PART I

WEB STRUCTURE MINING

In the first part of the book we discuss basic ideas and techniques for extracting text information from the Web, including collecting and indexing web documents and searching and ranking web pages by their textual content and hyperlink structure. We first discuss the motivation to organize the web content and find better ways for web search to make the vast knowledge on the Web easily accessible. Then we describe briefly the basics of the Web and explore the approaches taken by web search engines to retrieve web pages by keyword search. To do this we look into the technology for text analysis and search developed earlier in the area of information retrieval and extended recently with ranking methods based on web hyperlink structure.

All that may be seen as a preprocessing step in the overall process of data mining the web content, which provides the input to machine learning methods for extracting knowledge from hypertext data, discussed in the second part of the book.

Data Mining the Web: Uncovering Patterns in Web Content, Structure, and Usage
By Zdravko Markov and Daniel T. Larose

CHAPTER 1

INFORMATION RETRIEVAL AND WEB SEARCH

WEB CHALLENGES

As originally proposed by Tim Berners-Lee [1], the Web was intended to improve the management of general information about accelerators and experiments at CERN. His suggestion was to organize the information used at that institution in a graphlike structure where the nodes are documents describing objects, such as notes, articles, departments, or persons, and the links are relations among them, such as "depends on," "is part of," "refers to," or "uses." This seemed suitable for a large organization like CERN, and soon after it appeared that the framework proposed by Berners-Lee was very general and would work very well for any set of documents, providing flexibility and convenience in accessing large amounts of text. A very important development of this idea was that the documents need not be stored at the same computer or database but rather, could be distributed over a network of computers. Luckily, the infrastructure for this type of distribution, the Internet, had already been developed. In short, this is how the Web was born.

Looking at the Web many years later and comparing it to the original proposal of 1989, we see two basic differences:

1. The recent Web is huge and grows incredibly fast. About 10 years after the Berners-Lee proposal, the Web was estimated to have 150 million nodes (pages) and 1.7 billion edges (links). Now it includes more than 4 billion pages, with about 1 million added every day.

Data Mining the Web: Uncovering Patterns in Web Content, Structure, and Usage
By Zdravko Markov and Daniel T. Larose

2. The formal semantics of the Web is very restricted—nodes are simply web pages and links are of a single type (e.g., "refer to"). The meaning of the nodes and links is not a part of the web system; rather, it is left to web page developers to describe in the page content what their web documents mean and what types of relations they have with the documents to which they are linked. As there is neither a central authority nor editors, the relevance, popularity, and authority of web pages are hard to evaluate. Links are also very diverse, and many have nothing to do with content or authority (e.g., navigation links).

The Web is now the largest, most open, most democratic publishing system in the world. From a publishers' (web page developers') standpoint, this is a great feature of the Web—any type of information can be distributed worldwide with no restriction on its content, and most important, using the developer's own interpretation of the web page and link meaning. From a web user's point of view, however, this is the worst thing about the Web. To determine a document's type the user has to read it all. The links simply refer to other documents, which means again that reading the entire set of linked documents is the only sure way to determine the document types or areas. This type of document access is directly opposite to what we know from databases and libraries, where all data items or documents are organized in various ways: by type, topic, area, author, year, and so on. Using a library in a "weblike" manner would mean that one has first to read the entire collection of books (or at least their titles and abstracts) to find the one in the area or topic that he or she needs. Even worse, some web page publishers cheat regarding the content of their pages, using titles or links with attractive names to make the user visit pages that he or she would never look at otherwise.

At the same time, the Web is the largest repository of knowledge in the world, so everyone is tempted to use it, and every time that one starts exploring the Web, he or she knows that the piece of information sought is "out there." But the big question is how to find it. Answering this question has been the basic driving force in developing web search technologies, now widely available through web search engines such as Google, Yahoo!, and many others. Other approaches have also been taken: Web pages have been manually edited and organized into topic directories, or data mining techniques have been used to extract knowledge from the Web automatically.

To summarize, the challenge is to bring back the semantics of hypertext documents (something that was a part of the original web proposal of Berners-Lee) so that we can easily use the vast amount of information available. In other words, we need to *turn web data into web knowledge*. In general, there are several ways to achieve this: Some use the existing Web and apply sophisticated search techniques; others suggest that we change the way in which we create web pages. We discuss briefly below the three main approaches.

Web Search Engines

Web search engines explore the existing (semantics-free) structure of the Web and try to find documents that match user search criteria: that is, to bring semantics into the process of web search. The basic idea is to use a set of words (or terms) that the user

specifies and retrieve documents that include (or do not include) those words. This is the *keyword search* approach, well known from the area of information retrieval (IR). In web search, further IR techniques are used to avoid terms that are too general and too specific and to take into account term distribution throughout the entire body of documents as well as to explore document similarity. Natural language processing approaches are also used to analyze term context or lexical information, or to combine several terms into phrases. After retrieving a set of documents ranked by their degree of matching the keyword query, they are further ranked by importance (popularity, authority), usually based on the web link structure. All these approaches are discussed further later in the book.

Topic Directories

Web pages are organized into hierarchical structures that reflect their meaning. These are known as *topic directories*, or simply *directories*, and are available from almost all web search portals. The largest is being developed under the Open Directory Project (`dmoz.org`) and is used by Google in their Web Directory: "the Web organized by topic into categories," as they put it. The directory structure is often used in the process of web search to better match user criteria or to specialize a search within a specific set of pages from a given category. The directories are usually created manually with the help of thousands of web page creators and editors. There are also approaches to do this automatically by applying machine learning methods for classification and clustering. We look into these approaches in Part II.

Semantic Web

Semantic web is a recent initiative led by the web consortium (`w3c.org`). Its main objective is to bring formal knowledge representation techniques into the Web. Currently, web pages are designed basically for human readers. It is widely acknowledged that the Web is like a "fancy fax machine" used to send good-looking documents worldwide. The problem here is that the nice format of web pages is very difficult for computers to understand—something that we expect search engines to do. The main idea behind the semantic web is to add formal descriptive material to each web page that although invisible to people would make its content easily understandable by computers. Thus, the Web would be organized and turned into the largest knowledge base in the world, which with the help of advanced reasoning techniques developed in the area of artificial intelligence would be able not just to provide ranked documents that match a keyword search query, but would also be able to answer questions and give explanations. The web consortium site (`http://www.w3.org/2001/sw/`) provides detailed information about the latest developments in the area of the semantic web.

Although the semantic web is probably the future of the Web, our focus is on the former two approaches to bring semantics to the Web. The reason for this is that web search is the data mining approach to web semantics: extracting knowledge from web data. In contrast, the semantic web approach is about turning web pages into formal knowledge structures and extending the functionality of web browsers with knowledge manipulation and reasoning tools.

CRAWLING THE WEB

In this and later sections we use basic web terminology such as *HTML, URL, web browsers*, and *servers*. We assume that the reader is familiar with these terms, but for the sake of completeness we provide a brief introduction to web basics.

Web Basics

The Web is a huge collection of documents linked together by references. The mechanism for referring from one document to another is based on hypertext and embedded in the HTML (HyperText Markup Language) used to encode web documents. HTML is primarily a typesetting language (similar to Tex and LaTex) that describes how a document should be displayed in a browser window. Browsers are computer programs that read HTML documents and display them accordingly, such as the popular browsers Microsoft Internet Explorer and Netscape Communicator. These programs are clients that connect to web servers that hold actual web documents and send those documents to the browsers by request. Each web document has a web address called the URL (universal resource locator) that identifies it uniquely. The URL is used by browsers to request documents from servers and in hyperlinks as a reference to other web documents. Web documents associated with their web addresses (URLs) are usually called *web pages*.

A URL consists of three segments and has the format

```
<protocol name>://<machine name>/<file name>,
```

where `<protocol name>` is the protocol (a language for exchanging information) that the browser and the server use to communicate (HTTP, FTP, etc.), `<machine name>` is the name (the web address) of the server, and `<file name>` is the directory path showing where the document is stored on the server. For example, the URL

```
http://dmoz.org/Computers/index.html
```

points to an HTML document stored on a file named "index.html" in the folder "Computers" located on the server "dmoz.org." It can also be written as

```
http://dmoz.org/Computers/
```

because the browser automatically looks for a file named index. html if only a folder name is specified.

Entering the URL in the address window makes the browser connect to the web server with the corresponding name using the HyperText Transport Protocol (HTTP). After a successful connection, the HTML document is fetched and its content is shown in the browser window. Some intermediate steps are taking place meanwhile, such as obtaining the server Internet address (called the IP address) from a domain name server (DNS), establishing a connection with the server, and exchanging commands. However, we are not going into these details, as they are not important for our discussion here.

Along with its informational content (formatted text and images), a web page usually contains URLs pointing to other web pages. These URLs are encoded in the tag structure of the HTML language. For example, the document index.html at `http://dmoz.org/Computers/` includes the following fragment:

```
<table border=0>
<tr><td valign=top><ul>
<li><a href="/Computers/Algorithms/"><b>Algorithms</b></a>
<i>(367)</i>
```

The URL in this HTML fragment, `/Computers/Algorithms/`, is the text that appears quoted in the `<a>` tag preceded by `href`. This is a local URL, a part of the complete URL (`http://dmoz.org/Computers/Algorithms/`), which the browser creates automatically by adding the current protocol name (`http`) and server address (`dmoz.org`). Here is another fragment from the same page that includes absolute URLs.

```
<b>Visit our sister sites</b>
<a href="http://www.mozilla.org/">mozilla.org</a>|
<a href="http://chefmoz.org/">ChefMoz</a>
```

Another important part of the web page linking mechanism is the *anchor*, the text or image in the web page that when clicked makes the browser fetch the web page that is pointed to by the corresponding link. Anchor text is usually displayed emphasized (underlined or in color) so that it can be spotted easily by the user. For example, in the HTML fragment above, the anchor text for the URL `http://mozilla.org/` is "mozilla.org" and that for `http://chefmoz.org/` is "ChefMoz."

The idea of the anchor text is to suggest the meaning or content of the web page to which the corresponding URL is pointing so that the user can decide whether or not to visit it. This may appear similar to Berners-Lee's idea in the original web proposal to attach different semantics to the web links, but there is an important difference here. The anchor is simply a part of the web page content and does not affect the way the page is processed by the browser. For example, spammers may take advantage of this by using anchor text with an attractive name (e.g., summer vacation) to make user visit their pages, which may not be as attractive (e.g., online pharmacy). We discuss approaches to avoid this later.

Formally, the Web can be seen as a *directed graph*, where the nodes are web pages and the links are represented by URLs. Given a web page *P*, the URLs in it are called *outlinks*. Those in other pages pointing to *P* are called *inlinks* (or *backlinks*).

Web Crawlers

Browsing the Web is a very useful way to explore a collection of linked web documents as long as we know good starting points: URLs of pages from the topic or area in which we are interested. However, general search for information about a specific topic or area through browsing alone is impractical. A better approach is to have web pages organized by topic or to search a collection of pages indexed by keywords. The former is done by topic directories and the latter, by search engines. Hereafter we

shall see how search engines collect web documents and index them by the words (terms) they contain. First we discuss the process of collecting web pages and storing them in a local repository. Indexing and document retrieval are discussed in the next section.

To index a set of web documents with the words they contain, we need to have all documents available for processing in a local repository. Creating the index by accessing the documents directly on the Web is impractical for a number of reasons. Collecting "all" web documents can be done by browsing the Web systematically and exhaustively and storing all visited pages. This is done by *crawlers* (also called *spiders* or *robots*).

Ideally, all web pages are linked (there are no unconnected parts of the web graph) and there are no multiple links and nodes. Then the job of a crawler is simple: to run a complete *graph search algorithm*, such as *depth-first* or *breadth-first* search, and store all visited pages. Small-scale crawlers can easily be implemented and are a good programming exercise that illustrates both the structure of the Web and graph search algorithms. There are a number of freely available crawlers from this class that can be used for educational and research purposes. A good example of such a crawler is WebSPHINX (`http://www.cs.cmu.edu/~rcm/websphinx/`).

A straightforward use of a crawler is to visualize and analyze the structure of the web graph. We illustrate this with two examples of running the WebSPHINX crawler. For both runs we start with the Data Mining home page at CCSU at `http://www.ccsu.edu/datamining/`. As we want to study the structure of the web locally in the neighborhood of the starting page, we have to impose some limits on crawling. With respect to the web structure, we may limit the depth of crawling [i.e., the number of hops (links) to follow and the size of the pages to be fetched]. The region of the web to be crawled can also be specified by using the URL structure. Thus, all URLs with the same server name limit crawling within the specific server pages only, while all URLs with the same folder prefixes limit crawling pages that are stored in subfolders only (subtree).

Other limits are dynamic and reflect the time needed to fetch a page or the running time of the crawler. These parameters are needed not only to restrict the web area to be crawled but also to avoid some traps the crawler may fall into (see the discussion following the examples). Some parameters used to control the crawling algorithm must also be passed. These are the graph search method (depth-first or breadth-first) as well as the number of threads (crawling processes running in parallel) to be used. Various other limits and restrictions with respect to web page content can also be imposed (some are discussed in Chapter 2 in the context of page ranking). Thus, for the first example we set the following limits: depth = 3 hops, page size = 30 kB (kilobytes), page timeout = 3 seconds, crawler timeout = 30 seconds, depth-first search, threads = 4. The portion of the web graph crawled with this setting is shown in Figure 1.1. The starting page is marked with its name and URL. Note that due to the dynamic limits and varying network latency, every crawl, even those with the same parameters, is different. In the one shown in Figure 1.1, the crawler reached an interesting structure called a *hub*. This is the page in the middle of a circle of multiple pages. A *hub page* includes a large number of links and is usually some type of directory or reference site that points to many web pages. In our example

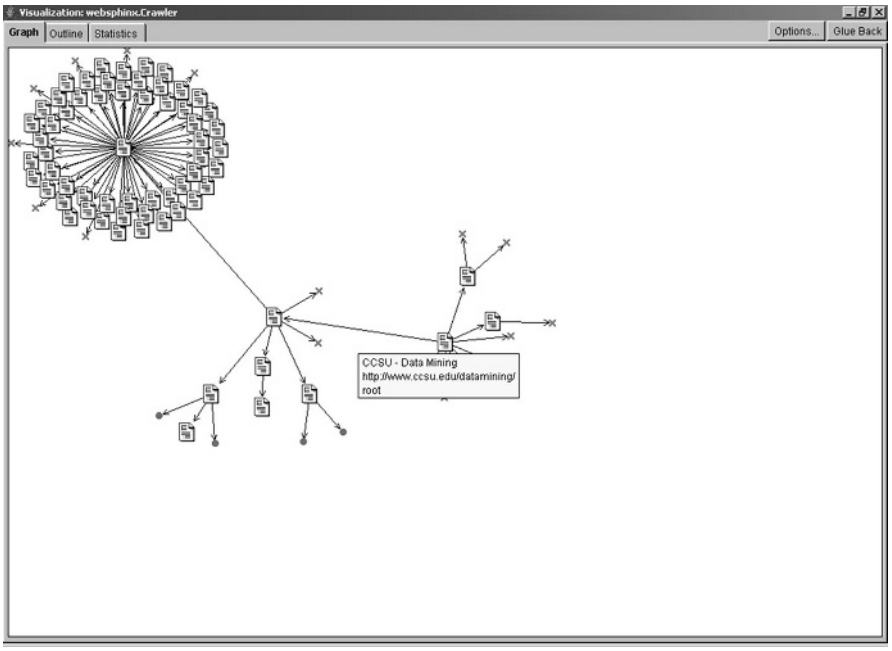

Figure 1.1 Depth-first web crawling limited to depth 3.

the hub page is KDnuggets.com, one of the most comprehensive and well-organized repositories of information about data mining.

Another crawl with the same parameters and limits, but using a breadth-first search, is shown in Figure 1.2. The web graph here is more uniformly covered because of the nature of the search algorithm—all immediate neighbors of a given page are explored before going to further pages. Therefore, the breadth-first crawl discovered another hub page that is closer to the starting point. It is the resources page at CCSU—Data Mining. In both graphs, the $\times$'s mean that some limits have been reached or network exceptions have occurred, and the dots are pages that have not yet been explored, due to the crawler timeout.

The web graph shown by the WebSPHINX crawler is actually a tree, because only the links followed are shown and the pages are visited only once. However, the Web is not a tree, and generally there is more than one inlink to a page (occurrences of the page URL in other web pages). In fact, these inlinks are quite important when analyzing the web structure because they can be used as a measure of web page popularity or importance. Similar to the hubs, a web page with a large number of inlinks is also important and is called an *authority*. Finding good authorities is, however, not possible using the local crawls that we illustrated with the examples above and generally requires analyzing a much larger portion of the web (theoretically, the entire Web, if we want to find all inlinks).

Although there is more than one inlink to some of the pages in our example (e.g., the CCSU or the CCSU—Data Mining home pages are referred to in many other pages), these links come from the same site and are included basically for navigation

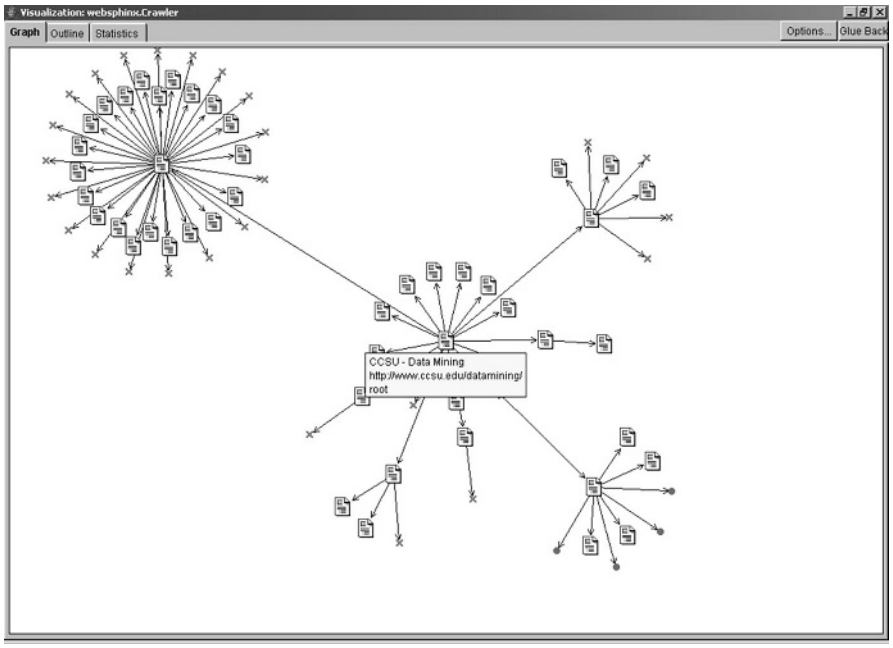

Figure 1.2 Breadth-first web crawling limited to depth 3.

purposes. Such links do not reflect the actual popularity of the web pages to which they point. This is a situation similar to self-citation in scientific literature, which is hardly considered as a good measure of authority. We discuss these issues in more depth later in the context of page ranking.

Although visualizing the web graph is a nice feature of web crawlers, it is not the most important. In fact, the basic role of a crawler that is part of a search engine is to collect information about web pages. This may be web page textual content, page titles, headers, tag structure, or web links structure. This information is organized properly for efficient access and stored in a local repository to be used for indexing and search (see the next section). Thus, a crawler is not only an implementation of a graph search algorithm, but also an HTML parser and analyzer, and much more. Some of the extended functionalities of web crawlers are discussed next.

The Web is far from an ideal graph structure such as the one shown in Figures 1.1 and 1.2. Crawling the Web involves interaction with hundreds of thousands of web servers, designed to meet different goals, provide different services such as database access and user interactions, generate dynamic pages, and so on. Another very important factor is the huge number of pages that have to be visited, analyzed, and stored. Therefore, a web crawler designed to crawl the entire Web is a sophisticated program that uses advanced programming technology to improve its time and space efficiency and usually runs on high-performance parallel computers. Hereafter we provide a brief account of common problems that large-scale crawlers are faced with

and outline some solutions. We are not going into technical details because this is aside from our main goal: analyzing the web content.

- The process of fetching a web page involves some network latency (sometimes a "timeout"). To avoid waiting for the current page to load in order to continue with the next page, crawlers fetch multiple pages simultaneously. In turn, this requires connecting to multiple servers (usually thousands) at the same time, which is achieved by using parallel and distributed programming technology such as multithreading (running multiple clients concurrently) or nonblocking sockets and event handlers.
- The first step in fetching a web page is address resolution, converting the symbolic web address into an IP address. This is done by a DNS server that the crawler connects. Since multiple pages may be located at a single server, storing addresses already looked up in a local cache allows the crawler to avoid repeating DNS requests and consequently, improves its efficiency and minimizes the Internet traffic.
- After fetching a web page it is scanned and the URLs are extracted—these are the outlinks that will be followed next by the crawler. There are many ways to specify an URL in HTML. It may also be specified by using the IP address of the server. As the mapping between server names and IP addresses is many-to-many,[1] this may result in multiple URLs for a single web page. The problem is aggravated by the fact that browsers are tolerant of pages that have the wrong syntax. As a result, HTML documents are not designed with enough care and often include wrongly specified URLs as well as other malicious structures. All this makes parsing and extracting URLs from HTML documents not an easy task. The solution is to use a well-designed and robust parser and after extracting the URLs to convert them into a canonical form. Even so, there are traps that the crawler may fall into. The best policy is to collect statistics regularly about each crawl and use them in a special module called a *guard*. The purpose of the guard is to exclude outlinks that come from sites that dominate the crawler collection of pages. Also, it may filter out links to dynamic pages or forms as well as to nontextual pages (e.g., images, scripts).
- Following the web page links may bring the crawler back to pages already visited. There may also exist identical web pages at different web addresses (called *mirror sites*). To avoid following identical links and fetching identical pages multiple times, the crawler should keep caches for URLs and pages (this is another reason for putting URLs into canonical form). Various hashing techniques are used for this purpose.
- An important part of the web crawler system is the *text repository*. Yahoo! claimed that in August 2005 their index included 20 billion pages [2], 19.2 of them web documents. With an average of 10 kB for a web document, this

[1] A server may have more than one IP address, and different host names may be mapped onto a single IP address. The former is usually done for load balancing of servers that handle a large number of requests, and the latter, for organizing web pages into more logical host names than the number of IP addresses available (virtual hosting).

makes about 200,000 GB (gigabytes) of storage. Managing such a huge repository is a challenging task. Note that this is the crawler repository, not the indexed collection of web pages used to answer search queries. The latter is of comparable size, but even more complicated because of the need for fast access. The crawler repository is used to store pages, maintain the URL and document caches needed by the crawler, and provide access for building indices at the next stage. To minimize storage needs, the web pages are usually compressed, which reduces the storage requirements two- to threefold. For large-scale crawlers the text repository may be distributed over a number of storage servers.

- The purpose of a web crawler used by a search engine is to provide local access to the most recent versions of possibly all web pages. This means that the Web should be crawled regularly and the collection of pages updated accordingly. Having in mind the huge capacity of the text repository, the need for regular updates poses another challenge for the web crawler designers. The problem is the high cost of updating indices. A common solution is to append the new versions of web pages without deleting the old ones. This increases the storage requirements but also allows the crawler repository to be used for archival purposes. In fact, there are crawlers that are used just for the purposes of archiving the web. The most popular web archive is the Internet Archive at `http://www.archive.org/`.
- The Web is a live system, it is constantly changing—new features emerge and new services are offered. In many cases they are not known in advance, or even worse, web pages and servers may behave unpredictably as a result of bugs or malicious design. Thus, the web crawler should be a very robust system that is updated constantly in order to respond to the ever-changing Web.
- Crawling of the Web also involves interaction of web page developers. As Brin and Page [5] mention in a paper about their search engine Google, they were getting e-mail from people who noticed that somebody (or something) visited their pages. To facilitate this interaction there are standards that allow web servers and crawlers to exchange information. One of them is the *robot exclusion protocol*. A file named robots.txt that lists all path prefixes of pages that crawlers should not fetch is placed in the http root directory of the server and read by the crawlers before crawling of the server tree.

So far we discussed crawling based on the syntax of the web graph: that is, following links and visiting pages without taking into account their semantics. This is in a sense equivalent to *uninformed graph search*. However, let's not forget that we discuss web crawling in the context of web search. Thus, to improve its efficiency, or for specific purposes, crawling can also be done as a *guided* (*informed*) *search*. Usually, crawling precedes the phase of web page evaluation and ranking, as the latter comes after indexing and retrieval of web documents. However, web pages can be evaluated while being crawled. Thus, we get some type of enhanced crawling that uses page ranking methods to achieve focusing on interesting parts of the Web and avoiding fetching irrelevant or uninteresting pages.

INDEXING AND KEYWORD SEARCH

Generally, there are two types of data: structured and unstructured. *Structured data* have keys (attributes, features) associated with each data item that reflect its content, meaning, or usage. A typical example of structured data is a relational table in a database. Given an attribute (column) name and its value, we can get a set of tuples (rows) that include this value. For example, consider a table that contains descriptions of departments in a school described by a number of attributes, such as subject, programs offered, areas of specialization, facilities, and courses. Then, by a simple query, we may get all departments that, for example, have computer labs. In SQL (Structured Query Language) this query is expressed as `select * from Departments where facilities=``computer lab´´`. A more common situation is, however, to have the same information specified as a one-paragraph text description for each department. Then looking for departments with computer labs would be more difficult and generally would require people to read and understand the text descriptions.

The problem with using structured data is the cost associated with the process of structuring them. The information that people use is available primarily in unstructured form. The largest part of it are text documents (books, magazines, newspapers) written in natural language. To have content-based access to these documents, we organize them in libraries, bibliography systems, and by other means. This process takes a lot of time and effort because it is done by people. There are attempts to use computers for this purpose, but the problem is that content-based access assumes understanding the meaning of documents, something that is still a research question, studied in the area of artificial intelligence and natural language processing in particular. One may argue that natural language texts are structured, which is true as long as the language syntax (grammatical structure) is concerned. However, the transition to meaning still requires semantic structuring or understanding. There exists a solution that avoids the problem of meaning but still provides some types of content-based access to unstructured data. This is the *keyword search* approach known from the area of *information retrieval* (IR). The idea of IR is to retrieve documents by using a simple Boolean criterion: the presence or absence of specific words (keywords, terms) in the documents (the question of meaning here is left to the user who formulates the query). Keywords may be combined in disjunctions and conjunctions, thus providing more expressiveness of the queries. A keyword-based query cannot identify the matching documents uniquely, and thus it usually returns a large number of documents. Therefore, in IR there is a need to rank documents by their relevance to the query. *Relevance ranking* is an important difference with querying structured data where the result of a query is a set (unordered collection) of data items.

IR approaches are applicable to bibliographic databases, collections of journal and newspaper articles, and other large text document collections that are not well structured (not organized by content), but require content-based access. In short, IR is about *finding relevant data using irrelevant keys*. The Web search engines rely heavily on IR technology. The web crawler text repository is very much like the document collection for which the IR approaches have been developed. Thus, having a web crawler, the implementation of IR-based keyword search for the Web is straightforward. Because of their internal HTML tag structure and external web link

Figure 1.3 Directory page for a collection of web documents.

structure, the web documents are richer than simple text documents. This allows search engines to go further and provide more sophisticated methods for matching keyword queries with web documents and to do better relevance ranking. In this section we discuss standard IR techniques for text document processing. The enhancements that come from the Web structure are discussed in the next sections.

To illustrate the basic keyword search approach to the Web, we consider again the unstructured version of our example with the departments and make it more realistic by taking the web page that lists all departments in the school of Arts and Sciences at CCSU (Figure 1.3). The information about each department is provided in a separate web page linked to the department name listed on the main page. We include one of those pages in Figure 1.4 (the others have a similar format).

The first step is to fetch the documents from the Web, remove the HTML tags, and store the documents as plain text files. This can easily be done by a web crawler (the reader may want to try WebSPHINX) with proper parameter settings. Then the keyword search approach can be used to answer such queries as:

1. Find documents that contain the word *computer* and the word *programming*.
2. Find documents that contain the word *program*, but not the word *programming*.
3. Find documents where the words *computer* and *lab* are adjacent. This query is called *proximity query*, because it takes into account the lexical distance between words. Another way to do it is by searching for the phrase *computer lab*.

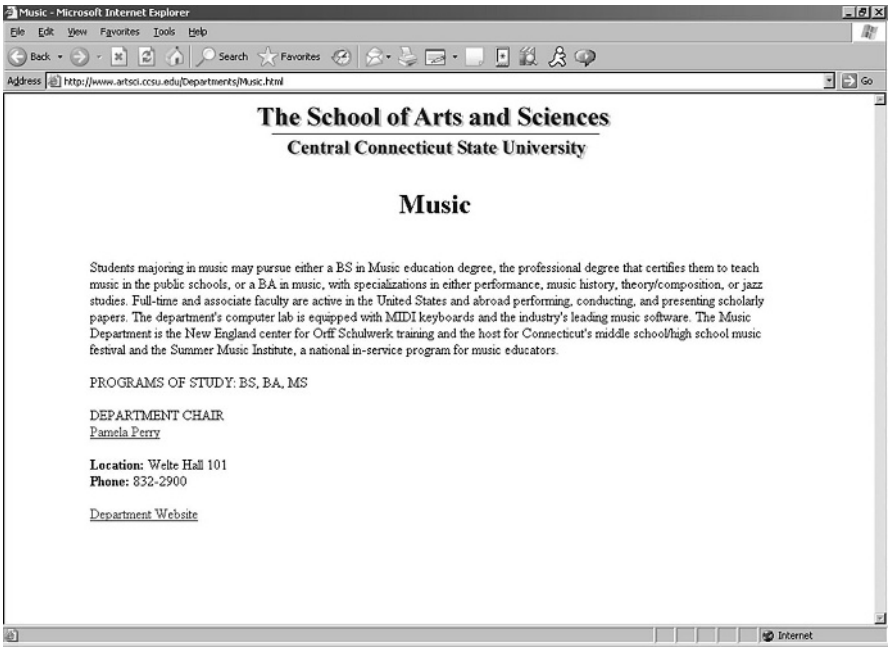

Figure 1.4 Sample web document.

Answering such queries can be done by scanning the content of the documents and matching the keywords against the words in the documents. For example, the music department document shown in Figure 1.4 will be returned by the second and third queries.

Document Representation

To facilitate the process of matching keywords and documents, some preprocessing steps are taken first:

1. Documents are *tokenized*; that is, all punctuation marks are removed and the character strings without spaces are considered as tokens (words, also called *terms*).
2. All characters in the documents and in the query are converted to upper or lower case.
3. Words are reduced to their canonical form (stem, base, or root). For example, variant forms such as *is* and *are* are replaced with *be*, various endings are removed, or the words are transformed into their root form, such as *programs* and *programming* into *program*. This process, called *stemming*, uses morphological information to allow matching different variants of words.
4. Articles, prepositions, and other common words that appear frequently in text documents but do not bring any meaning or help distinguish documents are

TABLE 1.1 Basic Statistics for A&S Documents

Document ID	Document Name	Words	Terms
d_1	Anthropology	114	86
d_2	Art	153	105
d_3	Biology	123	91
d_4	Chemistry	87	58
d_5	Communication	124	88
d_6	Computer Science	101	77
d_7	Criminal Justice	85	60
d_8	Economics	107	76
d_9	English	116	80
d_{10}	Geography	95	68
d_{11}	History	108	78
d_{12}	Mathematics	89	66
d_{13}	Modern Languages	110	75
d_{14}	Music	137	91
d_{15}	Philosophy	85	54
d_{16}	Physics	130	100
d_{17}	Political Science	120	86
d_{18}	Psychology	96	60
d_{19}	Sociology	99	66
d_{20}	Theatre	116	80
Total number of words/terms		2195	1545
Number of different words/terms		744	671

called *stopwords*. Examples are *a*, *an*, *the*, *on*, *in*, and *at*. These words are usually removed.

The collection of words that are left in the document after all those steps is different from the original document and may be considered as a *formal representation* of the document. To emphasize this difference, we call the words in this collection *terms*. The collection of words (terms) in the entire set of documents is called the *text corpus*.

Table 1.1 shows some statistics about documents from the school of Arts and Sciences (A&S) that illustrate this process (the design department is not included because the link points directly to the department web page). The words are counted after tokenizing the plain text versions of the documents (without the HTML structures). The term counts are taken after removing the stopwords but without stemming.

The terms that occur in a document are in fact the *parameters* (also called *features*, *attributes*, or *variables* in different contexts) of the document representation. The types of parameters determine the type of document representation:

- The simplest way to use a term as a feature in a document representation is to check whether or not the term occurs in the document. Thus, the term is considered as a Boolean attribute, so the representation is called *Boolean*.

- The value of a term as a feature in a document representation may be the number of occurrences of the term (*term frequency*) in the document or in the entire corpus. Document representation that includes the term frequencies but not the term positions is called a *bag-of-words representation* because formally it is a multiset or bag (a type of set in which each item may occur numerous times).
- Term positions may be included along with the frequency. This is a "complete" representation that preserves most of the information and may be used to generate the original document from its representation.

The purpose of the document representation is to help the process of keyword matching. However, it may also result in loss of information, which generally increases the number of documents in response to the keyword query. Thus, some irrelevant documents may also be returned. For example, stemming of *programming* would change the second query and allow the first one to return more documents (its original purpose is to identify the Computer Science department, but stemming would allow more documents to be returned, as they all include the word *program* or *programs* in the sense of "program of study"). Therefore, stemming should be applied with care and even avoided, especially for Web searches, where a lot of common words are used with specific technical meaning. This problem is also related to the issue of context (lexical or semantic), which is generally lost in keyword search. A partial solution to the latter problem is the use of proximity information or lexical context. For this purpose a richer document representation can be used that preserves term positions. Some punctuation marks can be replaced by placeholders (tokens that are left in a document but cannot be used for searching), so that part of the lexical structure of the document, such as sentence boundaries, can be preserved. This would allow answering queries such as "Find documents containing *computer* and *programming* in the same sentence." Another approach, called *part-of-speech tagging*, is to attach to words tags that reflect their part-of-speech roles (e.g., verb or noun). For example, the word *can* usually appears in the stopword list, but as a noun it may be important for a query.

For the purposes of searching small documents and document collections such as the CCSU Arts and Sciences directory, direct text scanning may work well. This approach cannot, however, be scaled up to large documents and/or collections of documents such as the Web, due to the prohibitive computational cost. The approach used for the latter purposes is called an *inverted index* and is central to IR. The idea is to switch the roles of document IDs and terms. Instead of accessing documents by IDs and then scanning their content for specific terms, the terms that documents contain are used as access keys. The simplest form of an inverted index is a *document–term matrix*, where the access is by terms (i.e., it is transposed to *term–document matrix*).

The term–document matrix for our department example has 20 rows, corresponding to documents, and 671 columns, corresponding to all the different terms that occur in the text corpus. In the *Boolean* form of this matrix, each cell contains 1 if the term occurs in the document, and 0 otherwise. We assign the documents as rows because this representation is also used in later sections, but in fact, the table is accessed by columns. A small part of the matrix is shown in Table 1.2 (instead of names, document IDs are used).

TABLE 1.2 Boolean Term–Document Matrix

Document ID	*lab*	*laboratory*	*programming*	*computer*	*program*
d_1	0	0	0	0	1
d_2	0	0	0	0	1
d_3	0	1	0	1	0
d_4	0	0	0	1	1
d_5	0	0	0	0	0
d_6	0	0	1	1	1
d_7	0	0	0	0	1
d_8	0	0	0	0	1
d_9	0	0	0	0	0
d_{10}	0	0	0	0	0
d_{11}	0	0	0	0	0
d_{12}	0	0	0	1	0
d_{13}	0	0	0	0	0
d_{14}	1	0	0	1	1
d_{15}	0	0	0	0	1
d_{16}	0	0	0	0	1
d_{17}	0	0	0	0	1
d_{18}	0	0	0	0	0
d_{19}	0	0	0	0	1
d_{20}	0	0	0	0	0

Using the term–document matrix, answering the keyword search queries is straightforward. For example, query 1 returns only d_6 (Computer Science document), because it has 1's in the columns *programming* and *computer*, while query 2 returns all documents with 1's in the column *program*, excluding d_6, because the latter has 1 in the column *programming*. The proximity query (number 3), however, cannot be answered using a Boolean representation. This is because information about the term positions (offsets) in the document is lost. The problem can be solved by using a richer representation that includes the position for each occurrence of a term. In this case, each cell of the term–document matrix contains a list of integers that represent the term offsets for each of its occurrences in the corresponding document. Table 1.3 shows the version of the term–document matrix from Table 1.2 that includes term positions. Having this representation, the proximity query can also be answered. For document d_{14} (Music department) the matrix shows the following position lists: [42] for *lab* and [41] for *computer*. This clearly shows that the two terms are adjacent and appear in the phrase *computer lab*.

The term position lists also show the term frequencies (the length of these lists). For example, the term *computer* occurs six times in the Computer Science document and once in the Biology, Chemistry, Mathematics, and Music documents. Obviously, this is a piece of information that shows the importance of this particular feature for those documents. Thus, if *computer* is the query term, clearly the most relevant document returned would be Computer Science. For the other four documents, additional keywords may be needed to get a more precise *relevance ranking*. These issues are further discussed in the next sections.

TABLE 1.3 Term–Document Matrix with Term Positions

Document ID	*lab*	*laboratory*	*programming*	*computer*	*program*
d_1	0	0	0	0	[71]
d_2	0	0	0	0	[7]
d_3	0	[65,69]	0	[68]	0
d_4	0	0	0	[26]	[30,43]
d_5	0	0	0	0	0
d_6	0	0	[40,42]	[1,3,7,13,26,34]	[11,18,61]
d_7	0	0	0	0	[9,42]
d_8	0	0	0	0	[57]
d_9	0	0	0	0	0
d_{10}	0	0	0	0	0
d_{11}	0	0	0	0	0
d_{12}	0	0	0	[17]	0
d_{13}	0	0	0	0	0
d_{14}	[42]	0	0	[41]	[71]
d_{15}	0	0	0	0	[37,38]
d_{16}	0	0	0	0	[81]
d_{17}	0	0	0	0	[68]
d_{18}	0	0	0	0	0
d_{19}	0	0	0	0	[51]
d_{20}	0	0	0	0	0

Implementation Considerations

The Boolean representation of a term–document matrix is simple and can easily be implemented as a relational table. We use this representation later in the book for the purposes of document classification and clustering. However, for large document collections (such as those used by search engines) and for incorporating term positions, the amount of space needed is too large and does not allow straightforward implementation using a relational database. In these cases more advanced methods such as *B-trees* and *hash tables* are used. The idea is to implement the mappings directly from terms to documents and term positions. For example, the following structures can be used for this purpose:

$$
\begin{aligned}
&\textit{lab} \rightarrow d_{14}/42\\
&\textit{laboratory} \rightarrow d_3/65, 69\\
&\textit{programming} \rightarrow d_6/40, 42\\
&\textit{computer} \rightarrow d_3/68; d_4/26; d_6/1, 3, 7, 13, 26, 34; d_{12}/17; d_{14}/41
\end{aligned}
$$

There are two problems associated with this representation:

1. The efficiency of creating the data structure implementing the index
2. The efficiency of updating the index

Both issues are critical, especially for the indices used by web search engines. To get an idea of the magnitude of the problem, we provide here some figures from experiments performed with the GOV2 collection reported at the Text Retrieval Conference 2004-terabyte (TB) track. The GOV2 document collection is 426 GB and contains 25 million documents taken from the .gov web domain, including HTML and text, plus the extracted text of PDF, Word, and postscript files. For one of the submissions to this track (Indri), the index size was 224 GB and took 6 hours to build on a cluster of six computers. Given these figures, we can also get an idea about the indices build by web search engines. Assuming a web document collection of 20 billion documents (the size of the document collection that Yahoo! claimed to index in August 2005), its size can be estimated to be 500 TB (for comparison, the books in the U.S. Library of Congress contain approximately 20 TB of text). Simple projection suggests an index size of about 200 TB and an indexing time of 6000 hours (!). This amount of memory can be managed by recent technology. Moreover, there exist compression techniques that can substantially reduce the memory requirements. This indexing time is, however, prohibitive for search engines because the web pages change at a much quicker rate. The web indices should be built quickly and, most important, updated at a rate equal to the average rate of updating web pages.

There is another important parameter in indexing and search: the *query time*. It is assumed that this time should be in the range of seconds (typically, less than a second). The problem is that when the index is compressed, the time to update it and the access time (query time) both increase. Thus, the concern is to find the right balance between memory and time requirements (a version of the time–space complexity trade-off well known in computing).

Relevance Ranking

The Boolean keyword search is simple and efficient, but it returns a set (unordered collection) of documents. As we mentioned earlier, information retrieval queries are not well defined and cannot uniquely identify the resulting documents. The average size of a web search query is two terms. Obviously, such a short query cannot specify precisely the information needs of web users, and as a result, the response set is large and therefore useless (imagine getting a list of a million documents from a web search engine in random order). One may argue that users have to make their queries specific enough to get a small set of all relevant documents, but this is impractical. The solution is to rank documents in the response set by relevance to the query and present to the user an ordered list with the top-ranking documents first. The Boolean term–document matrix cannot, however, provide ordering within the documents matching the set of keywords. Therefore, additional information about terms is needed, such as counts, positions, and other context information. One straightforward approach is to incorporate the term count (frequencies). This is done in the term frequency–inverse document frequency (TFIDF) framework used widely in IR and Web search. Other approaches using positions and lexical and web context are discussed in later sections.

Vector Space Model

The *vector space model* defines documents as vectors (or points) in a multidimensional Euclidean space where the axes (dimensions) are represented by terms. Depending on the type of vector components (coordinates), there are three basic versions of this representation: Boolean, term frequency (TF), and term frequency–inverse document frequency (TFIDF).

Assume that there are n documents $d_1, d_2, \ldots, d_n$ and m terms $t_1, t_2, \ldots, t_m$. Let us denote as n_{ij} the number of times that term t_i occurs in document d_j. In a *Boolean representation*, document d_j is represented as an m-component vector $\vec{d}_j = (d_j^1\, d_j^2 \cdots d_j^m)$, where[2]

$$d_j^i = \begin{cases} 0 & \text{if } n_{ij} = 0 \\ 1 & \text{if } n_{ij} > 0 \end{cases}$$

For example, in Table 1.2 the documents from our department collection are represented in five-dimensional space, where the axes are *lab*, *laboratory*, *programming*, *computer*, and *program*. In this space the Computer Science document is represented by the Boolean vector

$$\vec{d}_6 = (0\ \ 0\ \ 1\ \ 1\ \ 1)$$

As we mentioned earlier, the Boolean representation is simple, easy to compute, and works well for document classification and clustering. However, it is not suitable for keyword search because it does not allow document ranking. Therefore, we focus here on the TFIDF representation.

In the *term frequency* (TF) approach, the coordinates of the document vector $\vec{d}_j$ are represented as a function of the term counts, usually normalized with the document length. For each term t_i and each document d_j, the TF (t_i, d_j) measure is computed. This can be done in different ways; for example:

- Using the sum of term counts over all terms (the total number of terms in the document):

$$\mathrm{TF}(t_i, d_j) = \begin{cases} 0 & \text{if } n_{ij} = 0 \\ \dfrac{n_{ij}}{\sum_{k=1}^{m} n_{kj}} & \text{if } n_{ij} > 0 \end{cases}$$

- Using the maximum of the term count over all terms in the document:

$$\mathrm{TF}(t_i, d_j) = \begin{cases} 0 & \text{if } n_{ij} = 0 \\ \dfrac{n_{ij}}{\max_k n_{kj}} & \text{if } n_{ij} > 0 \end{cases}$$

- Using a log scale to condition the term count (this approach is used in the Cornell SMART system [3]):

$$\mathrm{TF}(t_i, d_j) = \begin{cases} 0 & \text{if } n_{ij} = 0 \\ 1 + \log(1 + \log n_{ij}) & \text{if } n_{ij} > 0 \end{cases}$$

[2] For compactness of presentation here and throughout the book, we interchange the row and column notation for vectors where appropriate.

This approach does not use the document length; rather, the counts are just smoothed by the log function.

In the Boolean and TF representations, each coordinate of a document vector is computed locally, taking into account only the particular term and document. This means that all axes are considered to be equally important. However, terms that occur frequently in documents may not be related to the content of the document. This is the case with the term *program* in our department example. Too many vectors have 1's (in the Boolean case) or large values (in TF) along this axis. This in turn increases the size of the resulting set and makes document ranking difficult if this term is used in the query. The same effect is caused by stopwords such as *a*, *an*, *the*, *on*, *in*, and *at* and is one reason to eliminate them from the corpus.

The basic idea of the *inverse document frequency* (IDF) approach is to scale down the coordinates for some axes, corresponding to terms that occur in many documents. For each term t_i the IDF measure is computed as a proportion of documents where t_i occurs with respect to the total number of documents in the collection. Let $D = \bigcup_1^n d_j$ be the document collection and D_{t_i} the set of documents where term t_i occurs. That is, $D_{t_i} = \{d_j | n_{ij} > 0\}$. As with TF, there are a variety of ways to compute IDF; some take a simple fraction $|D|/|D_{t_i}|$, others use a log function such as

$$\text{IDF}(t_i) = \log \frac{1 + |D|}{|D_{t_i}|}$$

In the TFIDF representation each coordinate of the document vector is computed as a product of its TF and IDF components:

$$d_j^i = \text{TF}(t_i, d_j)\text{IDF}(t_i)$$

To illustrate the approach we represent our department documents in the TFIDF framework. First we need to compute the TF component for each term and each document. For this purpose we use a term–document matrix with term positions (Table 1.3) to get the counts n_{ij}, which are equal to the length of the lists with positions. These counts then have to be scaled with the document lengths (the number of terms taken from Table 1.1). The result of this is shown in Table 1.4, where the vectors are rows in the table (the first column is the vector name and the rest are its coordinates).

Note that the coordinates of the document vectors changed their scale, but relative to each other they are more or less the same. This is because the factors used for scaling down the term frequencies are similar (documents are similar in length). In the next step, IDF will, however, change the coordinates substantially.

Using the log version of the IDF measure, we get the following factors for each term (in decreasing order):

lab	*laboratory*	*programming*	*computer*	*program*
3.04452	3.04452	3.04452	1.43508	0.559616

These numbers reflect the specificity of each term with respect to the document collection. The first three get the biggest value, as they occur in only one document each. The term *computer* occurs in five documents and *program* in 11. The document vector

TABLE 1.4 Document Vectors with TF Coordinates

Document ID	TF Coordinates				
$\vec{d}_1$	0	0	0	0	0.012
$\vec{d}_2$	0	0	0	0	0.010
$\vec{d}_3$	0	0.022	0	0.011	0
$\vec{d}_4$	0	0	0	0.017	0.034
$\vec{d}_5$	0	0	0	0	0.011
$\vec{d}_6$	0	0	0.026	0.078	0.039
$\vec{d}_7$	0	0	0	0	0.033
$\vec{d}_8$	0	0	0	0	0.013
$\vec{d}_9$	0	0	0	0	0
$\vec{d}_{10}$	0	0	0	0	0
$\vec{d}_{11}$	0	0	0	0	0
$\vec{d}_{12}$	0	0	0	0.015	0
$\vec{d}_{13}$	0	0	0	0	0
$\vec{d}_{14}$	0.011	0	0	0.011	0.011
$\vec{d}_{15}$	0	0	0	0	0.037
$\vec{d}_{16}$	0	0	0	0	0.010
$\vec{d}_{17}$	0	0	0	0	0.012
$\vec{d}_{18}$	0	0	0	0	0
$\vec{d}_{19}$	0	0	0	0	0.015
$\vec{d}_{20}$	0	0	0	0	0

TF components are now multiplied by the IDF factors. In this way the vector coordinates corresponding to rare terms (*lab*, *laboratory*, and *programming*) increase, and those corresponding to frequent ones (*computer* and *program*) decrease. For example, the Computer Science (CS) document vector with TF only is

$$\vec{d}_6 = (0 \;\; 0 \;\; 0.026 \;\; 0.078 \;\; 0.039)$$

whereas after applying IDF, it becomes

$$\vec{d}_6 = (0 \;\; 0 \;\; 0.079 \;\; 0.112 \;\; 0.022)$$

In this vector the term *computer* is still the winner (obviously, the most important term for CS), but the vector is now stretched out along the *programming* axis, which means that the term *programming* is more relevant to identifying the document than the term *program* (quite true for CS, having in mind that *program* also has other non-CS meanings).

Document Ranking

In the Boolean model the query terms are simply matched against the document vectors, and the documents that match the query exactly are returned. In the TFIDF model, exact matching is not possible; therefore, we need some *proximity measure* between the query and the documents in the collection. The basic idea is to represent the query as a vector (called a *query vector*) in the document vector space and then

to use the metric properties of vector spaces. For this purpose we first consider the keyword query as a document. For example, the query that is supposed to return all documents containing the terms *computer* and *program* is represented as a document $q = \{computer, program\}$. As each term occurs once, its TF component is $\frac{1}{2}$ (normalized with the document length of 2). Thus, the TF vector in five-dimensional space is

$$\vec{q} = (0 \;\; 0 \;\; 0 \;\; 0.5 \;\; 0.5)$$

which after scaling with IDF becomes

$$\vec{q} = (0 \;\; 0 \;\; 0 \;\; 0.718 \;\; 0.28)$$

When we specify a Boolean query we usually assume that the terms are equally important for the document we are looking for. However, it appears that the *importance of the keywords depends on the document collection*. Thus, the search engine automatically adjusts the importance of each term in the query. For example, the term *computer* seems to be more important than *program* simply because *program* is a more common term (occurs in more documents) in this particular collection. The situation may change if we search a different collection of documents (e.g., in the area of CS only).

Given a query vector $\vec{q}$ and document vectors $\vec{d}_j$, $j = 1, 2, \ldots, 20$, the objective of a search engine is to order (rank) the documents with respect to their proximity to $\vec{q}$. The result list should include a number of top-ranked documents. There are several approaches to this type of ranking. One option is to use the *Euclidean norm of the vector difference* $\|\vec{q} - \vec{d}_j\|$, defined as

$$\|\vec{q} - \vec{d}_j\| = \sqrt{\sum_{i=1}^{m} \left(q^i - d_j^i\right)^2}$$

This measure is, in fact, the *Euclidian distance* between the vectors considered as points in Euclidean space, and being a *metric function*, it has some nice properties, such as the triangle inequality. However, it depends greatly on the length of the vectors to be compared. This property is not in agreement with one of the basic assumptions in IR: that similar documents (in terms of their relevance to the query) also have to be close in the vector space. For example, a large and a small document will be at a great distance even though they may both be relevant to the same query. To avoid this, the document and the query vectors are normalized to unit length before taking the vector difference. This approach still has a drawback because queries are very short and when scaled down with the query length (typically, 2), their vectors tend to be at a great distance from large documents.

Another approach is to use the cosine of the angle between the query vector and the document vectors. When the vectors are normalized, this measure is equivalent to the *dot product* $\vec{q} \cdot \vec{d}_j$, defined as

$$\vec{q} \cdot \vec{d}_j = \sum_{i=1}^{m} q^i d_j^i$$

This measure, known as *cosine similarity*, is the one used primarily in IR and web search.

TABLE 1.5 Cosine Similarity and Distances with $\vec{q} = (0\ 0\ 0\ 0.932\ 0.363)$

Document ID	TFIDF Coordinates (Normalized)					$\vec{q} \cdot \vec{d}_j$ (rank)[a]	$\lvert\vec{q} - \vec{d}_j\rvert$ (rank)[a]
$\vec{d}_1$	0	0	0	0	1	0.363	1.129
$\vec{d}_2$	0	0	0	0	1	0.363	1.129
$\vec{d}_3$	0	**0.972**	0	0.234	0	0.218	1.250
$\vec{d}_4$	**0**	**0**	**0**	**0.783**	**0.622**	**0.956 (1)**	**0.298 (1)**
$\vec{d}_5$	0	0	0	0	1	0.363	1.129
$\vec{d}_6$	**0**	**0**	**0.559**	**0.811**	**0.172**	**0.819 (2)**	**0.603 (2)**
$\vec{d}_7$	0	0	0	0	1	0.363	1.129
$\vec{d}_8$	0	0	0	0	1	0.363	1.129
$\vec{d}_9$	0	0	0	0	0	0	1
$\vec{d}_{10}$	0	0	0	0	0	0	1
$\vec{d}_{11}$	0	0	0	0	0	0	1
$\vec{d}_{12}$	0	0	0	1	0	0.932	0.369
$\vec{d}_{13}$	0	0	0	0	0	0	1
$\vec{d}_{14}$	**0.890**	**0**	**0**	**0.424**	**0.167**	**0.456 (3)**	**1.043 (3)**
$\vec{d}_{15}$	0	0	0	0	1	0.363	1.129
$\vec{d}_{16}$	0	0	0	0	1	0.363	1.129
$\vec{d}_{17}$	0	0	0	0	1	0.363	1.129
$\vec{d}_{18}$	0	0	0	0	0	0	1
$\vec{d}_{19}$	0	0	0	0	1	0.363	1.129
$\vec{d}_{20}$	0	0	0	0	0	0	1

[a] The rank is shown only for documents that include both terms (*computer* and *program*)

Table 1.5 illustrates the query processing and document ranking approach discussed so far with the department example. The query is "*computer* AND *program*," represented by the normalized query vector $\vec{q} = (0\ 0\ 0\ 0.932\ 0.363)$. The document vectors are generated from those shown in Table 1.4 by applying IDF scaling and normalization. The last two columns show the cosine similarity (dot product) and the distance (norm of the vector difference) between those vectors and the query vector. The documents that include both terms (*computer* and *program*) are emphasized and their ranking is shown in parentheses.

First let us look at the document vectors. Those with just one nonzero coordinate look like Boolean vectors. This is because of the normalization step, which scales the coordinates so that the vector norm is equal to 1. Another interesting effect due to normalization is demonstrated by vectors $\vec{d}_6$ and $\vec{d}_{12}$. Both documents include the term *computer*, but the TFIDF component for *computer* in $\vec{d}_6$ is lower than the one in $\vec{d}_{12}$. The explanation is that the normalization step scaled up the *computer* coordinate of $\vec{d}_{12}$ to 1 because that was the only nonzero coordinate, whereas the same coordinate of $\vec{d}_6$ was scaled down due to the presence of two other nonzero coordinates. Generally, this shows the *importance of the choice of terms to represent documents*. In this particular case the problem is caused by the limited number of terms used (only five). One straightforward solution is to use all 671 terms that occur in the entire document collection. However, in large collections the number of terms is usually tens of thousands, and most important, they are not distributed uniformly

over the documents. Moreover, the documents are of different lengths, which again may cause a lot of 0's in the document vectors. All this results in *extremely sparse distribution* of the document vectors, especially those collected from the Web. In this respect the sparsely populated Table 1.5 seems to represent well the general situation with document vector space.

Table 1.5 shows the similarity of all document vectors with the query vector. However, to answer the query (*computer* AND *program*), only the documents that include both keywords need to be considered. They are $\vec{d}_4$ (Chemistry), $\vec{d}_6$ (Computer Science), and $\vec{d}_{14}$ (Music), in the order of their ranking. Interestingly, both measures, maximum dot product and minimum distance, agree on the relevance of these documents to the query. We would also like to have these three documents ranked at the top among all documents. Another desired property would be the existence of a cutoff value that would allow us to distinguish the exact Boolean match with all keywords. However, this is not the case here. The ordering of documents that do not match both keywords is indicative for the differences between the two proximity measures. The cosine similarity ordering seems more natural, while the distance ranking looks peculiar. For example, at distance 1 to the query the documents are represented by all-zero vectors [i.e., none of the terms used in the representation (the dimensions) occur in those documents]. Strangely, one of the matches with the query ($\vec{d}_{14}$) is farther from the query than the all-zero vectors. There is a similar situation with cosine similarity: Document vector $\vec{d}_{12}$, with just one nonzero component (the one that matches one of the keywords), has the second-highest score among all the documents, but obviously this is an exception. In general, the cosine similarity measure seems more stable with respect to the choice of terms, which in turn may explain why it is the preferred proximity measure for IR systems.

The results above suggest that terms have to be chosen such that the zero-valued coordinates of the vectors are minimized. One approach to achieving this is to use terms with high TF scores. For example, the term counts may be taken on the entire corpus and then the top frequency terms chosen as dimensions of the vector space. In this way we can have more nonzero components in each vector. However, as we have already seen, these frequent terms do not reflect the content and meaning of a document. In fact, the important terms are the more document-specific terms (i.e., those with high IDF scores). Thus, the question is how to balance the TF and IDF contributions when we choose terms (features) to represent documents. In a more general context, this problem, called *feature selection*, plays an important role in document classification and clustering. In later chapters we shall discuss it in more detail.

Relevance Feedback

Keyword queries are often incomplete or ambiguous. The response from such queries may not return the relevant documents that match user information needs or may include many irrelevant documents. So queries have to be specialized and refined, which is usually done through advanced search options, available in most search engines. This means, however, that the user needs to know more about the document searched, which contradicts the basic philosophy of information retrieval, which is

about search for information, not documents. The relevance feedback approach refines the query automatically using user feedback as to the relevance of the result. This can be done by providing some type of rating for each document in the result list. In the initial response these ratings may be the document ranks or simply binary labels indicating the relevance or irrelevance for each document. For example, the top 10 documents in the ranked list may be considered as relevant and the rest as irrelevant.

After the initial response the user evaluates the actual relevance of each document (e.g., by reading its content) and is provided with the option to change the relevance suggested by the system. This information, called *relevance feedback*, is then sent back to the search engine and the query is repeated. At this point the relevance feedback is used to adjust the original query vector. This can be done using *Rocchio's method*, a simple and popular technique known from early IR systems and used recently in related areas, such as machine learning. The idea is to update the query vector using a linear combination of the previous query vector $\vec{q}$ and the document vectors $\vec{d}_j$ of relevant and irrelevant documents. That is,

$$\vec{q}' = \alpha\,\vec{q} + \beta \sum_{d_j \in D_+} \vec{d}_j - \gamma \sum_{d_j \in D_-} \vec{d}_j$$

where α, β, and γ are adjustable parameters and D_+ and D_- are the sets of relevant and irrelevant documents provided by the user. These sets can also be determined automatically (the approach is then called *pseudorelevance feedback*): for example, by assuming that the top 10 documents returned by the original query belong to D_+ and the rest to D_-. Because the set of irrelevant documents is usually much larger, we may want not to use D_- (i.e., set $\gamma = 0$). Also, not all terms have to be included in the equation. The reason is that terms with high TF may occur in many documents and thus contribute too much to the corresponding component of the query vector. This would shift the focus to unimportant terms and may call up documents that are more irrelevant. To avoid this, terms are ordered in decreasing order by their IDF score, and a given number of terms from the top of the list (e.g., 10) are chosen.

To illustrate the approach, let us try to improve the search results shown in Table 1.5. Let $\alpha = 1$, $\beta = 0.5$, $\gamma = 0$, and D_+ be the set of three relevant documents returned by the original query. Let us also use only the top three terms from the list sorted by IDF score (*lab*, *laboratory*, and *programming*), thus excluding *computer* and *program*, which occur in more documents and have lower IDF scores (see the earlier table showing the IDF scores). The new query vector is computed as

$$\begin{aligned}\vec{q}' &= \vec{q} + 0.5\,(\vec{d}_4 + \vec{d}_6 + \vec{d}_{14}) \\ &= \begin{pmatrix}0\\0\\0\\0.932\\0.363\end{pmatrix} + 0.5\left[\begin{pmatrix}0.\\0\\0\\0\\0\end{pmatrix} + \begin{pmatrix}0\\0\\0.559\\0\\0\end{pmatrix} + \begin{pmatrix}0.89\\0\\0\\0\\0\end{pmatrix}\right] = \begin{pmatrix}0.445\\0\\0.28\\0\\0.363\end{pmatrix}\end{aligned}$$

Note that the last two coordinates of the document vectors are replaced with 0's, as we decided to exclude these terms due to low IDF scores. Before the second run the query vector is normalized to $\vec{q}' = (0.394\ \ 0\ \ 0.248\ \ 0.824\ \ 0.321)$. The resulting list of documents ranked by cosine similarity (shown in parentheses in Table 1.5) is now $d_6(0.863)$, $d_4(0.846)$, and $d_{14}(0.754)$. This ranking seems a little more natural because the Computer Science document (d_6), which has a higher count for the term *computer* (the more important query term), is now ranked before the Chemistry document (d_4), which has a smaller count for the same term. Also, the incorrectly ranked document from the original query d_{12} (now with cosine similarity 0.824) is one position down.

The general effect of pseudorelevance feedback is that the query becomes more similar to the relevant documents returned from the original query. Consequently, on the second run the relevant documents' vectors are grouped around bigger and more homogeneous vectors (with more uniform distribution of terms, such as d_6 in the example), and those with scattered terms (e.g., d_{12}) are pushed away. When the user provides the feedback, the group of relevant vectors may be moved toward a user-specified set of relevant vectors.

Relevance feedback is a standard technique in classical IR. However, it is not popular for web search mainly because web users generally expect instant results from their queries. Also, user feedback would increase the computational cost for handling the millions of queries that search engines have to deal with every second. The reason we include the discussion of relevance feedback here is that it contributes further to better understanding the vector space model and the TFIDF framework.

There also exist probabilistic approaches to relevance feedback that try to model the mapping between queries and relevant documents using statistical techniques. Basically, these techniques assume term independence (with respect to other terms, queries, and document relevance) and calculate conditional probabilities for document relevance. We are not discussing these approaches here because similar ones exist in the more general context of document classification, where the document relevance can be seen as a category (class) label of a document and *machine learning* techniques can be used to learn mappings between queries and documents, considering user feedback as a training set of examples. We discuss some of these techniques in later chapters.

Advanced Text Search

The commonly used text search queries include only individual terms, and by default most search engines assume that all of the terms specified must occur in the documents returned (they are implicitly AND-ed). Advanced search options also allow the use of "OR" or "NOT" Boolean operators. All these constraints can be implemented easily during the retrieval phase, when documents are looked up in the inverted index. After (or while) obtaining a set of documents that satisfy the Boolean Query, the TFIDF measure is used to compute the proximity of the document vectors to the query vector. This allows the documents retrieved to be ordered by relevance to the query.

Another advanced search option is *phrase search*. Documents that include given phrases can be retrieved using the standard term-based inverted index, as it also

contains term position information. We have illustrated this with the phrase *computer lab* (see Table 1.3) found in the Music document (d_{14}) because *computer* and *lab* occurred in successive positions (41 and 42) in that particular document.

Ranking documents retrieved by phrase search is, however, more difficult. Using combinations of the TFIDF measures of the terms that occur in the phrase is not appropriate because these measures are computed independently for the individual terms. So we need the TF and IDF values for the phrase itself. Once we have those measures, phrases can be added as new dimensions to the document vector space, and cosine similarity can be used for relevance ranking. Thus, the question is how to identify potentially useful phrases from a given corpus. A collection of phrases that occur in a corpus is called a *phrase dictionary*.

The phrase dictionary may be built manually or derived from the corpus automatically. Most approaches use statistical methods first to extract possible phrases and then linguistic tools or manual editing to refine the phrase dictionary. Phrases typically consist of two or three words. In a large corpus two or three words may occur together by chance or they may be a pattern (i.e., a phrase). The statistical approach tries to answer this question by estimating the probabilities of occurrences of the terms individually and as a phrase. For example, if two terms t_1 and t_2 are independent, the probability of their cooccurrence is $P(t_1\, t_2) = P(t_1)P(t_2)$. However if "$t_1\, t_2$" is a phrase, the probability $P(t_1\, t_2)$ would significantly differ from the product $P(t_1)P(t_2)$. Statistical tests such as likelihood ratio are used to determine this.

Phrases provide context for terms, but they play the same role as that played by individual terms: They add new features to the document model (dimensions in vector space). Another, richer context for terms is provided by *tagging*. We have already mentioned part-of-speech tagging, where words are associated with their role in the sentence and the same words with different tags are used as dimensions in vector space. This approach allows queries to be more specific and unambiguous.

So far we have assumed that keywords in queries can match exactly words that occur in documents. In practice, however, various languages and dialects are used and words are often misspelled. Thus, if only exact matching is used, many relevant documents may be missed. Generally, there are two approaches to solving this problem. One is to extend the process of stemming with some conflation mechanism that may handle misspelling and dialects. The difficulties with this approach are that such mechanisms are developed mostly for English and other Western languages. Also, a lot of common words are used with specific technical meaning.

The other approach is to try to find the closest match of the query term to terms in the inverted index. This can be done by *approximate string matching*. One popular approach for this is to decompose words into subsequences of characters with fixed length called *n-grams* (or *q-grams*). For example, the word *program* may be represented by a sequence of 2-grams as {pr, ro, og, gr, ra, am} and its misspelling *prorgam* as {pr, ro, or, rg, ga, am}. So they overlap in three of the six 2-grams and may be considered close.

To use n-grams in keyword search, the query term is first looked up in a index of n-grams (n is usually 2 to 4) and is slightly modified so that a set of variant terms is obtained. Then the inverted index is used with each one those terms. The closest match is determined by comparing the relevance of the documents retrieved.

Using the HTML Structure in Keyword Search

So far we have ignored the rich HTML structure of web documents. However, HTML tags provide a lot of context information that may be very useful in keyword search. Basically, the tags that add to or modify the meaning of web page text are important for this purpose. These are:

- *Titles and metatags that provide meta information about the web page*. For example, the following fragments from the Music page (Figure 1.4) provide information about the title of the page, its authors, and the software used to create it:

  ```
  <title>Music</title>
  <meta name="Author" content="John Smith ">
  <meta name="GENERATOR" content="Microsoft FrontPage 5.0">
  ```

  This information is included in the "head" area of the web page, and with the exception of the title is not displayed by the browser.

- *Headings and font modifiers used to separate or emphasize parts of the text* (e.g., <h2> · · · </h2>, · · · , · · · , <p>,
). For example, the title of the web page is generated with the following structure:

  ```
  <h2 align="center">
  <font color="#000080"><big>Music</big></font>
  </h2>
  ```

- *Anchor text*. For example, the following anchor occurs in the department directory page (Figure 1.3):

  ```
  <a href="ASLinks/Chairs.html" target="_top">
  <font color="#000000">
  Department Chairs, Locations, Phone Numbers<br>
  </font></a>
  ```

  The anchor text here explains briefly the content of the page to which it links.

HTML tags have two basic uses in web search. First, the terms that occur in their context may be tagged and indexed accordingly. For this purpose the main index can be extended with tagged terms, or separate indices can be built for faster access. This will allow web documents to be retrieved by specific parts of their HTML structure. For example, Google advanced search options allow specifying exactly where the terms should occur in the page: in the title, in the text (excluding the title), in the page URL, or in links (anchor text) pointing to the page. Some of these HTML structures may even replace full text indexing. For example, one of the early versions of Google built at Stanford University indexed only the titles of 16 million web pages and was very successful because of the small and efficient index (and also because of the use of hyperlink-based ranking, which we discuss later). However, the lack of authority and editorial control on web publishing allows many web pages to have no titles, or titles that are irrelevant to the page content. The same is true for other tags generally

supposed to provide metainformation about the web page. All this made the designers of web search engines take the full-text indexing approach.

The other use of HTML tags is for relevance ranking. The specific HTML scope where keyword terms occur in a document may affect its ranking. This can be achieved by assigning different weights to terms occurring in different HTML structures. These weights are then used to modify the corresponding TFIDF components of the query and document vectors, which in turn affect their cosine similarity and consequently the relevance ranking of the documents in the response. Typically, words in titles, emphasized text, heading, and anchors may get higher scores and thus increase the relevance score of the documents in which they occur. This approach was popular in the early search engines and worked well for providing more natural relevance ranking. The reason for this is that these are techniques used in traditional typesetting and, more recently, in web page design to emphasize important words and phrases that have high relevance to the document content and meaning.

With the appearance of spam, however, HTML tags became a tool for making search engines index web pages with content irrelevant to the indexed terms or for getting top ranking in search engines. One popular way to achieve this was to include in the web page invisible words (text with the same foreground and background color) that will be indexed by search engines but will not be seen by web users. Metatags were also used by spammers to get top ranking in search engines because the metainformation they include is not displayed by browsers but is taken into account for relevance ranking. All this shifted the emphasis of search engines from the HTML tags to the page hyperlink structure. We discuss link-based ranking in detail in Chapter 2.

Still, one HTML structure plays a significant role in web page indexing and search. This is the *anchor tag*, which actually implements the main feature of the web pages, the hyperlinks. As we mentioned earlier, the purpose of web search is to access unstructured data by content. The discussion so far was focused on the approaches to model the web page content. Hyperlinks and especially, *anchor text* provide additional content description of web pages. For example, the anchor text "Department Chairs, Locations, Phone Numbers" (from the A&S directory page shown in Figure 1.4) includes the term *phone*, which in fact is not present in the content of the page to which it points (`http://www.artsci.ccsu.edu/ASLinks/Chairs.html`). The latter contains a table in which the phone numbers are listed in a column named "Ext." Obviously, when crawled and indexed, this page will not be included in the index entry for the term *phone* and consequently cannot be retrieved by keyword search with the term *phone*. However, this term may be taken from the anchor text that points to the page and is included in the set of terms representing the document. Weight that would increase the TFIDF score of the document vector along this dimension may also be assigned to such external terms, because they are often more relevant to the page content than are terms from the original page. The reason for this is that the pages that link to a particular page provide independent and authoritative judgment about its content. In some cases the anchor text may be used for indexing instead of the actual page content. For example, the web page with the phone numbers mentioned above can be indexed by all the terms that occur in the anchor text pointing to it: *department*, *chairs*, *locations*, *phone*, and *numbers*. More terms may be collected from other pages pointing to it. This idea was implemented in one of the first search

engines, the World Wide Web Worm system [4], and later used by Lycos and Google. This allows search engines to increase their indices with pages that have never been crawled, are unavailable, or include nontextual content that cannot be indexed, such as images and programs. As reported by Brin and Page [5] in 1998, Google indexed 24 million pages and over 259 million anchors.

EVALUATING SEARCH QUALITY

Information retrieval systems do not have formal semantics (such as that of databases), and consequently, the query and the set of documents retrieved (the response of the IR system) cannot be mapped one to one. Therefore, some measures are used to evaluate the degree of fitness (accuracy) of the response. A standard benchmark for this purpose is the recall-precision measure, which is also used in related areas (such as machine learning and data mining).

Assume that there is a set of queries Q and a set of documents D, and for each query $q \in Q$ submitted to the system we have:

- The response set of documents (retrieved documents) $R_q \subseteq D$
- The set of relevant documents D_q selected manually from the entire collection of documents D (i.e., $D_q \subseteq D$)

The proportion of retrieved relevant documents to all retrieved documents is called *precision* and is defined as

$$\text{precision} = \frac{|D_q \cap R_q|}{|R_q|}$$

Clearly, the value of the precision is between 0 and 1: where 0 is the worst case—no relevant documents are retrieved—and 1 is the best case—all documents retrieved are relevant. Precision 1 is not, however, all that an ideal IR system should provide, because there may be relevant documents that are not retrieved. *Recall* is the measure that accounts for this. It represents the proportion of relevant documents retrieved to all relevant documents. Formally,

$$\text{recall} = \frac{|D_q \cap R_q|}{|D_q|}$$

Again, the best case is 1—all relevant documents are retrieved—and the worst case is 0—no relevant documents are retrieved.

Recall and precision determine the relationship between two sets of documents: relevant (D_q) and retrieved (R_q). Ideally, these sets coincide (precision and recall are both 1), but this never happens in real systems. Generally, there is some overlap ($D_q \cap R_q \subset D_q$) which we would like to maximize, or the set retrieved is too large ($D_q \subset R_q$) and we want to exclude from it documents that are irrelevant. Interestingly, achieving the maximum value for each of the two measures individually is trivial. By using a very general query (e.g., a query including terms that occur in all documents), the response set will be the entire collection of documents D, and thus the recall

will be 1. However, the precision will be low because all irrelevant documents will also be included in the response. Inversely, with a very restrictive query, a small subset of relevant-only documents may be retrieved easily. For example, one of the relevant documents may be used as a query, and then the precision will be 1. These observations suggest that there is an important trade-off between precision and recall. A plot of precision against recall generally slopes down with increasing recall. Thus, a better IR system will have its recall–precision curve above that of a poorer system.

The set-valued recall–precision framework is oversimplified and is commonly used only to illustrate the general idea or in areas where ranking is not possible. Real IR systems, such as web search engines, return thousands of documents. Considering them as a set as well as computing the set D_q is practically impossible. Obviously, the document ranks have to be taken into consideration. For this purpose we modify the setting as follows. Consider that the response to a query q is now not a set but a *list* $R_q = (d_1, d_2, \ldots, d_m)$ of *ranked documents* (highest ranks first). Then using the set of relevant documents D_q, for each document $d_i \in R_q$ we can compute its relevance r_i as a Boolean value. That is,

$$r_i = \begin{cases} 1 & \text{if } d_i \in D_q \\ 0 & \text{otherwise} \end{cases}$$

We also add a parameter $k \geq 0$ that represents the number of documents from the top of the list R_q that we consider. Thus, we define *precision* at rank k as

$$\text{precision } (k) = \frac{1}{k} \sum_{i=1}^{k} r_i$$

and *recall* at rank k as

$$\text{recall } (k) = \frac{1}{|D_q|} \sum_{i=1}^{k} r_i$$

If we fix k and consider the top k elements from R_q as a set, the new measures work exactly the same as the set-based measures. The parameter k allows us to see how recall and precision change with increasing k (i.e., decreasing rank). The *average precision* is the measure that accounts for this:

$$\text{average precision} = \frac{1}{|D_q|} \sum_{k=1}^{|D|} r_k \times \text{precision } (k)$$

The average precision is a useful measure that combines precision and recall and also evaluates document ranking. The maximal value of average precision is 1, reached when all relevant documents are retrieved and ranked in the response list before any irrelevant documents. Note that the sum goes over all documents in the collection D. Although the system provides ranking only for the documents in the response list R_q, we assume that all documents in D are ordered by their rank. In practice, to compute the average precision we first go over the ranked documents from the response list R_q and then continue with the rest of the documents from D. Also, we assume that r_i's are computed for all documents in D. Thus, the maximum of 1 is reached when R_q

TABLE 1.6 Document Ranking, Relevance, Recall, and Precision

k	Document Index	r_k	recall (k)	precision (k)
1	4	1	0.333	1
2	12	0	0.333	0.5
3	6	1	0.667	0.667
4	14	1	1	0.75
5	1	0	1	0.6
6	2	0	1	0.5
7	3	0	1	0.429
8	5	0	1	0.375
9	7	0	1	0.333
10	8	0	1	0.3
11	15	0	1	0.273
12	16	0	1	0.25
13	17	0	1	0.231
14	19	0	1	0.214
15	9	0	1	0.2
16	10	0	1	0.188
17	11	0	1	0.176
18	13	0	1	0.167
19	18	0	1	0.158
20	20	0	1	0.15

includes *all* relevant documents. It may also include irrelevant ones, but they should occur after the relevant documents.

To generate a plot of precision against recall, *interpolated precision* is used. First, the actual recall levels *recall(k)* are computed for each k corresponding to a relevant document from set D_q, that is, for those k's for which $r_k = 1$. Then for each standard value of recall ρ (e.g., $\rho = 0,\ 0.1,\ 0.2, \ldots, 1$) the interpolated precision is the maximum precision computed for recall levels greater than or equal to ρ (interpolated precision is defined as 0 for recall 0). To get the average performance of an IR system on a set of queries Q at each level of recall, the interpolated precision is averaged over all $q \in Q$.

Let us illustrate the recall–precision evaluation technique with our department example. The initial data needed for this purpose include the ranking of all documents and the corresponding r_k's. As the ranking shown in Table 1.5 is nearly perfect, we modify it a bit to get a more interesting situation. The new ranking that we are going to evaluate here is based purely on the cosine similarity (the original ranking was done only on documents that include both keywords). The relevant documents that form the list R_q (determined manually) are d_4 (Chemistry), d_6 (Computer Science), and d_{14} (Music): that is, those that include both *computer* and *program*. Thus, we rank all documents in the collection as shown in Table 1.6.

As the recall values increase with k, the precision interpolated at each standard recall level ρ is computed as the maximum precision in all rows, starting with the first one (from the top) in which the actual recall value is greater than or equal to ρ. Thus,

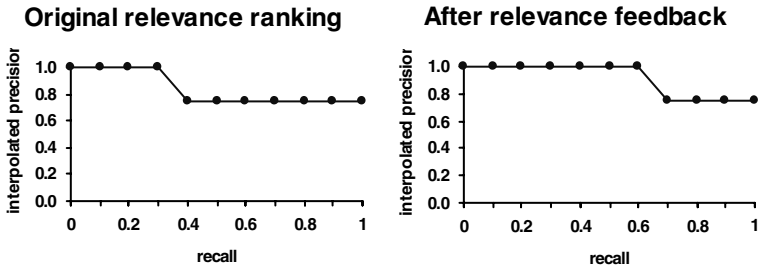

Figure 1.5 Interpolated precision against recall before and after relevance feedback.

for recall levels of 0, 0.1, 0.2, and 0.3, the interpolated precision of 1 is computed as the maximum precision on rows 1 to 20. For recall levels 0.4, 0.5, and 0.6, the interpolated precision is 0.75 (maximum precision on rows 3 to 20), and it is also 0.75 for levels 0.7 to 1 (maximum precision on rows 4 to 20). The plot of the precision interpolated against recall computed as described is shown on the left in Figure 1.5. For comparison the right side of the figure shows the precision against recall for the ranking produced by Rocchio's method as described in the section "Relevance Feedback" (the sequence of r_k's starts with $1, 1, 0, 1, 0, \ldots$). The curve on the right of the figure is above that on the left, which indicates that relevance feedback improves the performance of an IR system.

Let us also compute the average precision for the two rankings shown in Figure 1.5. We use Table 1.6 and average the precision in the rows where $r_k = 1$(rows 1, 3, and 4). Thus, for the original ranking we have

$$\text{average precision} = \tfrac{1}{3}(1 + 0.667 + 0.75) = 0.806$$

The relevance feedback swaps the values of r_2 and r_3 and changes the precisions accordingly. Thus, we have

$$\text{average precision} = \tfrac{1}{3}(1 + 1 + 0.75) = 0.917$$

Clearly, the average precision also indicates that the relevance feedback improves document ranking.

The recall–precision framework is a useful method for evaluating IR system performance. It is, however, important to note that it has its limitations. As we already mentioned for large document collections such as the Web, it is not possible to use the response set R_q explicitly. For example, a Web search with the query that we have used for our small collection of 20 documents, *computer program*, submitted to Google returns about 504 million documents (!). Of course, the recall and precision can be computed on the first 10 or 20 documents, which is still useful. There is another critical issue, however, that may require a different approach. The classical IR recall–precision evaluation is based entirely on the document content (the TFIDF vector space model). As we shall see in Chapter 2, other measures, such as popularity and authority, are also important and have to be taken into account.

SIMILARITY SEARCH

We have assumed that web user information needs are represented by keyword queries, and thus document relevance is defined in terms of how close a query is to documents found by the search engine. Because web search queries are usually incomplete and ambiguous, many of the documents returned may not be relevant to the query. However, once a relevant document is found, a larger collection of possibly relevant documents may be found by retrieving documents similar to the relevant document. This process, called *similarity search*, is implemented in some search engines (e.g., Google) as an option to find pages similar or related to a given page. The intuition behind similarity search is the *cluster hypothesis* in IR, stating that documents similar to relevant documents are also likely to be relevant. In this section we discuss mostly approaches to similarity search based on the content of the web documents. Document similarity can also be considered in the context of the web link structure. The latter approach is discussed briefly in the section "Authorities and Hubs" in Chapter 2.

Cosine Similarity

The query-to-document similarity which we have explored so far is based on the vector space model. Both queries and documents are represented by TFIDF vectors, and similarity is computed using the metric properties of vector space. It is also straightforward to compute similarity between documents. Moreover, we may expect that document-to-document similarity would be more accurate than query-to-document similarity because queries are usually too short, so their vectors are extremely sparse in the highly dimensional vector space. Thus, given a document d and a collection D, the problem is to find a number (usually, 10 or 20) of documents $d_i \in D$ which have the largest value of cosine similarity to d: that is, the maximum value of the dot product $\vec{d}.\vec{d_i}$.

In similarity search we are not concerned with a small query vector, so we are free to use more (or all) dimensions of the vector space to represent our documents. In this respect it will be interesting to investigate how the dimensionality of vector space affects the similarity search results. This issue is related to *feature selection*, a problem that we mentioned earlier and will revisit later. Generally, several options can be explored:

- *Using all terms from the corpus*. This is the easiest option but may cause problems if the corpus is too large (such as the document repository of a web search engine).
- *Selecting terms with high TF scores* (usually based on the entire corpus). This approach prefers terms that occur frequently in many documents and thus makes documents look more similar. However, this similarity is not indicative of document content.
- *Selecting terms with high IDF scores*. This approach prefers more document-specific terms and thus better differentiates documents in vector space. However,

it results in extremely sparse document vectors, so that similarity search is too restricted to closely related documents.

- *Combining the TF and IDF criteria*. For example, a preference may be given to terms that maximize the product of their TF (on the entire corpus) and TDF scores. As this is the same type of measure used in the vector coordinates (the difference is that the TF score in the vector is taken on the particular document), the vectors will be better populated with nonzero coordinates. We explore this option when we discuss document clustering and classification later in the book.

Let us illustrate the similarity search basics with our department example. We shall represent all documents as TFIDF vectors with different dimensionality (to examine the effect of feature selection). For this purpose we create two lists, including all 671 terms in the corpus (see the basic statistics in Table 1.1), one ordered by their global (for the corpus) frequency scores and another ordered by their IDF scores. The two halves of Table 1.7 give us an idea of what these lists look like. The left half shows the 10 most frequent terms along with their frequencies, IDF scores, and the number of documents in which they occur. The right half shows the terms with the top 10 IDF scores, their frequency counts, and the number of documents in which they occur.

In fact, the table shows the beginning and end of the frequency-ordered list of terms and illustrates nicely the basic properties of the IDF measure. The highest-frequency terms usually occur in many documents and have low IDF scores. Using these terms as dimensions of vector space would make all documents too similar. On the other hand, each of the top IDF-scored terms occurs in one document only (these are, in fact, the department names). Obviously, in a 20-dimensional vector space created with the top 20 terms from this list, the documents will be represented by orthogonal vectors and thus will be perfectly differentiated. However, in such a space all vectors will be equally dissimilar one to another (with cosine similarity 0).

To further illustrate the observations above, increasingly large samples are taken from the top of the TF list, the IDF list, and the TF∗IDF list (ordered by the product of TF and IDF) and the documents are ordered by their cosine similarity to the Computer

TABLE 1.7 Term Frequencies and IDF Scores

Term	Count	IDF	Docs.	Term	IDF	Count	Docs.
department	65	0.049	20	*english*	3.045	5	1
study	28	0.049	20	*psychology*	3.045	5	1
students	26	0.336	15	*chemistry*	3.045	6	1
ba	22	0.272	16	*communication*	3.045	6	1
website	21	0.049	20	*justice*	3.045	7	1
location	21	0.049	20	*criminal*	3.045	8	1
programs	21	0.405	14	*theatre*	3.045	8	1
832	20	0.100	19	*anthropology*	3.045	9	1
phone	20	0.049	20	*sociology*	3.045	10	1
chair	20	0.049	20	*music*	3.045	12	1

TABLE 1.8 Experiments with Cosine Similarity

Sample	$i/\vec{d}_i \cdot \vec{d}_6$ (Indices of Documents Ordered by $\vec{d}_i \cdot \vec{d}_6$)				
100 TF	17/0.23,	3/0.20,	4/0.18,	12/0.17,	14/0.05
200 TF	12/0.19,	17/0.19,	4/0.16,	3/0.13,	14/0.08
300 TF	17/0.21,	12/0.19,	4/0.17,	3/0.13,	14/0.08
400 TF	17/0.21,	12/0.19,	4/0.17,	3/0.13,	14/0.13
500 TF	17/0.21,	12/0.20,	4/0.17,	3/0.13,	14/0.13
600 TF	17/0.24,	4/0.22,	12/0.2,	3/0.16,	14/0.13
100–500 IDF	20/0,	19/0,	18/0,	17/0,	16/0
600 IDF	12/0.08,	14/0.05,	2/0.03,	15/0.03,	17/0.02
100 TF*IDF	**17/0.42**,	**12/0.22**,	**4/0.20**,	**3/0.09**,	**1/0.07**
200 TF*IDF	17/0.26,	12/0.14,	4/0.14,	1/0.06,	2/0.05
300 TF*IDF	17/0.16,	4/0.13,	12/0.12,	1/0.06,	2/0.05
400 TF*IDF	17/0.16,	4/0.13,	12/0.08,	1/0.06,	2/0.04
500 TF*IDF	17/0.17,	12/0.13,	4/0.12,	1/0.06,	3/0.05
600 TF*IDF	17/0.19,	12/0.19,	4/0.18,	3/0.14,	14/0.10
671 ALL	17/0.24,	4/0.22,	12/0.20,	3/0.16,	14/0.13

Science document ($\vec{d}_6$). Table 1.8 summarizes the results. The results from TF and TF∗IDF sampling generally look more stable with increasing sample size and tend toward the results obtained from the complete set of features. This comes as no surprise, because with many frequent terms the document vectors are well populated with nonzero coordinates, so that adding new features does not change similarity values very much. The situation with IDF sampling is different. With 100 to 500 features, all vectors are orthogonal (the dot product with $\vec{d}_6$ is 0). With 600 features the vectors are somewhat more populated, but many are still orthogonal. They are also very sparsely populated; for example, the 600-dimensional IDF query vector $\vec{d}_6$ has only 43 nonzero coordinates.

Another interesting observation is based on the similarity values. Because all vectors are normalized to unit length, the dot product values can be compared directly, even for vectors with different number of coordinates. Thus, the similarity values may be used as an objective measure of the *quality of feature selection*. Not surprisingly, the highest values are achieved with the 100 TF*IDF sample (shown in boldface). This is an additional argument that a good balance between TF and IDF measures could bring the best results.

Jaccard Similarity

There is an alternative to cosine similarity, which appears to be more popular in the context of similarity search (we discuss the reason for this later). It takes all terms that occur in the documents but uses the simpler Boolean document representation. The idea is to consider only the nonzero coordinates (i.e., those that are 1) of the Boolean vectors. The approach uses the *Jaccard coefficient*, which is generally defined (not only for Boolean vectors) as the percentage of nonzero coordinates that are different in

the two vectors. In our particular case the similarity between two Boolean document vectors $\text{sim}(\vec{d}_1, \vec{d}_2)$ is defined as the proportion of coordinates that are 1 in both $\vec{d}_1$ and $\vec{d}_2$ to those that are 1 in $\vec{d}_1$ or $\vec{d}_2$. Thus, formally,

$$\text{sim}(\vec{d}_1, \vec{d}_2) = \frac{\left|\{j | d_1^j = 1 \wedge d_2^j = 1\}\right|}{\left|\{j | d_1^j = 1 \vee d_2^j = 1\}\right|}$$

As each 1 in the document vector represents a term that occurs in the document, this formula can be rewritten using sets of terms. Thus, we arrive at an alternative formulation of the Jaccard coefficient defined on *sets*. Let us denote the set of terms that occur in document d as $T(d)$. Then the similarity between two documents $\text{sim}(d_1, d_2)$ is defined as

$$\text{sim}(d_1, d_2) = \frac{|T(d_1) \cap T(d_2)|}{|T(d_1) \cup T(d_2)|}$$

$\text{sim}(d_1, d_2)$ has some nice properties that are important in the context of a similarity search. For example, the similarity reaches its maximum (1) if the two documents are identical [i.e., $\text{sim}(d, d) = 1$] and is symmetrical [i.e., $\text{sim}(d_1, d_2) = \text{sim}(d_2, d_1)$]. However, it is not a formal metric (distance function), as it does not satisfy the triangle equality. Note, however, that $1 - \text{sim}(d_1, d_2)$ is a metric called the *Jaccard metric*.

Direct computation of the Jaccard coefficient is straightforward, but with large documents and collections it may lead to a very inefficient similarity search. Also, finding similar documents at query time is impractical because it may take quite a long time. Therefore, some optimization techniques are used and most of the similarity computation is done offline (i.e., for each document from the collection, a number of nearest documents are precomputed). The inverted index provides a good deal of information that may be used for this purpose. The idea is to create a list of all document pairs sorted by the similarity of the documents in each pair. Then the k most similar documents to a given document d are those that are paired with d in the first k pairs from the list. Theoretically, the number of document pairs is $n(n-1)/2$ for n documents. However, two simple heuristics may drastically reduce the number of candidate pairs:

1. Frequent terms that occur in many documents (say, more than 50% of the collection) are eliminated because they cause even loosely related documents to look similar.
2. Only documents that share at least one term are used to form a pair.

Let us illustrate the basic steps of precomputing document similarity with our department collection. To simplify the discussion, in the first two steps we use the five-column term–document matrix shown in Table 1.2. In step 3, however, we compute the Jaccard coefficient using the complete set of terms for each document.

1. For each term, create a set of documents that includes the term. At this point we eliminate three terms (*lab*, *laboratory*, and *programming*) because their respective sets include only one document (no document pair can be created).

Thus, we end up with (for brevity, only the document indices are shown)

[(*program*, {1,2,4,6,7,8,14,15,16,17,19}), (*computer*, {3,4,6,12,14})]

(One may decide to eliminate *program*, due to its high frequency, but we leave it, because otherwise the example would be trivial.)

2. Create pairs of documents from each set in item 1, store them in a single file, and sort the file by the frequency counts of the pairs. The result of this step is a list of 78 pairs (counts follow the slash): [(4,6)/2, (4,14)/2, (6,14)/2, (1,2)/1, (1,4)/1, (1,6)/1, . . .]. Thus, counts represent the number of terms shared by the documents in the pair. At this point more candidate pairs can be eliminated by setting a threshold for the minimal number of shared terms.
3. Compute the similarity between the documents in each pair and sort the list of pairs accordingly. The beginning of the sorted list is as follows:

[(7,15)/0.208, (1,17)/0.196, (15,19)/0.192, (8,17)/0.189, (3,12)/0.186, (17,19)/0.185, (12,15)/0.185, (12,17)/0.18, (1,19)/0.178, (4,14)/0.176, (**6**,**12**)/**0**.**175**, (3,4)/0.173, (12,19)/0.170, (4,19)/0.168, (7,17)/0.159, (8,19)/0.158, (8,7)/0.156, (8,12)/0.153, (1,14)/0.145, (1,15)/0.142, (7,12)/0.141, (1,12)/0.140, (4,7)/0.137, (15,17)/0.136, (**4**,**6**)/**0**.**136**, (4,12)/0.136, (7,14)/0.136, (**6**,**14**)/**0**.**135**, (12,14)/0.135, . . .]

Having done this computation, we are now able to answer similarity search queries very quickly. For example, to find the documents most similar to document 6 (computer science), we go through the list from left to right and report (in the order of occurrence) the other document in each pair that contains 6. Thus, we get 12 (Mathematics), 4 (Chemistry), and 14 (Music) for the part of the list that is shown above (the corresponding pairs are shown in boldface). The complete list of documents most similar to document 6 is [12,4,14,17,3,15,19,7,2,1,16,8].

It is interesting to compare these results with the TFIDF similarity results shown in Table 1.7. The closest match is with the list produced with all features, where the five most similar documents are the same but are ranked differently ([17,4,12,3,14]). Which of the two rankings is more trustworthy? The ranking produced by the cosine similarity may look a bit strange, because it picks the Political Science document (17), whereas generally, Computer Science as a subject may be considered closer to Mathematics (12). Obviously, the documents in both pairs, (6,17) and (6,12), share a lot of terms, but in TFIDF ranking not only is the term overlap taken into account but the TF and IDF measures as well. They bring more information into the similarity ranking process, which allows more accurate computation of similarity to be done, For example, both the Computer Science and Political Science documents have five occurrences of the term *science*, while Computer Science and Mathematics have one occurrence of the term *sciences*. Also, the term *science* has a relatively high IDF score. All this is taken into account by the cosine similarity measure but is simply ignored by the Jaccard measure.

There have been studies that compare the two measures for various tasks and in various domains. In many areas the two measures show comparable results (see,

e.g., [6]). It seems, however, that for the purpose of document similarity search, the Jaccard measure is preferable. The reason for this is primarily scalability, which is an issue in large document collections such as the Web. There exist methods for approximate computation of the Jaccard coefficient that work quite well in these cases. Broder [7] proposed a method for estimating the resemblance between two documents using a set representation of document subsequences called shingles (see the next section). In fact, his method estimates the Jaccard coefficient on two sets by representing them as smaller sets called *sketches*, which are then used instead of the original documents to compute the Jaccard coefficient. Sketches are created by choosing a random permutation, which is used to generate a sample for each document. Most important, sketches have a fixed size for all documents. In a large document collection each document can be represented by its sketch, thus substantially reducing the storage requirements as well as the running time for precomputing similarity between document pairs. The method was evaluated by large-scale experiments with clustering of all documents on the Web [8]. Used originally in a clustering framework, the method also suits very well the similarity search setting.

Document Resemblance

So far we have discussed two approaches to document modeling: the TFIDF vector and set representations. Both approaches try to capture document semantics using the terms that documents contain as descriptive features and ignoring any information related to term positions, ordering, or structure. The only relevant information used for this purpose is whether or not a particular term occurs in the documents (the *set-of-words approach*) and the frequency of its occurrence (the *bag-of-words approach*). For example, the documents "Mary loves John" and "John loves Mary" are identical, because they include the same words with the same counts, although they have different meanings. The idea behind this representation is that *content* is identified with *topic* or *area* but not with *meaning* (that is why these approaches are also called *syntactic*). Thus, we can say that the topic of both documents is people and love, which is the meaning of the terms that occur in the documents.

Assume, however, that the task is to find identical or nearly identical documents, or documents that share phrases, sentences, or paragraphs. Obviously, set-based representation is not appropriate for such tasks. In a similarity search a query may return many copies of the same document (sometimes with slight modifications) that are stored at different web locations (mirror sites). Such pages may also be fetched multiple times by the web crawler if it keeps track only of the URLs of pages that have been visited. To avoid such situations, some mechanism for detecting duplicates or near duplicates of documents is needed. Detecting shared sentences, paragraphs, or other structures of text is a useful technique for identifying cases of plagiarism or studying stylistic properties of texts. Some figures obtained in the clustering study of the Web that we mentioned earlier [8] illustrate the magnitude of the problem. Among the 30 million web pages that were analyzed, there were 2.1 million clusters containing only identical documents (5.3 million documents).

There is a technique that extends the set-of-words approach to sequences of words. The idea is to consider the document as a sequence of words (terms) and

extract from this sequence short subsequences of fixed length called *n-grams* or *shingles*. The document is then represented as a set of such n-grams. For example, the document "Mary loves John" can be represented by the set of 2-grams {[Mary, loves], [loves, John]} and "John loves Mary" by {[John, loves], [loves, Mary]}. Now these four 2-grams are the features that represent our documents. In this representation the documents do not have any overlap. We have already mentioned n-grams as a technique for approximate string matching but they are also popular in many other areas where the task is detecting subsequences such as spelling correction, speech recognition, and character recognition.

Shingled document representation can be used for estimating document *resemblance*. Let us denote the set of shingles of size w contained in document d as $S(d,w)$. That is, the set $S(d,w)$ contains all w-grams obtained from document d. Note that $T(d) = S(d,1)$, because terms are in fact 1-grams. Also, $S(d,|d|) = d$ (i.e., the document itself is a w-gram, where w is equal to the size of the document). The *resemblance* between documents d_1 and d_2 is defined by the Jaccard coefficient computed with shingled documents:

$$r_w(d_1,d_2) = \frac{|S(d_1,w) \cap S(d_2,w)|}{|S(d_1,w) \cup S(d_2,w)|}$$

The same technique for precomputing document similarity can be used with the shingled document representation. The advantage here is that after obtaining document pairs along with those that are too dissimilar, we can also eliminate those that are too similar in terms of resemblance [with large values of $r_w(d_1,d_2)$]. In this way, duplicates or near duplicates can be eliminated from the similarity search results.

Although the number of shingles needed to represent each document is roughly the same as the number of terms needed for this purpose, the storage requirements for shingled document representation increase substantially. A straightforward representation of w-word shingles as integers with a fixed number of bits results in a w-fold increase in storage. For example, if the term IDs are represented by 32-bit numbers, a four-word shingle will take 128 bits. There are, however, hashing (or fingerprinting) techniques that can be used to reduce the storage requirements. Each shingle may be hashed into a number with a fixed number of bits using a fingerprinting function (see [7]). Then instead of the complete set of shingles $S(d,w)$ for each document, only shingles with 0 modulo p (some suitable prime number) are kept. Let $L(d)$ be the set of shingles that are $S(d,w)$ that are 0 modulo p. Then the estimated value of the resemblance between documents d_1 and d_2 is

$$r_e(d_1,d_2) = \frac{|L(d) \cap L(d_2)|}{|L(d_1) \cup L(d_2)|}$$

$L(d)$ is a smaller set of shingles called a *sketch* of document d. By choosing a proper value for p, the storage for $L(d)$, and consequently the storage needed for precomputing resemblance for pairs of documents, can be reduced. Of course, this comes at the expense of less accurate estimation of resemblance.

REFERENCES

1. Tim Berners-Lee, *Information Management: A Proposal*, CERN, Geneva, Switzerland, 1989–1990, http://www.w3.org/History/1989/proposal.html.
2. Tim Mayer, Our blog is growing up—and so has our index, *Yahoo! Search Blog*, Aug. 2005, http://www.ysearchblog.com/archives/000172.html.
3. C. Buckley, Implementation of the SMART information retrieval system, Technical Report 85-686, Cornell University, Ithaca, NY, 1985.
4. Oliver A. McBryan, GENVL and WWW: tools for taming the Web, presented at the First International Conference on the World Wide Web, CERN, Geneva, Switzerland, May 25–27, 1994, http://www.cs.colorado.edu/home/mcbryan/mypapers/www94.ps.
5. Sergey Brin and Lawrence Page, The anatomy of a large-scale hypertextual Web search engine, in *Proceedings of the 7th World Wide Web Conference* (WWW7), 1998, http://www7.scu.edu.au/1921/com1921.htm.
6. L. Lee, Measures of distributional similarity. *Proc. ACL*, 1999.
7. A. Broder, On the resemblance and containment of documents, in *Proceedings on Compression and Complexity of Sequences* (SEQUENCES'97), pp. 21–29, IEEE Computer Society, Los Alamitos, CA, 1998.
8. A. Broder, S. Glassman, M. Manasse, and G. Zweig, Syntactic clustering of the Web, in *Proceedings of the 6th International World Wide Web Conference*, Apr. 1997, pp. 393–404.

EXERCISES

1. Use the WebSPHINX crawler (http://www.cs.cmu.edu/~rcm/websphinx/, also available from the book series Web site www.dataminingconsultant.com), to collect the department web pages listed in the department directory page (Figure 1.3). Use the following parameters:

- *Starting URL:* http://www.artsci.ccsu.edu/Departments.htm
- *Crawl:* the Web (or the server)
- Depth: 1 hop

a. Save the pages as separate files in a directory (action: save, on pages: text). The crawler creates a directory tree automatically and saves the web pages as HTML documents. See how the directory structure matches the URL structure of the corresponding pages.

b. Convert the web documents into text documents. For example, use the "Save As . . . " option of the Internet Explorer with "Save as type: Text File (*.txt)."

c. Save all documents in a single file (action: concatenate, on pages: text). Convert it to text format [as done in part (b)] and examine its content.

2. Download and install the Weka data mining system (http://www.cs.waikato.ac.nz/~ml/weka/). Read the documentation and try some examples to familiarize yourself with its use (e.g., the weather data provided with the installation).

3. Create a data file in ARFF format (see a description of the format at http://www.cs.waikato.ac.nz/~ml/weka/). Follow the steps below.

a. Use the concatenation of the web documents (Exercise 1c) and create a text file where each document is represented on a separate line in plain text format. For example, this can be done by loading the concatenation in MS Word and then saving the file in plain text format without line breaks.

b. Enclose the document content in quotation marks (") and add the document name at the beginning of each line and a file header at the beginning of the file:

```
@relation departments_string

@attribute document_name string
@attribute document_content string

@data
Anthropology, " Anthropology consists of four ...
...
```

This representation uses two attributes: `document_name` and `document_content`, both of type string. An example of such a data file is "Departments-string.arff," available from the book series Web site, `www.dataminingconsultant.com`. Note that the representation in "Departments-string.arff" uses an additional class attribute (see Chapter 5), which is defined in the file header, and its values are added at the end of each line in the data section (after @data).

c. Load the file in the Weka system using the "Open file" button in "Preprocess" mode. After successful loading the system shows some statistics about the number of attributes, their type, and the number of instances (rows in the data section or documents).

d. Choose the StringToNominal filter and apply it to the first attribute, document_name. Then choose the StringToWordVector filter and apply it with "outputWordCounts = true" (you may also change the setting of "onlyAlphabeticTokens" and "useStoplist" to see how the results change).

e. Now you have a document–term matrix loaded in Weka. Use the "Edit" option to see it in a tabular format, where you can also change its content or copy it to other applications (e.g., MS Excel). Once created in Weka the table can be stored in an ARFF file through the "Save" option. Figure E1.3e shows a screenshot of a part of the document–term table.

f. Weka can also show some interesting statistics about the terms. In the visualization area (preprocess mode), change the class to document_name. Then you will see the distribution of terms over documents as bar diagrams. The screenshot in Figure E1.3f shows some of these diagrams.

g. Examine the diagrams (the color indicates the document) and find the most specific terms for each document. For example, compare the diagrams of *anthropology* and *chair* and explain the difference. Which one is more representative, and for which document?

4. Similar to Exercise 3, create the Boolean and TFIDF representation of the document collection. Examples of these representations are provided in the files "Departments-binary.arff" and "Departments-TFIDF.arff," available from the book Web site, `www.dataminingconsultant.com`.

a. To obtain the Boolean representation, apply the NumericToBinary filter to the word-count representation. What changed in the diagrams?

b. For the TFIDF representation, use the original string representation and apply the StringToWordVector filter with IDFTransform = true. Examine the document–term table and the diagrams. Explain why some columns (e.g., *chair* and *website*) are all zero. See these columns in the book versions of the same document collection:

Viewer

Relation: departments_string-weka.filters.unsupervised.attribute.Remove-R3-weka.filters.unsupervised.attribute.Stri...

No.	document_name Nominal	anthropology Numeric	applied Numeric	archaeology Numeric	attend Numeric	ba Numeric	background Numeric	behavioral Numeric
1	Anthropology	9.0	1.0	2.0	1.0	1.0	1.0	1.0
2	Art	0.0	0.0	0.0	0.0	1.0	0.0	0.0
3	Biology	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	Chemistry	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	Communication	0.0	0.0	0.0	0.0	1.0	0.0	0.0
6	Computer	0.0	0.0	0.0	0.0	0.0	0.0	0.0
7	Justice	0.0	0.0	0.0	0.0	1.0	0.0	0.0
8	Economics	0.0	0.0	0.0	0.0	1.0	0.0	0.0
9	English	0.0	1.0	0.0	0.0	1.0	0.0	0.0
10	Geography	0.0	0.0	0.0	0.0	1.0	0.0	0.0
11	History	0.0	0.0	0.0	0.0	1.0	0.0	0.0
12	Math	0.0	0.0	0.0	0.0	2.0	0.0	0.0
13	Languages	0.0	0.0	0.0	0.0	2.0	0.0	0.0
14	Music	0.0	0.0	0.0	0.0	2.0	0.0	0.0
15	Philosophy	0.0	1.0	0.0	0.0	1.0	0.0	0.0
16	Physics	0.0	1.0	0.0	0.0	0.0	0.0	0.0
17	Political	0.0	1.0	0.0	0.0	2.0	0.0	0.0
18	Psychology	0.0	0.0	0.0	0.0	2.0	0.0	0.0
19	Sociology	0.0	0.0	0.0	0.0	1.0	0.0	0.0
20	Theatre	0.0	0.0	0.0	0.0	2.0	0.0	0.0

Undo OK Cancel

Figure E1.3e

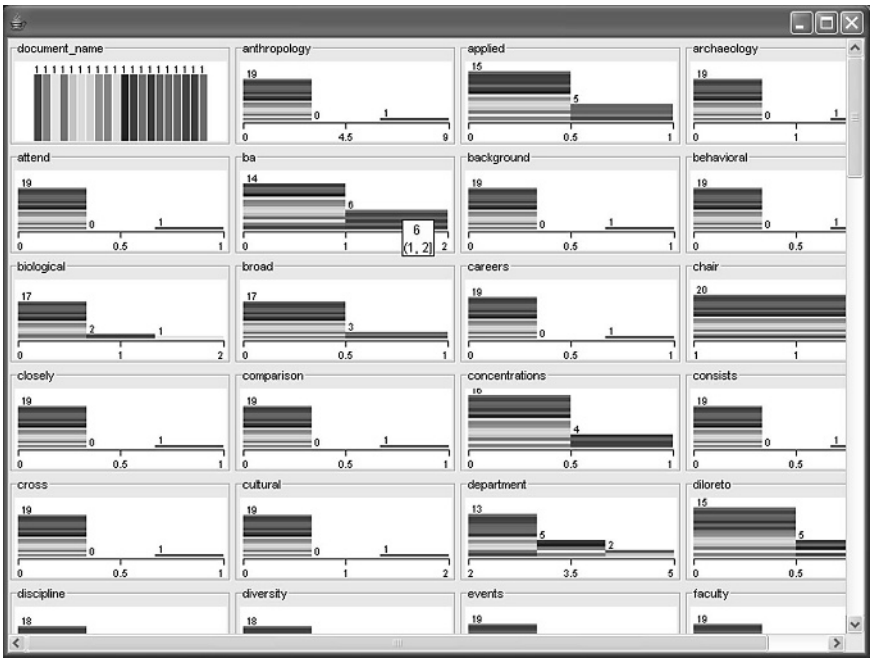

Figure E1.3f

"Departments-book-binary.arff" and "Departments-book-TFIDF.arff" (also available from the book Web site). Why are the Weka and the book versions slightly different? (See what is behind the "More" button of the StringToWordVector parameter setting window.)

5. Collect web documents from other domains (use the WebSPHINX crawler or web search) and follow the preceding steps to create ARFF data files for the term-count, Boolean, and TFIDF representations. Then load the files into Weka and analyze the document collections by examining the document–term table or the term distribution diagrams.

6. Find proper sets of keywords and evaluate the precision and recall provided by Google when searching documents in the CCSU A&S collection.

a. Use the keywords *computer* and *program* with advanced search limited within the server domain. The query will be

```
computer program site:www.artsci.ccsu.edu
```

b. Using the query results, create a table similar to Table 1.6. Then compute the interpolated precision for different recall levels and create charts similar to those in Figure 1.5.

c. Compare the charts based on the results from Google search with those based on cosine similarity (Figure 1.5).

d. Use fewer or more terms and create the corresponding charts. Try terms with different IDF scores (use Figure 1.6). Compute the average precision. Analyze the results.

e. Use terms that occur in all 20 documents (e.g., *department*, *phone*, *chair*). Explain why these documents are not always among the top 20 in the result list. Which documents occur in the top?

f. Search for specific web pages (e.g., department Web sites) in wider domains (e.g., `www.ccsu.edu`) or in the entire Web. Use a different number of keywords and compute the precision and recall. Analyze the results.