the M-step which is proportional to the probability of the word in the prior $p(w \mid \theta'_j)$. Specifically, the pseudo count is $\mu p(w \mid \theta'_j)$ for word w. The denominator needs to be adjusted accordingly (adding μ which is the sum of all the pseudo counts for all the words) to ensure the estimated word probabilities for a topic sum to one.

Here, $\mu \in [0, +\infty)$ is a parameter encoding the strength of our prior. If $\mu = 0$, we recover the original EM algorithm for PLSA, i.e., with no prior influence. A more interesting case is when $\mu = +\infty$, in such a case, the M-step is simply to set the estimated probability of a word $p(w \mid \theta_j)$ to the prior $p(w \mid \theta'_j)$, i.e., the word distribution is *fixed* to the prior. This is why we can interpret our heuristic inclusion of a background word distribution as a topic in PLSA as simply imposing such an infinitely strong prior on one of the topics. Intuitively, in Bayesian inference, this means that if the prior is infinitely strong, then no matter how much data we collect, we will not be able to override the prior. In general, however, as we increase the amount of data, we will be able to let the data dominate the estimate, eventually overriding the prior completely as we collect infinitely more data. A prior on the

coverage distribution π can be added in a similar way to the updating formula for $\pi_{d,j}$ to force the updated parameter value to give some topics higher probabilities by reducing the probabilities of others. In the extreme, it is also possible to achieve the effect of setting the probability of a topic to zero by using an infinitely strong prior that gives such a topic a zero probability.

PLSA is a generative model for modeling the words in a given document, but it is not a generative model for documents since it cannot give a probability of a new unseen document; it cannot give a distribution over all the possible documents. However, we sometimes would like to have a generative model for documents. For example, if we can estimate such a model for documents in each topic category, then we would be able to use the model for text categorization by comparing the probability of observing a document from the generative model of each category and assigning the document to the category whose generative model gives the highest probability to the document. The difficulty in giving a new unseen document a probability using PLSA is that the topic coverage parameter in PLSA is tied to an observed document, and we do not have available in the model the coverage of topics in a new unseen document, which is needed in order to generate words in a new document. Although it is possible to use a heuristic approach to estimate the topic coverage in an unseen document, a more principled way to solve the problem is to add priors on the parameters of PLSA and make a Bayesian version of the model. This has led to the development of the Latent Dirichlet Allocation (LDA) model.

Specifically, in LDA, the topic coverage distribution (a multinomial distribution) for each document is assumed to be drawn from a prior Dirichlet distribution, which defines a distribution over the entire space of the parameters of a multinomial distribution, i.e., a vector of probabilities of topics. Similarly, all the word distributions representing the latent topics in a collection of text are also assumed to be drawn from another Dirichlet distribution. In PLSA, both the topic coverage distribution and the word distributions are assumed to be (unknown) parameters in the model. In LDA, they are no longer parameters of the model since they are assumed to be drawn from the corresponding Dirichlet (prior) distributions.

Thus, LDA only has parameters to characterize these two kinds of Dirichlet distributions. Once these parameters are fixed, the behavior of these two Dirichlet distributions would be fixed, and thus the behavior of the entire generative model would also be fixed. Once we have sampled all the word distributions for the whole collection (which shares these topics), and the topic coverage distribution for a document, the rest of the process of generating words in the document is exactly the same as in PLSA. The generalization of PLSA to LDA by imposing Dirichlet priors is illustrated in Figure 17.36, where we see that the Dirichlet distribution governing

$$\text{PLSA} \rightarrow \text{LDA}$$
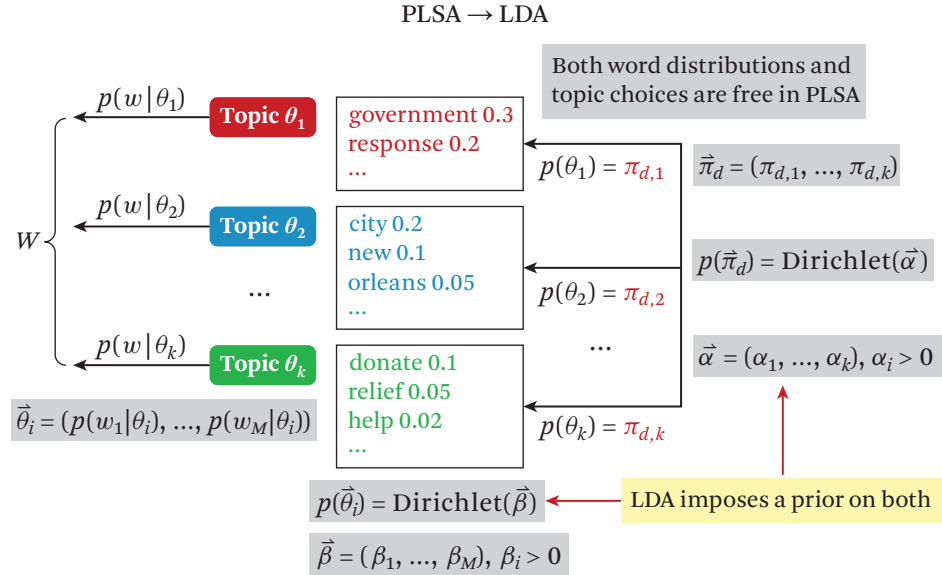
Figure 17.36   Illustration of LDA as PLSA with a Dirichlet prior.

the topic coverage has k parameters, $\alpha_1, \ldots, \alpha_k$, and the Dirichlet distribution governing the topic word distributions has M parameters, $\beta_1, \ldots, \beta_M$. Each α_i can be interpreted as the pseudo count of the corresponding topic θ_i according to our prior, while each β_i can be interpreted as the pseudo count of the corresponding word w_i according to our prior. With no additional knowledge, they can all be set to uniform counts, which in effect, assumes that we do not have any preference for any word in each word distribution and we do not have any preference for any topic either in each document.

The likelihood function of LDA is given in Figure 17.37 where we also make a comparison between the likelihood of PLSA and that of LDA. The comparison allows us to see that both PLSA and LDA share the common generative model component to define the probability of observing a word w in document d from a mixture model involving k word distributions, $\theta_1, \ldots, \theta_k$, representing k topics with a topic coverage distribution $\pi_{d,j}$. Indeed, such a mixture of unigram language models is the common component in most topic models, and is key for modeling documents with multiple topics covered in the same document. However, the likelihood function for a document and the entire collection C is clearly different with LDA adding the uncertainty of the topic coverage distribution and the uncertainty of all the word distributions in the form of an integral.

Likelihood Functions for PLSA vs. LDA

PLSA

$$p_d(w|\{\theta_j\}, \{\pi_{d,j}\}) = \sum_{j=1}^{k} \pi_{d,j} p(w|\theta_j)$$

$$\log p(d|\{\theta_j\}, \{\pi_{d,j}\}) = \sum_{w \in V} c(w, d) \log \left[\sum_{j=1}^{k} \pi_{d,j} p(w|\theta_j)\right]$$

$$\log p(C|\{\theta_j\}, \{\pi_{d,j}\}) = \sum_{d \in C} \log p(d|\{\theta_j\}, \{\pi_{d,j}\})$$

Core assumption
in all topic models

PLSA component

LDA

$$p_d(w|\{\theta_j\}, \{\pi_{d,j}\}) = \sum_{j=1}^{k} \pi_{d,j} p(w|\theta_j)$$

$$\log p(d|\vec{\alpha}, \{\theta_j\}) = \int \left[\sum_{w \in V} c(w, d) \log \left[\sum_{j=1}^{k} \pi_{d,j} p(w|\theta_j)\right]\right] p(\vec{\pi}_d|\vec{\alpha}) d\vec{\pi}_d$$

$$\log p(C|\vec{\alpha}, \vec{\beta}) = \int \sum_{d \in C} \log p(d|\vec{\alpha}, \{\theta_j\}) \prod_{j=1}^{k} p(\theta_j|\vec{\beta}_j) d\theta_1 \cdots d\theta_k$$

Added by LDA
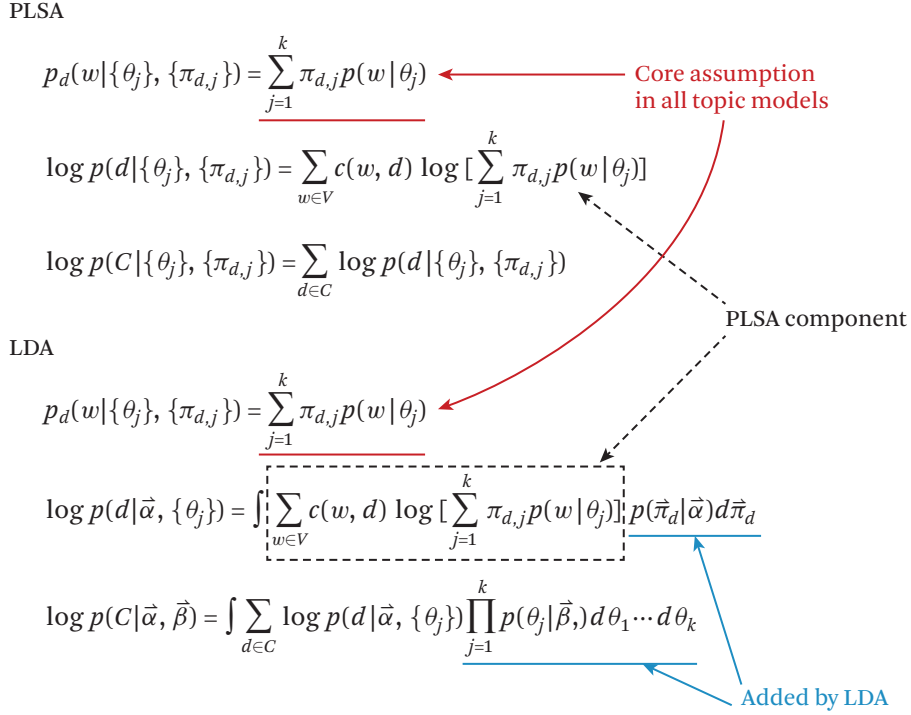
Figure 17.37   **Likelihood function of PLSA and LDA.**

Although the likelihood function of LDA is more complicated than PLSA, we can still use the MLE to estimate its parameters, $\vec{\alpha}$ and $\vec{\beta}$:

$$(\hat{\vec{\alpha}}, \hat{\vec{\beta}}) = \arg \max_{\vec{\alpha}, \vec{\beta}} \log p(C \mid \vec{\alpha}, \vec{\beta}). \tag{17.7}$$

Naturally, the computation required to solve such an optimization problem is more complicated than LDA.

It is now easy to see that LDA has only $k + M$ parameters, far fewer than PLSA. However, the cost is that the interesting output that we would like to generate in topic analysis, i.e., the k word distributions $\{\theta_i\}$ characterizing all the topics in a collection, and the topic coverage distribution $\{\pi_{d,j}\}$ for each document, is unfortunately, no longer immediately available to us after we estimate all the parameters. Indeed, as usually happens in Bayesian inference, to obtain values of such latent variables in LDA, we must rely on posterior inference. That is, we must compute

$p(\{\theta_i\}, \{\pi_{d,j}\} \mid C, \alpha, \beta)$ as follows by using Bayes' Rule:

$$p(\{\theta_i\}, \{\pi_{d,j}\} \mid C, \alpha, \beta) = \frac{p(C \mid \{\theta_i\}, \{\pi_{d,j}\})\, p(\{\theta_i\}, \{\pi_{d,j}\} \mid \alpha, \beta)}{p(C \mid \alpha, \beta)}. \qquad (17.8)$$

This gives us a posterior distribution over all the possible values of these interesting variables, from which we can then further obtain a point estimate or compute other interesting properties that depend on the distribution. The computation process is once again complicated due to the integrals involved in some of the probabilities. Many different inference algorithms have been proposed. A very popular and efficient approach is collapsed Gibbs sampling, which works in a very similar way to the EM algorithm of PLSA.

Empirically, LDA and PLSA have been shown to work similarly on various tasks when using such a model to learn a low-dimensional semantic representation of documents (by using $\pi_{d,j}$ to represent a document in the k-dimensional space). The learned word distributions also tend to look very similar.

17.6 Evaluating Topic Analysis

Topic analysis evaluation has similar difficulties to information retrieval evaluation. In both cases, there is usually not one true answer, and evaluation metrics heavily depend on the human issuing judgements. What defines a topic? We addressed this issue the best we could when defining the models, but the challenging nature of such a seemingly straightforward question complicates the eventual evaluation task.

Log-likelihood and model perplexity are two common evaluation measures used by language models, and they can be applied for topic analysis in the same way. Both are predictive measures, meaning that held-out data is presented to the model and the model is applied to this new information, calculating its likelihood. If the model generalizes well to this new data (by assigning it a high likelihood or low perplexity), then the model is assumed to be sufficient.

In Chapter 13, we mentioned Chang et al. [2009]. Human judges responded to intrusion detection scenarios to measure the coherency of the *topic-word distributions*. A second test that we didn't cover in the word association evaluation is the *document-topic distribution* evaluation. This test can measure the coherency of topics discovered from documents through the previously used intrusion test.

The setup is as follows: given a document d from the collection the top three topics are chosen; call these most likely topics θ_1, θ_2, and θ_3. An additional low-probability topic θ_u is also selected, and displayed along with the top three topics.

The title and a short snippet is shown from d along with the top few high-probability words from each topic. The human judge must determine which θ is θ_u. As with the word intrusion test, the human judge should have a fairly easy task if the top three topics make sense together and with the document title and snippet. If it's hard to discern θ_u, then the top topics must not be an adequate representation of d. Of course, this process is repeated for many different documents in the collection.

Directly from Chang et al. [2009]:

> . . . we demonstrated that traditional metrics do not capture whether topics are coherent or not. Traditional metrics are, indeed, negatively correlated with the measures of topic quality.

"Traditional metrics" refers to log-likelihood of held-out data in the case of generative models. This misalignment of results is certainly a pressing issue, though most recent research still relies on the traditional measures to evaluate new models.

Downstream task improvement is perhaps the most effective (and transparent) evaluation metric. If a different topic analysis variant is shown to statistically significantly improve some task precision, then an argument may be made to prefer the new model. For example, if the topic analysis is meant to produce new features for text categorization, then classification accuracy is the metric we'd wish to improve. In such a case, log-likelihood of held-out data and even topic coherency is not a concern if the classification accuracy improves—although model interpretability may be compromised if topics are not human-distinguishable.

17.7   Summary of Topic Models

In summary, we introduced techniques for topic analysis in this chapter. We started with the simple idea of using one term to represent a topic, and discussed the deficiency of such an approach. We then introduced the idea of representing a topic with a word distribution, or a unigram language model, and introduced the PLSA model, which is a mixture model with k unigram language models representing k topics. We also added a pre-specified background language model to help discover discriminative topics, because this background language model can help attract the common terms. We used the maximum likelihood estimator (computed using the EM algorithm) to estimate the parameters of PLSA. The estimated parameter values enabled us to discover two things, one is k word distributions with each one representing a topic, and the other is the proportion of each topic in each document.

The topic word distributions and the detailed characterization of coverage of topics in each document can enable further analysis and applications. For exam-

ple, we can aggregate the documents in a particular time period to assess the coverage of a particular topic in the time period. This would allow us to generate a temporal trend of topics. We can also aggregate topics covered in documents associated with a particular author to reveal the expertise areas of the author. Furthermore, we can also cluster terms and cluster documents. In fact, each topic word distribution can be regarded as a cluster (for example, the cluster can be easily obtained by selecting the top N words with the highest probabilities). So we can generate term clusters easily based on the output from PLSA. Documents can also be clustered in the same way: we can assign a document to the topic cluster that's covered most in the document. Recall that $\pi_{d,j}$ indicates to what extent each topic θ_j is covered in document d. We can thus assign the document to the topical cluster that has the highest $\pi_{d,j}$. Another use of the results from PLSA is to treat the inferred topic coverage distribution in a document as an alternative way of representing the document in a low-dimensional semantic space where each dimension corresponds to a topic. Such a representation can supplement the bag-of-words representation to enhance inexact matching of words in the same topic, which can generally be beneficial (e.g., for information retrieval, text clustering, and text categorization).

Finally, a variant of PLSA called latent Dirichlet allocation (LDA) extends PLSA by adding priors to the document-topic distributions and topic-word distributions. These priors can force a small number of topics to dominate in each document, which makes sense because usually a document is only about one or two topics as opposed to a true mixture of all k topics. Secondly, adding these priors can give us sparse word distributions in each topic as well, which mimics the Zipfian distribution of words we've discussed previously. Finally, LDA is a generative model, which can be used to simulate (generate) values of parameters in the model as well as apply the model to a new, unseen document [Blei et al. 2003].

Bibliographic Notes and Further Reading

We've mentioned the original PLSA paper [Hofmann 1999] and its successor LDA [Blei et al. 2003]. Asuncion et al. [2009] compares various inference methods for topic models and concludes that they are all very similar. For evaluation, we've referenced Chang et al. [2009] in this chapter, and it showed that convenient mathematical measures such as log-likelihood are not correlated with human measures. For books, Koller and Friedman [2009] is a large and detailed introduction to probabilistic graphical models. Bishop [2006] covers graphical models, mixture models, EM, and inference in the larger scope of machine learning. Steyvers and Griffiths

[2007] is a short summary of topic models alone. In the exercises, we mention supervised LDA [McAuliffe and Blei 2008]. There are many other variants of LDA such as MedLDA [Zhu et al. 2009] (another supervised model which attempts to maximize the distance between classes) and LabeledLDA [Ramage et al. 2009] (which incorporates metadata tags).

Exercises

17.1. What is the input and output of the two-topic mixture model?

17.2. What is the input and output of PLSA?

17.3. For a product review dataset, there are k different product types. The true value of k is in the range [2, 5]. How many product types do you think there were? How can you use topic analysis to help you? (A product type is something like "CPU" or "router").

17.4. Give an idea about how you could use topic models to enhance search results. What type of access mode does your suggestion support?

17.5. Give an idea about how you could use topic models for a document representation in vector space. What does a similarity measure capture for this representation?

17.6. Sketch an idea about how you could use PLSA to model topical trends over time, given a dataset of documents that are tagged with dates.

17.7. Chapter 18 discusses sentiment analysis and opinion mining. In order to discover positive and negative sentiment topics, we set $k = 2$ and run a topic analysis method. What is an issue with this idea?

17.8. We mentioned that PLSA is a discriminative model and LDA is a generative model. Discuss how these differences affect:

   (a) defining the model,
   (b) incorporating prior knowledge,
   (c) learning the model parameters (inference), and
   (d) applying the model to new data.

17.9. An alternative topic analysis evaluation scheme is to hold out a certain number of words in the vocabulary from some documents. Explain how this can be used to evaluate topic models. Does this evaluate the topic-document distributions, topic-word distributions, or both?

17.10. Supervised LDA (sLDA) is a probabilistic model over labeled documents, where each document contains some real-valued response variable. For example, if the dataset is movie reviews, the response variable could be the average rating. Explain what additional knowledge sLDA can discover in comparison to LDA or PLSA aside from predicting response variables for a new document.

Opinion Mining and Sentiment Analysis

In this chapter, we're going to talk about mining a different kind of knowledge, namely knowledge about the observer or humans that have generated the text data. In particular, we're going to talk about opinion mining and sentiment analysis. As we discussed earlier, text data can be regarded as data generated from humans as subjective sensors. In contrast, we have other devices such as video recorders that can report what's happening in the real world to generate data. The main difference between text data and other data (like video data) is that it has rich opinions, and the content tends to be subjective because it's generated from humans, as shown in Figure 18.1. This is actually a unique advantage of text data as compared to other data because this offers us a great opportunity to understand the observers—we can mine text data to understand their opinions.

Let's start with the concept of an opinion. It's not that easy to formally define an opinion, but for the most part we would define an opinion as a subjective statement describing what a person believes or thinks about something, as shown in Figure 18.2.

Let's first look at the key word *subjective* in the figure; this is in contrast with an *objective* statement or factual statement. This is a key differentiating factor from opinions which tends to be not easy to prove wrong or right, because it reflects what the person thinks about something. In contrast, an objective statement can usually be proved wrong or right. For example, you might say a computer has a screen and a battery. Clearly, that's something you can check; either it has a battery or doesn't. In contrast with this, think about a sentence such as, "This laptop has the best battery life" or "This laptop has a nice screen." These statements are more subjective and it's very hard to prove whether they are wrong or right.

The word *person* indicates an opinion holder. When we talk about an opinion, it's about an opinion held by *someone*. Of course, an opinion will depend on culture,
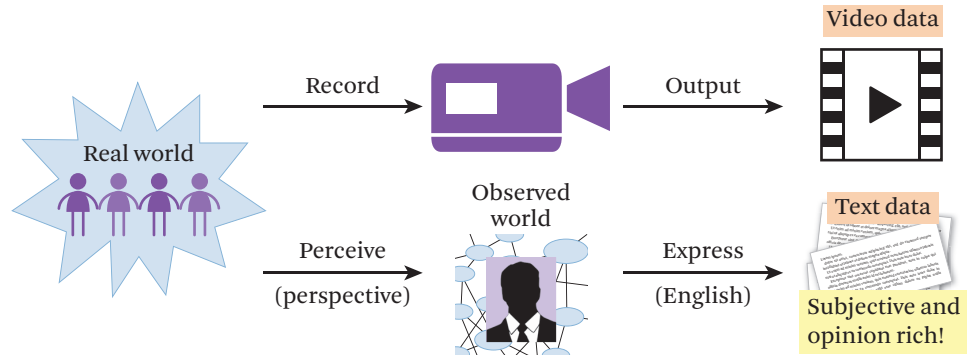
Figure 18.1 **Objective vs. Subjective Sensors.**
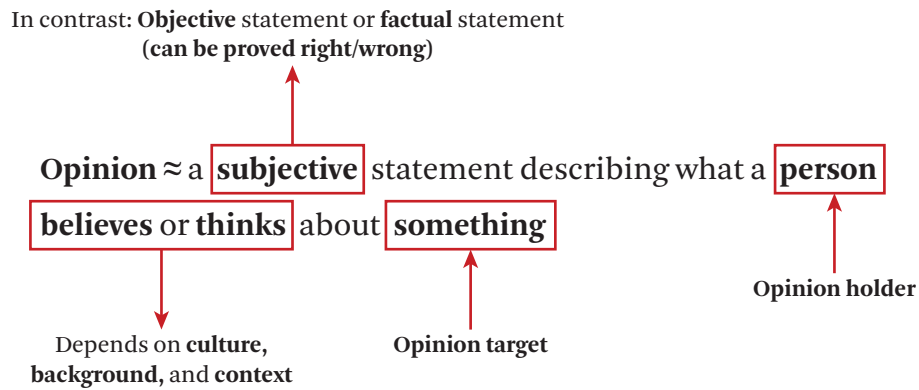
Figure 18.2 **Definition of "opinion."**

background, and the context in general. This thought process shows that there are multiple elements that we need to include in order to characterize opinions.

The next logical question is "What's a basic opinion representation?" It should include at least three elements. First, it has to specify who the opinion holder is. Second, it must also specify the target, or what the opinion is about. Third, of course, we want the opinion content. If you can identify these, we get a basic understanding of opinions. If we want to understand further, we need an enriched opinion representation. That means we also want to understand, for example, the context of the opinion and in what situation the opinion was expressed. We would also like to understand the opinion sentiment; i.e., whether it is a positive or negative feeling.

A Sentence in the News (Implicit Holder and Target)
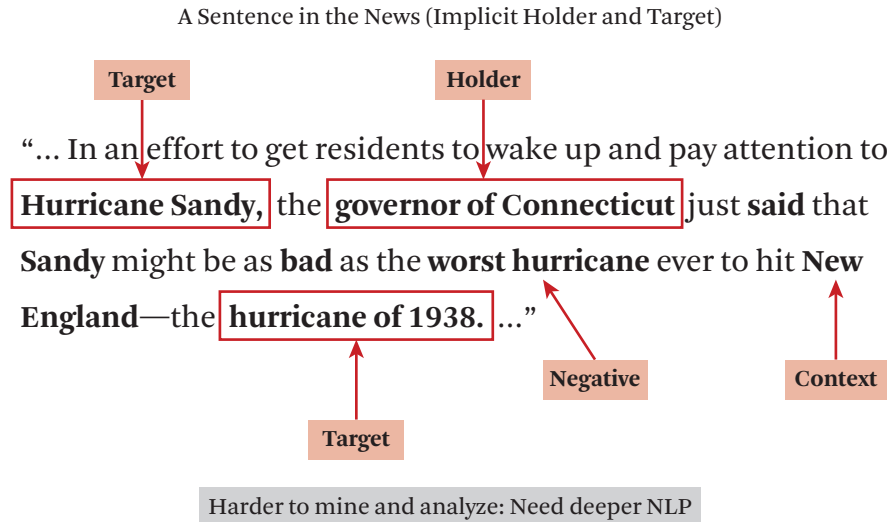
Figure 18.3   **A sentence from news with sentiment.** (Courtesy of © 2012 Henry Blodget / Business Insider)

Let's take a simple example of a product review. In this case, we already know the opinion holder and the target. When the review is posted, we can usually extract this information. Additional understanding by analyzing the user-generated text adds value to mining the opinions.

Figure 18.3 shows a sentence extracted from a news article. In this case, we have an implicit holder and an implicit target since we don't automatically know this information. This makes the task harder. As humans, we can identify the opinion holder as the governor of Connecticut. We can also identify the target, Hurricane Sandy, but there is also another target mentioned which is the hurricane of 1938. What's the opinion? There is negative sentiment indicated by words like *bad* and *worst*. We can also identify context, which is New England. All these elements must be extracted by using NLP techniques. Analyzing the sentiment in news is still quite difficult; it's more difficult than the analysis of opinions in product reviews.

There are also some other interesting variations. First, let's think about the opinion holder. The holder could be an individual or it could be group of people. Sometimes, the opinion is from a committee or from a whole country of people. Opinion targets will vary greatly as well; they can be about one entity, a particular person, a particular product, a particular policy, and so on. An opinion could also only be about one attribute of a particular entity. For example, it could just be

about the battery of a smartphone. It could even be someone else's opinion, and one person might comment on another person's opinion. Clearly, there is much variation here that will cause the problem to take different forms.

Opinion *content* can also vary on the surface: we can identify a one-sentence opinion or a one-phrase opinion. We can also have longer text to express an opinion, such as a whole news article. Furthermore, we can identify the variation in the sentiment or emotion of the opinion holder. We can distinguish positive vs. negative or neutral sentiment.

Finally, the opinion context can also vary. We can have a simple context, like a different time or different locations. There could be also complex contexts, such as some background of a topic being discussed. When an opinion is expressed in a particular discourse context, it has to be interpreted in different ways than when it's expressed in another context.

From a computational perspective, we're mostly interested in what opinions can be extracted from text data. One computational objective might be to determine the target of an opinion. For example, "I don't like this phone at all," is clearly an opinion by the speaker about a phone. In contrast, the text might also report opinions about others. One could make an observation about another person's opinion and report this opinion. For example, "I believe he loves the painting." That opinion is really expressed from another person; it doesn't mean this person loves that painting. Clearly, these two kinds of opinions need to be analyzed in different ways. Sometimes, a reviewer might mention opinions of his or her friend.

Another complication is that there may be indirect opinions or inferred opinions that can be obtained by making inferences about what is expressed in the text that might not necessarily look like opinion. For example, one statement might be, "This phone ran out of battery in only one hour." Now, this is in a way a factual statement because it's either true or false. However, one can also infer some negative opinions about the quality of the battery of this phone, or the opinion about the battery life. These are interesting variations that we need to pay attention to when we extract opinions.

The task of opinion mining can be defined as taking contextualized input to generate a set of opinion representations, as shown in Figure 18.4. Each representation should identify the opinion holder, target, content, and context. Ideally, we can also infer opinion sentiment from the comment and the context to better understand the opinion. Often, some elements of the representation are already known. We just saw an example in the case of a product review where the opinion holder and the opinion target are often explicitly identified.
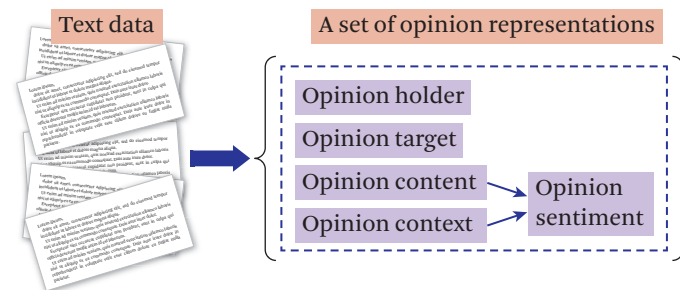
Figure 18.4 The task of opinion mining.

Opinion mining is important and useful for three major reasons. First, it can aid decision support; it can help us optimize our decisions. We often look at other people's opinions by reading their reviews in order to make a decision such as which product to buy or which service to use. We also would be interested in others' opinions when we decide whom to vote for. Policymakers may also want to know their constituents' opinions when designing a new policy. The second application is to understand people. For example, it could help understand human preferences. We could optimize a product search engine or optimize a recommender system if we know what people are interested in. It also can help with advertising; we can have targeted advertising if we know what kind of people tend to like which types of products. The third kind of application is aggregating opinions from many humans at once to assess a more general opinion. This would be very useful for business intelligence where manufacturers want to know where their products have advantages or disadvantages. What are the winning features of their products or competitors' products? Market research has to do with understanding consumer opinions. Data-driven social science research can benefit from this because they can do text mining to understand group opinions. If we aggregate opinions from social media, we can study the behavior of people on social networks. In general, we can gain a huge advantage in any prediction task because we can leverage the text data as extra data to any problem.

18.1 Sentiment Classification

If we assume that most of the elements in an opinion representation are already known, then our only task may be sentiment classification. That is, suppose we know the opinion holder and the opinion target, and also know the content and

the context of the opinion. The only component remaining is to decide the opinion sentiment of the review. Sentiment classification can be defined more specifically as follows: the input is an opinionated text object and the output is typically a sentiment label (or a sentiment tag) that can be defined in two ways. One is **polarity analysis,** where we have categories such as positive, negative, or neutral. The other is **emotion analysis** that can go beyond polarity to characterize the precise feeling of the opinion holder. In the case of polarity analysis, we sometimes also have numerical ratings as you often see in some reviews on the Web. A rating of five might denote the most positive, and one may be the most negative, for example. In emotion analysis there are also different ways to design the categories. Some typical categories are happy, sad, fearful, angry, surprised, and disgusted. Thus, the task is essentially a classification task, or categorization task, as we've seen before.

If we simply apply default classification techniques, the accuracy may not be good since sentiment classification requires some improvement over regular text categorization techniques. In particular, it needs two kind of improvements. One is to use more sophisticated features that may be more appropriate for sentiment tagging. The other is to consider the *order* of these categories, especially in polarity analysis since there is a clear order among the choices. For example, we could use **ordinal regression** to predict a value within some range. We'll discuss this idea in the next section.

For now, let's talk about some features that are often very useful for text categorization and text mining in general, but also especially needed for sentiment analysis. The simplest feature is character n-grams, i.e., sequences of n adjacent characters treated as a unit. This is a very general and robust way to represent text data since we can use this method for any language. This is also robust to spelling errors or recognition errors; if you misspell a word by one character, this representation still allows you to match the word as well as when it occurs in the text correctly. Of course, such a representation would not be as discriminating as words.

Next, we have word n-grams, a sequence of *words* as opposed to *characters*. We can have a mix of these with different n-values. Unigrams are often very effective for text processing tasks; it's mostly because words are the basic unit of information used by humans for communication. However, unigram words may not be sufficient for a task like sentiment analysis. For example, we might see a sentence, "It's not good" or "It's not as good as something else." In such a case, if we just take the feature *good*, that would suggest a positive text sample. Clearly, this would not be accurate. If we take a bigram ($n = 2$) representation, the bigram *not good* would appear, making our representation more accurate. Thus, longer n-grams are generally more discriminative. However, long n-grams may cause overfitting because

they create very unique features that machine learning programs associate as being highly correlated with a particular class label when in reality they are not. For example, if a 7-gram phrase appears only in a positive training document, that 7-gram would always be associated with positive sentiment. In reality, though, the 7-gram just happened to occur with the positive document and no others because it was so rare.

We can consider n-grams of part-of-speech tags. A bigram feature could be an adjective followed by a noun. We can mix n-grams of words and n-grams of POS tags. For example, the word *great* might be followed by a noun, and this could become a feature—a hybrid feature—that could be useful for sentiment analysis.

Next, we can have word classes. These classes can be syntactic like POS tags, or could be semantic by representing concepts in a thesaurus or ontology like Word-Net [Princeton University 2010]. Or, they can be recognized name entities (like people or place), and these categories can be used to enrich the representation as additional features. We also can learn word clusters since we've talked about mining associations of words in Chapter 13. We can have clusters of paradigmatically related words or syntagmatically related words, and these clusters can be features to supplement the base word representation.

Furthermore, we can have a frequent pattern syntax which represents a frequent word set; these are words that do not necessarily occur next to each other but often occur in the same context. We'll also have locations where the words may occur more closely together, and such patterns provide more discriminative features than words. They may generalize better than just regular n-grams because they are frequent, meaning they are expected to occur in testing data, although they might still face the problem of overfitting as the features become more complex. This is a problem in general, and the same is true for parse tree-based features, e.g., frequent subtrees. Those are even more discriminating, but they're also more likely to cause overfitting.

In general, pattern discovery algorithms are very useful for feature construction because they allow us to search a large space of possible features that are more complex than words, and natural language processing is very important to help us derive complex features that can enrich text representations.

As we've mentioned in Chapter 15, feature design greatly affects categorization accuracy and is arguably the most important part of any machine learning application. It would be most effective if you can combine machine learning, error analysis, and specific domain knowledge when designing features. First, we want to use domain knowledge, that is, a specialized understanding of the problem. With this, we can design a basic feature space with many possible features for the machine

learning program to work on. Machine learning methods can be applied to select the most effective features or to even construct new features. These features can then be further analyzed by humans through error analysis, using evaluation techniques we discuss in this book. We can look at categorization errors and further analyze what features can help us recover from those errors or what features cause overfitting. This can lead into feature validation that will cause a revision in the feature set. These steps are then iterated until a desired accuracy is achieved.

In conclusion, a main challenge in designing features is to optimize a tradeoff between exhaustivity and specificity. This tradeoff turns out to be very difficult. Exhaustivity means we want the features to have high coverage on many documents. In that sense, we want the features to be frequent. Specificity requires the features to be discriminative, so naturally the features tend to be less frequent. Clearly, this causes a tradeoff between frequent versus infrequent features. Particularly in our case of sentiment analysis, feature engineering is a critical task.

18.2   Ordinal Regression

In this section, we will discuss ordinal logistic regression for sentiment analysis. A typical sentiment classification problem is related to rating prediction because we often try to predict sentiment value on some scale, e.g., positive to negative with other labels in between. We have an opinionated text document d as input, and we want to generate as output a rating in the range of 1 through k. Since it's a discrete rating, this could be treated as a categorization problem (finding which is the correct of k categories). Unfortunately, such a solution would not consider the order and dependency of the categories. Intuitively, the features that can distinguish rating 2 from 1 may be similar to those that can distinguish k from $k - 1$. For example, positive words generally suggest a higher rating. When we train a categorization problem by treating these categories as independent, we would not capture this. One approach that addresses this issue is **ordinal logistic regression**.

Let's first think about how we use logistic regression for binary sentiment (which is a binary categorization problem). Suppose we just wanted to distinguish positive from negative. The predictors (features) are represented as X, and we can output a score based on the log probability ratio:

$$\log \frac{p(Y = 1 \mid X)}{p(Y = 0 \mid X)} = \log \frac{p(Y = 1 \mid X)}{1 - p(Y = 1 \mid X)} = \beta_0 + \sum_{i=1}^{M} x_i \beta_i, \qquad (18.1)$$

or the conditional probability

$$p(Y = 1 \mid X) = \frac{\exp\left\{\beta_0 + \sum_{i=1}^{M} x_i \beta_i\right\}}{1 + \exp\left\{\beta_0 + \sum_{i=1}^{M} x_i \beta_i\right\}}. \tag{18.2}$$

There are M features all together and each feature value x_i is a real number. As usual, these features can be a representation of a text document. X is a binary response variable 0 or 1, where 1 means X is positive and 0 means X is negative. Of course, this is then a standard two category categorization problem and we can apply logistic regression. You may recall from Chapter 10 that in logistic regression, we assume the log probability that $Y = 1$ is a linear function of the features. This would allow us to also write $p(Y = 1 \mid X)$ as a transformed form of the linear function of the features. The β_i's are parameters. This is a direct application of logistic regression for binary categorization.

If we have multiple categories or multiple levels, we will adapt the binary logistic regression problem to solve this multilevel rating prediction, as illustrated in Figure 18.5. The idea is that we can introduce multiple binary classifiers; in each case we ask the classifier to predict whether the rating is j or above. So, when $Y_j = 1$, it means the rating is j or above. When it's 0, that means the rating is lower than j. If we want to predict a rating in the range of 1 to k, we first have one classifier to distinguish k versus the others. Then, we're going to have another classifier to distinguish $k - 1$ from the rest. In the end, we need a classifier to distinguish between 2 and 1 which altogether gives us $k - 1$ classifiers.

$$Y_j = \begin{cases} 1 & \text{rating is } j \text{ or above} \\ 0 & \text{rating is lower than } j \end{cases} \qquad \textbf{Predictors: } X = (x_1, x_2, ..., x_M), x_i \in \Re$$
$$\textbf{Rating: } r \in \{1, 2, ..., k\}$$

$$\log \frac{p(Y_j = 1 \mid X)}{p(Y_j = 0 \mid X)} = \log \frac{p(r \geq j \mid X)}{1 - p(r \geq j \mid X)} = \alpha_j + \sum_{i=1}^{M} x_i \beta_{ji} \quad \beta_{ji} \in \Re$$

Rating

k — Classifier 1
k − 1 — Classifier 2
k − 2
...
2 — Classifier $k - 1$
1

$$p(r \geq j \mid X) = \frac{e^{\alpha_j + \sum_{i=1}^{M} x_i \beta_{ji}}}{e^{\alpha_j + \sum_{i=1}^{M} x_i \beta_{ji}} + 1}$$
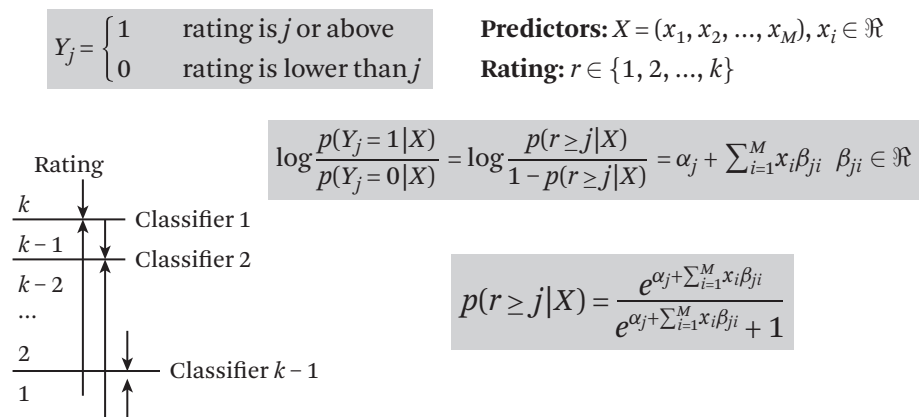
Figure 18.5 Logistic regression for multiple-level sentiment analysis.

Text object: $X = (x_1, x_2, ..., x_M), x_i \in \Re$

Rating: $r \in \{1, 2, ..., k\}$

After training $k - 1$
Logistic regression classifiers

$$p(r \geq j | X) = \frac{e^{\alpha_j + \sum_{i=1}^{M} x_i \beta_{ji}}}{e^{\alpha_j + \sum_{i=1}^{M} x_i \beta_{ji}} + 1}$$

$j = k, k - 1, ..., 2$

Flowchart:

$p(r \geq k | X) > 0.5?$ — Yes → $r = k$
No ↓
$p(r \geq k - 1 | X) > 0.5?$ — Yes → $r = k - 1$
...
No ↓
$p(r \geq 2 | X) > 0.5?$ — Yes → $r = 2$
No ↓
$r = 1$
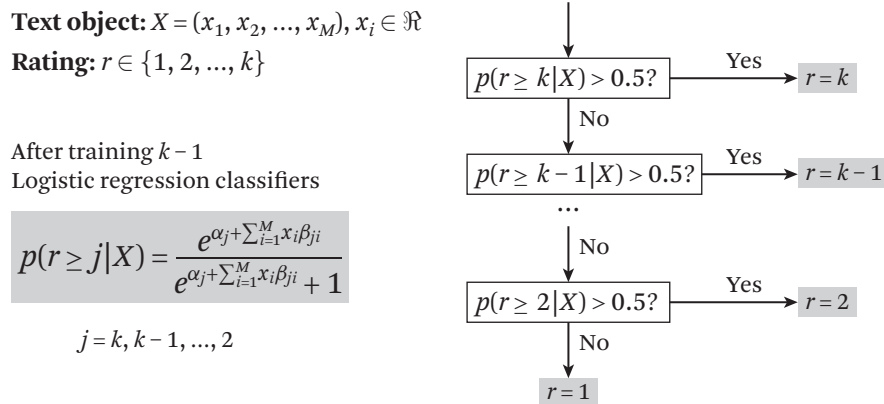
Figure 18.6   **Multi-level logistic regression for sentiment analysis: prediction of ratings.**

With this modification, each classifier needs a different set of parameters, yielding many more parameters overall. We will index the logistic regression classifiers by an index j, which corresponds to a rating level. This is to make the notation more consistent with what we show in the ordinal logistic regression. So, we now have $k - 1$ regular logistic regression classifiers, each with its own set of parameters. With this approach, we can now predict ratings, as shown in Figure 18.6.

After we have separately trained these $k - 1$ logistic regression classifiers, we can take a new instance and then invoke classifiers sequentially to make the decision. First, we look at the classifier that corresponds to the rating level k. This classifier will tell us whether this object should have a rating of k or not. If the probability according to this logistic regression classifier is larger than 0.5, we're going to say yes, the rating is k. If it's less than 0.5, we need to invoke the next classifier, which tells us whether it's at least $k - 1$. We continue to invoke the classifiers until we hit the end when we need to decide whether it's 2 or 1.

Unfortunately, such a strategy is not an optimal way of solving this problem. Specifically, there are two issues with this approach. The first problem is that there are simply too many parameters. For each classifier, we have $M + 1$ parameters with $k - 1$ classifiers all together, so the total number of parameters is $(k - 1) \cdot (M + 1)$. When a classifier has many parameters, we would in general need more training data to help us decide the optimal parameters of such a complex model.

The second problem is that these $k - 1$ classifiers are not really independent. We know that, in general, words that are positive would make the rating higher for *any* of these classifiers, so we should be able to take advantage of this fact. This is

Key idea: $\forall i = 1, ..., M, \forall j = 3, ..., k, \beta_{ji} = \beta_{j-1i}$

$\rightarrow$ Share training data $\rightarrow$ Reduce number of parameters

$$\log\frac{p(Y_j = 1|X)}{p(Y_j = 0|X)} = \log\frac{p(r \geq j|X)}{1 - p(r \geq j|X)} = \alpha_j + \sum_{i=1}^{M} x_i \beta_i \quad \beta_i \in \Re$$

Rating

k

Classifier 1

k − 1

Classifier 2

k − 2

...

$$p(r \geq j|X) = \frac{e^{\alpha_j + \sum_{i=1}^{M} x_i \beta_i}}{e^{\alpha_j + \sum_{i=1}^{M} x_i \beta_i} + 1}$$

2

1

Classifier k − 1   How many parameters are there in total? $M + k - 1$
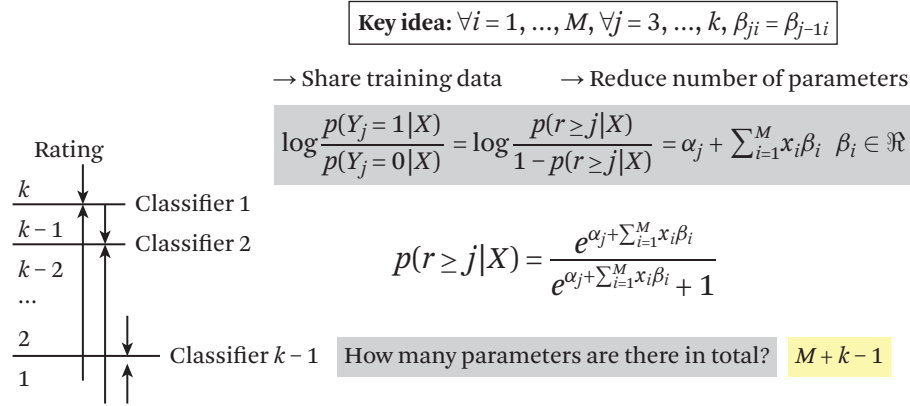
Figure 18.7   **The idea of ordinal logistic regression.**

precisely the idea of ordinal logistic regression, which is an improvement over the $k - 1$ independent logistic regression classifiers, as shown in Figure 18.7.

The improvement is to tie the β parameters together; that means we are going to assume the β values are the same for all the $k - 1$ classifiers. This encodes our intuition that positive words (in general) would make a higher rating more likely. In fact, this would allow us to have two benefits. One is to reduce the number of parameters significantly. The other is to allow us share the training data amongst all classifiers since the parameters are the same. In effect, we have more data to help us choose good β values.

The resulting formula would look very similar to what we've seen before, only now the β parameter has just one index that corresponds to a single feature; it no longer has the other indices that correspond to rating levels. However, each classifier still has a distinct predicted rating value. Of course, this value is needed to predict the different rating levels. So α_j is different since it depends on j, but the rest of the parameters (the β_i's) are the same. We now have $M + k - 1$ parameters.

It turns out that with this idea of tying all the parameters, we end up having a similar way to make decisions, as shown in Figure 18.8.

More specifically, the criteria whether the predictor probabilities are at least 0.5 or above is equivalent to whether the score of the object is larger than or equal to α_k. The scoring function is just taking a linear combination of all the features with the β values. This means now we can simply make a rating decision by looking at the value of this scoring function and seeing which bracket it falls into. In this approach, we're going to score the object by using the features and trained parameter values.

$$p(r \geq j | X) \geq 0.5 \Leftrightarrow \frac{e^{\alpha_j + \text{score}(X)}}{e^{\alpha_j + \text{score}(X)} + 1} \geq 0.5 \Leftrightarrow \text{score}(X) \geq -\alpha_j$$
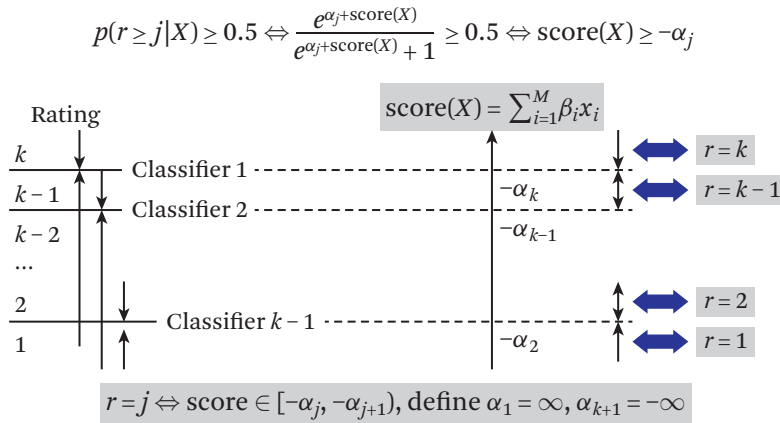
Figure 18.8   **The decision process with ordinal logistic regression.**

This score will then be compared with a set of trained α values to see which range the score is in. Then, using the range, we can decide which rating the object should receive.

18.3   Latent Aspect Rating Analysis

In this section, we're going to continue discussing opinion mining and sentiment analysis. In particular, we're going to introduce Latent Aspect Rating Analysis (LARA) which allows us to perform detailed analysis of reviews with overall ratings.

Figure 18.9 shows two hotel reviews. Both reviewers are given five stars. If you just look at the overall score, it's not very clear whether the hotel is good for its location or for its service. It's also unclear specifically why a reviewer liked this hotel. What we want to do is to decompose this overall rating into ratings on different aspects such as value, room, location, and service. If we can decompose the overall ratings into ratings on these different aspects, we can obtain a much more detailed understanding of the reviewers' opinions about the hotel. This would also allow us to rank hotels along different dimensions such as value or room quality.

Using this knowledge, we can better understand how the reviewers view this hotel from their own perspective. Not only do we want to infer these aspect ratings, we also want to infer the aspect *weights*. That is, some reviewers may care more about value as opposed to the service. Such a case is what's shown on the left for the weight distribution, where you can see most weight is placed on value. Clearly, different users place priority on different rating aspects. For example, imagine a

Hotel XXX

Reviewer 1: ★★★★★
"Great location + spacious room = happy traveler"
Stayed for a weekend in July. Walked everywhere,
enjoyed comfy bed and quiet hallways....

Reviewer 2: ★★★★★
"Terrific service and gorgeous facility"
I stayed at the hotel with my young daughter
for three nights June 17–20, 2010 and absolutely
loved the hotel. The room was one of the nicest
I've ever stayed in ...

How to infer aspect ratings?

★★★★   Value
★★★★★ Rooms
★★★★★ Location
★★★★   Service

★★★★   Value
★★★★★ Rooms
★★★★★ Location
★★★★   Service

How to infer aspect weights?

Value   Location   Service   ...

Value   Location   Service   ...

Figure 18.9   **Motivation of LARA.**

hotel with five stars for value. Despite this, it might still be very expensive. If a
reviewer really cares about the value of a hotel, then the five-star review would most
likely mean a competitive price. In order to interpret the ratings on different aspects
accurately, we also need to know these aspect weights. When these different aspects
are combined together with specific weights for each user, we can have a much
more detailed understanding of the overall opinion.

Thus, the task is to take these reviews and their overall ratings as input and
generate both the aspect ratings and aspect weights as output. This is called Latent
Aspect Rating Analysis (LARA).

More specifically, we are given a set of review articles about a topic with overall
ratings, and we hope to generate three things. One is the *major aspects* commented
on in the reviews. Second is *ratings on each aspect*, such as value and room service.
Third is the *relative weights placed on different aspects* by each reviewer. This task
has many applications. For example, we can do opinion-based entity ranking or
we can generate an aspect-level opinion summary. We can also analyze reviewers'
preferences, compare them, or compare their preferences on different hotels. All
this enables personalized product recommendation.

As in other cases of these advanced topics, we won't cover the technique in detail.
Instead, we will present a basic introduction to the technique developed for this
problem, as shown in Figure 18.10. First, we will talk about how to solve the problem
in two stages. Later, we mention that we can do this in a unified model.
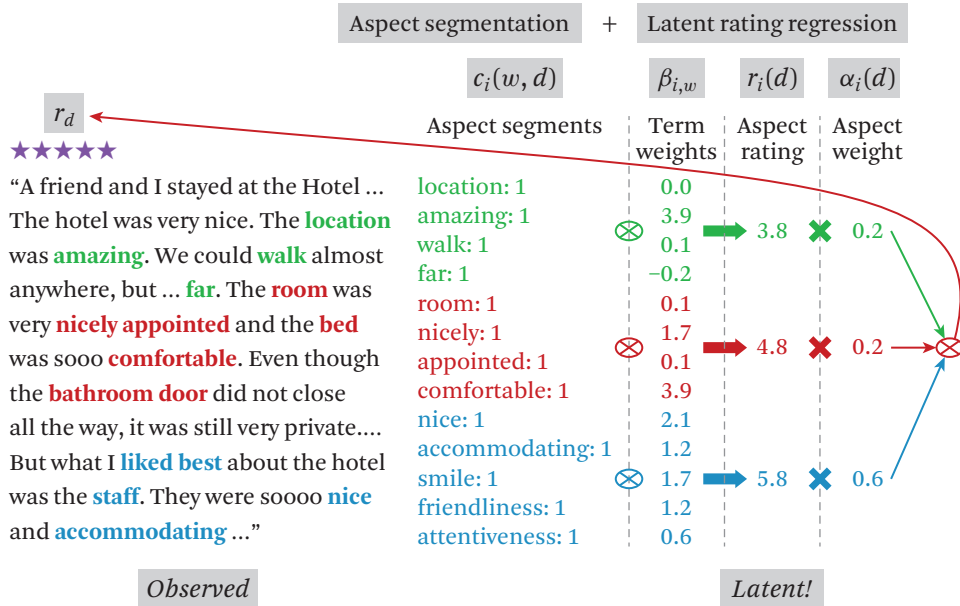
Figure 18.10   **A two-step approach to solving the LARA problem.** (Courtesy of Hongning Wang)

As input, we are given a review with the overall rating. First, we will segment the aspects; we're going to pick out what words are talking about location, what words are talking about room condition, and so on. In particular, we will obtain the counts of all the words in each segment, denoted by $c_i(w, d)$, where i is a particular segment index. This can be done by using seed words like *location*, *room*, or *price* to retrieve the aspect label of each segment. From those segments, we can further mine correlated words with these seed words, which allows us to segment the text into partitions discussing different aspects. Later, we will see that we can also use unsupervised models to do the segmentation.

In the second stage, Latent Rating Regression, we're going to use these words and their frequencies in different aspects to predict the overall rating. This prediction happens in two stages. In the first stage, we're going to use the weights of these words in each aspect to predict the aspect rating. For example, if in the discussion of location, you see a word like *amazing* mentioned many times, it will have a high weight (in the figure it's given a weight of 3.9). This high weight increases the aspect rating for location. In the case of another word like *far*, which is mentioned many times, the weight will decrease. The aspect ratings assume that it will be a weighted combination of these word frequencies where the weights are the senti-

ment weights of the words. Of course, these sentiment weights might be different for different aspects. For each aspect i we have a set of term sentiment weights for word w denoted as $\beta_{i,w}$.

In the second stage, we assume that the overall rating is simply a weighted combination of these aspect ratings. We assume we have aspect weights $\alpha_i(d)$, and these will be used to take a weighted average of the aspect ratings $r_i(d)$. This method assumes the overall rating is simply a weighted average of these aspect ratings, which allows us to predict the overall rating based on the observable word frequencies.

On the left side of Figure 18.10 is all the observed information, r_d (the overall rating) and $c_i(w, d)$. On the right side is all the latent (hidden) information that we hope to discover. This is a typical case of a generative model where we embed the interesting latent variables. Then, we set up a generative probability for the overall rating given the observed words. We can adjust these parameter values to maximize the conditional probability of the observed rating given the document. We have seen such cases before in other models such as PLSA, where we predict topics in text data. Here, we're predicting the aspect ratings and other parameters.

More formally, the data we are modeling here is a set of review documents with overall ratings, as shown in Figure 18.11. Each review document is denoted as d and the overall ratings denoted by r_d. We use $c_i(w, d)$ to denote the count of word w in aspect segment i. The model is going to predict the rating based on d, so we're interested in the rating regression problem of $p(r_d \mid d)$. This model is set

- Data: a set of review documents with overall ratings: $C = \{(d, r_d)\}$
  - d is pre-segmented into k aspect segments
  - $c_i(w, d)$ = count of word w in aspect segment i (zero if w didn't occur)
- Model: predict rating based on d: $p(r_d|d)$

Overall rating = weighted average of aspect ratings

Multivariate Gaussian prior

$$r_d \sim N(\textstyle\sum_{i=1}^{k}\alpha_i(d)r_i(d), \underline{\delta^2}),$$

$$\vec{\alpha}(d) \sim N(\underline{\vec{\mu}}, \Sigma)$$

$$r_i(d) = \textstyle\sum_{w \in V} c_i(w, d)\underline{\beta_{i,w}}$$

$$\beta_{i,w} \in \Re$$

Aspect-specific sentiment of w

Aspect rating = sum of sentiment weights of words in the aspect
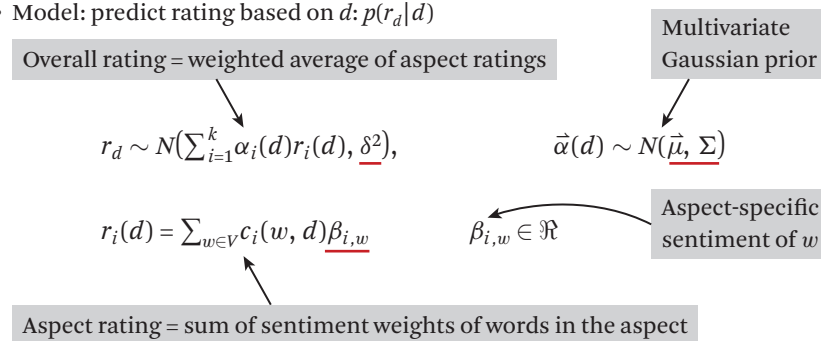
Figure 18.11   Latent rating regression.

up as follows. r_d is assumed to follow a normal distribution with a mean that is a weighted average of the aspect ratings $r_i(d)$ and variance δ^2. Of course, this is just our assumption. As always, when we make this assumption, we have a formal way to model the problem and that allows us to compute the interesting quantities—in this case, the aspect ratings and the aspect weights.

Each aspect rating $r_i(d)$ is assumed to be a sum of sentiment weights of words in aspect i. The vector of weights α for the aspects in the overall rating is it-self drawn from another multivariate Gaussian distribution, $\alpha(d) \sim \mathcal{N}(\mu, \Sigma)$. This means when we generate our overall rating, we're going to first draw a set of α values from this multivariate Gaussian prior distribution. Once we get these α values, we're going to compute the weighted average of aspect ratings as the mean of the normal distribution to generate the overall rating r_d. Note that β is indexed by both i and w. That gives us a way to model different aspect segments of the same word, since the same word might have positive sentiment for one aspect and negative for another.

How can we estimate all these parameters? Let's collectively denote them as $\Lambda = (\beta_{i,w}, \mu, \Sigma, \delta^2)$. As usual, we can use the maximum likelihood estimate which yields parameters that maximize observed ratings conditioned on their respective reviews, as shown in Figure 18.12.

Once we estimate the parameters, we can easily compute the aspect rating for a particular aspect by taking all counts of the words that occurred in segment i and multiplying them by $\beta_{i,w}$, summing over all words. The sum would be zero for words that do not occur, so we can simply take the sum of all the words in the vocabulary.

- Maximum likelihood estimate
  - Parameters: $\Lambda = (\{\beta_{i,w}\}, \vec{\mu}, \Sigma, \delta^2)$
  - ML estimate: $\Lambda^* = \text{argmax}_\Lambda \prod_{d \in C} p(r_d | d, \Lambda)$
- Aspect rating for aspect i

$$r_i(d) = \sum_{w \in V} c_i(w, d)\beta_{i,w}$$

  $c_i(w, d) = 0$ for words not occurring in aspect segment i

- Aspect weights: $\alpha_i(d) =$ weight on aspect i

$$\vec{\alpha}(d)^* = \text{argmax}_{\vec{\alpha}(d)} p(\vec{\alpha}(d) | \mu, \Sigma) p(r_d | d, \{\beta_{i,w}\}, \delta^2, \vec{\alpha}(d))$$

Maximum a posteriori   Prior   Likelihood
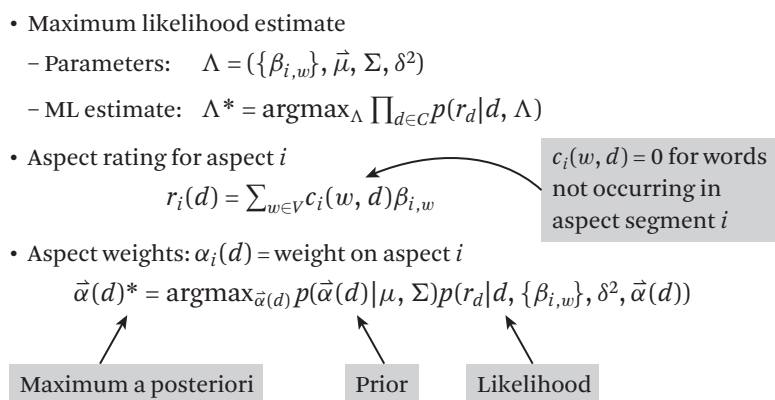
Figure 18.12   **Latent rating regression estimation.**

To compute the $\alpha_i(d)$ values, we must use maximum a posteriori. This means that we maximize the product of the prior of α (according to our assumed multivariate Gaussian distribution) and the likelihood of the rating r_d:

$$\alpha(d)^* = \arg\max\nolimits_{\alpha(d)} p(\alpha(d) \mid \mu, \Sigma) p(r_d \mid d, \beta_{i,w}, \delta^2, \alpha(d)). \tag{18.3}$$

The likelihood rating is the probability of generating this observed overall rating given this particular α value and some other parameters. For more details about this model, we refer the reader to Wang et al. [2010].

Earlier, we talked about how to solve the LARA problem in two stages. First, we did segmentation of different aspects, and then used a latent regression model to learn the aspect ratings and weights. It's also possible to develop a unified generative model for solving this problem. That is, we not only model the generation of overall ratings based on text, but also model the generation of the text itself. A natural solution would be to use a topic model.

Given an entity, we can assume there are aspects that are described by word distributions (i.e., topics). We then use a topic model to model the generation of the reviewed text. We assume words in the review text are drawn from these distributions in the same way as we assumed in PLSA. Then, we can plug in the latent regression model to use the text to further predict the overall rating. To predict the overall rating based on the generated text, we first predict the aspect rating and then combine them with aspect weights to predict the overall rating. This gives us a unified generative model, where we model both the generation of text and the overall rating conditioned on text. We don't have space to discuss this model in detail, so we refer reader to Wang et al. [2011] for additional reading.

Let's look at some applications enabled by using these kinds of generative models. First, consider the decomposed ratings for some hotels that have the same overall rating. If you just look at the overall rating, you can't really tell much difference between these hotels, but by decomposing these ratings into aspect ratings we can see some hotels have higher ratings for some dimension (like value) while others might score better in other dimensions (like location). This breakdown can give us detailed opinions at the aspect level.

Another application is that you can compare different reviews on the same hotel. At a high level, overall ratings may look the same, but after decomposing the ratings, you might see that they have high scores on different dimensions. This is because the model can discern differences in opinions of different reviewers. Such a detailed understanding can help us learn about the reviewers and better incorporate their feedback.

Value	Rooms	Location	Cleanliness
resort 22.80	view 28.05	restaurant 24.47	clean 55.35
value 19.64	comfortable 23.15	walk 18.89	smell 14.38
excellent 19.54	modern 15.82	bus 14.32	linen 14.25
worth 19.20	quiet 15.37	beach 14.11	maintain 13.51
bad −24.09	carpet −9.88	wall −11.70	smelly −0.53
money −11.02	smell −8.83	bad −5.40	urine −0.43
terrible −10.01	dirty −7.85	road −2.90	filthy −0.42
overprice −9.06	stain −5.85	website −1.67	dingy −0.38

Figure 18.13 **Sentiment lexicon learned by LARA.** (Based on results from Wang et al. [2010])

In Figure 18.13, we show some highly weighted words and the negatively weighted words for each of the four aspect dimensions: value, room, location, and cleanliness. Thus, we can also learn sentiment information directly from the data. This kind of lexicon is very useful because in general, a word like *long* may have different sentiment polarities for different contexts. If we see "The battery life of this laptop is long," then that's positive. But if we see "The rebooting time for the laptop is long," then that's clearly not good. Even for reviews about the same product (i.e., a laptop) the word *long* is ambiguous. However, with this kind of lexicon, we can learn whether a word is positive or negative for a particular aspect. Such a lexicon can be directly used to tag other reviews about hotels or tag comments about hotels in social media.

Since this is almost completely unsupervised aside from the overall ratings, this can allow us to learn from a potentially larger amount of data on the internet to create a topic-specific sentiment lexicon. Recall that the model can infer whether a reviewer cares more about service or the price. How do we know whether the inferred weights are correct? This poses a very difficult challenge for evaluation.

Figure 18.14 shows prices of hotels in different cities. These are the prices of hotels that are favored by different groups of reviewers. Here we show the ratio of importance of value to other aspects. For example, we have value vs. location. In the figure, "top ten" refers to the reviewers that have the highest ratios by a particular measure. This means these top ten reviewers tend to put a lot of weight on value as compared with other dimensions. The bottom ten refers to reviewers that have put higher weights on other aspects than value; these are people who care about

Top-10: Reviewers with the highest Val/X ratio (emphasize "value")
Bot-10: Reviewers with the lowest Val/X ratio (emphasize a non-value aspect)

City	Avg. price	Group	Val/Loc	Val/Rm	Val/Ser
Amsterdam	241.6	top-10	190.7	214.9	221.1
		bot-10	**270.8**	**333.9**	**236.2**
San Francisco	261.3	top-10	214.5	249.0	225.3
		bot-10	**321.1**	**311.1**	**311.4**
Florence	272.1	top-10	269.4	248.9	220.3
		bot-10	**298.9**	**293.4**	**292.6**

Higher!
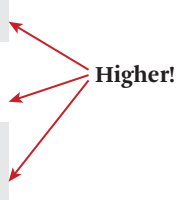
Figure 18.14   **Sample results showing inferred weights are meaningful.** (Based on results from Wang et al. [2010])

another dimension and don't care so much about value, at least compared to the top ten group.

These ratios are computed based on the inferred weights from the model. We can see the average prices of hotels favored by the top ten reviewers are indeed much cheaper than those that are favored by the bottom ten. This provides some indirect way of validating the inferred weights. Looking at the average price in these three cities, you can actually see the top ten group tends to have below average prices, whereas the bottom half (that cares about aspects like service or room condition) tend to have hotels that have higher prices than average.

With these results, we can build many interesting applications. For example, a direct application would be to generate a collective summary for each aspect, including the positive sand negative sentences about each aspect. This is more informative than the original review that just has an overall rating and review text.

Figure 18.15 shows some interesting results on analyzing user rating behavior. What you see is average weights along different dimensions by different groups of reviewers. On the left side you see the weights of viewers that like the expensive hotels. They gave the expensive hotels five stars, with heavy aspect weight on service. That suggests that people like expensive hotels because of good service, which is not surprising. This is another way to validate the model by the inferred weights.

The five-star ratings on the right side correspond to the reviewers that like the cheaper hotels. As expected, they put more weight on value. If you look at when they didn't like cheaper hotels, you'll see that they tended to have more weights on the condition of the room cleanliness. This shows that by using this model, we can

	Expensive Hotel		Cheap Hotel	
	5 Stars	3 Stars	5 Stars	3 Stars
Value	0.134	0.148	(0.171)	0.093
Room	0.098	0.162	0.126	0.121
Location	0.171	0.074	0.161	0.082
Cleanliness	0.081	**0.163**	0.116	**0.294**
Service	(0.251)	0.101	0.101	0.049

People like expensive hotels because of good service.

People like cheap hotels because of good value.

Figure 18.15 **Analysis of reviewer preferences.** (Based on results from Wang et al. [2010])

infer some information that's very hard to obtain even if you read all the reviews. This is a case where text mining algorithms can go beyond what humans can do, to discover interesting patterns in the data. We can compare different hotels by comparing the opinions from different consumer groups in different locations. Of course, the model is quite general, so it can be applied to any reviews with an overall ratings.

Finally, the results of applying this model for personalized ranking or recommendation of entities are shown in Figure 18.16. Because we can infer the reviewers' weights on different dimensions, we can allow a user to indicate what they actually care about. For example, we have a query here that shows 90% of the weight should be on value and 10% on others. That is, this user just cares about getting a cheap hotel—an emphasis on the value dimension. With this model, we can find the reviewers whose weights are similar to the query user's. Then, we can use those reviewers to recommend hotels; this is what we call personalized or query specific recommendation. The non-personalized recommendations are shown on the top, and you can see the top results generally have much higher price than the bottom group. That's because the reviewers on the bottom cared more about the value. This shows that by doing text mining we can better understand and serve the users.

To summarize our discussion, sentiment analysis is an important topic with many applications. Text sentiment analysis can be readily done by using just text categorization, but standard techniques tend to be insufficient so we need to have an enriched feature representation. We also need to consider the order of the sentiment categories if there are more than two; this is where our discussion on ordinal regression comes into play. We have also shown that generative models are pow-

Query: 0.9 value 0.2 others		Hotel	Overall Rating	Price	Location
	Approach 2	Majestic Colonial	5.0	339	Punta Cana
Non-personalized		Agua Resort	5.0	753	Punta Cana
		Majestic Elegance	5.0	537	Punta Cana
		Grand Palladium	5.0	277	Punta Cana
		Iberostar	5.0	157	Punta Cana
	Approach 1	Elan Hotel Modern	5.0	216	Los Angeles
Personalized		Marriott San Juan Resort	4.0	354	San Juan
		Punta Cana Club	5.0	409	Punta Cana
(Query-specific)		Comfort Inn	5.0	155	Boston
		Hotel Commonwealth	4.5	313	Boston

Figure 18.16   **Personalized entity ranking.** (Based on results from Wang et al. [2010])

erful for mining latent user preferences, in particular the generative model for mining latent rating regression. Here, we embedded some interesting preference information and sentiment by weighting words in the model. For product reviews, the opinion holder and the opinion target are clear, making them easy to analyze. There, of course, we have many practical applications. Opinion mining from news and social media is also important, but that's more difficult than analyzing review data mainly because the opinion holders and opinion targets are not clearly defined. This challenge calls for more advanced natural language processing.

For future reading on topics presented in this chapter, we suggest [Pang and Lee 2008], a comprehensive survey on opinion mining and sentiment analysis.

18.4   Evaluation of Opinion Mining and Sentiment Analysis

In this chapter we investigated opinion mining and sentiment analysis from the viewpoint of both classification (or regression) and topic analysis. Thus, evaluation from these perspectives will be similar to those discussed in Chapter 15 (categorization) and Chapter 17 (topic analysis).

From a classification viewpoint, we can use a dataset with documents labeled as positive or negative, or as ratings on a numerical scale as described in this chapter. This then becomes the standard machine learning testing setup where we can use techniques such as cross-fold validation to determine the effectiveness of our method. Additionally, feature selection can show which features are the most

useful for determining whether a sentence (e.g.) is positive or negative. Based on the useful features, we can either adjust the algorithm or try to fine-tune the feature set.

From a topic analysis viewpoint, we would like to ensure that the topics are coherent and have a believable coverage and distribution over the documents in the dataset. We mentioned that corpus log likelihood is a way to test how well the model fits to the data. While this evaluation metric doesn't always agree with human judges [Chang et al. 2009], it does serve as a sanity check or proxy for true usefulness.

Additionally, we can test the effectiveness of adding opinion mining and sentiment analysis to an existing system by the method discussed in Chapter 13. That is, we can compare the system's performance before sentiment analysis, and then compare its performance afterwards. If the system performs statistically significantly better under some evaluation metric relevant to the task at hand, then the sentiment analysis is a definitive improvement.

Bibliographic Notes and Further Reading

Opinion mining and sentiment analysis have been extensively studied. Two excellent books on this general topic are the book Opinion Mining and Sentiment Analysis [Pang and Lee 2008] and the book Sentiment Analysis and Opinion Mining [Liu 2012]. Multiple extensions of topic models for analyzing opinionated topics have been made (e.g., topic-sentiment mixture model [Mei et al. 2007a], multi-grain topic model [Titov and McDonald 2008], and aspect and sentiment unification model [Jo and Oh 2011]). Techniques for Latent Aspect Rating Analysis are mainly covered in two KDD papers [Wang et al. 2010], [Wang et al. 2011].

Exercises

18.1. In this chapter, we mainly discussed how to determine overall sentiment for a text object. Imagine that we already have the sentiment information as part of the object and we are instead interested in identifying the *target* of the sentiment. Brainstorm some ideas using NLP techniques mentioned in this book.

18.2. Meta has an implementation of the LDA topic modeling algorithm, which contains symmetric priors for the word and topic distributions. Modify Meta to contain non-uniform priors on the topic-word distributions to encode some additional knowledge into the prior. For example, in the sentiment analysis task, create two prior distributions that have high weights on particular "good" and "bad" topic distributions. That is, the prior for the "good" topic should probably weight the

term *excellent* relatively high and the "bad" topic should weight the term *broken* relatively high.

18.3. A simple version of sentiment analysis is called *word valence scoring*. A finite list of words has a positive or negative valence score. When determining the sentiment for a sentence, the valence scores are added up for each word in the sentence; if the result is positive, the sentiment is positive. If the result is negative, the sentiment is negative. For example, some valences could be bad $= -4.5$, awesome $=$ 6.3, acceptable $= 1.1$. What is a potential weakness to this method?

18.4. How can we automatically create the sentiment word valence scores based on a list of sentences labeled as positive or negative?

18.5. The techniques discussed in this chapter can be applied to other problems aside from sentiment analysis in particular. Name some applications that also would benefit from the general methods discussed in this chapter. Explain how you would implement them.

18.6. Cross-fold validation is a useful way to evaluate classifiers as mentioned in Chapter 15. How can cross-fold validation be used to detect potential overfitting?

18.7. We mentioned that a main challenge in designing features is to optimize the tradeoff between exhaustivity and specificity. Can you design an experimental training setup that takes these variables into account?

18.8. In LARA, why do you imagine the vector of weights α is drawn from a multivariate Gaussian distribution? That is, why not use some other distribution?

18.9. In LARA, each word is assigned an aspect, but this means that one sentence may be assigned many different aspects. Can you outline an adjustment to ensure that each sentence only covers one topic or a small number of topics? Similarly, can we ensure that sequences of words all belong to the same aspect?

18.10. Instead of a classification task, it may be beneficial to instead rank text objects by how positive they are (so low-ranking documents are very negative). Outline a few methods on how this may be achieved. What are the benefits and downsides compared to classification?

18.11. Imagine that you have an unlabeled dataset of product reviews. How can you design a sentiment classifier based on this dataset without manually labeling all of the documents?

19 Joint Analysis of Text and Structured Data

In this chapter, we discuss techniques for joint analysis of text and structured data, which not only enriches text analysis, but is often necessary for many prediction problems involving both structured data and unstructured text data. Due to the complexity of many of these methods and the limited space, we will only give a brief introduction to the main ideas and show sample results to provide a sense about what kind of applications these techniques can support. Details of these techniques can be found in the references provided at the end of this chapter.

19.1 Introduction

In real-world big data applications, we would have both structured data and unstructured text data available to help us make predictions and support decision making. It is thus important to analyze both kinds of data jointly, especially when the goal is to predict some latent real-world variable that is not directly observed in the data. To illustrate this problem setup and the big picture of data mining in general, we show the **data mining loop** in Figure 19.1.

In this figure, we see that there are multiple sensors—including human sensors—to report what we have seen in the real world in the form of data. The data include both non-text data and text data. Our goal is to see if we can predict some values of important real world variables that matter to us. For example, we might be interested in predicting the condition of a bridge, the weather, stock prices, or presidential election results. We are interested in predicting such variables because we might want to act on the inferred values or make decisions based on the inferred values.

How can we get from the data to these predicted values? We'll first have to do data mining and analysis of the data. In general, we should try to leverage all the data that we can collect, and joint mining of non-text and text data is critical. Through
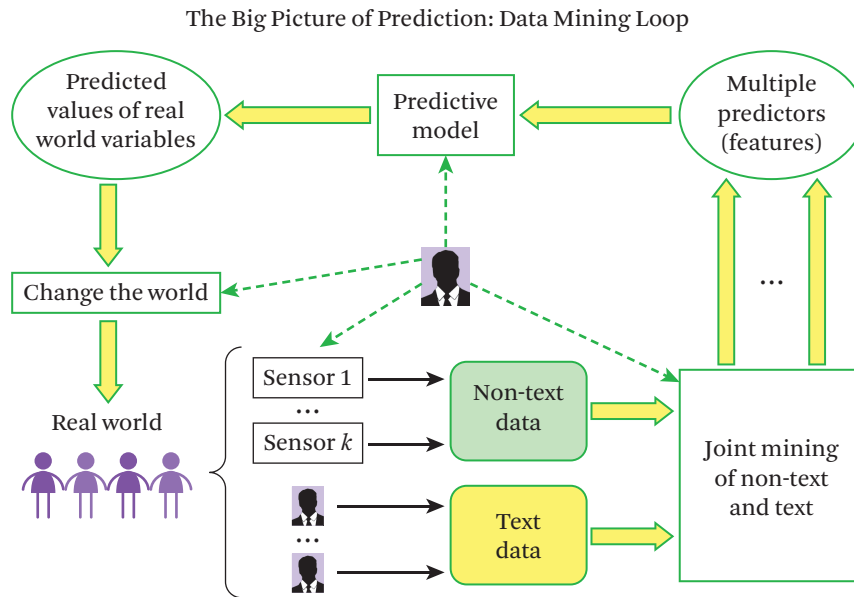
Figure 19.1   **The data mining loop.**

analysis of all the data, we can generate multiple predictors of the interesting variables to us. We call these predictors features, and they can further be combined and put into a predictive model to actually predict the value of any interesting variable. The prediction results would then allow us to act and change the world. This is the general process for making a prediction based on any data.

It's important to emphasize that a human actually plays a very important role in this process, especially because of the involvement of text data. First, a human user would be involved in the mining of the data. The user can control the generation of these features and can even manually create features. Second, humans can help understand the text data, because text data are created to be consumed by humans, and humans can consume and interpret text data much more effectively than a machine. The challenge, of course, is when there is an enormous amount of text data, since it would not be feasible for humans to read and digest all the information. Thus, machines must help and that's why we need to do text data mining. Sometimes machines can even "see" patterns in data that humans may not see even if they have the time to read all the data.

Next, humans also must be involved in building, adjusting and testing a predictive model. In particular, we humans will have important domain knowledge about

the prediction problem that we can build into the predictive model. Once we have the predicted values for the variables, humans would be involved in taking actions to change the world or make decisions based on these particular values.

Finally, it's interesting that a human could be involved in controlling the sensors to collect the *most useful data* for prediction. Thus, this forms a data mining loop because as we perturb the sensors, they will collect additional new and potentially more useful data, allowing us to improve the prediction. In this loop, humans will recognize what additional data will need to be collected. Machines can help humans identify what data should be collected next. In general, we want to collect data that is most useful for learning. The study of how to identify data points that would be most helpful for machine learning is often referred to as active learning, which is an important subarea in machine learning.

There is a loop from data acquisition to data analysis; from data mining to prediction of values; from actions to change the world; and finally, we observe what happens. We can then decide what additional data have to be collected and adjust the sensors accordingly. Analysis of the prediction errors can help reveal what additional data we need to acquire in order to improve the accuracy of prediction. This big picture is actually very general and can serve as a model for many important applications of big data.

Since the focus of the book is on text data, it is useful to consider the special case of the loop shown in Figure 19.2 where the goal is to use text data to infer values of some other variables in the real world that may not be directly related to the text. Such an analysis task is different from a task such as topic mining where the goal is to directly characterize the content of text. In text-based prediction, our goal can be to infer any information about the world. This is, of course, only possible if there exist clues in the text data about the target variable; fortunately, this is often the case since people report everything in text data. In many cases (e.g., stock price prediction), the non-text data (historical stock prices) are much more effective for prediction than text data, though text data can generally help provide additional indicators for the prediction task.

Sometimes text data contain more useful indicators than non-text data, and text data alone may also be sufficient for making predictions. Typically, in such a case, the prediction is about human behavior or human preferences or opinions. In general though, text data will be put together with non-text data.

In all cases of text-based prediction, there are two important questions. First, what features (indicators) are most useful for the prediction task? Second, how can we generate such effective indicators from text? For convenience, we will use the term "feature" and "indicator" interchangeably. The first question has much to do
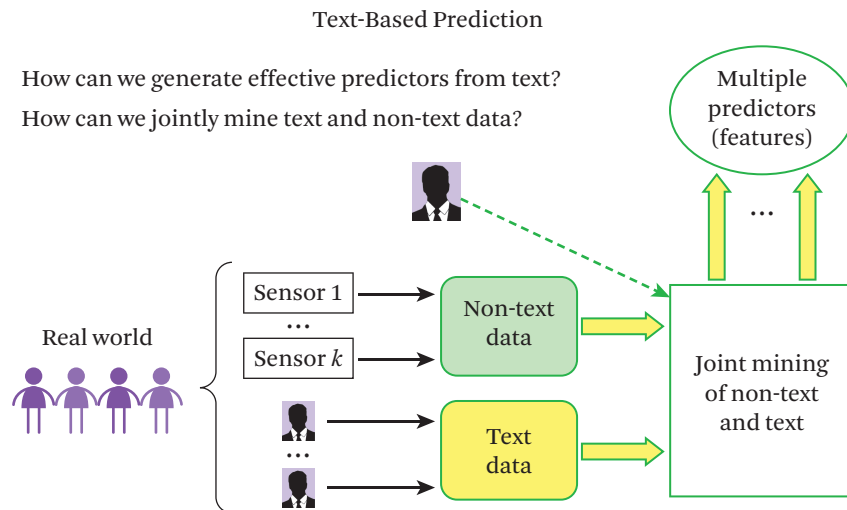
Figure 19.2   **Illustration of text-based prediction.**

with the specific prediction problem, and is thus inevitably application-specific. However, there are some generic features we can often start with (like *n*-grams or topics) as discussed in some previous chapters. Supervised learning methods can be used to learn what features are most effective. The second question has been addressed to some extent in the previous chapters of the book since many techniques that we have introduced can be potentially used to obtain features from text data. For example, topic mining can be very useful to generate topic-based indicators or predictors that can be further fed into a predictive model. Such topic-based features can be mixed with word-based features to enrich the feature representation. Sizes of a certain cluster of terms or cluster of documents may also be potential features. A set of terms with paradigmatic relations may be a better indicator than any single term, and sentiment tags that we may be able to generate based on text data are yet another kind of useful feature for some prediction problems.

What has not been discussed is how we can jointly mine text and non-text data together to discover interesting knowledge that could not be otherwise discovered by using either one alone. This interaction is the topic of the chapter.

The benefit of joint analysis of text and non-text data can be seen from two differ-ent perspectives. First, non-text data can enrich text analysis. Specifically, non-text data can often provide a context for mining text data and thus enable us to partition

data in different ways based on the companion non-text data (e.g., partitioning text based on time or location). This opens up possibilities of *contextual text mining*, or mining text data in the context defined by non-text data to discover context-specific knowledge (such as topics associated with a specific non-text variable such as time), or patterns across different contexts like temporal trends. Second, text data can help interpret patterns discovered from non-text data. For example, if a frequent pattern is discovered from non-text data, we can separate the text data associated with the data instances where the pattern occurs from those associated with the instances that do not match the pattern. We can then analyze the difference between these two sets of text data, which may be associated with the meaning of the pattern, and thus can offer insights about how to interpret the pattern which would otherwise be hard to interpret by only looking at the non-text data. This technique is called pattern annotation and discussed in detail in Mei et al. [2006].

19.2   Contextual Text Mining

In this section, we discuss how to use non-text data as context to enrich topic analysis of text data. Such analysis techniques can be regarded as an extension of topic analysis to further reveal the correlation of topics and any associated context (e.g., time or location). When topics represent opinions, we may also reveal context-dependent opinions. **Contextual text mining** can be very useful for text-based prediction because it allows us to combine non-text data with text data to derive potentially very effective sophisticated predictors.

Contextual text mining is generally useful because text often has rich contextual information. First, text data almost always have metadata available (such as time, location, author, and source of the data), which can be regarded as direct context information. Second, text data may also have indirect context which refers to additional data related to the metadata. For example, from the authors of a text article, we can further obtain additional context such as the social network of the author, the author's age, or the author's location. Such information is not in general directly related to the text, yet through such a propagation process, we can connect all of them. There may also be other articles from the same source as a current article, and we can connect these articles from the same source and make them related as well. In general, any related data can be regarded as context.

What can the context of text data be used for? Context can be used to partition text data in many interesting ways. It can almost allow us to partition text data in any way that we want, and thus enables comparative analysis of text to be done across any context dimension that is interesting to us.

Context = Partitioning of Text

Enable discovery of knowledge associated with different context as needed
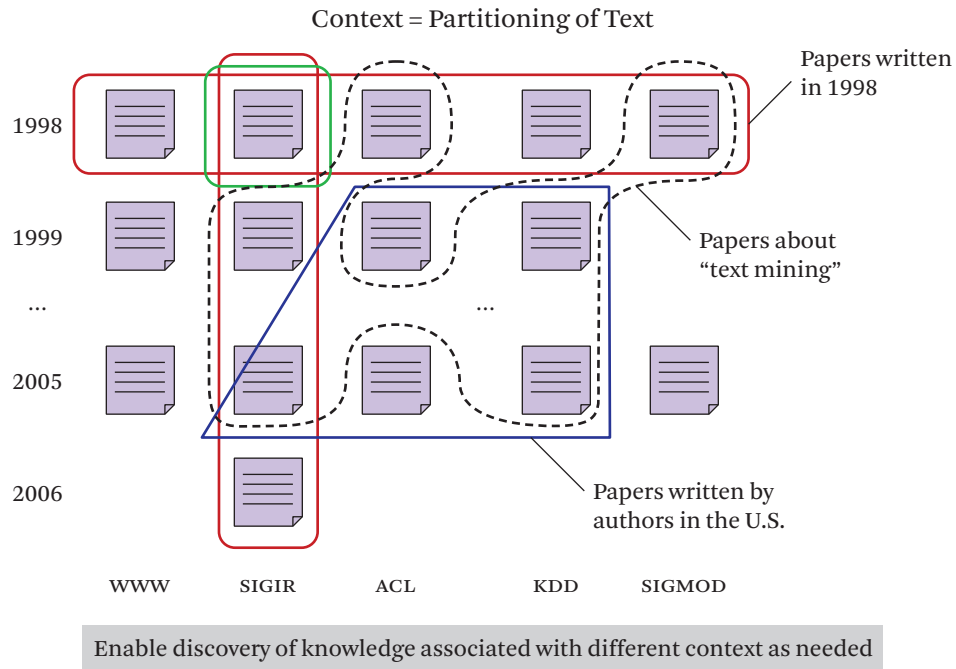
Figure 19.3   **Context leads to partitioning of text data.**

As a specific example, we show how some context variables enable partitioning of research papers in many different ways in Figure 19.3. The horizontal dimension of the figure shows different conference venues where the papers are published, and the vertical dimension shows the time of a publication. We can treat each paper as a separate unit; in this case, a paper ID serves as the context and each paper has its own context. We can treat all the papers published in 1998 as one group, and partition papers by the year due to the availability of time as a non-text variable. Such a partitioning would allow us to compare topics in different years. Similarly, we can partition the data based on the venues; we can group all the papers published in SIGIR and compare them with those published in KDD or ACL. This comparison is enabled by the availability of the non-text variable of the conference venue. Furthermore, we can also partition the data to obtain the papers written by authors in the U.S. by using additional context of the authors. Such a contextual view of the data would allow us to compare papers written by American authors with those written by authors in other countries.

Sometimes, we can use topics to partition the data without involving non-text data. For example, we can obtain a set of papers about the topic "text mining," and compare them with the papers about another topic. Note that these partitions can be intersected with each other to generate even more complicated partitions. So, we may form constraints on non-text variables to create interesting contexts for partitioning text data, which can then facilitate discovery of knowledge associated with different contexts.

The incorporation of non-text contextual variables enables the association of topics from text data with potentially many different contexts, generating interesting and useful patterns. For example, in comparing topics over time, we can see topical trends. Comparing topics in different contexts can also reveal differences about the two contexts.

There are many interesting questions that require contextual text mining to answer. For example, to answer a question such as "What topics have been getting increasing attention recently in data mining research?" we would need to analyze text in the context of time. Is there any difference in the responses of people in different regions to an event? To answer such a question, location can be the context. What are the common research interests of two researchers? In this case, authors can be the context. Is there any difference in the research topics published by authors in the U.S. and those outside? Here, the context would include the authors and their affiliation and location. This is a case where we need to go beyond just the authors and further look at the additional information connected to the author.

Is there any difference in the opinions of all the topics, expressed in one social network compared to another? In this case, the social network of authors and the topic can be a context. Are there topics in news data whose coverage is correlated with sudden changes in stock prices? Such a question can be addressed by using a time series such as stock prices as context. What issues mattered in the 2012 presidential campaign and election? Here time serves again as context. Clearly, contextual text mining can have many applications.

19.3 Contextual Probabilistic Latent Semantic Analysis

In this section, we briefly introduce a specific technique for contextual text mining called **Contextual Probabilistic Latent Semantic Analysis** (CPLSA). CPLSA is an extension of PLSA to incorporate context variables into a generative model so that both the selection of topics and the topic word distributions can depend on the context associated with text.

Recall that in PLSA the text data are generated by first selecting a topic and then generating a word from a topic. The topics are shared by all the documents in the collection, but the selection probability (i.e., coverage of topics) is specific to a document. In CPLSA, the generation process is similar, but since we assume that we have context information (time or location) about a document, the generation of words in the document may be conditioned on the specific context of the document. Instead of assuming just one set of common topics for the collection, we assume that there may be variations of this set of topics depending on the context. For example, we might have a particular view of all the topics imposed by a particular context (such as a particular time period or a particular location), so we may have multiple sets of comparable topics that represent different views of these topics associated with different contexts.

In Figure 19.4, we use a collection of blog articles about Hurricane Katrina to illustrate this idea. In such a collection, we can imagine potential topics such as government response, donation, and flooding of New Orleans. These are shown as different "themes," each represented by a word distribution. Besides these themes, we also show three potentially different views of these three themes (topics): `View1` is associated with a location context (Texas) and contains Texas-specific word distributions for all the three themes shown in the figure, which may reflect how the authors in Texas talk about these topics, which presumably would be different from how the authors in Illinois talk about them, which can be represented as a different view. Similarly, `View2` is associated with a time context (July 2005), and `View3` is associated with a context of author occupation (a sociologist).

The selection of topics when generating words in a document can also be influenced by the context of the document. For example, the authors in Texas may tend to cover one particular aspect more than another, while the authors in other locations may be different. Similarly, contexts such as the time and author occupation may also suggest a preference for certain topics. In the standard PLSA, we assume that every document has its own preference for topics; this can be regarded as a special case of CPLSA where we have taken each document ID as a context for the document. The different topic selection preferences of different contexts are illustrated at the bottom of Figure 19.4.

To generate a word in document d, we would first choose a particular view of all the topics based on the context of the document. For example, the view being chosen may be the location (Texas); in such a case, we would be using the Texas-specific topics to generate the word. The next step is to decide a topic coverage, which again depends on the context. We may choose the coverage associated with the time (July 2005). Note that this same coverage would be used when generating
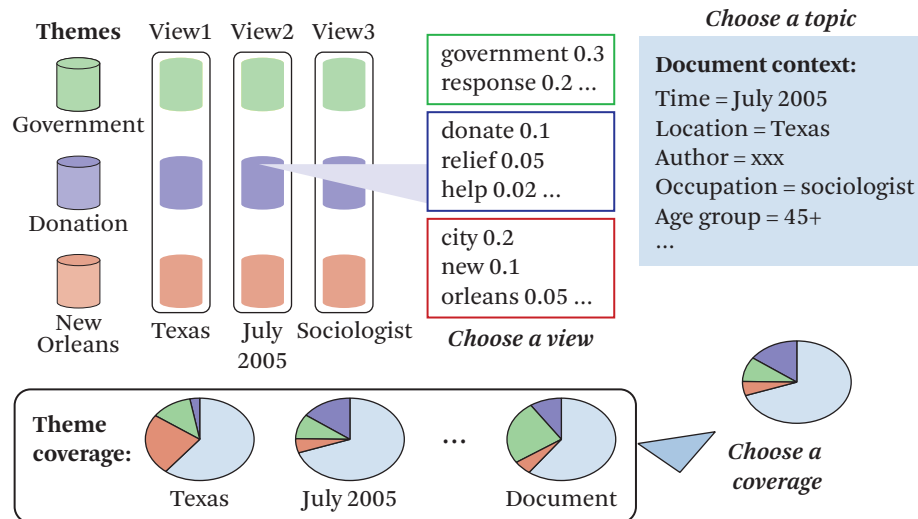
Figure 19.4 **CPLSA as a generative model depending on context.** (Courtesy of Qiaozhu Mei)

all documents with July 2005 as the time context. This is very different from the PLSA where each document has its own coverage preference, which is independent of each other. The dependency (on context) introduced here enables all the text with a particular context associated to contribute to the learning of the topic coverage.

Once a view of topics and a topic coverage distribution have been chosen, the rest of the generation process is exactly the same as in PLSA. That is, we will use the topic coverage and selection distribution to sample a topic, and then use the corresponding word distribution of the topic (i.e., chosen view of the topic) to generate a word. In such a generation process, each word can be potentially generated using a different view and a different topic coverage distribution depending on the contexts chosen to direct the generation process. Note that the context that determines the choice of view of a topic can be different from the context chosen to decide the topic coverage. This is illustrated in Figure 19.5 where we show that all the words in the document have now been generated by using CPLSA, which is essentially a mixture model with many component models associated with different contexts.

In CPLSA, both the views of topics and the topic coverage depend on context, so it enables discovery of different variations of the same topic in different contexts (due to the dependency of a view of topics on context) and different topic coverages in different contexts (due to the dependency of topic coverage on context).
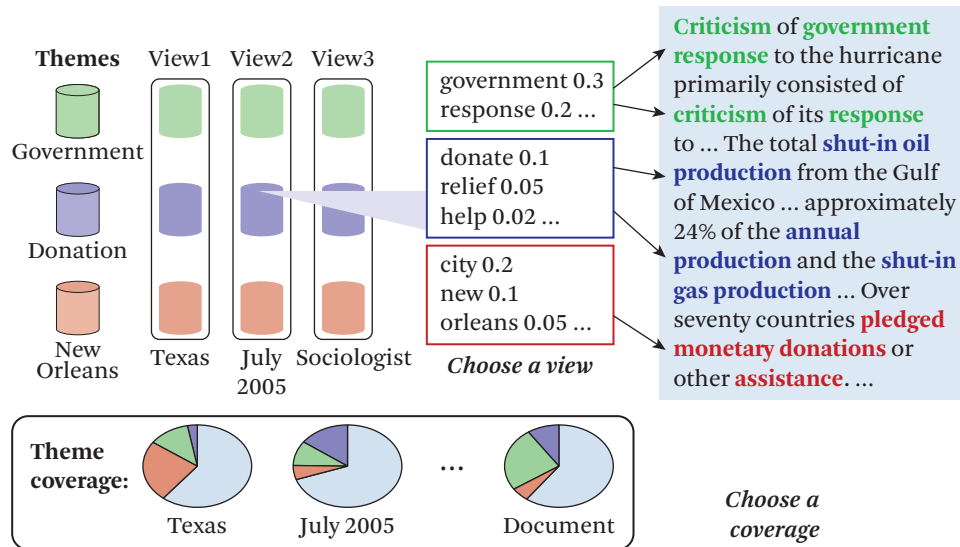
Figure 19.5 **Generation of all words in a document using CPLSA.** (Courtesy of Qiaozhu Mei)

Such context-dependent topic patterns can be very useful for answering various questions as we mentioned earlier.

The standard PLSA can easily be seen as a special case of CPLA when we have used just one single view of topics by using the whole collection as context, and used each document ID as a context for deciding topic coverage. As a result, what we can discover using PLSA is just *one single* set of topics characterizing the content in the text data with no way to reveal the difference of topics covered in different contexts. The standard PLSA can only reveal the coverage of topics in each document, but cannot discover the topic coverage associated with a particular context.

In contrast, CPLSA would provide more flexibility to embed the context variables as needed to enable discovery of multiple views of topics and context-specific coverage of topics, thus enriching the topical patterns that can be discovered.

Since CPLSA remains a mixture model, we can still use the EM algorithm to solve the problem of parameter estimation, although the number of parameters to be estimated would be significantly larger than PLSA. Thus, theoretically speaking, CPLSA can allow us to discover any context-specific topics and topic coverage distributions. However, in reality, due to the inevitable sparsity of data, we must restrict the space of the context variables to control the complexity of the model. Once estimated, the parameters of CPLSA would naturally contain context variables, including particularly many conditional probabilities of topics (given a certain context)

The common theme indicates that "United Nations" is involved in both wars

	Cluster 1	Cluster 2	Cluster 3
Common theme	united 0.042 nations 0.04 ...	killed 0.035 month 0.032 deaths 0.023 ...	...
Iraq theme	n 0.03 weapons 0.024 inspections 0.023 ...	troops 0.016 hoon 0.015 sanches 0.012 ...	...
Afghan theme	northern 0.04 alliance 0.04 kabul 0.03 taleban 0.025 aid 0.02 ...	taleban 0.026 rumsfeld 0.02 hotel 0.012 front 0.011 ...	...

Collection-specific themes indicate different roles of "United Nations" in the two wars

Figure 19.6   **Sample results of comparing two sets of news articles.** (Results were generated using the method in Zhai et al. [2004], which is a special case of CPLSA)

which are precisely what we hope to discover in contextual text mining. For details of CPLSA, readers can refer to Mei and Zhai [2006].

We now show some sample results of CPLSA. First, in Figure 19.6, we show results from comparing two sets of news articles about the Iraq and Afghanistan wars, respectively, including high probability words in both common topics and collection-specific topics. In this case, the context is the topic and leads to a partitioning of news articles into two sets corresponding to these two wars, and CPLSA degenerates to a simpler cross-collection mixture model [Zhai et al. 2004].

We have 30 articles on the Iraq war and 26 articles on the Afghanistan war. Our goal is to compare the two sets of articles to discover the common topics shared by the two sets and understand the variations of these topics in each set. The results in Figure 19.6 show that CPLSA can discover meaningful topics. The first column (Cluster 1) shows a very meaningful common topic about the United Nations on the first row, which intuitively makes sense given the topics of these two collections. Such a topic may not be surprising to people who know the background about these articles, but the result shows that CPLSA can automatically discover it.

What's more interesting, however, is the two cells of word distributions shown on the second and third rows in the first column, right below the cell about the United Nations. These two cells show the Iraq-specific view of the topic about the United Nations and the Afghanistan view of the same topic, respectively. The results show that in the Iraq War, the United Nations was more involved in weapon inspections, whereas in the Afghanistan War, it was more involved in, perhaps, aid to the Northern Alliance. These two context-specific views of the topic of the United Nations show different variations of the topic in the two wars, and reveal a more detailed understanding of topics in a context-specific way. This table is not only immediately useful for understanding the major topics and their variations in these two sets of news articles, but can also serve as entry points to facilitate browsing into very specific topics in the text collection; e.g., each cell can be made clickable to enable a user to examine the relevant discussion in the news articles in detail.

The second column shows another shared common topic about fatalities, which again should not be surprising given that these articles are about wars. It also again confirms that CPLSA is able to extract meaningful common topics. As in the case of the first column, the collection-specific topics in the third column also further show the variations of the topic in these two different contexts.

In Figure 19.7, we show the temporal trends of topics discovered from blog articles about Hurricane Katrina. The x-axis is the time, and the y-axis is the coverage of a topic. The plots are enabled directly by the parameters of CPLSA where we used time and location as context.

In Figure 19.7, we show a visualization of the trends of topics over time. The top plot shows the temporal trends of two topics. One is oil price, and one is about the flooding of the city of New Orleans. The plot is based on the conditional probability of a topic given a particular time period, which is one of the parameters in CPLSA for capturing time-dependent coverage of topics. Here we see that initially the two curves tracked each other very well. Interestingly, later, the topic of New Orleans was mentioned again but oil prices was not. This turns out to be the time period when another hurricane (Hurricane Rita) hit the region, which apparently triggered more discussion about the flooding of the city, but not the discussion of oil price.

The bottom figure shows the coverage of the topic about flooding of the city New Orleans by blog article authors in different locations (different states in the U.S.). We see that the topic was initially heavily covered by authors in the victim areas (e.g., Louisiana), but the topic was then picked up by the authors in Texas, which might be explained by the move of people from the state of Louisiana to Texas. Thus, these topical trends not only are themselves useful for revealing the topics

(a) Theme life cycles in Texas
(Hurricane Katrina)

(b) Theme "New Orleans" over states
(Hurricane Katrina)
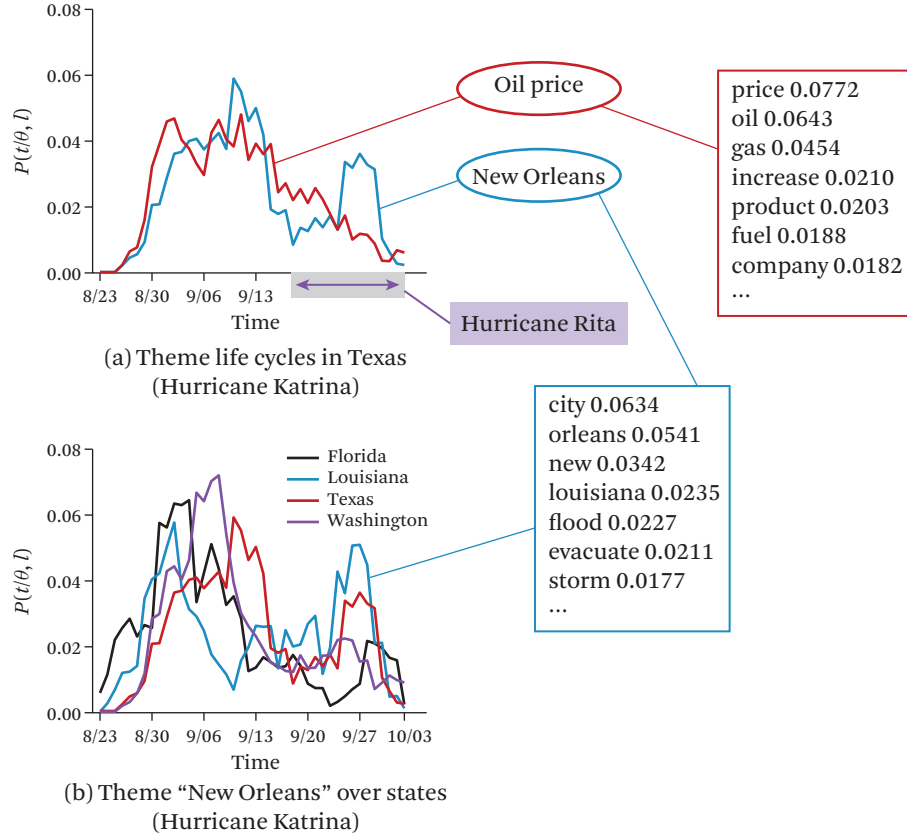
Figure 19.7 **Temporal trends of topics discovered from blog articles about Hurricane Katrina.** (Based on results from Mei et al. [2006a])

and their dynamics over time, but also enable comparative analysis of topics across different contexts to help discover interesting patterns and potentially interesting events associated with the patterns.

In Figure 19.8, we show spatiotemporal patterns of the coverage of the topic of government response in the same data set of blog articles about Hurricane Katrina. These visualizations show the distribution of the coverage of the topic in different weeks of the event over the states in the U.S. We see that initially, the coverage is concentrated mostly in the victim states in the south, but the topic gradually spread to other locations over time. In week four (shown on the bottom left), the coverage distribution pattern was very similar to that of the first week (shown on the top left). This can again be explained by Hurricane Rita hitting the region around

(a) Week One: 08/23–08/29

(b) Week Two: 08/30–09/05

(c) Week Three: 09/06–09/12

(d) Week Four: 09/13–09/19

Theme 1 Government Response
bush 0.0716374 president 0.0610942 federal 0.0514114 govern 0.0476977 fema 0.0474692 administrate 0.0233903 response 0.0208351 brown 0.0199573 blame 0.0170033 governor 0.0142153

(d) Week Five: 09/20–09/26
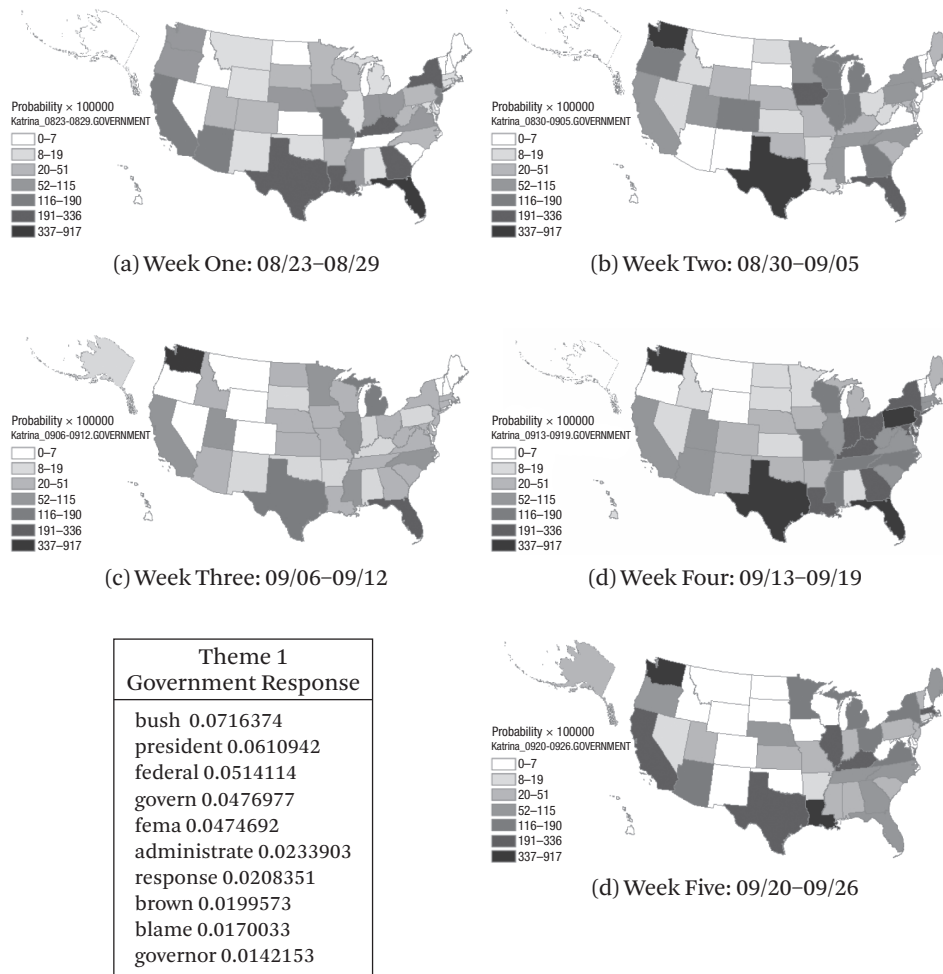
Figure 19.8 **Spatiotemporal trends of the coverage of the topic about the governmental response.** (Based on results from Mei et al. [2006a])

that time. These results show that CPLSA can leverage location and time as context to reveal interesting topical patterns in text data. Note that the CPLSA model is completely general, so it can be easily applied to other kinds of text data to reveal similar spatiotemporal patterns or topical patterns.

In Figure 19.9, we show yet another application of CPLSA for analysis of the impact of an event. The basic idea is to compare the views of topics covered in text before and after an event so as to reveal any difference. The difference can be

vector	0.0514
concept	0.0298
extend	0.0297
model	0.0291
space	0.0236
boolean	0.0151
function	0.0123
feedback	0.0077
...	

xml	0.0678
email	0.0197
model	0.0191
collect	0.0187
judgment	0.0102
rank	0.0097
subtopic	0.0079
...	

SIGIR papers

Topic:
Retrieval models

term	0.1599
relevance	0.0752
weight	0.0660
feedback	0.0372
independence	0.0311
model	0.0310
frequent	0.0233
probabilistic	0.0188
document	0.0173
...	

A seminal paper [Croft & Ponte 98]

1992

Start of TREC

1998

Year

probabilist	0.0778
model	0.0432
logic	0.0404
ir	0.0338
boolean	0.0281
algebra	0.0200
estimate	0.0119
weight	0.0111
...	

model	0.1687
language	0.0753
estimate	0.0520
parameter	0.0281
distribution	0.0268
probable	0.0205
smooth	0.0198
markov	0.0137
likelihood	0.0059
...	

Figure 19.9   **Using CPLSA for event impact analysis.** (Courtesy of Qiaozhu Mei)

assumed to be potentially related to the impact of the event. The results shown here are the topics discovered from research articles on information retrieval, particularly SIGIR papers. The topic we are focusing on is about the retrieval models (shown on the left). The goal is to analyze the impact of two events. One is the launch of the Text and Retrieval Conference (TREC) around 1992, a major annual evaluation effort sponsored by the U.S. government, which is known to have made a huge impact on the topics of research in information retrieval. The other event is the publication of a seminal paper in 1998 by Ponte and Croft [1998], in which the language modeling approach to information retrieval was introduced. The paper is also known to have made a high impact on information retrieval research. To understand the impact of these two events, we can use time periods before and after an event as different contexts and apply CPLSA.

The results on the top show that before TREC, the study of retrieval models was mostly on the vector space model and Boolean models, but after TREC, the study of retrieval models apparently explored a variety application tasks (e.g., XML

retrieval, email retrieval, and subtopic retrieval, which are connected to some tasks introduced in TREC over the years). The results on the bottom show that before the language modeling paper was published in 1998, the study of retrieval models focused on probabilistic, logic, and Boolean models, but after 1998, there was a clear focus on language modeling approaches and parameter estimation, which is an integral part of studies of language models for IR. Thus, these results can help potentially reveal the impact of an event as reflected in the text data.

19.4   Topic Analysis with Social Networks as Context

In this section, we discuss how to mine text data with a social network as context. The context of a text article can sometimes form a network; the authors of research articles might form collaboration networks. Similarly, authors of social media content might form social networks via a social network platform such as Twitter or Facebook. For example, in Twitter, people might follow each other, whereas in Facebook, people might be friends. Such a network context can indirectly connect the content written by the authors involved in a network.

Similarly, locations associated with text can also be connected to form geographical networks, which again add indirect connections between text data that would otherwise not be directly connected.

In general, you can imagine the metadata of the text data can easily form a network if we can identify some relations among the metadata. When we have network context available, it offers an interesting opportunity to jointly analyze text and its associated network context. The benefit of such a joint analysis is as follows. First, we can use a network to impose some constraints on topics of text. For example, it's reasonable to assume that authors connected in collaboration networks tend to write about similar topics. Such heuristics can be used to guide us in analyzing topics. Second, text can also help characterize the content associated with each subnetwork. For example, the difference between the opinions expressed in two subnetworks can be revealed by doing this type of joint analysis.

We now introduce a specific technique for analysis of text with network as context called a network supervised topic model. The general idea of such a model is illustrated in Figure 19.10. First, we can view any generative model (e.g., PLSA) as defining an optimization problem where the variables are the parameters (denoted by Λ here in the figure), and the objective function is the likelihood function. From the perspective of mining text data, the estimated parameter values Λ^* can be regarded as the output of the mining algorithm based on the model. With this view, we can then potentially use any context information of the text data to impose constraints or preferences on the parameters so as to incorporate domain

- Probabilistic topic modeling as optimization: maximize likelihood

$$\Lambda^* = \arg\max_\Lambda p(\text{TextData}|\Lambda)$$

- Main idea: network imposes restraints on model parameters Λ
  - The text at two adjacent nodes of the network tends to cover similar topics
  - Topic distributions are smoothed over adjacent nodes
  - Add network-induced regularizers to the likelihood objective function
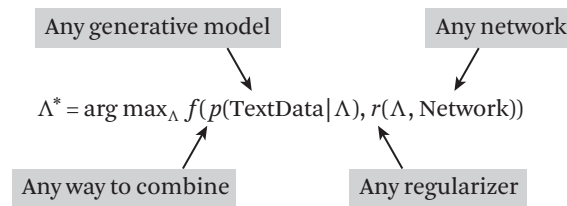
Figure 19.10   The general idea of a regularizing likelihood function.

knowledge or any preferences dictated by the user or application. This would lead to a modification to the original optimization problem such that the likelihood is no longer the only objective to optimize.

Following this thinking, the main idea of performing joint analysis of text and associated network context is to use the network to impose some constraints on the model parameters. For example, the text at adjacent nodes of the network can be assumed to cover similar topics. Indeed, in many cases, they do tend to cover similar topics; two authors collaborating with each other tend to publish papers on similar topics. Such a constraint or preference would essentially attempt to smooth the topic distributions on the graph or network so that adjacent nodes will have very similar topic distributions. This means we expect them to share a common distribution on the topics, or have just slight variations of the topic coverage distribution.

We add a network-induced regularizer to the likelihood objective function, as shown in Figure 19.10. That is, instead of optimizing the probability $p(\text{TextData} \mid \Lambda)$, we will optimize another function f, which combines the likelihood function $p(\text{TextData} \mid \Lambda)$ with a regularizer $r(\Lambda, Network)$ defined based on whatever preferences we can derive from the network context. When optimizing the new objective function f, we would seek a compromise between parameter values that maximize the likelihood and those that satisfy our regularization constraints or preferences. Thus we may also view the impact of the network context as imposing some prior on the model parameters if we view the new optimization problem conceptually as Bayesian inference of parameter values, even though we do not have any explicitly defined prior distribution of parameters.

Note that such an idea of regularizing likelihood function is quite general; indeed, the probabilistic model can be any generative model for text (such as a language model), and the network can be also any network or graph that connects the text objects that we hope to analyze. The regularizer can also be any regularizer that we would like to use to capture different heuristics suitable for a particular application; it may even be a combination of multiple regularizers. Finally, the function f can also vary, allowing for many different ways to combine the likelihood function with the regularizers. Another variation is to specify separate constraints that must be satisfied based on network context, making a constrained optimization problem.

Although the idea is quite general, in practice, the challenge often lies in how to instantiate such a general idea with specific regularizers so as to make the optimization problem remain tractable. Below we introduce a specific instantiation called NetPLSA (shown in Figure 19.11), which is an extension of PLSA to incorporate network context by implementing the heuristic that the neighbors on the network must have similar topic distributions.

As shown in Figure 19.11, the new modified objective function is a weighted sum of the standard PLSA likelihood function and a regularizer where the parameter $\lambda \in [0, 1]$ controls the weight on the regularizer. Clearly, if $\lambda = 0$, the model reduces to the standard PLSA. In the regularizer, we see that the main constraint is the
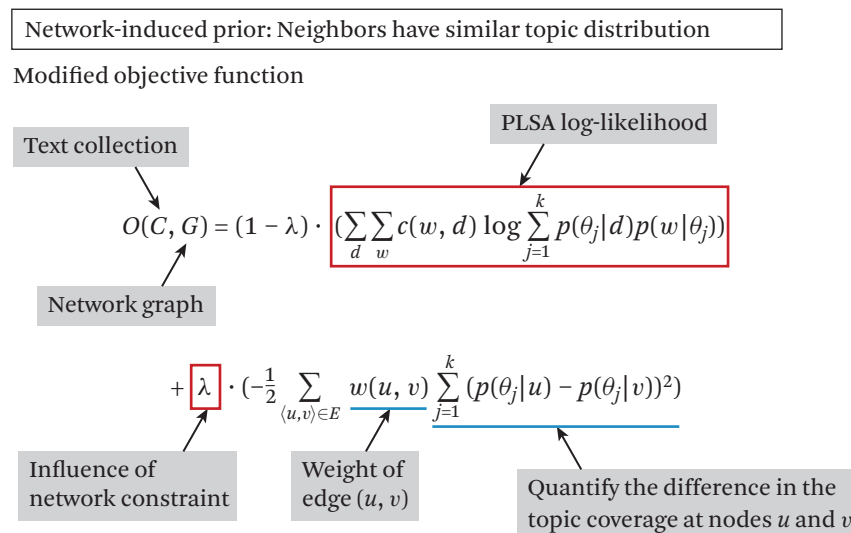
Figure 19.11   The NetPLSA model.

square loss defined on the difference of the topic selection probabilities of the two neighboring nodes u and v: $\sum_{j=1}^{k}(p(\theta_j \mid u) - p(\theta_j \mid v))^2$, which strongly prefers to give $p(\theta_j \mid u)$ and $p(\theta_j \mid v)$ similar values. In front of this regularization term, we see a weight $w(u, v)$, which is based on our network context where the edges may be weighted. This weight states that the more connected the two nodes are, the more important it is to ensure the two nodes have similar topics. In the case when the edges are not weighted, we may set $w(u, v) = 1$ if there exists an edge between u and v, and $w(u, v) = 0$ otherwise, essentially to keep only the regularizer for edges that exist on the graph. Note that there's a negative sign in front of the regularizer because while we want to maximize the likelihood part, we want to minimize the loss defined by the regularizer.

Such a modified optimization problem can still be solved using a variant of the EM algorithm, called General EM, where in the M-step, the algorithm does not attempt to find a maximum of the auxiliary function, but instead, just finds a new parameter value that would increase the value of the auxiliary function, thus also ensuring an increase of the likelihood function due to the fact that the auxiliary function is a lower bound of the original function (see Section 17.3.5 on the EM algorithm for more explanation about this). The whole algorithm is still a hill-climbing algorithm with guarantee of convergence to a local maximum.

In Figure 19.12, we show the four major topics discovered using the standard PLSA from a bibliographic database data set DBLP which consists of titles of papers from four research communities, including information retrieval (IR), data mining

Topic 1		Topic 2		Topic 3		Topic 4	
term	0.02	peer	0.02	visual	0.02	interface	0.02
question	0.02	patterns	0.01	analog	0.02	towards	0.02
protein	0.01	mining	0.01	neurons	0.02	browsubg	0.02
training	0.01	clusters	0.01	vlsi	0.01	xml	0.01
weighting	0.01	stream	0.01	motion	0.01	generation	0.01
multiple	0.01	frequent	0.01	chip	0.01	design	0.01
recognition	0.01	e	0.01	natural	0.01	engine	0.01
relations	0.01	page	0.01	cortex	0.01	service	0.01
library	0.01	gene	0.01	spike	0.01	social	0.01

Figure 19.12   **Sample results of PLSA.** (Based on results from Mei et al. [2008])

Information Retrieval		Data Mining		Machine Learning		Web	
retrieval	0.13	mining	0.11	neural	0.06	web	0.05
information	0.05	data	0.06	learning	0.02	services	0.03
document	0.03	discovery	0.03	networks	0.02	semantic	0.03
query	0.03	databases	0.02	recognition	0.02	services	0,03
text	0.03	rules	0.02	analog	0.01	peer	0.02
search	0.03	association	0.02	vlsi	0.01	ontologies	0.02
evaluation	0.02	patterns	0.02	neurons	0.01	rdf	0.02
user	0.02	frequent	0.01	gaussian	0.01	management	0.01
relevance	0.02	streams	0.01	network	0.01	onotology	0.01

Figure 19.13 **Sample results of NetPLSA (in comparison to PLSA).** (Based on results from Mei et al. [2008])

(DM), machine learning (ML), and World Wide Web (Web). The data set has been constructed by pooling together papers from these research communities, and our goal is to see if NetPLSA can more successfully learn topics well aligned to the communities than the standard PLSA. The results in Figure 19.12 show that PLSA is unable to generate the four communities that correspond to our intuition. The reason was because they are all mixed together and there are many words that are shared by these communities, and the co-occurrence statistics in the data are insufficient for separating them.

In contrast, the results of NetPLSA, shown in Figure 19.13, are much more meaningful, and the four topics correspond well to the four communities that we intend to discover from the data set. Indeed, it is very easy to label them with the four communities as shown in the table. The reason why NetPLSA can separate these communities well and discover more meaningful topics is because of the influence of the network context. Since our network is the collaboration network of authors, when we impose the preference for two nodes connected in the network to have similar topics, the model would further tune the discovered topics to better reflect the topics worked on by authors involved in the same collaboration network. As a result, the topics would be more coherent and also better correspond to the communities (represented by subnetworks of collaboration). These results are also useful for characterizing the content associated with each subnetwork of collaboration.

Taking a more general view of text mining in the context of networks, we can treat text data as living in a rich information network environment. That is, we can

connect all the related data together as a big network, and associate text data with various structures in the network. For example, text data can be associated with the nodes of the network; such a case can be analyzed by using NetPLSA as we have just discussed. However, text data can also be associated with edges in a network, paths, or even subnetworks to help discover topics or perform comparative analysis.

19.5 Topic Analysis with Time Series Context

In many applications, we may be interested in mining text data to understand events that happened in the real world. As a special case, we may be interested in using text mining to understand a time series. For example, we might have observed a sudden drop in prices on the stock market in a particular time period and would like to see if the companion text data such as news might help explain what happened. If the crashing time of the stocks corresponds to a time when a particular news topic suddenly appeared in the news stream, there might be a potential relationship between the topic and the stock crash. Similarly, one might also be interested in understanding what topics reported in the news stream were relevant for a presidential election, and thus interested in finding topics in news stream that are correlated with the fluctuation of the Presidential Prediction Market (which measures people's opinions toward each presidential candidate).

All these cases are special cases of a general problem of joint analysis of text and a time series to discover causal topics. Here we use the term *causal* in a non-rigorous way to refer to any topic that might be related to the time series and thus can be *potentially* causal. This analysis task is illustrated in Figure 19.14.

The input includes a time series plus text data that are produced in the same time period, also known as a companion text stream. The time series can be regarded as a

- Input:
  - Time series
  - Text data produced in a similar time period (text stream)
- Output:
  - Topics whose coverage in the text stream has strong correlations
    with the time series ("causal" topics)
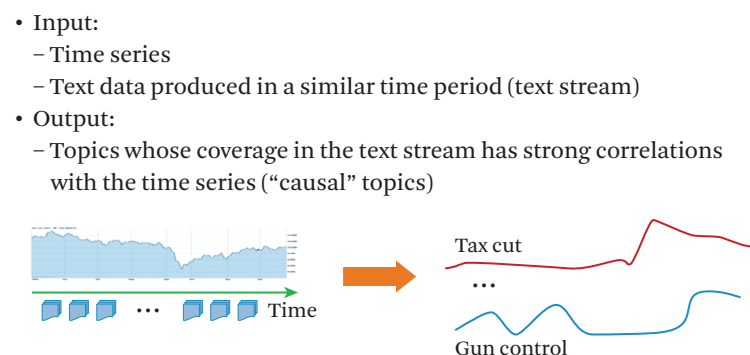
Figure 19.14 The task of causal topic mining.

context for analyzing the text data. The output that we want to generate is the topics whose coverage in the text stream has strong correlations with the time series. That is, whenever the topic is mentioned frequently in the text stream, the time series variable tends to have higher (or lower) values.

We call these topics causal topics since they can *potentially* explain the cause of fluctuation of the time series and offer insights for humans to further analyze the topics for better understanding of the time series. They can also be useful features for predicting time series.

Intuitively, the output is similar to what we generate by using a topic model, but with an important difference. In regular topic modeling, our goal is to discover topics that best explain the content in the text data, but in our setup of discovering causal topics, the topics to be discovered should not only be semantically meaning-ful and coherent (as in the case of regular topic modeling), but also be correlated with the external time series.

To solve this problem, a natural idea is to apply a model such as CPLSA to our text stream so as to discover a number of topics along with their coverage over time. This would allow us to obtain a time series for each topic representing its coverage in the text such as the temporal trends shown in Figure 19.7. We can then choose the topics from this set that have the strongest correlation with the external time series.

However, this approach is not optimal because the content of the topics would have been discovered solely based on the text data (e.g., maximizing the likelihood function) without consideration of the time series at all. Indeed, the discovered topics would tend to be the major topics that explain the text data well (as they should be), but they are not necessarily correlated with time series. Even if we choose the best ones from them, the most correlated topics might still have a low correlation, and thus not be very useful from the perspective of discovering causal topics.

One way to improve this simple approach is to use time series context to not only select the topics with the highest correlations with the time series, but also influence the content of topics. One approach is called Iterative Causal Topic Modeling, shown in Figure 19.15.

The idea of this approach is to do an iterative adjustment of topics discovered by topic models using time series to induce a prior. Specifically, as shown in Figure 19.15, we first take the text stream as input and apply regular topic modeling to generate a number of topics (four shown here). Next, we use the external time series to assess which topic is more causally related (correlated) with the external time series by using a causality measure such as Granger Test. For example, in this
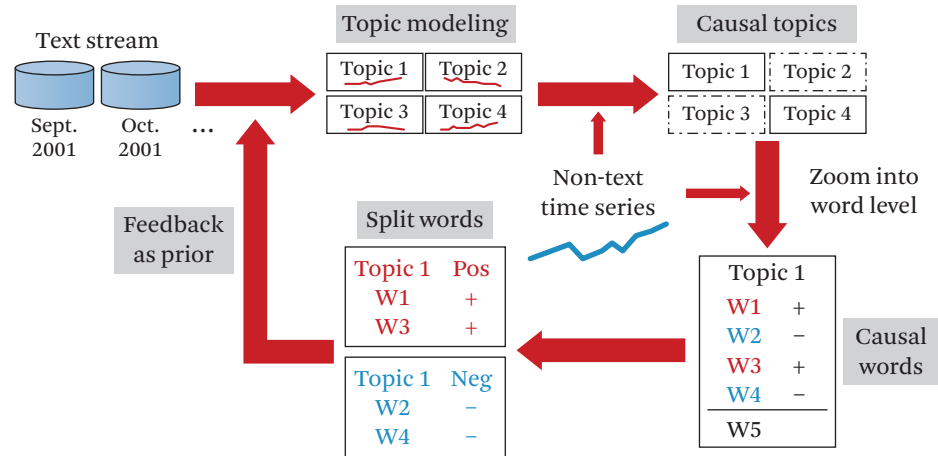
Figure 19.15   **Causal topic discovery algorithm.** (Adapted from Kim et al. [2013])

figure, topic 1 and topic 4 may be more correlated than topic 2 and topic 3. The simple approach that we discussed earlier would have just stopped here and taken topics 1 and 4 as potential causal topics. However, here we go further to improve them by zooming into the word level to further identify the words that are most strongly correlated with the time series. Specifically, we can look into each word in the top ranked words for each topic (those with highest probabilities), and compute the correlation of each word with the time series.

This would allow us to further separate those words into three groups: strongly positively correlated words; strongly negatively correlated words; and weakly correlated words. The first two groups can then each be regarded as seeds to define two new subtopics that can be expected to be positively and negatively correlated with the time series, respectively. The figure shows a potential split of topic 1 into two such potentially more correlated subtopics: one with w_1 and w_3 (positive) and one with w_2 and w_4 (negative). However, these two subtopics may not necessarily be coherent semantically. To improve the coherence, the algorithm would not take these directly as topics, but rather feed them as a prior to the topic model so as to steer the topic model toward discovering topics matching these two subtopics. Thus, we can expect the topics discovered by the topic model in the next iteration to be more correlated with the time series than the original topics discovered from the previous iteration. Once we discover a new generation of topics, we can repeat the process to analyze the words in correlated topics and generate another set of seed topics, which would then be fed into the topic model again as prior.
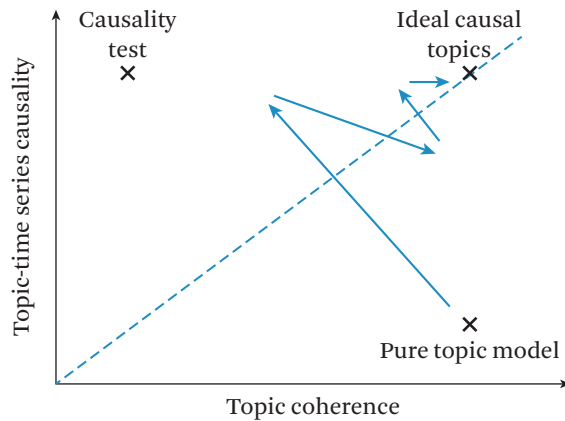
Figure 19.16   **Alternating optimization of coherence and causality/correlation.** (Courtesy of Hyun Duk Kim)

   The whole process is seen as a heuristic way of optimizing causality and coherence, which is precisely our goal in discovery of causal topics. When applying the topic model, we ensure the semantic coherence in the discovered topics, but when splitting a topic into positively and negatively subtopics, we improve the correlation with the time series, essentially iteratively improving both coherence and correlation (causality), as illustrated in Figure 19.16.

   Here we see that the pure topic models will be very good at maximizing topic coherence, thus scoring high on the x-axis, meaning the discovered topics will all be meaningful. If we only use a causality test or correlation measure, then we would generate a set of words that are strongly correlated with the time series, thus scoring high on the y-axis (causality), but they aren't necessarily coherent semantically. Our goal is to have a causal topic that scores high, in both topic coherence and correlation. The approach discussed above can be regarded as an alternate way to maximize both axes. When we apply the topic models we're maximizing the coherence, while when we decompose the topic model words into sets of words that are very strongly correlated with the time series, we would select the most strongly correlated words with the time series. Thus we are, in effect, pushing the model back to the causal dimension to make it better in causal scoring. When we apply the selected words as a prior to guide topic models in topic discovery, we again go back to optimize the coherence. Eventually, such an iterative process can be expected to reach a compromise of semantic coherence and strong correlation with time series.

This general approach relies on two specific technical components: a topic model and a causality measure. The former has already been introduced earlier in the book, so we briefly discuss the latter. There are various ways to measure causality between two time series. The simplest measure is Pearson correlation. Pearson correlation is one of the most common methods used to measure the correlation between two variables. It gives us a correlation value in the range of $[-1, +1]$, and the sign of the output value indicates the orientation of the correlation (which we will exploit to quantify the impact in the case of a causal relation). We can also measure the significance of the correlation value. If used directly, the basic Pearson correlation would have zero lag because it compares values on the same time stamp. However, we can compute a lagged correlation by shifting one of the input time series variables by the lag and measuring the Pearson correlation after the shift.

A more common method for causality test on time series data is the Granger Test. The Granger test performs a statistical significance test with different time lags by using autoregression to see if one time series has a causal relationship with another series. Formally, let y_t and x_t be two time series to be tested, where we hope to see if x_t has Granger causality for y_t with a maximum p time lag. The basic formula for the Granger test is the following:

$$y_t = a_0 + a_1 y_{t-1} + \ldots + a_p y_{t-p} + b_1 x_{t-1} + \ldots + b_p x_{t-p}. \tag{19.1}$$

We then perform an F-test to evaluate if retaining or removing the lagged x terms would make a statistically significant difference in fitting the data. Because the Granger test is essentially an F-test, it naturally gives us a significance value of causality. We can estimate the impact of x on y based on the coefficients of the x_i terms; for example, we can take the average of the x_i term coefficients, $\frac{\sum_{i=1}^{p} b_i}{p}$, use it as an "impact value." The impact values can be used to assign weights to the selected seed words so that highly correlated words would have a higher probability in the prior that we pass to topic modeling.

We now show some sample results generated by this approach to illustrate the applications that it can potentially support. First, we show a sample of causal topics discovered from a news data set when using two different stock time series as context in Figure 19.17.

The text data set here is the *New York Times* news articles in the time period of June 2000 through December 2011. The time series used is the stock prices of two companies, American Airlines (AAMRQ) and Apple Inc. (AAPL). If we are to use a topic model to mine the news data set to discover topics, we would obtain topics that are neutral to both American Airlines and Apple.

AAMRQ (American Airlines)	AAPL (Apple)
russia russian putin	paid notice st
europe european	russia russian europe
germany	olympicgames olympics
bush gore presidential	she her ms
police court judge	oil ford prices
airlines airport air	black fashion blacks
united trade terrorism	computer technology software
food foods cheese	internet com web
nets scott basketball	football giants jets
tennis williams open	japan japanese plane
awards gay boy	Topics are biased toward each time series
moss minnesota chechnya	

Figure 19.17   **Sample results: topics mined using two different stock time series.** (Adapted from Kim et al. [2013])

We would like to see whether we can discover biased topics toward either American Airlines or Apple when we use their corresponding time series as context. The results here show that the iterative causal topic modeling approach indeed generates topics that are more tuned toward each stock time series. Specifically, on the left column, we see topics highly relevant to American Airlines, including, e.g., a topic about airport and airlines, and another about terrorism (the topic is relevant because of the September 11th terrorist attack in 2001, which negatively impacted American Airlines), though the correlation of other topics with American Airlines stock is not obvious. In contrast, on the right column, we see topics that are clearly more related to AAPL, including a topic about computer technology, and another about the Web and Internet. While not all topics can be easily interpreted, it is clear the use of the time series as context has impacted the topics discovered and enabled discovery of specific topics that are intuitively related to the time series. These topics can serve as entry points for analysts to further look into the details for any potential causal relations between topics and time series. These results also clearly suggest the important role that humans must play in any real application of causal topic discovery, but these topics can be immediately used as features in a predictive model (for predicting stock prices). It is reasonable to assume that some of these topics would make better features than simple features such as *n*-grams or ordinary topics discovered from the collection without considering the time series context.

Top three words in significant topics from *New York Times*

Text: *New York Times* (May 2000 – Oct. 2000)
Time series: Iowa Electronic Market
http://tippie.uiowa.edu/iem/

tax cut 1
screen pataki guiliani
enthusiasm door symbolic
oil energy prices
news w top
pres al vice
love tucker presented
partial **abortion** privatization
court supreme **abortion**
gun control nra

Issues known to be important in the 2000 presidential election

Figure 19.18 **Sample results of causal topics discovered using Presidential Prediction Market.** (Adapted from Kim et al. [2013])

In Figure 19.18, we see some additional results from analyzing presidential election time series. The time series data used here is from the Iowa Electronic market, and the text data is the *New York Times* data from May–October 2000 that matched at least one candidate's name (i.e., either *Bush* or *Gore*). The goal here was to see if we can use causal topic mining to help understand what issues mattered in the 2000 presidential campaign and election. The results shown here are the top three words from the most significant (causal) topics from *New York Times*. Intuitively, they are indeed quite related to the campaign. The high relevance of topics discovered is at least partly due to the use of the presidential candidate names as an additional context (i.e., as filters), which helped eliminate a lot of non-relevant text data. However, what is interesting is that the list does contain a few topics that are known to be important in that presidential election, including tax cut, oil energy, abortion, and gun control (see Kim et al. [2013] for a more detailed discussion of the results).

19.6 Summary

In this chapter, we discussed the general problem of analyzing both structured data and unstructured text data in a joint manner, which is needed for predictive modeling based on big data. We specifically focused on the discussion of text-based prediction, which is generally very useful for big data applications that involve text data. Since text-based prediction can help us infer new knowledge about the world,

some of which can even go beyond what's discussed in the content of text, text-based prediction is often very useful for optimizing our decision making, especially when combined with other non-text data that are often also available, and it has widespread applications.

Non-text data can provide a context for mining text data, while text data can also help interpret patterns discovered from non-text data (such as pattern annotation). The joint analysis of text and non-text data is a relatively new active research frontier. In this chapter, we covered a number of general techniques that combine topic analysis with non-text data, including contextual probabilistic latent semantic analysis (CPLSA) that embeds context variables such as time and location directly in a topic model, network-supervised topic modeling that uses companion network/graph structure of text data to regularize topic discovery, and time series as context for discovering potentially causal topics from text data. Due to space limitations, we only provided a brief introduction to the high-level ideas of these approaches without elaboration, but we have included sample results of all of them to help understand the potential applications that they can enable. These approaches are all general, and thus they can be potentially applied to many different domains.

Bibliographic Notes and Further Reading

The dissertation Mei [2009] has an excellent discussion of contextual text mining with many specific explorations of using context for text analysis, notably with topic modeling. Specifically, both the contextual probabilistic latent semantic analysis model [Mei and Zhai 2006] and topic modeling with network as context [Mei et al. 2008] are discussed in detail in the dissertation. The iterative topic modeling approach with time series for supervision is described in Kim et al. [2013]. A general discussion of text-driven forecasting and its many applications can be found in Smith [2010]. Text and the companion structured data can often be generally modeled as an information network. A systematic discussion of algorithms for analyzing information networks can be found in Sun and Han [2012]. Interactive joint analysis of text and structured data can also be supported by combining the traditional Online Analytical Processing (OLAP) techniques with topic modeling to enable users to drill-down and roll-up in the "text dimension" using a hierarchical topic structure as described in Zhang et al. [2009].

Exercises

1. We gave several examples of how to integrate text and structured data into one application. Brainstorm some other ideas that were not discussed in this

chapter. What types of techniques would be used to support the application? Consider, for example, clustering, categorization, or topic analysis.

2. In our discussion on contextual text mining, we showed several ways to partition data, e.g., by year or venue for research papers. Imagine that we would like to add another context variable such as author affiliation. Can we simply filter out results based on a target affiliation or is it necessary to rerun the entire application to incorporate this additional knowledge?

3. In contextual text mining, we gave several example queries.

   - What topics have been getting increasing attention recently in data mining research?
   - Is there any difference in the responses of people in different regions to an event?
   - What are the common research interests of two researchers?
   - Is there any difference in the research topics published by authors in the U.S. and those outside?
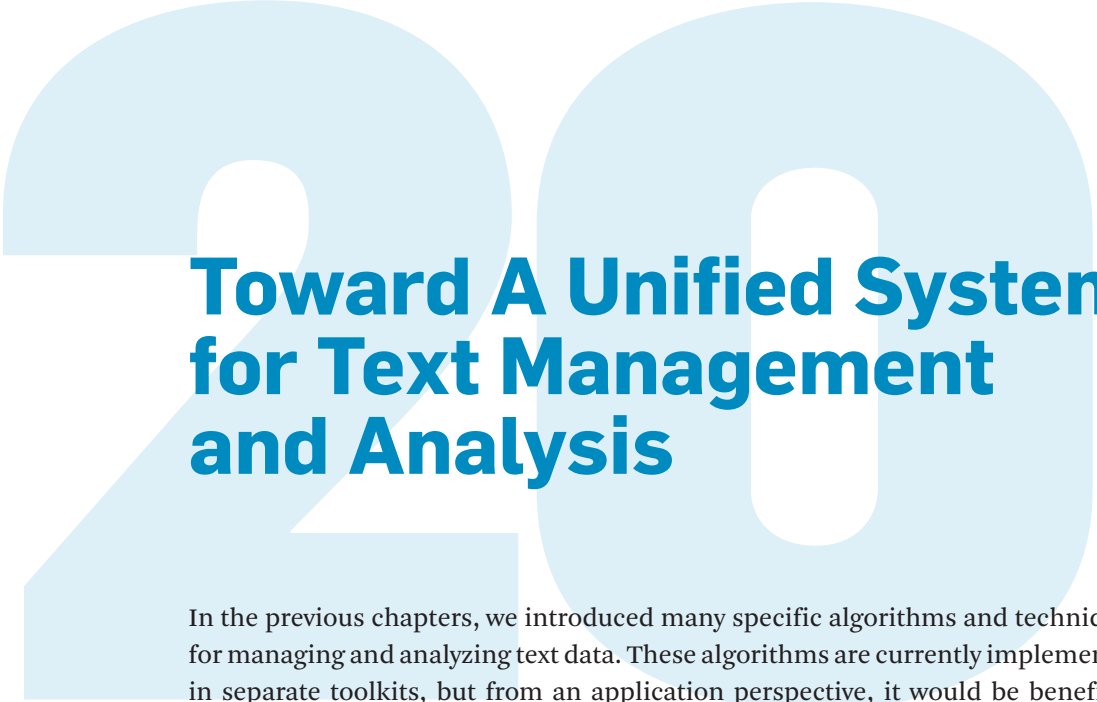
   What would actual answers returned by such a system look like? For example, what actual object(s) would be returned by the system and what would human users have to do in order to interpret them?

4. In what ways is CPLSA related to LARA or other sentiment topic mining algorithms? Does one have an advantage over the other in certain tasks?

5. For topic analysis with network context, we used networks to enforce constraints on topic models. Can we use the same model to predict when links will be formed between nodes? If not, is it possible to adjust the model to support this?

6. METME includes a graph library. Can you combine this with METME's topic models to create a network-constrained topic analysis? For example, the Yelp academic dataset (https://www.yelp.com/academic_dataset) contains both text and network information in addition to business review data.

7. The resolution of time series data will have some effect on joint applications with text data. For example, consider intraday stock data on the resolution of one second (that is, there is a data point for each symbol for every second). Would text mining applications integrating newspaper articles be able to take advantage of such data? How can this data best be leveraged?

8. In time series applications, data is often streaming (receiving updates at specified time intervals). Do the discussed models support streaming time series (and text) data? Is there any way to adjust them if they don't?

UNIFIED TEXT DATA MANAGEMENT ANALYSIS SYSTEM

Toward A Unified System for Text Management and Analysis

In the previous chapters, we introduced many specific algorithms and techniques for managing and analyzing text data. These algorithms are currently implemented in separate toolkits, but from an application perspective, it would be beneficial to develop a unified system that can support potentially all these algorithms in a general way so that it can be used in many different applications. In this chapter, we discuss how we can model these different algorithms as operators in a unified analysis framework and potentially develop a general system to implement such a framework.

From a user's perspective, we distinguished (at a very high level) two related application tasks: text data access and text analysis. The goal of text data access is to enable users to identify and obtain the most useful text data relevant to an application problem, which is often achieved through the support of a search engine. The current search engines primarily support querying, which is, however, only one way to help users find relevant documents. Browsing is another useful form of information access where the users can take the initiative to navigate through a structure into relevant information. The support of browsing is generally achieved by adding structures to text data by using clustering, categorization, or topic analysis. In addition to supporting querying and browsing, a search engine can also support recommendation of information, thus providing multi-mode information access through both pull mode (querying and browsing) and push mode (recommendation).

Since text data are created by humans and often meant to be consumed by humans—and the current NLP techniques are still not mature enough for computers to have accurate understanding of text data—it is generally necessary to involve

humans in any text data application. In this sense, a text data management and analysis system should serve as an intelligent assistant for users requiring information, or the analysts that would like to leverage text data for intelligence to optimize decision making.

Thus, it is quite important to optimize the collaboration of humans and machines. This means that we should take advantage of a computer's ability to handle large amounts of text data while exploiting humans' expertise in detailed language understanding and assessing knowledge for decision making. Supporting an interactive process to engage users in a "dialogue" with the intelligent text information system is generally beneficial as it enables the users and system to have more communications between each other and assist each other to work together toward accomplishing the common goal of solving a user's problem by analyzing text data.

Looking at the problem in this way, we can easily see the possibility of dividing the work between a human user and a machine in different ways. At one extreme, we would mostly rely on users to complete access and analysis tasks, and have the computer to do only the support that the computer can robustly provide. This is the current scenario when people use a search engine to perform mostly manual text mining, as shown in Figure 20.1. For example, we can all use a web search engine to help us get access to the most relevant information buried in the large amounts of text data on the Web, and then read all the relevant information, which is essentially manual text mining and analysis. In such a scenario, the search engine
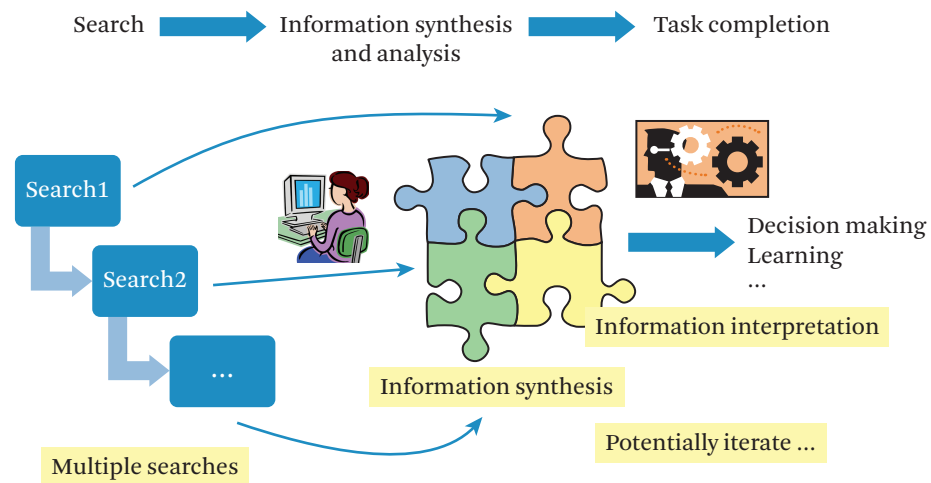
Figure 20.1   Manual analysis based on a search engine.
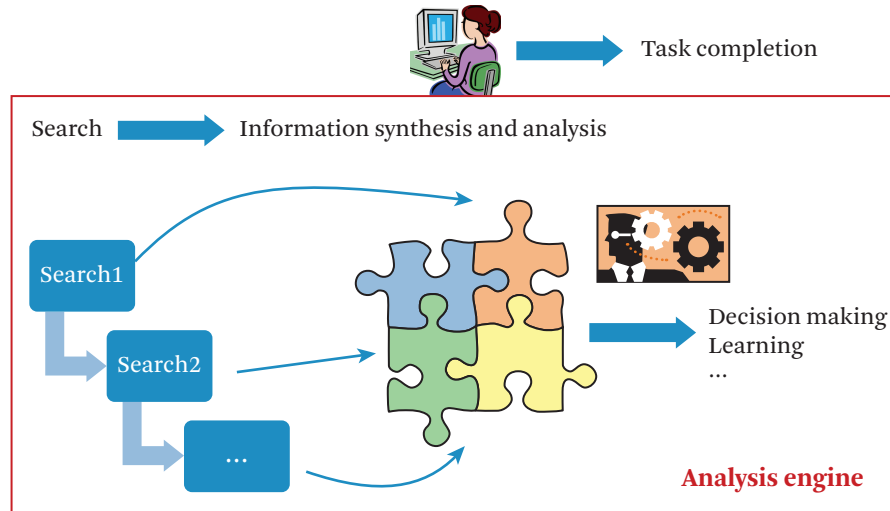
Figure 20.2   Extending a search engine to create an analysis engine that directly supports a user's task.

plays two important roles. For one, it enables a user to quickly identify the most relevant text data, which is often what we really need for solving a problem, thus avoiding dealing with a huge amount of non-relevant text data. Such a strategy of data selection (reduction) is logically the very first step we should take when tackling the scalability problem caused by the size of data. Secondly, it provides knowledge provenance in the sense that it allows a user to easily navigate into any source to examine its reliability in depth. At this extreme, the computer has done the minimum, if anything at all, to support text analysis. However, the system (i.e., search engine) is very robust and can handle large amounts of data quickly.

At the other extreme, the system can attempt to provide task support directly to the user so as to minimize the user's effort, as illustrated in Figure 20.2.

Providing such a support for arbitrary tasks or even a large class of tasks is impossible due to the inability of computers to understand natural language and the lack of knowledge about specific task requirements. As a result, we can only build very specialized predictive models for a particular problem, where the features generally have to be designed by humans. This kind of system may provide maximum support for a decision task, but the function of the system is inevitably restricted to a specific task. Since predictive modeling generally requires the use of machine learning techniques and combining features extracted from both text and non-text

data, it clearly requires additional support beyond text management and analysis. Yet, we can envision the possibility of developing a relatively general text analysis system to help users identify and extract effective features for a particular prediction task. In order to ensure generality, such a text analysis system must provide relatively general text analysis operators that can be applied to many different problems; most algorithms we introduced in this book are of this kind of nature and can thus be implemented in a system as general operators. However, each specific text analysis application would inevitably have different requirements, thus the operators must also be standardized and compatible with each other so that they can be flexibly combined to support potentially many different workflows. Note that text data access functions such as querying, browsing and recommendation, can all be regarded as instances of specific operators to be "blended" with other operators such as categorization, clustering, or topic analysis.

The benefit of such an operator-based text analysis system for supporting text mining and analysis is similar to the benefit that a comprehensive package of car-repairing toolkits brings to car repairers. In order to diagnose the problem of a car and fix it, repairers need to use many different tools and combine them in an ad hoc way to open up components in a car or probe a component with electrical testers. Although the actual workflow often varies depending on the specific problem and models of a car, a common set of tools is often sufficient to accommodate all kinds of tasks. In much the same way, in order to integrate scattered information, digest all the latent relevant knowledge about a problem in text data, analysts can also combine various text analysis tools to open up complex connections between entities or topics buried in text data, identify interesting patterns, and compute useful features for prediction tasks. As in the case of car repairing, although the actual workflow may vary dramatically depending on the specific application, a common set of extensible and trainable analysis tools might also be sufficient to accommodate all kinds of text analysis applications. Most algorithms we introduced in this book can potentially serve as such analysis tools, thus providing a basis for developing a unified system to enable analysts to perform text analysis in such a way. Although there are still many challenges in designing and building such a system, it is an important goal to work on.

20.1   Text Analysis Operators

Relational databases are able to support many different applications via a set of common operators as defined in a query language such as SQL. Similarly, we may identify a common set of operators for supporting text analysis. In this section, we discuss some possibilities of defining text analysis operators.

First, we would need to define the data types that we may be interested in processing. Naturally, the most important data type is a TEXTOBJECT, which can be defined as a sequence of words in a vocabulary set V. Clearly, a word, a phrase, a sentence, a paragraph, and an article can all be regarded as specific instances of the type TEXTOBJECT. We can then also naturally define derived data types based on TEXTOBJECT, including, e.g., TEXTOBJECTSET, which naturally captures instances such as a collection of text articles, a set of sentences, which further covers a set of terms as a special case when a sentence is just a term. Another possibility is TEXTOBJECTSEQUENCE, where we care about order of the text objects; TEXTOBJECTSEQUENCE can capture interesting data structures such as a ranked list of articles or sentences. Again, a special case is a ranked list of terms.

Based on these types, we may also further define a topic as a WEIGHTEDTEXT-OBJECTSET, where each text object is associated with a numerical weight. As a special case, we can have words as text objects, and thus have a word distribution represented as a WEIGHTEDTEXTOBJECTSET. WEIGHTEDTEXTOBJECTSEQUENCE can cover a ranked list of search results with scores.

Second, we need to define operators on top of various data types. Here we can potentially have a very large number of operators, depending on specific text analysis algorithms. Here we briefly discuss a few most commonly used algorithms that we covered in the previous chapters of the book, and show that even with a few operators, we can already potentially combine them in many different ways to flexibly support very different workflows required in a particular application context.

For example, in Figure 20.3, we show the following operators. Since our goal is mainly to present some speculative ideas, we present these operators in a mostly informal and non-rigorous way.

> **Select.** TEXTOBJECTSET → TEXTOBJECTSET. The **Select** operator maps a set of text objects into a subset of text objects. Querying, browsing, and recommendation can all be regarded as an instance of **Select**. We can additionally further apply a **Ranking** operator to each selection.
>
> **Split.** TEXTOBJECTSET → TEXTOBJECTSET, . . . , TEXTOBJECTSET. The **Split** operator maps a set of text objects into multiple subsets of text objects. Text categorization and text clustering are all instances of **Split**.
>
> **Union and Intersection.** these are standard set operators that can be applied to any sets.
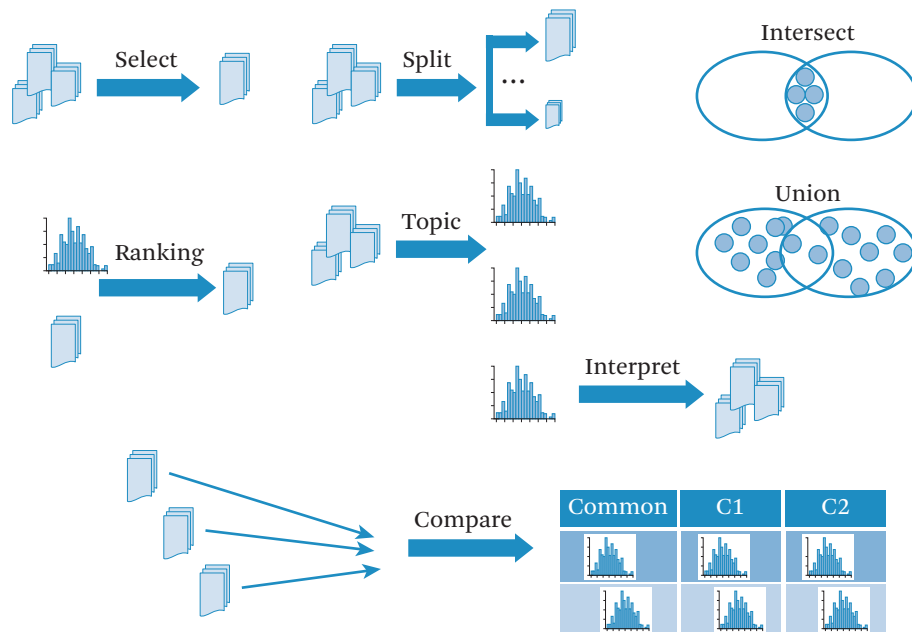
Figure 20.3 Illustration of potential operators for text analysis.

Ranking. (WEIGHTED) TEXTOBJECTSET → TEXTOBJECTSEQUENCE. The **Ranking** operator takes as input a weighted set of text objects that specifies the perspective of ranking and a set of text objects, and it produces a sequence of text objects (sorted in order) as output. As a special case, the WEIGHTED TEXTOBJECTSET can be a word distribution representing a query language model.

TopicExtraction. TEXTOBJECTSET → TOPICSET. The **TopicExtraction** operator maps a set of text objects into a set of topics.

Interpret. {TOPIC, TEXTOBJECTSET} → TEXTOBJECTSET. The **Interpret** operator maps a topic and a set of text objects into another set of text objects.

Compare. {TEXTOBJECTSET, . . . , TEXTOBJECTSET} → TOPICSET, . . . , TOPICSET. The **Compare** operator maps a set of comparable sets of text objects into a set of common topics covered in all the comparable sets of text objects, and sets of context-specific topics.

The formalization of some of the operators is illustrated in Figure 20.4, where θ denotes a weighted word vector. Even with these few operators, we already have some interesting combinations. For example, in Figure 20.5, we see an example of

- $C = \{D1, \ldots, Dn\}$; $S, S1, S2, \ldots, Sk$ subset of C
- Select Operator
  - Querying(Q): $C \rightarrow S$
  - Browsing: $C \rightarrow S$
- Split
  - Categorization (supervised): $C \rightarrow S1, S2, \ldots, Sk$
  - Clustering (unsupervised): $C \rightarrow S1, S2, \ldots, Sk$
- Interpret
  - $C \times \theta \rightarrow S$
- Ranking
  - $\theta \times Si \rightarrow$ ordered Si

Figure 20.4   Formalization of text analysis operators.

$$\text{Interpret}(\text{Compare}(\text{Select}(T1, C), \text{Select}(T2, C), \ldots, \text{Select}(Tk, C)), C)$$
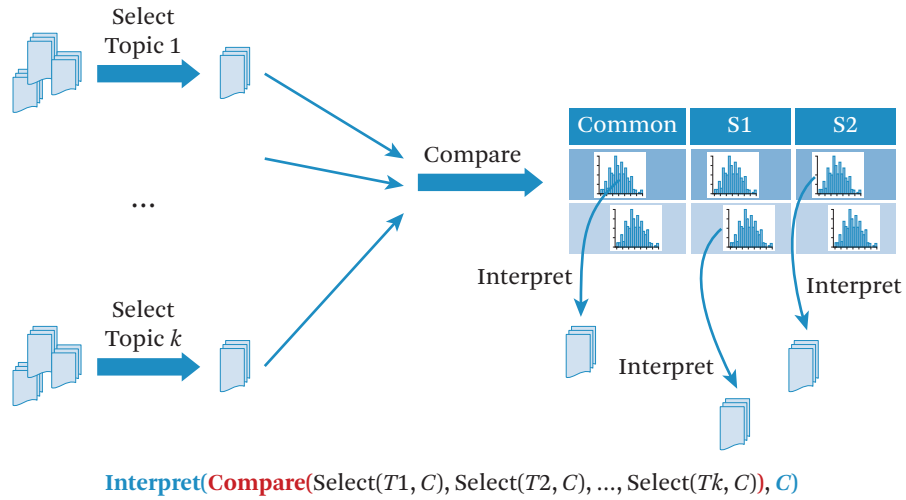
Figure 20.5   Example of combination of topic selection, comparison, and interpretation.

combination of multiple topic selections followed by a comparison operator, which would then be followed by an **Interpret** operator.

In Figure 20.6, we show another example of combination of a **Split** operator followed by a comparison operator, which would then be followed by an **Interpret** operator. It is easy to imagine that there are many other ways to combine these operators.

$$\textbf{Interpret}(\textbf{Compare}(\text{Split}(S, k)), C)$$
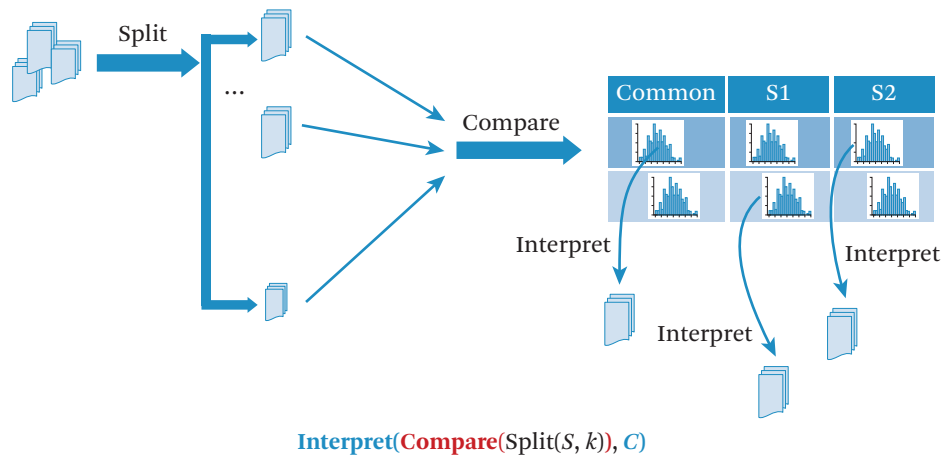
Figure 20.6 Example of combination of topic splitting, comparison, and interpretation.

20.2 System Architecture

In general, we can envision a high-level architecture of a unified system for supporting text management and analysis, as shown in Figure 20.7, where multiple levels of services are provided to users, including a preprocessing step of natural language processing, a low-level service for multi-mode text data access, which includes querying, browsing, and recommendation, a medium-level service for text data analysis, which includes general analysis operators that can be combined with each other, and high-level application support, which includes support of application-specific user tasks. It is important that a user has access to all these levels via a unified interaction interface where the user also has access to a working space that is personalized according to a specific user and a specific application task. In this figure, we also show the availability of the non-textual data, which would generally need a database system to manage it and serve some other modules such as text analysis (in this case, non-text data can be used as context for analyzing text data). Finally, we see that we can often further apply general data mining algorithms such as EM and predictive modeling to process the results that our analysts have obtained.

As an example, consider a news mining application. If a user performs a keyword search, a large number of documents containing matching text may be returned. If we run a topic analysis on the set of returned articles, we can enable browsing the results to allow the user to more efficiently sift through the data. If the user
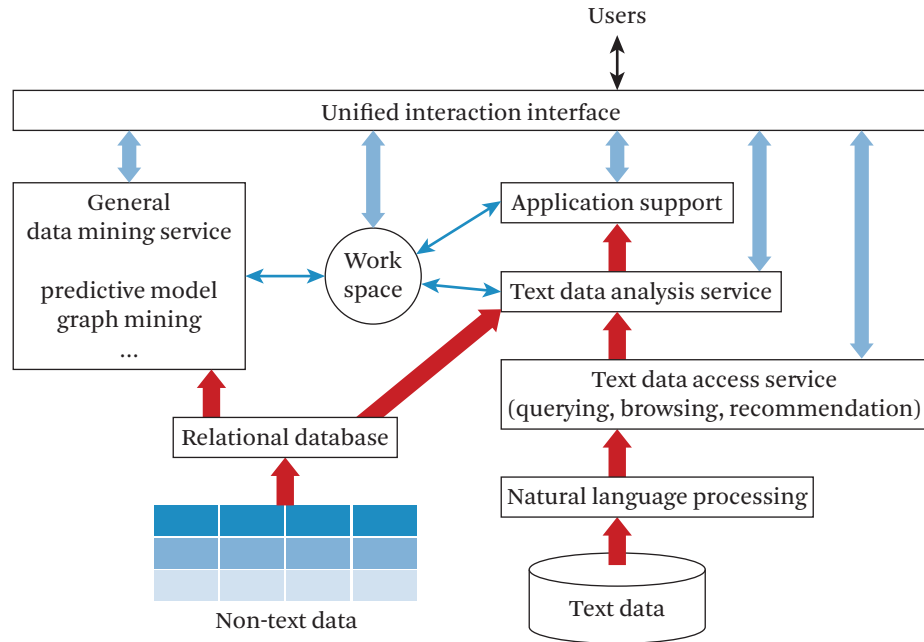
Figure 20.7   **High level architecture of a unified text analysis system.**

finds two promising clusters, they can be merged together and searched again with different keywords. This process may be repeated until the user's information need is satisfied.

20.3 META as a Unified System

In this section, we discuss some interesting possibilities of extending META to a general and unified text data management and analysis system. In one direction, we can envision implementing the architecture shown in Figure 20.8 where we extend a search engine to support various topic models, which can further enable improved text categorization, text summarization, and text clustering.

Indexes (forward and inverted) are a common storage mechanism for most text mining applications in META. Analyzers and tokenizers are the first operators applied on text, where various features are created and term IDs are assigned. If the terms are not stored in an index, they are usually stored in a `meta::sequence`, which maintains term order for further analysis. Additionally, most text mining applications take an index as input.
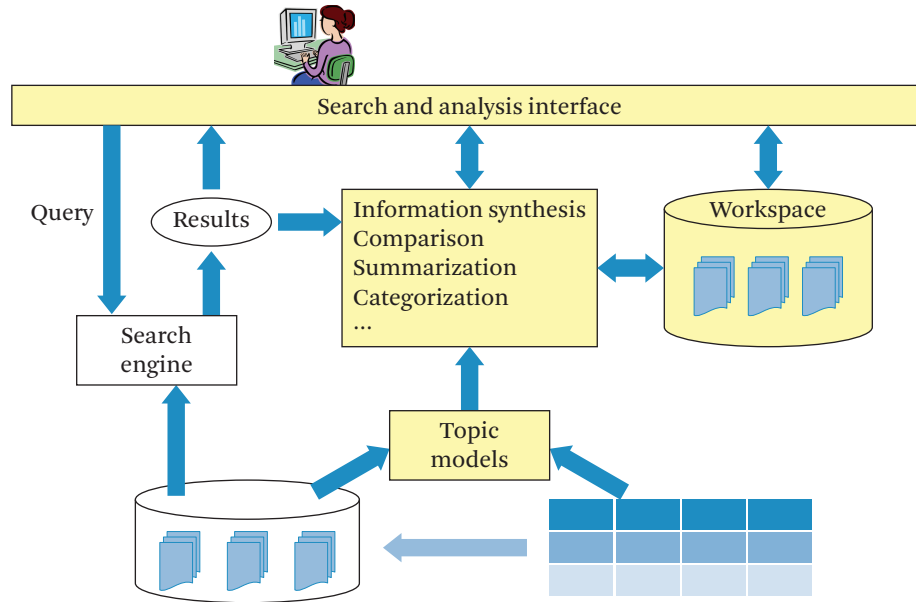
Figure 20.8   Extension of a search engine to create a general topic analysis engine.

This common structure for all tasks enables higher-level "wrapper" functions to be implemented that act in the same way as previously described in this chapter. For example:

- TextObject is represented in MeTA as a `meta::corpus::document`

- TextObjectSet is `meta::index::forward_index` or `meta::index::inverted_index`

- One example of a TextObjectSequence is `meta::sequence::sequence`.

- One example of a WeightedTextObjectSequence is the output from the `score` function in `meta::index::ranker`.

If we make higher-level functions that pass these objects to different analysis components, we enable this unified view of text analysis. Using `meta::corpus::metadata` allows basic structured data to be stored for each document. More advanced structured operations like those found in databases are not currently implemented, but simple filtering and grouping commands are easily supported, such as the filtering functions in `meta::index::ranker` and `meta::classify::classifier::knn`.

The index structure also facilitates the idea mentioned earlier: advanced text mining techniques are run on a smaller relevant subset of the entire dataset. For this, we can take the output from searching an inverted index and run (e.g.) topic modeling algorithms on the relevant documents.

Thus, the analysis operators such as **Select** and **Split** take META's `index` objects as input and return objects such as a `sequence`. The algorithms that run for **Select** could be a filter or search engine, and **Split** could be `meta::topics::lda_model` or some other clustering algorithm.

Bayesian Statistics

Here, we examine Bayesian statistics in more depth as a continuation of Chapter 2 and Chapter 17.

A.1 Binomial Estimation and the Beta Distribution

From section 2.1.5, we already know the likelihood of our binomial distribution is

$$p(D \mid \theta) = \theta^H (1 - \theta)^T, \tag{A.1}$$

but what about the prior, $p(\theta)$? A prior should represent some "prior belief" about the parameters of the model. For our coin flipping (i.e., binomial distribution), it would make sense to have the prior also be proportional to the powers of θ and $(1 - \theta)$. Thus, the posterior will also be proportional to those powers:

$$p(\theta \mid D) \propto p(\theta)p(D \mid \theta)$$
$$= \theta^a (1 - \theta)^b \theta^H (1 - \theta)^T$$
$$= \theta^{a+H} (1 - \theta)^{b+T}.$$

So we need to find some distribution of the form $P(\theta) \propto \theta^a (1 - \theta)^b$. Luckily, there is something called the Beta distribution. We say $x \sim Beta(\alpha, \beta)$ if for $x \in [0, 1]$

$$p(x \mid \alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1 - x)^{\beta-1}. \tag{A.2}$$

This is the probability density function (pdf) of the Beta distribution. But what is

$$\frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)}? \tag{A.3}$$

The $\Gamma(\cdot)$ is the Gamma function. It can be thought of as the continuous version of the factorial function. That is, $\Gamma(x) = (x - 1)\Gamma(x - 1)$. Or rather for an $x \in \mathbb{Z}^+$, $\Gamma(x) = (x - 1)!$. That still doesn't explain the purpose of that constant in front of $x^{\alpha-1}(1 - x)^{\beta-1}$.

In fact, this constant just ensures that given the α and β parameters, the Beta distribution still integrates to one over its support. As you probably recall, this is a necessity for a probability distribution. Mathematically, we can write this as

$$\int_0^1 x^{\alpha-1}(1-x)^{\beta-1}dx = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}. \tag{A.4}$$

Note that the sum over the support of x is the reciprocal of that constant. If we divide by it (multiply by reciprocal), we will get one as desired:

$$\int_0^1 \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}dx = 1. \tag{A.5}$$

If you're proficient in calculus (or know how to use Wolfram Alpha or similar), you can confirm this fact for yourself.

One more note on the Beta distribution: its expected value is

$$\frac{\alpha}{\alpha+\beta}. \tag{A.6}$$

We'll see how this can be useful in a minute. Let's finally rewrite our estimate of $p(\theta \mid D)$. The data we have observed is H, T. Additionally, we are using the two *hyperparameters* α and β for our Beta distribution prior. They're called hyperparameters because they are parameters for our prior distribution.

$$p(\theta \mid H, T, \alpha, \beta) \propto p(H, T \mid \theta)p(\theta \mid \alpha, \beta)$$
$$\propto \theta^H(1-\theta)^T\theta^{\alpha-1}(1-\theta)^{\beta-1}$$
$$= \theta^{H+\alpha-1}(1-\theta)^{T+\beta-1}.$$

But this is itself a Beta distribution! Namely,

$$p(\theta \mid H, T, \alpha, \beta) = Beta(H+\alpha, T+\beta). \tag{A.7}$$

Finally, we can get our Bayesian parameter estimation. Unlike maximum likelihood estimation (MLE), where we have the parameter that maximizes our data, we integrate over all possible θ, and find its expected value given the data, $E[\theta \mid D]$. In this case, our "data", is the flip results and our hyperparameters α and β:

$$E[\theta \mid D] = \int_0^1 p(x = H \mid \theta)p(\theta \mid D)d\theta = \frac{H+\alpha}{H+T+\alpha+\beta}. \tag{A.8}$$

We won't go into detail with solving the integral since that isn't our focus. What we do see, though, is our final result. This result is general for any Bayesian estimate of a binomial parameter with a Beta prior.
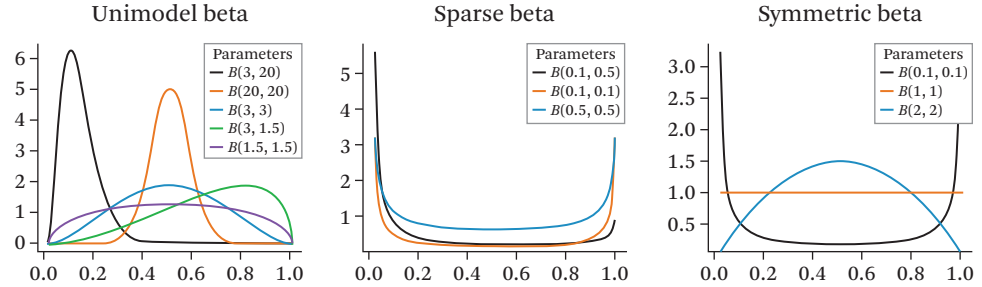
Figure A.1   How the α and β parameters affect the shape of the Beta distribution. Shown from left to right are the unimodal Beta $(\alpha, \beta > 1)$, the sparse Beta $(\alpha, \beta < 1)$, and the symmetric Beta $(\alpha = \beta)$.

A.2 Pseudo Counts, Smoothing, and Setting Hyperparameters

How can we interpret our result for a Bayesian estimate of binomial parameters?

$$E[\theta \mid D] = \frac{H + \alpha}{H + T + \alpha + \beta} \tag{A.9}$$

We know that the Beta and binomial distributions are similar. In fact, their relationship can be stated as *the Beta distribution is the conjugate prior of the binomial distribution*. All distributions in the exponential family have conjugate priors. The relationship is such: given a likelihood from an X distribution, picking the conjugate prior distribution of X (say it's Y) will ensure that the posterior distribution is also a Y distribution.

For our coin flipping case, the likelihood was a binomial distribution. We picked our prior to be the Beta distribution, and our posterior distribution ended up also being a Beta distribution—this is because we picked the conjugate prior!

In any event, the whole reasoning behind having a prior is so we can include some reasonable guess for the parameters before we even see any data. For coin flipping, we might want to assume a "fair" coin. If for some reason we believe that the coin may be biased, we can incorporate that knowledge as well.

If we look at the estimate for θ, we can imagine how setting our hyperparameters can influence our prediction. Recall that θ is the probability of heads; if we want to make our estimate biased toward more heads, we can set an $\alpha > \beta$ since this increases θ. This agrees with the mean of the prior as well: $\frac{\alpha}{\alpha+\beta}$. Setting the mean equal to 0.8 means that our prior belief is a coin that lands heads 80% of the time.

This can be accomplished with $\alpha = 4$, $\beta = 1$, or $\alpha = 16$, $\beta = 4$, or even $\alpha = 0.4$, $\beta = 0.1$. But what is the difference? Figure A.1 shows a comparison of the Beta distribution with varying parameters. It's also important to remember that a draw from a Beta prior $\theta \sim \text{Beta}(\alpha, \beta)$ gives us a *distribution*. Even though it's a single value on the range $[0, 1]$, we are still using the prior to produce a probability distribution.

Perhaps we'd like to choose a unimodal Beta prior, with a mean 0.8. As we can see from Figure A.1, the higher we set α and β, the sharper the peak at 0.8 will be. Looking at our parameter estimation,

$$\frac{H + \alpha}{H + T + \alpha + \beta},\tag{A.10}$$

we can imagine the hyperparameters as pseudo counts—counts from the outcome of experiments already performed. The higher the hyperparameters are, the more pseudo counts we have, which means our prior is "stronger." As the total number of experiments increases, the sum $H + T$ also increases, which means we have less dependence on our priors.

Initially, though, when $H + T$ is relatively low, our prior plays a stronger role in the estimation of θ. As we all know, a small number of flips will not give an accurate estimate of the true θ—we'd like to see what our estimate becomes as our number of flips approaches infinity (or some "large enough" value). In this sense, our prior also *smooths* our estimation. Rather than the estimate fluctuating greatly initially, it could stay relatively smooth if we have a decent prior.

If our prior turns out to be incorrect, eventually the observed data will overshadow the pseudo counts from the hyperparameters anyway, since α and β are held constant.

A.3 Generalizing to a Multinomial Distribution

At this point, you may be able to rationalize how Dirichlet prior smoothing for information retrieval language models or topic models works. However, our probabilities are over words now, not just a binary heads or tails outcome. Before we talk about the Dirichlet distribution, let's figure out how to represent the probability of observing a word from a vocabulary.

For this, we can use a categorical distribution. In a text information system, a categorical distribution could represent a unigram language model for a single document. Here, the total number of outcomes is $k = |V|$, the size of our vocabulary. The word at index i would have a probability p_i of occurring, and the sum of all words' probabilities would sum to one.

The categorical distribution is to the multinomial distribution as the Bernoulli is to the binomial. The multinomial is the probability of observing each word k_i occur x_i times in a total of n trials. If we're given a document vector of counts, we can use the multinomial to find the probability of observing documents with those counts of words (regardless of position). The probability density function is given as follows:

$$p(X_1 = x_i, \ldots, X_k = x_k) = \frac{n!}{x_1! \cdots x_k!} p_1^{x_1} \cdots p_k^{x_k}, \tag{A.11}$$

where

$$\sum_{i=1}^{k} x_i = n, \qquad \sum_{i=1}^{k} p_i = 1.$$

We can also write its pdf as

$$p(X_1 = x_i, \ldots, X_k = x_k) = \frac{\Gamma\left(\sum_{i=1}^{k} x_i + 1\right)}{\prod_{i=1}^{k} \Gamma(x_i + 1)} \prod_{i=1}^{k} p_i^{x_i}. \tag{A.12}$$

It should be straightforward to relate the more general multinomial distribution to its binomial counterpart.

A.4 The Dirichlet Distribution

We now have the likelihood function determined for a distribution with k outcomes. The conjugate prior to the multinomial is the Dirichlet. That is, if we use a Dirichlet prior, the posterior will also be a Dirichlet.

Like the multinomial, the Dirichlet is a distribution over positive vectors that sum to one. (The "simplex" is the name of the space where these vectors live.) Like the Beta distribution, the parameters of the Dirichlet are reals. Here's the pdf:

$$p(\theta \mid \vec{\alpha}) = \frac{\Gamma\left(\sum_{i=1}^{k} \alpha_i\right)}{\prod_{i=1}^{k} \Gamma(\alpha_i)} \prod_{i=1}^{k} \theta_i^{\alpha_i - 1}. \tag{A.13}$$

In this notation we have $p(\theta \mid \vec{\alpha})$. θ is what we draw from the Dirichlet; in the Beta, it was the parameter to be used in the binomial. Here, it is the vector of parameters to be used in the multinomial. In this sense, the Dirichlet is a distribution that produces distributions (so is the Beta!). The hyperparameters of the Dirichlet are also a vector (denoted with an arrow for emphasis). Instead of just two hyperparameters as in the Beta, the Dirichlet needs k—one for each multinomial probability.
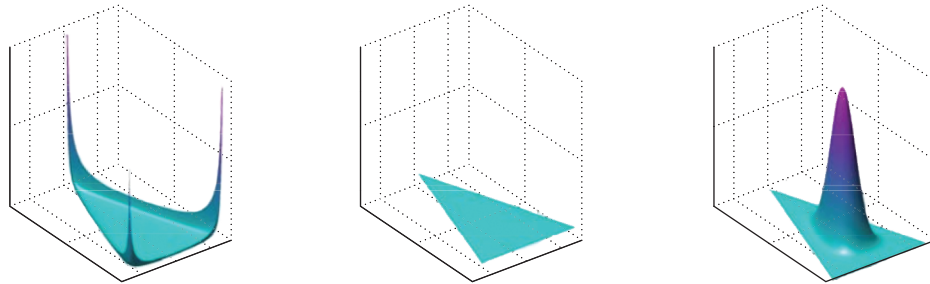
Figure A.2 **How the α parameter affects the shape and sparsity of the Dirichlet distribution of three parameters. From left to right, $\alpha = 0.1, 1, 10$.** (From Bishop [2006])

When used as a prior, we usually don't have any specific information about the individual indices in the Dirichlet. Because of this, we set them all to the same value. So instead of writing $p(\theta \mid \vec{\alpha})$, where $\vec{\alpha}$ is (e.g.) $\{0.1, 0.1, \ldots, 0.1\}$, we simply say $p(\theta \mid \alpha)$, where alpha is a scalar representing a vector of identical values. $\theta \sim Dir(\alpha)$ and $\theta \sim Dir(0.1)$ are also commonplace, as is $\theta \sim \text{Beta}(\alpha, \beta)$ or $\theta \sim \text{Beta}(0.4, 0.1)$.

Figure A.2 shows how the choice of α characterizes the Dirichlet. The higher the area, the more likely a point (representing a vector) will be drawn.

Let's take a moment to understand what a point drawn from the Dirichlet means. Look at the far right graph in Figure A.2. If the point we draw is from the peak of the graph, we'll get a multinomial parameter vector with a roughly equal proportion of each of the three components. For example, $\theta = \{0.33, 0.33, 0.34\}$. With $\alpha = 10$, it's very likely that we'll get a θ like this. In the middle picture, we're unsure what kind of θ we'll draw. It is equally likely to get an even mixture, uneven mixture, or anywhere in between. This is called a uniform prior—it can represent that we have no explicit information about the prior distribution. Finally, the plot on the left is a sparse prior (like a Beta where $\alpha, \beta < 1$).

Note: a uniform prior does *not* mean that we get an even mixture of components; it means it's equally likely to get *any* mixture. This could be confusing since the distribution we draw may actually be a uniform distribution.

A sparse prior is actually quite relevant in a textual application; if we have a few dimensions with very high probability and the rest with relatively low occurrences, this should sound just like Zipf's law. We can use a Dirichlet prior to enforce a sparse word distribution per topic ($\theta = \{0.9, 0.02, 0.08\}$). In topic modeling, we can use a Dirichlet distribution to force a sparse topic distribution per document. It's most likely that a document mainly discusses a handful of topics while the rest are

largely unrepresented, just like the words *the*, *to*, *of*, and *from* are common while many words such as *preternatural* are rare.

A.5 Bayesian Estimate of Multinomial Parameters

Let's do parameter estimation with our multinomial distribution and relate it to the Beta-binomial model from before. For MLE, we would have

$$\theta_i = \frac{x_i}{n}. \tag{A.14}$$

Using Bayes' rule, we represent the posterior as the product of the likelihood (multinomial) and prior (Dirichlet):

$$p(\theta \mid D, \alpha) \propto p(D \mid \theta) p(\theta \mid \alpha)$$

$$\propto \prod_{i=1}^{k} \theta_i^{x_i} \prod_{i=1}^{k} \theta_i^{\alpha_i - 1}$$

$$= \prod_{i=1}^{k} \theta_i^{x_i + \alpha_i - 1}.$$

We say these are proportional because we left out the constant of proportionality in the multinomial and Dirichlet distributions (the ratio with Gammas). We can now observe that the posterior is also a Dirichlet as expected due to the conjugacy.

To actually obtain the Bayesian estimate, we'd need to fully substitute the multinomial and Dirichlet distributions into the posterior and integrate over all θs to get our estimate. Since this isn't a note on calculus, we simply display the final answer as

$$E[\theta_i \mid D] = \frac{x_i + \alpha_i}{n + \sum_{j=1}^{k} \alpha_j}. \tag{A.15}$$

This looks very similar to the binomial estimation! We see the Dirichlet hyperparameters act as pseudo counts, smoothing our estimate.

In Dirichlet prior smoothing for information retrieval, we have the formula:

$$p(w \mid d) = \frac{c(w, d) + \mu p(w \mid C)}{|d| + \mu}. \tag{A.16}$$

So we have $x = c(w, d)$ and $n = |d|$, the count of the current word in a document and the length of the document respectively. Then we have $\alpha_i = \mu p(w \mid C)$ and $\mu = \sum_{j=1}^{k} \alpha_j$, the number of pseudo counts for word w and the total number of pseudo counts. Can you tell what the vector of hyperparameters for query likelihood

smoothing would be now? It's

$$\vec{\mu} = \big\{ \mu p(w_1 \mid C), \mu p(w_2 \mid C), \dots, \mu p(w_k \mid C) \big\} \tag{A.17}$$

In other words, the Dirichlet prior for this smoothing method is proportional to the background, collection language model.

Looking back at Add-1 smoothing, we can imagine this as a special case of Dirichlet prior smoothing. If we drew the uniform distribution from our Dirichlet, we'd get

$$p(w \mid d) = \frac{c(w, d) + 1}{|d| + |V|} \tag{A.18}$$

This implies that each word is equally likely in our collection language model, which is most likely not the case. Note $|V| = k$, or $\sum_{i=1}^{k} 1$ since $\vec{\mu} = \{1, 1, \dots, 1\}$.

A.6 Conclusion

Starting with the Bernoulli distribution for a single coin flip, we expanded it into a set of trials with the binomial distribution. We investigated parameter estimation via MLE, and then moved onto a Bayesian approach. We compared the Bayesian result to smoothing with pseudo counts and saw how hyperparameters affected the distribution. Once we had this foundation, we moved onto multidimensional distributions capable of representing individual words. We saw how the Beta-binomial model is related to the Dirichlet-multinomial model, and inspected it in the context of Dirichlet prior smoothing for query likelihood in IR.

Expectation-Maximization

The Expectation-Maximization (EM) algorithm is a general algorithm for maximum-likelihood estimation where the data are "incomplete" or the likelihood function involves latent variables. Note that the notion of "incomplete data" and "latent variables" are related: when we have a latent variable, we may regard our data as being incomplete since we do not observe values of the latent variables; similarly, when our data are incomplete, we often can also associate some latent variable with the missing data. For language modeling, the EM algorithm is often used to estimate parameters of a mixture model, in which the exact component model from which a data point is generated is hidden from us.

Informally, the EM algorithm starts with randomly assigning values to all the parameters to be estimated. It then iteratively alternates between two steps, called the expectation step (i.e., the "E-step") and the maximization step (i.e., the "M-step"), respectively. In the E-step, it computes the expected likelihood for the complete data (the so-called Q-function) where the expectation is taken with respect to the computed conditional distribution of the latent variables (i.e., the "hidden variables") given the current settings of parameters and our observed (incomplete) data. In the M-step, it re-estimates all the parameters by maximizing the Q-function. Once we have a new generation of parameter values, we can repeat the E-step and another M-step. This process continues until the likelihood converges, reaching a local maxima. Intuitively, what EM does is to iteratively augment the data by "guessing" the values of the hidden variables and to re-estimate the parameters by assuming that the guessed values are the true values.

The EM algorithm is a hill-climbing approach, thus it can only be guaranteed to reach a local maxima. When there are multiple maximas, whether we will actually reach the global maxima depends on where we start; if we start at the "right hill," we will be able to find a global maxima. When there are multiple local maximas, it is often hard to identify the "right hill." There are two commonly used strategies

to solving this problem. The first is that we try many different initial values and choose the solution that has the highest converged likelihood value. The second uses a much simpler model (ideally one with a unique global maxima) to determine an initial value for more complex models. The idea is that a simpler model can hopefully help locate a rough region where the global optima exists, and we start from a value in that region to search for a more accurate optima using a more complex model.

Here, we introduce the EM algorithm through a specific problem—estimating a simple mixture model. For a more in-depth introduction to EM, please refer to McLachlan and Krishnan [2008].

B.1 A Simple Mixture Unigram Language Model

In the mixture model feedback approach [Zhai and Lafferty 2001], we assume that the feedback documents $\mathcal{F} = \{d_1, \ldots, d_k\}$ are "generated" from a mixture model with two multinomial component models. One component model is the background model $p(w \mid C)$ and the other is an unknown topic language model $p(w \mid \theta_F)$ to be estimated. (w is a word.) The idea is to model the common (non-discriminative) words in $\mathcal{F}$ with $p(w \mid C)$ so that the topic model θ_F would attract more discriminative content-carrying words.

The log-likelihood of the feedback document data for this mixture model is

$$\log L(\theta_F) = \log p(\mathcal{F} \mid \theta_F) = \sum_{i=1}^{k} \sum_{j=1}^{|d_i|} \log((1 - \lambda) p(d_{ij} \mid \theta_F) + \lambda p(d_{ij} \mid C)),$$

where d_{ij} is the jth word in document d_i, $|d_i|$ is the length of d_i, and λ is a parameter that indicates the amount of "background noise" in the feedback documents, which will be set empirically. We thus assume λ to be known, and want to estimate $p(w \mid \theta_F)$.

B.2 Maximum Likelihood Estimation

A common method for estimating θ_F is the maximum likelihood (ML) estimator, in which we choose a θ_F that maximizes the likelihood of $\mathcal{F}$. That is, the estimated topic model (denoted by $\hat{\theta}_F$) is given by

$$\hat{\theta}_F = \arg \max_{\theta_F} L(\theta_F) \tag{B.1}$$

$$= \arg \max_{\theta_F} \sum_{i=1}^{k} \sum_{j=1}^{|d_i|} \log((1 - \lambda) p(d_{ij} \mid \theta_F) + \lambda p(d_{ij} \mid C)). \tag{B.2}$$

The right side of this equation is easily seen to be a function with $p(w \mid \theta_F)$ as variables. To find $\hat{\theta}_F$, we can, in principle, use any optimization methods. Since the function involves a logarithm of a sum of two terms, it is difficult to obtain a simple analytical solution via the Lagrange Multiplier approach, so in general, we must rely on numerical algorithms. There are many possibilities; EM happens to be just one of them which is quite natural and guaranteed to converge to a local maxima, which, in our case, is also a global maximum, since the likelihood function can be shown to have one unique maximum.

B.3 Incomplete vs. Complete Data

The main idea of the EM algorithm is to "augment" our data with some latent variables so that the "complete" data has a much simpler likelihood function—simpler for the purpose of finding a maximum. The original data are thus treated as "incomplete." As we will see, we will maximize the incomplete data likelihood (our original goal) through maximizing the expected complete data likelihood (since it is much easier to maximize) where expectation is taken over all possible values of the hidden variables (since the complete data likelihood, unlike our original incomplete data likelihood, would contain hidden variables).

In our example, we introduce a binary hidden variable z for each *occurrence* of a word w to indicate whether the word has been "generated" from the background model $p(w \mid C)$ or the topic model $p(w \mid \theta_F)$. Let d_{ij} be the jth word in document d_i. We have a corresponding variable z_{ij} defined as follows:

$$z_{ij} = \begin{cases} 1 & \text{if word } d_{ij} \text{ is from background} \\ 0 & \text{otherwise.} \end{cases}$$

We thus assume that our complete data would have contained not only all the words in $\mathcal{F}$, but also their corresponding values of z. The log-likelihood of the complete data is thus

$$L_c(\theta_F) = \log\, p(\mathcal{F}, \mathbf{z} \mid \theta_F)$$

$$= \sum_{i=1}^{k} \sum_{j=1}^{|d_i|} [(1 - z_{ij}) \log((1 - \lambda)p(d_{ij} \mid \theta_F)) + z_{ij} \log(\lambda p(d_{ij} \mid C))].$$

Note the difference between $L_c(\theta_F)$ and $L(\theta_F)$: the sum is outside of the logarithm in $L_c(\theta_F)$, and this is possible because we assume that we *know* which component model has been used to generated each word d_{ij}.

What is the relationship between $L_c(\theta_F)$ and $L(\theta_F)$? In general, if our parameter is θ, our original data is X, and we augment it with a hidden variable H, then

$p(X, H \mid \theta) = p(H \mid X, \theta) p(X \mid \theta)$. Thus,

$$L_c(\theta) = \log p(X, H \mid \theta) = \log p(X \mid \theta) + \log p(H \mid X, \theta) = L(\theta) + \log p(H \mid X, \theta).$$

B.4   A Lower Bound of Likelihood

Algorithmically, the basic idea of EM is to start with some initial guess of the parameter values $\theta^{(0)}$ and then iteratively search for better values for the parameters. Assuming that the current estimate of the parameters is $\theta^{(n)}$, our goal is to find another $\theta^{(n+1)}$ that can improve the likelihood $L(\theta)$.

Let us consider the difference between the likelihood at a potentially better parameter value θ and the likelihood at the current estimate $\theta^{(n)}$, and relate it with the corresponding difference in the complete likelihood:

$$L(\theta) - L(\theta^{(n)}) = L_c(\theta) - L_c(\theta^{(n)}) + \log \frac{p(H \mid X, \theta^{(n)})}{p(H \mid X, \theta)}. \tag{B.3}$$

Our goal is to maximize $L(\theta) - L(\theta^{(n)})$, which is equivalent to maximizing $L(\theta)$. Now take the expectation of this equation w.r.t. the conditional distribution of the hidden variable given the data X and the current estimate of parameters $\theta^{(n)}$, i.e., $p(H \mid X, \theta^{(n)})$. We have

$$L(\theta) - L(\theta^{(n)}) = \sum_H L_c(\theta) p(H \mid X, \theta^{(n)}) - \sum_H L_c(\theta^{(n)}) p(H \mid X, \theta^{(n)})$$

$$+ \sum_H p(H \mid X, \theta^{(n)}) \log \frac{p(H \mid X, \theta^{(n)})}{p(H \mid X, \theta)}.$$

Note that the left side of the equation remains the same as the variable H does not occur there. The last term can be recognized as the KL-divergence of $p(H \mid X, \theta^{(n)})$ and $p(H \mid X, \theta)$, which is always non-negative. We thus have

$$L(\theta) - L(\theta^{(n)}) \geq \sum_H L_c(\theta) p(H \mid X, \theta^{(n)}) - \sum_H L_c(\theta^{(n)}) p(H \mid X, \theta^{(n)})$$

or

$$L(\theta) \geq \sum_H L_c(\theta) p(H \mid X, \theta^{(n)}) + L(\theta^{(n)}) - \sum_H L_c(\theta^{(n)}) p(H \mid X, \theta^{(n)}). \tag{B.4}$$

We thus obtain a lower bound for the original likelihood function. The main idea of EM is to maximize this lower bound so as to maximize the original (incomplete) likelihood. Note that the last two terms in this lower bound can be treated as constants as they do not contain the variable θ, so the lower bound is essentially

the first term, which is the expectation of the complete likelihood, or the so-called "Q-function" denoted by $Q(\theta; \theta^{(n)})$.

$$Q(\theta; \theta^{(n)}) = E_{p(H|X,\theta^{(n)})}[L_c(\theta)] = \sum_H L_c(\theta) p(H \mid X, \theta^{(n)}).$$

The Q-function for our mixture model is the following

$$Q(\theta_F; \theta_F^{(n)}) = \sum_{\mathbf{z}} L_c(\theta_F) p(\mathbf{z} \mid \mathcal{F}, \theta_F^{(n)})$$

$$= \sum_{i=1}^{k} \sum_{j=1}^{|d_i|} [p(z_{ij} = 0 \mid \mathcal{F}, \theta_F^{(n)}) \log((1 - \lambda) p(d_{ij} \mid \theta_F))$$

$$+ p(z_{ij} = 1 \mid \mathcal{F}, \theta_F^{(n)}) \log(\lambda p(d_{ij} \mid \mathcal{C}))]$$

B.5 The General Procedure of EM

Clearly, if we find a $\theta^{(n+1)}$ such that $Q(\theta^{(n+1)}; \theta^{(n)}) > Q(\theta^{(n)}; \theta^{(n)})$, then we will also have $L(\theta^{(n+1)}) > L(\theta^{(n)})$. Thus, the general procedure of the EM algorithm is the following.

1. Initialize $\theta^{(0)}$ randomly or heuristically according to any prior knowledge about where the optimal parameter value might be.

2. Iteratively improve the estimate of θ by alternating between the following two-steps:

    1. the E-step (expectation): Compute $Q(\theta; \theta^{(n)})$, and

    2. the M-step (maximization): Re-estimate θ by maximizing the Q-function:

    $$\theta^{(n+1)} = argmax_\theta Q(\theta; \theta^{(n)}).$$

3. Stop when the likelihood $L(\theta)$ converges.

As mentioned earlier, the complete likelihood $L_c(\theta)$ is much easier to maximize as the values of the hidden variable are assumed to be known. This is why the Q-function, which is an expectation of $L_c(\theta)$, is often much easier to maximize than the original likelihood function. In cases when there does not exist a natural latent variable, we often introduce a hidden variable so that the complete likelihood function is easy to maximize.

The major computation to be carried out in the E-step is to compute $p(H \mid X, \theta^{(n)})$, which is sometimes very complicated. In our case, this is simple:

$$p(z_{ij} = 1 \mid \mathcal{F}, \theta_F^{(n)}) = \frac{\lambda p(d_{ij} \mid C)}{\lambda p(d_{ij} \mid C) + (1 - \lambda) p(d_{ij} \mid \theta_F^{(n)})}. \tag{B.5}$$

And, of course, $p(z_{ij} = 0 \mid \mathcal{F}, \theta_F^{(n)}) = 1 - p(z_{ij} = 1 \mid \mathcal{F}, \theta_F^{(n)})$. Note that, in general, z_{ij} may depend on all the words in $\mathcal{F}$. In our model, however, it only depends on the corresponding word d_{ij}.

The M-step involves maximizing the Q-function. This may sometimes be quite complex as well. But, again, in our case, we can find an analytical solution. In order to achieve this, we use the Lagrange multiplier method since we have the following constraint on the parameter variables $\{p(w \mid \theta_F)\}_{w \in V}$, where V is our vocabulary:

$$\sum_{w \in V} p(w \mid \theta_F) = 1.$$

We thus consider the following auxiliary function:

$$g(\theta_F) = Q(\theta_F; \theta_F^{(n)}) + \mu(1 - \sum_{w \in V} p(w \mid \theta_F))$$

and take its derivative with respect to each parameter variable $p(w \mid \theta_F)$

$$\frac{\partial g(\theta_F)}{\partial p(w \mid \theta_F)} = \left[\sum_{i=1}^{k} \sum_{j=1, d_{ij}=w}^{|d_i|} \frac{p(z_{ij} = 0 \mid \mathcal{F}, \theta_F^{(n)})}{p(w \mid \theta_F)} \right] - \mu. \tag{B.6}$$

Setting this derivative to zero and solving the equation for $p(w \mid \theta_F)$, we obtain

$$p(w \mid \theta_F) = \frac{\sum_{i=1}^{k} \sum_{j=1, d_{ij}=w}^{|d_i|} p(z_{ij} = 0 \mid \mathcal{F}, \theta_F^{(n)})}{\sum_{i=1}^{k} \sum_{j=1}^{|d_i|} p(z_{ij} = 0 \mid \mathcal{F}, \theta_F^{(n)})} \tag{B.7}$$

$$= \frac{\sum_{i=1}^{k} p(z_w = 0 \mid \mathcal{F}, \theta_F^{(n)}) c(w, d_i)}{\sum_{i=1}^{k} \sum_{w' \in V} p(z_{w'} = 0 \mid \mathcal{F}, \theta_F^{(n)}) c(w', d_i)}. \tag{B.8}$$

Note that we changed the notation so that the sum over each word position in document d_i is now a sum over all the distinct words in the vocabulary. This is possible, because $p(z_{ij} \mid \mathcal{F}, \theta_F^{(n)})$ depends only on the corresponding word d_{ij}. Using word w, rather then the word *occurrence* d_{ij}, to index z, we have

$$p(z_w = 1 \mid \mathcal{F}, \theta_F^{(n)}) = \frac{\lambda p(w \mid C)}{\lambda p(w \mid C) + (1 - \lambda) p(w \mid \theta_F^{(n)})}. \tag{B.9}$$

We therefore have the following EM updating formulas for our simple mixture model:

$$p(z_w = 1 \mid \mathcal{F}, \theta_F^{(n)}) = \frac{\lambda p(w \mid C)}{\lambda p(w \mid C) + (1 - \lambda) p(w \mid \theta_F^{(n)})} \quad \text{(E)} \qquad \text{(B.10)}$$

$$p(w \mid \theta_F^{(n+1)}) = \frac{\sum_{i=1}^{k}(1 - p(z_w = 1 \mid \mathcal{F}, \theta_F^{(n)}))c(w, d_i)}{\sum_{i=1}^{k} \sum_{w' \in V}(1 - p(z_{w'} = 1 \mid \mathcal{F}, \theta_F^{(n)})c(w', d_i))} \quad \text{(M). (B.11)}$$

Note that we never need to *explicitly* compute the Q-function; instead, we compute the distribution of the hidden variable z and then directly obtain the new parameter values that will maximize the Q-function.

KL-divergence and Dirichlet Prior Smoothing

This appendix is a more detailed discussion of the KL-divergence function and its relation to Dirichlet prior smoothing in the generalized query likelihood smoothing framework. We briefly touched upon KL-divergence in Chapter 7 and Chapter 13.

As we have seen, given two probability mass functions $p(x)$ and $q(x)$, $D(p\|q)$, the Kullback-Leibler divergence (or relative entropy) between p and q is defined as

$$D(p\|q) = \sum_x p(x) \log \frac{p(x)}{q(x)}.$$

It is easy to show that $D(p\|q)$ is always non-negative and is zero if and only if $p = q$. Even though it is not a true distance between distributions (because it is not symmetric and does not satisfy the triangle inequality), it is still often useful to think of the KL-divergence as a "distance" between distributions [Cover and Thomas 1991].

C.1 Using KL-divergence for Retrieval

Suppose that a query **q** is generated by a generative model $p(\mathbf{q} \mid \theta_Q)$ with θ_Q denoting the parameters of the query unigram language model. Similarly, assume that a document **d** is generated by a generative model $p(\mathbf{d} \mid \theta_D)$ with θ_D denoting the parameters of the document unigram language model. If $\widehat{\theta}_Q$ and $\widehat{\theta}_D$ are the estimated query and document language models, respectively, then, the relevance value of **d** with respect to **q** can be measured by the following *negative* KL-divergence function [Zhai and Lafferty 2001]:

$$-D(\widehat{\theta}_Q\|\widehat{\theta}_D) = \sum_w p(w \mid \widehat{\theta}_Q) \log p(w \mid \widehat{\theta}_D) + \left(- \sum_w p(w \mid \widehat{\theta}_Q) \log p(w \mid \widehat{\theta}_Q)\right).$$

Note that the second term on the right-hand side of the formula is a query-dependent constant, or more specifically, the entropy of the query model $\widehat{\theta}_Q$. It can be ignored for the purpose of ranking documents. In general, the computation of the above formula involves a sum over all the words that have a non-zero probability according to $p(w \mid \widehat{\theta}_Q)$. However, when $\widehat{\theta}_D$ is based on certain general smoothing method, the computation would only involve a sum over those that both have a non-zero probability according to $p(w \mid \widehat{\theta}_Q)$ and occur in document **d**. Such a sum can be computed much more efficiently with an inverted index.

We now explain this in detail. The general smoothing scheme we assume is the following:

$$
p(w \mid \widehat{\theta}_D) = \begin{cases} p_s(w \mid \mathbf{d}) & \text{if word } w \text{ is seen} \\ \alpha_d p(w \mid \mathcal{C}) & \text{otherwise.} \end{cases}
$$

where $p_s(w \mid \mathbf{d})$ is the smoothed probability of a word seen in the document, $p(w \mid \mathcal{C})$ is the collection language model, and α_d is a coefficient controlling the probability mass assigned to unseen words, so that all probabilities sum to one. In general, α_d may depend on d. Indeed, if $p_s(w \mid \mathbf{d})$ is given, we must have

$$
\alpha = \frac{1 - \sum_{w:c(w;d)>0} p_s(w \mid d)}{1 - \sum_{w:c(w;d)>0} p(w \mid \mathcal{C})}.
$$

Thus, individual smoothing methods essentially differ in their choice of $p_s(w \mid \mathbf{d})$.

The collection language model $p(w \mid \mathcal{C})$ is typically estimated by $\frac{c(w,\mathcal{C})}{\sum_{w'} c(w',\mathcal{C})}$, or a smoothed version $\frac{c(w,\mathcal{C})+1}{V+\sum_{w'} c(w',\mathcal{C})}$, where V is an estimated vocabulary size (e.g., the total number of distinct words in the collection). One advantage of the smoothed version is that it would never give a zero probability to any term, but in terms of retrieval performance, there will not be any significant difference in these two versions, since $\sum_{w'} c(w', \mathcal{C})$ is often significantly larger than V.

It can be shown that with such a smoothing scheme, the KL-divergence scoring formula is essentially (the two sides are equivalent for ranking documents)

$$
\sum_w p(w \mid \widehat{\theta}_Q) \log p(w \mid \widehat{\theta}_D) = \left[\sum_{w:c(w;d)>0,\, p(w|\widehat{\theta}_Q)>0} p(w \mid \widehat{\theta}_Q) \log \frac{p_s(w \mid \mathbf{d})}{\alpha_d p(w \mid \mathcal{C})} \right]
$$

$$
+ \log \alpha_d. \tag{C.1}
$$

Note that the scoring is now based on a sum over all the terms that both have a non-zero probability according to $p(w \mid \widehat{\theta}_Q)$ and occur in the document, i.e., all "matched" terms.

C.2 Using Dirichlet Prior Smoothing

Dirichlet prior smoothing is one particular smoothing method that follows the general smoothing scheme mentioned in the previous section. In particular,

$$p_s(w \mid \mathbf{d}) = \frac{c(w, d) + \mu p(w \mid \mathcal{C})}{|d| + \mu}$$

and

$$\alpha_d = \frac{\mu}{\mu + |d|}.$$

Plugging these into equation C.1, we see that with Dirichlet prior smoothing, our KL-divergence scoring formula is

$$\left[\sum_{w : c(w;d) > 0, \, p(w|\widehat{\theta}_Q) > 0} p(w \mid \widehat{\theta}_Q) \log(1 + \frac{c(w, \mathbf{d})}{\mu p(w \mid \mathcal{C})}) \right] + \log \frac{\mu}{\mu + |d|}.$$

C.3 Computing the Query Model $p(w \mid \widehat{\theta}_Q)$

You may be wondering how we can compute $p(w \mid \widehat{\theta}_Q)$. This is exactly where the KL-divergence retrieval method is *better* than the simple query likelihood method— we can have *different* ways of computing it! The simplest way is to estimate this probability by the maximum likelihood estimator using the query text as evidence, which gives us

$$p_{ml}(w \mid \widehat{\theta}_Q) = \frac{c(w, \mathbf{q})}{|q|}.$$

Using this estimated value, you should see easily that the KL-divergence scoring formula is essentially the same as the query likelihood retrieval formula as presented in Zhai and Lafferty [2004].

A more interesting way of computing $p(w \mid \widehat{\theta}_Q)$ is to exploit feedback documents. Specifically, we can interpolate the simple $p_{ml}(w \mid \widehat{\theta}_Q)$ with a *feedback model* $p(w \mid \theta_F)$ estimated based on feedback documents. That is,

$$p(w \mid \widehat{\theta}_Q) = (1 - \alpha) p_{ml}(w \mid \widehat{\theta}_Q) + \alpha p(w \mid \theta_F), \tag{C.2}$$

where α is a parameter that needs to be set empirically. Please note that this α is *different* from α_d in the smoothing formula.

Of course, the next question is how to estimate $p(w \mid \theta_F)$? One approach is to assume the following two component mixture model for the feedback documents,

where one component model is $p(w \mid \theta_F)$ and the other is $p(w \mid C)$, the collection language model.

$$\log p(\mathcal{F} \mid \theta_F) = \sum_{i=1}^{k} \sum_{w} c(w; d_i) \log((1 - \lambda) p(w \mid \theta_F) + \lambda p(w \mid C)),$$

where $F = \{d_1, \ldots, d_k\}$ is the set of feedback documents, and λ is yet another parameter that indicates the amount of "background noise" in the feedback documents, and that needs to be set empirically. Now, given λ, the feedback documents $\mathcal{F}$, and the collection language model $p(w \mid C)$, we can use the EM algorithm to compute the maximum likelihood estimate of θ_F, as detailed in Appendix B.

References

C. C. Aggarwal. 2015. *Data Mining - The Textbook*. Springer. DOI: 10.1007/978-3-319-14142-8. 296

C. C. Aggarwal and C. Zhai, editors. 2012. *Mining Text Data*. Springer. DOI: 10.1007/978-1-4614-3223-4. 296, 315

J. Allen. 1995. *Natural Language Understanding*. 2nd ed. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA. 54

G. Amati and C. J. Van Rijsbergen. October 2002. Probabilistic models of information retrieval based on measuring the divergence from randomness. *ACM Trans. Inf. Syst.*, 20(4):357–389. DOI: 10.1145/582415.582416. 87, 88, 90, 111

A. U. Asuncion, M. Welling, P. Smyth, and Y. W. Teh. 2009. On smoothing and inference for topic models. In *UAI 2009, Proc. of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence, Montreal, QC, Canada*, June 18-21, 2009, pp. 27–34. 385

R. A. Baeza-Yates and B. A. Ribeiro-Neto. 2011. *Modern Information Retrieval - the concepts and technology behind search*. 2nd ed. Pearson Education Ltd., Harlow, UK. http://www.mir2ed.org/. xvii, 18, 19

Y. Bar-Hillel, *The Present Status of Automatic Translation of Languages*, in Advances in Computers, vol. 1 (1960), pp. 91–163.

R. Belew. 2008. *Finding Out About: A Cognitive Perspective on Search Engine Technology and the WWW*. Cambridge University Press. 18

N. J. Belkin and W. B. Croft. 1992. Information filtering and information retrieval: Two sides of the same coin? *Commun. ACM*, 35(12):29–38. DOI: 10.1145/138859.138861. 84

C. M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ. 19, 37, 312, 385, 462

D. M. Blei, A. Y. Ng, and M. I. Jordan. March 2003. Latent Dirichlet Allocation. *J. of Mach. Learn. Res.*, 3:993–1022. 385

J. S. Breese, D. Heckerman, and C. Kadie. 1998. Empirical analysis of predictive algorithms for collaborative filtering. In *Proc. of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, UAI'98, Morgan Kaufmann Publishers Inc. pp. 43–52, San Francisco, CA. http://dl.acm.org/citation.cfm?id=2074094.2074100. 235

P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. Della Pietra, and J. C. Lai. 1992. Class-based N-gram Models of Natural Language. *Comput. Linguist.*, 18(4):467–479. 273, 288, 290, 291

C. Buckley. 1994. Automatic query expansion using smart: Trec 3. In *Proc. of The third Text REtrieval Conference (TREC-3*, pp. 69–80. 144

S. Büttcher, C. Clarke, and G. V. Cormack. 2010. *Information Retrieval: Implementing and Evaluating Search Engines*. The MIT Press. xvii, 18, 165

F. Cacheda, V. Carneiro, D. Fernández, and V. Formoso. 2011. Comparison of collaborative filtering algorithms: Limitations of current techniques and proposals for scalable, high-performance recommender systems. *ACM Trans. Web*, 5(1):2:1–2:33. DOI: 10.1145/1921591.1921593. 235

C. Campbell and Y. Ying. 2011. *Learning with Support Vector Machines*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers. DOI: 10.2200/S00324ED1V01Y201102AIM010. 311

J. Carbonell and J. Goldstein. 1998. The Use of MMR, Diversity-based Reranking for Reordering Documents and Producing Summaries. In *Proc. of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '98,ACM, pp. 335–336, New York. DOI: doi=10.1.1.188.3982 321, 327

C.-C. Chang and C.-J. Lin. 2011. LIBSVM: A Library for Support Vector Machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27. 58

J. Chang, S. Gerrish, C. Wang, J. L. Boyd-graber, and D. M. Blei. 2009. Reading Tea Leaves: How Humans Interpret Topic Models. In Y. Bengio, D. Schuurmans, J.D. Lafferty, C.K.I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, Curran Associates, Inc. 22, pp. 288–296. 272, 383, 384, 385, 410

K. W. Church and P. Hanks. 1990. Word association norms, mutual information, and lexicography. *Comput. Linguist.*, 16(1):22–29. http://dl.acm.org/citation.cfm?id=89086.89095. 273

T. Cover and J. Thomas. 1991. *Elements of Information Theory*. New York: Wiley. DOI: DOI: 10.1002/047174882X 37, 473

B. Croft, D. Metzler, and T. Strohman. 2009. *Search Engines: Information Retrieval in Practice*, 1st ed., Addison-Wesley Publishing Company. xvii, 18, 165

D. Das and A. F. T. Martins. 2007. A Survey on Automatic Text Summarization. Technical report, Literature Survey for the Language and Statistics II course at Carnegie Mellon University. 318, 321, 327

R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. 2008. LIBLINEAR: A Library for Large Linear Classification. *J. Mach. Learn. Res.*, 9:1871–1874. 58

H. Fang, T. Tao, and C. Zhai. 2004. A formal study of information retrieval heuristics. In *Proc. of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '04, ACM, pp. 49–56, New York. DOI: 10.1145/1008992.1009004. 129

H. Fang, T. Tao, and C. Zhai. April 2011. Diagnostic evaluation of information retrieval models. *ACM Trans. Inf. Syst.*, 29(2):7:1–7:42. DOI: 10.1145/1961209.1961210. 88, 90, 129

R. Feldman and J. Sanger. 2007. *The Text Mining Handbook - Advanced Approaches in Analyzing Unstructured Data*. Cambridge University Press. 18

E. A. Fox, M. A. Gonçalves, and R. Shen. 2012. *Theoretical Foundations for Digital Libraries: The 5S (Societies, Scenarios, Spaces, Structures, Streams) Approach*. Synthesis Lectures on Information Concepts, Retrieval, and Services. Morgan & Claypool Publishers. DOI: 10.2200/S00434ED1V01Y201207ICR022. 80

W. B. Frakes and R. A. Baeza-Yates, editors. 1992. *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 18

K. Ganesan, C. Zhai, and J. Han. 2010. Opinosis: A graph-based approach to abstractive summarization of highly redundant opinions. In *Proc. of the 23rd International Conference on Computational Linguistics*, COLING '10, Association for Computational Linguistics, pp. 340–348, Stroudsburg, PA. 327

K. Ganesan, C. Zhai, and E. Viegas. 2012. Micropinion generation: an unsupervised approach to generating ultra-concise summaries of opinions. In *Proc. of the 21st World Wide Web Conference 2012, WWW 2012*, Lyon, France, April 16-20, 2012, pages 869–878. DOI: 10.1145/2187836.2187954 323

J. Gantz, and D. Reinsel. 2012. The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East, IDC Report, December, 2012. 3

A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, and D. B. Rubin. 1995. *Bayesian Data Analysis*. Chapman & Hall. 37

S. Ghemawat, H. Gobioff, and S.-T. Leung. 2003. *The Google file system*. In *Proc. of the nineteenth ACM symposium on Operating systems principles* (SOSP '03). ACM, New York, 29–43. 195

M. A. Gonçalves, E. A. Fox, L. T. Watson, and N. A. Kipp. 2004. Streams, structures, spaces, scenarios, societies (5s): A formal model for digital libraries. *ACM Trans. Inf. Syst.*, 22(2):270–312. DOI: 10.1145/984321.984325. 84

D. A. Grossman and O. Frieder. Kluwer, 2004. *Information Retrieval - Algorithms and Heuristics, Second Edition*, vol. 15 of *The Kluwer International Series on Information Retrieval*. DOI: 10.1007/978-1-4020-3005-5. 18

G. Hamerly and C. Elkan. 2003. Learning the k in k-means. In *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS December 8-13, 2003*, Vancouver and Whistler, British Columbia, Canada], pp. 281–288. DOI: doi=10.1.1.9.3574 295

J. Han. 2005. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA. 296

D. Harman. 2011. *Information Retrieval Evaluation*. Synthesis Lectures on Information Concepts, Retrieval, and Services. Morgan & Claypool Publishers. DOI: 10.1145/215206.215351 168, 188

M. A. Hearst. 2009. *Search User Interfaces*. 1st ed. Cambridge University Press, New York. 19, 85

J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. 2004. Evaluating Collaborative Filtering Recommender Systems. *ACM Trans. Inf. Syst.*, 22(1):5–53. DOI: 10.1145/963770.963772 235

J. L. Hodges and E. L. Lehmann. 1970. *Basic Concepts of Probability and Statistics*. Holden Day, San Francisco. 36

T. Hofmann. 1999. Probabilistic Latent Semantic Analysis. In *Proc. of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, UAI'99, Morgan Kaufmann Publishers Inc., pp. 289–296, San Francisco, CA. DOI: 10.1145/312624.312649 370, 385

A. Huang. 2008. Similarity Measures for Text Document Clustering. In *Proc. of the Sixth New Zealand Computer Science Research Student Conference (NZCSRSC2008)*, Christchurch, New Zealand, pages 49–56. 280

F. Jelinek. 1997. *Statistical Methods for Speech Recognition*. MIT Press, Cambridge, MA. 30, 54

J. Jiang. 2012. Information extraction from text, In Charu C. Aggarwal and ChengXiang Zhai (Eds.), Mining Text Data, Springer, pp. 11–41. 19, 55

S. Jiang and C. Zhai. 2014. Random walks on adjacency graphs for mining lexical relations from big text data. In *2014 IEEE International Conference on Big Data, Big Data 2014*, Washington, DC, USA, October 27-30, pages 549–554. DOI: 10.1109/BigData.2014.7004272. 273

Y. Jo and A. H. Oh. 2011. Aspect and sentiment unification model for online review analysis. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, ACM, pp. 815–824, New York. DOI: 10.1145/1935826.1935932. 410

T. Joachims, L. Granka, B. Pan, H. Hembrooke, F. Radlinski, and G. Gay. 2007. Evaluating the accuracy of implicit feedback from clicks and query reformulations in web search. *ACM Trans. Inf. Syst.*, 25(2). DOI: 10.1145/1229179.1229181. 144

D. Jurafsky and J. H. Martin. 2009. *Speech and Language Processing*. 2nd ed. Prentice-Hall, Inc., Upper Saddle River, NJ. 19, 54

D. Kelly. 2009. Methods for Evaluating Interactive Information Retrieval Systems with Users. *Foundations and Trends in Information Retrieval*, 3(1-2):1–224. DOI: 10.1561/1500000012 168, 188

D. Kelly and J. Teevan. 2003. Implicit feedback for inferring user preference: A bibliography. *SIGIR Forum*, 37(2):18–28. DOI: 10.1145/959258.959260. 144

H. D. Kim, M. Castellanos, M. Hsu, C. Zhai, T. Rietz, and D. Diermeier. 2013. Mining causal topics in text data: iterative topic modeling with time series feedback. In *Proc. of the 22nd ACM international conference on Conference on information and knowledge management*, CIKM '13, ACM pages 885–890, New York, NY. DOI: 10.1145/2505515.2505612. 435, 438, 439, 440

J. M. Kleinberg. 1999. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632. DOI: 10.1145/324133.324140. 216

J. M. Kleinberg. 2002. An impossibility theorem for clustering. In *Advances in Neural Information Processing Systems 15 [Neural Information Processing Systems, NIPS 2002*, December 9-14, 2002, Vancouver, British Columbia, Canada], pp. 446–453. http://papers.nips.cc/paper/2340-an-impossibility-theorem-for-clustering. 296

D. Koller and N. Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press. 385

J. Lafferty and C. Zhai. 2003. Probabilistic relevance models based on document and query generation. In W. Bruce Croft and John Lafferty, editors, *Language Modeling and Information Retrieval*. Kluwer Academic Publishers. DOI: 10.1007/978-94-017-0171-6_1 87, 113

D. Lin. 1999. Automatic identification of non-compositional phrases. In *Proc. of the 37th Annual Meeting of the Association for Computational Linguistics on Computational Linguistics*, ACL '99, Association for Computational Linguistics, pages 317–324, Stroudsburg, PA. DOI: 10.3115/1034678.1034730. 273, 291

J.Lin and C. Dyer. 2010. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers. DOI: 10.2200/S00274ED1V01Y201006HLT007. 198, 216

Bing Liu. 2012. *Sentiment Analysis and Opinion Mining*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers. DOI: 10.2200/S00416ED1V01Y201204HLT016. 410

T.-Y. Liu. 2009. Learning to rank for information retrieval. *Found. Trends Inf. Retr.*, 3(3):225–331. DOI: 10.1561/1500000016. 216

Y. Lv and C. Zhai. 2009. A comparative study of methods for estimating query language models with pseudo feedback. In *Proc. of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, ACM, pp. 1895–1898, New York. DOI: 10.1145/1645953.1646259. 144

Y. Lv and C. Zhai. 2010. Positional relevance model for pseudo-relevance feedback. In *Proc. of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '10, ACM, pages 579–586, New York. DOI: 10.1145/1835449.1835546. 144

Y. Lv and C. Zhai. 2011. Lower-bounding Term Frequency Normalization. In *Proc. of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pp. 7–16. DOI: 10.1145/2063576.2063584 88, 110

P. Lyman, H. R. Varian, K. Swearingen, P. Charles, N. Good, L.L. Jordan, and J. Pal. 2003. How much information? http://www2.sims.berkeley.edu/research/projects/how-much-info-2003. 3

C. D. Manning and H. Schütze. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA. 19, 54, 273

C. D. Manning, P. Raghavan, and H. Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press, New York. xvii, 18, 165, 315

M. E. Maron and J. L. Kuhns. 1960. On relevance, probabilistic indexing and information retrieval. *Journal of the ACM*, 7:216–244. DOI: 10.1145/321033.321035 87

S. Massung and C. Zhai. 2015. SyntacticDiff: Operator-Based Transformation for Comparative Text Mining. In *Proc. of the 3rd IEEE International Conference on Big Data*, pp. 571–580. 306

S. Massung and C. Zhai. 2016. Non-Native Text Analysis: A Survey. *The Journal of Natural Language Engineering*, 22(2):163–186. DOI: 10.1017/S1351324915000303 306

S. Massung, C. Zhai, and J. Hockenmaier. 2013. Structural Parse Tree Features for Text Representation. In *IEEE Seventh International Conference on Semantic Computing*, pp. 9–13. DOI: 10.1109/ICSC.2013.13 305

J. D. McAuliffe and D. M. Blei. 2008. Supervised topic models. In J.C. Platt, D. Koller, Y. Singer, and S.T. Roweis, eds., *Advances in Neural Information Processing Systems 20*, pages 121–128. Curran Associates, Inc. 386

G. J. McLachlan and T. Krishnan. 2008. *The EM algorithm and extensions*. 2nd ed. Wiley Series in Probability and Statistics. Hoboken, NJ., Wiley. http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+52983362X&sourceid=fbw_bibsonomy. DOI: 10.1002/9780470191613 466

Q. Mei. 2009. Contextual text mining. Ph.D. Dissertation, University of Illinois at Urbana-Champaign. 440

Q. Mei and C. Zhai. 2006. A mixture model for contextual text mining. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, ACM, pp. 649–655, New York. DOI: 10.1145/1150402.1150482. 423, 440

Q. Mei, D. Xin, H. Cheng, J. Han, and C. Zhai. 2006. Generating semantic annotations for frequent patterns with context analysis. In *Proc. of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, ACM, pp. 337–346, New York. DOI: 10.1145/1150402.1150441. 417

Q. Mei, C. Liu, H. Su, and C. Zhai. 2006. A probabilistic approach to spatiotemporal theme pattern mining on weblogs. In *Proc.of the 15th international conference on World Wide Web (WWW '06)*. ACM. New York, 533–542. DOI: 10.1145/1135777.1135857. 425, 426

Q. Mei, X. Ling, M. Wondra, H. Su, and C. Zhai. 2007a. Topic sentiment mixture: Modeling facets and opinions in weblogs. In *Proc. of the 16th International Conference on World Wide Web*, WWW '07, ACM, pp. 171–180, New York. DOI: 10.1145/1242572.1242596. 410

Q. Mei, X. Shen, and C. Zhai. 2007b. Automatic labeling of multinomial topic models. In *Proc. of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Jose, California, August 12-15, 2007, pp. 490–499. DOI: 10.1145/1281192.1281246. 278

Q. Mei, D. Cai, D. Zhang, and C. Zhai. 2008. Topic modeling with network regularization. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, ACM, pp. 101–110, New York. DOI: 10.1145/1367497.1367512. 431, 432, 440

T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur. 2010. Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association*, Makuhari, Chiba, Japan, September 26-30, 2010, pp. 1045–1048. http://www.isca-speech.org/archive/interspeech_2010/i10_1045.html. 292

T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013*. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, NV, pp. 3111–3119. 273, 292, 293

T. M. Mitchell. 1997. *Machine learning*. McGraw Hill Series in Computer Science. McGraw-Hill. 19, 37, 315

M.-F. Moens. 2006. *Information Extraction: Algorithms and Prospects in a Retrieval Context (The Information Retrieval Series)*. Springer-Verlag New York, Inc., Secaucus, NJ. DOI: 10.1007/978-1-4020-4993-4. 55

I. J. Myung. 2003. Tutorial on maximum likelihood estimation. *J. Math. Psychol.*, 47(1):90–100. DOI: 10.1016/S0022-2496(02)00028-7. 36

A. Nenkova and K. McKeown. 2012. A survey of text summarization techniques. In Charu C. Aggarwal and C. Zhai, eds, *Mining Text Data*, pp. 43–76. Springer US. DOI: 10.1007/978-1-4614-3223-4_3. 327

L. Page, S. Brin, R. Motwani, and T. Winograd. 1999. The PageRank Citation Ranking: Bringing Order to the Web. http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf. 216

B. Pang and L. Lee. 2008. Opinion Mining and Sentiment Analysis. *Foundations and Trends in Information Retrieval*, 2(1-2):1–135. DOI: 10.1561/1500000011 409, 410

J. M. Ponte and W. B. Croft. 1998. A language modeling approach to information retrieval. In *Proc. of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '98, ACM, pp. 275–281, New York, NY. DOI: 10.1145/290941.291008. 87, 90, 128, 427

J. R. Quinlan. 1986. Induction of Decision Trees. *Machine Learning*, 1(1):81–106. DOI: 10.1007/BF00116251. 301

D. R. Radev, H. Jing, M. Styś, and D. Tam. 2004. Centroid-based summarization of multiple documents. *Information Processing & Management*, 40(6):919–938. DOI: 10.1016/j.ipm.2003.10.006. 327

D. Ramage, D. Hall, R. Nallapati, and C. D. Manning. 2009. Labeled lda: A supervised topic model for credit attribution in multi-labeled corpora. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1 - Volume 1*, EMNLP '09, Association for Computational Linguistics, pages 248–256, Stroudsburg, PA. 386

E. Reiter and R. Dale. 2000. *Building Natural Language Generation Systems*. Cambridge University Press, New York. 324, 327

F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor. 2010. *Recommender Systems Handbook*. 1st ed. Springer-Verlag New York, Inc. DOI: 10.1007/978-0-387-85820-3 235

C. J. Van Rijsbergen. 1979. *Information Retrieval*. 2nd ed. Butterworth-Heinemann, Newton, MA.

S. Robertson and K. Sparck Jones. 1976. Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27:129–146. 87

S. E. Robertson. 1997. Readings in Information Retrieval. In *The Probability Ranking Principle in IR*, San Francisco, CA, Morgan Kaufmann Publishers Inc. pp. 281–286. 84, 85

S. Robertson and H. Zaragoza. 2009. The Probabilistic Relevance Framework: BM25 and Beyond. *Found. Trends Inf. Retr.*, 3(4):333–389. DOI: 10.1561/1500000019. 88, 89, 129

S. Robertson, H. Zaragoza, and M. Taylor. 2004. Simple BM25 Extension to Multiple Weighted Fields. In *Proc. of the Thirteenth ACM International Conference on Information and Knowledge Management*, CIKM '04, pp. 42–49. DOI: 10.1145/1031171.1031181 110

C. Roe. 2012. The growth of unstructured data: what to do with all those zettabytes? http://www.dataversity.net/the-growth-of-unstructured-data-what-are-we-going-to-do-with-all-those-zettabytes/. 3

R. Rosenfeld. 2000. Two decades of statistical language modeling: Where do we go from here. In *Proceedings of the IEEE*. 54

G. Salton. 1989. *Automatic Text Processing: The Transformation, Analysis and Retrieval of Information by Computer*. Addison-Wesley. 18

G. Salton and M. McGill. 1983. *Introduction to Modern Information Retrieval*. McGraw-Hill. 18

G. Salton, A. Wong, and C. S. Yang. 1975. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620. 87

G. Salton and C. Buckley. 1990. Improving retrieval performance by relevance feedback. *Journal of the American Society for Information Science*, 41:288–297. 144

M. Sanderson. 2010. Test Collection Based Evaluation of Information Retrieval Systems. *Foundations and Trends in Information Retrieval*, 4(4):247–375. 168, 188

M. Sanderson and W. B. Croft. 2012. The history of information retrieval research. *Proc. of the IEEE*, 100(Centennial-Issue):1444–1451, 2012. DOI: 10.1109/JPROC.2012.2189916. 85

S. Sarawagi. 2008. Information extraction. *Found. Trends databases*, 1(3):261–377. DOI: 10.1561/1900000003. 19, 55

F. Sebastiani. 2002. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47. DOI: 10.1145/505282.505283. 315

G. Shani and A. Gunawardana. 2011. Evaluating Recommendation Systems. In *Recommender Systems Handbook*, 2nd ed., pp. 257–297. Springer, New York, NY. DOI: 10.1007/978-0-387-85820-3_8. 235

F. Silvestri. 2010. Mining query logs: Turning search usage data into knowledge. *Found. Trends Inf. Retr.*, 4:1–174. DOI: 10.1561/1500000013 144

A. Singhal, C. Buckley, and Mandar Mitra. 1996. Pivoted document length normalization. In *Proc. of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '96,ACM, pp. 21–29, New York. DOI: 10.1145/243199.243206. 89, 106, 128

N. Smith. 2010. Text-driven forecasting. http://www.cs.cmu.edu/ñasmith/papers/smith.whitepaper10.pdf. 440

Mark D. Smucker, James Allan, and Ben Carterette. 2007. A Comparison of Statistical Significance Tests for Information Retrieval Evaluation. In *Proc. of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, ACM, pp. 623–632, New York. DOI: 10.1145/1321440.1321528. 185

K. Sparck Jones and P. Willett, eds. 1997. *Readings in Information Retrieval*. San Francisco, CA, Morgan Kaufmann Publishers Inc. 18, 85, 188

N. Spirin and J. Han. May 2012. Survey on Web Spam Detection: Principles and Algorithms. *SIGKDD Explor. Newsl.*, 13(2):50–64. DOI: 10.1145/2207243.2207252. 191

E. Stamatatos. 2009. A Survey of Modern Authorship Attribution Methods. *J. Am. Soc. Inf. Sci. Technol.*, 60(3):538–556. DOI: 10.1002/asi.v60:3 305

M. Steinbach, G. Karypis, and V. Kumar. 2000. A comparison of document clustering techniques. In *KDD Workshop on Text Mining*. 296

J. Steinberger and K. Jezek. 2009. Evaluation measures for text summarization. *Computing and Informatics*, 28(2):251–275. 327

M. Steyvers and T. Griffiths. 2007. Probabilistic topic models. *Handbook of Latent Semantic Analysis*, 427(7):424–440. 385, 386

Y. Sun and J. Han. 2012. *Mining Heterogeneous Information Networks: Principles and Methodologies*. Morgan & Claypool Publishers. DOI: 10.2200/S00433ED1V01Y201207DMK005. 440

I. Titov and R. McDonald. 2008. Modeling online reviews with multi-grain topic models. In *Proc. of the 17th International Conference on World Wide Web*, WWW '08, ACM, pp. 111–120, New York. DOI: 10.1145/1367497.1367513. 410

H. Turtle and W. B. Croft. 1990. Inference networks for document retrieval. In *Proc. of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '90, ACM, pp. 1–24, New York. DOI: 10.1145/96749.98006. 88

Princeton University. 2010. About wordnet. http://wordnet.princeton.edu. 395

C. J. van Rijsbergen. 1979. *Information Retrieval*. Butterworths. 18

H. Wang, Yue Lu, and C. Zhai. 2010. Latent Aspect Rating Analysis on Review Text Data: A Rating Regression Approach. In *Proc. of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, ACM, pp. 783–792, New York. DOI: 10.1145/1835804.1835903. 318, 327, 405, 406, 407, 408, 409, 410

H. Wang, Y. Lu, and C. Zhai. 2011. Latent Aspect Rating Analysis Without Aspect Keyword Supervision. In *Proc. of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, ACM, pp. 618–626, New York. DOI: 10.1145/2020408.2020505. 318, 327, 405, 410

J. Weizenbaum. 1966. ELIZA—A Computer Program for the Study of Natural Language Communication Between Man and Machine, Communications of the ACM 9 (1): 36–45, DOI: 10.1145/265153.365168. 44

J. S. Whissell and C. L. A. Clarke. 2013. Effective Measures for Inter-document Similarity. In *Proc. of the 22nd ACM International Conference on Conference on Information & Knowledge Management*, CIKM '13, ACM, pages 1361–1370, New York. DOI: 10.1145/2505515.2505526. 279

R. W. White and R. A. Roth. 2009. *Exploratory Search: Beyond the Query-Response Paradigm*. Synthesis Lectures on Information Concepts, Retrieval, and Services. Morgan & Claypool Publishers. DOI: 10.2200/S00174ED1V01Y200901ICR003. 85

R. W. White, B. Kules, S. M. Drucker, and m.c. schraefel. 2006. Introduction. *Commun. ACM*, 49(4):36–39. DOI: 10.1145/1121949.1121978. 85

I. H. Witten, A. Moffat, and T. C. Bell. 1999. *Managing Gigabytes (2Nd Ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., San Francisco, CA. 18, 165

C.F J. Wu. 1983. On the convergence properties of the EM algorithm. *Ann. of stat.*, 95–103. 368

J. Xu and W. B. Croft. 1996. Query expansion using local and global document analysis. In *Proc. of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '96, ACM, pp. 4–11, New York. DOI: 10.1145/243199.243202. 144

Y. Yang. 1999. An evaluation of statistical approaches to text categorization. *Journal of Information Retrieval*, 1:67–88. 315

C. Zhai. 1997. Exploiting context to identify lexical atoms—a statistical view of linguistic context. In *Proc. of the International and Interdisciplinary Conference on Modelling and Using Context (CONTEXT-97)*, pages 119–129. Rio de Janeiro, Brazil. 273, 291

C. Zhai. 2008. *Statistical Language Models for Information Retrieval*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers. DOI: 10.2200/S00158ED1V01Y200811HLT001. 55, 87, 128, 129

C. Zhai and J. Lafferty. 2001. Model-based Feedback in the Language Modeling Approach to Information Retrieval. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*, CIKM '01, ACM, pp. 403–410, New York. DOI: 10.1145/502585.502654. 143, 466, 473

C. Zhai and J. Lafferty. 2004. A Study of Smoothing Methods for Language Models Applied to Information Retrieval. *ACM Trans. Inf. Syst.*, 22(2):179–214. 475

C. Zhai, P. Jansen, E. Stoica, N. Grot, and D. A. Evans. 1998. Threshold Calibration in CLARIT Adaptive Filtering. In *Proc. of Seventh Text REtrieval Conference (TREC-7)*, pp. 149–156. 227

C. Zhai, P. Jansen, and D. A. Evans. 2000. Exploration of a heuristic approach to threshold learning in adaptive filtering. In *SIGIR*, ACM, pp. 360–362. DOI: 10.1145/345508.345652. 235

C. Zhai, A. Velivelli, and B. Yu. 2004. A cross-collection mixture model for comparative text mining. In *Proc. of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, ACM, pp. 743–748, New York. DOI: 10.1145/1014052.1014150. 423

D. Zhang, C. Zhai, J. Han, A. Srivastava, and N. Oza. 2009. Topic modeling for OLAP on multidimensional text databases: topic cube and its applications. *Stat. Anal. Data Min.* 2, 5–6 (December 2009), 378–395. DOI: 10.1002/sam.v2.5/6. 440

J. Zhu, A. Ahmed, and E. P. Xing. 2009. Medlda: Maximum margin supervised topic models for regression and classification. In *Proc. of the 26th Annual International Conference on Machine Learning*, ICML '09, ACM, pp. 1257–1264, New York. DOI: 10.1145/1553374.1553535. 386

G. K. Zipf. 1949. *Human Behavior and the Principle of Least-Effort*. Cambridge, MA, Addison-Wesley, 162

Index

Authors' Biographies

ChengXiang Zhai

ChengXiang Zhai is a Professor of Computer Science and Willett Faculty Scholar at the University of Illinois at Urbana–Champaign, where he is also affiliated with the Graduate School of Library and Information Science, Institute for Genomic Biology, and Department of Statistics. He received a Ph.D. in Computer Science from Nanjing University in 1990, and a Ph.D. in Language and Information Technologies from Carnegie Mellon University in 2002. He worked at Clairvoyance Corp. as a Research Scientist and then Senior Research Scientist from 1997–2000. His research interests include information retrieval, text mining, natural language processing, machine learning, biomedical and health informatics, and intelligent education information systems. He has published over 200 research papers in major conferences and journals. He served as an Associate Editor for *Information Processing and Management*, as an Associate Editor of *ACM Transactions on Information Systems*, and on the editorial board of *Information Retrieval Journal*. He was a conference program co-chair of ACM CIKM 2004, NAACL HLT 2007, ACM SIGIR 2009, ECIR 2014, ICTIR 2015, and WWW 2015, and conference general co-chair for ACM CIKM 2016. He is an ACM Distinguished Scientist and a recipient of multiple awards, including the ACM SIGIR 2004 Best Paper Award, the ACM SIGIR 2014 Test of Time Paper Award, Alfred P. Sloan Research Fellowship, IBM Faculty Award, HP Innovation Research Program Award, Microsoft Beyond Search Research Award, and the Presidential Early Career Award for Scientists and Engineers (PECASE).

Sean Massung
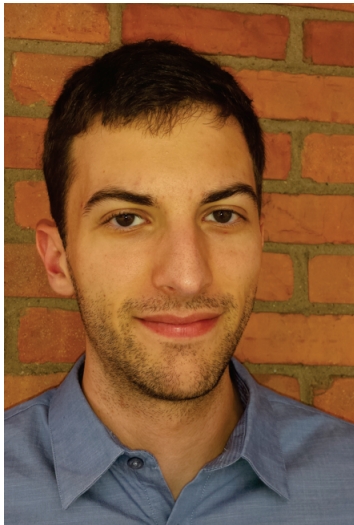
Sean Massung is a Ph.D. candidate in computer science at the University of Illinois at Urbana–Champaign, where he also received both his B.S. and M.S. degrees. He is a co-founder of META and uses it in all of his research. He has been instructor for CS 225: Data Structures and Programming Principles, CS 410: Text Information Systems, and CS 591txt: Text Mining Seminar. He is included in the 2014 List of Teachers Ranked as Excellent at the University of Illinois and has received an Outstanding Teaching Assistant Award and CS@Illinois Outstanding Research Project Award. He has given talks at Jump Labs Champaign and at UIUC for Data and Information Systems Seminar, Intro to Big Data, and Teaching Assistant Seminar. His research interests include text mining applications in information retrieval, natural language processing, and education.