An Introduction to Information Retrieval

Draft of February 17, 2007

An Introduction to Information Retrieval

Christopher D. Manning
Prabhakar Raghavan
Hinrich Schütze

Cambridge University Press
Cambridge, England

Printed on February 17, 2007

Website: http://www.informationretrieval.org/

By Christopher D. Manning, Prabhakar Raghavan & Hinrich Schütze

Comments, corrections, and other feedback most welcome at:

informationretrieval@yahoogroups.com

Brief Contents

Contents

List of Tables

List of Figures

Acknowledgments 1

Dinquan Chen, Elmer Garduno, Sergio Govoni, Corinna Habets, Ralf Jankowitsch, Vinay Kakade, Mei Kobayashi, Wessel Kraaij, Paul McNamee Scot Olsson, Klaus Rothenhäusler, Helmut Schmid, Jason Utt, Mike Walsh, Changliang Wang, Thomas Zeume,

Acknowledgments 2

Omar Alonso, Vo Ngoc Anh, Eric Brown, Stefan Büttcher, Doug Cutting, Johannes Fürnkranz, Gonzalo Navarro, Paul Ogilvie, Jan Pedersen, Sabine Schulte im Walde, John Tait,

Table of Notations

γ	γ code
γ	Classification or clustering function: $\gamma(d)$ is d's class or cluster
γ	Weight of negative documents in Rocchio relevance feedback
γ_k	Prior in EM clustering for cluster k
ω_k	Cluster ω_k in classification
$\arg\max_x f(x)$	The value of x for which f reaches its maximum
$\arg\min_x f(x)$	The value of x for which f reaches its minimum
c, c_j	Class or category in classification
$c(w)$	Count of word w (Chapter 12)
C	Set $\{c_1, \ldots, c_j, \ldots, c_J\}$ of all classes
d	Document
$\lvert d \rvert$	Number of tokens in d (alternative to L_d)
$\vec{d}$	Document vector
$\lvert \vec{d} \rvert$	Length (or Euclidean norm) of $\vec{d}$
D	Set $\{d_1, \ldots, d_n, \ldots, d_N\}$ of all documents
f_i	Frequency of term with rank i
H	Entropy; harmonic number
J	Number of classes
k	Top k retrieved documents; top k selected features from the vocabulary V
k	Number of nearest neighbors in kNN
K	Number of clusters

L, L_d, L_t	Average length of a document, training document, and test document (in tokens)
M	Size of vocabulary ($\|V\|$)
N	Number of documents in the retrieval collection
N_j	Number of documents in class c_j
$N(\omega)$	Number of times the event ω occurred
n	Number of attributes in the representation of d: $< x_1, x_2, \ldots, x_n >$
n	Number of postings
S	Positions in the test document that contain words from the vocabulary V
T	Total number of tokens in the document collection
T_j	Number of tokens in documents in class c_j
T_{ji}	Number of occurrences of word i in class c_j
t	Structural term (word + context in XML retrieval)
$\vec{u}_j$	Rocchio centroid of class c_j
V	Vocabulary $\{w_1, \ldots, w_i, \ldots, w_M\}$ of terms indexed
$\vec{V}(d)$	Unnormalized document vector
$\vec{v}(d)$	Normalized document vector
$\vec{w}^{\mathrm{T}}\vec{x} = b$	Hyperplane
$\vec{x} = (x_1, \ldots, x_i, \ldots, x_M)$	Term incidence vector; more generally: document feature representation
X_i	Random variable for attribute i
$\mathbb{X}$	Instance space in text classification
x_i	Sequence $\{x_1, \ldots, x_i, \ldots, x_{L_t}\}$ of tokens in the test document
$\|\vec{x} - \vec{y}\|$	Euclidean distance of $\vec{x}$ and $\vec{y}$

1 *Information retrieval using the Boolean model*

INFORMATION RETRIEVAL

The meaning of the term *information retrieval* can be very broad. Just getting a credit card out of your wallet so that you can type in the card number is a form of information retrieval. However, as an academic field of study, *information retrieval* might be defined thus:

> Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfy an information need from within large collections (usually on local computer servers or on the internet).

As defined in this way, information retrieval used to be an activity that only a few people engaged in: reference librarians, paralegals, and similar professional searchers. However, the world has changed, and now hundreds of millions of people engage in information retrieval every day when they use a web search engine or search their email. Information retrieval is fast becoming the dominant form of information access, overtaking traditional database-style searching (the sort that is going on when a clerk says to you: "I'm sorry, I can only look up your order if you can give me your Order ID").

IR can also cover other kinds of data and information problems beyond what we have specified in the core definition above. The term "unstructured data" refers to data which does not have clear, semantically overt, easy-for-a-computer structure. It is the opposite of structured data, the canonical example of which is a conventional database, such as those companies usually use to maintain product inventories and personnel records. However, almost no data is truly "unstructured". This is definitely true of all text data if one considers the latent linguistic structure of human languages. But even accepting that the intended notion of structure is overt structure, most text has structure, such as headings and paragraphs and footnotes, which is commonly represented in the documents by overt markup (such as the coding underlying web pages). We wish to also facilitate "semi-structured" search such as finding a document where the Title contains Java and the Body contains threading.

The field of information retrieval also covers operations typically done in browsing document collections or further processing a set of retrieved documents. Given a set of documents, clustering is the task of coming up with a good grouping of the documents based on their contents. It is similar to arranging books on a bookshelf according to their topic. Given a set of topics (or standing information needs), classification is the task of deciding which topic(s), if any, each of a set of documents belongs to. It is often approached by first manually classifying some documents and then hoping to be able to classify new documents automatically.

In this chapter we will begin with an information retrieval problem, and introduce the idea of a term-document matrix (Section 1.1) and the central inverted index data structure (Section 1.2). We will then examine the Boolean retrieval model and how queries are processed (Section 1.3).

1.1 An example information retrieval problem

A fat book which many people own is Shakespeare's Collected Works. Suppose you wanted to sort out which plays of Shakespeare contain the words Brutus AND Caesar AND NOT Calpurnia. One way to do that is to start at the beginning and to read through all the text, noting for each play whether it contains Brutus and Caesar and excluding it from consideration if it contains Calpurnia. Computers can do such things faster than people. The simplest form of document retrieval is to do this sort of linear scan through documents. This process is commonly referred to as *grepping* through text, after the Unix command `grep`, which performs this process. Grepping through text can be a very effective process, especially given the speed of modern computers, and often allows useful possibilities for wildcard pattern matching through the use of *regular expressions*. With modern computers, for simple querying of modest collections (the size of Shakespeare's Collected Works is a bit under one million words of text in total), you really need nothing more.

GREPPING

REGULAR EXPRESSIONS

But for many purposes, you do need more:

1. To process large document collections quickly. The amount of online data has grown at least as quickly as the speed of computers, and we would now like to be able to search collections that total in the order of billions to trillions of words.

2. To allow more flexible matching operations. For example, to find the word Romans NEAR countrymen, where NEAR is perhaps defined as within 5 words is not feasible with `grep`.

3. To allow ranked retrieval: in many cases you want the best answer to an information need among many documents that contain certain words.

	Anthony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth	...
Anthony	1	1	0	0	0	1	
Brutus	1	1	0	1	0	0	
Caesar	1	1	0	1	1	1	
Calpurnia	0	1	0	0	0	0	
Cleopatra	1	0	0	0	0	0	
mercy	1	0	1	1	1	1	
worser	1	0	1	1	1	0	
...							

► **Figure 1.1** A term-document incidence matrix. Matrix element (i, j) is 1 if the play in column j contains the word in row i, and is 0 otherwise.

INDEX

The way to avoid linearly scanning the texts for each query is to *index* the documents in advance. Let us stick with Shakespeare's Collected Works, and use it to introduce the basics of the Boolean retrieval model. Suppose we record for each document – here a play of Shakespeare's – whether it contains each word out of all the words Shakespeare used (Shakespeare used about 32,000 different words). The result is a binary term-document incidence matrix, as in Figure 1.1. Here the word *term* roughly means "word" but is normally preferred in the information retrieval literature since some of the indexed units, such as perhaps 3.2 or & are not usually thought of as words. Now, depending on whether we look at the matrix rows or columns, we can have a vector for each term, which shows the documents it appears in, or a vector for each document, showing the terms that occur in it.[1]

TERM

To answer the query Brutus AND Caesar AND NOT Calpurnia, we take the vectors for Brutus, Caesar and Calpurnia, complement the last, and then do a bitwise AND:

110100 AND 110111 AND 101111 = 100100

The answers for this query are thus *Anthony and Cleopatra* and *Hamlet* (Figure 1.1).

BOOLEAN RETRIEVAL MODEL

The *Boolean retrieval model* refers to being able to ask any query which is in the form of a Boolean expression of terms. That is, terms can be combined with the operators AND, OR, and NOT. Such queries effectively view each document as a set of words.

Let us now consider a more realistic scenario, and simultaneously take the chance to introduce some key terminology and notation. Suppose we have

1. Formally, we take the transpose of the matrix to be able to get the terms as column vectors.

Antony and Cleopatra, Act III, Scene ii
Agrippa [Aside to Domitius Enobarbus]: Why, Enobarbus,
When Antony found Julius Caesar dead,
He cried almost to roaring; and he wept
When at Philippi he found Brutus slain.

Hamlet, Act III, Scene ii
Lord Polonius: I did enact Julius Caesar: I was killed i' the
Capitol; Brutus killed me.

▶ **Figure 1.2** Results from Shakespeare for the query Brutus AND Caesar AND NOT Calpurnia.

DOCUMENT

$N = 1$ million documents. By a *document* we mean whatever units we have decided to build a retrieval system over. They might be individual memos, or chapters of a book (see Section 2.1.2 (page 18) for further discussion). We will refer to the group of documents over which we perform retrieval as the (document) *collection*. It is sometimes also refered to as a *corpus* (a *body* of texts). Suppose each document is about 1000 words long (2–3 book pages). If we assume an average of 6 bytes per word including spaces and punctuation, then this is a document collection about 6 GB in size. Typically, there might be about $M = 500{,}000$ distinct terms in these documents. There is nothing special about the numbers we have chosen, and they might vary by an order of magnitude or more, but they give us some idea of the dimensions of the kinds of problems we need to handle. We will discuss and model these size assumptions in Section 5.1 (page 65). If our goal is to satisfy arbitrary information needs posed as queries over this document collection, then this is referred to as the *ad-hoc retrieval* task.

COLLECTION
CORPUS

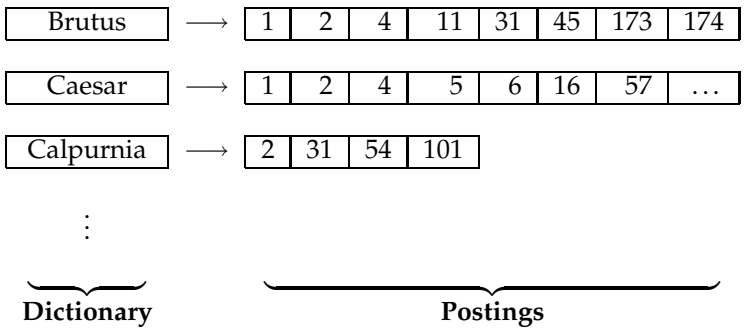
AD-HOC RETRIEVAL

We now cannot build a term-document matrix in a naive way. A 500K $\times$ 1M matrix has half-a-trillion 0's and 1's. But the crucial observation is that the matrix is extremely sparse, that is, it has few non-zero entries. Because each document is 1000 words long, the matrix has no more than one billion 1's, so a minimum of 99.8% of the cells are zero. A much better representation is to record only the things that do occur, that is, the 1 positions.

This idea is central to the first major concept in information retrieval, the *inverted index*. The name is actually redundant and silly: an index always maps back from terms to the parts of a document where they occur. Nevertheless, *inverted index*, or sometimes *inverted file*, has become the standard term in information retrieval.[2] The basic idea of an inverted index is shown

INVERTED INDEX

2. Some information retrieval researchers prefer the term inverted file, but index construction and index compression are much more common than inverted file construction and inverted file compression. For consistency, we use (inverted) index throughout this book.



► **Figure 1.3** The two parts of an inverted index. The dictionary is usually kept in memory, with pointers to each postings list, which is stored on disk.

DICTIONARY
VOCABULARY
LEXICON
POSTINGS LIST
POSTING
POSTINGS

in Figure 1.3. We keep a *dictionary* of terms (sometimes also referred to as a *vocabulary* or *lexicon*), and then for each term, we have a list that records which documents the term occurs in. This list is conventionally called a *postings list*, and each item in it which is a term combined with a document ID (and, later, often a position in the document) is called a *posting*. Taken together, all the postings lists are referred to as the *postings*. The dictionary in Figure 1.3 has been sorted alphabetically and the postings list is sorted by document ID. We will see why this is useful in Section 1.3, below, but will later we also consider alternatives to doing this (Section 7.2.1).

1.2 A first take at building an inverted index

To be able to gain the speed benefits of indexing at retrieval time, we have to have built the index in advance. The major steps in this are:

1. Collect the documents to be indexed.
   Friends, Romans, countrymen. | So let it be with Caesar | ...
2. Tokenize the text , turning each document is a list of tokens: Friends | Romans | countrymen | So | ...
3. Linguistic preprocessing. The result is that each document is a list of normalized tokens: friend | roman | countryman | so | ...
4. Index the documents that each token occurs in by creating an inverted index, consisting of a dictionary and postings.

We will discuss the earlier stages of processing, that is, steps 1–3, in Chapter 2, and until then you can think of *tokens*, *normalized tokens*, and *terms* loosely as just *words*. Here, we assume that the first 3 steps have already been done, and we examine building a basic inverted index.

DOCID

Within a document collection, we assume that each document has a unique serial number, known as the document identifier (*docID*). During index construction, we can simply assign successive integers to each new document when it is first encountered. The input to indexing will be a list of normalized tokens for each document, which we can equally think of as a list of pairs of term and docID, as in Figure 1.4. The core indexing step is *sorting* this list so that the terms are alphabetical, giving us the representation in the middle column of Figure 1.4. Multiple occurrences of the same term from the same document are then merged, and an extra column is added to record the frequency of the term in the document, giving the righthand column of Figure 1.4.[3] Frequency information is unnecessary for the basic Boolean search engine with which we begin our discussion, but as we will see soon, it is useful or needed in many extended searching paradigms, and it also allows us to improve efficiency at query time, even with a basic Boolean retrieval model.

SORTING

DICTIONARY
POSTINGS

Instances of the same term are then grouped, and the result is split into a *dictionary* and *postings*, as shown in Figure 1.5. The postings are secondarily sorted by docID. Since a term generally occurs in a number of documents, this organization already reduces the storage requirements of the index and, moreover, it provides the basis for efficient retrieval. Indeed, this inverted index structure is essentially without rivals as the most efficient structure for supporting ad hoc text search.

In the resulting index, we pay for storage of both the dictionary and the postings lists. The latter are much larger, but the dictionary is normally kept in memory, while postings lists are normally kept on disk, so the size of each is important, and in Chapter 5 we will examine how each can be optimized for storage and access efficiency. What data structure should be used for a postings list? A fixed length array would be silly as some words occur in many documents, and others in very few. For an in-memory postings list, singly linked lists are generally preferred, since they allow easy dynamic space allocation, support advanced indexing strategies, and make insertion of documents into the postings list easy (following updates, such as when recrawling the web for updated documents). However, they require a space overhead for pointers. If updates are infrequent, variable length arrays may work very well, or we could use a hybrid scheme with a linked list of fixed length arrays for each term. When postings lists are stored on disk, they are

3. Unix users can note that these steps are equivalent to use of the `sort` and then `uniq -c` commands.

Doc 1
I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:

term	**docID**
I	1
did	1
enact	1
julius	1
casear	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

$\Longrightarrow$

term	**docID**
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

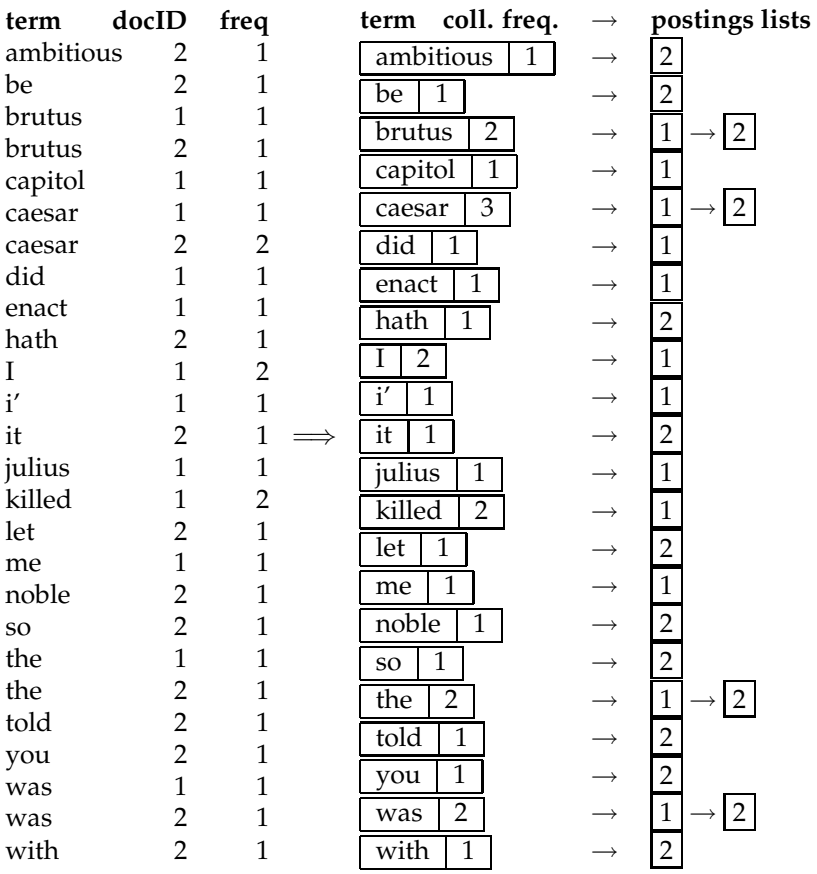
$\Longrightarrow$

term	**docID**	**term freq.**
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

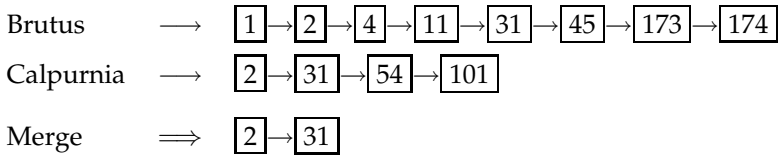
► **Figure 1.4** Indexing, part one. The initial list of terms in each document (tagged by their documentID) is sorted alphabetically, and duplicate occurrences of a term in a document are collapsed, but the term frequency is preserved. (We will use it later in weighted retrieval models.)

Doc 1
I did enact Julius Caesar: I was killed i' the Capitol; Brutus killed me.

Doc 2
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious:



► **Figure 1.5** Indexing, part two. The dictionary stores the terms, and has a pointer to the postings list for each term. It commonly also stores other summary information such as the total frequency of each term in the collection, and/or the number of documents in which each term occurs. Each postings list stores the list of documents in which a term occurs, and may store other information such as the term frequency in each document and the position(s) of the term in the document.



► **Figure 1.6** Merging the postings lists for Brutus and Calpurnia from Figure 1.3.

stored (perhaps compressed) as a contiguous run of postings without explicit pointers (as in Figure 1.3), so as to minimize the size of the postings list and the number of disk seeks to read a postings list into memory.

1.3 Processing Boolean queries

How do we process a query in the basic Boolean retrieval model? Consider processing the *simple conjunctive query*:

SIMPLE CONJUNCTIVE QUERIES

(1.1) Brutus AND Calpurnia

over the inverted index partially shown in Figure 1.3 (page 5). This is fairly straightforward:

1. Locate Brutus in the Dictionary
2. Retrieve its postings
3. Locate Calpurnia in the Dictionary
4. Retrieve its postings
5. "Merge" the two postings lists, as shown in Figure 1.6.

POSTINGS MERGE

The *merging* operation is the crucial one: we want to efficiently intersect postings lists so as to be able to quickly find documents that contain both terms.

There is a simple and effective method of doing a merge whereby we maintain pointers into both lists and walk through the two postings lists simultaneously, in time linear in the total number of postings entries, which we present in Figure 1.7. At each step, we compare the docID pointed to by both pointers. If they are the same, we put that docID in the result list, and advance both pointers. Otherwise we advance the pointer pointing to the smaller docID. If the lengths of the postings lists are x and y, this merge takes $O(x + y)$ operations. To obtain this result, it is crucial that postings are sorted by docID.

We would like to be able to extend the merge operation to process more complicated queries like:

```
MERGE(p, q)
1  answer ← ⟨ ⟩
2  while p ≠ NIL and q ≠ NIL
3  do if docID[p] = docID[q]
4        then ADD(answer, docID[p])
5        else if docID[p] < docID[q]
6                then p ← next[p]
7                else q ← next[q]
8  return answer
```

► **Figure 1.7** Algorithm for the merging (or intersection) of two postings lists.

(1.2) (Brutus OR Caesar) AND NOT Calpurnia

QUERY OPTIMIZATION *Query optimization* is the process of selecting how to organize the work of answering a query so that the least total amount of work needs to be done by the system. A major element of this for Boolean queries is the order in which postings lists are accessed. What is the best order for query processing? Consider a query that is an AND of t terms, for instance:

(1.3) Brutus AND Calpurnia AND Caesar

For each of the t terms, we need to get its postings, then AND them together. The standard heuristic is to process terms in order of increasing frequency: if we start with the smallest postings list, and then keep cutting the size of intermediate results further, then we are likely to do the least amount of total work. So, for the postings lists in Figure 1.3, we execute the above query as:

(1.4) (Calpurnia AND Brutus) AND Caesar

This is a first justification for keeping the frequency of terms in the dictionary: it allows us to make this ordering decision based on in-memory data before accessing any postings list.

Consider now the optimization of more general queries, such as:

(1.5) (madding OR crowd) AND (ignoble OR strife)

As before, we will get the frequencies for all terms, and we can then (conservatively) estimate the size of each OR by the sum of the frequencies of its disjuncts. We can then process the query in increasing order of the size of each disjunctive term.

For arbitrary Boolean queries, we have to evaluate and temporarily store the answers for intermediate terms in a complex expression. However, in many circumstances, either because of the nature of the query language, or

```
MERGE(⟨t_i⟩)
1  terms ← SORTBYFREQ(⟨t_i⟩)
2  result ← postings[first[terms]]
3  terms ← rest[terms]
4  while terms ≠ NIL and result ≠ NIL
5  do list ← postings[first[terms]]
6     MERGEINPLACE(result, list)
7  return result
```

► **Figure 1.8** Algorithm for merging (or intersection) of a set of postings lists.

just because this is the most common type of query that users submit, a query is purely conjunctive. In this case, rather than regarding merging postings lists as a symmetric operation, it is more efficient to intersect each retrieved postings list with the current intermediate result in memory, where we begin by loading the postings list of the least frequent term into memory. This algorithm is shown in Figure 1.8. The merge operation is then asymmetric: the intermediate result list is always shorter than the list it is merged with, and in many cases it might be orders of magnitude shorter. The postings merge can still be done by an adaptation of the algorithm in Figure 1.7, so that it calculates the result in place by updating the first list. When the difference between the list lengths is very large, opportunities to use alternative techniques open up. One alternative is to do the intersection by performing a sequence of binary searches in the long postings list for each term in the intermediate result list. Another possibility is to store the long postings list as a hashtable, so that membership of an intermediate result item can be calculated in constant rather than linear or log time. However, such alternative techniques are difficult to combine with postings list compression of the sort discussed in Chapter 5. Moreover, standard postings list merge operations remain necessary when both terms of a query are very common.

1.4 Boolean querying, extend Boolean querying, and ranked retrieval

RANKED RETRIEVAL MODELS

The Boolean retrieval model contrasts with *ranked retrieval models* such as the vector space model (Chapter 7), in which users largely use free-text queries rather than a precise language for building up query expressions, and the system decides which documents best satisfy the query. Despite decades of academic research on the advantages of ranked retrieval, systems implementing the Boolean retrieval model were the main or only search option

provided by large commercial information providers for three decades until approximately the arrival of the World Wide Web (in the early 1990s).

However, these systems did not have just the basic Boolean operations (AND, OR, and NOT) which we have presented so far. A strict Boolean expression over terms with an unordered results set is too limited for many of the searching needs that people have, and these systems implemented extended Boolean retrieval models by incorporating additional operators such as term proximity operators.

Example 1.1: Commercial Boolean searching: WestLaw WestLaw (http://www.westlaw.com/) is the largest commercial legal search service (in terms of the number of paying subscribers), with over half a million subscribers performing millions of searches a day over tens of terabytes of text data. The service was started in 1975. In 2005, Boolean search ("Terms and Connectors") was still the default on WestLaw, and used by a large percentage of users, although ranked free-text querying ("Natural Language") was added in 1992. Example Boolean queries are shown in Figure 1.9. Typical expert queries are carefully defined and refined to try to match suitable documents. Submitted queries average about ten words in length. Many users, particularly professionals, prefer Boolean query models. Boolean queries are precise: a document either matches the query or it does not. This offers the user greater control and transparency over what is retrieved. And some domains, such as legal materials, allow an effective means of document ranking within a Boolean model: WestLaw returns documents in reverse chronological order, which is in practice quite effective. This does not mean though that Boolean queries are more effective for professional searchers. Indeed, experimenting on a WestLaw subcollection, Turtle (1994) found that natural language queries produced better results than Boolean queries prepared by WestLaw's own reference librarians for the majority of the information needs in his experiments.

In later chapters we will consider in detail richer query languages and the sort of augmented index structures that are needed to handle them efficiently. Here we just mention a few of the main additional things we would like to be able to do:

1. It is often useful to search for compounds or phrases that denote a concept such as "operating system". And as the WestLaw example shows, we might also wish to do proximity queries such as Gates NEAR Microsoft. To answer such queries, the index has to be augmented to capture the position of terms in documents.

2. A Boolean model only records term presence or absence, but often we would like to accumulate evidence, giving more weight to documents that have a term several times as opposed to ones that contain it only once. To

Information need: What is the statute of limitations in cases involving the federal tort claims act?
Query: limit! /3 statute action /s federal /2 tort /3 claim

Information need: Requirements for disabled people to be able to access a workplace.
Query: disab! /p access! /s work-site work-place (employment /3 place)
Information need: Is a host responsible for drunk guests' behavior?
Query: host! /p (responsib! or liab!) /p (intoxicat! or drunk!) /p guest

► **Figure 1.9** Example Boolean queries on WestLaw. Note the long, precise queries and the use of proximity operators, both uncommon in web search. Unlike web search conventions, a space between words represents disjunction (the tightest binding operator), & is AND and /s, /p, and /*n* ask for matches in the same sentence, same paragraph or within *n* words respectively. The exclamation mark (!) gives a trailing wildcard query (see Chapter 3); thus liab! matches all words starting with liab. Queries are usually incrementally developed until they obtain what look to be good results to the user.

be able to do this we need term frequency information for documents in postings lists.

3. Straight Boolean queries just retrieve a set of matching documents, but commonly we wish to have an effective method to order (or "rank") the returned results. This requires having a mechanism for determining a document score which encapsulates how good a match a document is for a query.

In this chapter, we have looked at the structure and construction of a basic inverted index, comprising a dictionary and postings list. We introduced the Boolean retrieval model, and examined how to do efficient retrieval via linear time merges and simple query optimization. The following chapters provide more details on selecting the set of terms in the dictionary and building indices that support these kinds of options. These are the basic operations that support unstructured ad hoc searching for information. This search methodology has recently conquered the world, powering not only web search engines but the kind of unstructured search that lies behind modern large eCommerce websites.

1.5 References and further reading

The practical pursuit of information retrieval began in the late 1940s (Cleverdon 1991, Liddy 2005). A great increase in the production of scientific literature, much in the form of less formal technical reports rather than traditional

journal articles, coupled with the availability of computers led to interest in automatic document retrieval. However, in those days, document retrieval was always based on author, title, and keywords; full-text search came much later.

The article of Bush (1945) provided lasting inspiration for the new field:

> "Consider a future device for individual use, which is a sort of mechanized private file and library. It needs a name, and, to coin one at random, "memex" will do. A memex is a device in which an individual stores all his books, records, and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory."

The term *Information Retrieval* was coined by Calvin Mooers in 1948/1950. In 1958, much newspaper attention was paid to demonstrations at a conference (see Taube and Wooster 1958) of IBM "auto-indexing" machines, based primarily on the work of H. P. Luhn. Commercial interest quickly gravitated towards boolean retrieval systems, but the early years saw a heady debate over various disparate technologies for retrieval systems. For example Mooers (1961) dissented:

> "It is a common fallacy, underwritten at this date by the investment of several million dollars in a variety of retrieval hardware, that the algebra of George Boole (1847) is the appropriate formalism for retrieval system design. This view is as widely and uncritically accepted as it is wrong."

Witten et al. (1999) is the standard reference for an in-depth comparison of the space and time efficiency of the inverted index versus other possible data structures. We will also discuss other more recent work in Chapter 5.

1.6 Exercises

Exercise 1.1 [⋆]

For the queries below, can we still run through the merge in time $O(x + y)$, where x and y are the lengths of the postings lists for Brutus and Caesar? If not, what can we achieve?

a. Brutus AND NOT Caesar

b. Brutus OR NOT Caesar

Exercise 1.2 [⋆]

What about the time complexity for an arbitrary Boolean formula? For instance, consider:

c. (Brutus OR Caesar) AND NOT (Anthony OR Cleopatra)

Can we always merge in linear time? Linear in what? Can we do better than this?

Exercise 1.3

We can use distributive laws for AND and OR to rewrite queries. Suppose you rewrite the above query (from conjunctive normal form to disjunctive normal form) using the distributive laws. Would the result be more or less efficiently evaluated than the original form?

Exercise 1.4

Recommend a query processing order for

d. (tangerine OR trees) AND (marmalade OR skies) AND (kaleidoscope OR eyes)

given the following postings list sizes:

Term	Postings size
eyes	213312
kaleidoscope	87009
marmalade	107913
skies	271658
tangerine	46653
trees	316812

Exercise 1.5

If the query is:

e. friends AND romans AND (NOT countrymen)

how could we use the frequency of countrymen in evaluating the best query evaluation order?

Exercise 1.6

Extend the merge to an arbitrary Boolean query. Can we always guarantee execution in time linear in the total postings size? Hint: Begin with the case of a Boolean formula query: each query term appears only once in the query.

Exercise 1.7

For a conjunctive query, is processing postings lists in order of size guaranteed to be optimal? Explain why it is, or given an example where it isn't.

2 *The dictionary and postings lists*

Recall the major steps in inverted index construction:

1. Collect the documents to be indexed.
2. Tokenize the text.
3. Linguistic preprocessing.
4. Indexing.

In this chapter we will first examine some of the substantive linguistic issues of tokenization and linguistic preprocessing, which determine which terms are indexed in the dictionary. Tokenization is the process of chopping character streams into *tokens*, while linguistic preprocessing then deals with building equivalence classes of tokens which are the set of *terms* that are indexed. Indexing itself is covered in Chapters 1 and 4. In the second half of this chapter, we then start to examine extended postings list data structures that support faster querying and extended Boolean models, such as handling phrase and proximity queries.

TOKEN
TERM

2.1 Document delineation and character sequence decoding

2.1.1 Obtaining the character sequence in a document

Digital documents that are input to an indexing process are typically bytes in a file or on a web server. The first step of processing is to convert this byte sequence into a linear sequence of characters. For the case of plain English text in ASCII encoding, this is trivial. But often things get much more complex. The sequence of characters may be encoded by one of various single byte or multibyte encoding schemes, such as Unicode UTF-8, or various national or vendor-specific standards. The correct encoding has to be determined. This can be regarded as a machine learning classification problem, as discussed in

Chapter 13,[1] but is often handled by heuristic methods, user selection, or by using provided document metadata. Once the encoding is determined, decoding to a character sequence has to be performed. The choice of encoding might be saved as it gives some evidence about what language the document is written in.

Even for text-format files, additional decoding may then need to be done. For instance, XML character entities such as `&` need to be decoded to give the & character. Alternatively, the characters may have to be decoded out of some binary representation like Microsoft Word DOC files and/or a compressed format such as zip files. Again, the document format has to be determined, and then an appropriate decoder has to be used. Finally, the textual part of the document may need to be extracted out of other material that will not be processed. You might want to do this with XML files if the markup is going to be ignored; you would almost certainly want to do this with postscript or PDF files. We will not deal further with these issues here, and will assume henceforth that our documents are a list of characters. But commercial products can easily live or die by the range of document types and encodings that they support. Users want things to just work with their data as is. Often, they just see documents as text inside applications and are not even aware of how it is encoded on disk.

The idea that text is a linear sequence of characters is also called into question by some writing systems, such as Arabic, where text takes on some three dimensional and mixed order characteristics, as shown in Figures 2.1 and 2.2. But, despite some complicated writing system conventions, there is an underlying sequence of sounds being represented and hence an essentially linear structure remains, and this is what is represented in the digital representation of Arabic, as shown in Figure 2.1.

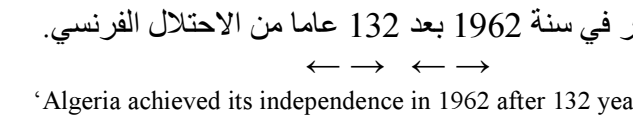
2.1.2 Choosing a document unit

DOCUMENT

The next phase is to determine what the *document* unit for indexing is. In the simplest case, each file in a document collection is a document. But there are many cases in which you might want to do something different. A traditional Unix (mbox-format) email file stores a sequence of email messages (a folder) in one file, but one might wish to regard each email message as a separate document. Many email messages now contain attached documents, and you might then want to regard the email message and each contained attachment as separate documents. If an email message has an attached zip file, one might want to decode the zip file and regard each file it contains as a

1. A classifier is a function that takes instances of some sort and assigns them to one of a number of distinct classes. Usually classification is done by probabilistic models or other machine learning methods, but it can also be done by hand-written rules.

► **Figure 2.1** Arabic script example. This is an example of a vocalized Modern Standard Arabic word. The writing is from right to left and letters undergo complex mutations as they are combined. Additionally, the representation of short vowels (here, /i/ and /u/) and the final /n/ (nunation) departs from strict linearity by being represented as diacritics above and below letters. Nevertheless, the represented text is still clearly a linear ordering of characters representing sounds. Full vocalization, as here, normally appears only in the Koran and children's books. Day-to-day text is unvocalized (short vowels are not represented but the letter for ā would still appear) or partially vocalized, with short vowels inserted in places where the writer perceives ambiguities. These choices add further complexities to indexing.



► **Figure 2.2** The conceptual linear order of characters is not necessarily the order that you see on the page. In languages that are written right-to-left, such as Hebrew and Arabic, it is quite common to also have left-to-right text interspersed, such as numbers and dollar amounts, as shown in the Arabic example above. With modern Unicode representation concepts, the order of characters in files matches the conceptual order, and the reversal of displayed characters is handled by the rendering system, but this may not be true for documents in older encodings. You need at least to ensure that indexed text and query terms are representing characters in the same order.

separate document. Sometimes people index each paragraph of a document as a separate pseudo-document, because they believe it will be more helpful for retrieval to be returning small pieces of text so that the user can find the relevant sentences of a document more easily. Going in the opposite direction, various pieces of web software take things that you might regard as a single document (a Powerpoint file or a LaTeX document) and split them into separate HTML pages for each slide or subsection, stored as separate files. In these cases, you might want to combine multiple files into a single document. For now, we will assume that some suitable size unit has been chosen, perhaps together with an appropriate way of dividing or aggregating files. See Chapter 10 for discussion of simultaneously indexing documents at multiple levels of granularity.

2.2 Determining dictionary terms

2.2.1 Tokenization

Given a character sequence and a defined document unit, tokenization is the job of chopping it up into pieces, called *tokens*, perhaps at the same time throwing away certain characters, such as punctuation. These tokens are often loosely referred to as terms or words, but it is sometimes important to make a type/token distinction. A *token* is an instance of a character sequence in some particular document. A *type* is the class of all tokens containing the same character sequence. A term is a (perhaps normalized) type that is indexed in the IR system's dictionary. Here is an example of tokenization:

TOKENS

TOKEN

TYPE

Input: Friends, Romans, Countrymen, lend me your ears;
Output: Friends | Romans | Countrymen | lend | me | your | ears

Each such token is now a candidate for an index entry, after further processing is done, which we describe later in the chapter.

The major question of this phase is what are the correct tokens to emit? For the example above, this looks fairly trivial: you chop on whitespace and throw away punctuation characters. This is a starting point, but even for English there are a good number of tricky cases. What do you do about the various uses of the *apostrophe* for possession and contractions:

APOSTROPHE

Mr. O'Neill thinks that the boys' stories about Chile's capital aren't amusing.

For *O'Neill*, which of the following is the desired tokenization?

neill
oneill
o'neill
o' | neill
o | neill ?

And for *aren't*, is it:

aren't
arent
are | n't
aren | t ?

The last looks intuitively bad. For all of them, the choices determine which Boolean queries will match. A query of neill AND capital will match in three

cases but not the other two. In how many cases would a query of o'neill AND capital match? If no preprocessing of a query is done, then it would match in only one of the five cases. For either Boolean or free-text queries, you always want to do the exact same tokenization of document and query terms. This can be done by processing queries with the same tokenizer that is used for documents.[2] This guarantees that a sequence of characters in a text will always match the same sequence typed in a query.[3]

LANGUAGE IDENTIFICATION

These issues of tokenization are clearly language-specific. It thus requires the language of the document to be known. *Language identification* based on classifiers that look at character-sequence level features is highly effective; most languages have distinctive signature patterns.

For most languages and particular domains within them there are unusual specific tokens that we wish to recognize as terms, such as the programming languages C++ and C#, aircraft names like B-52, or a T.V. show name such as M*A*S*H – which is sufficiently integrated into popular culture that one finds usages such as *M*A*S*H-style hospitals*. Computer technology has introduced new types of character sequences that a tokenizer should probably tokenize as a single token, including email addresses (jblack@mail.yahoo.com), web URLs (http://stuff.big.com/new/specials.html), numeric IP addresses (142.32.48.231), package tracking numbers (1Z9999W99845399981), and more. One possible solution is to omit from indexing tokens such as monetary amounts, numbers, and URLs, since their presence greatly expands the size of the dictionary. However, this comes at a large cost in restricting what people can search for. For instance, people might want to search in a bug database for something like a line number where an error occurs. For items such as dates, if they have a clear semantic type, such as the date of an email, then they are often indexed separately as document metadata.

HYPHENS

In English, *hyphenation* is used for various purposes ranging from splitting up vowels in words (*co-education*) to joining nouns as names (*Hewlett-Packard*) to a copyediting device to show word grouping (*the hold-him-back-and-drag-him-away maneuver*). It is easy to feel that the first example should be regarded as one token (and is indeed more commonly written as just *coeducation*), the last should be separated into words, and that the middle case is kind of unclear. Handling hyphens automatically can thus be complex: it can either be done as a classification problem, or more commonly by some heuristic rules, such as allowing short hyphenated prefixes on words, but

2. For the free-text case, this is straightforward. The Boolean case is more complex: if terms in the input are tokenized into multiple tokens, this can be represented by combining them with an AND or as a phrase query, but it is harder to correct things if the user has separated terms that were tokenized together in the document processing.

3. This tokenization may result in multiple words from one query word, which is unproblematic in the free text case, and which in the Boolean case should be regarded as terms joined by an AND or as a multiterm phrase query, which we discuss in Section 2.3.2.

not longer hyphenated forms. One effective strategy in practice is to encourage users to enter hyphens wherever they may be possible, and then for, say, *over-eager* to search for over-eager OR over eager OR overeager.

Conceptually, splitting on white space can also split what should be regarded as a single token. This occurs most commonly with names (*San Francisco, Los Angeles*) but also with borrowed foreign phrases (*au fait*) and compounds that are sometimes written as a single word and sometimes space separated (such as *white space* vs. *whitespace*). Another case with internal spaces that one might wish to regard as a single token includes phone numbers ((800) 234-2333) and dates (Mar 11, 1983). The problems of hyphens and non-separating whitespace can even interact. Advertisements for air fares frequently contain items like *San Francisco-Los Angeles*, where simply doing whitespace splitting would give unfortunate results. In such cases issues of tokenization interact with handling phrase queries (which we discuss later in the chapter), particularly if we would like queries for all of *lowercase*, *lower-case* and *lower case* to return the same results. The last two can be handled by splitting on hyphens and using a phrase index, as discussed later in this chapter. Also getting the first case right would depend on knowing that it is sometimes written as two words and also indexing it in this way. For some Boolean retrieval systems, such as WestLaw and Lexis-Nexis, if you search using the hyphenated form, then it will generalize the query to cover all three of the one word, hyphenated, and two word forms, but if you query using either of the other two forms, you get no generalization.

Each new language presents some new issues. For instance, French has a variant use of the apostrophe for a reduced definite article 'the' before a word beginning with a vowel (e.g., *l'ensemble*) and has some uses of the hyphen with postposed clitic pronouns in imperatives and questions (e.g., *donne-moi* 'give me'). Getting the first case correct will affect the correct indexing of a fair percentage of nouns and adjectives: you would want documents mentioning both *l'ensemble* and *un ensemble* to be indexed under *ensemble*. Other languages extend the problem space in new ways. German writes *compound nouns* without spaces (e.g., *Computerlinguistik* 'computational linguistics'; *Lebensversicherungsgesellschaftsangestellter* 'life insurance company employee'). This phenomenon reaches its limit case with major East Asian Languages (e.g., Chinese, Japanese, Korean, and Thai), where text is written without any spaces between words. An example is shown in Figure 2.3. One approach here is to perform *word segmentation* as prior linguistic processing. Methods of word segmentation vary from having a large dictionary and taking the longest dictionary match with some heuristics for unknown words to the use of machine learning sequence models such as hidden Markov models or conditional random fields, trained over hand-segmented words. All such methods make mistakes sometimes, and so one is never guaranteed a consistent unique tokenization. The other approach is to abandon word-based

COMPOUNDS

WORD SEGMENTATION

莎拉波娃现在居住在美国东南部的佛罗里达。今年 4 月 9 日，莎拉波娃在美国第一大城市纽约度过了 18 岁生日。生日派对上，莎拉波娃露出了甜美的微笑。

► **Figure 2.3** Chinese text example. The figure shows the standard unsegmented form of Chinese text (using the simplified characters that are standard in mainland China). There is no whitespace between words, nor even between sentences – the apparent space after the Chinese period (○) is just a typographical illusion caused by placing the character on the left side of its square box. The first sentence is just words in Chinese characters with no spaces between them. The second and third sentences show arabic numerals and punctuation occurring to break up the Chinese characters, but there are still no spaces between words.

a and as at be by for from has he
in is it its of on s said that the to
was will with year

► **Figure 2.4** A stop list of 25 words common in Reuters-RCV1.

indexing and to do all indexing via just short subsequences of characters (character *k*-grams), regardless of whether particular sequences cross word boundaries or not. Two reasons why this approach is appealing are that an individual Chinese character is more like a syllable than a letter and usually has some semantic content, and that, given the lack of standardization of word breaking in the writing system, it is not always clear where word boundaries should be placed anyway. Even in English, some cases of where to put word boundaries are just orthographic conventions – think of *notwithstanding* vs. *not to mention* or *into* vs. *on to* – but people are educated to write the words with consistent use of spaces.

2.2.2 Dropping common terms: stop words

STOP WORDS

Sometimes, some extremely common and semantically non-selective words are excluded from the dictionary entirely. These words are called *stop words*. The general strategy for determining a stop list is to sort the terms by frequency, and then to take the most frequent terms, often hand-filtered for their semantic content relative to the domain of the documents being indexed, as a *stop list*, the members of which are then discarded during indexing. An example of a stop list is shown in Figure 2.4. Using a stop list significantly reduces the number of postings that a system has to store: we will present some statistics on this in Chapter 5 (see Table 5.1, page 66). And a lot of the time not indexing stop words does little harm: keyword searches with terms

STOP LIST

like the and by don't seem very useful. However, this is not true for phrase searches. The phrase query "President of the United States", which contains two stop words, is more precise than President AND "United States". The meaning of flights to London is likely to be lost if the word to is stopped out. Some special query types are disproportionately affected. Some song titles and well known pieces of verse consist entirely of words that are usually on stop lists ("To be or not to be", "Let It Be", "I don't want to be", ...).

The general trend in IR systems over time has been from standard use of quite large stop lists (200–300 terms) to very small stop lists (7–12 terms) to no stop list whatsoever. Web search engines often do not use stop lists. Some of the design of modern IR systems has focused precisely on how we can exploit the statistics of language so as to be able to cope with common words in better ways. We will show in Chapter 5 how good compression techniques greatly reduce the cost of storing the postings for common words. And we will discuss in Chapter 7 how an IR system with impact-sorted indexes can terminate scanning a postings list early when weights get small, and hence it does not incur a large additional cost on the average query even though postings lists for stop words are very long. So for most modern IR systems, the additional cost of including stop words is not that big – neither in terms of index size nor in terms of query processing time.

2.2.3 Normalization (equivalence classing of terms)

Having cut our documents (and also our query, if it is a free text query) into tokens, the easy case is if tokens in the query just match tokens in the token list of the document. However, there are many cases when things are not quite the same but you would like a match to occur. For instance, if someone queries on *USA*, he might hope that it would also match with *U.S.A.*

EQUIVALENCE CLASSES

The most standard way to normalize is to implicitly create *equivalence classes*, which are normally named after one member of the set. For instance, if the tokens *anti-discriminatory* and *antidiscriminatory* are both mapped onto the latter, in both the document text and queries, then searches for one term will retrieve documents that contain either.

The advantage to doing things by just having mapping rules that remove things like hyphens is that the equivalence classing to be done is implicit, rather than being fully calculated in advance: the terms that happen to become identical as the result of these rules are the equivalence classes. It is only easy to write rules of this sort that remove stuff. Since the equivalence classes are implicit, it is not obvious when you might want to add things. For instance, it would be hard to know to turn *antidiscriminatory* into *anti-discriminatory*.

An alternative to creating equivalence classes is to maintain relations between unnormalized tokens. This method can be extended to hand-constructed

Query term	Terms in documents that should be matched
Windows	Windows
windows	Windows, windows
window	window, windows

► **Figure 2.5** An example of how asymmetric expansion of query terms can usefully model users' expectations.

lists of synonyms such as *car* and *automobile*, a topic we discuss further in Chapter 9. These term relationships can be achieved in two ways. The usual way is to index unnormalized tokens and to maintain a query expansion list of multiple dictionary entries to consider for a certain query term. A query term is then effectively a disjunction of several postings lists. The alternative is to perform the expansion during index construction. When the document contains automobile, we index it under car as well (and, usually, also vice-versa). Use of either of these methods is considerably less efficient than equivalence classing, as one has more postings to store and merge. The first method adds a query expansion dictionary and requires more processing at query time, while the second method requires more space for storing postings. In most circumstances, expanding the space required for the postings lists is seen as more disadvantageous.

However, these approaches are more flexible than equivalence classes because the expansion lists can overlap while not being identical. This means there can be an asymmetry in expansion. An example of how such an asymmetry can be exploited is shown in Figure 2.5: if the user enters windows, we wish to allow matches with the capitalized *Windows* operating system, but this is not plausible if the user enters window, even though it is plausible for this query to also match lowercase *windows*.

The best amount of equivalence classing or query expansion to do is a fairly open question. Doing some definitely seems a good idea. But doing a lot can easily have unexpected consequences of broadening queries in unintended ways. For instance, equivalence-classing *U.S.A.* and *USA* to the latter by deleting periods from tokens might at first seem very reasonable, given the prevalent pattern of optional use of periods in acronyms. However, if I put in as my query term *C.A.T.*, I might be rather upset if it matches every appearance of the word *cat* in documents.[4]

Below we present some of the other forms of normalization that are commonly employed and how they are implemented. In many cases they seem helpful, but they can also do harm. One can worry about many details of

4. At the time we wrote this chapter (Aug. 2005), this was actually the case on Google: the top result for the query *C.A.T.* was a site about cats, the Cat Fanciers Web Site http://www.fanciers.com/.

equivalence classing, but it often turns out that providing processing is done consistently to the query and to documents, the fine details may not have much aggregate effect on performance.

Accents and diacritics. Diacritics on characters in English have a fairly marginal status, and one might well want *resume* and *résumé* to match, or *naive* and *naïve*. This can be done by normalizing tokens to remove diacritics. In many other languages, diacritics are a regular part of the writing system and distinguish different sounds. Occasionally words are distinguished only by their accents. Nevertheless, the important question is usually not prescriptive or linguistic but is a question of how users are likely to write queries for these words. In many cases, users will enter queries for words without diacritics, whether for reasons of speed, laziness, limited software, or habits born of the days when it was hard to use non-ASCII text on many computer systems. In these cases, it might be useful to equate all words to a form without diacritics.

Capitalization/case-folding. A common strategy is to do case folding by reducing all letters to lower case. Often this is a good idea: it will allow instances of *Automobile* at the beginning of a sentence to match with a query of *automobile*. It will also help on a web search engine when most of your users type in *ferrari* when they are interested in a *Ferrari* car. On the other hand, this case folding can equate things that might better be kept apart. Many proper nouns are derived from common nouns and so are distinguished only by case, including companies (*General Motors*, *The Associated Press*), government organizations (*the Fed* vs. *fed*) and person names (*Bush*, *Black*). We already mentioned an example of unintended query expansion with acronyms, which involved not only acronym normalization (*C.A.T.* $\rightarrow$ *CAT*) but also case-folding (*CAT* $\rightarrow$ *cat*).

For English, an alternative to lowercasing every token is to just lowercase some tokens. The simplest heuristic is to lowercase words at the beginning of sentences, which are usually ordinary words that have been capitalized, and also everything in a title that is all uppercase or in title case where non-function words are all capitalized, leaving mid-sentence capitalized words as capitalized (which is usually correct). This will mostly avoid case-folding in cases where distinctions should be kept apart. The same task can be done more accurately by a machine learning sequence model which uses more features to make the decision of when to case-fold. This is known as *truecasing*. However, trying to get things right in this way probably doesn't help if your users usually use lowercase regardless of the correct case of words. Thus, lowercasing everything often remains the most practical solution.

TRUECASING

ノーベル平和賞を受賞したワンガリ・マータイさんが名誉会長を務めるＭＯＴＴＡＩＮＡＩキャンペーンの一環として、毎日新聞社とマガジンハウスは「私 の、もったいない」を募集します。皆様が日ごろ「もったいない」と感じて実践していることや、それにまつわるエピソードを８００字以内の文章にまとめ、簡 単な写真、イラスト、図などを添えて１０月２０日までにお送りください。大賞受賞者には、５０万円相当の旅行券とエコ製品２点の副賞が贈られます。

► **Figure 2.6** Example of Japanese text. As well as not segmenting words, as in the earlier Chinese example, Japanese makes use of multiple writing systems, intermingled together. The part in latin letters is actually a Japanese expression, but has been taken up as the name of an environmental campaign by 2004 Nobel Peace Prize winner Wangari Maathai. His name is written using the katakana syllabary in the middle of the first line. The first four characters of the final line shows a monetary amount that one would like to match with ¥500,000 (500,000 Japanese yen).

Other issues in English. Other possible normalizations are quite idiosyncratic and particular to English. For instance, you might wish to equate *ne'er* and *never* or the British spelling *colour* and the American spelling *color*. Dates, times and similar items come in multiple formats, presenting additional challenges. You might wish to collapse together *3/12/91* and *Mar. 12, 1991*.

Other languages. Other languages again present distinctive issues in equivalence classing. The French word for *the* has distinctive forms based not only on the gender (masculine or feminine) and number of the following noun, but also depending on whether the following word begins with a vowel. In such cases one may wish to equivalence class these various forms of *the*: *le*, *la*, *l'*, *les*.

Japanese is a well-known difficult writing system, as illustrated in Figure 2.6. Modern Japanese is standardly an intermingling of multiple alphabets, principally Chinese characters, the two syllabaries (hiragana and katakana) and western characters (Latin letters, Arabic numerals, and various symbols). While there are strong conventions and standardization through the education system over the choice of writing system, in many cases the same word can be written with multiple writing systems. For example, a word may be written in katakana for emphasis (somewhat like italics). Or a word may sometimes be written in hiragana and sometimes in Chinese characters. Successful retrieval thus requires complex equivalence classing across the writing systems. In particular, an end user might commonly present a

query entirely in hiragana, because it is easier to type, just as Western end users commonly use all lowercase.

Document collections being indexed can include documents from many different languages. Or a single document can easily contain text from multiple languages. For instance, a French email might quote clauses from a contract document written in English. Most commonly, the language is detected and language-particular tokenization and normalization rules are applied at a predetermined granularity, such as whole documents or individual paragraphs, but this still will not correctly deal with cases where language changes occur for brief quotations. When document collections contain multiple languages, a single index may have to contain terms of several languages. One option is to run a language identification classifier on documents and then to tag terms in the dictionary for their language. Or this tagging can simply be omitted, since collisions across languages are relatively rare.

When dealing with foreign or complex words, particularly foreign names, the spelling may be unclear or there may be variant transliteration standards giving different spellings (for example, *Chebyshev* and *Tchebycheff* or *Beijing* and *Peking*). One way of dealing with this is to use heuristics to equivalence class or expand terms with phonetic equivalents. The traditional and best known such algorithm is the Soundex algorithm, which we cover in Section 3.3 (page 48).

2.2.4 Stemming and lemmatization

For grammatical reasons, documents are going to use different forms of a word, such as *organize*, *organizes*, and *organizing*. Additionally, there are families of derivationally related words with similar meanings, such as *democracy*, *democratic*, and *democratization*. In many situations, it seems as if it would be useful for a search for one of these words to return documents that contain another word in the set.

The goal of both stemming and lemmatization is to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form. For instance:

am, are, is ⇒ be
car, cars, car's, cars' ⇒ car

The result of this mapping of text will be something like:

the boy's cars are different colors ⇒
the boy car be differ color

STEMMING However, the two terms differ in their flavor. *Stemming* suggests a crude

LEMMATIZATION

heuristic process that chops off the ends of words in the hope of achieving this goal correctly most of the time, whereas *lemmatization* suggests doing this properly with the use of a dictionary and morphological analysis of words, normally aiming to return the base or dictionary form of a word. If confronted with the token *saw*, stemming might return just *s*, whereas lemmatization would attempt to return either *see* or *saw* depending on whether the use of the token was as a verb or a noun. The two may also differ in that stemming most commonly collapses derivationally related words, whereas lemmatization commonly only collapses the different inflectional forms of a lemma. Linguistic processing at one of these levels is often supplied as additional plug-in components to the indexing process, and a number of such components exist, both commercial and open-source.

PORTER STEMMER

The most common algorithm for stemming English, and one that has repeatedly been shown to be empirically very effective, is *Porter's algorithm* (Porter 1980). The entire algorithm is too long and intricate to present here, but we will indicate its general nature. Porter's algorithm consists of 5 phases of word reductions, applied sequentially. Within each phase there are various conventions to select rules, such as selecting the one that applies to the longest suffix. In the first phase, this convention is used with the following rules:

Rule			**Example**		
SSES	$\rightarrow$	SS	caresses	$\rightarrow$	caress
IES	$\rightarrow$	I	ponies	$\rightarrow$	poni
SS	$\rightarrow$	SS	caress	$\rightarrow$	caress
S	$\rightarrow$		cats	$\rightarrow$	cat

Many of the later rules make use of a concept of the *measure* of a word, which loosely checks whether it is long enough that it is reasonable to regard the matching portion as a suffix rather than part of the stem of a word. For example, the rule:

$(m > 1)$ EMENT $\rightarrow$

would map *replacement* to *replac*, but not *cement* to *c*.

The canonical site for the Porter Stemmer (Porter 1980) is:

http://www.tartarus.org/~martin/PorterStemmer/

Other stemmers exist, including the older, one-pass Lovins stemmer (Lovins 1968), and newer entrants like the Paice/Husk stemmer (Paice 1990); see:

http://www.cs.waikato.ac.nz/~eibe/stemmers/
http://www.comp.lancs.ac.uk/computing/research/stemming/

Figure 2.7 presents an informal comparison of the different behavior of these stemmers. Stemmers use language-particular rules, but they require less

Sample text: Such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation

Lovins stemmer: such an analys can reve featur that ar not eas vis from th vari in th individu gen and can lead to a pictur of expres that is mor biolog transpar and acces to interpres

Porter stemmer: such an analysi can reveal featur that ar not easili visibl from the variat in the individu gene and can lead to a pictur of express that is more biolog transpar and access to interpret

Paice stemmer: such an analys can rev feat that are not easy vis from the vary in the individ gen and can lead to a pict of express that is mor biolog transp and access to interpret

► **Figure 2.7** A comparison of three stemming algorithms on a sample text.

knowledge than a complete dictionary and morphological analysis to correctly lemmatize words. Particular domains may also require special stemming rules. However, the exact stemmed form does not matter, only the equivalence classes it forms.

LEMMATIZER

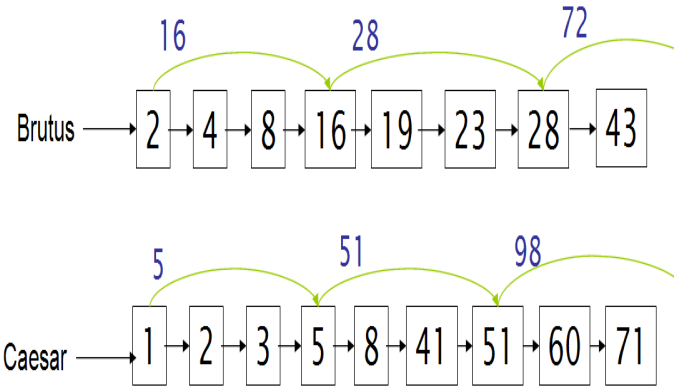
Rather than using a stemmer, one can use a *lemmatizer*, a tool from Natural Language Processing which does full morphological analysis and can accurately identify the lemma for each word. Doing full morphological analysis produces at most very modest benefits for retrieval. It is hard to say more, because either form of normalization tends not to improve information retrieval performance in aggregate – at least not by very much. While it helps a lot for some queries, it equally hurts performance a lot for others. Stemming increases recall while harming precision. As an example of what can go wrong, note that the Porter stemmer stems all of the following words:

operate operating operates operation operative operatives operational

to oper. However, since *operate* in its various forms is a common verb, we would expect to lose considerable precision on queries such as the following with Porter stemming:

operational AND research
operating AND system
operative AND dentistry

For a case like this, moving to using a lemmatizer would not fix things: particular inflectional forms are used in particular collocations.



► **Figure 2.8** Postings lists with skip pointers. The postings merge can use a skip pointer when the end point is still less than the item on the other list.

2.3 Postings lists, revisited

In the remainder of this chapter, we will discuss extensions to postings list structure and ways to increase the efficiency of using postings lists.

2.3.1 Faster postings merges: Skip pointers

Recall the basic postings list merge operation from Chapter 1, shown in Figure 1.6: we walk through the two postings lists simultaneously, in time linear in the total number of postings entries. If the list lengths are m and n, the merge takes $O(m + n)$ operations. Can we do better than this? That is, empirically, can we usually process postings list merges in sublinear time? We can, if the index isn't changing too fast.

SKIP LIST

One way to do this is to use a *skip list* by augmenting postings lists with skip pointers (at indexing time), as shown in Figure 2.8. Skip pointers are effectively *shortcuts* that allow us to avoid processing parts of the postings list that will not figure in the search results. The two questions are then where to place skip pointers and how to do efficient merging that takes into account skip pointers.

Consider first efficient merging, with Figure 2.8 as an example. Suppose we've stepped through the lists in the figure until we have matched 8 on each list and moved it to the results list. We advance both pointers, giving us 16 on the upper list and 41 on the lower list. The smallest item is then the element 16 on the top list. Rather than simply advancing the upper pointer, we first check the skip list pointer and note that 28 is also less than 41. Hence

we can follow the skip list pointer, and avoid stepping to 19 and 23 on the upper list.

Where do we place skips? There is a tradeoff. More skips means shorter skip spans, and that we are more likely to skip. But it also means lots of comparisons to skip pointers, and lots of space storing skip pointers. Fewer skips means few pointer comparisons, but then long skip spans which means that there will be fewer opportunities to skip. A simple heuristic for placing skips, which has been found to work well in practice is that for a postings list of length P, use $\sqrt{P}$ evenly-spaced skip pointers. This heuristic can be improved upon; it ignores any details of the distribution of query terms.

Building effective skip pointers is easy if an index is relatively static; it is harder if a postings list keeps changing because of updates (a malicious deletion strategy can render skip lists ineffective).

2.3.2 Phrase queries

Many complex or technical concepts and many organization and product names are multiword compounds or other phrases. We would like to be able to pose a query such as Stanford University by treating it as a phrase so that a sentence in a document like *The inventor Stanford Ovshinsky never went to university.* is not a match. Most recent search engines support a double quotes syntax ("stanford university") for *phrase queries*, which has proven to be very easily understood and successfully used by users. Around 10% of web queries are now phrase queries, and many more are implicit phrase queries (such as person names), entered without use of double quotes. To be able to support such queries, it is no longer sufficient for postings lists to be simply lists of documents that contain individual terms. In this section we consider two approaches to supporting phrase queries and their combination. An important criterion is that not only should phrase queries be possible, but they should also be efficient. A related but distinct concept is term proximity weighting, where a document is preferred to the extent that the query terms appear close to each other in the text. This technique is covered in Chapter 7 in the context of ranked retrieval.

PHRASE QUERIES

Biword indexes

One approach to handling phrases is to consider every pair of consecutive terms in a document as a phrase. For example the text "Friends, Romans, Countrymen" would generate the biwords:

friends romans
romans countrymen

In this model, we treat each of these biwords as a dictionary term. The ability to do two-word-phrase query processing is immediate. Longer phrases can be processed by breaking them down. The query *stanford university palo alto* can be broken into the Boolean query on biwords:

stanford university AND university palo AND palo alto

One would expect this query to work fairly well in practice, but there can and will be occasional false positives. Without examining the documents, we cannot verify that the documents matching the above Boolean query do actually contain the original 4 word phrase.

PHRASE INDEX

The concept of a biword index can be extended to longer sequences of words, and if it includes variable length word sequences, it is generally referred to as a *phrase index*. Maintaining exhaustive phrase indexes for phrases of length greater than two is a daunting prospect, but towards the end of this section we discuss the utility of partial phrase indexes in compound indexing schemes.

For many text indexing purposes, nouns have a special status in describing the concepts people are interested in searching for. But related nouns can often be divided from each other by various function words, in phrases such as *the abolition of slavery* or *renegotiation of the constitution*. These needs can be incorporated into the biword indexing model in the following way. First, we tokenize the text and perform part-of-speech-tagging (POST).[5] We can then bucket terms into (say) Nouns, including proper nouns, (N) and function words, including articles and prepositions, (X), among other classes. Now deem any string of terms of the form NX*N to be an extended biword. Each such extended biword is made a term in the dictionary. For example:

renegotiation	of	the	constitution
N	X	X	N

To process a query using such an extended biword index, we need to also parse it into N's and X's, and then segment the query into extended biwords, which can be looked up in the index.

Some isues with using biword or extended biword indexes include:

- Parsing longer queries into Boolean queries. From the algorithm that we have so far, the query

  tangerine trees and marmalade skies

  is parsed into

5. Part of speech taggers classify words as nouns, verbs, etc. – or, in practice, often as finer-grained classes like "plural proper noun". Many fairly accurate (c. 96% accuracy) part-of-speech taggers now exist, usually trained by machine learning methods on hand-tagged text. See, for instance, Manning and Schütze (1999, ch. 10).

⟨be, 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 17, 191, 291, 430, 434;
5: 363, 367; . . . ⟩

► **Figure 2.9** Positional index example. For example, be has total collection frequency 993,477, and occurs in document 1 at positions 7, 18, 33, etc.

tangerine trees AND trees marmalade AND marmalade skies

Which does not seem a fully desirable outcome.

- False positives. As noted before, index matches will not necessarily contain longer queries.
- Index blowup. Using a biword dictionary has the result of greatly expanding the dictionary size.
- No natural treatment of individual terms. Individual term searches are not naturally handled in a biword index, and so a single word index must be present in addition.

Positional indexes

POSITIONAL INDEX

For the reasons given, biword indexes are not the standard solution. Rather, *positional indexes* are most commonly employed. Here, for each term in the dictionary, we store in the postings lists entries of the form docID: position1, position2, . . . , as shown in Figure 2.9. As discussed in Chapter 5, one can compress position values/offsets substantially. Nevertheless, use of a positional index expands required postings storage significantly.

To process a phrase query such as to be or not to be, you extract inverted index entries for each distinct term: to, be, or, not. As before, you would start with the least frequent term and then work to further restrict the list of possible candidates. In the merge operation, the same general technique is used as before, but rather than simply checking that both documents are on a postings list, you also need to check that their positions of appearance in the document are compatible with the phrase query being evaluated. This requires working out offsets between the words. Suppose the two postings lists for to and be are:

to: 993427; 2:5[1,17,74,222,551]; 4:5[8,16,190,429,433]; 7:3[13,23,191]; . . .
be: 178239; 1:2[17,19]; 4:5[17,191,291,430,434]; 5:3[14,19,101]; . . .

We first look for documents that contain both terms. Then, we look for places in the lists where there is an occurrence of *be* with a token index one higher than a position of *to*, and then we look for another occurrence of each word with token index 4 higher than the first occurrence. In the above lists, the pattern of occurrences that is a possible match is:

to: . . . ; 4:. . . ,429,433; . . .
be: . . . ; 4:. . . ,430,434; . . .

The same general method is applied for within k word proximity searches, of the sort we saw in Figure 1.9 (page 13):

LIMIT! /3 STATUTE /3 FEDERAL /2 TORT

Here, $/k$ means "within k words of". Clearly, positional indexes can be used for such queries; biword indexes cannot. More is still needed to handle "within the same sentence" or "within the same paragraph" proximity searches. We discuss issues of document zones in Chapter 6.

Positional index size. Even if we compress position values/offsets as we discuss in Chapter 5, adopting a positional index expands postings storage requirements substantially. However, most applications have little choice but to accept this since most users, in either Boolean or free text query interfaces, now expect to have the functionality of phrase and proximity searches.

Let's work out the space implications of having a positional index. We need an entry for each occurrence of a term, not just one per document in which it occurs. The index size thus depends on the average document size. The average web page has less than 1000 terms, but things like SEC stock filings, books, and even some epic poems easily reach 100,000 terms. Consider a term with frequency 1 in 1000 terms on average. The result is that large documents cause a two orders of magnitude increase in the postings list size:

Document size	Postings	Positional postings
1000	1	1
100,000	1	100

While the exact numbers depend on the type of documents and the language being indexed, some rough rules of thumb are to expect a positional index to be 2 to 4 times as large as a non-positional index, and to expect a compressed positional index to be about half the size of the original uncompressed documents. Specific numbers for an example collection are given in Table 5.1 (page 66) and Table 5.6 (page 79).

Combination schemes

The strategies of biword indexes and positional indexes can be fruitfully combined. If users commonly query on particular phrases, such as Michael

Jackson, it is quite inefficient to keep merging positional postings lists. A combination strategy uses a phrase index, or just a biword index, for certain queries and uses a positional index for other phrasal queries. Good queries to include in the phrase index are ones known to be common based on recent previous querying behavior. But this is not the only criterion: the most expensive phrasal queries to evaluate are ones where the individual words are common but the desired phrase is comparatively rare. Adding *Britney Spears* as a phrase index entry may only give a speedup factor to that query of about 3, since most documents that mention either word are valid return documents, whereas adding *The Who* as a phrase index entry may speed up that query by a factor of 1000. Hence, having the latter is more desirable, even if it is a relatively less common query.

Willams et al. (2004) evaluate an even more sophisticated scheme which employs indexes of both these sorts and additionally a partial next word index as a halfway house between the first two strategies. They conclude that such a strategy allows a typical mixture of web phrase queries to be completed in one quarter of the time taken by use of a positional index alone, while taking up 26% more space than use of a positional index alone.

Changing technology

Choosing the optimal encoding for an inverted index is an ever-changing game, because it is strongly dependent on underlying computer technologies and their relative speeds and sizes. Traditionally, CPUs were slow, and so highly compressed techniques were not optimal. Now CPUs are fast and disk is slow, so reducing disk postings list size is paramount. However, if you're running a search engine with everything in memory then the equation changes again.

2.4 References and further reading

EAST ASIAN LANGUAGES

Exhaustive discussion of the character-level processing of *East Asian languages* can be found in Lunde (1998). For further discussion of Chinese word segmentation, see Sproat et al. (1996), Sproat and Emerson (2003), Tseng et al. (2005), and Gao et al. (2005).

Lita et al. (2003) present a method for truecasing. Natural language processing work on computational morphology is presented in (Sproat 1992, Beesley and Karttunen 2003).

LANGUAGE IDENTIFICATION

Language identification was perhaps first explored in cryptography; for example Konheim (1981) presents a character-level n-gram language identification algorithm. While other methods such as looking for particular distinctive function words and letter combinations have been used, with the

advent of widespread digital text, many people explored the character *n*-gram technique, and found it to be highly successful (Beesley 1998, Dunning 1994, Cavnar and Trenkle 1994). Written language identification is regarded as a fairly easy problem, while spoken language identification remains more difficult; see Hughes et al. (2006) for a recent survey.

STEMMING

Experiments on and discussion of the positive and negative impact of *stemming* can be found in the following works: Salton (1989), Harman (1991), Krovetz (1995), Hull (1996).

SKIP LISTS

The classic presentation of skip pointers for IR can be found in Moffat and Zobel (1996). Extended techniques are discussed in Boldi and Vigna (2005). The main paper in the algorithms literature is Pugh (1990), which uses multilevel skip pointers to give expected $O(\log n)$ list access (the same expected efficiency as using a tree data structure) with less implementational complexity. Using skip pointers is not necessarily effective in practice. While Moffat and Zobel (1996) reported conjunctive queries running about five times faster with the use of skip pointers, Bahle et al. (2002) report that, with modern CPUs, using skip lists instead slows down search because it expands the size of the postings list (i.e., disk I/O dominates performance).

2.5 Exercises

Exercise 2.1

Are the following statements true or false?

a. In a Boolean retrieval system, stemming never lowers precision.
b. In a Boolean retrieval system, stemming never lowers recall.
c. Stemming increases the size of the lexicon.
d. Stemming should be invoked at indexing time but not while doing a query

Exercise 2.2

The following pairs of words are stemmed to the same form by the Porter stemmer. Which pairs would you argue shouldn't be conflated. Give your reasoning.

a. abandon/abandonment
b. absorbency/absorbent
c. marketing/markets
d. university/universe
e. volume/volumes

Exercise 2.3

Given the partial postings list in Figure 2.9, which of the documents could contain the phrase *to be or not to be*?

Exercise 2.4

Shown below is a portion of the positional index in the format term: doc1: position1, position2, . . . ; doc2: position1, position2, . . . ; etc.

angels: 2:36,174,252,651; 4:12,22,102,432; 7:17;
fools: 2:1,17,74,222; 4:8,78,108,458; 7:3,13,23,193;
fear: 2:87,704,722,901; 4:13,43,113,433; 7:18,328,528;
in: 2:3,37,76,444,851; 4:10,20,110,470,500; 7:5,15,25,195;
rush: 2:2,66,194,321,702; 4:9,69,149,429,569; 7:4,14,404;
to: 2:47,86,234,999; 4:14,24,774,944; 7:199,319,599,709;
tread: 2:57,94,333; 4:15,35,155; 7:20,320;
where: 2:67,124,393,1001; 4:11,41,101,421,431; 7:16,36,736;

Which document(s) if any meet each of the following queries, where each expression within quotes is a phrase query?

a. "fools rush in"

b. "fools rush in" AND "angels fear to tread"

Exercise 2.5

We have a two-word query. For one term the postings list consists of the following 16 entries:

[4,6,10,12,14,16,18,20,22,32,47,81,120,122,157,180]

and for the other it is the one entry postings list [47]. Work out how many comparisons would be done to intersect the two postings lists, assuming: (i) standard postings lists, and (ii) using postings lists stored with skip pointers, with a skip length of $\sqrt{P}$, as suggested in Section 2.3.1. Briefly justify your answer.

Exercise 2.6

Adapt the linear merge of postings to handle proximity queries. Can you make it work in linear time for any token proximity value k?

3 *Tolerant retrieval*

In Chapters 1 and 2 we developed the ideas underlying inverted indexes for handling Boolean and proximity queries. Here we begin by studying *wildcard queries*: a query such as *a*e*i*o*u*, which seeks documents containing any word that includes all the five vowels in sequence. The * symbol indicates any (possibly empty) string of characters. We then turn to other forms of imprecisely posed queries, focusing on spelling errors. Users make spelling errors either by accident, or because the term they are searching for (e.g., Chebyshev) has no unambiguous spelling in the collection.

3.1 Wildcard queries

A query such as mon* is known as a *trailing wildcard query*, because the * symbol occurs only once, at the end of the search string. A search tree on the dictionary is a convenient way of handling trailing wildcard queries: we walk down the tree following the symbols m, o and n in turn, at which point we can enumerate the set W of terms in the dictionary with the prefix mon. Finally, we use the inverted index to retrieve all documents containing any term in W. Typically, the search tree data structure most suited for such applications is a *B-tree* – a search tree in which every internal node has a number of children in the interval $[a, b]$, where a and b are appropriate positive integers. For instance when the index is partially disk-resident, the integers a and b are determined by the sizes of disk blocks. Section 3.4 contains pointers to further background on search trees and B-trees.

But what about wildcard queries in which the * symbol is not constrained to be at the end of the search string? Before handling the general case, we mention a slight generalization of trailing wildcard queries. First, consider *leading wildcard queries*, or queries of the form *mon. Consider a *reverse B-tree* on the dictionary – one in which each root-to-leaf path of the B-tree corresponds to a term in the dictionary written *backwards:* thus, the term lemon would, in the B-tree, be represented by the path root-n-o-m-e-l. A walk down

the reverse B-tree then enumerates all terms R in the dictionary with a given suffix.

In fact, using a regular together with a reverse B-tree, we can handle an even more general case: wildcard queries in which there's a single * symbol, such as se*mon. To do this, we use the regular B-tree to enumerate the set W of dictionary terms beginning with the prefix se, then the reverse B-tree to enumerate the set R of dictionary terms ending with the suffix mon. Next, we take the intersection $W \cap R$ of these two sets, to arrive at the set of terms that begin with the prefix se and end with the suffix mon. Finally, we use the inverted index to retrieve all documents containing any terms in this intersection. We can thus handle wildcard queries that contain a single * symbol using two B-trees, the normal B-tree and a reverse B-tree.

3.1.1 General wildcard queries

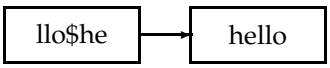
We now study two techniques for handling general wildcard queries. Both techniques share a common strategy: express the given wildcard query w as a Boolean query Q on a specially constructed index, such that the answer to Q is a superset of the set of dictionary terms matching w. Then, we check each term in the answer to Q against w, discarding those dictionary terms in the answer that do not match w. At this point we have the dictionary terms matching w and can resort to the standard inverted index.

Permuterm indexes

PERMUTERM INDEX

Our first special index for general wildcard queries is the *permuterm index*, a form of inverted index. First, we introduce a special symbol $ into our character set, to mark the end of a term; thus, the term hello is represented as hello$. Next, we construct a permuterm index, in which the dictionary consists of all rotations of each term (with the $ terminating symbol appended). The postings for each rotation consist of all dictionary terms containing that rotation. Figure 3.1 gives an example of such a permuterm index entry for a rotation of the term hello. We refer to the set of rotated terms in the permuterm index as the *permuterm dictionary*.

How does this index help us with wildcard queries? Consider the wildcard query m*n. The key is to *rotate* such a wildcard query so that the * symbol appears at the end of the string – thus the rotated wildcard query becomes n$m*. Next, we look up this string in the permuterm index, where the entry n$m* points to the terms man and men. What of longer terms matching this wildcard query, such as moron? The permuterm index contains six rotations of moron, including n$moro. Now, n$m is a prefix of n$moro. Thus, when we traverse the B-tree into the permuterm dictionary seeking n$m, we find n$moro in the sub-tree, pointing into the original lexicon term moron.



► **Figure 3.1** Example of an entry in the permuterm index.

Exercise 3.1

In the permuterm index, each permuterm dictionary term points to the original lexicon term(s) from which it was derived. How many original lexicon terms can there be in the postings list of a permuterm dictionary term?

Exercise 3.2

Write down the entries in the permuterm index dictionary that are generated by the term mama.

Exercise 3.3

If you wanted to search for s*ng in a permuterm wildcard index, what key(s) would one do the lookup on?

Now that the permuterm index enables us to identify the original lexicon terms matching a wildcard query, we look up these terms in the standard inverted index to retrieve matching documents. We can thus handle any wildcard query with a single * symbol. But what about a query such as fi*mo*er? In this case we first enumerate the terms in the lexicon that are in the permuterm index of er$fi*. Not all such lexicon terms will have the string mo in the middle - we filter these out by exhaustive enumeration, checking each candidate to see if it contains mo. In this example, the term fishmonger would survive this filtering but filibuster would not. We then run the surviving terms through the regular inverted index for document retrieval. One disadvantage of the permuterm index is that its dictionary becomes quite large, including as it does all rotations of each term.

Notice the close interplay between the B-tree and the permuterm index above; indeed, it suggests that the structure should perhaps be viewed as a permuterm B-tree; however, we follow traditional terminology here in describing the permuterm index as distinct from the B-tree that allows us to select the rotations with a given prefix.

► **Figure 3.2** Example of a postings list in a 3-gram index. Here the 3-gram etr is illustrated.

3.1.2 *k*-gram indexes

We now present a second technique, known as the *k*-gram index, for processing wildcard queries. A *k-gram* is a sequence of *k* characters. Thus cas, ast and stl are all 3-grams occurring in the word castle. We use a special character $ to denote the beginning or end of a word, so the full set of 3-grams generated for castle is: $ca, cas, ast, stl, tle, le$.

k-GRAM INDEX A *k-gram index* is an index in which the dictionary consists of all *k*-grams that occur in any word in the lexicon. Each postings list points from a *k*-gram to all lexicon words containing that *k*-gram. For instance, the 3-gram etr would point to lexicon words such as metric and retrieval. An example is given in Figure 3.2.

How does such an index help us with wildcard queries? Consider the wildcard query re*ve. We are seeking documents containing any term that begins with re and ends with ve. Accordingly, we run the Boolean query $re AND ve$. This is looked up in the 3-gram index and yields a list of matching terms such as relive, remove and retrieve. Each of these matching terms is then looked up in the inverted index to yield documents matching the query.

There is however a difficulty with the use of *k*-gram indexes, that demands one further step of processing. Consider using the 3-gram index described above for the query red*. Following the process described above, we first issue the Boolean query $re AND red to the 3-gram index. This leads to a match on words such as retired, which contain the conjunction of the two 3-grams $re and red, yet do not match the original wildcard query red*.

To cope with this, we introduce a *post-filtering* step, in which the words enumerated by the Boolean query on the 3-gram index are checked individually against the original query red*. This is a simple string-matching operation and weeds out words such as retired that do not match the original query. Words that survive are then run against the inverted index as usual.

Exercise 3.4

Consider again the query fi*mo*er from Section 3.1.1. What Boolean query on a bigram index would be generated for this query? Can you think of a word that meets this Boolean query but does not satisfy the permuterm query in Section 3.1.1?

Exercise 3.5

Give an example of a sentence that falsely matches the wildcard query mon*h if the search were to simply use a conjunction of bigrams.

We have seen that a wildcard query can result in multiple words being enumerated, each of which becomes a one-word query on the inverted index. Search engines do allow the combination of wildcard queries using Boolean operators, for example, re*d AND fe*ri. What is the appropriate semantics for such a query? Since each wildcard query turns into a disjunction of one-word queries, the appropriate interpretation of this example is that we have a conjunction of disjunctions: we seek all documents that contain any word matching re*d *and* any word matching fe*ri.

Even without Boolean combinations of wildcard queries, the processing of a wildcard query can be quite expensive. A search engine may support such rich functionality, but most commonly, the capability is hidden behind an interface (say an "Advanced Query" interface) that most users never use. Surfacing such functionality often encourages users to invoke it, increasing the processing load on the engine.

3.2 Spelling correction

We next look at the problem of correcting spelling errors in queries: for instance, we may wish to retrieve documents containing the word carrot when the user types the query carot. We look at two approaches to this problem: the first based on *edit distance* and the second based on *k-gram overlap*. For instance, Google reports (http://www.google.com/jobs/britney.html) that the following are all observed misspellings of the query britney spears: britian spears, britney's spears, brandy spears and prittany spears (interestingly, all misspellings listed in this report appear to spell spears correctly). Before getting into the algorithmic details of these methods, we first review how search engines surface spell-correction as a user experience.

EDIT DISTANCE

3.2.1 Implementing spelling correction

Search engines implement this feature in one of several ways:

1. On the query carot always retrieve documents containing carot as well as any "spell-corrected" version of carot, including carrot and tarot.

2. As in (1) above, but only when the query term carot is absent from the dictionary.

3. As in (1) above, but only when the original query (in this case carot) returned fewer than a preset number of documents (say fewer than five documents).

4. When the original query returns fewer than a preset number of documents, the search interface presents a *spell suggestion* to the end user: this suggestion consists of the spell-corrected query term(s).

3.2.2 Forms of spell correction

We focus on two specific forms of spell correction that we refer to as *isolated word* correction and *context-sensitive* correction. In isolated word correction, we attempt to correct a single query term at a time – even when we have a multiple-term query. The carot example exemplifies this type of correction. Such isolated word correction would fail to detect, for instance, that the query flew form Heathrow contains a mis-spelling of the word from – because each term in the query is correctly spelled in isolation.

We begin by examining two methods for isolated-word correction: edit distance, and k-gram overlap. We then proceed to context-sensitive correction.

3.2.3 Edit distance

Given two character strings S_1 and S_2, the *edit distance* between them is the minimum number of *edit operations* required to transform S_1 into S_2. Most commonly, the edit operations allowed for this purpose are: (i) insert a character into a string; (ii) delete a character from a string and (iii) replace a character of a string by another character. For example, the edit distance between cat and dog is 3. In fact, the notion of edit distance can be generalized to allowing varying weights for different kinds of edit operations, for instance a higher weight may be placed on replacing the character s by the character p, than on replacing it by the character a (the latter being closer to s on the keyboard). However, the remainder of our treatment here will focus on the case in which all edit operations have the same weight.

Exercise 3.6

If $|S|$ denotes the length of string S, show that the edit distance between S_1 and S_2 is never more than $\max\{S_1, S_2\}$.

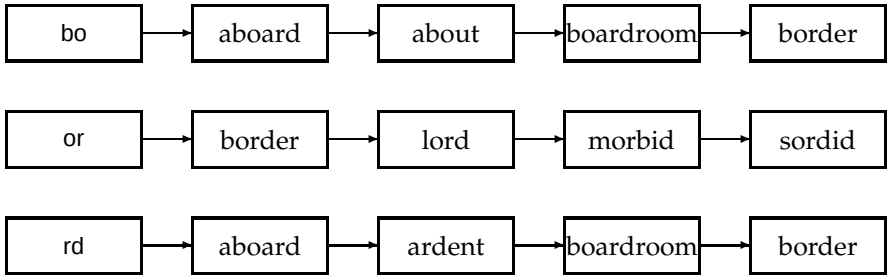
```
m[i,j] = d(s1[1..i], s2[1..j])

m[0,0] = 0
m[i,0] = i, i=1..|s1|
m[0,j] = j, j=1..|s2|

m[i,j] = min(m[i-1,j-1]
             + if s1[i]=s2[j] then 0 else 1 fi,
             m[i-1, j] + 1,
             m[i, j-1] + 1 ), i=1..|s1|, j=1..|s2|
```

► **Figure 3.3** Dynamic programming algorithm for computing the edit distance between strings s1 and s2.

It is well-known how to compute the (weighted) edit distance between two strings in time $O(|S_1| * |S_2|)$, where $|S|$ denotes the length of a string S. The idea is to use the dynamic programming algorithm in Figure 3.3.

The spell correction problem however is somewhat different from that of computing edit distance: given a set $\mathcal{S}$ of strings (corresponding to terms in the dictionary) and a query string q, we seek the string(s) in $\mathcal{S}$ of least edit distance from q. We may view this as a decoding problem, but one in which the codewords (the strings in $\mathcal{S}$) are prescribed in advance. The obvious way of doing this is to compute the edit distance from q to each string in $\mathcal{S}$, before selecting the string(s) of minimum edit distance. This exhaustive search is inordinately expensive. Accordingly, a number of heuristics are used in practice to efficiently retrieve dictionary terms likely to have low edit distance to the query q.

The simplest such heuristic is to restrict the search to dictionary terms beginning with the same letter as the query string; the hope would be that spelling errors do not occur in the first character of the query. A more sophisticated variant of this heuristic is to use the permuterm index, omitting the end-of-word symbol. Consider the set of all rotations of the query string q. For each rotation r from this set, we traverse the B-tree into the permuterm index, thereby retrieving all dictionary terms that have a rotation beginning with r. For instance, if q is mase and we consider the rotation r=sema, we would retrieve dictionary terms such as semantic and semaphore. Unfortunately, we would miss more pertinent dictionary terms such as mare and mane. To address this, we refine this rotation scheme: for each rotation, we omit a suffix of ℓ characters before performing the B-tree traversal. The value of ℓ could depend on the length of q; for instance, we may set it to 2. This ensures that each term in the set R of dictionary terms retrieved includes a



► **Figure 3.4** Matching at least two of the three 2-grams in the query bord.

"long" substring in common with q.

Exercise 3.7

Show that if the query term q is edit distance 1 from a dictionary term t, then the set R includes t.

3.2.4 *k*-gram indexes

We can use the k-gram index of Section 3.1.2 to assist with retrieving dictionary terms with low edit distance to the query q. In fact, we will use the k-gram index to retrieve dictionary terms that have many k-grams in common with the query. We will argue that for reasonable definitions of "many k-grams in common", the retrieval process is essentially that of a single scan through the postings for the k-grams in the query string q.

The 2-gram (or *bigram*) index in Figure 3.4 shows (a portion of) the postings for the three bigrams in the query bord. Suppose we wanted to retrieve dictionary terms that contained at least two of these three bigrams. A single scan of the postings (much as in Chapter 1) would let us enumerate all such words; in the example of Figure 3.4 we would enumerate aboard, boardroom and border.

This straightforward application of the linear scan merge of postings immediately reveals the shortcoming of simply requiring matched dictionary terms to contain a fixed number of k-grams from the query q: terms like boardroom, an implausible "correction" of bord, get enumerated. Consequently, we require more nuanced measures of the overlap in k-grams between a dictionary term and q. The linear scan merge can be adapted when the measure of overlap is the *Jaccard coefficient* for measuring the overlap between two sets

JACCARD COEFFICIENT

A and B, defined to be $|A \cap B|/|A \cup B|$. The two sets we consider are the set of k-grams in the query q, and the set of k-grams in a dictionary term. As the scan proceeds, we proceed from one dictionary term to the next, computing on the fly the Jaccard coefficient between the query q and a dictionary term t. If the coefficient exceeds a preset threshold, we add t to the output; if not, we move on to the next term in the postings. To compute the Jaccard coefficient, we need the set of k-grams in q and t.

Exercise 3.8

Compute the Jaccard coefficients between the query bord and each of the terms in Figure 3.4 that contain the bigram or.

Since we are scanning the postings for all k-grams in q, we immediately have these k-grams on hand. What about the k-grams of t? In principle we could enumerate these on the fly from t; in practice this is not only slow but potentially infeasible since, in all likelihood, the postings entries themselves do not contain the complete string t but rather an integer encoding of t. The crucial observation is that we only need the length of the string t, to compute the Jaccard coefficient. To see this, recall the example of Figure 3.4 and consider the point when the postings scan for query $q =$ bord reaches term $t =$ boardroom. We know that two bigrams match. If the postings stored the (pre-computed) number of bigrams in boardroom (namely, 8), we have all the information we require. For the Jaccard coefficient is $2/(8+3-2)$; the numerator is obtained from the number of postings hits (2, from bo and rd) while the denominator is the sum of the number of bigrams in bord and boardroom, less the number of postings hits.

We could replace the Jaccard coefficient by other measures that allow efficient on the fly computation during postings scans. How do we use these for spell correction? One method that has some empirical support is to first use the k-gram index to enumerate a set of candidate dictionary terms that are potential corrections of q. We then compute the edit distance from q to each term in this set, selecting terms from the set with small edit distance to q.

3.2.5 Context sensitive spelling correction

Isolated word correction would fail to correct typographical errors such as flew form Heathrow, where all three query terms are correctly spelled. When a phrase such as this retrieves few documents, a search engine may offer the corrected query flew from Heathrow. The simplest way to do this is to enumerate corrections of each of the three query terms (using the methods above) even though each query term is correctly spelled, then try substitutions of each correction in the phrase. For the example flew form Heathrow, we enumerate such phrases as fled form Heathrow and flew fore Heathrow. For each such

substitute phrase, the engine runs the query and determines the number of matching results.

This enumeration can be expensive if we find many corrections of the individual terms, since we could encounter a large number of combinations of alternatives. Several heuristics are used to trim this space. In the example above, as we expand the alternatives for flew and form, we retain only the most frequent combinations in the collection. For instance, we would retain flew from as an alternative to try and extend to a three-term corrected query, but perhaps not fled fore or flea form. The choice of these alternatives is governed by the relative frequencies of biwords occurring in the collection: in this example, the biword fled fore is likely to be rare compared to the biword flew from. Then, we only attempt to extend the list of top biwords (such as flew from) in the first two terms, to corrections of Heathrow. As an alternative to using the biword statistics in the collection, we may use the logs of queries issued by users; these could of course include queries with spelling errors.

Exercise 3.9

Consider the four-term query catched in the rye and suppose that each of the query terms has five alternative terms suggested by isolated word correction. How many possible corrected phrases must we consider if we do not trim the space of corrected phrases, but instead try all six variants for each of the terms?

Exercise 3.10

For each of the prefixes of the query — thus, catched, catched in and catched in the — we have a number of substitute prefixes arising from each term and its alternatives. Suppose that we were to retain only the top 10 of these substitute prefixes, as measured by its number of occurrences in the collection. We eliminate the rest from consideration for extension to longer prefixes: thus, if batched in is not one of the 10 most common 2-term queries in the collection, we do not consider any extension of batched in as possibly leading to a correction of catched in the rye. How many of the possible substitute prefixes are we eliminating at each phase?

Exercise 3.11

Are we guaranteed that retaining and extending only the 10 commonest substitute prefixes of catched in will lead to one of the 10 commonest substitute prefixes of catched in the?

3.3 Phonetic correction

Our final technique for tolerant retrieval has to do with *phonetic* correction: misspellings that arise because the user types a word that sounds like the target word. The main idea here is to generate, for each term, a "phonetic hash" so that similar-sounding words hash to the same value. The idea owes its origins to work in international police departments from the early 20th century, seeking to match names for wanted criminals despite the names being

spelled differently in different countries. It is mainly used to correct phonetic misspellings in proper nouns.

SOUNDEX Algorithms for such phonetic hashing are collectively known as *Soundex* algorithms; they all build on the following scheme:

1. Turn every term to be indexed into a 4-character reduced form. Build an inverted index from these reduced forms to the original terms; call this the soundex index.
2. Do the same with query terms.
3. When the query calls for a soundex match, search this soundex index.

The variations in different soundex algorithms have to do with the conversion of terms to 4-character forms. A commonly used conversion results in a 4-character code, with the first character being a letter of the alphabet and the other three being digits between 0 and 9.

1. Retain the first letter of the word.
2. Change all occurrences of the following letters to '0' (zero): 'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Change letters to digits as follows:

   B, F, P, V to 1.

   C, G, J, K, Q, S, X, Z to 2.

   D,T to 3.

   L to 4.

   M, N to 5.

   R to 6.
4. Remove all pairs of consecutive digits.
5. Remove all zeros from the resulting string. Pad the resulting string with trailing zeros and return the first four positions, which will consist of a letter followed by three digits.

This algorithm rests on a few observations: (1) vowels are viewed as interchangeable, in transcribing names; (2) consonants with similar sounds (e.g., D and T) are put in equivalence classes. This leads to related names often having the same soundex codes.

For an example of a soundex map, Hermann maps to H655. Given a query (say herman), we compute its soundex code and then retrieve all dictionary terms matching this soundex code from the soundex index, before running the resulting query on the standard term-document inverted index.

3.4 References and further reading

Garfield (1976) gives one of the first complete descriptions of the permuterm index. One of the earliest formal treatments of spelling correction was due to Damerau (1964). The notion of edit distance is due to Levenshtein (1965). Peterson (1980) and Kukich (1992) developed variants of methods based on edit distances, culminating in a detailed study of several methods due to Zobel and Dart (1995). The soundex algorithm is attributed to Margaret K. Odell and Robert C. Russelli (from U.S. patents granted in 1918 and 1922); the version described draws on Bourne and Ford (1961).

Probabilistic models ("noisy channel" models) for spelling correction were pioneered by Kernighan et al. (1990) and further developed by Brill and Moore (2000) and Toutanova and Moore (2002).

Knuth Knuth (1997) is a comprehensive source for information on search trees, including B-trees.

4 *Index construction*

In this chapter, we look at how to construct an inverted index. To do this, we essentially have to perform a sort of the postings file. This is non-trivial for the large data sets that are typical in modern information retrieval. We will first introduce block merge indexing, an efficient single-machine algorithm designed for static collections (Section 4.1). For very large collections like the web, indexing has to be *distributed* over large compute clusters with hundreds or thousands of machines (Section 4.2). Collections with frequent changes require *dynamic indexing* so that changes in the collection are immediately reflected in the index (Section 4.3). Finally, some complicating issues that can arise in indexing – such as security and indexes for weighted retrieval – will be covered in Section 4.4.

Other issues are outside the scope of this chapter. Frequently, documents are not on a local file system, but have to be spidered or crawled, as we discuss in Chapter 20. Also, the indexer needs raw text, but documents are encoded in many ways, as we mentioned in Chapter 2. Finally documents are often encapsulated in varied content management systems, email applications and databases. While most of these applications can be accessed via http, native APIs are usually more efficient. Although we do not discuss these issues, the reader should be aware that building the subsystem that feeds raw text to the indexing process can in itself be a challenging problem.

4.1 Construction of large indexes

The basic steps in constructing a non-positional index are depicted in Figures 1.4 and 1.5 (page 8). We first make a pass through the collection assembling all postings entries (i.e., termID-docID pairs). We then sort the entries with the termID as the dominant key and docID as the secondary key. Finally, we organize the docIDs for each term into a postings list and compute statistics like term and document frequency. For small collections, all this can

► **Figure 4.1** Document from the Reuters newswire. The Reuters-RCV1 collection consists of 800,000 documents of the type shown that were disseminated by Reuters in 1996 and 1997.

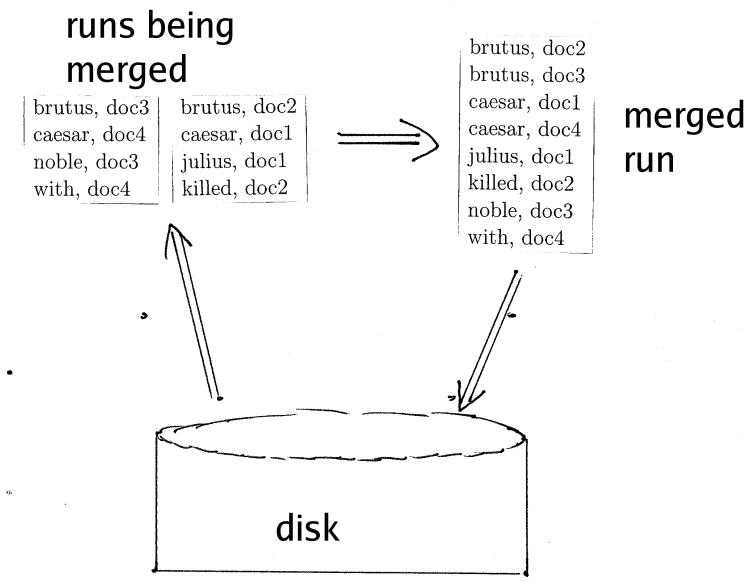
symbol	statistic	value
N	documents	800,000
L	avg. # word tokens per document	200
M	word types	400,000
	avg. # bytes per word token (incl. spaces/punct.)	6
	avg. # bytes per word token (without spaces/punct.)	4.5
	avg. # bytes per word type	7.5
	non-positional postings	100,000,000

► **Table 4.1** Collection statistics for Reuters-RCV1. Values are rounded for the computations in this chapter. The unrounded values are: 806,791 documents, 222 terms (or word tokens) per document, 391,523 distinct terms, 6.04 bytes per term with spaces and punctuation, 4.5 bytes per term without spaces and punctuation, 7.5 bytes per word type and 96,969,056 non-positional postings.

be done in memory. In this chapter, we describe methods for large collections that require the use of secondary storage for index construction.

REUTERS-RCV1 We will work with the *Reuters-RCV1* collection as our model collection in this chapter, a collection with roughly one gigabyte of text. It consists of about 800,000 documents that were sent over the Reuters newswire during a one year period between August 20, 1996, and August 19, 1997. A typical document is shown in Figure 4.1. Reuters-RCV1 covers a wide range of international topics, including politics, business, sports and (as in the example) science. Some key statistics of the collection are shown in Table 4.1.[1]

1. The numbers in this table correspond to the third line in Table 5.1, page 66. For the definitions



► **Figure 4.2** Merging in block merge indexing. Two blocks ("runs being merged") are loaded from disk into memory, sorted in memory ("merged run") and then written back to disk.

Reuters-RCV1 has 100 million postings. If we use 12 bytes per posting (4 bytes each for termID, docID and frequency), the index will demand about 1.2 gigabyte of temporary storage. Typical collections today are often one or two orders of magnitude larger than Reuters-RCV1. You can easily see how such collections will overwhelm even large computers if we tried to sort their postings files in memory. If the size of the postings file is within a small factor of available memory, then the compression techniques introduced in Chapter 5 can help; but the postings file of many large collections cannot be fit into memory even after compression.

With main memory insufficient, we have to go to disk. We could apply a generic sorting algorithm with disk instead of main memory as storage, but this would be too slow since it would require too many random disk seeks.

BLOCK MERGE ALGORITHM

One solution is the *block merge algorithm*, which implements a good tradeoff between fast but scarce memory and slow but plentiful disk. The first step of the algorithm is to accumulate postings in memory until a block of a fixed size is full. We choose the block size to fit comfortably into memory to permit a fast in-memory sort. The block is then "inverted" (sorted to construct an

of word token and word type, see Chapter 2, page 20.

inverted index for the text covered by it) and written to disk. Applying this to Reuters-RCV1 and assuming we can fit 1.6 million postings into memory, we end up with 64 blocks with 1.6 million postings each.

In the second step, the algorithm merges the 64 blocks into one large index. (An example with two blocks is shown in Figure 4.2.) To do this, we maintain small read and write buffers for the 64 blocks we are reading and the final index we are writing.

How expensive is block merge indexing? It is dominated by the time it takes to read and write blocks, the disk transfer time. For Reuters-RCV1 this time is 4 hours:

$$\begin{aligned} & 64 \text{ blocks} \times 1.6 \times 10^6 \text{ entries} \times 12 \text{ bytes} \\ \times \quad & b \\ \times \quad & 2 \text{ transfers} \\ \approx \quad & 41 \text{ minutes} \end{aligned} \tag{4.1}$$

where 1.6×10^6 is the block size and $b = 10^{-6}$ seconds/byte is the byte transfer rate. Note that each byte in a merge step needs to be read and written and is hence transferred twice.

In addition to disk transfer, for sorting we also need to scan through the collection to create the initial list of postings, sort the 64 blocks, and write them to disk before they are processed by block merge. Exercise 4.4 asks you to compute the total index construction time that includes these steps.

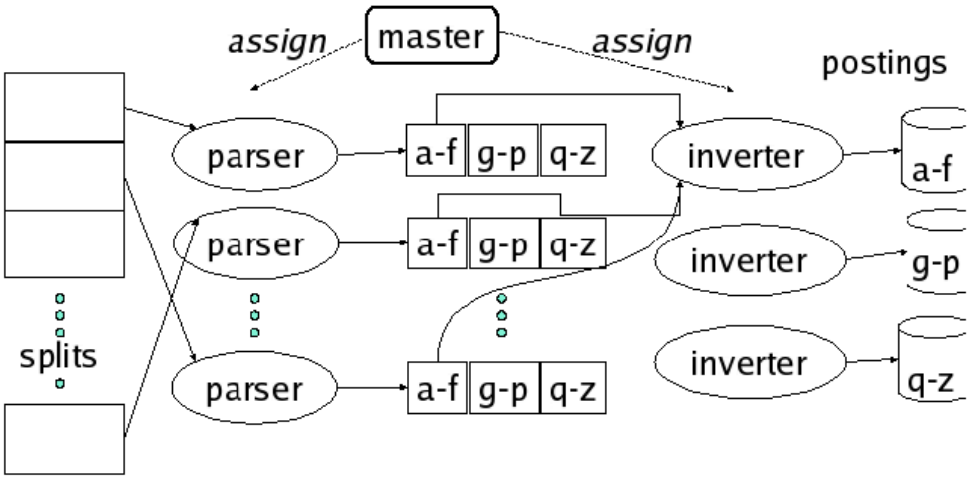
The attentive reader will have noticed that Reuters-RCV1 is not particularly large in an age when 100 gigabyte are standard on personal computers. Indeed, at the time of writing (August 2006), we were able to sort the Reuters-RCV1 postings file on a not overly beefy server in about 20 minutes simply by using the standard unix command sort. The techniques we have described are needed, however, for collections that are several orders of magnitude larger.

4.2 Distributed indexing

Collections are often so large that index construction cannot be performed on a single machine. This is particularly true of the World Wide Web for which we need large compute clusters to construct any reasonably sized web index. Web search engines therefore use *distributed indexing* algorithms for index construction. Here, we introduce *MapReduce*, an architecture for distributed computing that can be applied to index construction.

MAPREDUCE

MapReduce is designed for large compute clusters. The point of a cluster is to solve large computing problems using cheap commodity machines



► **Figure 4.3** An example of distributed indexing with MapReduce. In the map phase, the collection is partitioned into *n* splits and the master assigns splits to parsers until the entire collection is parsed. Each parser writes its postings to a local file that is segmented according to the key, in this case the initial letter. The postings for terms starting with a–f are collected in the first segment etc. The segments themselves are not sorted. In the reduce phase, the master assigns each key range to an inverter. An inverter collects all segments for its key range, sorts them, and writes to the sorted postings file ("postings") for that range. Adapted from Dean and Ghemawat (2004).

(or *nodes*) as opposed to building a supercomputer with specialized hardware. While hundreds or thousands of machines are available in such clusters, individual machines can fail at any time. One requirement for robust distributed indexing is therefore that we divide the work up into chunks that we can easily assign and – in case of failure – reassign. These chunks must be of a size that a machine built from standard parts (processor, memory, disk) can handle. A *master node* directs the process of assigning and reassigning tasks to individual worker nodes.

MASTER NODE

The map and reduce phases of MapReduce split up the computing job into chunks that are manageable for standard machines. The various steps are shown in Figure 4.3. First, the input data, in our case a collection of web pages, is split into *n splits* where the size of the split is determined by the computing environment (e.g., 16 MB or 64 MB if that is the block size of the file system). Splits are not preassigned to machines, but are instead assigned by the master node on an ongoing basis: as a machine finishes processing one split, it is assigned the next one. If a machine dies or becomes a laggard

SPLITS

due to hardware problems, the split it is working on is simply reassigned to another machine.

KEY-VALUE PAIRS

In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of *key-value pairs*. For indexing, a key value pair has the form (term,docID) and, not surprisingly, is nothing other than a posting. The map phase consists of mapping splits of the input data to key-value pairs. This amounts to a parsing task in indexing (processing of markup, stemming, tokenization etc.). Each parser writes its output to a local intermediate file.

For the reduce phase, we want all values for a given key to be stored close together, so that they can be read and processed quickly. This is achieved by partitioning the keys into j partitions and having the parsers write key-value pairs for each partition into a separate segment file. In Figure 4.3, the partitions are according to initial word letters: a–f, g–p, q–z, so $j = 3$. (We chose these key ranges for ease of exposition. In general, key ranges are not contiguous.) The partitions are defined by the person who operates the indexing system (Exercise 4.9). The parsers then write corresponding segment files, one for each partition.

INVERTERS

Sorting the segment files is the task of the *inverters* in the reduce phase. The master assigns each partition to a different inverter (and, as in the case of parsers, reassigns partitions in case of failing or slow inverters). Each partition (corresponding to k segment files, one on each parser) is processed by one inverter. Finally, key-value pairs are sorted and written to the final sorted postings lists ("postings" in the figure). This completes the construction of the inverted index.

Parsers and inverters are not separate sets of machines. The master identifies idle machines and assigns tasks to them. So the same machine can be a parser in the map phase and an inverter in the reduce phase. And there are often other jobs that run in parallel with index construction, so in between being a parser and an inverter a machine might do some crawling or another unrelated task.

The data reduction in the reduce phase of distributed indexing is comparatively small. This makes distributed indexing a somewhat atypical application of MapReduce. More typical is a word counter, which uses word-count pairs as key-value pairs, corresponding to the frequency of a word in a document. The reduce phase reduces all entries for a word to a single count, a much larger reduction of the data volume than in indexing.

To minimize write times before the data are reduced, parsers write segment files to *local disk*. In the reduce phase, the master communicates to an inverter the locations of the relevant segment files (e.g., for the a-f partition). Each partition file only requires one sequential read since all data relevant to a particular inverter were written to a single segment file by the parser. This setup minimizes the amount of network traffic needed during indexing.

Schema of map and reduce functions		
map:	input	$\rightarrow$ list(k, v)
reduce:	(k,list(v))	$\rightarrow$ output
Instantiation of the schema for index construction		
map:	web collection	$\rightarrow$ list(term, docID)
reduce:	(term,list(docID))	$\rightarrow$ inverted list

► **Figure 4.4** Map and reduce functions in MapReduce. In general, the map function produces a list of key-value pairs. All values for a key are collected into one list in the reduce phase. This list is then further processed (e.g., sorted). The instantiations of the two functions for index construction are shown.

Figure 4.4 shows the general schema of the MapReduce functions. Input and output are often lists of key-value pairs themselves, so that several MapReduce jobs can be run in sequence. In fact, this was the design of the Google indexing system in 2004. What we have just covered in this section corresponds to only one of 5 to 10 MapReduce operations in that indexing system.

MapReduce offers a robust and conceptually simple framework for implementing index construction in a distributed environment. By providing a semi-automatic method for splitting index construction into smaller tasks, it can scale to almost arbitrarily large collections, given compute clusters of sufficient size.

For large web search engines, the construction of the web index closely interacts with the distribution of the index over query servers, the machines that process queries and return hit lists. This topic will be covered in Section 20.3 (page 326).

4.3 Dynamic indexing

Thus far we have assumed that the document collection is static. This is fine for collections that change infrequently or never like the Bible or Shakespeare. But most collections constantly add, delete and update documents. This means that new terms need to be added to the dictionary; and postings lists need to be updated for existing terms.

The simplest way to achieve this is to periodically reconstruct the index from scratch. This is a good solution if the number of changes over time is small and a delay in making new documents searchable is acceptable – or if enough resources are available to construct a new index while the old one is still available for querying. See below.

If there is a requirement that new documents be included quickly, one solution is to maintain two indexes: a large main index and a small *auxiliary index*

AUXILIARY INDEX

```
LMERGEPOSTING(indexes, J_0, posting)
 1  J_0 ← MERGE(J_0, {posting})
 2  if |J_0| = n
 3    then for i ← 0 to ∞
 4         do if I_i ∈ indexes
 5              then J_{i+1} ← MERGE(I_i, J_i)
 6                   indexes = indexes − {I_i}
 7              else I_i ← J_i
 8                   indexes = indexes ∪ {I_i}
 9                   BREAK
10         J_0 ← ∅

LOGARITHMICMERGE()
1  J_0 ← ∅
2  indexes ← ∅
3  while true
4  do LMERGEPOSTING(indexes, J_0, GETNEXTPOSTING())
```

▶ **Figure 4.5** Logarithmic merging. Each posting is initially added to in-memory index J_0 by LMERGEPOSTING. When J_0 reaches its maximum size n it is stored on disk as I_0 or, if I_0 exists, merged with I_0 into a new index J_1. Then J_1 is either stored as I_1 or merged into J_2 and so on. LOGARITHMICMERGE shows how J_0 and *indexes* are initialized. A search request is serviced by querying in-memory J_0 and all indexes I_i in *indexes* and merging the results.

that stores new documents. The auxiliary index is kept in memory. Searches are run across both indexes and results merged. Deletions are stored in an invalidation bit vector. We can then filter out deleted documents before returning the search result. Documents are updated by deleting and reinserting them.

If the size of the auxiliary index is n and we have indexed T postings so far, then each posting has been processed T/n times, resulting in an overall time complexity of $O(T^2)$ for this index construction scheme. We can do better by bounding the number of indexes to be $O(log_r T)$. As before, up to n postings are accumulated in memory. When the limit n is reached, an index I_0 with $r^0 n$ postings is created on disk. The next time J_0 is full, it is merged with I_0 to create an index I_1 of size $r^1 n$ etc. This algorithm is called *logarithmic merging* and given in pseudocode for $r = 2$ in Figure 4.5. Overall index construction time is $O(T \log T)$. We trade this efficiency gain for a slow-down of query processing by a factor of $\log T$ since we now need to merge results from $\log T$ indexes as opposed to just two indexes (main and auxiliary) before.

LOGARITHMIC MERGING

Having multiple indexes complicates the maintenance of collection-wide statistics. For example, it affects the spell correction algorithm in Section 3.2 (page 43) that selects the corrected alternative with the most hits. With multiple indexes and an invalidation bit vector, the correct number of hits for a term is no longer a simple lookup. And we will see in Chapter 6 that collection-wide statistics are also important in ranking. Finally, we need to merge very large indexes in logarithmic indexing, an expensive operation that slows down response times of the search system during the merge. Large merges are rare, but rare periods of poor performance may not be acceptable.

Because of these drawbacks of multiple indexes, some large search engines do not do dynamic indexing. Instead, a new index is built from scratch periodically. Query processing is then switched from the new index and the old index is deleted.

4.4 Other types of indexes

This chapter only describes construction of non-positional indexes. Except for the much larger data volume we need to accommodate, the only difference for positional indexes is that (termID, docID, position) triples instead of (termID, docID) pairs have to be sorted. With this minor change, the algorithms discussed here can all be applied to positional indexes.

In the indexes we have considered so far, postings occur in document order. As we will see in the next chapter, this is advantageous for compression – instead of docIDs we can compress smaller *gaps* between IDs, thus reducing space requirements for the index. However, this structure for the index is not optimal when we build *weighted* (Chapters 6 and 7) – as opposed to Boolean – retrieval systems. In weighted systems, postings are often ordered according to weight or impact, with the highest-weighted postings occurring first. With this organization, scanning of long postings lists during query processing can usually be terminated early when weights have become so small that any further documents can be predicted to be of low similarity to the query (see Chapter 6). In a docID-sorted index, new documents are always inserted at the end of postings lists. In an impact-sorted index, the insertion can occur anywhere, thus complicating the update of the inverted index.

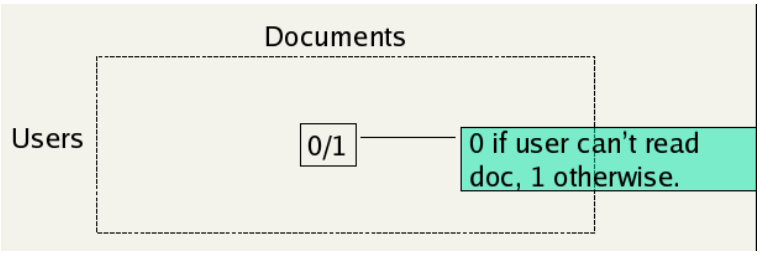
WEIGHTED RETRIEVAL SYSTEMS

SECURITY

Security is an important consideration for retrieval systems in corporations. The average employee should not be able to find the salary roster of the corporation, but it needs to be accessible by search to authorized managers. Users' result lists must not contain documents they are barred from opening since the very existence of a document can be sensitive information.

ACCESS CONTROL LISTS

User authorization is often mediated through *access control lists* or ACLs. ACLs can be incorporated into the retrieval system by treating as query terms



► **Figure 4.6** Access control lists in retrieval. Access restrictions can be incorporated into a retrieval system by inverting the user-document matrix shown. Element (i, j) is 1 if user i has access to document j and 0 otherwise. During query processing, a user's access postings list is intersected with the result list returned by the text part of the index.

names of users that can access a document (Figure 4.6) and then inverting the resulting user-document matrix. The inverted ACL index has, for each user, a "postings list" of documents they can access. Search results are then intersected with the user's access list. However, such an index is difficult to maintain when access permissions change and requires the processing of very long postings lists for users with access to large document subsets. User membership is therefore often verified directly on the ACL even though this slows down retrieval.

4.5 References and further reading

Witten et al. (1999), Chapter 5, contains an extensive treatment of the subject of index construction. The block merge algorithm is called text-based partitioning there, in contrast to lexicon-based partitioning, which processes the collection several times, each time for a different partition of the dictionary. MapReduce can be viewed as combining text-based and lexicon-based partitioning. Witten et al. present additional indexing algorithms with different tradeoffs of memory, disk space and time. In general, block merge indexing does well on all three counts. However, if speed of indexing is the main criterion, then other algorithms may be a better choice (see Witten et al. (1999), Tables 5.4 and 5.5). Heinz and Zobel (2003) have recently described a more efficient version of block-merge indexing that does not require the vocabulary to fit into main memory. This article is recommended as an up-do-date in-depth treatment of static index construction.

Harman and Candela (1990) show how to construct an index if little disk space is available, less than twice the size of the collection.

The MapReduce architecture was introduced by Dean and Ghemawat (2004).

symbol	statistic	value
s	disk seek	10 ms = 10^{-2} s
b	block transfer from disk (per byte)	1 μs = 10^{-6} s
p	other processor ops (e.g., compare&swap words)	0.1 μs = 10^{-7} s

► **Table 4.2** System parameters for the exercises. A disk seek is the time needed to position the disk head in a new position. The block transfer rate is the rate of transfer when the head is in the right position.

An open source implementation of MapReduce is available: http://lucene.apache.org/hadoop/. Ribeiro-Neto et al. (1999) and Melnik et al. (2001) describe other approaches to distributed indexing.

Logarithmic index construction was first published by Lester et al. (2005) and Büttcher and Clarke (2005), but it has been part of the Lucene information retrieval system (http://lucene.apache.org) for much longer. Query performance is poor during ongoing merges in logarithmic schemes. Improved dynamic indexing methods are discussed by Büttcher et al. (2006) and Lester et al. (2006). The latter paper also discusses the re-build strategy of replacing the old index by one built from scratch.

Heinz et al. (2002) compare data structures for accumulating text vocabularies in memory.

Büttcher and Clarke (2005) discuss security models for a common inverted index for multiple users.

A detailed characterization of the Reuters-RCV1 collection can be found in (Lewis et al. 2004).

4.6 Exercises

Exercise 4.1

If we need $n \log_2 n$ comparisons (where n is the number of postings) and 2 disk seeks for each comparison, how much time would index construction for Reuters-RCV1 take if we used disk instead of memory for storage? Use the system parameters in Table 4.2.

Exercise 4.2

Block merge indexing bears similarity to mergesort (Cormen et al. 1990). Why can't we construct the index by running a standard version of mergesort on the postings file?

Exercise 4.3

The estimate of 41 minutes for disk transfer time (Equation (4.1)) is pessimistic. Why would the actual transfer time be smaller?

	step	time
1	reading of collection	
2	64 initial quicksorts of 1.6M records each	
3	writing of 64 blocks	
4	total disk transfer time for merging	
5	time of actual merging	
	total	

► **Table 4.3** The five steps in constructing an index for Reuters-RCV1 in block merge indexing.

symbol	statistic	value
N	# documents	1,000,000,000
L	# terms per document	1000
M	# distinct terms	44,000,000

► **Table 4.4** Collection statistics for a large collection.

Exercise 4.4

Total index construction time in block merge indexing is broken down in Table 4.3. Fill out the time column of the table for Reuters-RCV1 assuming a system with the parameters given in Table 4.2.

Exercise 4.5

Repeat Exercise 4.4 for the larger collection in Table 4.4. Choose a block size that is realistic for current technology (remember that a block should easily fit into main memory). How many blocks do you need?

Exercise 4.6

The dictionary has to be created on the fly in block merge indexing to avoid an extra pass for compiling the dictionary. How would you do this on-the-fly creation?

Exercise 4.7

Compare memory, disk and time requirements of the naive in-memory indexing algorithm (assuming the collection is small enough) and block merge indexing.

Exercise 4.8

For $n = 15$ splits, $k = 10$ segments and $j = 3$ key partitions, how long would distributed index creation take for Reuters-RCV1 in a MapReduce architecture? Base your assumptions about cluster machines on Table 4.2.

Exercise 4.9

For optimal load balancing, the inverters in MapReduce must get segmented postings files of similar sizes. For a new collection, the distribution of key-value pairs may not be known in advance. How would you solve this problem?

Exercise 4.10

We claimed above that an auxiliary index can impair the quality of collection statistics. An example is the term weighting method idf, which is defined as $\log(N/df_i)$ where N is the total number of documents and df_i is the number of documents that term i occurs in (Section 6.2.1, page 88). Show that even a small auxiliary index can cause significant error in idf when it is computed on the main index only. Consider a rare term that suddenly occurs frequently (e.g., Flossie as in Tropical Storm Flossie).

Exercise 4.11

The algorithm in Figure 4.5 implements logarithmic merging for $r = 2$. Modify it so that an arbitrary r can be specified as a parameter.

Exercise 4.12

Can spell correction compromise document-level security? Consider the case where a spelling correction is based on documents the user doesn't have access to.

5 *Index compression*

Chapters 1 and 4 introduced the inverted index as the central data structure in information retrieval. The inverted index realizes a good time/space tradeoff, but we'd like to reduce its space requirements further by employing a number of compression techniques. Information retrieval involves large data sets, so index compression can mean significant cost savings for operational systems and is of great practical importance.

This chapter first gives a statistical characterization of the distribution of the entities we want to compress – terms and postings in large collections (Section 5.1). We then look at compression of the dictionary, using the dictionary-as-a-string method and blocked storage (Section 5.2). Section 5.3 describes two techniques for compressing the postings file, variable byte encoding and γ encoding.

5.1 Statistical properties of terms in information retrieval

As in the last chapter, we will use Reuters-RCV1 as our model collection (see Table 4.1, page 52). The term and postings statistics of Reuters-RCV1 in Table 5.1 show that preprocessing can affect the size of the dictionary and index greatly. Stemming and case folding reduce the number of word types by more than 30%,[1] and the number of non-positional postings by 4% and 3%, respectively. The treatment of the most frequent words is also important. The *rule of 30* states that the 30 most common words account for 30% of the tokens in written text (31% in the table). Eliminating the 150 commonest terms from indexing (as stop words; cf. Section 2.2.2, page 23) will cut 25–30% of the non-positional postings. But, while a stop list of 150 words reduces the number of postings by a quarter, this size reduction does not carry over to the size of the compressed index. As we will see later in this chapter, the postings lists of frequent words require only a few bits per posting after compression.

RULE OF 30

1. For the definitions of word token and word type, see Chapter 2, page 20.

	word types			non-positional postings			positional postings (word tokens)		
	size	Δ	cumul.	size	Δ	cumul.	size	Δ	cumul.
unfiltered	484,494			109,971,179			197,879,290		
no numbers	473,723	-2%	-2%	100,680,242	-8%	-8%	179,158,204	-9%	-9%
case folding	391,523	-17%	-19%	96,969,056	-3%	-12%	179,158,204	-0%	-9%
30 stop words	391,493	-0%	-19%	83,390,443	-14%	-24%	121,857,825	-31%	-38%
150 stop words	391,373	-0%	-19%	67,001,847	-30%	-39%	94,516,599	-47%	-52%
stemming	322,383	-17%	-33%	63,812,300	-4%	-42%	94,516,599	-0%	-52%

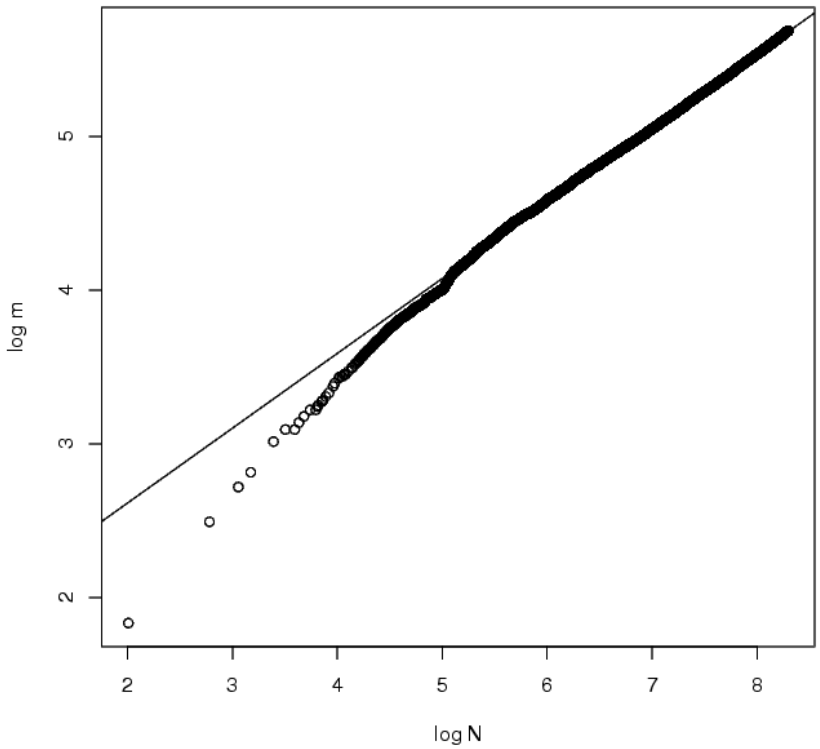
► **Table 5.1** The effect of preprocessing on the size of the inverted index for Reuters-RCV1. The numbers of word types (i.e., distinct words), non-positional postings, and positional postings (i.e., word tokens) roughly correspond to the (uncompressed) sizes of the dictionary, the postings file of a non-positional index and the postings file of a positional index, respectively. "Δ" indicates the reduction in size from the previous line, except that "30 stop words" and "150 stop words" both use "case folding" as their reference line. "cumul." is cumulative reduction (from unfiltered). Stemming was performed using the Porter stemmer (Chapter 2, page 29).

The deltas in the table are in a range typical of large collections. Note, however, that the percentage reductions can be very different for some text collections. For example, for a collection of web pages with a high proportion of non-English text, stemming with the Porter stemmer would reduce vocabulary size less than for an English-only collection.

LOSSLESS COMPRESSION
LOSSY COMPRESSION

The compression techniques described in the remainder of this chapter are *lossless*, that is, all information is preserved. Better compression ratios can be achieved with *lossy compression*, which discards some information. Case folding, stemming and stop word elimination are forms of lossy compression. Similarly, the vector space model (Chapter 7) and dimensionality reduction techniques like Latent Semantic Indexing (Chapter 18) create compact representations from which the original collection cannot be fully restored. Lossy compression makes sense when the "lost" information is unlikely ever to be used by the search system. For example, web search is characterized by a large number of documents, short queries, and users who only look at the first few pages of results. As a consequence, postings of documents that would only be used for hits far down the list can be discarded. Thus, there are retrieval scenarios where lossy methods are appropriate and achieve the best compression of the inverted index.

Before introducing techniques for compressing the dictionary, we want to estimate the number of word types in a collection. It is sometimes said that languages have a vocabulary of a certain size. The largest English dictionaries have about 500,000 entries. But dictionary size as an estimate of the



► **Figure 5.1** Heaps' law. Vocabulary size M as a function of collection size T (number of tokens) for Reuters-RCV1. For these data, the line $\log_{10} M = 0.49 * \log_{10} T + 1.64$ is the best least squares fit, so $k = 10^{1.64} \approx 44$ and $b = 0.49$. AXIS TITLES: CHANGE LOG N TO LOG 10 T AND LOG m TO LOG 10 M.

vocabulary in information retrieval is not very useful. Dictionaries do not include most names of people, locations, products and scientific entities like genes. These names still need to be included in the inverted index, so our users can search for them.

HEAPS' LAW A better way of getting a handle on $M = |V|$ is *Heaps' law*, which estimates vocabulary size as a function of collection size:

$$M = kT^b \tag{5.1}$$

where T is the number of tokens in the collection. Typical values for the parameters k and b are: $30 \leq k \leq 100$ and $b \approx 0.5$. The motivation for Heaps' law is that the simplest possible relationship between collection size and vocabulary size is linear in log-log space and the assumption of linearity

term	freq.	pointer to postings list	postings list
a	999,712	⟶	...
aardvark	92	⟶	...
...	5	⟶	...
zztop	71	⟶	...
space needed:	40 bytes	4 bytes	4 bytes

► **Figure 5.2** Storing the dictionary as an array of fixed-width entries.

is usually born out in practice as shown in Figure 5.1 for Reuters-RCV1. In this case, the fit is excellent for $T > 10^5 = 100{,}000$, for the parameter values $b = 0.49$ and $k = 44$. For example, for the first 1,000,020 tokens Heaps' law predicts 38,323 types:

$$44 \times 1{,}000{,}020^{0.49} \approx 38{,}323$$

The actual number is 38,365 types, very close to the prediction.

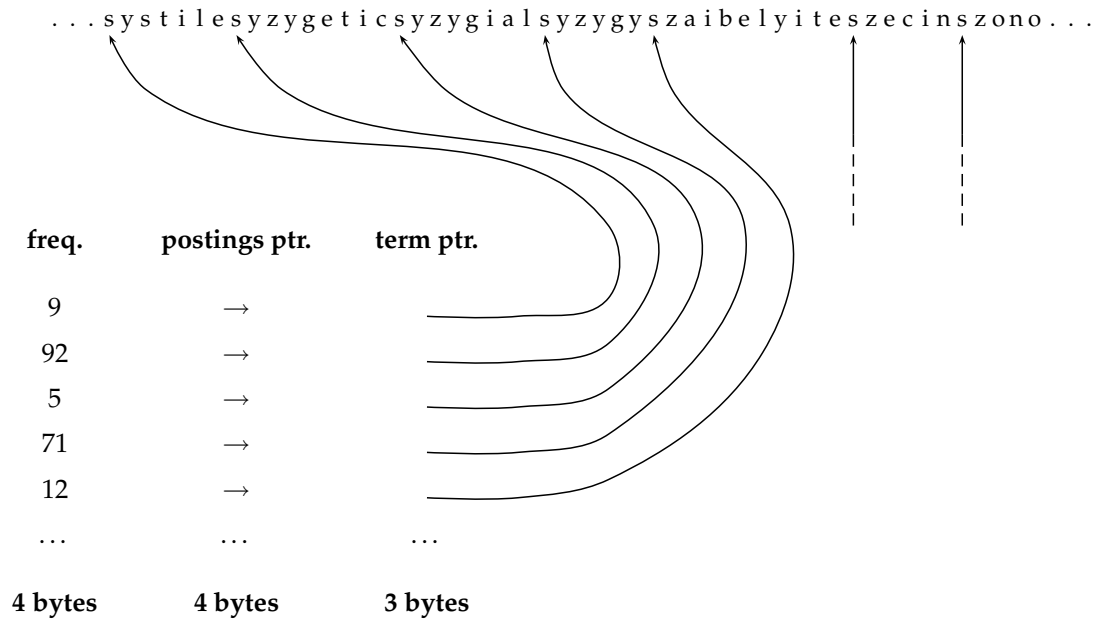
The parameter k is quite variable because vocabulary growth depends a lot on the nature of the collection and how it is processed. Case-folding and stemming reduce the growth rate of the vocabulary whereas including numbers and spelling errors will significantly increase it. Regardless of the values of the parameters for a particular collection, Heaps' law suggests that: (i) the dictionary size will continue to increase with more documents in the collection, rather than a maximum vocabulary size being reached, and (ii) the size of the dictionary will be quite large for large collections. So dictionary compression is important for an effective information retrieval system.

5.2 Dictionary compression

The dictionary is small compared to the postings file, so, if everything were on disk, dictionary compression would not save a lot of space. The main goal of compressing the dictionary is being able to fit it in main memory, or at least a large portion of it, in order to support a high throughput in query processing. This section presents a series of dictionary representations that achieve increasingly higher compression ratios.

5.2.1 Dictionary-as-a-string

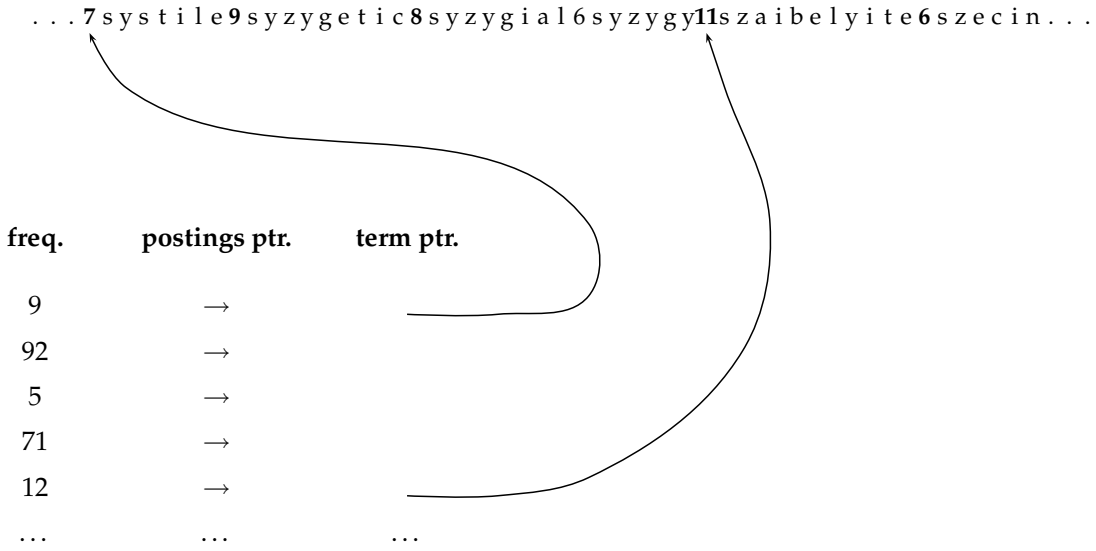
The simplest data structure for the dictionary is an array of fixed-width entries as shown in Figure 5.2. Assuming a unicode representation, we allocate



► **Figure 5.3** Dictionary-as-a-string storage. Pointers mark the end of the preceding term and the beginning of the next. For example, the first three terms in this example are systile (frequency 9), syzygetic (frequency 92) and syzygial (frequency 5).

2×20 bytes for the term itself since few terms have more than 20 characters in English, 4 bytes for its frequency and 4 bytes for the pointer to its postings list. Terms are looked up by binary search. For Reuters-RCV1, we need $M \times (2 \times 20 + 4 + 4) = 400{,}000 \times 48 = 19.2$ MB for storing the dictionary in this scheme.

Using fixed-width entries for terms is clearly wasteful. The average length of a dictionary word in English is about 8 characters, so on average we are wasting 12 characters in the fixed-width scheme. Also, we have no way of storing terms with more than 20 characters like hydrochlorofluorocarbons and supercalifragilisticexpialidocious. We can overcome these shortcomings by storing the dictionary terms as one long string of characters as shown in Figure 5.3. The pointer to the next term is also used to demarcate the end of the current term. As before, we locate terms in the data structure by way of binary search in the (now smaller) table. This scheme saves us 60% compared to fixed-width storage – 24 bytes on average of the 40 bytes we allocated for terms before. However, we now also need to store term pointers. The term pointers resolve $400{,}000 \times 8 = 3.2 \times 10^6$ positions, so they need to be



► **Figure 5.4** Blocked storage with four terms per block. The first block consists of systile, syzygetic, syzygial, and syzygy with lengths 7, 9, 8 and 6 characters, respectively. Each term is preceded by a byte encoding its length, indicating how many bytes to skip to reach subsequent terms.

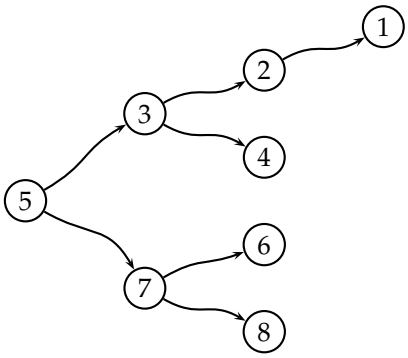
$\log_2$ 3,200,000 $\approx$ 22 bits or 3 bytes long.

In this new scheme, we need 400,000 $\times$ $(4 + 4 + 3 + 2 \times 8)$ $=$ 10.8 MB for the Reuters-RCV1 dictionary: 4 bytes each for frequency and postings pointer, 3 bytes for the term pointer, and 2×8 bytes on average for the term. So we have reduced the space requirements by almost half from 19.2 MB to 10.8 MB.

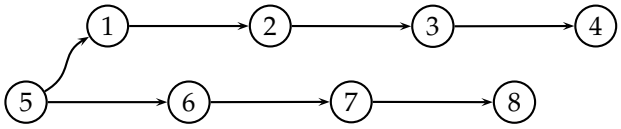
5.2.2 Blocked storage

The dictionary can be further compressed by grouping terms in the string into blocks of size k and keeping a term pointer only for the first term of each block (see Figure 5.4). The length of the term is stored in the string as an additional byte at the beginning of the term. We thus eliminate $k - 1$ term pointers, but need an additional k bytes for storing the length of each term. For $k = 4$, we save $(k - 1) \times 3 = 9$ bytes for term pointers, but need an additional $k = 4$ bytes for term lengths. So the total space requirements for the dictionary of Reuters-RCV1 are reduced by 5 bytes per 4-term block, or a total of 400,000 $\times$ $1/4 \times 5 = 0.5$ MB to 10.3 MB.

By increasing the block size k, we can further compress the dictionary.



► **Figure 5.5** Search of the uncompressed dictionary.



► **Figure 5.6** Search of a dictionary compressed by blocking with $k = 4$.

One block in blocked compression ($k = 4$) ...
8 a u t o m a t a **8** a u t o m a t e **9** a u t o m a t i c **10** a u t o m a t i o n

⇓

... further compressed with front coding.
8 a u t o m a t ∗ a **1** ⋄ e **2** ⋄ i c **3** ⋄ i o n

► **Figure 5.7** Front coding. A sequence of words with identical prefix ("automat" in this example) is encoded by marking the end of the prefix with ∗ and replacing it with ⋄ in subsequent words. As before, the first byte of each entry encodes the number of characters.

However, there is a tradeoff between compression and query processing speed. Searching the uncompressed eight-term dictionary in Figure 5.5 takes on average $(1 + 2 \times 2 + 4 \times 3 + 4)/8 \approx 2.6$ steps (assuming each term is equally likely to come up in a query). With blocks of size $k = 4$, we need $(1 + 2 \times 2 + 2 \times 3 + 2 \times 4 + 5)/8 = 3$ steps on average, 20% more (Figure 5.6). By increasing k, we can get the size of the compressed dictionary arbitrarily close to the minimum of $400{,}000 \times (4 + 4 + 1 + 2 \times 8) = 10$ MB, but query processing becomes prohibitively slow for large values of k.

representation	size in MB
dictionary, fixed-width	19.2
dictionary, term pointers into string	10.8
∼, with blocking	10.3
∼, with blocking & front coding	7.9

► **Table 5.2** Dictionary compression for Reuters-RCV1.

There are even more extreme compression schemes for the dictionary. The observation that long runs of terms in an alphabetically ordered dictionary have a common prefix leads to *front coding* as shown in Figure 5.7. A common prefix is identified in the first term of a block and then referred to with a special character in subsequent entries. In the case of Reuters, front coding saves another 2.4 MB.

FRONT CODING

Other extreme compression schemes rely on minimal perfect hashing, i.e., a hash function that maps M terms onto $[1, \ldots, M]$ without collisions. However, perfect hashes cannot be adapted incrementally and are therefore not very usable in a dynamic environment – each change in the vocabulary would require the creation of a new perfect hash function.

Even with extreme compression, it may not be feasible to store the entire dictionary in main memory for very large text collections. If the dictionary has to be partitioned onto pages that are stored on disk, then a B-tree can be used on the first term of each page. For processing most queries, the search system has to go to disk anyway to fetch the postings. So one additional seek for retrieving the term's dictionary page from disk can be tolerated.

Table 5.2 summarizes the four dictionary representation schemes covered in this section.

5.3 Postings file compression

The size of Reuters-RCV1 is about 800,000 × 200 × 6 bytes = 960 MB (recall that RCV1 has 800,000 documents, 200 tokens per document and 6 characters per token on average, Table 4.1, page 52). Document identifiers for this collection are $\log_2 800{,}000 \approx 20$ bits long. To reduce the size of the postings file, we need to devise methods that use fewer than 20 bits per document.

To devise a more efficient representation, we observe that the postings for frequent terms are close together. Imagine going through the documents of a collection one by one and looking for a frequent term like computer. We will find a document containing computer, then we skip a few documents that don't contain it, then there is again a document with the term and so on (see Table 5.3). The key idea is that the *gaps* between postings are short,

	encoding	postings list							
the	docIDs	...	283042		283043		283044	283045	...
	gaps			1		1	1		...
computer	docIDs	...	283047		283154		283159	283202	...
	gaps			107		5	43		...
arachnocentric	docIDs	252000		500100					
	gaps		248100						

► **Table 5.3** Encoding gaps instead of document ids. Instead of storing the document ids 283047, 283154, 283159 ... for computer, we store the gaps 14, 107, 5, ..., and analogously for arachnocentric and the. The first docID is left unchanged in gap encoding (only shown for arachnocentric). Except for rare terms like arachnocentric, storing gaps is more efficient than storing docIDs.

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

► **Table 5.4** Variable byte (VB) encoding. Gaps are encoded using an integral number of bytes. The first bit, the continuation bit, of each byte indicates whether the code ends with this byte (1) or not (0).

requiring a lot less space than 20 bits to store. In fact, gaps for the most frequent terms such as the and for are mostly equal to 1. But the gaps for a rare term that occurs only once or twice in a collection (e.g., arachnocentric in RCV1, Table 5.3) have the same order of magnitude as the document ids and will need 20 bits. For an economical representation of this distribution of gaps, we need a *variable encoding* method that uses fewer bits for short gaps.

We will achieve this by encoding small numbers in less space than large numbers. We look at two types of methods: bytewise compression and bitwise compression. As the names suggest, these methods attempt to encode gaps with the minimum number of bytes and bits, respectively.

5.3.1 Variable byte codes

VARIABLE BYTE ENCODING

CONTINUATION BIT

Variable byte encoding uses an integral number of bytes to encode a gap. The last 7 bits of a byte are "payload" and encode part of the gap. The first bit of the byte is a *continuation bit*. It is set to 1 for the last byte of the encoded gap and to 0 otherwise. To decode a variable byte code, we read a sequence of bytes with continuation bit 0 terminated by a byte with continuation bit 1. We then extract and concatenate the 7-bit parts. Table 5.4 gives an example

of a variable byte encoding of a postings list.[2]

With variable byte compression, the size of the compressed index for Reuters-RCV1 is 116 MB, a 70% reduction of the size of the uncompressed index (400 MB, Table 5.6).

The same general idea can also be applied to larger or smaller units: 32-bit words, 16-bit words, and 4-bit words or *nibbles*. Larger words further decrease the amount of bit manipulation necessary. In the limit, no bit manipulation whatsoever is needed for the word size of the operating system used. But compression ratios are less favorable for larger units than for bytes. For Reuters-RCV1, "word-aligned" compression amounts to no compression for words larger than $\log_2 800{,}000 \approx 19.6$ bits. Word sizes smaller than bytes get better compression ratios at the cost of more bit manipulation. In general, bytes offer a good tradeoff between compression ratio and speed of decompression.

For most information retrieval systems variable byte encoding offers an excellent tradeoff between time and space. They are also simple to implement – most of the alternatives referred to in Section 5.4 below are considerably more complex. But if disk space is a scarce resource, we can achieve better compression ratios by using bit-level encodings, in particular two closely related encodings: γ codes, the subject of the next section, and δ codes (Exercise 5.7).

5.3.2 γ codes

Variable byte codes use an adaptive number of *bytes* depending on the length of the binary encoding of a gap. Bit-level codes adapt the length of the code on the finer grained *bit* level. The simplest bit-level code is *unary code*. The unary code of n is a string of n 1's followed by a 0 (see the first two columns of Table 5.5). Obviously, this is not a very efficient encoding, but it will come in handy in γ-encoding in a moment.

UNARY CODE

Since we need n bits each for the 2^n gaps G with $2^n \leq G < 2^{n+1}$ (assuming they are all equally likely), an encoding must use at least $\log_2 G$ bits for G. Our goal is to get as close to this optimum as possible.

A method that is within a factor of optimal is γ *encoding*. γ codes implement variable length encoding by splitting the representation of a gap G into a pair of *length* and *offset*. *offset* is G in binary, but with the leading 1 removed.[3] For example, for 13 *offset* is 101. *length* encodes the length of *offset* in unary code. For 13, the length of *offset* is 3 bits, which is 1110 in unary. The γ code of 13 is therefore 1110101.

γ ENCODING

2. Note that the origin is 0 in the table. Since we never need to encode a docid or a gap of 0, in practice the origin is usually 1, so that 10000000 encodes 1, 10000101 encodes 6 (not 5 as in the table) etc. The origin is 0 in the table to keep things simple.
3. We assume here that G has no leading 0s. If there are any, they are removed too.

number	unary code	length	offset	γ code
0	0			
1	10	0		0
2	110	10	0	10,0
3	1110	10	1	10,1
4	11110	110	00	110,00
9	1111111110	1110	001	1110,001
13		1110	101	1110,101
24		11110	1000	11110,1000
511		111111110	11111111	111111110,11111111
1025		11111111110	0000000001	11111111110,0000000001

► **Table 5.5** Some examples of unary and γ codes. Unary codes are only shown for the smaller numbers. Commas in γ codes are for readability only and are not part of the actual codes.

A γ code is decoded by first reading the unary code up to the 0 that terminates it. Now we know how long the offset is. The offset can then be read correctly and the 1 that was chopped off in encoding is prepended. The right hand column of Table 5.5 gives some examples of γ codes.

The length of *offset* is $\lfloor \log_2 G \rfloor$ bits and the length of *length* is $\lfloor \log_2 G \rfloor + 1$ bits, so the length of the entire code is $2 \times \lfloor \log_2 G \rfloor + 1$. γ codes are always of odd length and they are optimal within a factor of 2 of what we claimed to be the optimal encoding length $\log_2 G$ above. We derived this optimum from the assumption that the 2^n gaps between 2^n and 2^{n+1} are equiprobable. But this need not be the case. In general, we don't know the distribution of gaps a priori.

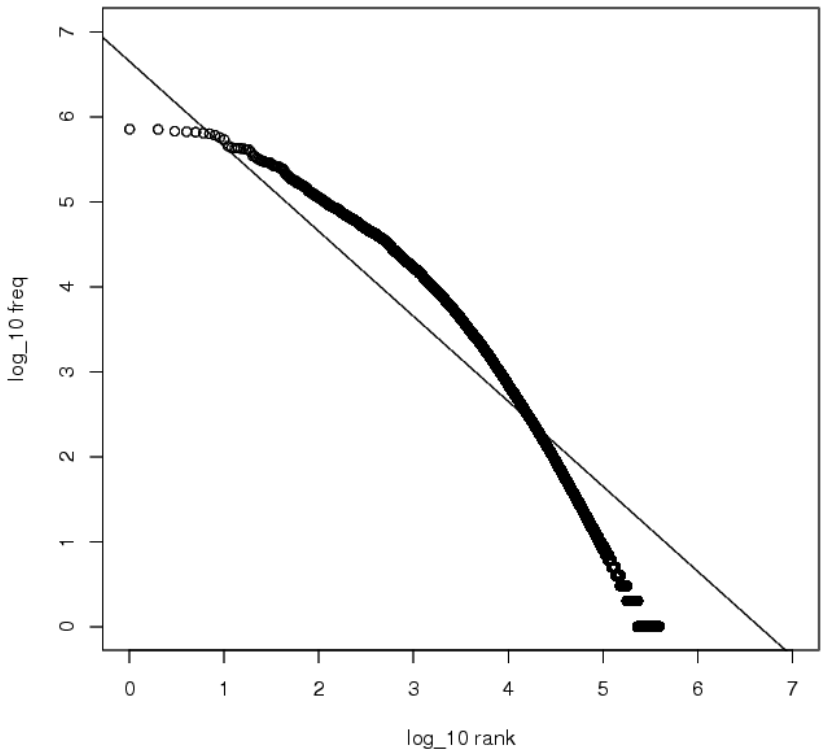
The characteristic of a probability distribution P that determines its coding properties is its entropy $H(P)$. It can be shown that the lower bound for the expected length $E(L)$ of a code L is $H(P)$ if certain conditions hold. It can further be shown that for $1 < H(P) < \infty$, γ encoding is within a factor of 3 of this optimal encoding, approaching 2 for large $H(P)$:

$$\frac{E(L_\gamma)}{H(P)} \leq 2 + \frac{1}{H(P)} \leq 3$$

What is remarkable about this result is that it holds for any probability distribution P. So without knowing anything about the properties of the distribution of gaps, we can apply γ codes and be certain that they are within a factor of ≈ 2 of the optimal code for distributions of large entropy. A code like γ-code with the property of being within a factor of optimal for any distribution P is called *universal*.

UNIVERSAL CODE

In addition to universality, γ codes have two other properties that are use-



► **Figure 5.8** Zipf's law for Reuters-RCV1. Frequency is plotted as a function of frequency rank for the terms in the Reuters-RCV1 collection. The line is the distribution predicted by Zipf's law.

PREFIX-FREE

ful for index compression. First, they are *prefix-free*, i.e., no γ code is the prefix of another. This means that there is always a unique decoding of a sequence of γ-codes (and we don't need delimiters between them, which would decrease the efficiency of the code). The second property is that γ codes are

PARAMETER-FREE

parameter-free. For many other efficient codes, we have to fit a model (e.g., the Bernoulli distribution) to the distribution of gaps in the index. This complicates the implementation of compression and decompression. In decompression, the parameters need to be stored and retrieved. And in dynamic indexing, the distribution of gaps can change, so that the original parameters are no longer appropriate. These problems are avoided with a parameter-free code.

How much compression of the inverted index do γ codes achieve? To answer this question we need a model of the distribution of terms in the

ZIPF'S LAW collection. A commonly used model is *Zipf's law*. It states that, if term 1 is the most common term in the collection, term 2 the next most common etc, then the frequency f_i of the ith most common term is proportional to $1/i$:

$$f_i \propto \frac{1}{i} \tag{5.2}$$

So if the most frequent term occurs f_1 times, then the second most frequent term has half as many occurrences, the third most frequent term a third as many occurrences, etc. The intuition is that frequency decreases very rapidly with rank. Equation 5.2 is one of the simplest ways of formalizing such a rapid decrease and it has been found to be a reasonably good model.

The log-log graph in Figure 5.8 plots the frequency of a term as a function of its rank for Reuters-RCV1. A line with slope -1, corresponding to the prediction made by Zipf's law, is also shown. The fit of the data to the law is not particularly good, but good enough to serve as a model for term distributions in our calculations.

We first derive the term frequency of the ith term from Zipf's law. The frequency f_i is proportional to the inverse of the rank i, that is, there is a constant c such that:

$$f_i = \frac{c}{i} \tag{5.3}$$

We can choose c such that the f_i are relative frequencies and sum to 1:

$$1 = \sum_{i=1}^{M} \frac{c}{i} = c \sum_{i=1}^{M} \frac{1}{i} = c\, H_M \tag{5.4}$$

$$c = \frac{1}{H_M} \tag{5.5}$$

where M is the number of distinct terms and H_M is the Mth *harmonic number* and is $\approx \ln M$.[4] Reuters-RCV1 has $M = 400{,}000$ distinct terms, so we have

$$c = \frac{1}{H_M} \approx \frac{1}{\ln M} = \frac{1}{\ln 400{,}000} \approx \frac{1}{13}$$

Thus the ith term has a frequency of roughly $1/(13i)$, and the average number of occurrences of term i in a document of length L is:

$$L\frac{c}{i} \approx \frac{200 \times \frac{1}{13}}{i} \approx \frac{15}{i}$$

4. Note that, unfortunately, the conventional symbol for both entropy and harmonic number is H. Context should make clear which is meant in this chapter.

	N documents
Lc most frequent terms	
	N gaps of 1 each
Lc next most frequent terms	
	$N/2$ gaps of 2 each
Lc next most frequent terms	
	$N/3$ gaps of 3 each
...	...

▶ **Figure 5.9** Stratification of terms for estimating the size of a γ encoded inverted index.

Now we have derived term statistics that characterize the distribution of terms in the collection and, by extension, the distribution of gaps in the postings lists. From these statistics, we can calculate the space requirements for an inverted index compressed with γ encoding. We first stratify the vocabulary into blocks of size $Lc = 15$. On average, term i occurs $15/i$ times per document. So the average number of occurrences $\overline{f}$ per document is $1 \leq \overline{f}$ for terms in the first block, $\frac{1}{2} \leq \overline{f} < 1$ for terms in the second block, $\frac{1}{3} \leq \overline{f} < \frac{1}{2}$ for terms in the third block etc. We will make the simplifying assumption that all gaps for a given term have the same size as shown in Figure 5.9. Assuming such a uniform distribution of gaps, we then have gaps of size 1 in block 1, gaps of size 2 in block 2, etc.

For the total number of documents N, there are N/j gaps of size j in block j. Encoding the gaps with γ codes, the number of bits needed for the postings list of a term in the jth block (corresponding to one row in the figure) will therefore be:

$$\begin{aligned} \textit{bits-per-row} &= \frac{N}{j} \times (2 \times \lfloor \log_2 j \rfloor + 1) \\ &\approx \frac{2N \log_2 j}{j} \end{aligned}$$

To encode the entire block, we need $(2NLc \log_2 j)/j$ bits. There are $M/(Lc)$ blocks, so the postings file as a whole will take up:

$$\sum_{j=1}^{\frac{M}{Lc}} \frac{2NLc \log_2 j}{j} \tag{5.6}$$

representation	size in MB
dictionary, fixed-width	19.2
dictionary, term pointers into string	10.8
∼, with blocking	10.3
∼, with blocking & front coding	7.9
collection (text, xml markup etc)	3600.0
collection (text)	960.0
term incidence matrix	4000.0
postings, uncompressed (32-bit words)	400.0
postings, uncompressed (20-bit words)	250.0
postings, variable byte encoded	116.0
postings, γ encoded	101.0

► **Table 5.6** Index and dictionary compression for Reuters-RCV1. The compression ratio depends on the proportion of actual text in the collection. Reuters-RCV1 contains a large amount of XML markup. Using the two best compression schemes, γ encoding and blocked storage, the ratio collection size to compress index is therefore especially small for Reuters-RCV1: $(101 + 7.9)/3600 \approx 0.03$

For Reuters-RCV1, $\frac{M}{Lc} \approx 400{,}000/15 \approx 27{,}000$ and

$$\sum_{j=1}^{27{,}000} \frac{2 \times 10^6 \times 15 \log_2 j}{j} \approx 224 \text{ MB} \tag{5.7}$$

So the postings file of the compressed inverted index for our 960 MB collection has a size of 224 MB, one fourth the size of the original collection.

When we run γ compression on Reuters-RCV1, the actual size of the compressed index is even lower: 101 MB, a bit more than a tenth of the size of the collection. The reason for the discrepancy between predicted and actual value is that Zipf's law is not a very good approximation of the actual distribution of term frequencies for Reuters-RCV1. The Zipf model predicts an index size of 251 MB for the unrounded numbers from Table 4.1. If term frequencies are generated from the Zipf model and a compressed index is created for these artificial terms, then the compressed size is 254 MB. So to the extent that the assumptions about the distribution of term frequencies are accurate, the predictions of the model are correct.

Table 5.6 summarizes the compression techniques covered in this chapter. The term incidence matrix (Figure 1.1, page 3) for Reuters-RCV1 consists of $400{,}000 \times 800{,}000 = 40 \times 8 \times 10^9$ bits or 40 GB.

γ codes achieve great compression ratios – about 15% better than variable byte codes for Reuters-RCV1. But they are expensive to decode. This is because many bit-level operations – shifts and masks – are necessary to decode a sequence of γ codes as the boundaries between codes will usually be

somewhere in the middle of a machine word. As a result, query processing is more expensive for γ codes than on an uncompressed index. Whether variable byte or γ encoding is more advantageous for an application depends on the relative weights we give to conserving disk space versus maximizing query response times.

The compression ratio for the index in Table 5.6 is about 25% (400 MB vs. 101 / 116 MB). So the two index compression schemes achieve the primary objective of compression – to reduce the need for disk space. Another advantage is often more important: query processing on a compressed index can in some cases be faster than for an uncompressed index. This is because the joint cost of transferring a compressed postings list and decompressing it can be smaller than transferring a much larger uncompressed postings list. So this is one of the few cases where there is no time/space tradeoff. With a good compression scheme we make optimal use of both time and space.

5.4 References and further reading

Heaps' law was introduced by Heaps (1978). See also Baeza-Yates and Ribeiro-Neto (1999). A detailed study of vocabulary growth in large collections is (Williams and Zobel 2005). Zipf's law is due to Zipf (1949). Other term distribution models, including K mixture and two-poisson model, are discussed by Manning and Schütze (1999), Chapter 15. Carmel et al. (2002) show that lossy compression can achieve good compression without a significant decrease in retrieval performance.

Dictionary compression is covered in detail by Witten et al. (1999), Chapter 4, which is recommended as additional reading.

Subsection 5.3.1 is based on Scholer et al. (2002). They find that variable byte codes process queries twice as fast as both bit-level compressed indexes and uncompressed indexes with a 30% penalty in lost compression ratio compared to the best bit-level compression method. They also show that compressed indexes can be superior to uncompressed indexes not only in disk usage, but also in query processing speed. Compared to variable byte codes, "variable nibble" codes showed 5%–10% better compression and up to a third worse query performance in one experiment (Anh and Moffat 2005). In recent work, Anh and Moffat (2005; 2006a) have constructed word-aligned binary codes that are both faster in decompression and more space efficient than variable byte codes.

δ codes (Exercise 5.7) and γ codes were introduced in (Elias 1975), which proves that both codes are universal. In addition, δ codes are asymptotically optimal for $H(P) \rightarrow \infty$.

Several additional index compression techniques are covered by Witten et al. (1999) (Sections 3.3 and 3.4 and Chapter 5). They recommend using

local compression methods such as Bernoulli for index compression, methods that explicitly model the probability distribution of gaps for each term. They show that their compression ratio is up to 25% better than γ codes for large collections. However, Bernoulli methods require the estimation of parameters, which adds complexity to the implementation and changes over time for dynamic collections. γ encoding achieves good compression and is parameter-free. δ codes perform better than γ codes if large numbers (greater than 15) dominate.

This chapter only looks at index compression for Boolean retrieval. For weighted retrieval (Chapter 6), it is advantageous to order postings according to term frequency instead of docID. During query processing, the scanning of many postings lists can then be terminated early because any smaller weights cannot change the ranking of the highest ranked k documents found so far. Document length is precomputed and stored separately in weighted retrieval. It is not a good idea to precompute and store weights in the index (as opposed to frequencies) because they cannot be compressed as well as integers. See Persin et al. (1996) and Anh and Moffat (2006b) for representing the importance of a term by its term frequency or by an integer *impact*.

IMPACT

Document compression is also important in an efficient information retrieval system. de Moura et al. (2000) and Brisaboa et al. (2007) show how certain compression schemes allow direct searching of words and phrases in the compressed text, which is not possible with standard text compression utilities like gzip and compress.

5.5 Exercises

Exercise 5.1

Estimate the space usage of the Reuters dictionary with blocks of size $k = 8$ and $k = 16$ in blocked dictionary storage.

Exercise 5.2

Estimate the time needed for term lookup in the compressed dictionary of Reuters with block sizes of $k = 4, 8, 16$. What is the slowdown compared to $k = 1$ (uncompressed dictionary)?

Exercise 5.3

Is binary search really a good idea for searching the dictionary? What are the alternatives?

Exercise 5.4

Compute variable byte codes for the numbers in Tables 5.3 and 5.5.

Exercise 5.5

Compute variable byte and γ codes for the postings list 777, 17743, 294068, 31251336. Use gaps instead of docIDs where possible.

Exercise 5.6

From the following sequence of γ coded gaps, reconstruct first the gap sequence and then the postings sequence.
1110001110101011111101101111011

Exercise 5.7

δ-CODES

γ-codes are relatively inefficient for large numbers (e.g., 1025 in Table 5.5) as they encode the length of the offset in inefficient unary code. *δ-codes* differ from γ-codes in that they encode the first part of the code (*length*) in γ-code instead of in unary code. The encoding of *offset* is the same. For example, the δ-code of 7 is 10,0,11 (we add commas for readability). 10,0 is the γ-code for *length* (2 bits) and the encoding of *offset* (11) is unchanged. Compute the δ-codes for the other numbers in Table 5.5. For what range of numbers is the δ-code shorter than the γ-code?

Exercise 5.8

We have defined unary codes as being "10": sequences of 1s terminated by a 0. Interchanging the roles of 0s and 1s yields an equivalent "01" unary code. When this 01 unary code is used, the construction of a γ-code can be stated as follows: 1. Write G down in binary using $b = \lfloor \log_2 j \rfloor + 1$ bits. 2. Prepend $(b - 1)$ 0s. Show that this method produces a well-defined alternative γ-code.

Exercise 5.9

Unary code is not a universal code in the sense defined above. However, there exists a distribution over code words for which unary code is optimal. Which distribution is this?

Exercise 5.10

Give some examples of terms that violate the assumption that gaps all have the same size (which we made when estimating the space requirements of a γ encoded index). What are general characteristics of these terms?

Exercise 5.11

If a term's distribution is not uniform and its gaps are of variable size, will that increase or decrease the size of the γ-compressed postings list?

Exercise 5.12

Work out the sum in Equation 5.7 and show it adds up to about 251 MB. Use the numbers in Table 4.1, but don't round Lc, c and the number of vocabulary blocks.

Exercise 5.13

Go through the above calculation of index size and explicitly state all the approximations that were made to arrive at Expression 5.6.

Exercise 5.14

For a collection of your choosing determine the number of documents and terms and the average length of a document. (i) How large is the inverted index predicted to be by Equation 5.6? Implement an indexer that creates a γ-compressed inverted index for the collection. How large is the actual index? (ii) Implement an indexer that uses variable byte encoding. How large is the variable byte encoded index?

γ encoded gap sequence of run 1	1110110111111001011111111110100011111001
γ encoded gap sequence of run 2	1111101000011111100010001111111001000001111101010 1

► **Table 5.7** Two gap sequences to be merged in block merge indexing.

Exercise 5.15

Adapt the space analysis in Equation 5.6 to a positional index compressed with γ codes.

Exercise 5.16

To be able to hold as many postings as possible in main memory, it is a good idea to compress intermediate index files during index construction. This makes merging runs in block merge indexing more complicated. As an example, work out the γ encoded merged sequence of the gaps in Table 5.7.

Exercise 5.17

Show that the size of the vocabulary is finite according to Zipf's law and infinite according to Heaps' law. What implications does this result have for trying to derive Heaps' law from Zipf's law?

6 *Scoring and term weighting*

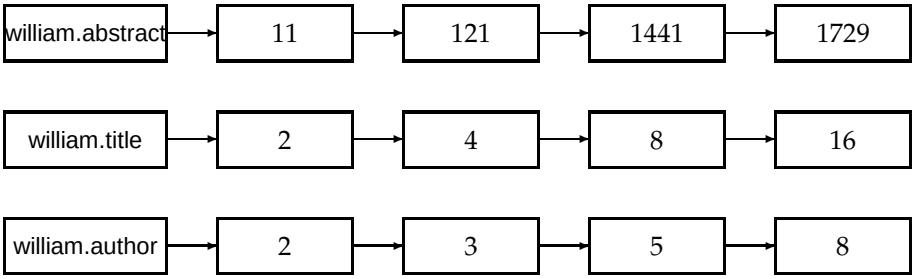
Thus far we have dealt with indexes that support Boolean queries: a document either matches or does not match a query. The resulting number of matching documents can, in the case of large document collections, be far in excess of the number a human user could possibly sift through. Accordingly, it is essential for search engines to rank-order the documents matching a query. To do this, an engine computes, for each matching document, a score with respect to the query at hand. In this chapter we initiate the study of assigning a score to a (query, document) pair. Before doing this, we first study parametric and zone indexes, which extend the applicability of inverted indexes to scoring. We then consider the problem of gauging the importance of a term in a document, for the purposes of scoring the document on queries that include the term.

6.1 Parametric and zone indexes

We have thus far viewed a document as a sequence of terms. In fact, most documents in the real world have additional structure. Digital documents generally encode, in machine-recognizable form, certain *meta-data* associated with each document. This meta-data would generally include *fields* such as the date of creation and the format of the document, and often the author and possibly the title of the document. Consider queries of the form "find documents authored by William Shakespeare in 1601, containing the phrase alas poor Yorick". To support such queries we must build, in addition to the text inverted indexes already discussed, *parametric indexes*: inverted indexes on each kind of meta-data (author, date and so on). For each field, we build an inverted index whose dictionary consists of all distinct values occurring in that field, and postings point to documents with that field value. For instance, for the author field, the dictionary consists of all authors of documents in the collection; the postings list for a particular author consist of all documents with that author. Query processing then consists as usual of

FIELD

PARAMETRIC INDEX



► **Figure 6.1** Basic zone index ; zones are encoded as extensions of dictionary entries.

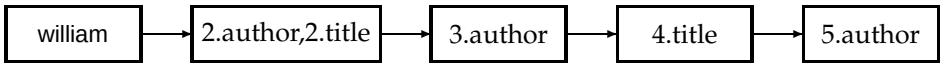
postings merges, except that we may merge postings from text as well as parametric indexes. Where the field values can be ordered (as in the case of dates, for instance) we additionally build a search tree on the ordered universe of values to support range queries. Figure 6.1 illustrates this. Some of the fields may assume ordered values, such as dates; in the example query above, the year 1601 is one such field value. The engine may support querying ranges on such ordered values; to this end, a structure like a B-tree may be used on the field's dictionary.

ZONE

Zones are similar to fields, except the contents of a zone can be an arbitrary body of text. Whereas a field may assume a relatively small set of values, a zone can be thought of as an unbounded amount of text. For instance, document titles and abstracts are generally treated as zones. We may build a separate inverted index for each zone of a document, to support queries such as "find documents with merchant in the title and william in the author list and the phrase gentle rain in the body". This has the effect of building an index that looks like Figure 6.1.

In fact, we can reduce the size of the dictionary by encoding the zone in which a term occurs in the postings. In Figure 6.2 for instance, we show how occurrences of bill in the title and author zones of various documents are encoded. Such an encoding is useful when the size of the dictionary is a concern (because we require the dictionary to fit in main memory). But there is another important reason why the encoding of Figure 6.2 is useful: the efficient computation of scores using a technique we will call *weighted zone scoring*.

WEIGHTED ZONE SCORING



► **Figure 6.2** Zone index in which the zone is encoded in the postings rather than the dictionary.

6.1.1 Weighted zone scoring

Given a Boolean query q and a document d, weighted zone scoring assigns to the pair (q, d) a score in the interval $[0, 1]$, by computing a linear combination of *zone scores*, where each zone of the document contributes a Boolean value. More specifically, consider a document with ℓ zones. Let $w_1, \ldots, w_\ell \in [0, 1]$ such that $\sum_{i=1}^{\ell} w_i = 1$. For $1 \leq i \leq \ell$, let s_i be the Boolean score denoting a match (or absence thereof) between q and the ith zone. For instance, the Boolean score from the title zone could be defined to be 1 if all the query term(s) occur in the title, and zero otherwise. Alternatively, the Boolean score could be defined to be 1 if *any* of the query terms occurs in the title. Then, the weighted zone score is defined to be $\sum_{i=1}^{\ell} w_i s_i$.

Example 6.1: Consider just the query shakespeare in a collection in which each document has three distinct zones: *author, title* and *body*. Weighted zone scoring in such a collection would demand three weights w_1, w_2 and w_3, respectively corresponding to the *author, title* and *body* zones. Suppose we set $w_1 = 0.2, w_2 = 0.3$ and $w_3 = 0.5$ (so that the three weights add up to 1); this corresponds to an application in which a match in the *author* zone is least important to the overall score, the *title* zone somewhat more, and the *body* contributes as much as either *author* and *title*.

Thus if the term shakespeare were to appear in the *title* and *body* zones of a document, the score of this document would be 0.7.

Exercise 6.1

In the above worked example with weights $w_1 = 0.2, w_2 = 0.31$ and $w_3 = 0.49$, what are all the distinct score values a document may get?

6.2 Term frequency and weighting

Thus far, scoring has hinged on whether or not a query term is present in a document (or a zone within a document). We take the next logical step: a document that mentions a query term more often has more to do with that

FREE TEXT QUERY

query. To motivate this we introduce the notion of a *free text query*: a query in which the terms of the query are typed freeform into the search interface, without any connecting search operators (such as Boolean operators). This query style, which is extremely popular on the web, views the query as simply a set of terms. A plausible scoring mechanism then is to compute a score that is the sum, over the query terms, of the match scores between each term and the document. How do we determine such a match scores between a term and each document?

To this end, we assign to each term in a document a *weight* for that term in that document, that depends on the number of occurrences of the term in the document. The simplest approach is to assign the weight to be equal to the number of occurrences of the term t in document d. This weighting scheme is

TERM FREQUENCY

referred to as *term frequency* and is denoted $\text{tf}_{t,d}$, with the subscripts denoting the term and the document in order.

For a document d, the set of weights (determined by the tf weighting function above, or indeed any weighting function that maps the number of occurrences of t in d to a positive real value) may be viewed as a vector, with one component for each distinct term. In this view of a document, known in the

BAG OF WORDS

literature as the *bag of words model*, the exact ordering of the terms in a document is ignored. The vector view only retains information on the number of occurrences. Thus, the document "Mary is quicker than John" is, in this view, identical to the document "John is quicker than Mary". Nevertheless, it seems intuitive that two documents with similar vector representations are similar in content. We will develop this intuition further below in Chapter 7. Before doing so we first study the question: are all words in a document equally important? Clearly not; in Chapter 5 (page 23) we looked at the idea of *stop words* — words that we decide not to index at all.

6.2.1 Inverse document frequency

Raw term frequency as above suffers from a critical problem: all terms are considered equally important when it comes to assessing relevancy on a query. In fact, as discussed in Chapter 5, certain terms have little or no discriminating power in determining relevance. For instance, a collection of documents on the insurance industry is likely to have the term insurance in almost every document. To this end, we introduce a mechanism for attenuating the effect of terms that occur too often in the collection to be meaningful for relevance determination. An immediate idea is to scale down the term weights of terms with high *collection frequency*, defined to be the total number of occurrences of a term in the corpus. The idea would be to reduce the tf weight of a term by a factor that grew with its collection frequency.

Instead, it is more commonplace to use for this purpose the *document frequency* df_t, defined to be the number of documents in the corpus that contain

Word	cf	df
ferrari	10422	17
insurance	10440	3997

► **Figure 6.3** Collection frequency (cf) and document frequency (df) behave differently.

term	df_t	idf_t
calpurnia	1	6
animal	100	4
sunday	1000	3
fly	10,000	2
under	100,000	1
the	1,000,000	0

► **Figure 6.4** Example of idf values. Here we give the idf's of terms with various frequencies in a corpus of 1,000,000 documents.

a term t. The reason to prefer df to cf is illustrated in Figure 6.3, where a simple example shows that collection frequency (cf) and document frequency (df) can behave rather differently. In particular, the cf values for both ferrari and insurance are roughly equal, but their df values differ significantly. This suggests that the few documents that do contain ferrari mention this term frequently, so that its cf is high but the df is not. Intuitively, we want such terms to be treated differently: the few documents that contain ferrari should get a significantly higher boost for a query on ferrari than the many documents containing insurance get from a query on insurance.

How is the document frequency df of a term used to scale its weight? Denoting as usual the total number of documents in a corpus by N, we define the *inverse document frequency* (idf) of a term t as follows:

INVERSE DOCUMENT FREQUENCY

$$\text{idf}_t = \log \frac{N}{\text{df}_t}. \tag{6.1}$$

Thus the idf of a rare term is high, whereas the idf of a frequent term is likely to be low. Figure 6.4 gives an example of idf's in a corpus of 1,000,000 documents; in this example logarithms are to the base 10.

Exercise 6.2

Why is the idf of a term always finite?

Exercise 6.3

What is the idf of a term that occurs in every document? Compare this with the use of stop word lists.

6.2.2 tf-idf weighting

We now combine the above expressions for term frequency and inverse document frequency, to produce a composite weight for each term in each document. The *tf-idf* weighting scheme assigns to term t a weight in document d given by

$$\text{tf-idf}_{t,d} = \text{tf}_{t,d} \times \text{idf}_t. \tag{6.2}$$

In other words, $\text{tf-idf}_{t,d}$ assigns to term t a weight in document d that is

1. highest when t occurs many times within a small number of documents (thus lending high discriminating power to those documents);
2. lower when the term occurs fewer times in a document, or occurs in many documents (thus offering a less pronounced relevance signal);
3. lowest when the term occurs in virtually all documents.

At this point, we may view each document as a *vector* with one component corresponding to each term, together with a weight for each component that is given by (6.2). This vector form will prove to be crucial to scoring and ranking; we will develop these ideas in Chapter 7. As a first step, we introduce the *overlap score measure*: the score of a document d is the sum, over all query terms, of the number of times each of the query terms occurs in d. We can refine this idea so that we add up not the number of occurrences of each query term t in d, but instead the tf-idf weight of each term in d.

$$\text{Score}(q, d) = \sum_{t \in q} \text{tf-idf}_{t,d}. \tag{6.3}$$

Exercise 6.4

Can the tf-idf weight of a term in a document exceed 1?

Exercise 6.5

How does the base of the logarithm in (6.1) affect the score calculation in (6.3)? How does the base of the logarithm affect the relative scores of two documents on a given query?

Exercise 6.6

If the logarithm in (6.1) is computed base 2, suggest a simple approximation to the idf of a term.

6.3 Variants in weighting functions

A number of alternative schemes to tf and tf-idf have been considered; we discuss some of the principal ones here.

6.3.1 Sublinear tf scaling

Do twenty occurrences of a term truly carry twenty times the significance of a single occurrence? Accordingly, there has been considerable research into weight functions other than the number of occurrences of a term. A common weighting function is the logarithmic function, which assigns a weight given by

$$\text{wf}_{t,d} = \begin{cases} 1 + \log \text{tf}_{t,d} & \text{if } \text{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}. \tag{6.4}$$

In this form, we may replace tf by some other function wf as in (6.4), to obtain:

$$\text{wf-idf}_{t,d} = \text{wf}_{t,d} \times \text{idf}_t. \tag{6.5}$$

Equation (6.3) can then be modified by replacing tf-idf by wf-idf as defined in (6.5). Indeed, we may choose to use any weighting function we want; below we discuss several choices.

6.3.2 Maximum tf normalization

One well-studied technique is to normalize the individual tf weights for a document by the maximum tf in that document. For each document d, let $\text{tf}_{\max}(d) = \max_\tau \text{tf}_{\tau,d}$, where τ ranges over all terms in d. Then, we compute a normalized term frequency for each term t in document d by

$$\text{ntf}_{t,d} = a + (1-a)\frac{\text{tf}_{t,d}}{\text{tf}_{\max}(d)}, \tag{6.6}$$

where a is a value between 0 and 1 and is generally set to 0.5. The term a in (6.6) is a *smoothing* term whose role is to damp the contribution of the second term – which may be viewed as a scaling down of tf by the largest tf value in the dictionary. The main idea of maximum tf normalization is to mitigate the following anomaly: we observe higher term frequencies in longer documents, merely because longer documents tend to repeat the same words over and over again. To appreciate this, consider the following extreme example: supposed we were to take a document d and create a new document d' by simply appending a copy of d to itself. While d' should be no more relevant to any query than d is, the use of (6.3) would assign it twice as high a score as d. Replacing $\text{tf-idf}_{t,d}$ in (6.3) by $\text{ntf-idf}_{t,d}$ eliminates the anomaly in this example. Maximum tf nomalization does suffer from a couple of issues:

1. A document may contain an outlier term with an unusually large number of occurrences of that term, not representative of the content of that document.

2. More generally, a document in which the most frequent term appears roughly as often as many other terms should be treated differently from one with a more skewed distribution.

To understand these effects, we examine the effect of document length on relevance. We will give a more nuanced treatment of compensating for document length in Section 7.1.3.

6.3.3 The effect of document length

The above discussion of weighting ignores the length of documents in computing term weights. However, document lengths are material to these weights, for several reasons. First, longer documents will – as a result of containing more terms – have higher tf values. Second, longer documents contain more distinct terms. These factors conspire to raise the scores of longer documents, which (at least for some information needs) is unnatural. Longer terms can broadly be lumped into two categories: (1) *verbose* documents that essentially repeat the same content – in these, the length of the document does not alter the relative weights of different terms; (2) documents covering multiple different topics, in which the search terms probably match small segments of the document but not all of it – in this case, the relative weights of terms are quite different from a single short document that matches the query terms.

6.3.4 Learning weight functions

The tf-idf scheme for determining weights of terms in documents was the result of early research built on the intuition in Section 6.2.1 above. Separately, we have the problem of determining weights for terms occurring in the different zones of a document: does a term occurring in the title of a document count as much as a term in boldface in the body, twice as much, or half as much? In fact, we have an even more general problem here: computing scores for documents that contain not only zones but also rich formatting features (text in boldface, emphasized text, etc.) and even richer meta-data as we will see in the context of the Web (Chapter 19). The terms contained in formatted blocks, as well as other document statistics, are indicators of the relevance of the document. This is true even of document features that have little to do with the query terms. For instance, in a collection of answers to frequently asked questions (FAQs) that are generated by a community of software users, answers with many exclamation marks or the excessive use of capitalized letters may consistently turn out to be of low quality. In this setting, a plausible definition of the overall score of a document could be

$$\sum_{t \in q} \text{tf-idf}_{t,d} - c_1 n_!$$

where $n_!$ is the number of exclamation marks in a document c_1 is a well-chosen constant. In designing a formula for the overall score for a document, how do we determine weights (such as the constant c_1) for these different features?

Increasingly, the assignment of such weights uses the following general approach:

- Given a corpus of documents, the engine is provided with a set of *training queries*, together with the documents deemed to be the correct results for these training queries. The methodology for assembling such a collection will be covered in more detail in Chapter 8.

- The relative weights of terms (and more generally, any features of a document that we choose to introduce into this process) are then "learned" from these samples, in order that the final scores and documents approximate the given training data. To this end frequency data such as tf and idf, as well as weights for zones, are viewed as features in a machine learning problem, in a manner similar to the discussion of classification in Chapter 13.

The particular form of the learned function depends on the learning algorithm used – it could be a decision tree, a linear function of the document features, or any other representation that a learning algorithm constructs. We start by identifying the set of *ranking features*, such as the vector space score for each document matching the query as an exact phrase, or scores from matching sub-phrases. We have a class of *functional forms* that define the manner in which a composite score for each document can be derived from these features. The learning problem is to select the best member of the class of functional forms.

Example 6.2: The set of features in a specific instance could include vector space scores as above, as well as the number of query terms present in the document. We may seek a form for the score formula that raises the score significantly if all or more query terms are present in the document. The set of functional forms could be all linear combinations of the feature variables. Thus we seek to find the best linear combination of the feature variables, producing a composite score for each document.

Other classes of functional forms could include decision trees, or indeed any class of functional forms from which we can efficiently pick a "good" function. How do we determine which linear combination (or more generally, which of a class of functional forms) is good? Here we resort to supervised machine learning: given example data (test queries, together with matching documents for which we have user-generated ideal rankings), we

"learn" a function from the class of functional forms that does a good job of approximating the user-generated preferences.

The expensive component of this methodology is the availability of user-generated ideal rankings, especially in a corpus that changes frequently (such as the Web). Thus the class of functional forms must be rich enough to parsimoniously learn from sparse user-generated rankings, while simple enough to be computationally tractable.

6.3.5 Query-term proximity

Especially for free-text queries on the web (Chapter 19), users expect a document containing most or all of the query terms to be ranked higher than one containing fewer query terms. Consider a query with two or more query terms, $t_1, t_2, \ldots, t_k$. Let ω be the width of the smallest window in a document d that contains all the query terms, measured in the number of words in the window. For instance, if the document were to simply consist of the sentence The quality of mercy is not strained., the smallest window for the query strained mercy would be 4. In cases where the document does not contain all the query terms, we can set ω to be some enormous number. Intuitively, the smaller that ω is, the better that d matches the query. One could consider variants in which only non-stopwords are considered in computing ω. How can we design the scoring function to depend on ω? The answer goes back to Section 6.3.4 – we treat the integer ω as yet another feature in the scoring function, whose importance is assigned by the methodology of Section 6.3.4.

Exercise 6.7

Explain how the postings merge first introduced in Section 1.3 can be adapted to find the smallest integer ω that contains all query terms. How does this procedure work when not all query terms are present in a document?

References and further reading

Luhn Luhn (1957; 1958) describes some of the earliest reported applications of term weighting. His paper dwells on the importance of medium frequency terms (terms that are neither too commonplace nor too rare) and may be thought of as anticipating tf-idf and related weighting schemes. Spärck Jones Spärck Jones (1972) builds on this intuition through detailed experiments showing the use of inverse document frequency in term weighting. A series of extensions and theoretical justifications of idf are due to Salton and Buckley (1996) Robertson and Spärck Jones (1976a), Robertson and Spärck Jones (1976b), Croft and Harper (1979) and Papineni (2001). Robertson maintains a web page (http://www.soi.city.ac.uk/ ser/idf.html) containing the history of

idf, including soft copies of early papers that predated electronic versions of journal article.

We observed that by assigning a weight for each term in a document, a document may be viewed as a vector of term weights, one for each term in the collection. The SMART information retrieval system at Cornell Salton (1971) due to Salton and colleagues was perhaps the first to view a document as a vector of weights. We will develop this view further in Chapter 7.

Pioneering work on learning of ranking functions was done by Fuhr (1989), Fuhr and Pfeifer (1994), Cooper et al. (1994) and by Cohen et al. (1998). Clarke et al. (2000) and Song et al. (2005) treat the use of query term proximity in assessing relevance.

7 *Vector space retrieval*

7.1 Documents as vectors

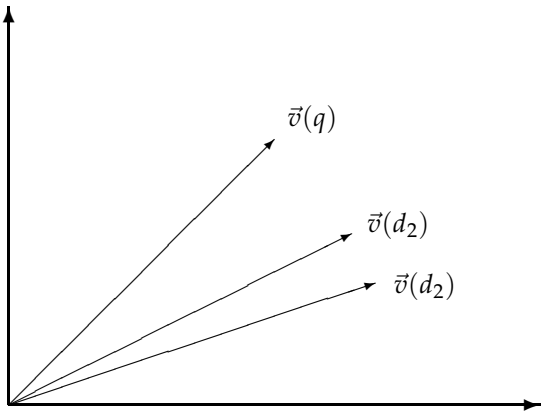
In Chapter 6 we saw how to represent each document as a vector, with a component for each term in the dictionary. The relative weights in the various components are a signature of the topics that are discussed in that document. Intuitively, two documents with similar vector representations discuss the same topics. In this chapter we develop this intuition into computational methods that exploit it for document retrieval. This representation of a set of documents as vectors in a common vector space is known as the *vector space model* and is fundamental to a host of information retrieval operations ranging from scoring documents on a query, document classification and document clustering. Below we first develop the basic ideas underlying vector space retrieval; a pivotal step in this development is the view (Section 7.1.2 below) of queries as vectors in the same vector space as the document collection. Following this we outline techniques for accelerating vector space retrieval; we end this chapter by placing vector space retrieval in perspective to other retrieval mechanisms described earlier (such as Boolean retrieval).

VECTOR SPACE MODEL

7.1.1 Inner products

We denote by $\vec{V}(d)$ the vector derived from document d. Generally, we will assume that the components are computed using the tf-idf weighting scheme of Chapter 6, although this particular weighting scheme is immaterial to the discussion that follows. The set of documents in a collection then turns into a vector space, with one axis for each term. This representation loses the relative ordering of the terms in each document; recall our example from Chapter 6, where we pointed out that the documents "Mary is quicker than John" and "John is quicker than Mary" are identical in such a *bag of words* representation.

How do we quantify the similarity between two documents in this vector space? A first attempt might consider the magnitude of the *vector difference*



► **Figure 7.1** Cosine similarity illustrated.

between two document vectors. This measure suffers from a drawback: two documents with very similar term distributions can have a significant vector difference simply because one is much longer than the other. Thus the relative distributions of terms may be identical in the two documents, but the absolute term frequencies of one may be far larger.

To compensate for the effect of document length, the standard way of quantifying the similarity between two documents d_1 and d_2 is to compute the *cosine similarity* of their vector representations $\vec{V}(d_1)$ and $\vec{V}(d_2)$

$$\text{sim}(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)||\vec{V}(d_2)|}, \tag{7.1}$$

where the numerator represents the *inner product* (also known as the dot product) of the vectors $\vec{V}(d_1)$ and $\vec{V}(d_2)$, while the denominator is the products of their lengths. The effect of the denominator is to *normalize* the vectors $\vec{V}(d_1)$ and $\vec{V}(d_2)$ to unit vectors $\vec{v}(d_1) = \vec{V}(d_1)/|\vec{V}(d_1)|$ and $\vec{v}(d_2) = \vec{V}(d_2)/|\vec{V}(d_2)|$. We can then rewrite (7.1) as

$$\text{sim}(d_1, d_2) = \vec{v}(d_1) \cdot \vec{v}(d_2). \tag{7.2}$$

Thus, (7.2) can be viewed as the inner product of the normalized versions of the two document vectors. What use is the similarity measure $\text{sim}(d_1, d_2)$? Consider again the set of N normalized document vectors $\vec{v}(d_1), \ldots, \vec{v}(d_N)$ representing the documents in a collection. Given a document d (potentially one of the d_i in the collection), consider searching for the documents in the collection most similar to d. Such a search is useful in a system where a user may identify a document and seek others like it – a feature often available

term	SaS	PaP	WH
affection	115	58	20
jealous	10	7	11
gossip	2	0	6

► **Figure 7.2** Term frequencies in three novels. The novels are Austen's *Sense and Sensibility, Pride and Prejudice* and Bronte's *Wuthering Heights*.

term	SaS	PaP	WH
affection	0.996	0.993	0.847
jealous	0.087	0.120	0.466
gossip	0.017	0	0.254

► **Figure 7.3** Term vectors for the three novels of Figure 7.2. These are based on raw term frequency only and are normalized as if these were the only terms in the collection. (Since affection and jealous occur in all three documents, their tf-idf weight would be 0 in most formulations.)

in the results lists of search engines as a *more like this* feature. The problem of finding the document(s) most similar to d then becomes one of finding the d_i with the highest inner products (sim values) $\vec{v}(d) \cdot \vec{v}(d_i)$. We could certainly do this by computing the inner products between $\vec{v}(d)$ and each of $\vec{v}(d_1), \ldots, \vec{v}(d_N)$, then picking off the highest resulting sim values.

Figure 7.2 shows the number of occurrences of three terms (affection, jealous and gossip) in each of the following three novels: Jane Austen's *Sense and Sensibility* (SaS), *Pride and Prejudice* (PaP) and Emily Brontë's *Wuthering Heights* (WH). Of course, there are many other terms occurring in each of these novels. Consider representing each of these novels as a unit vector in three dimensions, corresponding to these three terms; we use raw term frequencies for the purposes of this example, with no idf multiplier. The resulting weights are as shown in Figure 7.3.

Now consider the cosine similarities between pairs of the resulting three-dimensional vectors. A simple computation shows that sim($\vec{v}$(SAS), $\vec{v}$(PAP)) is 0.999, whereas sim($\vec{v}$(SAS), $\vec{v}$(WH)) is 0.888; thus, the two books authored by Austen (SaS and PaP) are considerably closer to each other than to Brontë's WH. In fact, the similarity between the first two is almost perfect (at least when restricted to the three terms we consider).

TERM-DOCUMENT MATRIX

A collection of N documents can thus be viewed as a collection of vectors, leading to a natural view of a collection as a *term-document matrix*: this is an $M \times N$ matrix whose rows represent the M terms (dimensions) of the N columns, each of which corresponds to a document. As always, the terms being indexed could be stemmed before indexing; for instance, jealous and

jealousy would under stemming be considered as a single dimension.

7.1.2 Queries as vectors

While the above approach to finding documents similar to a given document is intriguing, there is a far more compelling reason to represent documents as vectors. The idea is to view a *query* as a vector. Consider the query $q =$ jealous gossip. This query turns into the unit vector $\vec{v}(q) = (0, 0.707, 0.707)$ on the three coordinates of Figures 7.2 and 7.3. The key idea now: to assign to each document d a score equal to the inner product

$$\vec{v}(q) \cdot \vec{v}(d).$$

In the example of Figure 7.3, *Wuthering Heights* is the top-scoring document for this query with a score of 0.509, with *Pride and Prejudice* in second place with a score of 0.085, and *Sense and Sensibility* close behind with a score of 0.074.

The number of dimensions in general will be far larger than three: it will equal the number M of distinct terms being indexed.

To summarize, by viewing a query as a "bag of words", we are able to treat it as a very short document. As a consequence, we can use the cosine similarity between the query vector and a document vector as a measure bof the score of the document for that query. The scores can then be used to select the top-scoring documents for a query.

The most obvious way to implement this: compute the cosine similarities between the query vector and each document vector in the collection. Sort the resulting scores and select the top K documents, where K is an application-specific constant (for instance, an application may require the 10 highest-scoring documents for each query). This appears at first glance to be expensive — a single similarity computation can entail an inner product in tens of thousands of dimensions, demanding tens of thousands of arithmetic operations. We now study how to use an inverted index for this purpose, followed by a series of heuristics for improving on this.

Exercise 7.1

One measure of similarity of two unit vectors is the *Euclidean distance* between them. Given a query q and documents $d_1, d_2, \ldots$, we may rank the documents d_i in order of increasing Euclidean distance from q. Show that if q and the d_i are all normalized to unit vectors, then the rank ordering produced by Euclidean distance is identical to that produced by cosine similarities.

Exercise 7.2

Show that for the query affection, the ordering of the scores of the three documents in Figure 7.3 is the reverse of the ordering of the scores for the query jealous gossip.

Exercise 7.3

In turning a query into a unit vector in the simple above example, we assigned equal weights to each of the query terms. What other principled approaches are plausible?

Exercise 7.4

Consider the case of a query term that is not in the set of M indexed terms. How would one adapt the vector space representation to handle this case?

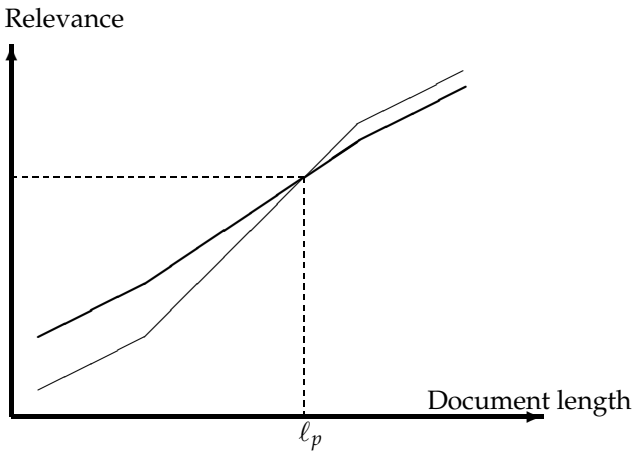
7.1.3 Pivoted normalized document length

In Section 7.1.1 we normalized each document vectors by the Euclidean length of the vector, so that all document vectors turned into unit vectors. In doing so, we eliminate all information on the length of the original document. However, in Section 6.3.3 we argued that the length of the a document (independent of its content) can directly affect the relevance of a document. Compensating for this phenomenon is a form of document length normalization that is independent of term and document frequencies. To this end, we introduce a form of normalizing the vector representation of a document, so that the resulting "normalized" documents are not necessarily of unit length. Then, when we compute the inner product score between a (unit) query vector and such a normalized document, the score is skewed to account for the effect of document length on relevance. This form of compensation for document length is known as *pivoted document length normalization*.

PIVOTED DOCUMENT LENGTH NORMALIZATION

Consider a document collection together with an ensemble of queries for that collection. Suppose that, for each query q and for each document d, were given a Boolean judgment of whether of whether or not d is relevant to the query q; in Chapter 8 we will see how to procure such a set of relevance judgments for a query ensemble and a document collection. Given this set of relevance judgments, we may compute a *probability of relevance* as a function of document length, averaged over all queries in the ensemble. The resulting function may look like the curve drawn in thin lines in Figure 7.4. To compute this curve, we bucket documents by length and compute the fraction of relevant documents in each bucket, then plot this fraction against the median document length of each bucket.

On the other hand, the curve in thick lines shows what might happen with the same documents and query ensemble if we were to use relevance as prescribed by cosine normalization (7.1) – thus, cosine normalization has a tendency to exaggerate the computed relevance vis-à-vis the true relevance, at the expense of longer documents. The thin and thick curves crossover at document length ℓ_p, which we refer to as the *pivot length*; dashed lines mark this point on the $x-$ and $y-$ axes. The idea of pivoted document length normalization would then be to "rotate" the cosine normalization curve so that it more closely matches the relevance vs. document length curve. As



► **Figure 7.4** Pivoted document length normalization - figure to be completed.

mentioned at the beginning of this section, we do so by using in (7.1) a normalization factor for each document vector that is not the Euclidean length of that vector, but instead one that is larger than the Euclidean length for documents of length less than ℓ_p, and smaller for longer documents.

7.2 Heuristics for efficient scoring and ranking

Consider again the query $q =$ jealous gossip. Two observations are immediate:

1. The unit vector $\vec{v}(q)$ has only two non-zero components.
2. These non-zero components are equal – in this case, both equal 0.707. Indeed, such equality holds for any query in which no term is repeated.

Next, we observe that we really are interested in the relative (rather than absolute) scores of the documents in the collection. To this end, it suffices to compute the cosine similarity from each document unit vector $\vec{v}(d)$ to $\vec{V}(q)$ (in which all non-zero components of the query vector are set to 1), rather than to the unit vector $\vec{v}(q)$: for any two documents d_1, d_2

$$\vec{V}(q) \cdot \vec{v}(d_1) > \vec{V}(q) \cdot \vec{v}(d_2) \Leftrightarrow \vec{v}(q) \cdot \vec{v}(d_1) > \vec{v}(q) \cdot \vec{v}(d_2). \tag{7.3}$$

For any document d, the cosine similarity $\vec{V}(q) \cdot \vec{v}(d)$ is the sum, over all terms in the query q, of the weights of those terms in d. This in turn can

be computed by a "postings merge": we walk through the postings in the inverted index for the terms in q, accumulating the total score for each document – very much as in processing a Boolean query, except we assign a positive score to each document that appears in any of the postings being traversed.

Given these scores (some of which could exceed 1), the final step before presenting results to a user is to pick out the K highest-scoring documents. While one could sort the complete set of scores, a better approach is to use a heap to retrieve only the top K documents in order. Where J is the number of documents with non-zero cosine scores, constructing such a heap can be performed in $2J$ comparison steps, following which each of the K highest scoring documents can be "read off" the heap with $\log J$ comparison steps.

7.2.1 Inexact top K document retrieval

Thus far, we have considered schemes by which the K documents that we produce for a query q are in fact exactly the K highest-scoring documents for that query. We now consider schemes by which we produce K documents that are *likely* to be among the K highest scoring documents for a query. In doing so, we hope to dramatically lower the cost of computing the K documents we output.

The principal cost in computing the output stems from computing cosine similarities between the query and a large number of documents. Having a large number of documents in contention also increases the selection cost in the final stage of culling the top K documents. Accordingly, we next consider a series of schemes designed to eliminate from consideration a large number of documents.

Before considering specific schemes, we note that inexact top-K retrieval is not necessarily, from the user's perspective, a bad thing. The top K documents by the cosine measure are already not necessarily the K best for the query. By using heuristics below, we are likely to retrieve K documents with cosine scores close to those of the top K documents; these may not, from the user's perception, necessarily be far less relevant than the top K.

Index elimination

For a multi-term query q, we now consider a number of heuristics for cutting down the number of documents for which we compute the cosine similarity to the query q. From the preceding discussion, it is clear we only consider documents containing at least one of the query terms. We can take this a step further using additional heuristics:

1. We only consider documents containing terms whose idf exceeds a preset threshold. Thus, in the postings traversal, we only traverse the postings

for terms with high idf. This has a fairly significant benefit: the postings lists of low-idf terms are generally long; with these removed from contention, the set of documents for which we compute cosines is greatly reduced. One way of viewing this heuristic: low-idf terms are treated as stopwords and do not contribute to scoring.

2. We only consider documents that contain many (and as a special case, all) of the query terms. This can be accomplished by viewing the query as a conjunctive query; during the postings traversal, we only compute scores for documents containing all (or many) of the query terms. A danger of this scheme is that by seeking all (or even many) query terms to be present in a document before considering it for cosine computation, we may end up with fewer than K candidate documents to be in the output.

Champion lists

The idea of *champion lists* is to precompute, for each term t in the dictionary, the set of the m documents with the highest weights for t; the value of m is chosen in advance, appropriately. For tf-idf weighting, these would be the m documents with the highest tf values for term t. We call this set of m documents the *champion list* for term t.

Now, given a query q we take the union of the champion lists for each of the terms comprising q. We now restrict cosine computation to only the documents in this union. A critical parameter in this scheme is the value M, which is highly application dependent. Indeed, there is no reason to have the same value of M for all terms in the dictionary.

Cluster pruning

In *cluster pruning* we have a preprocessing step during which we cluster the document vectors. Then at query time, we consider only documents in a small number of clusters as candidates for which we compute cosine scores. Specifically, the preprocessing step is as follows:

1. Pick $\sqrt{N}$ documents at random from the collection. Call these *leaders*.

2. For each document that is not a leader, we compute its nearest leader.

We refer to documents that are not leaders as *followers*. Intuitively, in the partition of the followers induced by the use of $\sqrt{N}$ randomly chosen leaders, the expected number of followers for each leader is $\simeq N/\sqrt{N} = \sqrt{N}$. Next, query processing proceeds as follows:

1. Given a query q, find the leader L that is closest to q. This entails computing cosine similarities from q to each of the $\sqrt{N}$ leaders.

2. The candidate set consists of L together with its followers. We compute the cosine scores for all documents in this candidate set.

The use of randomly chosen leaders for clustering is fast and likely to reflect the distribution of the document vectors in the vector space: a region of the vector space that is dense in documents is likely to produce multiple leaders and thus a finer partition into sub-regions.

Variations of cluster pruning introduce additional parameters a and b, both of which are positive integers. In the pre-processing step we attach each follower to its a (rather than single) closest leaders. At query time we consider the b (rather than one) leaders closest to the query q. Clearly, the basic scheme above corresponds to the case $a = b = 1$. Further, increasing a or b increases the likelihood of finding K documents that are more likely to be in the set of true top-scoring K documents, at the expense of more computation.

7.3 Interaction between vector space and other retrieval methods

We conclude this chapter by discussing how the vector space scoring model relates to the query operators we have studied before. The relationship should be viewed at two levels: in terms of the expressiveness of queries that a user may pose, and in terms of the index that supports the evaluation of the various retrieval methods. The reason to understand these two levels: in building a search engine, we may opt to support multiple query operators for an end user. In doing so we need to understand what components of the index can be shared, as well as how to handle user queries that mix various query operators.

FREE-TEXT RETRIEVAL

Vector space scoring supports so-called *free-text retrieval*, in which a query is specified as a set of words without any query operators connecting them. It allows documents matching the query to be scored and thus ranked, unlike the Boolean, wildcard and phrase queries studied earlier. How does such vector space retrieval and scoring relate to the techniques we studied earlier, which demanded that specific terms or text strings be present in a document to be retrieved?

Boolean retrieval

Clearly a vector space index can be used to answer Boolean queries (as long as the weight of a term t in the vector representation of document d is non-zero whenever term t occurs in document d), but not vice versa. There is no easy way of combining vector space and Boolean queries from a user's standpoint: vector space queries are fundamentally a form of *evidence accumulation*, where the presence of more query terms in a document adds to the score of a document. Boolean retrieval on the other hand, requires a user to

specify a formula for selecting documents through the presence of specific keywords, without inducing any relative ordering among them.

Wildcard queries

Wildcard and vector space queries require different indexes, except at the basic level that both can be implemented using postings and a dictionary (e.g., a dictionary of trigrams for wildcard queries). If a search engine allows a user to specify a wildcard operator as part of a free-text query (for instance, the query rom* restaurant), we may interpret the wildcard component of the query as spawning multiple terms in the vector space (in this example, rome and roma would be two such terms) all of which are added to the query vector. The vector space query is then executed as usual, with matching documents being scored and ranked; thus a document containing both rome and roma is likely to be scored higher than another containing only one of these variants of writing the capital of Italy. (The exact score ordering will of course depend on the relative weights of each term in matching documents.)

Phrase queries

The representation of documents as vectors is fundamentally lossy: the relative order of terms in a document is lost in the encoding of a document as a vector. Thus an index built for vector space retrieval cannot, in general, be used for phrase queries. Moreover, there is no way of demanding a vector space score for a phrase query — we only know the relative weights of each term in a document. Thus, on the query german shepherd, we could use vector space retrieval to identify documents heavy in these two terms, with no way of prescribing that they occur consecutively. Phrase retrieval, on the other hand, tells us of the existence of the phrase german shepherd in a document, without any indication of the relative frequency or weight of this phrase. Even if we were to try and somehow treat every biword as a term (and thus an axis in the vector space, a questionable encoding as different axes now get correlated), notions such as idf would have to be extended to such biwords. While these two retrieval paradigms (phrase and vector space) consequently have different implementations in terms of indexes and retrieval algorithms, they can in some cases be combined usefully, as detailed below.

7.3.1 Query parsing and composite scoring

Common search interfaces, particulary for consumer-facing search applications on the web, tend to mask query operators from the end user. The idea is to hide the complexity of these operators from the largely non-technical audience for such applications. Given such interfaces, how should a search

equipped with indexes for various retrieval operators treat a query consisting of, say three terms (for instance, rising interest rates)?

The answer of course depends on the user population, the query distribution and the collection of documents. Typically, a *query parser* is used to translate the user-specified keywords into a query with various operators that is executed against the underlying indexes. Sometimes, this execution can entail multiple queries against the underlying indexes; for instance, the query parser may issue a stream of queries as in this example:

1. Run the user-generated query string as a phrase query; in our example, the phrase query would be rising interest rates. Rank them by vector space scoring using as query the vector consisting of the 3 terms.
2. If fewer than ten documents contain the phrase rising interest rates, run the two 2-term phrase queries rising interest and interest rates; rank these using vector space scoring, as well.
3. If we still have fewer than ten results, run the vector space query consisting of the three query terms.

Each of these steps (if invoked) may yield a list of scored documents; we may interleave these lists, or we may choose to present all results from Step 1 followed by results from Step 2, each of these as a scored list. How do we make these choices when designing an application? The current state of the art here is still somewhat primitive and does not offer principled methodologies for synthesizing complex query executions like in the example above. At the end of this chapter we give pointers to some published work to this end, where the approach is to cast the problem as a machine learning problem as outlined in Section 6.3.4 from Chapter 6.

7.4 References and further reading

In weighted (as opposed to Boolean retrieval) postings lists are often impact or weight ordered instead of ordered according to docID. See Chapter 4, page 59 for further discussion.

Singhal et al. (1996)

Fast query processing with early termination in weighted indexes is described by Anh et al. (2001) and Anh and Moffat (2006b). Indexes optimized for fast weighted retrieval complicate Boolean and phrase searches since inverted lists are no longer in document order. See Anh and Moffat (2006c) for an index structure that supports both weighted and Boolean / phrase searches.

8 *Evaluation in information retrieval*

We have seen in the preceding chapters many alternatives in designing a system. How do we know which of these techniques are effective in which applications? Should we use stop lists? Should we stem? Should we use inverse document frequency weighting? Information retrieval has developed as a highly empirical discipline, requiring careful and thorough evaluation to demonstrate the superior performance of novel techniques on representative document collections.

In this chapter we begin with the straightforward notion of relevant and irrelevant documents and first present the formal evaluation methodology that has been developed for evaluating unranked retrieval results (Section 8.1). This includes explaining the kinds of evaluation measures that are standardly used for retrieval and related tasks like categorization and why they are appropriate. We then extend these notions and develop further measures for evaluating ranked retrieval results (Section 8.2) and discuss developing reliable and informative test collections (Section 8.3). We then step back to introduce the notion of user utility, and how it is approximated by the use of document relevance (Section 8.4). At the end of the day, the key measure is user happiness. Speed of response and the size of the index are factors in user happiness. It seems as if relevance of results should be the most important factor: blindingly fast, useless answers do not make a user happy. However, user perceptions do not always coincide with system designers' notions of quality. For example, user happiness commonly depends very strongly on user interface design issues, including the layout, clarity, and responsiveness of the user interface, which are independent of the quality of the results returned. We will touch on other measures of the quality of an IR system, and in particular will discuss generating high-quality result summary snippets, which strongly influence user utility, but are not being measured in the basic relevance ranking paradigm (Section 8.5).

8.1 Evaluating information retrieval systems and search engines

To measure information retrieval effectiveness in the standard way, we need three things:

1. A test collection of documents
2. A benchmark suite of information needs, expressible as queries
3. A binary assessment of either *relevant* or *not relevant* for each query-document pair.

RELEVANCE

The standard approach to information retrieval system evaluation revolves around the notion of *relevant* and *not relevant* documents. With respect to a user information need, a document is given a binary classification as either relevant or not relevant. The test collection and set of information needs need to be of a reasonable size: you need to average performance over fairly large test sets, as results are very variable over different documents and information needs.

INFORMATION NEED

Relevance is assessed relative to an information need *not* a query. For example, an information need might be:

> I'm looking for information on whether drinking red wine is more effective at reducing your risk of heart attacks than white wine.

This might be translated into a query such as:

wine AND red AND white AND heart AND attack AND effective

A document is relevant if it addresses the stated information need, not because it just happens to contain all the words in the query. This distinction is often misunderstood in practice, because the information need is not overt. But, nevertheless, an information need is present. If I type python into a web search engine, I might be wanting to know where I can purchase a pet python. Or I might be wanting information on the programming language Python. From a one word query, it is very difficult for a system to know what my information need is. But, nevertheless, I have one, and can judge the returned results on the basis of their relevance to it. To do a system evaluation, we require an overt expression of an information need, which can be used for judging returned documents as relevant or not relevant. At this point, we make a simplification: you could reasonably think that relevance is a scale, with some documents highly relevant and others marginally so. But for the moment, we will use just a binary decision of relevance. We discuss the reasons for using binary relevance judgements and alternatives in Section 8.4.3.

8.1.1 Standard benchmarks for relevance

Here we present some of the most standard benchmark data sets for information retrieval system evaluation.

CRANFIELD
- The *Cranfield* collection was the pioneering document collection in providing precise quantative metrics of information retrieval performance. Collected in the United Kingdom starting in the late 1950s, it contains 1398 abstracts of aerodynamics journal articles, a set of 225 queries, and exhaustive relevance judgements.

TREC
- *TREC*. The U.S. National Institute of Standards and Technology (NIST) has run a large IR test bed evaluation series since 1992. Within this framework, there have been many tracks over a range of different test collections, but the classic test collection is the one built up for the TREC Ad Hoc track during the first 8 TREC competitions between 1992 and 1999. Taken together, this 5 CD TREC Test Collection comprises several million documents and 450 information needs, which are specified in detailed text passages. Because the document collection is so large, there are not exhaustive relevance judgements. Rather, NIST assessor relevance judgements have been gathered only for the documents that were among the top-k returned for some system which was entered in the TREC competition in which the information need was first used.

- For text classification the most used data set is the Reuters-21578 collection (see Chapter 13, page 206). These newswire articles were originally collected and labeled by Carnegie Group, Inc. and Reuters, Ltd. in the course of developing the CONSTRUE text categorization system. Articles are labeled with some number of topical classifications (most commonly one, but there may be none or several). More recently, Reuters has released the much larger Reuters Corpus Volume 1 (RCV1); see Chapter 4, page 52.

- Another very widely used text classification collection is the 20 Newsgroups collection, a collection of 1000 articles from each of 20 Usenet newsgroups which was collected by Ken Lang, containing 18941 articles with the removal of duplicates.

8.1.2 Measures of retrieval performance

Given these ingredients, how is system performance measured? The two most frequent and basic measures for information retrieval performance are precision and recall. These are first defined for the simple case where an IR system returns a set of documents for a query. We will see later how to extend these notions to ranked retrieval situations.

PRECISION • *Precision* is the fraction of retrieved documents that are relevant

$$\text{Precision} = \frac{\#(\text{relevant items retrieved})}{\#(\text{retrieved items})} = P(\text{relevant}|\text{retrieved}) \tag{8.1}$$

RECALL • *Recall* is the fraction of relevant documents that are retrieved

$$\text{Recall} = \frac{\#(\text{relevant items retrieved})}{\#(\text{relevant items})} = P(\text{retrieved}|\text{relevant}) \tag{8.2}$$

These notions can be made clear by examining the following contingency table:

(8.3)

	Relevant	Not relevant
Retrieved	true positives (tp)	false positives (fp)
Not retrieved	false negatives (fn)	true negatives (tn)

Then:

$$\begin{aligned} P &= tp/(tp+fp) \\ R &= tp/(tp+fn) \end{aligned} \tag{8.4}$$

Given that there are two actual classes: relevant and not relevant, and that an information retrieval system can be thought of as a two class classifier which attempts to label them as such (it retrieves the subset of documents which it believes to be relevant), an obvious alternative that may occur to the reader is to judge an information retrieval system by its *accuracy*. The accuracy of a system is the fraction of its classifications that are correct. In terms of the contingency table above, accuracy $= (tp+tn)/(tp+fp+fn+tn)$. Indeed, precisely this measure is the measure normally used for machine learning classification problems.

ACCURACY

There is a good reason why accuracy is not an appropriate measure for information retrieval problems. In almost all circumstances, the data is extremely skewed: normally over 99.9% of the documents are in the not relevant category. In such circumstances, a system tuned to maximize accuracy will almost always declare every document not relevant. Even if the system is quite good, trying to label some documents as relevant will almost always lead to an unacceptably high rate of false positives. However, this behavior is completely unsatisfying to an information retrieval system user. A user is always going to want to see some documents, and can be assumed to have a certain tolerance for seeing some false positives providing that they get some useful information. The measures of precision and recall concentrate the evaluation on the return of true positives, asking what percentage of the

relevant documents have been found and how many false positives have also been returned.

The advantage of having the two numbers for precision and recall is that one is more important than the other in many circumstances. Typical web surfers would like every result on the first page to be relevant (high precision) but have not the slightest interest in knowing let alone looking at every document that is relevant. In contrast, various professional searchers such as paralegals and intelligence analysts are very concerned with trying to get as high recall as possible, and will tolerate fairly low precision results in order to get it. Nevertheless, the two quantities clearly trade off against one another: you can always get a recall of 1 (but very low precision) by retrieving all documents for all queries! Recall is a non-decreasing function of the number of documents retrieved. On the other hand, in a good system, precision usually decreases as the number of documents retrieved is increased. In general one wants to get some amount of recall while tolerating only a certain percentage of false positives.

That is, one wants to provide a measure that trades off precision versus recall. This also results in a single measure by which systems can be measured. The combined measure which is standardly used is called the *F measure*, which is the weighted harmonic mean of precision and recall:

F MEASURE

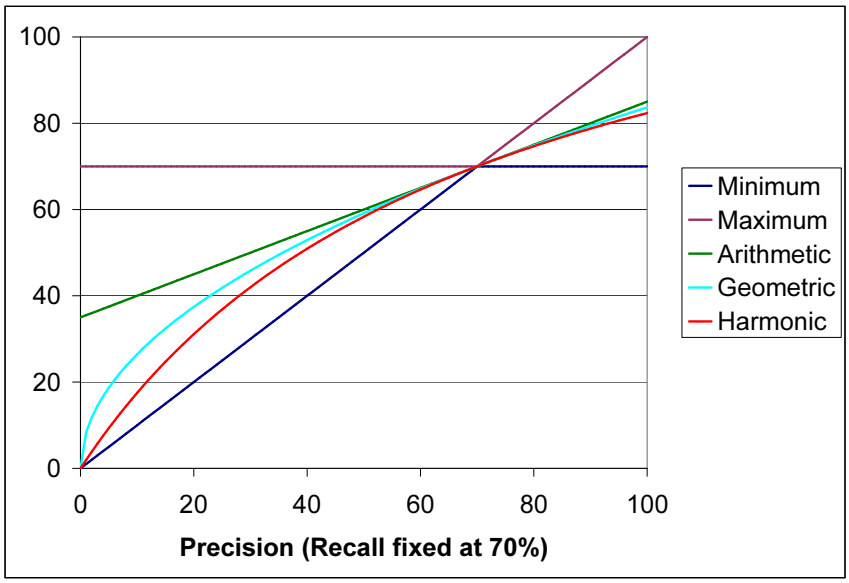
$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)PR}{\beta^2 P + R} \tag{8.5}$$

where $\alpha = 1/(\beta^2+1)$. The default is to equally weight precision and recall, giving a balanced F measure. This corresponds to making $\alpha = 1/2$ or $\beta = 1$. You will commonly see it written as F_1, which is short for $F_{\beta=1}$, even though the formulation above in terms of α more transparently exhibits the F measure as a weighted harmonic mean. When using $\beta = 1$, the formula on the right simplifies to the easy to remember formula:

$$F_{\beta=1} = \frac{2PR}{P+R} \tag{8.6}$$

However, using an even weighting is not the only choice. Values of $\beta < 1$ emphasize precision, while values of $\beta > 1$ emphasize recall. For example, a value of $\beta = 3$ or $\beta = 5$ might be used if recall is to be emphasized. Recall, precision, and the F measure are inherently measures between 0 and 1, but they are also very commonly written as percentages, on a scale between 0 and 100.

But what is a harmonic mean, and why do we use it, rather than the more usual average (arithmetic mean)? One hint as to the unsuitability of the arithmetic mean is to recall that we can always get 100% recall by just returning all documents, and therefore we can always get a 50% arithmetic mean by

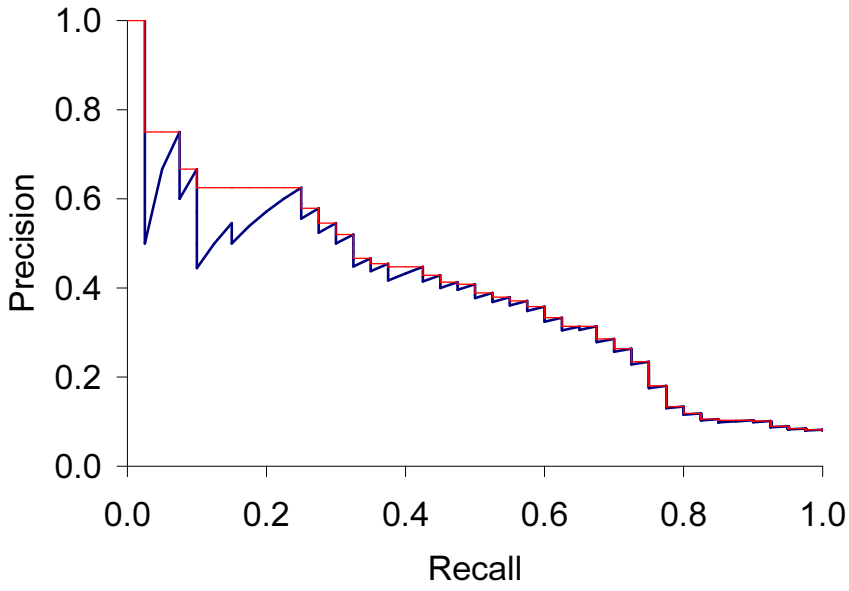


► **Figure 8.1** Graph comparing the harmonic mean to other means. The graph shows a slice through the calculation of various means of precision and recall for the fixed recall value of 70%. The harmonic means is always less than either the arithmetic or geometric mean, and often quite close to the minimum of the two numbers. When the precision is also 70%, all the measures coincide.

the same process. In contrast, if we assume that 1 document in 10000 is relevant to the query, the harmonic mean score of this strategy is 0.02%. Many readers will have seen the geometric mean of $x_1, \ldots, x_n = (x_1 \cdots x_n)^{1/n}$ and will know that the geometric mean is always less than or equal to the arithmetic mean (remember, $\sqrt{1 \times 9} < (1+9)/2$). The harmonic mean, the third of the classical Pythagorean means, is even more conservative: it is always less than or equal to the geometric mean. The harmonic mean is closer to the minimum of two numbers than to their arithmetic mean. See Figure 8.1.

8.2 Evaluation of ranked retrieval results

Precision, recall, and the F measure are set-based measures. They are computed using unordered sets of documents. We need to extend these measures (or to define new measures) if we are to evaluate the ranked retrieval results that are now standard in information retrieval. In a ranking context, appropriate sets of retrieved documents are naturally given by the top k retrieved documents. For each such set, the precision and recall can be calculated, and



► **Figure 8.2** Precision/Recall graph.

PRECISION-RECALL CURVE

the resulting values can then be plotted to give a *precision-recall curve*, such as the one shown in Figure 8.2. Precision-recall curves have a distinctive saw-tooth shape: necessarily, if the $(k+1)^{\text{th}}$ document retrieved is irrelevant then recall is the same as for the top k documents, but precision has dropped. If it is relevant, then both precision and recall increase, and the curve jags up and to the right. It is often useful to remove these jiggles and the standard way to do this is with an interpolated precision: the precision at a certain recall level r is defined as the highest precision found for any recall level $q \geq r$. The justification is that almost anyone would be prepared to look at a few more documents if it would increase the percentage of the viewed set that were relevant (that is, if the precision of the larger set is higher). Interpolated precision is shown by a thinner line in Figure 8.2. Note that with this definition, the interpolated precision at a recall of 0 is well-defined.

Precision-recall curves are often very informative. Nevertheless, just as with the introduction of the F measure, there is often a desire to boil this information down into a few numbers, or perhaps even a single number. The traditional way of doing this, used for instance in the first 8 TREC Ad Hoc evaluations, was an *11-point interpolated average precision*. For each information need, the interpolated precision is measured at the 11 recall levels of 0.0, 0.1, 0.2, ..., 1.0. The measured interpolated precision is then averaged (i.e.,

11-POINT INTERPOLATED AVERAGE PRECISION

Recall	Interp. Precision
0.0	1.00
0.1	0.67
0.2	0.63
0.3	0.55
0.4	0.45
0.5	0.41
0.6	0.36
0.7	0.29
0.8	0.13
0.9	0.10
1.0	0.08

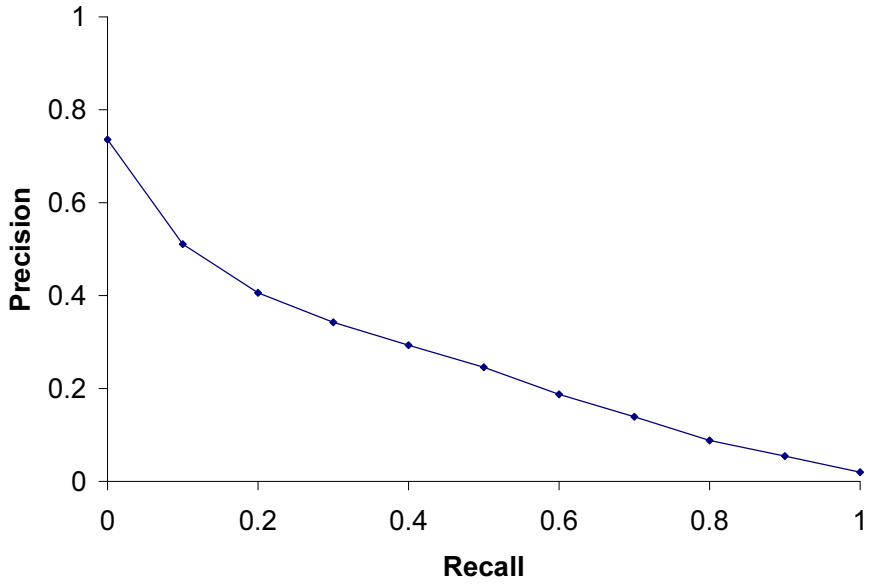
► **Table 8.1** Calculation of 11-point Interpolated Average Precision. This is for the precision-recall curve shown in Figure 8.2.

arithmetic mean) over the set of queries in the benchmark. For the precision-recall curve in Figure 8.2, these 11 values are shown in Table 8.1. Once they are averaged with other queries, a composite precision-recall curve showing 11 points can be graphed. Figure 8.3 shows an example graph of such results from a representative good system at TREC 8.

MEAN AVERAGE PRECISION

In recent years, other measures have tended to take center stage. Most standard among the TREC community is *Mean Average Precision* (MAP), which provides a single-figure measure of quality across recall levels. For one information need, it is the average of the precision value obtained for the top set of k documents existing after each relevant document is retrieved. When a relevant document is not retrieved at all, the precision value is taken to be 0. Under this scheme, fixed recall levels are not chosen, and there is no interpolation. The MAP value for a test collection is then the arithmetic mean of MAP values for individual information needs. (Note that this has the effect of weighting each information need equally in the final reported number, even if many documents are relevant to some queries whereas very few are relevant to other queries.) It should be noted that it is normally the case that calculated MAP scores vary widely across information needs when measured for the same system, for instance, between 0.1 and 0.7. There is normally more agreement in MAP for an individual information need across systems than for MAP scores for different information needs for the same system.

The above measures factor in precision at all recall levels. However, for many prominent applications, particularly web search, these numbers may not be highly relevant to users. What matters is rather how many good re-



► **Figure 8.3** Averaged 11-Point Precision/Recall graph across 50 queries for a representative TREC system. The Mean Average Precision for this system is 0.2553.

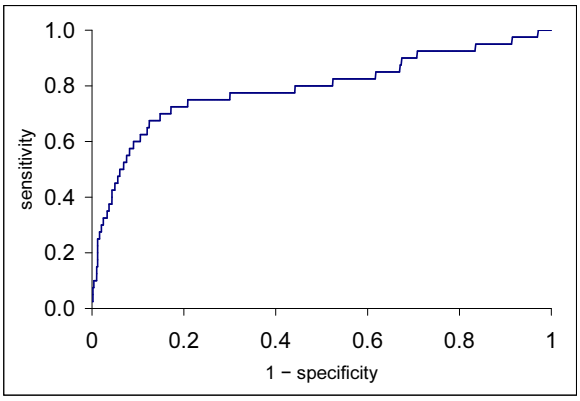
sults there are on the first page or the first three pages. This leads to measures of precision at fixed low levels of retrieved results, such as 10 or 30 documents. This is referred to as "Precision at k", for example "Precision at 10". This measure also makes for easy comparisons with systems that do not rank their returned results. It has the disadvantage that it does not average well, since the total number of relevant documents for a query has a strong influence on precision at k.

R-PRECISION

An alternative, which alleviates this problem, is *R-precision* (Allan 2005). It requires having a known set of relevant documents (though it may be incomplete, such as when this set is formed by examining the pooled top k results of systems in a competition). If this set has size Rel, then we calculate the precision on the top Rel documents returned. The idea is that this measure adjusts for the size of the set of relevant documents: A perfect system could score 1 on this metric for each query, whereas, even a perfect system could only achieve a precision at 20 of 0.4 if there were only 8 documents in the collection relevant to an information need. Averaging this measure across queries makes more sense, but the measure is perhaps harder to explain to naive users than Precision at k.

BREAK-EVEN POINT

Another measure that is sometimes used, given a precision-recall curve, is the *break-even point*: the value at which the precision is equal to the recall.



► **Figure 8.4** The ROC curve corresponding to the precision-recall curve in Figure 8.2.

Like the Precision at k, it describes only one point on the precision-recall curve, rather than attempting to summarize performance across the curve.

ROC CURVE

Another concept sometimes used in evaluation is an *ROC curve* ("ROC" stands for "Receiver Operating Characteristics", but knowing that does not really help most people). An ROC curve plots the true positive rate or sensitivity against the false positive rate or (1 − specificity). Here, *sensitivity* is just another term for recall and the false positive rate is given by $fp/(fp + tn)$. For unranked retrieval sets in IR, *specificity*, given by $tn/(fp + tn)$, was not seen as such a useful notion, as its value would be almost 1 for all information needs, since the set of true negatives is always so large (and, correspondingly, the value of the false positive rate would be almost 0; note that the "interesting part" of Figure 8.2 is 0 <recall< 0.4, a part which is compressed to a small corner of Figure 8.4). But it does make sense if looking over the full retrieval spectrum, and provides another way of looking at the data. Figure 8.4 shows the ROC curve corresponding to the precision-recall curve in Figure 8.2. An ROC curve always goes from the bottom left to the top right of the graph. For a good system, the graph climbs steeply on the left side. A common aggregate measure is then to report the area under the ROC curve. Finally, note that precision-recall curves are sometimes loosely referred to as ROC curves. This is understandable, but not quite accurate.

SENSITIVITY

SPECIFICITY

We have discussed computing precision, recall, F and ROC curves for the retrieval unit of documents. These measures have also been extended to measure the retrieval of units at different scales such as passages, or retrieval of facts or "named entities" (things like person names or car companies). Measurement at different scales may produce different results.

Judge 1	Judge 2	Number of docs
Relevant	Relevant	300
Relevant	Not relevant	20
Not relevant	Relevant	10
Not relevant	Not relevant	70

$P(A) = 370/400 = 0.925$
$P(nonrelevant) = (10 + 20 + 70 + 70)/800 = 0.2125$
$P(relevant) = (10 + 20 + 300 + 300)/800 = 0.7878$
$P(E) = 0.2125^2 + 0.7878^2 = 0.665$
$Kappa = (0.925 - 0.665)/(1 - 0.665) = 0.776$

▶ **Table 8.2** Calculating the kappa statistic.

8.3 From documents to test collections

Once one has a large collection of documents, to have a test collection, you also need test information needs and relevance assessments for them. The test information needs must be germane to the documents in the test collection, and appropriate for predicted usage of the system. These information needs are best designed by domain experts. Using random query terms as an information need is generally not a good idea. Then one needs to collect relevance assessments. This is a time-consuming and expensive process involving human beings, and raises the question as to whether human judgements of relevance are reliable and consistent.

KAPPA MEASURE

In the social sciences, a common measure for agreement between judges is the *Kappa measure*. It is designed for categorical judgments and corrects a simple agreement rate for the rate of chance agreement.

(8.7)
$$Kappa = \frac{P(A) - P(E)}{1 - P(E)}$$

where $P(A)$ is the proportion of the time the judges agreed, and $P(E)$ is the proportion of the time they would be expected to agree by chance. There is some latitude in how the latter is estimated: if we simply say we are making a 2 class decision and examine nothing more, then the expected chance agreement rate is best taken as 0.5. However, if we know that the classes are

MARGINAL

skewed, it is usual to use the *marginal* statistics to calculate expected agreement. The calculations are shown in Table 8.2. The kappa value will be 1 if two judges always agree, 0 if they agree only at the rate given by chance, and negative if they are worse than random. If there are more than two judges, it is normal to calculate an average pairwise kappa value. As a rule of thumb, a kappa value above 0.8 is taken as good agreement, and a kappa value be-

tween 0.67 and 0.8 is taken as fair agreement, and agreement below 0.67 is seen as data providing a dubious basis for an evaluation, though the details here depend on the purposes for which the data will be used.

Interjudge agreement of relevance has been measured within the TREC evaluations and for medical IR collections (Hersh et al. 1994). Within the above rules of thumb, the level of agreement normally falls in the range of "fair" (0.67–0.8). Taking one or other of two judges opinions as the *gold standard* can make a considerable absolute difference to reported scores, but has in general been found to have little impact to the relative performance of different systems, that is to their ranking. The fact that human agreement on a binary relevance judgement is quite modest is one reason for not requiring more fine-grained relevance labeling from the test set creator.

GOLD STANDARD

Can we avoid the use of human judgments for system evaluation? Not really: no purely formal criterion can assess the task we are interested in. Nevertheless, the need for human judgments makes experimental work hard, especially on a large scale. In some very special settings, we can use proxies. One example of this is for evaluating approximate vector space retrieval methods. In approximate vector space retrieval, we are using some approximation to the exact return of the vector space model. Given n document vectors and a query, an exact algorithm would return the k vectors closest to the query. For this, we know of no better way than to compute cosine similarities between the query and every document. However, there are many approximate retrieval schemes, such as cluster pruning, which we discussed in Chapter 7. Is there a surrogate way to measure the goodness of such an approximate retrieval scheme?

In this case there is. Let $G(q)$ be the "ground truth" of the actual k closest documents to a query q. Then let $A(q)$ be the k documents returned by the approximate algorithm A on query q. As a surrogate for precision and recall, we can measure $A(q) \cap G(q)$. Or, we can use other notions of how $A(q)$ compares to $G(q)$. Goodness can be measured here in terms of cosine promity to q: we sum up $q \cdot d$ over $d \in A(q)$ and compare this to the sum of $q \cdot d$ over $d \in G(q)$. This yields a measure of the relative goodness of A vis-à-vis G. According to it A may be 90% as good as the ground-truth G, without actually finding 90% of the documents in G. For scored retrieval, this may be acceptable. Many web search engines do not always return the same answers for a given query (for many reasons, including time-outs, and different computer clusters running different index versions).

8.4 A broader perspective: System quality and user utility

This chapter has so far concentrated on formal system evaluation measures for retrieved document set quality. This is both a narrow perspective on qual-

ity and a formal evaluation measure which is at some distance from our ultimate interest in measures of human utility: how satisfied is the user with a system? The standard way to measure human satisfaction is by various kinds of user studies which might include quantitative measures (time to complete a task) as well as vaguer quantitative measures (e.g., a score for satisfaction with the search engine) and qualitative measures (such as user comments on the search interface). In this section we will touch on problems with and justifications for working with document relevance, other system aspects that allow quantitiative evaluation, and the issue of user utility.

8.4.1 System issues

There are many practical benchmarks on which to rate an information retrieval system beyond its retrieval quality. These include:

- How fast does it index? (How many documents per hour does it index for a certain distribution over document sizes.)
- How fast does it search? (Its latency as a function of index size.)
- How expressive is its query language? (One might also be interested in its speed on complex forms of queries.)
- How large is its document collection? (Number of documents or its distribution of documents across topics.)

All the criteria above apart from query language expressiveness are straightforwardly *measurable*: we can quantify the speed or size. Various kinds of feature checklists can make query language expressiveness semi-precise. A system can also be evaluated on such metrics.

8.4.2 User utility

What we would like is a way of quantifying aggregate user happiness. One part of this is understanding the distribution of people we wish to make happy, and this depends entirely on the setting. For a web search engine, a happy search user is someone who finds what they want. One indirect measure of such users is that they tend to return to the same engine. Measuring the rate of return users is thus an effective metric. But advertisers are also users of modern web search engines. They are happy if customers click through to their sites and then make purchases. On an eCommerce web site, a user is presumably also at least potentially wanting to purchase something. We can perhaps measure the time to purchase, or the fraction of searchers who become buyers. On a shopfront web site, perhaps both the user's and the store owner's needs are satisified if a purchase is made. Nevertheless,

in general, we need to decide whether it is the end user's or the eCommerce site's owner's happiness that we are trying to optimize.

For an "enterprise" (company, government, or academic) intranet search engine, the relevant metric is more likely to be user productivity: how much time do users spend looking for information that they need. There are also many other practical criteria concerning such matters as information security which we will touch on further later.

User happiness is elusive to measure, and this is part of why the standard methodology above uses the proxy of relevance of search results. The standard direct way to get at user satisfaction is to run user studies, where people engage in tasks, and usually various metrics are measured, the participants are observed, and ethnographic interview techniques are used to get qualitative information on satisfaction. User studies are very useful in system design, but they are time consuming and expensive to do. They are also difficult to do well, and expertise is required to design the studies and to interpret the results. We will not discuss the details of human usability testing here.

8.4.3 Document relevance: critiques and justifications of the concept

The advantage of system evaluation, as enabled by the standard model of relevant and irrelevant documents, is that one then has a fixed setting in which one can manipulate parameters and carry out comparative experiments. Such formal testing is much less expensive and allows clearer diagnosis of the effect of changing parameters. Indeed, once one has a formal measure that one has confidence in, rather than doing experiments by hand, one can proceed to optimize performance by machine learning methods. Of course, if the formal measure poorly describes what users actually want, doing this will be effective in optimizing the formal measure but ineffective in improving user satisfaction. While the standard formal measures for IR evaluation are a simplification, our perspective is that in practice, the measures are good enough, and recent work in optimizing formal evaluation measures in IR has succeeded brilliantly. There are numerous examples of techniques developed in formal evaluation settings that improve performance in operational settings.

That is not to say that there are not abstractions going on and problems latent within them. The relevance of one document is treated as independent of other documents in the collection. Assessments have to be binary: there aren't any nuanced assessments of relevance. Assessments are not clear: people are not reliable assessors of relevance of a document to an information need. Judgements vary across people, as we discussed above. We also have to assume that a user's information need does not change as they start looking at retrieval results. Any results from one collection are heavily skewed by

the choice of collection, queries, and relevance judgment set: the results may not translate from one domain to another or to a different user population.

Some of these problems may be fixable. A number of recent evaluations, including INEX, some TREC tracks, and NCTIR (the east Asian IR evaluation forum) have adopted a graded notion of relevance with documents divided into 3 or 4 classes, distinguishing slightly relevant documents from highly relevant documents. See Section 10.4 for a detailed discussion of how this is implemented in the INEX evaluations.

MARGINAL RELEVANCE

One clear problem with the relevance-based assessment that we have presented is the distinction between Relevance vs. *Marginal Relevance*: whether a document still has distinctive usefulness after having looked at certain other documents (Carbonell and Goldstein 1998). Even if a document is highly relevant, its information can be completely redundant with other documents which have already been examined. The most extreme case of this is documents which are duplicates – a phenomenon that is actually very common on the World Wide Web – but it can also easily occur when several documents provide a similar precis of an event. In such circumstances, marginal relevance is clearly a better measure of utility to the user. Maximizing marginal relevance requires returning documents that exhibit diversity and novelty. One way to approach measuring this is by using distinct facts or entities as evaluation units. This perhaps more directly measures true utility to the user but doing this makes it harder to create an evaluation data set.

8.5 Results snippets

Having chosen or ranked the documents matching a query, we wish to present a results list that will be informative to the user. In many cases the user will not want to examine all the returned documents and so we want to make the results list informative enough that the user can do a final ranking of the documents for themselves based on relevance to their information need.[1] The

SNIPPET

standard way of doing this is to provide a *snippet*, a short summary of the document, which is designed so as to allow the user to decide its relevance. Typically, the snippet has the document title and a short summary, which is automatically extracted. The question is how to design the summary so as to maximize its usefulness to the user.

The two basic kinds of summaries are static summaries and dynamic, or query-dependent, summaries. A static summary of a document is always the same, regardless of the query that was used. Dynamic summaries are customized according to the user's information need as deduced from a query.

1. There are exceptions, in domains where recall is being emphasized. For instance, in many legal disclosure cases, an associate will review *every* document that matches a keyword search.

They attempt to explain why a particular document was retrieved for the query at hand.

A static summary is generally comprised of either or both a subset of the document and metadata associated with the document. The simplest form of summary takes the first two sentences or 50 words or a document, or extracts particular zones of a document, such as the title and author. Rather than zones within a document, the summary can instead use metadata associated with the document which is designed to give a summary. One example of this is the page `description` metadata which can appear in the `meta` element of the document metadata of a web HTML page. This summary is typically extracted and cached at indexing time, in such a way that it can be retrieved and presented quickly when displaying search results (whereas, having to access the actual document content might be a relatively expensive operation).

There has been extensive work within natural language processing (NLP) on better ways to do text summarization. Most such work still aims only to choose sentences from the original document to present and concentrates on how to select good sentences. The models used typically combine positional factors, which favor the first and last paragraphs of documents and the first and last sentences of paragraphs, with content factors, which emphasize sentences with key terms, which have low document frequency in the collection as a whole, but high frequency and good distribution across the particular document being returned. In the most sophisticated NLP approaches, the system synthesizes sentences for a summary, either by doing full text generation or by editing and perhaps combining sentences used in the document. For example, it might delete a relative clause or replace a pronoun with the noun phrase that it refers to. This last class of methods remains in the realm of research and is seldom used for search results: it is easier, safer, and often even better to just use sentences from the original document.

Dynamic summaries display one or more "windows" on to the document, aiming to present the pieces that have the most utility to the user in evaluating the document with respect to their information need. Usually these windows contain one or several of the query terms, and so are often referred to as keyword-in-context (KWIC) snippets, though sometimes they may still be pieces of the text such as the title that are selected for their information value just as in the case of static summarization. Dynamic summaries are generated in conjunction with scoring. If the query is found as a phrase, occurrences of the phrase in the document will be shown as the summary. If not, windows within the document that contain multiple query terms will be selected. Commonly these windows may just stretch some number of words to the left and right of the query terms, but this is another place where NLP techniques can usefully be employed: users prefer snippets that read well because they contain complete phrases.

Dynamic summaries are generally regarded as greatly improving the usability of IR systems, but they present a complication for IR system design. A dynamic summary cannot be precomputed, but, on the other hand, if a system has only a positional index, then it cannot (easily) reconstruct the context surrounding search engine hits in order to generate such a dynamic summary. (This is one reason for opting to go with static summaries.) The standard solution to this in the modern world of large and cheap disk drives is to locally cache all the documents at index time (notwithstanding that this approach raises various legal and information security and control issues that are far from resolved). Then, a system can seek to positions in the document based on the hits found in the positional index. For example, if the positional index says "the query is a phrase in position 4378", the system can start at a few words before this position in the cached document and stream out the content. However, beyond simply access to the text, producing a good KWIC snippet requires some care. Given a variety of keyword occurrences in a document, the goal is to choose fragments which are: (i) maximally informative about the discussion of those terms in the document, (ii) self-contained enough as to be easy to read, and (iii) short enough to fit within the normally strict constraints on the space available for summaries.

Let us note a couple of other practical aspects. Generating snippets must be fast since the system is typically generating many snippets for each query that it handles. Rather than caching an entire document, it is common to only cache a generous but fixed size prefix of the document, such as perhaps 10,000 characters. For most common, short documents, the entire document is thus cached, but huge amounts of local storage will not be wasted on potentially vast documents. Their summaries will be based on material in the document prefix, which is in general a useful zone in which to look for a document summary anyway. Secondly, if a document has been updated recently, these changes will not be in the cache (any more than they will be reflected in the index). In these circumstances, neither the index nor the summary will accurately reflect the contents of the document, but it is the differences between the summary and the actual document content that will be more glaringly obvious to the end user.

8.6 Conclusion

As recently as the 1990s, studies showed that most people preferred getting information from other people rather than information retrieval systems. Of course, in that time period, most people also interacted with human travel agents to book their travel. While academic discussion of this process is unfortunately scant, in the last decade, relentless optimization of formal measures of performance has driven web search engines to new levels of per-

formance where most people are satisfied most of the time, and web search has become a standard and often preferred source of information finding. For example, the 2004 Pew Internet Survey (Fallows 2004) found that "92% of Internet users say the Internet is a good place to go for getting everyday information."

8.7 References and further reading

Rigorous formal testing of IR systems was first done in the Cranfield experiments, beginning in the late 1950s. A retrospective discussion of the Cranfield test collection and experimentation with it can be found in Cleverdon (1991). The other seminal series of early IR experiments were those on the SMART system by Gerald Salton and colleagues (Salton 1971; 1991). The TREC experiments are covered in Voorhees and Harman (2005). User studies of IR system effectiveness began more recently (Saracevic and Kantor 1988; 1996).

The F measure (or, rather its inverse $E = 1 - F$) is introduced in van Rijsbergen (1979), together with an extensive theoretical discussion of the criteria that motivate use of this particular measure.

A standard program for evaluating IR systems which computes many measures of ranked retrieval performance is the `trec_eval` program used in the TREC competitions. It can be downloaded from: http://trec.nist.gov/trec_eval/.

The kappa statistic and its use for language-related purposes is discussed by Carletta (1996). For further discussion of it and alternative measures, which may in fact be better, see Lombard et al. (2002), Krippendorff (2003), and Di Eugenio and Glass (2004).

Voorhees (1998) is the standard article for examining variation in relevance judgements and their effects on retrieval system scores and ranking for the TREC Ad Hoc task. She concludes that although the numbers change, the rankings are quite stable. In contrast, Kekäläinen (2005) analyzes some of the later TRECs, exploring a 4-way relevance judgement and the notion of cumulative gain, arguing that the relevance measure used does substantially affect system rankings. See also Harter (1998).

Text summarization has been actively explored for many years. Modern work on sentence selection was initiated by Kupiec et al. (1995). More recent work includes Barzilay and Elhadad (1997) and Jing (2000) (together with a broad selection of work appearing at the yearly DUC conferences and at other NLP venues).

Exercise 8.1 [⋆]

Derive the equivalence between the two formulas for F measure shown in Equation (8.5), given that $\alpha = 1/(\beta^2 + 1)$.

Exercise 8.2 [⋆]

What are the possible values for interpolated precision at a recall level of 0?

Exercise 8.3 [⋆]

Must there always be a break-even point between precision and recall? Either show there must be or give a counter-example.

Exercise 8.4 [⋆]

What is the relationship between the value of F_1 and the break-even point?

9 *Relevance feedback and query expansion*

Examining the failures of a straightforward retrieval system, it is usually obvious to a human that you will only be able to do high recall retrieval by also retrieving documents that refer to the same concept but which use different words. For example, you would want a search for aircraft to match plane (providing that it is an *airplane*, not a woodworking plane), and for a search on thermodynamics to match references to heat in appropriate discussions.

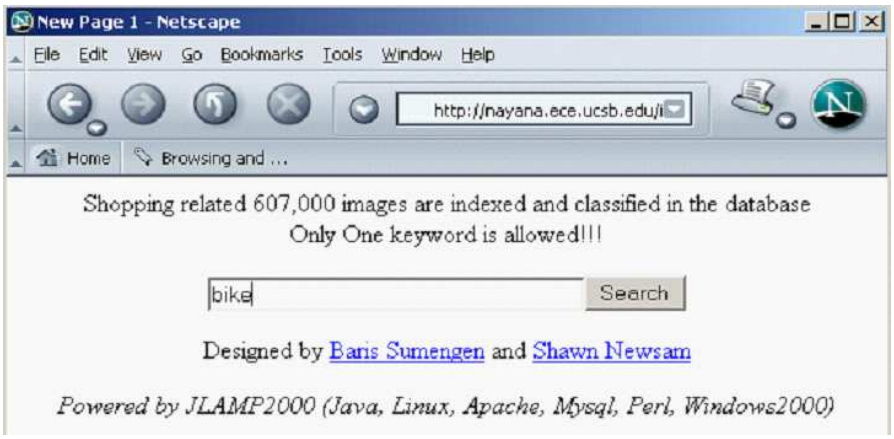
The methods for achieving this goal split into two major classes: global methods and local methods. Global methods are techniques for expanding or reformulating query terms which are independent of the query and results returned from it. The aim is that changes in the query wording will cause the new query to match other semantically similar terms. Global methods include:

- Query expansion/reformulation with a thesaurus (or WordNet)
- Query expansion via automatic thesaurus generation
- Techniques like spelling correction, which we discussed in Chapter 3

Local methods do adjustments to a query relative to the documents that initially appear to match the query. The basic methods here are:

- Relevance feedback
- Pseudo-relevance feedback (also known as Blind relevance feedback)
- (Global) indirect relevance feedback

In this chapter, we will mention all of these approaches, but we will concentrate on relevance feedback, which is one of the most used and most successful approaches.



► **Figure 9.1** An example of relevance feedback searching over images. Here, the user enters an initial query. From http://nayana.ece.ucsb.edu/imsearch/imsearch.html (Newsam et al. 2001).

9.1 Relevance feedback and pseudo-relevance feedback

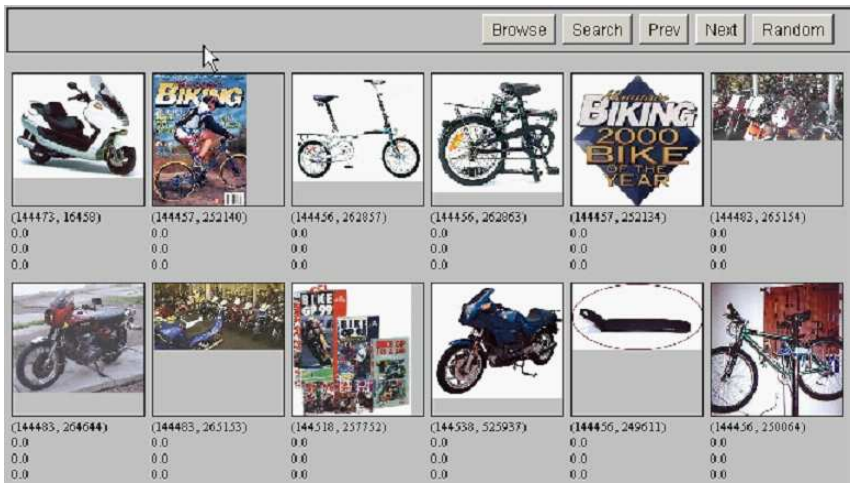
RELEVANCE FEEDBACK

The idea of *relevance feedback* is to involve the user in the retrieval process so as to improve the final result set. In particular, the user gives feedback on the relevance of documents in an initial set of results. The basic procedure is:

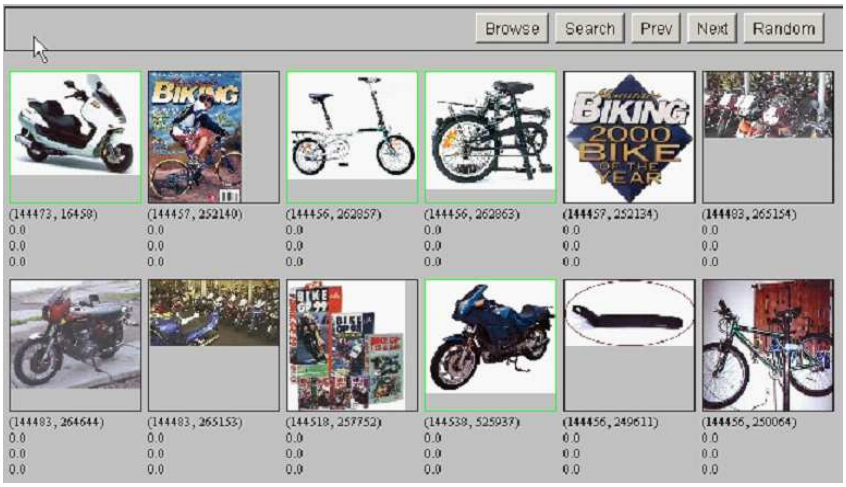
- The user issues a (short, simple) query.
- The system returns an initial set of retrieval results.
- The user marks some returned documents as relevant or not relevant.
- The system computes a better representation of the information need based on the user feedback.
- The system displays a revised set of retrieval results.

Relevance feedback can go through one or more iterations of this sort. The process exploits the idea that it may be difficult to formulate a good query when you don't know the collection well, but it is easy to judge particular documents, and so it makes sense to engage in iterative query refinement of this sort. In such a scenario, relevance feedback can also be effective in tracking a user's evolving information need: seeing some documents may lead the user to refine their understanding of the information they are seeking.

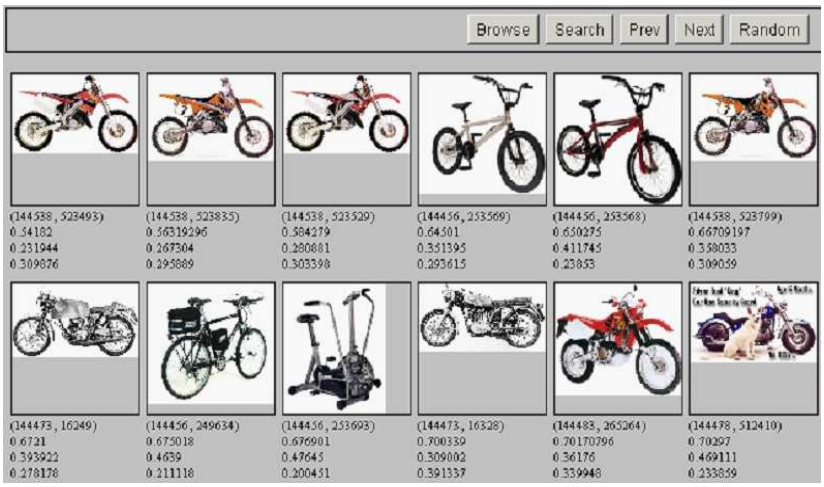
Image search provides a good example of relevance feedback. Not only is it easy to see the results at work, but this is a domain where a user can easily



► **Figure 9.2** An example of relevance feedback searching over images (continued). Here the user views the initial query results.



► **Figure 9.3** An example of relevance feedback searching over images (continued). Here the user provides feedback on relevant results.



► **Figure 9.4** An example of relevance feedback searching over images (continued). Here the user sees the revised result set.

have difficulty formulating what they want in words, but can easily indicate relevant or non-relevant images. Figure 9.1 shows a user entering an initial query for bike on the demonstraction system at:

http://nayana.ece.ucsb.edu/imsearch/imsearch.html

The initial results returned are shown in Figure 9.2. In Figure 9.3, the user has selected some of them as relevant. These will be used to refine the query, while other displayed results have no effect on the reformulation. Figure 9.4 then shows the new top-ranked results calculated after this round of relevance feedback.

Figure 9.5 shows a textual IR example where the user wishes to find out about new applications of space satellites.

9.1.1 The Rocchio Algorithm

The Rocchio Algorithm (Rocchio 1971) is the classic algorithm for implementing relevance feedback. It models a way of incorporating relevance feedback information into the vector space model of Chapter 7.

The underlying theory. We want to find a query vector that maximizes similarity with relevant documents while minimizing similarity with non-relevant documents. That is, if C_r is the set of relevant documents and C_{nr} is

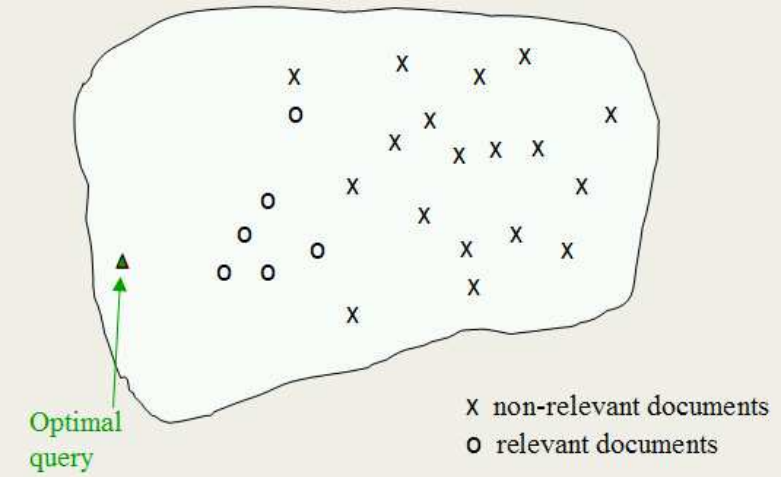
Query: New space satellite applications

+ 1. 0.539, 08/13/91, NASA Hasn't Scrapped Imaging Spectrometer
+ 2. 0.533, 07/09/91, NASA Scratches Environment Gear From Satellite Plan
3. 0.528, 04/04/90, Science Panel Backs NASA Satellite Plan, But Urges Launches of Smaller Probes
4. 0.526, 09/09/91, A NASA Satellite Project Accomplishes Incredible Feat: Staying Within Budget
5. 0.525, 07/24/90, Scientist Who Exposed Global Warming Proposes Satellites for Climate Research
6. 0.524, 08/22/90, Report Provides Support for the Critics Of Using Big Satellites to Study Climate
7. 0.516, 04/13/87, Arianespace Receives Satellite Launch Pact From Telesat Canada
+ 8. 0.509, 12/02/87, Telecommunications Tale of Two Companies

2.074 new 15.106 space
30.816 satellite 5.660 application
5.991 nasa 5.196 eos
4.196 launch 3.972 aster
3.516 instrument 3.446 arianespace
3.004 bundespost 2.806 ss
2.790 rocket 2.053 scientist
2.003 broadcast 1.172 earth
0.836 oil 0.646 measure

* 1. 0.513, 07/09/91, NASA Scratches Environment Gear From Satellite Plan
* 2. 0.500, 08/13/91, NASA Hasn't Scrapped Imaging Spectrometer
3. 0.493, 08/07/89, When the Pentagon Launches a Secret Satellite, Space Sleuths Do Some Spy Work of Their Own
4. 0.493, 07/31/89, NASA Uses 'Warm' Superconductors For Fast Circuit
* 5. 0.492, 12/02/87, Telecommunications Tale of Two Companies
6. 0.491, 07/09/91, Soviets May Adapt Parts of SS-20 Missile For Commercial Use
7. 0.490, 07/12/88, Gaping Gap: Pentagon Lags in Race To Match the Soviets In Rocket Launchers
8. 0.490, 06/14/90, Rescue of Satellite By Space Agency To Cost $90 Million

► **Figure 9.5** Example of relevance feedback on a text collection. Following the initial query, the user marks some relevant documents (shown with a plus sign). The query is then expanded by 18 terms with weights as shown. The revised top results are then shown. A * marks the documents which were judged relevant in the relevance feedback phase.



► **Figure 9.6** The Rocchio optimal query for separating relevant and non-relevant documents.

the set of non-relevant documents, then we wish to find:[1]

$$\vec{q} = \arg\max_{q}[\text{sim}(q, C_r) - \text{sim}(q, C_{nr})] \tag{9.1}$$

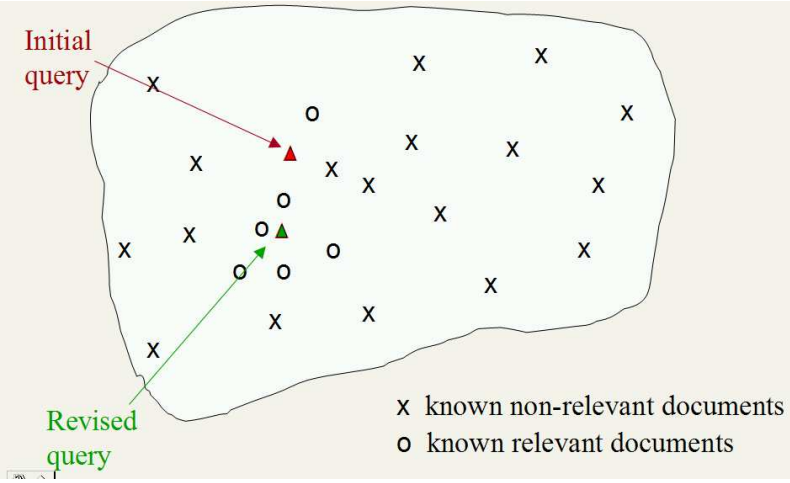
Under a model of cosine similarity, the optimal query vector $\vec{q}_{opt}$ for separating the relevant and non-relevant documents is:

$$\vec{q}_{opt} = \frac{1}{|C_r|}\sum_{\vec{d}_j \in C_r}\vec{d}_j - \frac{1}{|C_{nr}|}\sum_{\vec{d}_j \in C_{nr}}\vec{d}_j \tag{9.2}$$

That is, the optimal query is the difference between the centroids of the relevant and non-relevant documents; see Figure 9.6. However, this observation is not terrribly useful, precisely because the full set of relevant documents is not known: it is what we want to find.

ROCCHIO ALGORITHM **The Rocchio (1971) algorithm.** This was the relevance feedback mechanism introduced in and popularized by Salton's SMART system around 1970. In a real IR query context, we have a user query and partial knowledge of known relevant and irrelevant documents. The algorithm proposes using

1. In the equation, $\arg\max_x f(x)$ returns a value of x which maximizes the value of the function $f(x)$.



► **Figure 9.7** An application of Rocchio's algorithm. Some documents have been labeled as relevant and non-relevant and the initial query vector is moved in response to this feedback.

the modified query $\vec{q}_m$:

$$\vec{q}_m = \alpha\vec{q}_0 + \beta\frac{1}{|D_r|}\sum_{\vec{d}_j \in D_r}\vec{d}_j - \gamma\frac{1}{|D_{nr}|}\sum_{\vec{d}_j \in D_{nr}}\vec{d}_j \tag{9.3}$$

where q_0 is the original query vector, D_r and D_{nr} are the set of known relevant and non-relevant documents respectively, and α, β, and γ are weights attached to each term. These control the balance between trusting the judged document set versus the query: if we have a lot of judged documents, we would like a higher β and γ. Starting from q_0, the new query moves you some distance toward the centroid of the relevant documents and some distance away from the centroid of the non-relevant documents. This new query can be used for retrieval in the standard vector space model, as usual (see Chapter 7). Note that we can easily leave the positive quadrant of the vector space by subtracting off a non-relevant document's vector. In the Rocchio algorithm, negative term weights are ignored. That is, the term weight is set to 0. Figure 9.7 shows the effect of applying relevance feedback.

Relevance feedback can improve both recall and precision. But, in practice, relevance feedback has been shown to be most useful for increasing *recall* in situations where recall is important. This is partly because the technique expands the query, but it is also partly an effect of the use case: in this situation, users can be expected to take time to review results and to iterate

on the search. It is also the case that positive feedback turns out to be much more valuable than negative feedback, and so people set $\gamma < \beta$. For example, reasonable values might be $\alpha = 1$, $\beta = 0.75$, and $\gamma = 0.15$. In fact, many systems, such as the image search system shown above, allow only positive feedback, which is equivalent to setting $\gamma = 0$. Another alternative, is to use only the one highest-ranked non-relevant document as negative feedback (so $|D_n r| = 1$ in Equation (9.3)). While many of the experimental results comparing various relevance feedback variants are rather inconclusive,

IDE DEC-HI

some studies have suggested that this *Ide dec-hi* variant is the most effective or at least consistent (Ide 1971). In the other direction, another variant is to regard *all* documents in the collection apart from those judged relevant as non-relevant, rather than only ones that are explicitly judged non-relevant. However, Schütze et al. (1995) and Singhal et al. (1997) show that better results are obtained for routing by using only documents close to the query of interest rather than all documents.

Exercise 9.1

Why is positive feedback likely to be more useful than negative feedback to an IR system? Why might only using one non-relevant document be more effective than using several?

9.1.2 Probabilistic relevance feedback

Rather than reweighting the query in a vector space, if a user has told us some relevant and non-relevant documents, then we can proceed to build

CLASSIFIER

a *classifier*, such as with a Naive Bayes model (see Chapter 13). We can derive that the probability of a term t_k appearing in a document depending on whether it is relevant or not (expressed by the variable R) is:

$$P(t_k|R = true) = |D_{rk}|/|D_r| \tag{9.4}$$

$$P(t_k|R = false) = (N_k - |D_{rk}|)/(N - |D_r|) \tag{9.5}$$

where N is the total number of documents, N_k is the number that contain t_k, and D_{rk} is the number of known relevant documents containing t_k. Even though the set of known relevant documents is a perhaps small subset of the real set of relevant documents, assuming that the set of relevant documents is a small subset of the set of all documents, then the estimates given above will be reasonable.

At the end of the day, this gives another way of changing the query term weights. We will discuss such probabilistic approaches more in Chapters 11 and 13. But for the moment note that a disadvantage of this proposal is that the estimate uses only collection statistics and information about the term distribution within the documents judged relevant. It preserves no memory of the original query.

9.1.3 When does relevance feedback work?

The success of relevance feedback depends on certain assumptions. Firstly, the user has to have sufficient knowledge to be able to make an initial query which is at least somewhere close to the documents they desire. This is needed anyhow for successful information retrieval in the basic case, but it is important to see the kinds of problems that relevance feedback cannot solve alone. Cases where relevance feedback alone is not sufficient include:

- Misspellings. If the user spells a term in a different way to the way it is spelled in any document in the collection, then relevance feedback is unlikely to be effective. Rather one needs the spelling correction techniques of Chapter 3.

- Cross-language information retrieval. Documents in another language are not nearby in a vector space based on term distribution. Rather, documents in the same language cluster.

- Mismatch of searcher's vocabulary versus collection vocabulary. If the user searches for laptop but all the documents use the term notebook computer, then the query will fail, and relevance feedback is again most likely ineffective.

Secondly, the relevance feedback approach requires that relevance prototypes are well-behaved. Ideally, the term distribution in all relevant documents will be similar to that in the documents marked by the users, while the term distribution in all non-relevant documents will be different from those in relevant documents. Things will work well if all relevant documents are tightly clustered around a single prototype, or at least, if there are different prototypes, and relevant documents have significant vocabulary overlap, while similarities between relevant and irrelevant documents are small. Implicitly, the Rocchio relevance feedback model is treating relevant documents as a single *cluster*, which it models via the centroid of the cluster. So the approach does not work as well if there are several clusters. This can happen with:

- Subsets of the documents which use different vocabulary, such as Burma vs. Myanmar

- A query for which the answer set is inherently disjunctive, such as Pop stars who once worked at Burger King.

- Instances of a general concept, which often appear as a disjunction of more specific concepts. For example, *felines*.

Good editorial content in the collection can often provide a solution to this problem, for example if there is an article on the attitudes of different groups to the situation in Burma.

Relevance feedback is not necessarily popular with users. Users are often reluctant to provide explicit feedback, or in general do not wish to prolong the search interaction. Furthermore, it is often harder to understand why a particular document was retrieved after relevance feedback is applied.

Relevance feedback can also have practical problems. The long queries that are generated by straightforward application of relevance feedback techniques are inefficient for a typical IR engine. This results in a high compute cost to doing the retrieval and potentially long response times for the user. A partial solution to this is to only reweight certain prominent terms in the relevant documents, such as perhaps the top 20 terms by term frequency. Some experimental results have also suggested that using a limited number of terms like this may give better results (Harman 1992) though other work has suggested that using more terms is better in terms of retrieved document quality (Buckley et al. 1994b).

9.1.4 Relevance Feedback on the Web

Some web search engines offer a similar/related pages feature. This can be viewed as a particular simple form of relevance feedback. However, in general relevance feedback has been little used in web search. One exception was the Excite search engine, which initially provided full relevance feedback. However, the feature was in time dropped, due to lack of use. This is probably in part the general observation that on the web almost nobody uses advanced search interfaces and would like to complete their search in a single interaction. But it also probably reflects two other factors: relevance feedback is hard to explain to the average user, and relevance feedback is mainly a recall enhancing strategy, and web search users are almost never concerned with getting sufficient recall.

Spink et al. (2000) present results from the use of relevance feeback in the Excite engine. Only about 4% of query sessions from a user used the relevance feedback option, and these were usually exploiting the "More like this" link next to each result. But about 70% of users only looked at the first page of results and did not pursue things any further. So 4% is about one eighth of the people who did perform more than a minimal search. For people who used relevance feedback, results were improved about two thirds of the time.

Exercise 9.2

In Rocchio's algorithm, what weight setting for $\alpha/\beta/\gamma$ does a "Find pages like this one" search correspond to?

9.1.5 Evaluation of relevance feedback strategies

Interactive relevance feedback can give very substantial gains in retrieval performance (Salton 1989, Harman 1992, Buckley et al. 1994b).

There is some subtlety to evaluating the effectiveness of relevance feedback in a sound and enlightening way. The obvious first strategy is to start with an initial query q_0 and to compute a precision-recall graph. Following one round of feedback from the user, we compute the modified query q_m and again compute a precision-recall graph. Here, in both rounds we assess performance over all documents in the collection, which makes comparisons straightforward. And if you do this, you find spectacular gains from relevance feedback: gains on the order of 50% in mean average precision. But unfortunately it is cheating. The gains are partly due to the fact that known relevant documents (judged by the user) are now ranked higher. Fairness demands that we should only evaluate with respect to documents not seen by the user.

A second idea is to use documents in the *residual collection* (the set of documents minus those assessed relevant) for the second round of evaluation. This seems like a more realistic evaluation. Unfortunately, measures of performance can then often be lower than for the original query. This is particularly the case if there are few relevant documents, and so a fair proportion of them have been judged by the user in the first round. The relative performance of variant relevance feedback methods can be validly compared, but it is difficult to validly compare performance with and without relevance feedback because the collection size and the number of relevant documents changes from before the feedback to after it.

Thus neither of these methods is fully satisfactory. A third method is to have two collections, one which is used for the initial query and relevance judgements, and the second that is then used for comparative evaluation. The performance of both q_0 and q_m can be validly compared on the second collection. This scenario arises quite naturally in a document routing scenario where a user is assumed to have developed a query based on an existing document collection, and then the query will be applied to new documents as they come in. Such a scenario was explored in early TREC evaluations. Empirically, one round of relevance feedback is often very useful. Two rounds is sometimes marginally more useful.

9.1.6 Pseudo-relevance feedback

PSEUDO-RELEVANCE FEEDBACK
BLIND RELEVANCE FEEDBACK

Pseudo-relevance feedback, also known as *blind relevance feedback*, provides a method for automatic local analysis. It attempts to automate the manual part of relevance feedback, so that one gets improved retrieval performance without requiring an extended interaction with the user. The method is to

	Precision at $k = 50$	
Term weighting	no RF	pseudo RF
lnc.ltc	64.2%	72.7%
Lnu.ltu	74.2%	87.0%

► **Figure 9.8** Results showing pseudo relevance feedback greatly improving performance. These results are taken from the Cornell SMART system at TREC 4 (Buckley et al. 1996), and also contrast the use of two different length normalization schemes (L vs. l). Pseudo-relevance feedback consisted of adding 20 terms to each query.

do normal retrieval to find an initial set of most relevant documents, to then *assume* that the top m ranked documents are relevant, and finally to do relevance feedback as before under this assumption.

This automatic technique mostly works. Evidence suggests that it tends to work better than global analysis. It has been found to improve performance in the TREC ad-hoc task. See for example the results in Figure 9.8. But it is not without the dangers of an automatic process. For example, if the query is about copper mines and the top several documents are all about mines in Chile, then there may be query drift in that direction.

9.1.7 Indirect relevance feedback

IMPLICIT RELEVANCE FEEDBACK

One can also use indirect sources of evidence rather than explicit feedback on relevance as the basis for relevance feedback. This is often called *implicit (relevance) feedback*. Implicit feedback is less reliable than explicit feedback, but is more useful than pseudo relevance feedback, which contains no evidence of user judgements. Moreover, while users are often reluctant to provide explicit feedback, it is easy to collect implicit feedback in large quantities for a high volume system, such as a web search engine.

On the web, DirectHit introduced the idea of ranking more highly documents that users chose to look at more often. In other words, clicks on links were assumed to indicate that the page was likely relevant to the query. This approach makes various assumptions such as that the displayed summaries are equally good and about which parts of the page that the user has actually chosen to read. In the original DirectHit approach, this data about page goodness was gathered globally, rather than being user or query specific.

CLICKSTREAM MINING

This is one approach to the general area of *clickstream mining*.

9.1.8 Summary

Relevance feedback has been shown to be very effective at improving relevance of results. Requires enough judged documents, otherwise it's unstable

($\geq$ 5 is recommended). Requires queries for which the set of relevant documents is medium to large. Full relevance feedback is painful for the user. Full relevance feedback is not very efficient in most IR systems. Other types of interactive retrieval may improve relevance by as much with less work.

Other uses of relevance feedback include:

- Following a changing information need
- Maintaining an information filter (e.g., for a news feed)
- Active learning (deciding which examples it is most useful to know the class of to reduce annotation costs).

9.2 Global methods for query reformulation

9.2.1 Vocabulary tools for query reformulation

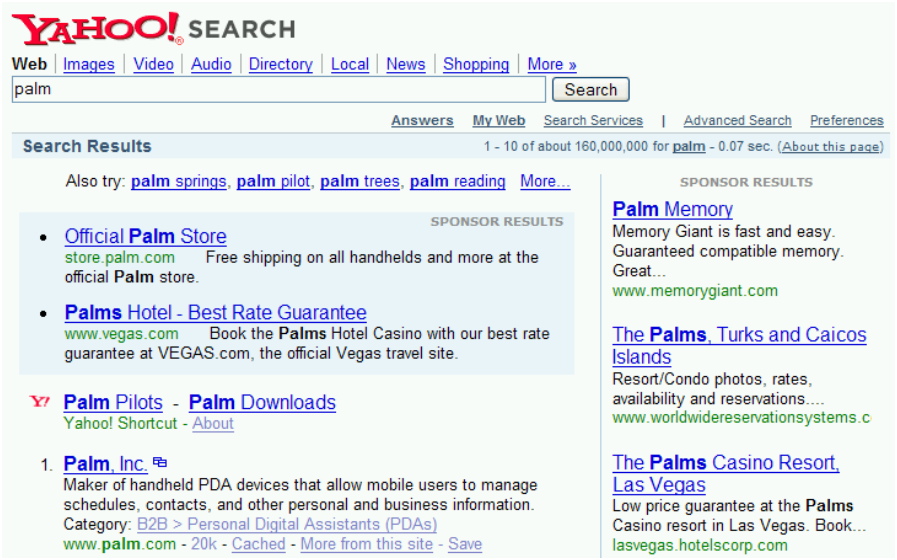
Various user supports in the search process are very helpful for the user in knowing how their search is or isn't working. This includes feedback information about words that were omitted from the query because they were on stop lists, what words were stemmed to, the number of hits on each term or phrase, and whether words were dynamically turned into phrases. Another source of information is the suggestion of search terms by means of a thesaurus or a controlled vocabulary. A user can also be allowed to browse lists of the terms that are in the inverted index, and thus find good terms that appear in the collection.

9.2.2 Query expansion

QUERY EXPANSION

In relevance feedback, users give additional input on documents (saying whether they are relevant/non-relevant), and this input is used to reweight the terms in the query for documents. In *query expansion*, users give additional input on query words or phrases, suggesting terms or saying whether they regard system suggestions as good or bad search terms. Figure 9.9 shows an example of query expansion options being presented in the Yahoo! web search engine. The central question in the use of query expansion is how to generate alternative or expanded queries for the user. The most common form of query expansion is global analysis, where by some means a form of thesaurus has been gathered. But it is also possible to do query expansion by local analysis, by analysing the documents in the result set. The distinction is again in whether the user is giving feedback on documents or the query terms.

Methods for building a thesaurus for query expansion include:



► **Figure 9.9** An example of query expansion in the interace of the Yahoo! web search engine in 2006. Note the expanded query suggestions appearing just below the "Search Results" bar.

- User query: `cancer`
- PubMed query: ("neoplasms"[TIAB] NOT Medline[SB]) OR "neoplasms"[MeSH Terms] OR cancer[Text Word]
- User query: `skin itch`
- PubMed query: ("skin"[MeSH Terms] OR ("integumentary system"[TIAB] NOT Medline[SB]) OR "integumentary system"[MeSH Terms] OR skin[Text Word]) AND (("pruritus"[TIAB] NOT Medline[SB]) OR "pruritus"[MeSH Terms] OR itch[Text Word])

► **Figure 9.10** Examples of query expansion via the PubMed thesaurus. When a user issues a query on the PubMed interface to Medline at http://www.ncbi.nlm.nih.gov/entrez/, their query is mapped on to the Medline vocabulary as shown.

- Use of a controlled vocabulary that is maintained by editors. Here, there are canonical terms for concepts. The subject headings of traditional library subject indices, such as the Library of Congress Subject Headings, or the Dewey Decimal system are examples of a controlled vocabulary. Use of a controlled vocabulary is quite common for well-resourced domains. A well known example is the Unified Medical Language System (UMLS) used with MedLine for querying the biomedical research literature. Examples are shown in Figure 9.10.
- A manual thesaurus. Here, people have built up sets of synonymous names for concepts. The UMLS metathesaurus is one example of a thesaurus. To take another, Statistics Canada maintains a thesaurus of preferred terms, synonyms, broader terms, and narrower terms for matters on which the government collects statistics, such as goods and services. This thesaurus is also bilingual English and French.
- An automatically derived thesaurus. Here, word co-occurrence statistics over a collection of documents in a domain are used to automatically induce a lexicon
- Query reformulations based on query log mining. Here, we exploit the manual query reformulations of other users to make suggestions to a new user. This requires a huge query volume, and is thus particularly appropriate to web search.

Thesaurus-based query expansion has the advantage of not requiring any user input. For each term, t, in a query, the query can be automatically expanded with synonyms and related words of t from the thesaurus. For example, in Figure 9.10, neoplasms was added to a search for cancer. Use of a thesaurus can be combined with ideas of term weighting: for instance, one might weight added terms less than original query terms. Use of query expansion generally increases recall and is widely used in many science/engineering fields. However, it may also significantly decrease precision, particularly when the query contains ambiguous terms. For example, if the user searches for interest rate, expanding the query to interest rate fascinate evaluate is unlikely to be useful. More seriously, there is a high cost to manually producing a thesaurus and then updating it for scientific changes. However, in general a domain-specific thesaurus is required: general thesauri and dictionaries give far too little coverage of the rich domain-particular vocabularies of most scientific fields.

9.2.3 Automatic thesaurus generation

As an alternative to the cost of a manual thesaurus, one can attempt to generate a thesaurus automatically by analyzing a collection of documents. There

Word	Nearest neighbors
absolutely	absurd, whatsoever, totally, exactly, nothing
bottomed	dip, copper, drops, topped, slide, trimmed
captivating	shimmer, stunningly, superbly, plucky, witty
doghouse	dog, porch, crawling, beside, downstairs
makeup	repellent, lotion, glossy, sunscreen, skin, gel
mediating	reconciliation, negotiate, case, conciliation
keeping	hoping, bring, wiping, could, some, would
lithographs	drawings, Picasso, Dali, sculptures, Gauguin
pathogens	toxins, bacteria, organisms, bacterial, parasite
senses	grasp, psyche, truly, clumsy, naive, innate

► **Figure 9.11** An example of an automatically generated thesaurus. This example is based on the work in Schütze (1998), which employs Latent Semantic Indexing (see Chapter 18).

are two main approaches. One is simply to exploit word cooccurrence and to say that co-occurring words are likely to be similar. The other is to use a shallow grammatical analysis of the text and to exploit grammatical relations or grammatical dependencies. That is, we say that, for example, entities that are grown, cooked, eaten, and digested, are more likely to be food items. Simply using word cooccurrence is more robust, but using grammatical relations is more accurate.

The simplest way to compute a co-occurrence thesaurus is based on term-term similarities, which can be derived from a term-document matrix A by calculating $C = AA^T$. If A_{ij} has a perhaps normalized weighted count w_{ij} for term t_i and document d_j, then C_{uv} has a similarity score between terms, with a larger number being better. Figure 9.11 shows an example of a thesaurus derived automatically in this way. While some of the thesaurus terms are good or at least quite suggestive, others are marginal or bad. The quality of the associations is quite typically a problem. Term ambiguity easily introduces irrelevant statistically correlated terms. A query for Apple computer may expand to Apple red fruit computer. In general one suffers from both false positives (words wrongly deemed similar) and false negatives. Moreover, since the terms in the automatic thesaurus are highly correlated in documents anyway (and often the same collection is used to derive the thesaurus as is being indexed), this form of query expansion may not retrieve many additional documents.

Query expansion is often effective in increasing recall. This is not always true with general thesauri, but it is fairly successful for subject-specific thesauri. In most cases, precision is decreased, often signficantly. Overall, query expansion is less successful than relevance feedback, though it may be as

good as pseudo-relevance feedback. It does, however, have the advantage of being much more understandable to the system user.

Exercise 9.3

If A is simply a Boolean cooccurrence matrix, then what do you get as the entries in C?

9.3 References and further reading

The main initial papers on relevance feedback using vector space models all appear in Salton (1971). Later work includes Salton and Buckley (1990) and the recent survey paper Ruthven and Lalmas (2003).

Use of clickthrough data on the web to provide indirect relevance feedback is studied in more detail in (Joachims 2002b, Joachims et al. 2005). The very successful use of web link structure (see Chapter 21) can also be viewed as implicit feedback, but provided by page authors rather than readers (though in practice most authors are also readers).

Qiu and Frei (1993) and Schütze (1998) discuss automatic thesaurus generation. Xu and Croft (1996) explore using both local and global query expansion.

10 *XML retrieval*

Information retrieval systems are often contrasted with relational databases. IR systems retrieve information from *unstructured text* – by which we mostly mean "raw" text without markup. Databases are designed for searching *structured data*: sets of records that have values for predefined attributes such as employee number, title and salary. Some highly structured search problems are best solved with a relational database, for example, if the employee table contains an attribute for short textual job descriptions. There are fundamental differences between information retrieval and database systems in terms of retrieval model, data structures and query language as shown in Table 10.1.[1]

There are two types of information retrieval problem that are intermediate between text retrieval and search over relational data. We discussed the first type, parametric search, in Section 6.1 (page 85). The second type, XML retrieval, is the subject of this chapter. We will view XML documents as trees that have *leaf nodes* containing text and *labeled internal nodes* that define the roles of the leaf nodes in the document. We call this type of text semistructured and retrieval over it *semistructured retrieval*. In the example in Figure 10.2, some of the leaves shown are Shakespeare, Macbeth, and Macbeth's castle, and the labeled internal nodes encode either the structure of the document (*title*, *act*, and *scene*) or metadata functions (*author*).

SEMISTRUCTURED RETRIEVAL

Semistructured retrieval has become increasingly important in recent years because of the growing use of *Extended Markup Language* or *XML*. XML is used for web content, for documents produced by office productivity suites, for the import and export of text content in general, and many other applications. These days, most semistructured data are encoded in XML. Here, we neglect the specifics that distinguish XML from other standards for semistructured data such as HTML and SGML.

EXTENDED MARKUP LANGUAGE XML

1. In most modern database systems, one can enable full-text search for text columns. This usually means that an inverted index is created and Boolean or vector space search enabled, effectively combining core database with information retrieval technologies.

	databases	information retrieval	semistructured retrieval
objects	record	unstructured document	tree with text at leaves
model	relational calculus	vector space & others	?
main data structure	table	inverted index	?
queries	SQL	text queries	?

► **Table 10.1** Databases, information retrieval and semistructured retrieval. There is no consensus yet as to what formal models, query languages and data structures are consistently successful for semistructured retrieval.

```
<play>
  <author>Shakespeare</author>
  <title>Macbeth</title>
  <act number="I">
    <scene number="VII">
      <title>Macbeth's castle</title>
      <verse>Will I with wine and wassail ...</verse>
    </scene>
  </act>
</play>
```

► **Figure 10.1** An XML document.

After presenting the basic concepts of XML, this chapter first discusses the challenges we face in semistructured retrieval. Then we describe JuruXML, a modification of the vector space model for XML retrieval. Section 10.4 presents INEX, a shared task evaluation that has been held for a number of years and currently is the most important venue for presenting XML retrieval research. Section 10.5 discusses some of the more "relational" aspects of XML.

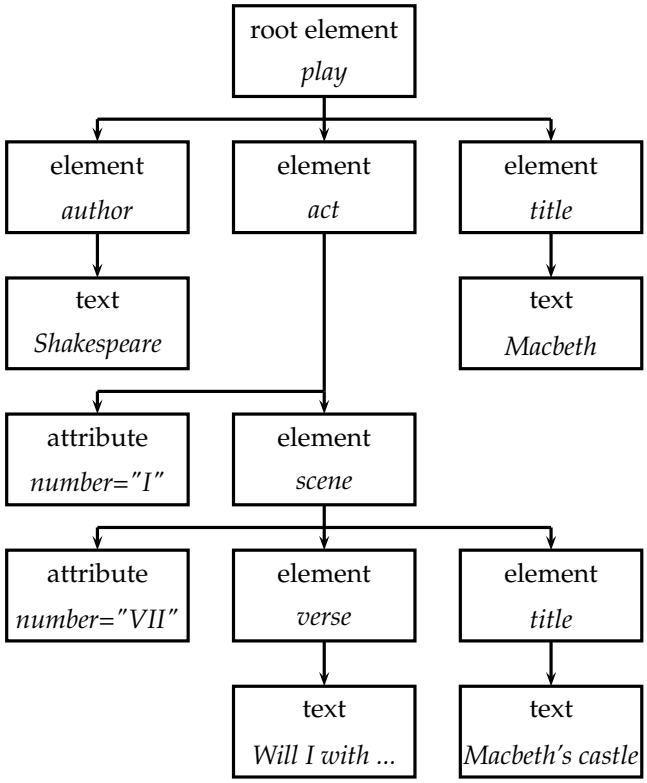
10.1 Basic XML concepts

XML ELEMENT
XML ATTRIBUTE

An XML document is an ordered, labeled tree. The nodes of the tree are *XML elements* and are written with an opening and closing *tag*. An element can have one or more *XML attributes*. One of the elements in the example XML document in Figure 10.1 is *scene*, which is enclosed by the two tags `<scene ...>` and `</scene>`. The element has an attribute *number* with value *VII* and two child elements, *title* and *verse*.

XML DOM

There is a standard way of accessing and processing XML documents, the XML Document Object Model or *DOM*. DOM represents elements, attributes



► **Figure 10.2** The XML document in Figure 10.1 as a DOM object. Parents point to their children.

and text within elements as nodes in a tree. Figure 10.2 shows the DOM representation of the XML document in Figure 10.1. With a DOM API, it is easy to process an XML document by starting at the root element and then descending down the tree from parents to children.

XPATH
XML CONTEXT

XPath is the standard for paths in XML. We will also refer to paths as *contexts* in this chapter. Only a small subset of XPath is needed for our purposes. The XPath expression `node` selects all nodes of that name. Successive elements of a path are separated by slashes, so `act/scene` selects all *scene* elements whose parent is an *act* element. Double slashes indicate that an arbitrary number of elements can intervene on a path: `play//scene` selects all *scene* elements occurring in a *play* element. In Figure 10.2 this set consists of the single *scene* element that is accessible via the path *play*, *act*, *scene* from the top. An initial slash starts the path at the root element. `/play/title` selects the play's title in Figure 10.1, `/play//title` selects the play's ti-

tle and the scene's title, and `/scene/title` selects no elements. For notational convenience, we allow the final element of a path to be a string, e.g. `title/"Macbeth"` for all titles containing the word Macbeth, even though this does not conform to the XPath standard.

SCHEMA

We also need the concept of *schema* in this chapter. A schema puts constraints on the structure of allowable XML documents for a particular application. A schema for Shakespeare's plays may stipulate that scenes can only occur as children of acts and that only acts and scenes have the *number* attribute. Two standards for schemas for XML documents are *XML DTD* (document type definition) and *XML Schema*. Users only can write structural queries for an XML retrieval system if they have some minimal knowledge about the schema of the underlying collection.

XML DTD

XML SCHEMA

10.2 Challenges in semistructured retrieval

In Chapter 2 (page 18) we briefly discussed the need of a document unit in indexing and retrieval. In unstructured retrieval, it is usually clear what the right document unit is: files on your desktop, email messages, web pages on the web etc. The first challenge in semistructured retrieval is that we don't have such a natural document unit or *indexing unit*. If we query Shakespeare's plays for Macbeth's castle, should we return the scene, the act or the whole play in Figure 10.2? In this case, the user is probably looking for the scene. On the other hand, an otherwise unspecified search for Macbeth should return the play of this name, not a subunit. One decision criterion that has been proposed for selecting the most appropriate part of a document is the *structured document retrieval principle*:

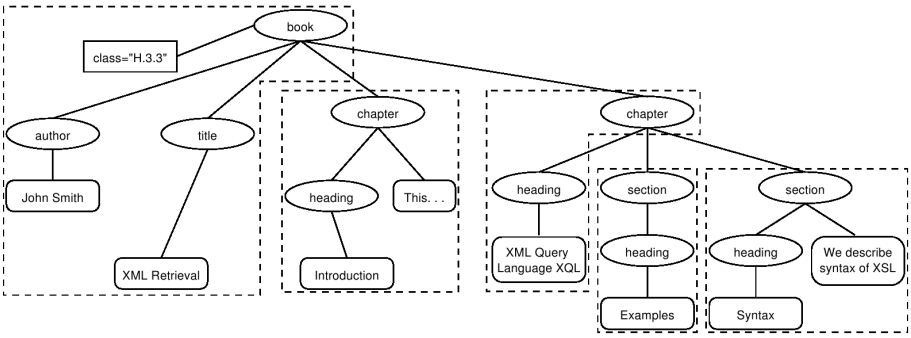
INDEXING UNIT

STRUCTURED DOCUMENT RETRIEVAL PRINCIPLE

> **Structured document retrieval principle.** A system should always retrieve the most specific part of a document answering the query.

This principle motivates a retrieval strategy that returns the smallest unit that contains the information sought, but does not go below this level. However, it can be hard to implement this principle algorithmically. Consider the query `title/"Macbeth"` applied to Figure 10.2. The title of the tragedy, *Macbeth*, and the title of Act 1, Scene 7, *Macbeth's castle*, are both good hits in terms of term matching. But in this case, the title of the tragedy, the higher node, is preferred. Deciding which level of the tree is right for answering a query is tricky.

There are at least three different approaches to defining the indexing unit in XML retrieval. One is to index all components that are eligible to be returned in a search result. All subtrees in Figure 10.1 meet this criterion, but typographical XML elements as in `<b>definitely</b>` or an ISBN number without context may not. This scheme has the disadvantage that search



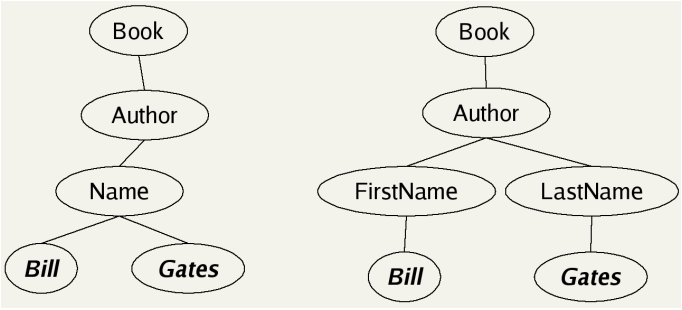
► **Figure 10.3** Indexing units in XML retrieval. Unlike conventional retrieval, XML retrieval does not have a natural indexing unit. In this example, books, chapters and sections have been designated to be indexing units, but without overlap. For example, the leftmost dashed indexing unit contains only those parts of the tree dominated by book that are not already part of other indexing units.

results will contain overlapping units that have to be filtered out in a post-processing step to reduce redundancy.

Another approach is to group nodes into non-overlapping pseudodocuments as shown in Figure 10.3. This avoids the overlap problem, but pseudodocuments may not make intuitive sense to the user. And they have to be fixed at indexing time, leaving no flexibility to answer queries at a more specific or more general level.

The third approach is to designate one XML element as the substitute for the document unit. This is the approach taken by the system in the next section where the document collection is a collection of articles from IEEE journals and each component dominated by an article node is treated as a document. As with the node groups in Figure 10.3, we have the problem that indexing units are fixed. However, we can attempt to extract the most relevant subcomponent from each hit in a postprocessing step.

The lack of a clear indexing unit is related to another challenge in XML retrieval: We need to distinguish different contexts of a term when we compute term statistics for ranking, in particular inverse document frequency (idf) statistics (Section 6.2.1, page 88). For example, the term Gates under the node Author is unrelated to an occurrence under a content node like Section if used to refer to the plural of gate. Computing a single document frequency for all occurrences of Gates is problematic unless the distribution of terms is similar in all content nodes. The simplest solution is to compute idf for term-context pairs (or *structural terms*). So we would compute different idf weights for `author/"Gates"` and `section/"Gates"`. Unfortunately, this



► **Figure 10.4** A schema mismatch. A difficulty in XML retrieval is that a query may not conform to the schema of the collection. Here, no exact match is possible between query (left tree) and document (right tree) because the Name node in the query has no corresponding node in the document.

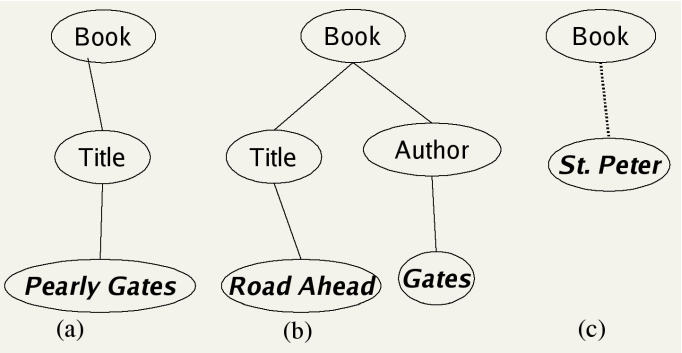
scheme will run into sparse data problems, a problem we will return to in Section 10.4 when discussing the Merge algorithm.

The schemas of XML documents in a collection can vary since semistructured documents often come from different sources. This presents yet another challenge. Comparable nodes may have different names and be represented differently structurally. In Figure 10.4, the node Name in the query (the left tree) corresponds to the two nodes FirstName and LastName in the document (the right tree). Some form of approximate matching of node names in combination with semi-automatic matching of different document structures can help here. Human editing of correspondences of nodes in different schemas will usually do better than completely automatic methods.

Schemas also pose a challenge for user interfaces if users are not familiar with the structure and naming conventions of the document collection. Consider the queries in Figure 10.5. For query (b) the user has made the assumption that the node referring to the creator of the document is called Author, but it could also be named Writer or Creator (the latter being the choice of the Dublin Core Metadata standard). Query (c) is a search for books that contain St. Peter anywhere. We will call them *extended queries* here since they will match with documents containing any number of nodes between Book and St. Peter. This corresponds to the double slash in XPath notation: `book//"St. Peter"`. It is quite common for users to issue such extended queries without specifying the exact structural configuration in which a query term should occur.

EXTENDED QUERIES

The user interface should expose the tree structure of the collection and allow users to specify the nodes they are querying. As a consequence the query interface is more complex than a search box for keyword queries in unstruc-



► **Figure 10.5** XML queries. Simple XML queries can be represented as trees. All documents matching the query tree are considered hits. The queries shown here search for books with the words Pearly Gates in the title (a); books with titles containing Road Ahead and authors containing Gates (b); and books that contain St. Peter in any node (c). The dashed line in the *extended query* (c) represents an arbitrary path (without restriction on the type of nodes on the path) between Book and St. Peter. The XPath notation for this query is `book//"St. Peter"`.

tured retrieval. This is one of the challenges currently being addressed by the research community.

10.3 A vector space model for XML retrieval

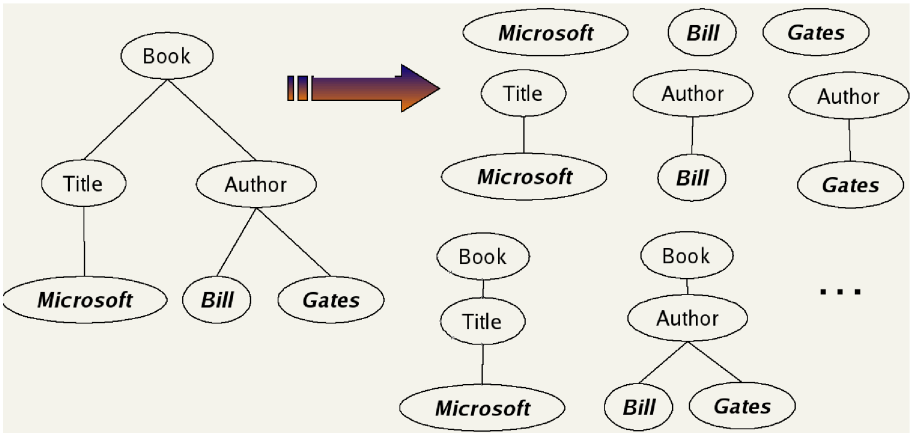
Unlike unstructured retrieval, XML retrieval requires taking into consideration the structural context of terms. A document authored by Bill Gates should match the second query in Figure 10.5, but not the first. We want to make use of the vector space model to represent this structural context. In unstructured retrieval, there would be a single axis for Gates. In XML retrieval, we must separate the title word Gates from the author name Gates. Thus, the axes must represent not only words, but also their position within the XML tree.

JURUXML

One of the first systems that took the vector space approach to XML retrieval was *JuruXML*. The basic document representation adopted in JuruXML is shown in Figure 10.6. The dimensions of the vector space are defined to be subtrees of documents that contain at least one lexicon term. We call a

STRUCTURAL TERM

subtree that functions as an axis a *structural term*. A subset of the possible structural terms is shown in the figure, but there are others (e.g., the subtree corresponding to the whole document with the leaf node Gates removed). There is a tradeoff between index size and accuracy of query results. In general, query results will only be completely accurate if we index all subtrees



► **Figure 10.6** A mapping of an XML document (left) to a set of structural terms (right).

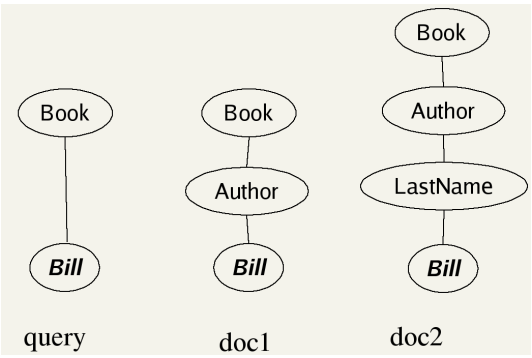
(Exercise 10.1), but this results in a very large index. A compromise is to index all paths that end in a single lexicon term. The document in the figure would then have 9 structural terms.

We can treat structural terms just like regular terms in ordinary vector space retrieval. For instance, we can compute term frequency weights and perform stemming and case folding. To compute idf weights for structural terms, we need a document unit. JuruXML assumes that there is such a document unit, e.g., the article or scene elements.

We also want to weight contexts in the query. Users often care more about some parts of the query than others. In query (b) in Figure 10.5, the user may want to give more weight to the author because she is not sure whether she remembers the title correctly. This weighting can either be done in the user interface as part of query input; or by the system designer in cases where the importance of different parts of the query is likely to be the same across users.

In JuruXML, it is assumed that all queries are extended ("double slash") queries. If a query can be made to match a document by inserting additional nodes as in Figure 10.7, then the document is a potential hit. We interpret every query as an extended query because requiring a decision for each part of the query tree as to whether the corresponding link is intended as exact or extended would put too much of a burden on the user.

Even if we allow extended matches, we still prefer documents that match the query structure closely. We ensure that retrieval results respect this preference by computing a weight for each match. A simple measure of the



► **Figure 10.7** Query-document matching for extended queries. Extended queries match a document if the query path can be transformed into the document path by insertion of additional nodes. The context resemblance function cr is a measure of how similar two paths are. Here we have $\text{cr}(q, d_1) = 3/4 = 0.75$ and $\text{cr}(q, d_2) = 3/5 = 0.6$.

CONTEXT RESEMBLANCE

"goodness of fit" between two paths is the following *context resemblance* function cr:

$$\text{cr}(q, d) = \frac{1 + |q|}{1 + |d|}$$

where q and d are the number of nodes in the query path and document path, respectively. The context resemblance function returns 0 if the query path cannot be extended to match the document path. Its value is 1.0 if q and d are identical. Two additional examples are given in Figure 10.7.

The query processing component needs to identify those structural terms in the dictionary that have a non-zero match with the query terms. This step is analogous to tolerant retrieval in unstructured retrieval that also requires a mapping from query terms to index terms (see Chapter 3).

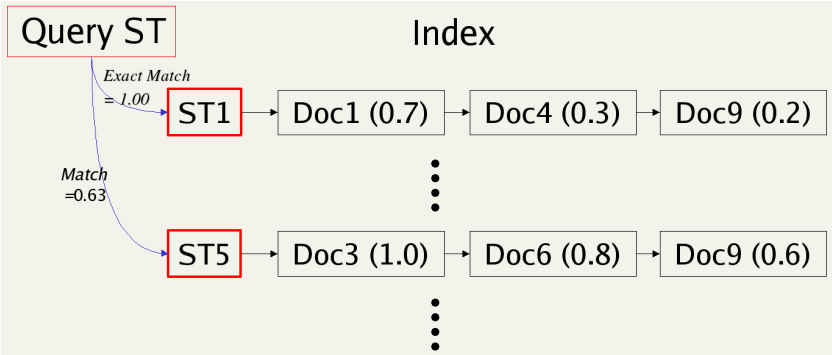
CONTEXT RESEMBLANCE SIMILARITY

The final score for a document is computed as a variant of the cosine measure (Equation (7.1), page 98). The *context resemblance similarity* between query and document is defined as:

$$\text{sim-cr}(q, d) = \frac{\sum_{t \in V} \sum_{c_k \in C} \sum_{c_l \in C} \text{weight}(q, t, c_k) \text{weight}(d, t, c_l) \text{cr}(c_k, c_l)}{|\vec{q}||\vec{d}|} \tag{10.1}$$

where V is the vocabulary of (non-structural) terms; C is the set of all contexts (paths) occurring in the collection and the queries; and $\text{weight}(d, t, c_k)$ is the weight of term t in context c_k in document d.

An example of an inverted index search in extended query matching is given in Figure 10.8. The structural term ST in the query occurs as ST1 in the



► **Figure 10.8** Inverted index search for extended queries.

Indexing
For each indexing unit i:
  Compute structural terms for i
Construct index
Search
Compute structural terms for query
For each structural term t:
  Find matching structural terms in dictionary
  For each matching structural term t':
    Compute matching weight $\text{cr}(t, t')$
Search inverted index with computed terms and weights
Return ranked list

► **Figure 10.9** Indexing and search in JuruXML.

index and has a match coefficient of 0.63 with a second term ST5 in the index. In this example, the highest ranking document is Doc9 with a similarity of $1.0 \times 0.2 + 0.63 \times 0.6 = 0.578$. Query weights are assumed to be 1.0.

Figure 10.9 summarizes indexing and query processing in JuruXML.

NOMERGE

MERGE

Idf weights for the just introduced retrieval algorithm are computed separately for each structural term. We can call this strategy *NoMerge* since contexts of different structural terms are not merged. An alternative is a *Merge* strategy that computes statistics for term (t, c) by collapsing all contexts c' that have a non-zero context resemblance with c. So for computing the document frequency of the structural term (atl,recognition), occurrences of recognition in contexts `fm/atl`, `article//atl` etc. would also be counted. This scheme addresses the sparse data problems that occur when computing idf weights for structural terms.

12,107	number of documents
494 MB	size
1995–2002	time of publication of articles
1,532	average number of XML nodes per document
6.9	average depth of a node
30	number of CAS topics
30	number of CO topics

► **Table 10.2** INEX 2002 collection statistics. There are two types of topics: content only (CO) and content and structure (CAS). An example of a CAS topic is given in Figure 10.11.

A second innovation in the Merge run is the use of a modified similarity function sim-qc, which replaces sim-cr when computing cosine similarity:

$$\text{sim-qc}(q,d) = \frac{\sum_{t\in V}\sum_{c\in C}\text{weight}(q,t)\text{weight}(d,t)\text{qc}(c)}{|\vec{q}||\vec{d}|} \tag{10.2}$$

where $\text{qc}(c) = |c| + 1$. So weight is a linear function of the path length or specificity of the context.

An evaluation of Merge and NoMerge and a comparison to other methods are the subject of the next section.

10.4 Evaluation of XML Retrieval

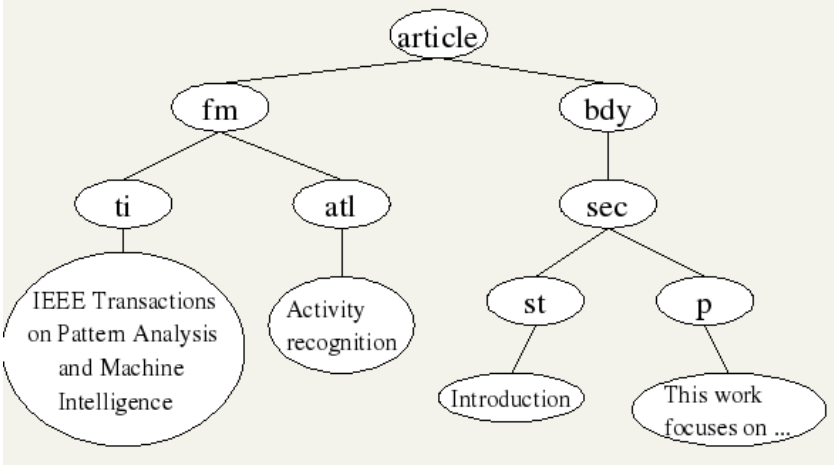
INEX

A large part of academic research on XML retrieval is conducted within the *INEX* (**IN***itiative for the* **E***valuation of* **X***ML retrieval*) program, a collaborative effort that includes reference collections, sets of queries, relevance judgments and a yearly meeting to present and discuss research results. What TREC is to unstructured information retrieval, INEX is to XML retrieval. We will learn in this section how a reference collection for XML retrieval can be set up, what evaluation measures have been proposed and what level of performance to expect on these measures.

The INEX 2002 collection consists of about 12,000 articles from IEEE journals. We give collection statistics in Table 10.2 and show the structure of the documents in Figure 10.10.

TOPICS
CO TOPICS
CAS TOPICS

There are two types of queries, called *topics*, in INEX: content-only or CO topics and content-and-structure or CAS topics. *CO topics* are regular keyword queries as in unstructured information retrieval. *CAS topics* have structural constraints in addition to keywords as shown in the example in Figure 10.11. These two different components of CAS queries make relevance assessments more complicated than in unstructured retrieval. INEX defines



► **Figure 10.10** Schema of the documents in the INEX collection. The element tags are **f**ront **m**atter (fm), **ti**tle (ti), **a**rticle **ti**tl**e** (atl), **b**o**dy** (bdy), **sec**tion (sec), **s**ection **t**itle (st) and **p**aragraph (p). Not shown is **ti**tle **g**roup (tig).

```
<te>article</te>
<cw>non-monotonic reasoning</cw>   <ce>bdy/sec</ce>
<cw>1999 2000</cw>                 <ce>hdr//yr</ce>
<cw>-calendar</cw>                 <ce>tig/atl</ce>
```

► **Figure 10.11** An INEX CAS topic. The topic specifies a search for articles on non-monotonic reasoning from 1999 or 2000 that are not calendars of events. The first line specifies that retrieved elements should be articles (te = target element). The other three lines are pairs of content word (cw) and content element (ce) conditions, indicating the content words that should occur (non-monotonic reasoning, 1999, 2000) or should not occur (calendar) in a particular context.

COMPONENT COVERAGE
TOPICAL RELEVANCE

component coverage and *topical relevance* as orthogonal dimensions of relevance. The component coverage dimension evaluates whether the element retrieved is "structurally" correct, i.e., neither too low nor too high in the tree. We distinguish four cases:

- Exact coverage (E). The information sought is the main topic of the component and the component is a meaningful unit of information.
- Too small (S). The information sought is the main topic of the component, but the component is not a meaningful (self-contained) unit of information.

- Too large (L). The information sought is present in the component, but is not the main topic.
- No coverage (N). The information sought is not a topic of the component.

The topical relevance dimension also has four levels: highly relevant (3), fairly relevant (2), marginally relevant (1) and irrelevant (0). Components are judged on both dimensions and the judgments are then combined into a digit-letter code. 2S is a fairly relevant component that is too small and 3E is a highly relevant component that has exact coverage. In theory, there are 16 combinations of coverage and relevance, but many cannot occur. For example, a non-relevant component cannot have exact coverage, so the combination 3N is not possible.

The relevance-coverage combinations are then "quantized" as follows:

$$\mathbf{q}(rel, cov) = \begin{cases} 1.00 & \text{if} \quad (rel, cov) = 3\text{E} \\ 0.75 & \text{if} \quad (rel, cov) \in \{2\text{E}, 3\text{L}\} \\ 0.50 & \text{if} \quad (rel, cov) \in \{1\text{E}, 2\text{L}, 2\text{S}\} \\ 0.25 & \text{if} \quad (rel, cov) \in \{1\text{S}, 1\text{L}\} \\ 0.00 & \text{if} \quad (rel, cov) = 0\text{N} \end{cases}$$

This evaluation scheme takes account of the fact that yes/no relevance judgements are less appropriate for XML retrieval than for unstructured retrieval. A 2S component provides incomplete information and may be difficult to interpret without more context, but it does answer the query partially. Quantization of relevance-coverage combinations avoids a binary choice and lets us grade it as "half-relevant". The number of relevant components in a retrieved set C of components can then be computed as:

$$\#(\text{relevant items retrieved}) = \sum_{c \in C} \mathbf{q}(rel(c), cov(c))$$

As an approximation, the standard definitions of precision, recall and F from Chapter 8 can be applied to this modified definition of relevant items retrieved, with some subtleties because we sum "fractional" relevance assessments. See the references given in Section 10.6 for further discussion.

One flaw of measuring relevance this way is that overlap is not accounted for. We discussed the concept of marginal relevance in the context of unstructured retrieval in Chapter 8 (page 123). This problem is worse in XML retrieval than unstructured retrieval because the same component can occur multiple times in a ranking as part of different higher-level components. The play, act, scene and title components on the path between the root node and Macbeth's castle in Figure 10.1 can all be returned in a result set, so that the leaf node occurs four times.

avg prec	organization	run ID	approach
0.271	IBM Haifa Labs	Merge	JuruXML
0.263	University of Michigan	Allow-duplicate	DB extended with Tree Algebra
0.242	IBM Haifa Labs	NoMerge	JuruXML
0.177	University of Amsterdam	UAmsI02NGram	vector space system

► **Table 10.3** Selected INEX results for content-and-structure (CAS) queries and the quantization function **q**.

Table 10.3 shows four runs from the three top-ranked groups at INEX 2002. The Merge and NoMerge runs are from the JuruXML system as described in the last section. The submission Allow-duplicate is from a database system that was extended with the capability of storing and searching for trees. UAmsI02NGram uses a vector space retrieval approach with pseudo-relevance feedback and applies structural XML constraints in a postfiltering step. The best run is the Merge run, which incorporates fewer structural constraints than the other systems and mostly relies on keyword matching. The fact that it does so well demonstrates the challenge of XML retrieval: Methods for structural matching need to be designed and executed well to achieve an improvement compared to unstructured retrieval methods.

The average precision numbers in Table 10.3 are respectable, but there obviously remains a lot of work to be done. Further analysis presented in the INEX 2002 proceedings shows that the best systems get about 50% of the first 10 hits right. However, this is only true on average. The variation across topics is large. The best run, Merge, had a median of 0.147, so performance for most topics is poor. A smaller number of topics, about a quarter, is handled well by Merge, with average precision higher than 0.5. These numbers are meant to give the reader a sense of what level of performance to expect from XML retrieval systems.

10.5 Text-centric vs. structure-centric XML retrieval

TEXT-CENTRIC XML RETRIEVAL

In JuruXML, XML structure serves as a framework within which conventional text matching is performed, exemplifying the *text-centric* approach to XML retrieval. While both structure and text are important, we give higher priority to text matching. We adopt unstructured retrieval methods to handle additional structural constraints.

STRUCTURE-CENTRIC XML RETRIEVAL

In contrast, *structure-centric XML retrieval* (as exemplified by the University of Michigan system) puts the emphasis on the structural aspects of a user query. A clear example is: "chapters of books that have the same title as the book". This query has no text component. It is purely structural.

Text-centric approaches are appropriate for data that are essentially text documents, marked up as XML to capture document structure. This is becoming a de facto standard for publishing text databases since most text documents have some form of interesting structure (paragraphs, sections, footnotes etc.). Examples include assembly manuals, journal issues, Shakespeare's collected works and newswire articles.

Structure-centric approaches are commonly used for data collections with complex structures that contain text as well as non-text data. A text-centric retrieval engine will have a hard time dealing with proteomic data that may be part of a biomedical publication – or with the representation of a street map that (together with street names and other textual descriptions) forms a navigational database.

We have treated XML retrieval here as an extension of conventional text retrieval: text fields are long (e.g., sections of a document), we must support inexact matches of paths and words and users want relevance-ranked results. Relational databases do not deal well with this "use case".

We have concentrated on applications where XML is used for little more than encoding a tree. For these applications, text-centric approaches suffice. But in reality, XML is a much richer representation formalism. In particular, attributes and their values usually have a database flavor and are best handled by a structure-centric approach or by parametric search.

Two other types of queries that cannot be handled in a vector space model are joins and ordering constraints. The following query requires a join:

> Find figures that decribe the Corba architecture and the paragraphs that refer to those figures.

This query imposes an ordering constraint:

> Retrieve the chapter of the book Introduction to algorithms that follows the chapter Binomial heaps.

The Corba query requires a join of paragraphs and figures. The Binomial heap query relies on the ordering of nodes in XML, in this case the ordering of chapter nodes underneath the book node. There are powerful query languages for XML that can handle attributes, joins and ordering constraints. The best know of these is XQuery, a language proposed for standardization by the W3C. It is designed to be broadly applicable in all areas where XML is used. At the time of this writing, little research has been done on testing the ability of XQuery to satisfy the demands of typical information retrieval settings. Efficiency is a major concern in this regard. Due to its complexity, it is challenging to implement an XQuery-based information retrieval system with the performance characteristics that users have come to expect.

Relational databases are better equipped to handle many structural constraints, particularly joins. But ordering is also difficult in a database frame-

work – the tuples of a relation in the relational calculus are not ordered. Still, many structure-centric XML retrieval systems are extensions of relational databases. If text fields are short, exact matches for paths and text are desired and retrieval results in form of unordered sets are ok, then a relational database is sufficient.

10.6 References and further reading

There are many good introductions to XML, including (Harold and Means 2004). The structured document retrieval principle is due to Fuhr and Großjohann (2001). JuruXML is presented in (Carmel et al. 2003). Section 10.4 follows the overview of INEX 2002 by Gövert and Kazai (2003), published in the proceedings of the meeting (Fuhr et al. 2003). The proceedings also contain papers by IBM Haifa Laboratory (Mass et al. 2002), the University of Amsterdam (Kamps et al. 2002), and the University of Michigan (Yu et al. 2002) with further details on the runs submitted by these groups.

The survey of automatic schema matching given by Rahm and Bernstein (2001) for databases is also applicable to XML.

The proposed standard for XQuery can be found at http://www.w3.org/TR/xquery/.

10.7 Exercises

Exercise 10.1

We discussed the tradeoff between accuracy of results and index size in vector space XML retrieval. If we only index structural terms that are paths ending in lexicon terms, what type of query can we not answer correctly?

Exercise 10.2

If we only index structural terms that are paths ending in lexicon terms, how many structural terms does the document in Figure 10.1 yield?

Exercise 10.3

If we only index structural terms that are paths ending in lexicon terms, what is the size of the index as a function of text size?

Exercise 10.4

If we index all structural terms (i.e., all subtrees), what is the size of the index as a function of text size?

11 *Probabilistic information retrieval*

We noted in Section 9.1.2 that if we have some known relevant and non-relevant documents, then we can straightforwardly start to estimate the probability of a term appearing in a relevant document $P(t_j|R = 1)$, and that this could be the basis of a classifier that determines whether documents are relevant or not. In this chapter, we more systematically introduce this probabilistic approach to IR, which provides a different retrieval model, with different techniques for setting term weights, and a different way of thinking about modelling information retrieval.

There is more than one possible retrieval model which has a probabilistic basis. Here, we will introduce probability theory for IR and the Probability Ranking Principle (Sections 11.1–11.2), and then concentrate on the *Binary Independence Model* (Section 11.3), which is the original and still most influential probabilistic retrieval model. Finally, we will briefly touch on related but extended methods using term counts, including the empirically successful Okapi BM25 weighting scheme, and Bayesian Network models for IR (Section 11.4). In the next chapter, we then present the alternate probabilistic language modeling approach to IR, which has been developed with considerable success in recent years.

11.1 Probability in Information Retrieval

In this section, we motivate the use of probabilistic methods in IR, and review a few probability basics. In information retrieval, the user starts with an *information need*, which they translate into a *query representation*. Similarly, there are *documents*, each of which has been converted into a *document representation* (the latter differing at least by how text is tokenized and so on, but perhaps containing fundamentally less information, as when a non-positional index is used). Based on the two representations, we wish to determine how well a document satisfies an information need. In the Boolean or vector space models of IR, matching is done in a formally defined but semantically im-

precise calculus of index terms. Given only a query, an IR system has an uncertain understanding of the original information need. And given the query and document representations, a system also has an uncertain guess of whether a document has content relevant to a query. The whole idea of probability theory is to provide a principled foundation for reasoning under uncertainty. This chapter provides one answer as to how to exploit this foundation to estimate how likely it is that a document is relevant to a query.

We hope that the reader has seen a little basic probability theory previously, but to very quickly review, for events a and b, there is the concept of a joint event $P(a, b)$, where both are true, and a conditional probability, such as $P(a|b)$, the probability of event a given that event b is true. Then the fundamental relationship between joint and conditional probabilities is given by:

$$P(a, b) = p(a \cap b) = P(a|b)P(b) = P(b|a)P(a) \tag{11.1}$$

Similarly,

$$P(a, b) = P(b|a)P(a) \tag{11.2}$$

$$P(\neg a, b) = P(b|\neg a)P(\neg a) \tag{11.3}$$

BAYES' RULE From these we can derive *Bayes' Rule* for inverting conditional probabilities:

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)} = \frac{P(b|a)}{\sum_{x \in \{a, \neg a\}} P(b|x)P(x)} P(a) \tag{11.4}$$

This equation can also be thought of as a way of updating probabilities: we start with a prior probability $P(a)$ and derive a posterior probability $P(a|b)$ after having seen the evidence b, based on the likelihood of b occurring. Finally, it is often useful to talk about the *odds* of an event, which provide a kind of multiplier for how probabilities change:

ODDS

$$\text{Odds: } O(a) = \frac{P(a)}{P(\neg a)} = \frac{P(a)}{1 - P(a)} \tag{11.5}$$

11.2 The Probability Ranking Principle

Let d be a document in the collection. Let R represent that the document is relevant with respect to a given query q, and let NR represent non-relevance. (This is the traditional notation. Using current notation for probabilistic models, it would be more standard to have a binary random variable $R_{d,q}$ for relevance to a particular query q, and then R would represent $R_{d,q} = 1$ and NR would represent $R_{d,q} = 0$.)

We assume the usual ranked retrieval setup, where there is a collection of documents, the user issues a query, and an ordered list of documents needs

to be returned. In this setup, the ranking method is the core of the IR system: in what order do we present the documents to the user. In a probabilistic model, the obvious answer to this question is to rank documents by the estimated probability of their relevance with respect to the information need. That is, we order documents d by $P(R|d,q)$.

PROBABILITY RANKING PRINCIPLE

This is the basis of the *Probability Ranking Principle* (PRP) (van Rijsbergen 1979, 113–114):

> "If a reference retrieval system's response to each request is a ranking of the documents in the collection in order of decreasing probability of relevance to the user who submitted the request, where the probabilities are estimated as accurately as possible on the basis of whatever data have been made available to the system for this purpose, the overall effectiveness of the system to its user will be the best that is obtainable on the basis of those data."

In the simplest case of the PRP, there are no selection costs or other utility concerns that would differentially weight actions or errors. The PRP in action then says to simply rank all documents by $P(R|x,q)$. In this case the Bayes Optimal Decision Rule is to use a simple even odds threshold:

$$x \text{ is relevant iff } P(R|x) > P(NR|x) \tag{11.6}$$

Theorem 11.1 *Using the PRP is optimal, in the sense that it minimizes the loss (Bayes risk) under 1/0 loss.*
Proof. *Ripley (1996). The proof assumes that all probabilities are known correctly, which is never the case in practice.*

In more complex models involving the PRP, we assume a model of retrieval costs. Let C_r be the cost of retrieval of a relevant document and C_{nr} the cost of retrieval of a non-relevant document. Then the Probability Ranking Principle says that if for a specific document x and for all documents x' not yet retrieved

$$C_r \cdot P(R|x) + C_{nr} \cdot P(NR|x) \leq C_r \cdot P(R|x') + C_{nr} \cdot P(NR|x') \tag{11.7}$$

then x is the next document to be retrieved. For the rest of this chapter, we will stick to the simple 1/0 loss case, and not further consider loss/utility models, but simply note that these give a formal framework where we can model differential costs of false positives and false negatives at the modeling stage, rather than simply at the evaluation stage, as discussed in Chapter 8.

To make a probabilistic retrieval strategy precise, we need to estimate how terms in documents contribute to relevance. Becoming more specific, we wish to know how statistics we can compute like term frequency, document frequency, and document length influence judgements about document relevance, and how they can be reasonably combined to estimate the probability

of document relevance. We then order documents by decreasing estimated probability of relevance.

What we wish to find is the probability $P(R|x)$ that a document x is relevant. Using Bayes Rule, we have that:

$$P(R|x) = \frac{P(x|R)P(R)}{P(x)} \tag{11.8}$$

$$P(NR|x) = \frac{P(x|NR)P(NR)}{P(x)} \tag{11.9}$$

Here, $P(x|R)$ and $P(x|NR)$ are the probability that if a relevant (or non-relevant, respectively) document is retrieved, then it is x. And $P(R)$ and $P(NR)$ indicate the prior probability of retrieving a relevant or non-relevant document respectively. If we knew the percentage of relevant documents in the collection, then we could estimate these quantities. Since a document must be either relevant or non-relevant to a query, we have that:

$$P(R|x) + P(NR|x) = 1$$

How do we compute all these probabilities? We never know the exact probabilities, and so we have to use estimates. The Binary Independence Model, which we present in the next section, makes some simple assumptions so as to provide easy ways of calculating the needed estimates. But before presenting this model in detail, let us note a few of the questionable assumptions of the model so far:

- Document relevance is again assumed to be independent. The relevance of each document is independent of the relevance of other documents. As we have noted, this is incorrect: it is especially harmful in practice if it allows a system to return duplicate or near duplicate documents.
- We are again working with just a Boolean model of relevance.
- The model assumes that the user has a single step information need. As discussed in Chapter 9, seeing a range of results might let the user refine their information need. Fortunately, as mentioned there, it is straightforward to extend the Binary Independence Model so as to provide a framework for relevance feedback, and we briefly present this model below.

11.3 The Binary Independence Model

BINARY INDEPENDENCE MODEL

The *Binary Independence Model* (BIM) which we present in this section is the model which has traditionally been used with the PRP. Here, "binary" means Boolean: documents are represented as Boolean term incidence vectors, just

as in the start of Chapter 1. A document is a vector $\vec{x} = (x_1, \ldots, x_n)$ where $x_i = 1$ iff term i is present in document x. Note that with this representation, many possible documents have the same vector representation. "Independence" means that terms are modeled as occurring in documents independently. The model recognizes no association between terms. This is far from correct, but a workable assumption: it is the "naive" assumption of Naive Bayes models; cf. Chapter 13. Indeed, the Binary Independence Model is just another name for the Bernoulli Naive Bayes model presented there.

11.3.1 Deriving a ranking function for query terms

Given a query q, for each document d, we need to compute $P(R|q,d)$. In the BIM, the documents and queries are both represented as binary term incidence vectors. Let x be the binary term incidence vector representing d. Then we wish to compute $P(R|q,x)$. In theory, we should also replace the query with a binary representation, but in general the query inherently is in this representation, so we ignore this distinction. We are interested only in the ranking of documents. It is therefore equivalent to rank documents by their odds of relevance. Given Bayes' Rule, we have:

$$O(R|q,x) = \frac{P(R|q,x)}{P(NR|q,x)} = \frac{\frac{P(R|q)P(x|R,q)}{P(x|q)}}{\frac{P(NR|q)P(x|NR,q)}{P(x|q)}} \tag{11.10}$$

So,

$$O(R|q,x) = \frac{P(R|q)}{P(NR|q)} \cdot \frac{P(x|R,q)}{P(x|NR,q)} \tag{11.11}$$

The left term on the right-hand side of Equation (11.11) is a constant for a given query. Since we are only ranking documents, there is thus no need for us to estimate it. The right-hand term does, however, require estimation. Using the independence (Naive Bayes) assumption, we have:

$$\frac{P(x|R,q)}{P(x|NR,q)} = \prod_{i=1}^{N} \frac{P(x_i|R,q)}{P(x_i|NR,q)} \tag{11.12}$$

So:

$$O(R|q,d) = O(R|q) \cdot \prod_{i=1}^{N} \frac{P(x_i|R,q)}{P(x_i|NR,q)} \tag{11.13}$$

Since each x_i is either 0 or 1, we can separate the terms to give:

$$O(R|q,d) = O(R|q) \cdot \prod_{i:x_i=1} \frac{P(x_i=1|R,q)}{P(x_i=1|NR,q)} \cdot \prod_{i:x_i=0} \frac{P(x_i=0|R,q)}{P(x_i=0|NR,q)} \tag{11.14}$$

Henceforth, let $p_i = P(x_i = 1|R, q)$ and $u_i = P(x_i = 1|NR, q)$.

Let us begin by assuming that for all terms not occurring in the query (i.e., $q_i = 0$) that $p_i = u_i$. That is, they are equally likely to occur in relevant and irrelevant documents. This assumption can be changed, for example when doing relevance feedback. Under this assumption, we need only consider terms in the products that appear in the query, and so,

$$O(R|q,d) = O(R|q) \cdot \prod_{i:x_i=q_i=1} \frac{p_i}{u_i} \cdot \prod_{i:x_i=0;q_i=1} \frac{1-p_i}{1-u_i} \tag{11.15}$$

The left product is over query terms found in the document and the right term is over query terms not found in the document.

We can manipulate this expression by including the query terms found in the document into the right product, but simultaneously dividing through by them in the left product, so the value is unchanged. Then we have:

$$O(R|q,d) = O(R|q) \cdot \prod_{i:x_i=q_i=1} \frac{p_i(1-u_i)}{u_i(1-p_i)} \cdot \prod_{i:q_i=1} \frac{1-p_i}{1-u_i} \tag{11.16}$$

The left product is still over query terms found in the document, but the right product is now over all query terms. That means that this right product is a constant for a particular query, just like the odds $O(R|q)$. So the only quantity that needs to be estimated to rank documents for relevance to a query is the left product. We can equally rank documents by the log of this term, since log is a monotonic function. The resulting quantity used for ranking is called the *Retrieval Status Value* (RSV) in this model:

RETRIEVAL STATUS VALUE

$$RSV = \log \prod_{i:x_i=q_i=1} \frac{p_i(1-u_i)}{u_i(1-p_i)} = \sum_{i:x_i=q_i=1} \log \frac{p_i(1-u_i)}{u_i(1-p_i)} \tag{11.17}$$

So everything comes down to computing the RSV. Let $c_i = \log \frac{p_i(1-u_i)}{u_i(1-p_i)} = \log p_i(1-u_i) - \log u_i(1-p_i)$. These c_i terms are log odds ratios for the terms in the query. How do we estimate these c_i quantities for a particular collection and query?

11.3.2 Probability estimates in theory

For each term i, what would these numbers look like for the whole collection? Consider the following contingency table of counts of documents in the collection:

(11.18)

Documents	Relevant	Non-relevant	Total
$x_i = 1$	s	$n - s$	n
$x_i = 0$	$S - s$	$(N - n) - (S - s)$	$N - n$
Total	S	$N - s$	N

Then the quantities we introduced earlier are: $p_i = s/S$ and $u_i = (n - s)/(N - S)$ and

$$c_i = K(N, n, S, s) = \log \frac{s/(S-s)}{(n-s)/((N-n)-(S-s))} \tag{11.19}$$

To avoid the possibility of zeroes (such as if every or no relevant document has a term) it is fairly standard to add 0.5 to each of the quantities in the center 4 terms, and then to adjust the marginal counts (the totals) accordingly (so, the bottom right cell totals $N + 2$). We will discuss further methods of smoothing estimated counts in Chapter 12, but this method will do for now. Then we have that:

$$\hat{c}_i = K(N, n, S, s) = \log \frac{(s+0.5)/(S-s+0.5)}{(n-s+0.5)/(N-n-S+s+0.5)} \tag{11.20}$$

11.3.3 Probability estimates in practice

Under the assumption that relevant documents are a very small percentage of the collection, it is plausible to approximate statistics for non-relevant documents by statistics from the whole collection. Under this assumption, u_i (the probability of term occurrence in non-relevant documents for a query) is n/N and

$$\log[(1 - u_i)/u_i] = \log[(N - n)/n] \approx \log N/n \tag{11.21}$$

In other words, we can provide a theoretical justification for the most frequently used form of idf weighting.

The above approximation technique cannot easily be extended to relevant documents. The quantity p_i (the probability of term occurrence in relevant documents) can be estimated in various ways:

- From the frequency of term occurrence in known relevant documents (if we know some). This is the basis of probabilistic approaches to relevance weighing in a relevance feedback loop, discussed below.
- As a constant, as was proposed in the Croft and Harper (1979) combination match model. For instance, we might assume that p_i is constant over all terms x_i in the query and that $p_i = 0.5$. This means that each term has even odds of appearing in a relevant document, and so the p_i and $(1 - p_i)$ factors cancel out in the expression for RSV. Such an estimate is weak, but doesn't violently disagree with our hopes for the search terms appearing in documents. Combining this method with our earlier approximation for u_i, the document ranking is determined simply by which query terms occur in documents scaled by their idf weighting. For short documents

(titles or abstracts) in situations in which iterative searching is undesirable, using this weighting term alone can be quite satisfactory, although in many other circumstances we would like to do more.

- It can be made proportional to the log of the probability of occurrence of the term in the collection (Greiff 1998).
- Iterative methods of estimation, which combine some of the above ideas, are discussed below.

11.3.4 Probabilistic approaches to relevance feedback

Accurate estimation of p_i points in the direction of using (pseudo-)relevance feedback, perhaps in an iterative process of estimation. The probabilistic approach to relevance feedback works as follows:

1. Guess a preliminary probabilistic description of R and use it to retrieve a first set of documents V. This can use the probability estimates of the previous section.
2. We interact with the user to refine the description. We learn from the user some definite subset members of $VR \subset R$ and $VNR \subset NR$.
3. We reestimate p_i and u_i on the basis of known relevant and irrelevant documents. If the sets VR and VNR are large enough, we might estimate these quantities directly from these documents as maximum likelihood estimates:
$$p_i = |VR_i|/|VR|$$
(where VR_i is the set of documents in VR containing x_i). In practice, we might wish to smooth these estimates, which we could do by adding $\frac{1}{2}$ to both the count of R_i and to relevant documents not containing the term, giving:
$$p_i = \frac{|VR_i| + 0.5}{VR + 1}$$
To allow for the judged sets being small, it is better to combine the new information with the original guess in a process of Bayesian updating. In this case we have:
$$p_i^{(t+1)} = \frac{|VR_i| + \kappa p_i^{(t)}}{|VR| + \kappa} \tag{11.22}$$
Here $p_i^{(t)}$ is the t^{th} estimate for p_i in an iterative updating process and κ is the weight given to the Bayesian prior. In the absence of other evidence (and assuming that the user is perhaps indicating roughly 5 relevant or

non-relevant documents) then a value of around 4 is perhaps appropriate. That is, the prior is strongly weighted so that the estimate does not change too much from the evidence provided by a very small number of documents.

4. Repeat the above process, generating a succession of approximations to R and hence p_i, until the user is satisfied.

It is also straightforward to derive a pseudo-relevance feedback version of this algorithm:

1. Assume that p_i is constant over all x_i in the query (as above).
2. Determine a guess for the size of the relevant document set. If unsure, a conservative (too small) guess is likely to be best. This motivates use of a fixed size set V of highest ranked documents.
3. Improve our guesses for p_i and u_i. We use the same methods as before for re-estimating p_i. If we simply use the distribution of x_i in the documents in V and let V_i be the set of documents containing x_i then we have:

$$p_i = |V_i|/|V|$$

and if we assume that documents that are not retrieved are not relevant then we have:

$$u_i = (n_i - |V_i|)/(N - |V|)$$

4. Go to step 2 until the ranking of the returned results converges.

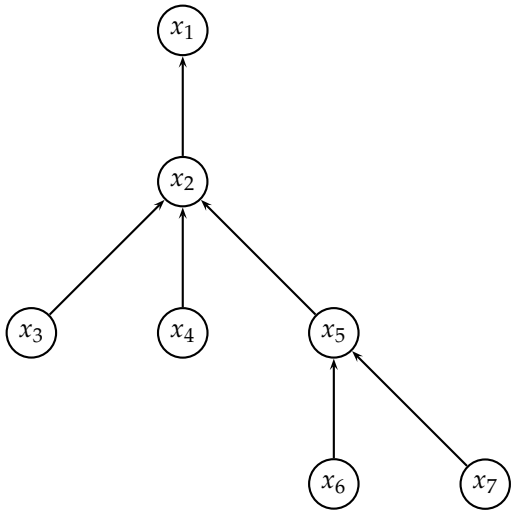
Note that once we have a real estimate for p_i then the c_i weights used in the RSV value look almost like a tf-idf value. For instance, using some of the ideas we saw above, we have:

$$c_i = \log[\frac{p_i}{1-p_i} \cdot \frac{1-u_i}{u_i}] \approx \log[\frac{|V_i|}{|V|-|V_i|} \cdot \frac{N}{n}] \tag{11.23}$$

But things aren't quite the same: the first term measures the percent of (estimated) relevant documents that the term occurs in and not term frequency, and if we divided the two terms apart using log identities, we would be *adding* the two log terms rather than multiplying them.

11.3.5 PRP and BIM

Getting reasonable approximations of the needed probabilities for a probabilistic IR model is possible. It requires some restrictive assumptions. In the BIM these are:



► **Figure 11.1** A tree of dependencies between terms.

- term independence
- terms not in the query don't affect the outcome
- a Boolean representation of documents/queries/relevance
- document relevance values are independent

A general problem seems to be that probabilistic models either require partial relevance information or else only allow one to derive apparently inferior term weighting models.

However, some of these assumptions can be removed. For example, we can remove the assumption that index terms are independent. This is very far from true in practice. The canonical case being pairs like Hong and Kong which are strongly dependent, but dependencies can be complex such as when the term New enters into various dependencies with other words such as York, England, City, Stock and Exchange, and University. van Rijsbergen (1979) proposed a simple, plausible model which allowed a tree structure of dependencies, as in Figure 11.1. In this model each term can be directly dependent on one other. When it was invented in the 1970s, estimation problems held back the practical success of this model, but the idea was reinvented as the

Tree Augmented Naive Bayes model by Friedman and Goldszmidt (1996), who used it with some success on various machine learning data sets.

11.4 An appraisal and some extensions

Probabilistic methods are one of the oldest formal models in IR. Even in the 1970s they were held out as an opportunity to place IR on a former theoretical footing, and with the resurgence of probabilistic methods in the 1990s, that hope has returned, and probabilistic methods are again one of the currently hottest topics in IR. Traditionally, probabilistic IR has had neat ideas but the methods have never won on performance. At the end of the day, things also are not that different: one builds an information retrieval scheme in the exact same way as we have been discussing, it's just that at the end, you score queries not by cosine similarity and tf-idf in a vector space, but by a slightly different formula motivated by probability theory. Indeed, sometimes people have adopted term weighting formulae from probabilistic models and used them in vector space (cosine calculation) retrieval engines. Here we briefly mention two influential extensions of the traditional probabilistic model, and in the next chapter, we look at the somewhat different probabilistic language modeling approach to IR.

11.4.1 Okapi BM25: a non-binary model

BM25 WEIGHTS

The BIM works reasonably in the contexts for which it was first designed (short catalog records and abstracts, of fairly consistent length), but for other search collections, it seems clear that a model should pay attention to term frequency and document length. The *BM25 weights* were developed as a way of building a probabilistic model sensitive to these quantities while not introducing too many additional parameters into the model (Spärck Jones et al. 2000). We will not develop the theory behind the model here, but just present a series of forms that build up to the now-standard form used for document scoring (there have been some variants). The simplest score for document j is just idf weighting of the query terms present:

$$RSV_j = \sum_{i \in q} \log \frac{N}{n} \tag{11.24}$$

We can improve on that by factoring in term frequency and document length:

$$RSV_j = \sum_{i \in q} \left[\log \frac{N}{n} \right] \cdot \frac{(k_1 + 1)\mathrm{tf}_{ij}}{k_1((1-b) + b \times (dl_j/avdl)) + \mathrm{tf}_{ij}} \tag{11.25}$$

If the query is long, then we might also put in similar weighting for query terms (this is appropriate if the queries are paragraph long information needs, but unnecessary for short queries):

$$RSV_j = \sum_{i \in q} \left[\log \frac{N}{n} \right] \cdot \frac{(k_1 + 1) tf_{ij}}{k_1((1-b) + b \times (dl_j / avdl)) + tf_{ij}} \cdot \frac{(k_3 + 1) tf_{iq}}{k_3 + tf_i q} \tag{11.26}$$

Finally, if we have available relevance judgements, then we can use the full form of Equation (11.20) in place of the left hand side term (which is what we saw in Equation (11.21)):

$$\begin{aligned} RSV_j &= \sum_{i \in q} \left[\log \frac{(|VR_i| + \frac{1}{2}) / (|VR| - |VR_i| + \frac{1}{2})}{(n - |VR_i| + \frac{1}{2}) / (N - n - |VR| + |VR_i| + \frac{1}{2})} \right. \\ &\qquad \left. \times \frac{(k_1 + 1) tf_{ij}}{k_1((1-b) + b(dl_j / avdl)) + tf_{ij}} \cdot \frac{(k_3 + 1) tf_{iq}}{k_3 + tf_{iq}} \right] \end{aligned} \tag{11.27}$$

Here, N, n, $|VR_i|$ and VR are used as in Sections 11.3.2 and 11.3.4, tf_{ij} and tf_{iq} are the frequency of the i^{th} term in the document and query respectively, and dl_j and $avdl$ are the document length and average document length of the whole collection. The first term reflects relevance feedback (or just idf weighting if no relevance information is available), the second term implements document term frequency and document length scaling, and the third term considers term frequency in the query (and is only necessary if the query is long). k_1 and k_3 are positive tuning parameters that scale the term frequency scaling inside the document and query respectively. A value of 0 corresponds to a binary model (no term frequency), and a large value corresponds to using raw term frequency. b is another tuning parameter ($0 \leq b \leq 1$) which determines the scaling by document length: $b = 1$ corresponds to fully scaling the term weight by the document length, while $b = 0$ corresponds to no length normalization. There is no length normalization of queries (it is as if $b = 0$ for this term). The tuning parameters should ideally be set to optimize performance on a development query collection. In the absence of such optimization, reasonable values are to set $k_1 = k_3 = 2$ and $b = 0.75$. Relevance feedback can also involve augmenting the query (automatically or with manual review) with some (10–20) of the top terms in the known-relevant documents as ordered by the relevance factor $\hat{c}_i$ from Equation (11.20). This BM25 term weighting formula has been used quite widely and quite successfully. See Spärck Jones et al. (2000) for extensive motivation and discussion of experimental results.

11.4.2 Bayesian network approaches to IR

BAYESIAN NETWORKS

Turtle and Croft (1989; 1991) introduced the use of *Bayesian networks* (Jensen

and Jensen 2001), a form of probabilistic graphical model for information retrieval. We skip the details because fully introducing the formalism of Bayesian networks would require much too much space, but conceptually, Bayesian networks use directed graphs to show probabilistic dependencies between variables, as in Figure 11.1, and have led to the development of sophisticated algorithms for propagating influence so as to allow learning and inference with arbitrary knowledge of arbitrary directed acyclic graphs. Turtle and Croft used a sophisticated network model to better model the complex dependencies between a document and a user's information need.

The model decomposes into two parts: a document collection network and a query network. The document collection network is large, but can be precomputed: it maps from documents to terms to concepts. The concepts are a thesaurus-based expansion of the terms appearing in the document. The query network is relatively small but a new network needs to be built each time a query comes in, and then attached to the document network. The query network maps from query terms, to query subexpressions (built using probabilistic or "noisy" versions of AND and OR operators), to the user's information need.

The result is a flexible probabilistic network which can generalize various simpler Boolean and probabilistic models. The system allowed efficient large-scale retrieval, and was the basis of the InQuery text retrieval system, used at the University of Massachusetts, and for a time sold commercially. On the other hand, the model still used various approximations and independence assumptions to make parameter estimation and computation possible. We would note that this model was actually built very early on in the modern era of using Bayesian networks, and there have been many subsequent developments in the theory, and the time is perhaps right for a new generation of Bayesian network-based information retrieval systems.

11.5 References and further reading

Longer introductions to the needed probability theory can be found in most introductory probability and statistics books, such as Grinstead and Snell (1997). An introduction to Bayesian utility theory can be found in Ripley (1996).

The probabilistic approach to IR originated in the UK in the 1950s. Robertson and Spärck Jones (1976b) introduces the main foundations of the BIM and van Rijsbergen (1979) presents in detail the classic BIM probabilistic model. The idea of the PRP is variously attributed to Stephen Robertsion, M. E. Maron and William Cooper (the term "Probabilistic Ordering Principle" is used in Robertson and Spärck Jones (1976b), but PRP dominates in later work). Fuhr (1992) is a more recent presentation of probabilistic

IR, which includes coverage of other approaches such as probabilistic logics and Bayesian networks. Crestani et al. (1998) is another survey, though it adds little not in the earlier sources. Spärck Jones et al. (2000) is the definitive presentation of probabilistic IR experiments by the "London school", and Robertson (2005) presents a retrospective on the group's participation in TREC evaluations, including detailed discussion of the Okapi BM25 model and its development.

11.6 Exercises

Exercise 11.1

Think through the differences between standard vector space tf-idf weighting and the BIM probabilistic retrieval model on the first iteration.

Exercise 11.2

Think through the differences between vector space (pseudo) relevance feedback and probabilistic (pseudo) relevance feedback.

11.6.1 Okapi weighting

OKAPI WEIGHTING

The so-called *Okapi* weighting, defined for a term t and a document d as follows, for positive constants k_1 and b:

$$\ln \frac{N - \mathrm{df}_t + 0.5}{\mathrm{df}_t + 0.5} \cdot \frac{(k_1 + 1)\mathrm{tf}_{t,d}}{k_1(1 - b + b(\ell(d)/\ell_{\mathrm{ave}})) + \mathrm{tf}_{t,d}}. \tag{11.28}$$

As before N is the total number of documents; $\ell(d)$ is the length of d and ℓ_{ave} is the average length of documents in the corpus. This weighting scheme has proven to work relatively well across a wide range of corpora and search tasks.

12 *Language models for information retrieval*

In the traditional probabilistic approach to IR, the user has an information need, and determines a query q which is run against documents d, and we try to determine the probability of relevance $P(R|q,d)$. The original language modeling approach bypasses overtly modeling the concept of relevance. It instead builds a probabilistic language model M_d from each document d, and ranks documents based on the probability of the model generating the query: $P(q|M_d)$.

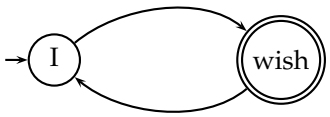
A common suggestion to users for coming up with good queries is to think of words that would likely appear in a relevant document, and to use those words as your query. The language modeling approach to IR directly models that idea: a document is a good match to a query if the document model is likely to generate the query, which will in turn happen if the document contains the query words often.

GENERATIVE MODEL

What do we mean by a document model generating a query? A traditional *generative model* of language of the kind familiar from formal language theory can be used either to recognize or to generate strings. For example, the finite automaton shown in Figure 12.1 can generate strings that include the examples shown. The full set of strings that can be generated is called the language of the automaton.

LANGUAGE MODEL

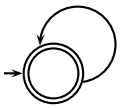
If instead each node has a probability distribution over generating different words, we have a language model. A (stochastic or probabilistic) *language model* is a function that puts a probability measure over strings drawn from some vocabulary. One simple kind of language model is equivalent to a probabilistic finite automaton consisting of just a single node with a single probability distribution of producing different words, as shown in Figure 12.2, coupled with a probability of stopping when in a finish state. Such a model places a probability distribution over any sequence of words. By construction, it also provides a model for generating text according to its distribution. To find the probability of a word sequence, we just multiply the probabilities



I wish
I wish I wish
I wish I wish I wish
I wish I wish I wish I wish I wish I wish
...

*wish I wish

► **Figure 12.1** A simple finite automaton and some of the strings in the language that it generates. → shows the start state of the automaton and a double circle indicates a (possible) finishing state.



the	0.2
a	0.1
frog	0.01
toad	0.01
said	0.03
likes	0.02
that	0.04

► **Figure 12.2** A one-state finite automaton that acts as a unigram language model together with a partial specification of the state emission probabilities.

which it gives to each word in the sequence. For example,

$$P(\text{frog said that toad likes frog}) = 0.01 \times 0.03 \times 0.04 \times 0.02 \times 0.01 = 0.0000000024 \quad (12.1)$$

Here we omit the probability of stopping after *frog*. An explicit stop probability is needed for the finite automaton to generate and give probabilities to finite strings, but we will in general omit mention of it, since, if fixed, it does not alter the ranking of documents.

Suppose, now, that we have two language models M_1 and M_2, shown partially in Figure 12.3. Each gives a probability estimate to a sequence of words, as shown in the example. The language model that gives the higher probability to the sequence of words is more likely to have generated the word sequence. For the sequence shown, we get:

Model M_1		Model M_2	
the	0.2	the	0.15
a	0.1	a	0.12
frog	0.01	frog	0.0002
toad	0.01	toad	0.0001
said	0.03	said	0.03
likes	0.02	likes	0.04
that	0.04	that	0.04
dog	0.005	dog	0.01
cat	0.003	cat	0.015
monkey	0.001	monkey	0.002
...	...	...	...

► **Figure 12.3** Partial specification of two unigram language models.

(12.2)

s	frog	said	that	toad	likes	that	dog
M_1	0.01	0.03	0.04	0.01	0.02	0.04	0.005
M_2	0.0002	0.03	0.04	0.0001	0.04	0.04	0.01

$P(s|M_1) = 0.00000000000048$
$P(s|M_2) = 0.000000000000000384$

and we see that $P(s|M_1) > P(s|M_2)$.

How do people build probabilities over word sequences? We can always use the chain rule to decompose the probability of a sequence of events into the probability of each successive events conditioned on earlier events:

$$P(w_1w_2w_3w_4) = P(w_1)P(w_2|w_1)P(w_3|w_1w_2)P(w_4|w_1w_2w_3)$$

UNIGRAM LANGUAGE MODEL

The simplest form of language model simply throws away all conditioning context, and estimates each word independently. Such a model is called a *unigram language model*:

$$P_{\text{uni}}(w_1w_2w_3w_4) = P(w_1)P(w_2)P(w_3)P(w_4)$$

Under this model the order of words is irrelevant, and so such models are sometimes called "bag of words" models as discussed in Chapter 6 (page 88). There are many more complex kinds of language models, such as bigram language models, which condition on the previous word,

$$P_{\text{bi}}(w_1w_2w_3w_4) = P(w_1)P(w_2|w_1)P(w_3|w_2)P(w_4|w_3)$$

and even more complex grammar-based language models such as probabilistic context-free grammars. However, most language-modeling work in IR

has used unigram language models, and IR is probably not the most productive place to try using complex language models, since IR does not directly depend on the structure of sentences to the extent that other tasks like speech recognition do. Moreover, since, as we shall see, IR language models are frequently estimated from a single document, there is often not enough training data and losses from sparseness outweigh any gains from richer models.

The fundamental problem in designing language models is that we generally do not know what exactly we should use as the model M_d. However, we do generally have a sample of text that is representative of that model. This problem makes a lot of sense in the original, primary uses of language models. For example, in speech recognition, we have a training sample of text, but we have to expect that in the future, users will use different words and in different sequences, which we have never observed before, and so the model has to generalize beyond the observed data to allow unknown words and sequences. This interpretation is not so clear in the IR case, where a document is finite and usually fixed. However, we pretend that the document d is only a representative sample of text drawn from a model distribution, we estimate a language model from this sample, use that model to calculate the probability of observing any word sequence, and finally rank documents according to their probability of generating the query.

12.1 The Query Likelihood Model

12.1.1 Using Query Likelihood Language Models in IR

QUERY LIKELIHOOD MODEL

Language modeling is a quite general formal approach to IR, with many variant realizations. The original and basic method for using language models in IR is the *query likelihood model*. In it, we construct from each document d in the collection a language model M_d. Our goal is to rank documents by $P(d|q)$, where the probability of a document is interpreted as the likelihood that it is relevant to the query. Using Bayes rule, we have:

$$P(d|q) = P(q|d)P(d)/P(q)$$

$P(q)$ is the same for all documents, and so can be ignored. The prior $P(d)$ is often treated as uniform across all d and so it can also be ignored, but we could implement a genuine prior which could include criteria like authority, length, genre, newness, and number of previous people who have read the document. But, given these simplifications, we return results ranked by simply $P(q|d)$, the probability of the query q given by the language model derived from d. The Language Modeling approach thus attempts to model the query generation process: Documents are ranked by the probability that a query would be observed as a random sample from the respective document model.

The most common way to do this is using the multinomial unigram language model, which is equivalent to a multinomial Naive Bayes model (page 198), where the documents are the classes, each treated in the estimation as a separate "language". Under this model, we have that:

$$P(q|M_d) = \prod_{w \in V} P(w|M_d)^{c(w)} \tag{12.3}$$

Usually a unigram estimate of words is used in IR. There is some work on bigrams, paralleling the discussion of van Rijsbergen in Chapter 11 (page 172), but it hasn't been found very necessary. While modeling term cooccurrence should improve estimates somewhat, IR is different to tasks like speech recognition: word order and sentence structure are not very necessary to modeling the topical content of documents.

For retrieval based on a probabilistic language model, we treat the generation of queries as a random process. The approach is to

1. Infer a language model for each document.
2. Estimate the probability of generating the query according to each of these models.
3. Rank the documents according to these probabilities.

The intuition is that the user has a prototype document in mind, and generates a query based on words that appear in this document. Often, users have a reasonable idea of terms that are likely to occur in documents of interest and they will choose query terms that distinguish these documents from others in the collection. Collection statistics are an integral part of the language model, rather than being used heuristically as in many other approaches.

12.1.2 Estimating the query generation probability

In this section we describe how to estimate $P(q|M_d)$. The probability of producing the query given the language model M_d of document d using maximum likelihood estimation (MLE) and given the unigram assumption is:

$$\hat{P}(q|M_d) = \prod_{t \in q} \hat{P}_{\text{mle}}(t|M_d) = \prod_{t \in q} \frac{tf_{t,d}}{dl_d} \tag{12.4}$$

where M_d is the language model of document d, $tf_{t,d}$ is the (raw) term frequency of term t in document d, and dl_d is the number of tokens in document d.

The classic problem with such models is one of estimation (the $\hat{}$ is used above to stress that the model is estimated). In particular, some words will

not have appeared in the document at all, but are possible words for the information need, which the user may have used in the query. If we estimate $\hat{P}(t|M_d) = 0$ for a term missing from a document d, then we get a strict conjunctive semantics: documents will only give a query non-zero probability if all of the query terms appear in the document. This may or may not be undesirable: it is partly a human-computer interface issue: vector space systems have generally preferred more lenient matching, though recent web search developments have tended more in the direction of doing searches with such conjunctive semantics. But regardless of one's approach here, there is a more general problem of estimation: occurring words are also badly estimated; in particular, the probability of words occurring once in the document is normally overestimated, since there one occurrence was partly by chance.

This problem of insufficient data and a zero probability preventing any non-zero match score for a document can spell disaster. We need to smooth probabilities: to discount non-zero probabilities and to give some probability mass to unseen things. There's a wide space of approaches to smoothing probability distributions to deal with this problem, such as adding a number (1, 1/2, or a small ϵ) to counts and renormalizing, discounting, Dirichlet priors and interpolation methods. A simple idea that works well in practice is to use a mixture between the document multinomial and the collection multinomial distribution.

The general approach is that a non-occurring term is possible in a query, but no more likely than would be expected by chance from the whole collection. That is, if $tf_{t,d} = 0$ then

$$\hat{P}(t|M_d) \leq cf_t/cs$$

where cf_t is the raw count of the term in the collection, and cs is the raw size (number of tokens) of the entire collection. We can guarantee this by mixing together a document-specific model with a whole collection model:

$$\hat{P}(w|d) = \lambda \hat{P}_{\text{mle}}(w|M_d) + (1-\lambda)\hat{P}_{\text{mle}}(w|M_c) \tag{12.5}$$

where $0 < \lambda < 1$ and M_c is a language model built from the entire document collection. This mixes the probability from the document with the general collection frequency of the word. Correctly setting λ is important to the good performance of this model. A high value of lambda makes the search "conjunctive-like" – suitable for short queries. A low value is more suitable for long queries. One can tune λ to optimize performance, including not having it be constant but a function of document size.

So, the general formulation of the basic LM for IR is:

$$P(q|d) \propto P(d) \prod_{t \in q} ((1-\lambda)P(t|M_c) + \lambda P(t|M_d))$$

The equation represents the probability that the document that the user had in mind was in fact this one.

Example 12.1: Suppose the document collection contains two documents:

- d_1: Xyz reports a profit but revenue is down
- d_2: Qrs narrows quarter loss but revenue decreases further

The model will be MLE unigram models from the documents and collection, mixed with $\lambda = 1/2$.

Suppose the query is *revenue down*. Then:

(12.6)
$$\begin{aligned} P(q|d_1) &= [(1/8 + 2/16)/2] \times [(1/8 + 1/16)/2] \\ &= 1/8 \times 3/32 = 3/256 \\ P(q|d_2) &= [(1/8 + 2/16)/2] \times [(0/8 + 1/16)/2] \\ &= 1/8 \times 1/32 = 1/256 \end{aligned}$$

So, the ranking is $d_1 > d_2$.

12.2 Ponte and Croft's Experiments

Ponte and Croft (1998) present the first experiments on the language modeling approach to information retrieval. Their basic approach where each document defines a language model is the model that we have presented until now. However, we have presented an approach where the language model is a mixture of two multinomials, much as in Miller et al. (1999), Hiemstra (2000) rather than Ponte and Croft's multivariate Bernoulli model. The use of multinomials has been standard in most subsequent work in the LM approach and evidence from text categorization (see Chapter 13) suggests that it is superior. Ponte and Croft argued strongly for the effectiveness of the term weights that come from the language modeling approach over traditional tf-idf weights. We present a subset of their results in Figure 12.4 where they compare tf-idf to language modeling by evaluating TREC topics 202–250 evaluated on TREC disks 2 and 3. The queries are sentence length natural language queries. The language modeling approach yields significantly better results than their baseline tf-idf based term weighting approach. And indeed the gains shown here have been extended in subsequent work.

12.3 Language modeling versus other approaches in IR

The language modeling approach provides a novel way of looking at the problem of text retrieval, which links it with a lot of recent work in speech

	Precision			
Rec.	tf-idf	LM	%chg	
0.0	0.7439	0.7590	+2.0	
0.1	0.4521	0.4910	+8.6	
0.2	0.3514	0.4045	+15.1	*
0.3	0.2761	0.3342	+21.0	*
0.4	0.2093	0.2572	+22.9	*
0.5	0.1558	0.2061	+32.3	*
0.6	0.1024	0.1405	+37.1	*
0.7	0.0451	0.0760	+68.7	*
0.8	0.0160	0.0432	+169.6	*
0.9	0.0033	0.0063	+89.3	
1.0	0.0028	0.0050	+76.9	
Ave	0.1868	0.2233	+19.55	*

► **Figure 12.4** Results of a comparison of tf-idf to language modeling (LM) term weighting by Ponte and Croft (1998). The version of tf-idf from the INQUERY IR system includes length normalization of tf. The table gives an evaluation according to 11-point average precision with significance marked with a * according to Wilcoxon signed rank test. While the language modeling approach always does better in these experiments, note that where the approach shows significant gains is at higher levels of recall.

and language processing. As Ponte and Croft (1998) emphasize, the language modeling approach to IR provides a different form of scoring matches between queries and documents, and the hope is that the probabilistic language modeling foundation improves the weights that are used, and hence the performance of the model. The major issue is estimation of the document model, such as choices of how to smooth it effectively. It has achieved very good retrieval results. Compared to other probabilistic approaches, such as BIM from Chapter 11, the main difference is that the LM approach attempts to do away with explicitly modeling relevance (whereas this is the central variable evaluated in the BIM approach). The LM approach assumes that documents and expressions of information problems are objects of the same type, and assesses their match by importing the tools and methods of language modeling from speech and natural language processing. The resulting model is mathematically precise, conceptually simple, computationally tractable, and intuitively appealing.

On the other hand, like all IR models, one can also raise objections to the model. The assumption of equivalence between document and information problem representation is unrealistic. Current LM approaches use very simple models of language, usually unigram models. Without an explicit notion

of relevance, relevance feedback is difficult to integrate into the model, as are user preferences or priors over document relevance. It also isn't easy to see how to accommodate notions of phrasal matching or passage matching or Boolean retrieval operators. Subsequent work in the LM approach has looked at addressing some of these concerns, including putting relevance back into the model and allowing a language mismatch between the query language and the document language.

The model has some relation to traditional tf-idf models. Term frequency is directly in tf-idf models, and much recent work has recognized the importance of document length normalization. The effect of doing a mixture of document generation probability with collection generation probability is a little like idf: terms rare in the general collection but common in some documents will have a greater influence on the ranking of documents. In most concrete realizations, the models share treating terms as if they were independent. On the other hand, the intuitions are probabilistic rather than geometric, the mathematical models are more principled rather than heuristic, and the details of how statistics like term frequency and document length are used differ. If one is concerned mainly with performance numbers, while the LM approach has been proven quite effective in retrieval experiments, there is little evidence that its performance exceeds a well-tuned traditional ranked retrieval system.

12.4 Extended language modeling approaches

In this section we briefly note some of the work that has taken place that extends the basic language modeling approach.

There are other ways that one could think of using the language modeling idea in IR settings, and many of them have been tried in subsequent work. Rather than looking at the probability of a document language model generating the query, you can look at the probability of a query language model generating the document. The main reason that doing things in this direction is less appealing is that there is much less text available to estimate a query language model, and so the model will be worse estimated, and will have to depend more on being smoothed with some other language model. On the other hand, it is easy to see how to incorporate relevance feedback into such a model: one can expand the query with terms taken from relevant documents in the usual way and hence update the query language model (Zhai and Lafferty 2001a). Indeed, with appropriate modeling choices, this approach leads to the BIR model of Chapter 11.

Rather than directly generating in either direction, one can make a language model from both the document and query, and then ask how different these two language models are from each other. Lafferty and Zhai (2001) lay

► **Figure 12.5** Three ways of developing the language modeling approach: query likelihood, document likelihood and model comparison.

out these three ways of thinking about things, which we show in Figure 12.5 and develop a general risk minimization approach for document retrieval. For instance, one way to model the risk of returning a document d as relevant to a query q is to use the *Kullback-Leibler divergence* between their respective language models:

KULLBACK-LEIBLER DIVERGENCE

$$R(d;q) = KL(d\|q) = \sum_w P(w|M_q) \log \frac{P(w|M_q)}{P(w|M_d)}$$

This asymmetric divergence measure coming from information theory shows how bad the probability distribution M_q is at modeling M_d. Lafferty and Zhai (2001) present results suggesting that a model comparison approach outperforms both query-likelihood and document-likelihood approaches.

Basic LMs do not address issues of alternate expression, that is, synonymy, or any deviation in use of language between queries and documents. Berger and Lafferty (1999) introduce translation models to bridge this query-document gap. A translation model lets you generate query words not in a document by translation to alternate terms with similar meaning. This also provides a basis for performing cross-lingual IR. Assuming a probabilistic lexicon *Lex* which gives information on synonymy or translation pairs, the nature of the translation query generation model is:

$$P(q|M_d) = \prod_{w \in q} \sum_{v \in Lex} P(v|M_d) T(w|v)$$

The left term on the right hand side is the basic document language model, and the right term performs translation. This model is clearly more computationally intensive and one needs to build a translation model, usually using separate resources (such as a dictionary or a statistical machine translation system's lexicon).

12.5 References and further reading

For more details on the basic concepts and smoothing methods for probabilistic language models, see either Manning and Schütze (1999, Ch. 6) or Jurafsky and Martin (2000, Ch. 6).

The important initial papers that originated the language modeling approach to IR are: (Ponte and Croft 1998, Hiemstra 1998, Berger and Lafferty 1999, Miller et al. 1999). Other relevant papers can be found in the next several years of SIGIR proceedings. Croft and Lafferty (2003) contains a collection of papers from a workshop on language modeling approaches and Hiemstra and Kraaij (2005) reviews one prominent thread of work on using language modeling approaches for TREC tasks. System implementers should consult Zhai and Lafferty (2001b), Zaragoza et al. (2003) for detailed empirical comparisons of different smoothing methods for language models in IR. Additionally, recent work has achieved some gains by going beyond the unigram model, providing the higher order models are smoothed with lower order models Gao et al. (2004), Cao et al. (2005). For a critical viewpoint on the rationale for the language modeling approach, see Spärck Jones (2004).

13 *Text classification and Naive Bayes*

Up until now, this book has principally discussed the process of *ad-hoc retrieval* where a user has a transient information need, which they try to address by posing one or more queries to a search engine. However, many users have ongoing information needs. For example, because of my job in the computer industry, I might need to track developments in *multicore computer chips*. One way of doing this is for me to issue the query multicore AND computer AND chip against an index of recent newswire articles each morning. Rather than doing this by hand, it makes sense if this task can be automated. Therefore, many systems support having *standing queries* which can be run against new documents.

STANDING QUERY

If my standing query is just multicore AND computer AND chip, I will tend to miss many relevant new articles which use other terms such as *multicore processors*. To achieve good recall, standing queries tend to be refined over time and to become quite complex. For example, for this information need, using a boolean search engine with stemming, I might end up with a query like (multicore OR multi-core) AND (chip OR processor OR microprocessor). However, rather than having to painstakingly build up such a standing query by hand, there might be better ways to address a long-term information need. For example, I might be able to just give the system some examples of news articles that are and aren't relevant to my interests, and have it determine which documents I need to see. To be able to better appreciate the range of approaches and also the many applications of the technologies that we introduce in the next several chapters, we will introduce the general notion of a *classification* problem. In this example, I am wanting do do two-class classification: dividing all new newswire articles into the two classes 'documents that talk about multicore computer chips' and 'documents not about multicore computer chips'.

CLASSIFICATION

In many text-processing applications, we wish to determine whether a document is in one of several pre-determined classes, which might be subject areas like *China* or *coffee*. These classes are usually referred to as *topics*, and the task is then called *topic classification*. An example appears in Fig-

TOPIC CLASSIFICATION

ure 13.1. In order to do topic classification, we need to build a topic classifier which can pick out documents on each topic of interest. A topic classifier for *China* will determine for each document whether it is relevant to *China* or not by looking for terms like China, Chinese and Beijing. These terms are more common in *China* than in non-*China* documents. The classification task is to determine such differences between the classes and to exploit them for assigning a document to its correct class.

VERTICAL SEARCH ENGINE

A second application of topic classifiers is to help build a vertical search engine. *Vertical search engines* restrict searches to a particular topic. For example, the query computer science on a vertical search engine for the topic *China* might be expected to return a list of Chinese computer science departments with higher precision and recall than the query computer science China on a general purpose search engine. This is because the vertical search engine does not include web pages in its index that contain the word china in a different sense (e.g., referring to a hard white ceramic), but does include relevant pages even if they don't explicitly mention the terms China or Chinese.

The notion of classifiers is very general. There are many applications of classification technology within and beyond information retrieval. Other IR examples include:

- Several of the preprocessing steps necessary for indexing as discussed in Chapter 2: detecting a document's encoding (ASCII, Unicode UTF-8 etc; page 17); word segmentation (Is the gap between two letters a word boundary or not? page 22); truecasing (page 26); and identifying the language of a document (page 36)
- The automatic detection of spam pages (which then are not included in the search engine index)
- The automatic detection of sexually explicit content (which is included in search results only if the user turns an option such as SafeSearch off)

While the classification task we will use as an example in this book is topic classification, this list shows the general importance of classification in information retrieval. Most retrieval systems today contain multiple components that use some form of classifier.

A computer is not essential for classification. Many classification tasks have traditionally been solved manually. Books in a library are assigned library of congress categories by a librarian. But manual classification is expensive to scale. Our initial example of a standing query illustrated the first alternative approach: classification by the use of rules, most commonly written by hand. Here are some rules one might use for the class *UK*:

london $\rightarrow$ *UK*
queen AND NOT (freddie OR mercury) $\rightarrow$ *UK*
buckingham AND palace $\rightarrow$ *UK*

A rule captures a certain combination of keywords that indicates a class. Hand-coded rules have good scaling properties, but creating the rules and maintaining them over time is labor-intensive. A technically skilled person (e.g., a domain expert who is good at writing regular expressions) can create rule sets that will rival in accuracy the automatically generated classifiers we will discuss shortly. But it can be hard to find someone with this specialized skill.

The third approach to text classification is based on machine learning and it is the approach that we focus on in this book. In machine learning, the set of rules or, more generally, the decision criterion of the topic classifier is learned automatically from training data. This approach is also called *statistical text classification* if the learning algorithm is statistical. In statistical text classification, we require a number of good example documents (or training documents) from each class. The need for manual classification is not eliminated since the training documents come from a person who has labeled them. But labeling documents is arguably an easier task than writing rules. Almost anybody can look at a document and decide whether or not it is about the geographic region China. Sometimes such labeling is already implicitly part of an existing workflow, for instance, if it is a staffer's job to pick out relevant news articles for a senator.

STATISTICAL TEXT CLASSIFICATION

We begin this chapter with a general introduction to the text classification problem including a formal definition (Section 13.1); we then cover Naive Bayes, a particularly simple and effective classification method (Sections 13.2–13.4). All of the classification algorithms we study view documents as vectors in high-dimensional spaces. To improve the efficiency of these algorithms, it is generally desirable to reduce the dimensionality of these spaces; to this end, a technique known as *feature selection* is commonly applied in text classification. Section 13.5 introduces two of the most widely used feature selection techniques. Section 13.6 covers evaluation of text classification. In the following two chapters, Chapters 14 and 15, we look at two other classification methods, vector space classification and support vector machines.

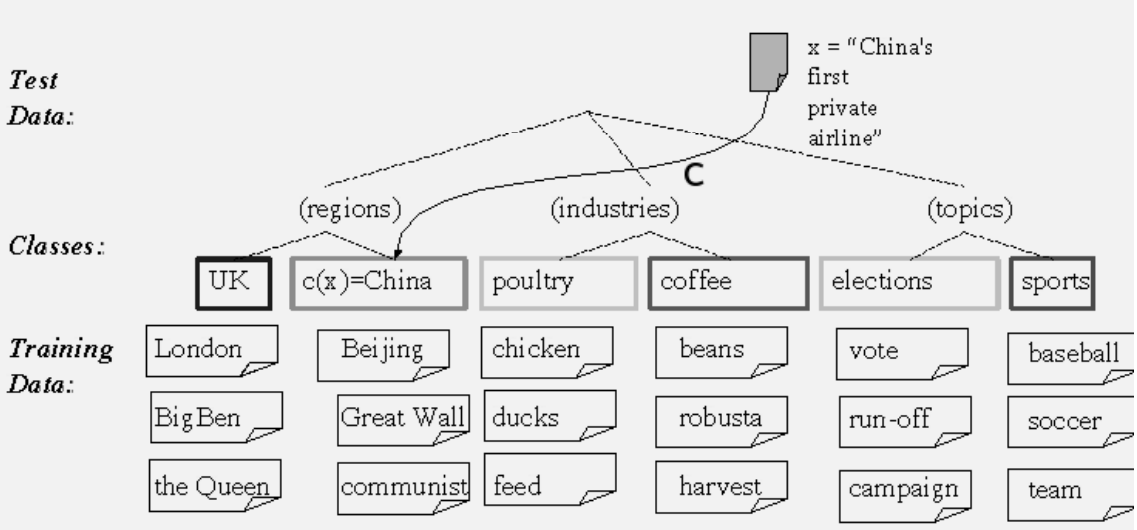
13.1 The text classification problem

In text classification, we are given a description of an instance $x \in \mathbb{X}$, where $\mathbb{X}$ is the *instance space*; and a fixed set of *classes* (or, equivalently, *categories*) $C = \{c_1, c_2, \ldots, c_J\}$. The instances are typically documents and the classes are typically human-defined for the needs of an application, as in the *China* example above. Given one or more training documents for each of the J classes, we wish to learn a *classification function* γ that maps instances to classes:

INSTANCE SPACE
CLASS
CATEGORY

CLASSIFICATION FUNCTION

$$\gamma : \mathbb{X} \rightarrow C \tag{13.1}$$



► **Figure 13.1** Training and test data in text classification. CHANGE NOTATION: gamma for c, d for x

SUPERVISED LEARNING

This type of learning is called *supervised learning* since a "supervisor" (the human who defines the classes and labels training documents for each class) functions as a teacher directing the learning process.

Figure 13.1 shows an example of text classification from the Reuters-RCV1 collection, introduced in Section 4.1, page 52. There are six classes (*UK*, *China*, ..., *sports*), each with three training documents. We show a few mnemonic words for each document's content. The *training data* provide some typical examples for each class, so that we can learn γ. Once we have learned it, we can apply γ to *test data*, for example the new document *China's first private airline* whose class is unknown. The classification function assigns the new document to class $\gamma(d) = China$, which is the correct assignment in this case.

TRAINING DATA

TEST DATA

The classes often have some interesting structure such as the hierarchy in Figure 13.1. There are two instances each of region categories, industry categories and topic categories. A hierarchy can be an important aid in solving a classification problem (see Section 13.7). But we will make the simplifying assumption here that the classes form a set with no relationship between the classes.

13.2 Naive Bayes text classification

NAIVE BAYES

The first machine learning method we look at is *Naive Bayes* (often abbrevi-

NB
GENERATIVE MODEL

ated as *NB*). As indicated by its name, it is a Bayesian method. A *generative model* (Chapter 12, page 177) – a model that generate documents – is associated with each class. By comparing the text in a document d to the text that would be generated by the model associated with a class c_j, we compute an estimate of the likelihood that d belongs to c_j. Bayesian text classification provides a generative model that allows us to infer from the text in d the probability $P(c_j|d)$ of being in each class c_j, and hence its unknown (or hidden) class.

PRIOR

To perform this inference, we first define the *prior class probability* or *prior* $P(c_j)$ of a class c_j as the probability that document d is in c_j, if we knew nothing about the text in d. (How do we compute the value of $P(c_j)$? This will be addressed shortly.) By multiplying $P(c_j)$ by the probability $P(d|c_j)$ that d was generated by c_j, we get (a quantity proportional to) the *posterior probability* $P(c_j|d)$. The posterior probability is the probability of class membership after we have taken into account the observable evidence. A classification decision is made by assigning the document to the class with the highest posterior probability. This classification principle is called *maximum a-posteriori* or *MAP*. It can be derived as follows.

POSTERIOR PROBABILITY

MAXIMUM A-POSTERIORI MAP

$$
\begin{aligned}
c_{\text{map}} &= \arg\max_{c_j \in C} P(c_j|d) \\
&= \arg\max_{c_j \in C} \frac{P(d|c_j)P(c_j)}{P(d)} && (13.2) \\
&= \arg\max_{c_j \in C} P(d|c_j)P(c_j) && (13.3)
\end{aligned}
$$

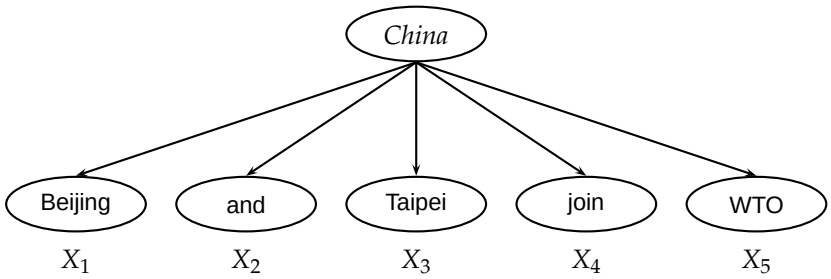
where Bayes' Rule (Equation (11.4), page 164) is applied in 13.2. We can drop the denominator in the last step since $P(d)$ is the same for all classes and does not affect the argmax.

ATTRIBUTE
FEATURE

We have to define the instance space $\mathbb{X}$ in which documents are represented. The generic representation we will start with is a tuple of *attributes* or *features* x_i: $d = \langle x_1, x_2, \ldots, x_n \rangle$. With this notation, 13.3 becomes:

$$
c_{\text{map}} = \arg\max_{c_j \in C} P(\langle x_1, x_2, \ldots, x_n \rangle | c_j) P(c_j) \qquad (13.4)
$$

To determine the MAP class, we need to estimate the parameters of the model: the priors and the conditional probabilities. Unfortunately, the conditional probabilities in Equation 13.4 cannot be estimated reliably. Even if the n attributes were binary, our Bayesian model would have $2^n|C|$ different parameters, one for each possible combination of n attribute values and a class. This being a very large quantity, estimating these parameters reliably is challenging unless our training collection is astronomical in size.



▶ **Figure 13.2** Naive Bayes conditional independence assumption. The variables X_i are independent of each other given the class *China*.

CONDITIONAL INDEPENDENCE ASSUMPTION

To reduce the number of parameters, we make what is called the Naive Bayes *Conditional Independence Assumption*. We assume that attributes are independent of each other given the class:

$$P(\langle x_1, x_2, \ldots, x_n \rangle | c_j) = \prod_{1 \leq i \leq n} P(x_i | c_j)$$

The maximization we perform to determine the class then becomes:

$$c_{\text{map}} = \arg\max_{c_j \in C} P(c_j) \prod_{1 \leq i \leq n} P(x_i | c_j) \tag{13.5}$$

FEATURE

Now we switch from a generic instance space $\mathbb{X}$ to the one typical of Naive Bayes in text classification where each attribute corresponds to a word[1] – and is then more commonly called a *feature*. The conditional independence assumption can be illustrated with the graphical model in Figure 13.2. The class generates each of the five word attributes with a certain probability, independent of the absence or presence of the other words. The fact that a document in the class *China* contains the word Taipei does not make it more likely or less likely that it also contains Beijing. Of course this assumption does not really hold for text data. Words *are* conditionally dependent on each other. But NB models perform well despite the conditional independence assumption. Equation 13.5 can now be rewritten as follows, letting w_i denote the ith word in the role of the ith feature:

$$c_{\text{map}} = \arg\max_{c_j \in C} P(c_j) \prod_{1 \leq i \leq n} P(w_i | c_j) \tag{13.6}$$

1. It would be more accurate to talk about terms, not words since text classification systems often preprocess text using the same normalization procedures as indexers (such as downcasing and stemming). We follow convention in the three text classification chapters and use the terms word and term interchangeably.

With the independence assumption, we only need to estimate $O(n|C|)$ independent parameters $P(w_i|c_j)$, one for each word-class combination, rather than $O(2^n|C|)$. The independence assumption reduces the number of parameters to be estimated by several orders of magnitude.

How do we perform the estimation? We first try the maximum likelihood estimate (MLE), which corresponds to the most likely value of each parameter given the training data. For the priors this estimate is:

$$\hat{P}(c_j) = \frac{N_j}{N} \tag{13.7}$$

where N_j is the number of documents of class c_j and N is the total number of documents.

To estimate the conditional probabilities, we first need to settle on a model of document generation. We choose the multinomial model here. Below we discuss a second model, the binomial model. The multinomial model was introduced for adhoc retrieval in Chapter 12: Equations (12.3) and (12.4) (page 181). There we estimated the probability of a *document model* (M_d) generating a *query word* (q). Here we want the probability of a *class model* generating a *document word* (or, more precisely, a word token). The MLE estimate for the conditional probabilities $\hat{P}(w_i|c_j)$ is then:

$$\hat{P}(w_i|c_j) = \frac{T_{ji}}{\sum_i T_{ji}}$$

where T_{ji} is the number of occurrences of w_i in training documents from class c_j, including multiple occurrences of a term in a document.

The problem with the MLE estimate is that it is zero for a word-class combination that did not occur in the training data. In the model in Figure 13.2, if occurrences of the word WTO in the training data only occurred in *China* documents, then the MLE estimates for the other classes, for example *UK*, will be zero:

$$\hat{P}(\text{WTO}|UK) = 0$$

Now a document consisting of the sentence *Britain is a member of the WTO* will get a conditional probability of zero for *UK* since we're multiplying the conditional probabilities for all words in Equation 13.5. Clearly, the model should assign a high probability to the *UK* class since the word Britain occurs. The problem is that the zero probability for WTO cannot be "conditioned away," no matter how strong the evidence for the class *UK* from other features. This is the problem of data *sparseness*. We encountered the same problem when applying the multinomial model to adhoc retrieval in Chapter 12 (page 181).

SPARSENESS

ADD-ONE SMOOTHING
LAPLACE SMOOTHING

To eliminate zeros, we use *add-one* or *Laplace* smoothing, which simply

adds one to each count:

$$\hat{P}(w_i|c_j) = \frac{T_{ji}+1}{\sum_i(T_{ji}+1)}$$

Laplace smoothing can be interpreted as a uniform prior (each word occurs once for each class) that is then updated as evidence from the training data comes in.[2]

We have left the exact definition of the w_i in $\hat{P}(w_i|c_j)$ vague so far. The index i might suggest that we have different random variable X_i for each position in the document. But the position of a word in a document by itself does not carry information about the class. There is a difference between *China sues France* and *France sues China*, but the occurrence of China in position 1 vs. 3 of the document is not useful for classification since we look at each word separately in classification. This way of processing the evidence is what we committed to when we made the conditional independence assumption.

Also, with different random variable X_i for each position i, we would have to estimate a different set of parameters for each i. The probability of bean appearing as the first word of a *coffee* document would be different from it appearing as the second word etc. This would again cause problems with data sparseness in estimation.

POSITIONAL INDEPENDENCE

For these reasons, we make a second independence assumption, *positional independence*: The conditional probabilities for a word are the same independent of position in the document.

$$P(X_{k_1} = w|c_j) = P(X_{k_2} = w|c_j)$$

for all positions k_1, k_2, words w and classes c_j. X_k is a random variable whose values are words appearing in position k of the document. Thus, we have a single multinomial distribution of words that is valid for all positions k_i and can use X as the symbol for its random variable. Positional independence is equivalent to adopting the *bag of words* model, which we introduced in the context of adhoc retrieval in Chapter 6 (page 88).

BAG OF WORDS

We have now introduced all the elements we need for training and applying an NB classifier. The complete algorithm is described in Figure 13.3.

What is the time complexity of Naive Bayes? The complexity of computing the parameters is $O(|C||V|)$ since the set of parameters consists of $|C||V|$ conditional probabilities and $|C|$ priors. The preprocessing necessary for computing the parameters (extracting the vocabulary, counting words etc.) can be done in one pass through the training data. The time complexity of this component is therefore $O(|D|L_d)$ where $|D|$ is the number of documents and

2. Note that this is a prior probability for the occurrence of a *word* as opposed to the prior probability of a *class* which we estimate in Equation (13.7) on the document level.

Training
From training data D, extract vocabulary V
$N \leftarrow$ number of documents in D
Calculate parameters $P(c_j)$ and $P(X = w_i|c_j)$
  For each c_j in C do
    $N_j \leftarrow$ number of documents in c_j
    $\hat{P}(c_j) \leftarrow \frac{N_j}{N}$
    $text_j \leftarrow$ the text of all documents in class c_j
    For each word $w_i \in V$:
      $T_{ji} \leftarrow$ number of occurrences of w_i in $text_j$
      $\hat{P}(X = w_i|c_j) = \frac{T_{ji}+1}{\sum_i (T_{ji}+1)}$

Testing
Positions $S \leftarrow$ all positions in current document that contain words in V
Return c_{NB}, where
  $c_{\text{NB}} = \arg\max_{c_j \in C} \; P(c_j) \prod_{k \in S} P(X = w_k|c_j)$

► **Figure 13.3** Naive Bayes algorithm (multinomimal model): Training and testing

mode	time complexity
training	$O(\|D\|L_d + \|C\|\|V\|)$
testing	$O(\|C\|L_t)$

► **Table 13.1** Training and test times for Naive Bayes.

L_d is the average length of a document. The time complexity of classifying a document is $O(|C|L_t)$ (where L_t is the average length of a test document) as can be observed from Figure 13.3.

Table 13.1 summarizes the time complexities. In general, we have $|C||V| < |D|L_d$, so both training and testing complexity is linear in the time it takes to scan the data. Since we have to look at the data at least once, Naive Bayes can be said to have optimal time complexity. Its efficiency is one reason why Naive Bayes is such a popular text classification algorithm.

Implementation note. In NB classification, many conditional probabilities are multiplied, one for each vocabulary word in the document. This can result in a floating point underflow. It is therefore better to perform the computation by adding logarithms of probabilities instead of multiplying probabilities. The class with the highest log probability score is still the most probable since $\log(xy) = \log(x) + \log(y)$ and the logarithm function is monotonic. Hence, the maximization that is actually done in most implementations of

	multinomial model	binomial model
event model	generation of token	generation of document
random variable(s)	$X = w$ iff w occurs at given pos	$U_w = 1$ iff w occurs in doc
document representation	$d =< w_1, w_2, \ldots, w_{L_d} >, w_k \in V$	$d =< e_1, e_2, \ldots, e_M >, e_i \in \{0, 1\}$
parameter estimation	$\hat{P}(X = w\|c_j)$	$\hat{P}(U_w = e\|c_j)$
decision rule: maximize	$\hat{P}(c_j) \prod_{k \in S} \hat{P}(X = w_k\|c_j)$	$\hat{P}(c_j) \prod_{w \in V} \hat{P}(U_w = e_w\|c_j)$
multiple occurrences	taken into account	ignored
length of docs	can handle long docs	works best for short docs
# features	can handle many	works best with few
estimate for term the	$\hat{P}(X = \text{the}\|c_j) \approx 0.05$	$\hat{P}(U_{\text{the}} = 1\|c_j) \approx 1.0$

► **Table 13.2** Multinomial vs. Binomial model.

Naive Bayes is:

$$c_{\text{NB}} = \arg\max_{c_j \in C} \log P(c_j) + \sum_{k \in S} \log P(X = w_k|c_j)$$

13.3 The multinomial versus the binomial model

MULTINOMIAL MODEL

MULTIVARIATE BINOMIAL MODEL

As we already mentioned there are two different ways we can set up an NB classifier. The model we have worked with so far is the *multinomial model*. It generates one word from the vocabulary in each position of the document. An alternative is the *multivariate binomial model* or *multivariate Bernoulli model* – or simply: *binomial model*. It is equivalent to the BIM of Section 11.3 (page 166), which generates an indicator for each word of the vocabulary, either 0 indicating absence or 1 indicating presence of the word in the document. We compare the two models in Table 13.2.

The different generation models imply different estimation strategies for the parameters. The binomial model estimates $\hat{P}(U_w = 1|c_j)$ as the *fraction of documents* of class c_j that contain word w where W_w is a random variable whose values 0 and 1 indicate absence and presence, respectively, of word w. In contrast, the multinomial model estimates $\hat{P}(X = w|c_j)$ as the *fraction of tokens* or *fraction of positions* in documents of class c_j that contain word w. This difference between document generation vs. token generation as the underlying event affects how multiple occurrences are used in classification. The binomial model uses binary occurrence information, ignoring the number of occurrences, whereas the multinomial model keeps track of multiple occurrences. As a result, the binomial model typically makes many mistakes when classifying long documents. For example, it may assign a book to the class *China* because of a single occurrence of the word China.

	$P(c_1\|d)$	$P(c_2\|d)$	class selected
actual probability	0.6	0.4	c_1
Naive Bayes estimate	0.009	0.001	c_1
$\sim$, normalized	0.9	0.1	c_1

► **Table 13.3** Correct estimation implies accurate prediction, but accurate prediction does not imply correct estimation.

13.4 Properties of Naive Bayes

Naive Bayes is so called because of its "naive" independence assumptions. The conditional independence assumption states that features are independent of each other given the class. This is hardly ever true for words in documents. In many cases, the opposite is true. The pairs hong and kong or london and english in Figure 13.4 are examples of highly dependent words. In addition, the multinomial model makes an assumption of positional independence. The binomial model ignores positions in documents altogether since it only cares about absence or presence. This "bag of words" model discards all information that is communicated by the order of words in natural language sentences. These independence assumptions are so sweeping that Naive Bayes is sometimes called *Idiot Bayes*. How can Naive Bayes be a good text classifier when its model of natural language is so oversimplified?

IDIOT BAYES

The answer lies in a paradox. Even though the *probability estimates* of Naive Bayes are of low quality, its *classification decisions* are surprisingly good. Unless they are normalized, NB probability estimates tend to be close to 0 since multiplying large numbers of conditional probabilities produces numbers close to 0. The winning class usually has a much larger probability than the other classes, so after normalization its probability will be close to 1. In either case, the normalized NB probability is seldom a good estimate of the actual probability. But the classification decision is based on which class gets the highest score. It does not matter how accurate the probabilities are. An example is shown in Table 13.3. Even though Naive Bayes fails to correctly estimate the actual probabilities, it assigns a higher probability to c_1 and therefore assigns d to the correct class. *Correct estimation implies accurate prediction, but accurate prediction does not imply correct estimation.* Naive Bayes is a classifier that estimates badly, but classifies well.

Even if it is not the method with the highest accuracy for text, Naive Bayes has many virtues that make it a strong contender for text classification. In contrast to methods like decision trees, it excels if there are many equally important features that jointly contribute to the classification decision. It is also somewhat robust to noise features and concept drift (the gradual change over time of the concept underlying a class like *US president* from Bill Clinton to

George W. Bush, see Section 13.7). Its main strength is its efficiency: Training and classification can be accomplished with one pass over the data. Because it combines efficiency with good accuracy it is often used as a baseline in text classification research. It is often the method of choice if: (i) squeezing out a few extra percentage points of accuracy is not worth the trouble in a text classification application, (ii) a very large amount of training data is available and there is more to be gained from training on a lot of data than using a better classifier on a smaller training set, or (iii) if its robustness to concept drift can be exploited.

In this book, we discuss Naive Bayes as a classifier for text. The independence assumptions do not hold for text. However, it can be shown that Naive Bayes is an optimal classifier in domains where the independence assumptions do hold.

13.5 Feature selection

Feature selection in text classification serves two main purposes. First, it makes training and applying a classifier more efficient by decreasing the size of the effective vocabulary. This is of particular importance for classifiers that, unlike Naive Bayes, are expensive to train. Secondly, feature selection often increases classification accuracy. By eliminating noise features, text classifiers avoid overfitting – basing decisions on rare events in the training set that do not generalize to test data. Without noise features, classification is more accurate. Of the two NB models, the binomial model is particularly sensitive to noise features. A well performing binomial NB classifier requires some form of feature selection.[3]

This section addresses feature selection for a single binary classifier. Section 13.5.4 briefly discusses optimizations for operational systems with a large number of binary classifiers.

13.5.1 Mutual information

MUTUAL INFORMATION

A common feature selection method is *mutual information* (MI):

$$I(W,C) = \sum_{e_w \in \{0,1\}} \sum_{e_c \in \{0,1\}} P(W = e_w, C = e_c) \log_2 \frac{P(W = e_w, C = e_c)}{P(W = e_w) P(C = e_c)} \tag{13.8}$$

3. Feature selection corresponds to two different formal processes in the two NB models. In the binomial model, the dimensionality of the underlying document representation is reduced. In the multinomial model, the sample space of the multinomial random variable is reduced – it has fewer possible outcomes after feature selection. As is customary, we call both of these processes feature selection here.

UK	
london	0.1925
uk	0.0755
british	0.0596
stg	0.0555
britain	0.0469
plc	0.0357
england	0.0238
pence	0.0212
pounds	0.0149
english	0.0126

China	
china	0.0997
chinese	0.0523
beijing	0.0444
yuan	0.0344
shanghai	0.0292
hong	0.0198
kong	0.0195
xinhua	0.0155
province	0.0117
taiwan	0.0108

poultry	
poultry	0.0013
meat	0.0008
chicken	0.0006
agriculture	0.0005
avian	0.0004
broiler	0.0003
veterinary	0.0003
birds	0.0003
inspection	0.0003
pathogenic	0.0003

coffee	
coffee	0.0111
bags	0.0042
growers	0.0025
kg	0.0019
colombia	0.0018
brazil	0.0016
export	0.0014
exporters	0.0013
exports	0.0013
crop	0.0012

elections	
election	0.0519
elections	0.0342
polls	0.0339
voters	0.0315
party	0.0303
vote	0.0299
poll	0.0225
candidate	0.0202
campaign	0.0202
democratic	0.0198

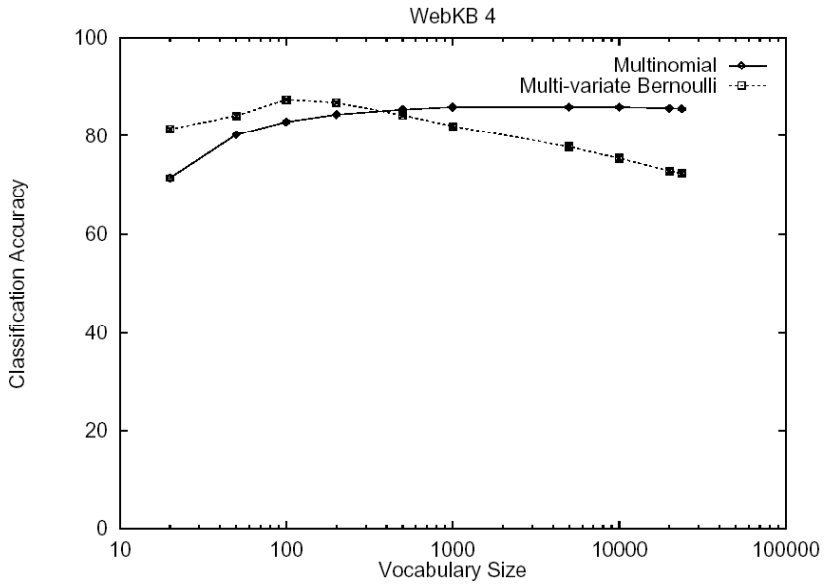
sports	
soccer	0.0681
cup	0.0515
match	0.0441
matches	0.0408
played	0.0388
league	0.0386
beat	0.0301
game	0.0299
games	0.0284
team	0.0264

► **Figure 13.4** Features with high mutual information scores for six Reuters-RCV1 classes.

where the value of W indicates presence or absence of the word w in a document and the value of C indicates membership in class c. To select k words $w_1, \ldots, w_k$ for class c, compute the values $I(W_1, C), \ldots, I(W_M, C)$ and select the k words with the highest values. Mutual information measures how much information – in the information-theoretic sense – a word contains about the class. If a word's distribution is the same in the class as it is in the collection as a whole, then $I(W, C) = 0$. MI reaches its maximum value if the word is a perfect indicator for class membership, that is, if the word is present in a document if and only if the document is in the class.

Figure 13.4 shows words with high mutual information scores for the six classes in Figure 13.1.[4] The selected words (e.g., london, uk, british for the class *UK*) are of obvious utility for making classifying decisions for their respective classes. At the bottom of the list for *UK* we find words like peripherals

4. Feature scores were computed on the first 100,000 documents, except for poultry, a rare class, for which 800,000 documents were used. Numbers and other special words were omitted from the top 10 lists.



► **Figure 13.5** Effect of feature set size on accuracy for multinomial and binomial (multivariate Bernoulli) models.

and tonight (not shown in the figure) that are clearly not helpful in deciding whether the document is in the class. As you might expect, keeping the informative terms and throwing away the non-informative ones tends to reduce noise and improve the classifier's accuracy.

Such an accuracy increase can be observed in Figure 13.5 where classification accuracy (the fraction of documents classified correctly) is shown as a function of vocabulary size after feature selection. The data set used for this experiment is a collection of web pages from universities. Classes are *student page*, *faculty page*, *department page* etc. Accuracy initially goes up as we add features to the vocabulary. It then peaks at about 100 features for the binomial and at about 10,000 features for the multinomial. More importantly, accuracy goes down rather steeply for the binomial once it has reached its peak because it is sensitive to noise features. There is no such decline for the multinomial: it takes the number of occurrences into account in parameter estimation and classification and therefore better distinguishes between good and bad discriminators of the class.

13.5.2 χ^2 feature selection

χ^2 FEATURE SELECTION

Another popular feature selection method is χ^2 (pronounced chi-square with

p	χ^2 critical value
0.1	2.71
0.05	3.84
0.01	6.63
0.005	7.88
0.001	10.83

► **Table 13.4** Critical values of the χ^2 distribution with one degree of freedom. For example, if W and C are independent, then $\Pr(X^2 > 6.63) < 0.01$. So for $X^2 > 6.63$ the assumption of independence can be rejected with 99% confidence.

chi as in kiting). In statistics, the χ^2 test is applied to test the independence of two random variables. In feature selection, these two variables are occurrence of the word (W) and occurrence of the class (C). We then rank words with respect to the following quantity:

$$X^2(W,C) = \sum_{e_w \in \{0,1\}} \sum_{e_c \in \{0,1\}} \frac{(O_{e_w e_c} - E_{e_w e_c})^2}{E_{e_w e_c}} \tag{13.9}$$

where, for example, O_{11} is the *observed* frequency of w and c occurring together (number of documents in c that contain w) and E_{11} is the *expected* frequency of w and c occurring together assuming that word and class are independent.

The expected frequency is the product of the marginals (see definition on page 119) times the number of documents. Here is an example of computing E_{11} and applying the test to the class *poultry* and the word export.

$$\begin{aligned} E_{11} &= N \times P(w) \times P(c) = N \times \frac{O_{11} + O_{01}}{N} \times \frac{O_{11} + O_{10}}{N} \\ &= N \times \frac{49 + 141}{N} \times \frac{49 + 27652}{N} \approx 6.6 \end{aligned}$$

where $N = O_{11} + O_{01} + O_{10} + O_{11} = 801{,}948$

	export $= 1$		export $= 0$	
poultry $= 1$	$O_{11} = 49$	$E_{11} \approx 6.6$	$O_{10} = 141$	$E_{10} \approx 183.4$
poultry $= 0$	$O_{01} = 27{,}652$	$E_{01} \approx 27{,}694.4$	$O_{00} = 774{,}106$	$E_{00} \approx 774{,}063.6$

$$X^2(W,C) = \sum_{e_w \in \{0,1\}} \sum_{e_c \in \{0,1\}} \frac{(O_{e_w e_c} - E_{e_w e_c})^2}{E_{e_w e_c}} \approx 284$$

It can be shown that if W and C are independent, then $X^2 \sim \chi^2$, where χ^2 is the χ^2 distribution, one of the best known distributions in statistics.

Some critical values of the distribution are given in Table 13.4 for one degree of freedom, the number of degrees in this case. A value of X^2 that is unexpectedly high indicates that the hypothesis of independence is incorrect. In our example, $X^2 \approx 284 > 10.83$. Based on Table 13.4, we can reject the hypothesis that *poultry* and export are independent with only a 0.001 chance of being wrong. Equivalently, we say that the outcome $X^2 \approx 284 > 10.83$ is *statistically significant* at the 0.001 level. If the two events are dependent, then the occurrence of one makes the occurrence of the other more likely (or less likely), so it should be helpful as a feature. This is the rationale of χ^2 feature selection.

STATISTICAL SIGNIFICANCE

An arithmetically simpler way of computing X^2 is the following:

$$X^2(W,C) = \frac{(O_{11}+O_{10}+O_{01}+O_{00}) \times (O_{11}O_{00}-O_{10}O_{01})^2}{(O_{11}+O_{01}) \times (O_{11}+O_{10}) \times (O_{10}+O_{00}) \times (O_{01}+O_{00})} \tag{13.10}$$

This is equivalent to Equation 13.9 (Exercise 13.11).

From a statistical point of view, χ^2 feature selection is problematic. For a test with one degree of freedom, the so-called Yates correction should be used (see Section 13.7), which makes it harder to reach statistical significance. Also, whenever a statistical test is used multiple times, then the probability of getting at least one error increases. If 1000 hypotheses are rejected, each with 0.05 error probability, then $0.05 * 1000 = 50$ calls of the test will be wrong on average. However, in text classification it rarely matters whether a few additional terms are added to the feature set or removed from it. Rather, the *relative* importance of features is important. The typical application of χ^2 (and MI) is to select the top k features where k is chosen based on some knowledge about the problem or by cross-validation. As long as χ^2 feature selection only ranks features with respect to their usefulness and is not used to make statements about statistical dependence or independence of variables, we need not be overly concerned that it does not adhere strictly to statistical theory.

13.5.3 Frequency-based feature selection

A third feature selection method is *frequency-based feature selection*, i.e., selecting the words that are most common in the class. This method will select some frequent words that have no specific information about the class, for example, the days of the week (Monday, Tuesday, ...), which are frequent across classes in newswire text. When hundreds or thousands of features are selected, then frequency-based feature selection does surprisingly well, with a typical relative accuracy decrease of around 5% (for example, from 80% to 76% accuracy). If somewhat suboptimal accuracy is acceptable, then

frequency-based feature selection is a good alternative to more complex methods. However, this does not apply when only a few features (on the order of 10) are selected. In this case, mutual information and χ^2 perform much better than frequency-based selection.

13.5.4 Comparison of feature selection methods

The selection criteria of mutual information and χ^2 are quite different. The independence of term and class can sometimes be rejected with high confidence even if the term carries little information useful for classification. This is particularly true for rare terms. If a word occurs once in a large collection and that one occurrence is in the *poultry* class, then this is statistically significant. But a single occurrence is not very informative according to the information-theoretic definition of information. Because its criterion is significance, χ^2 selects more rare terms (which are often less reliable indicators) than mutual information. But the selection criterion of mutual information also does not necessarily select the terms that maximize classification accuracy.

Despite the differences between the two methods, the classification accuracy of feature sets selected with χ^2 and MI does not seem to differ systematically. In most text classification problems there are a few strong indicators and many weak indicators. As long as all strong indicators and a large number of weak indicators are selected, accuracy is expected to be good. Both methods do this.

GREEDY FEATURE SELECTION

All three methods – MI, χ^2 and frequency-based – are *greedy* methods. They may select features that contribute no information in addition to previously selected features. In Figure 13.4, kong is selected as the seventh word even though it is highly correlated with previously selected hong and therefore redundant. Although such redundancy can negatively impact accuracy, non-greedy methods (see Section 13.7 for references) are rarely used in text classification due to their computational cost.

In an operational system with a large number of binary classifiers, it is desirable to select a single set of features that works well for all classifiers. One way of doing this is to compute the χ^2 statistic for an $n \times 2$ table where the columns are occurrence and non-occurrence of the word and each row corresponds to one of the classes (or, more precisely, each classifier's positive class). We can then select the k words with the highest χ^2 statistic as before.

More commonly, feature selection statistics are first computed separately for each classifier and then combined (for example, by averaging) into a single figure of merit. Classification accuracy almost always decreases when selecting k common features for a system with n classifiers as opposed to n different sets of size k. But the gain in efficiency due to a common document representation is often worth such a decrease.

```
<REUTERS TOPICS=''YES'' LEWISSPLIT=''TRAIN''
  CGISPLIT=''TRAINING-SET'' OLDID=''12981'' NEWID=''798''>
<DATE> 2-MAR-1987 16:51:43.42</DATE>
<TOPICS><D>livestock</D><D>hog</D></TOPICS>
<TITLE>AMERICAN PORK CONGRESS KICKS OFF TOMORROW</TITLE>
<DATELINE> CHICAGO, March 2 - </DATELINE><BODY>The American Pork
  Congress kicks off tomorrow, March 3, in Indianapolis with 160
  of the nations pork producers from 44 member states determining
  industry positions on a number of issues, according to the
  National Pork Producers Council, NPPC.
  Delegates to the three day Congress will be considering 26
  resolutions concerning various issues, including the future
  direction of farm policy and the tax law as it applies to the
  agriculture sector. The delegates will also debate whether to
  endorse concepts of a national PRV (pseudorabies virus) control
  and eradication program, the NPPC said. A large
  trade show, in conjunction with the congress, will feature
  the latest in technology in all areas of the industry, the NPPC
  added. Reuter
&#3;</BODY></TEXT></REUTERS>
```

► **Figure 13.6** A sample document from the Reuters-21578 collection.

class	# train	# test	class	# train	# test
earn	2877	1087	*trade*	369	119
acquisitions	1650	179	*interest*	347	131
money-fx	538	179	*ship*	197	89
grain	433	149	*wheat*	212	71
crude	389	189	*corn*	182	56

► **Table 13.5** The ten largest classes in the Reuters-21578 collection with number of documents in training and test sets.

13.6 Evaluation of text classification

Historically, the classic Reuters-21578 collection was the main benchmark for text classification evaluation. This is a collection of 21,578 newswire articles, but which is much smaller than and predates the Reuters RCV1 collection discussed in Chapter 4 (page 52). The articles are assigned classes from a set of 118 categories. A document may be assigned several classes or none, but the commonest case is single assignment (documents with at least one class received an average of 1.24 classes). The standard approach to this any-of or

	class 1	
	truth: yes	truth: no
call: yes	10	10
call: no	10	970

	class 2	
	truth: yes	truth: no
call: yes	90	10
call: no	10	890

	pooled table	
	truth: yes	truth: no
call: yes	100	20
call: no	20	1860

► **Table 13.6** Macro- and microaveraging. "Truth" is the true class and "call" the decision of the classifier. In this example, macroaveraged precision is $[10/(10+10)+10/(10+90)]/2 = (0.5+0.9)/2 = 0.7$. Microaveraged precision is $100/(100+20) \approx 0.83$.

multivalue problem (Chapter 14, page 227) is to learn 118 binary classifiers, one for each class. For each of these classifiers, we can measure recall, precision, and accuracy. In recent work, people almost invariably use the *ModApte split* which includes only documents with at least one class, and comprises 9603 training documents and 3299 test documents. The distribution of documents in classes is quite uneven, and some work classifies only for the 10 largest classes. They are listed in Table 13.5. A typical document with topics is shown in Figure 13.6.

MODAPTE SPLIT

We already saw one evaluation measure for text classifiers, accuracy, in Figure 13.5. We know from Section 8.1.2 (page 112) that classification accuracy is the proportion of correct decisions. This measure is appropriate if the population rate of the class is high, perhaps 10–20% and higher. But as was discussed in Section 8.1.2, accuracy is not a good measure for "small" classes since always saying "no", a strategy that defeats the purpose of building a classifier, will achieve high accuracy. The "always no" classifier achieves 99% accuracy for a class with population rate 1%. For small classes, precision, recall and F_1 are better measures. We will use *effectiveness* as a generic term for measures that evaluate the quality of classification decisions, including precision, recall, F_1 and accuracy. In this book, *performance* refers to the *efficiency* of classification and information retrieval systems. However, many researchers mean quality, not efficiency of text classification when they use the term performance.

EFFECTIVENESS

PERFORMANCE

As in the case of Reuters-21578 and Reuters-RCV1, most other evaluation benchmarks for text classification also contain a *set* of classes rather than a single class. This means that measures for individual classes have to be combined into one aggregate measure. There are two methods for doing this. *Macroaveraging* computes a simple average over classes. *Microaveraging* pools per-document decisions across classes, and then computes an effectiveness measure on the pooled contingency table. Table 13.6 gives an example.

MACROAVERAGING

MICROAVERAGING

The differences between the two methods can be large. Macroaveraging

Method	F_1 micro-avg.	F_1 macro-avg.
multinomial NB	0.80	0.47
SVM	0.89	0.60

► **Table 13.7** Experimental results for F_1 on Reuters-21578 (all classes).

gives equal weight to each class, whereas microaveraging gives equal weight to each per-document classification decision. Since the F_1 measure ignores true negatives and its magnitude is mostly determined by the number of true positives, large classes dominate small classes in microaveraging. In the example, microaveraged precision (0.83) is much closer to the precision of c_2 (0.9) because c_2 is five times larger than c_1. Microaveraged results are therefore really a measure of effectiveness on the large classes in a test collection. To get a sense of effectiveness on small classes, compute macroaveraged results.

Table 13.7 gives microaveraged and macroaveraged effectiveness of Naive Bayes for 90 classes in Reuters-21578. To give a sense of the relative effectiveness of Naive Bayes, we compare it to support vector machines (Chapter 15). Naive Bayes has a microaveraged F_1 of 80% which is 9% less than the SVM (89%), a 10% relative decrease. So there is a surprisingly small effectiveness penalty for its simplicity and efficiency. However, on small classes, many of which only have on the order of ten positive examples in the training set, Naive Bayes does much worse. Its macroaveraged F_1 is 13% below optimal, a 22% relative decrease.

13.7 References and further reading

Sebastiani (2002) gives a comprehensive review of text classification methods and results. Lewis (1998) focuses on the history of Naive Bayes classification. McCallum and Nigam (1998) discuss binomial and multinomial models and their accuracy for different collections. Figure 13.5 is from their paper. Friedman (1997) and Domingos and Pazzani (1997) analyze why Naive Bayes performs well although its probability estimates are poor. Ng and Jordan (2001) show that Naive Bayes is sometimes (though rarely) superior to discriminative methods because it more quickly reaches its optimal error rate. The problem of concept drift is discussed by Forman (2006) and Hand (2006).

POINTWISE MUTUAL INFORMATION

Yang and Pedersen (1997) review feature selection methods and their impact on classification effectiveness. They find that *pointwise mutual information* is not competitive with other feature selection methods. Yang and Pedersen refer to standard mutual information as information gain (see Ex-

(1) He moved from London, Ontario, to London, England.
(2) He moved from London, England, to London, Ontario.
(3) He moved from England to London, Ontario.

► **Table 13.8** A set of documents for which the Naive Bayes independence assumptions are problematic.

ercise 13.10, page 210). Snedecor and Cochran (1989) is a good reference for the χ^2 test in statistics, including the Yates' correction for continuity for two-by-two tables. Dunning (1993) discusses problems of the χ^2 test when counts are small. Non-greedy feature selection techniques are described by Hastie et al. (2001). Table 13.7 is based on (Li and Yang 2003).

HIERARCHICAL CLASSIFICATION

A number of approaches for *hierarchical classification* have been developed in order to deal with the common situation where the classes to be assigned have a natural hierarchical organization (Koller and Sahami 1997, Weigend et al. 1999, Dumais and Chen 2000). In a recent large study using the Yahoo! directory, Liu et al. (2005) conclude that hierarchical classification noticeably if still modestly outperforms flat classification.

13.8 Exercises

Exercise 13.1

Which of the documents in Table 13.8 have identical and different bag of words representations for (a) the binomial model (b) the multinomial model?

Exercise 13.2

The rationale for the positional independence assumption is that there is no useful information in the fact that a word occurs in position k of a document. Try to find exceptions. Consider formulaic documents with a fixed document structure.

Exercise 13.3

The class priors in Figure 13.3 are computed as the fraction of *documents* in the class as opposed to the fraction of *tokens* in the class. Why?

Exercise 13.4

Why is $|C||V| < |D|L_d$ in Table 13.1 expected to hold for most text collections?

Exercise 13.5

Why would a more complete name for the multinomial model be *univariate* multinomial model?

Exercise 13.6

Table 13.2 gives binomial and multinomial estimates for the word the. Explain the difference.

Exercise 13.7

What are the values of $I(W,C)$ and $X^2(W,C)$ if word and class are completely independent? What are the values if they are completely dependent?

Exercise 13.8

The feature selection method in Equation 13.8 is most appropriate for the binomial model. Why? How could one modify it for the multinomial model?

Exercise 13.9

Compute $I(\mathsf{export}, \mathsf{poultry})$ for the data on page 203.

Exercise 13.10

INFORMATION GAIN

Features can also be selected according to *information gain* (IG). Information gain is defined as:

$$\mathrm{IG}(D,f) = H(p_D) - \sum_{x \in \{D_{f^+}, D_{f^-}\}} \frac{|x|}{|D|} H(p_x)$$

where H is entropy and D, D_{f^+}, and D_{f^-} are the entire collection, the subcollection of documents with feature f, and the subcollection of documents without feature f, respectively. p_A is the class distribution in (sub)collection A, e.g., $p_A(c) = 0.25, p_A(\neg c) = 0.75$ if a quarter of the documents in A are in class c.

Show that mutual information and information gain are equivalent.

Exercise 13.11

Show that the two X^2 formulae (Equations (13.9) and (13.10)) are equivalent.

Exercise 13.12

In the χ^2 example on page 203 we have $|O_{11} - E_{11}| = |O_{10} - E_{10}| = |O_{01} - E_{01}| = |O_{00} - E_{00}|$. Show that this holds in general.

Exercise 13.13

χ^2 and mutual information do not distinguish between positively and negatively correlated features. Since most good text classification features are positively correlated (i.e., they occur more often in the class than outside), one may want to explicitly rule out the selection of "negative" indicators. How would you do this?

Exercise 13.14

Your task is to classify words as English or not English. Words are generated by a source with the following distribution:

event	word	English?	probability
1	ozb	0	4/9
2	uzu	0	4/9
3	zoo	1	1/18
4	bun	1	1/18

(i) Compute the parameters (priors and conditionals) of a multinomial Naive Bayes classifier that uses the letters b, n, o, u, and z as features. Assume a training set that reflects the probability distribution of the source perfectly. Make the same independence assumptions that are usually made for a multinomial classifier that uses words as features for text classification. Compute parameters using smoothing, in which

computed-zero probabilities are smoothed into probability 0.01, and computed-nonzero probabilities are untouched. (This simplistic smoothing may cause $P(A) + P(\neg A) > 1$, which can be corrected if we correspondingly smooth all complementary probability-1 values into probability 0.99. For this exercise, solutions may omit this correction to simplify arithmetic.) (ii) How does the classifier classify the word zoo? (iii) Classify the word zoo using a multinomial classifier as in part (i), but do not make the assumption of positional independence. That is, estimate separate parameters for each position in a word. You only need to compute the parameters you need for classifying zoo.

14 *Vector space classification*

The document representation in Naive Bayes essentially is a vector of word counts. In this chapter, we adopt a different representation for classification, the vector space model, developed in Chapter 7. It represents each document as a vector with one real-valued component (e.g., a tf-idf weight) for each term. Thus, the instance space, the domain of the classification function γ, is $\mathbb{R}^{|V|}$ instead of $\mathbb{N}^{|V|}$. This chapter introduces a number of classification methods that operate on real-valued vectors.

CONTIGUITY HYPOTHESIS

The basic hypothesis in using the vector space model for classification is the *contiguity hypothesis*: *documents in the same class form a contiguous region and regions of different classes don't overlap.* There are many classification tasks, in particular the type of topic classification that we encountered in Chapter 13, where classes can be distinguished by word patterns. For example, documents in the class *China* tend to have high values on the dimensions like Chinese, Beijing, and Mao whereas documents in the class *UK* tend to have high values for London, British and Queen. Documents of the two classes therefore form distinct contiguous regions as shown in Figure 14.1 and we can draw a line that separates them and classifies new documents. How exactly this is done is the main topic of this chapter.

The contiguity hypothesis is a fair characterization of the distribution of *China* and *UK* documents. It is easy to come up with examples, however, where the contiguity hypothesis does not hold. If the task is to decide whether the author of a text is male or female, then the distribution of documents corresponding to the two classes may well be indistinguishable.

Whether or not a set of documents is mapped into a contiguous region depends on the particular choices we make for the document representation: type of weighting, stop list etc. To see that the document representation is crucial, consider the two classes *written by a group* vs. *written by a single person*. Frequent occurrence of the first person pronoun I is evidence for the single-person class. But that information is deleted from the document representation if we use a stop list.

The similarity measure we employ also influences classification decisions. Consider a representation of web pages that consists of ordinary word features as well as hyperlink features. Hyperlink features (e.g., http://www.michaeljackson.com) are often better descriptors of content than word features (e.g., the words Michael Jackson). Thus, we prefer similarity measures that weight hyperlink features higher than word features. In this chapter, we use inner product similarity, the most commonly used similarity measure in vector space classification and weight all dimensions of the space equally.

This chapter introduces two vector space classification methods, Rocchio and kNN. Rocchio classification (Section 14.1) divides the vector space into regions centered on centroids or prototypes, one for each class, computed as the center of mass of all documents in the class. Rocchio classification is simple and efficient, but inaccurate if classes are not approximately spheres with similar radii.

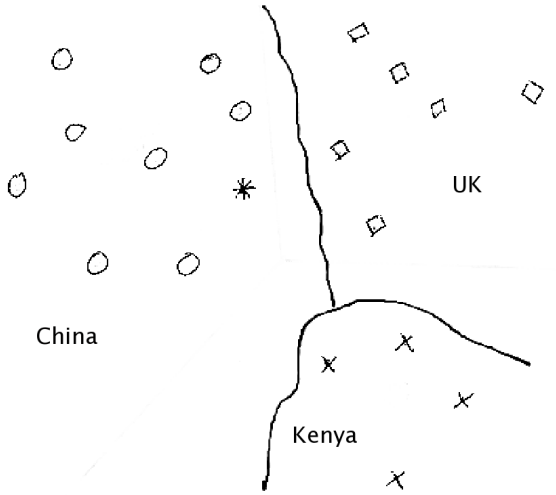
kNN or k nearest neighbor classification (Section 14.2) assigns the majority class of the k nearest neighbors to a test document. kNN requires no training in the sense of parameter estimation. If there exists a classifier with an error rate of 0, then 1NN will asymptotically approach that error rate as the training set increases. In practice, it is difficult to determine how close a given kNN classifier is to this optimum.

A large number of binary text classifiers can be viewed as simple linear classifiers – classifiers that compute a linear combination of the features and compare it to a threshold. Because of the bias-variance tradeoff (Section 14.3) more complex nonlinear models are not systematically better than linear models. Nonlinear models have more parameters to fit on a limited amount of training data and are therefore more likely to make mistakes for small and noisy data.

When applying binary classifiers to problems with more than two classes, we distinguish one-of tasks – mutually exclusive classes – and any-of tasks – a document can be assigned to any number of classes (Section 14.3.1). Binary classifiers solve any-of problems and can be combined to solve one-of problems.

14.1 Rocchio classification

Figure 14.1 shows three classes, *China*, *UK* and *Kenya*, in a two-dimensional (2D) space. Documents are shown as circles, diamonds and X's. The boundaries in the figure are chosen to separate the three classes, but are otherwise arbitrary. To classify a new document, depicted as a star in the figure, we determine the region it occurs in and assign it the class of that region. Our task in vector space classification is to devise algorithms that compute good



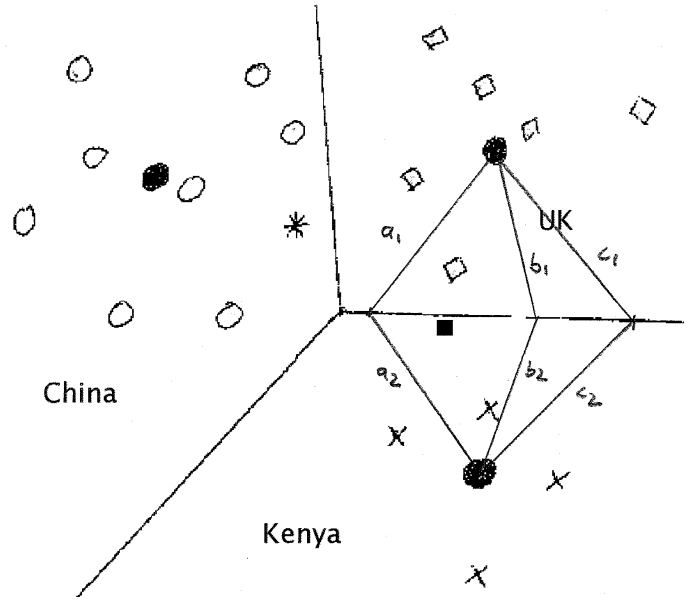
► **Figure 14.1** Vector space classification into three classes. Documents are classified as being relevant to China (circles), UK (diamonds), and Kenya (X's). Regions are demarcated by lines. A new document (shown as a star) is assigned the class of the region it occurs in, the class *China* in this case.

boundaries where "good" means high classification accuracy on data unseen during training.

In Figure 14.1 and the other illustrations in this chapter, we generally show points and distances between points instead of vectors and similarities between vectors. But as we know from Chapter 7 (page 100), Euclidean distance and cosine similarity are equivalent for proximity ranking of normalized vectors, so there is no harm in explaining vector space classification in terms of distances. Also, normalized vectors live on the unit sphere, a complication that the figures do not show. But unit normalization just means that there is one effective dimension less, e.g., unit vectors in 3D only vary on two dimensions – the 2D surface of the 3D sphere; and unit vectors in 1000D only vary on 999 dimensions – the 999D surface of the 1000D hypersphere. For our purposes, it does not matter whether a set of documents with effective dimensionality $n-1$ is represented as unit vectors in $\mathbb{R}^n$ or as unnormalized vectors in $\mathbb{R}^{n-1}$.

The main work we must do in vector space classification is to define the boundary lines between classes since they determine the classification decision. Perhaps the simplest way of doing this is to use *centroids*. The centroid

CENTROID



► **Figure 14.2** Rocchio classification. Centroids or prototypes (shown as solid circles) are computed as the average (or center of mass) of all points in a class. Boundaries between two classes are points of equal distance to the two centroids. For example, we have $a_1 = a_2, b_1 = b_2, c_1 = c_2$. The square is misclassified by Rocchio. It is a better fit for *UK* based on the overall distribution of points, but is closer to the centroid of *Kenya*.

of a class c is computed as the average or center of mass of its members:

$$\vec{\mu}(c) = \frac{1}{|c|} \sum_{\vec{x} \in c} \vec{x} \tag{14.1}$$

Three example centroids are shown in Figure 14.2. The boundary between two classes is then the set of points with equal distance from the two centroids. We again classify points in accordance with the region they fall into. Equivalently, we determine which centroid the point is closest to and assign it to the class of that centroid. The algorithm is summarized in Figure 14.3.

The assignment criterion in Figure 14.3 is inner product similarity. An alternative is cosine similarity:

$$\text{Assign the document to class } c = \arg\max_{c_j} \cos(\vec{\mu}(c_j), \vec{d})$$

The two assignment criteria are different since the centroid is in general not

Training
For each class c_j
  Compute centroid $\vec{\mu}(c_j)$
Testing: Classify test document $\vec{d}$
Assign the document to class $c = \arg\max_{c_j}(\vec{\mu}(c_j) \cdot \vec{d})$

► **Figure 14.3** Rocchio classification: Training and testing.

a vector that has unit length. However, centroids in most applications in information retrieval are computed for small regions of the surface of a unit hypersphere, so the length of the centroid is approximately one. But if we compute centroids for vectors distributed over large regions of the hypersphere, then their lengths can be very different from 1.0. In particular, the length of the centroid of two diametrically opposed vectors is zero. We present the inner product version of Rocchio classification here because it is simpler and more efficient. We need not compute the length of the Rocchio vectors as would be required by length normalization. We will also use centroids for clustering in Chapter 16. There, only inner product similarity allows a clean interpretation of the k-means clustering algorithm.

ROCCHIO CLASSIFICATION

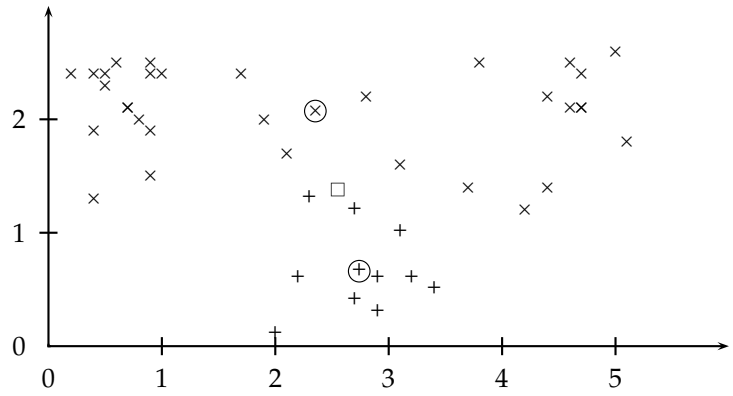
This centroid-based classification algorithm is a form of Rocchio relevance feedback (Section 9.1.1, page 132). It is therefore called *Rocchio classification*. The average of the relevant documents, corresponding to the most important component of the Rocchio vector in relevance feedback (Equation (9.3), page 135), is the centroid of the "class" of relevant documents. Centroids are also called *prototypes* in this context. Rocchio classification can be applied to $J > 2$ classes whereas Rocchio relevance feedback is designed to distinguish only two classes, relevant and non-relevant.

PROTOTYPE

In addition to respecting contiguity, the classes in Rocchio classification must also be of approximately spherical shape. In Figure 14.2, the solid square just below the boundary between *UK* and *Kenya* should intuitively be part of *UK* since *UK* is more scattered than *Kenya*. But the Rocchio classifier will put it in the *Kenya* category because details of the distribution of points in a class are ignored. Only the centroid is used for classification.

An even worse case for the assumption of sphericity is shown in Figure 14.4. The X class cannot be represented well with a single prototype because it has two centers. This type of *polymorphic class* will often be misclassified by Rocchio. A text classification example for polymorphism is a country like Burma, which changed its name to Myanmar in 1989. The two clusters before and after the name change need not be close to each other in space. We encountered the same problem with polymorphism in relevance feedback (Chapter 9, page 137).

POLYMORPHIC CLASS



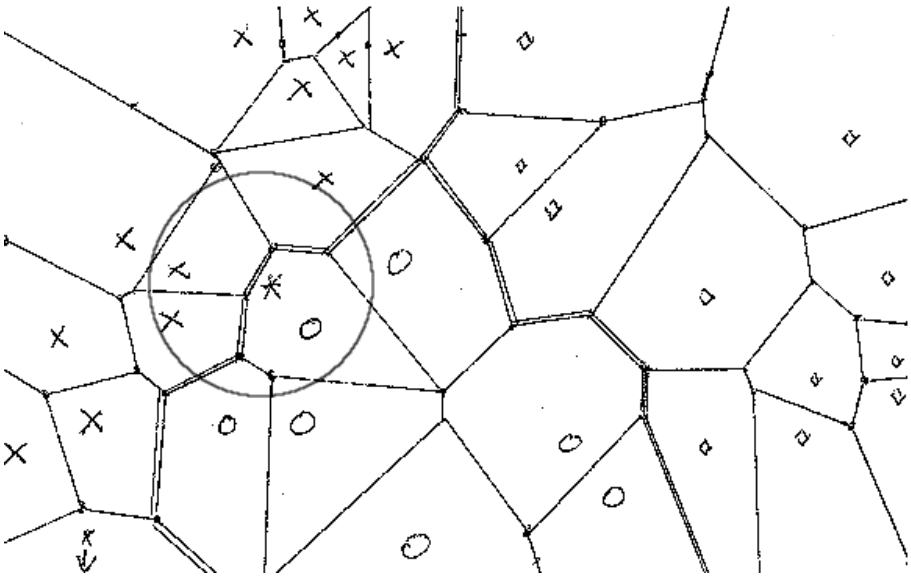
► **Figure 14.4** A polymorphic class with two different centers. The centroid of class X (upper circle) is not prototypical. It is located in a sparse region of the class between two dense regions. The square is misclassified as belonging to class X even though it would be a better fit for class +.

C	set of classes
D	training set
L_d	average length of a training document
$O(\|D\|L_d + \|C\|\|V\|)$	training time
L_t	average length of a test document
m_t	average vocabulary size of a test document
$O(L_t + \|C\|m_t)$	test time

► **Table 14.1** Training and test time for Rocchio classification. Centroid computation includes an $O(|C||V|)$ averaging step. Computing similarity as a simple inner product is $O(|C|m_t)$ because we need only consider non-zero components.

The Rocchio classifier can even misclassify documents in the training set because of its strict assumptions about how documents in a class are distributed (Exercise 14.3). In the next section, we will introduce a vector space classifier, kNN, that deals better with classes that have non-spherical, disconnected or other irregular shapes.

Table 14.1 gives the time complexity of Rocchio classification. Computing either the sum or the inner product of a sparse vector with a non-sparse vector (such as a centroid) takes time proportional to the number of non-zero entries in the sparse vector. Adding one document to a centroid is therefore $O(L_d)$ and computing one inner product is $O(L_t)$. Overall, training time is linear in the size of the collection. Rocchio classification and Naive Bayes thus have the same time complexity.



► **Figure 14.5** kNN classification. Decision boundaries in 1NN are based on a Voronoi tessellation: they consist of those lines of the tessellation that separate points from different classes – drawn as double lines here, separating the three classes: X, circle and diamond.

14.2 *k* nearest neighbor

k NEAREST NEIGHBOR CLASSIFICATION

Unlike Rocchio, *k nearest neighbor* or *kNN classification* determines the decision boundary locally. For 1NN we assign each document to the class of its closest neighbor. Decision boundaries are concatenated segments of the *Voronoi tessellation* as shown in Figure 14.5. The Voronoi tessellation of a set of objects decomposes space into Voronoi cells, where each object's cell consists of all points that are closer to the object than to other objects. In our case, the objects are documents. The cell boundary of a Voronoi cell is a sequence of linear segments (or hyperplanar segments in higher dimensions).

VORONOI TESSELLATION

1NN is not very robust. The classification decision of each document relies on the class of a single document, which may be incorrectly labeled or atypical. For improved robustness, we classify based on a neighborhood of the $k > 1$ closest neighbors, assigning documents to the majority class of their k closest neighbors in its categorical form, with ties broken randomly. In the probabilistic version of the method, the probability of membership in the class is estimated as the proportion of the k nearest neighbors in the class. Figure 14.5 gives an example of kNN classification for $k = 3$. Class mem-

Training
Preprocess documents in training set
Determine k (e.g., optimal k on held-out data)
Testing: Classify test document $\vec{d}$
Compute the similarity of all training documents with $\vec{d}$
Identify the set S_k of the k most similar training documents
For each class c
  Compute $N(S_k, c)$, the number of members of S_k in c
  Estimate $\hat{P}(c|\vec{d})$ as $N(S_k, c)/|S_k|$

► **Figure 14.6** kNN training and testing.

D	training set
L_d	average length of a training document
$O(1)$	training time (without preprocessing)
$O(\|D\|L_d)$	training time (with preprocessing)
L_t	average length of a test document
m_t	average vocabulary size of a test document
$O(L_t + \|D\|L_d) = O(\|D\|L_d)$	test time (docs were not preprocessed)
$O(L_t + \|D\|m_t) = O(\|D\|m_t)$	test time (docs were preprocessed)

► **Table 14.2** Training and test time for kNN classification.

bership probabilities of the star are estimated as $\hat{P}(\text{circle class}|\text{star}) = 1/3$, $\hat{P}(\text{X class}|\text{star}) = 2/3$, and $\hat{P}(\text{diamond class}|\text{star}) = 0$.

To determine the parameter k in kNN we can select the k that maximizes effectiveness on a held-out subset of the training set. But often k is chosen based on experience or knowledge about the classification problem at hand. In that case, k will usually be odd to make ties less likely. $k = 3$ and $k = 5$ are common choices.

Figure 14.6 summarizes the kNN algorithm and Table 14.2 gives its complexity. The time complexity of kNN has properties that are quite different from most other classification algorithms. First, test time is independent of the number of classes J. kNN has therefore a potential advantage for problems with large J. Secondly, test time is $O(|D|)$ – linear in the size of the training set as we need to compute the similarity of each training document with the test document. Training a kNN classifier simply consists of determining k and document preprocessing. In fact, if we preselect a value for k and do not preprocess, then kNN requires no training at all. In practice, we have to perform preprocessing steps like tokenization at some point. It makes more sense to preprocess once as part of the training phase rather than repeatedly every time we classify a document.

Testing in kNN classification consists of estimating class probabilities based on the k-neighborhood S_k of the test document. Time complexity is independent of k.[1] We rely solely on "memorizing" all examples in the training set and comparing the test document to them. For this reason, kNN is also called *memory-based learning* or *instance-based learning*. It is usually desirable to have as much training data as possible in machine learning. But in kNN large training sets come with a severe efficiency penalty.

MEMORY-BASED LEARNING
INSTANCE-BASED LEARNING

Can kNN be made more efficient? There are fast kNN algorithms for small k (Exercise 14.8). And there are approximations for large k that give error bounds for specific efficiency gains (see Section 14.4). These approximations have not been extensively tested for text classification applications, so it is not clear whether they can achieve much better efficiency than $O(|D|m_t)$ without a significant loss of accuracy.

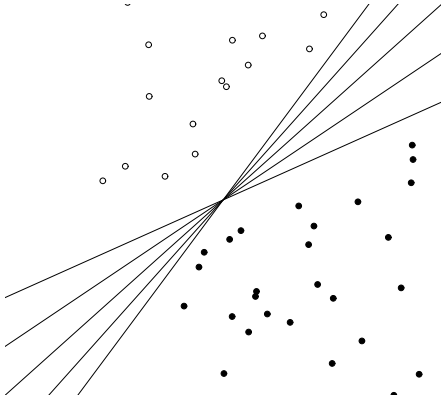
The attentive reader may have noticed the similarity between the problem of finding nearest neighbors of a test document and adhoc retrieval, where we search for the documents with the highest similarity to the query (Section 7.1.2, page 100). In fact, the two problems are both k nearest neighbor problems and only differ in the sparseness of the test document in kNN (10s or 100s of non-zero entries) and the query in adhoc retrieval (usually fewer than 10 non-zero entries). We introduced the inverted index for efficient adhoc retrieval in Chapter 1 (page 4). Is the inverted index also the solution for efficient kNN?

Searching for the best matches in the inverted index is restricted to those documents that have at least one term in common with the query. Thus, the inverted index will be efficient if the test document has no term overlap with a large number of training documents. How often this is the case depends on the classification problem. If documents are long and no stop list is used, then less time will be saved. But with short documents and a large stop list, an inverted index may well cut the average test time by a factor of 10 or more.

It is easy to construct synthetic data sets with sublinear search times. For example, search time is constant in the size of the collection if, as we generate new documents in the collection, terms are used for a period of time and then replaced by the next generation of terms. In this case, the lengths of inverted lists and hence search times are bounded by a constant. However, this type of word distribution is very improbable. In practice, kNN search is probably always $O(|D|)$ when implemented with an inverted index.

As we will see in the next chapter, kNN effectiveness is often close to the results for the best-performing classifiers in text classification (Table 15.2, page 244). However, the theoretical bounds for kNN are not as favorable for problems with an inherent error rate. The error of 1NN is asymptotically

1. Strictly speaking, classification requires an $O(k \log k)$ sorting step. This component is so small compared to O(|D|) that we ignore it here.



► **Figure 14.7** There is an infinite number of hyperplanes that separate two linearly separable classes.

bounded by twice the Bayes' error, that is, if the optimal classifier has an error rate of x, then 1NN has an asymptotic error rate of less than $2x$. Intuitively, this is the case because noise affects two components of kNN: the test document and the closest training document. The two sources of noise are additive, so the overall error of 1NN is twice the optimal error rate. For problems with Bayes error 0, the error rate of 1NN will approach 0 as the size of the training set increases.

14.3 Linear vs. nonlinear classifiers and the bias-variance tradeoff

The kNN classifier is nonlinear – it can model an arbitrarily complex decision boundary provided the training set is large enough. There is a tradeoff between the expressive power of such a complex nonlinear model and its accuracy on new data. In contrast, Rocchio and Naive Bayes are linear classifiers as we will show below. They decide class membership based on a linear combination of features. Their expressive power is low, but they are less sensitive to noise in small training sets if the linearity assumption is correct.

BIAS-VARIANCE TRADEOFF

Rocchio, Naive Bayes and kNN exemplify the *bias-variance tradeoff*. We can think of bias as a kind of domain knowledge that we build into a classifier. If we know that the decision boundary is linear, then the linear bias will improve generalization. If the decision boundary is not linear and we incorrectly bias the classifier, then classification accuracy will be low.

Rocchio and Naive Bayes are just two members of a large group of linear classification methods. In this section, we only consider binary classifiers,

w_i	x_i	w_i	x_i
0.70	prime	-0.71	dlrs
0.67	rate	-0.35	world
0.63	interest	-0.33	sees
0.60	rates	-0.25	year
0.46	discount	-0.24	group
0.43	bundesbank	-0.24	dlr

▶ **Table 14.3** A linear classifier. The variables (x_i) and parameters (w_i) of a linear classifier for the class *interest* (as in interest rate) in Reuters-21578. The threshold is $b = 0$. Words like dlr and world have negative parameter weights because they are indicators for the competing class *currency*.

which decide between two classes (e.g., *China* and *non-China*). Linear classifiers are the simplest binary classifiers. In two dimensions, a linear classifier is a line. Five example lines are shown in Figure 14.7.

The lines in Figure 14.7 have the functional form $w_1x_1 + w_2x_2 = b$. The classification rule of a linear classifier is to assign a document to class 1 if $w_1x_1 + w_2x_2 > b$ and to class 2 if $w_1x_1 + w_2x_2 \leq b$. Here, $(x_1, x_2)^T$ is the two-dimensional vector representation of the document and $(w_1, w_2)^T$ is the parameter vector defining (together with b) the decision boundary.

HYPERPLANE

A line corresponds to a *hyperplane* in higher dimensions. A line divides a plane in two, a plane divides 3-dimensional space in two, and hyperplanes divide higher dimensional spaces in two.

A hyperplane is given by:

$$\vec{w}^T\vec{x} = b \tag{14.2}$$

NORMAL VECTOR

where $\vec{w}$ is referred to as the *normal vector* and the criteria for assignment to classes 1 and 2 are $\vec{w}^T\vec{x} > b$ and $\vec{w}^T\vec{x} \leq b$, respectively. This definition of hyperplanes includes lines and planes. We call a hyperplane that is used for

DECISION HYPERPLANE

classification a *decision hyperplane*.

We now show that Rocchio and Naive Bayes are linear. To see this for Rocchio, observe that a document is on the decision boundary if its two inner products with the class centroids are equal:

$$\vec{\mu}(c_1) \cdot \vec{d} = \vec{\mu}(c_2) \cdot \vec{d} \tag{14.3}$$

$$\Leftrightarrow \quad (\vec{\mu}(c_1) - \vec{\mu}(c_2))^T \cdot \vec{d} = 0 \tag{14.4}$$

This corresponds to a linear classifier with normal vector $\vec{w} = \vec{\mu}(c_1) - \vec{\mu}(c_2)$ and $b = 0$.

We can derive the linearity of Naive Bayes from its decision rule, which

chooses the category c with the largest $\hat{P}(c|d)$ (Figure 13.3, page 197) where:

$$\hat{P}(c|d) = \hat{P}(c) \prod_{i \in S} \hat{P}(x_i|c)$$

and S is the set of positions in the document. Denoting the complement category as $\bar{c}$, we obtain for the log odds:

$$\log \frac{\hat{P}(c|d)}{\hat{P}(\bar{c}|d)} = \log \frac{\hat{P}(c)}{\hat{P}(\bar{c})} + \sum_{k \in S} \log \frac{\hat{P}(x_i|c)}{\hat{P}(x_i|\bar{c})} \tag{14.5}$$

We choose class c if the odds are greater than 1 or, equivalently, if the log odds are greater than 0. It is immediately obvious that Equation (14.5) is an instance of Equation (14.2) for $w_i = \log[\hat{P}(x_i|c)/\hat{P}(x_i|\bar{c})]$, $x_i =$ number of occurrences of w_i in d, and $b = -\log[\hat{P}(c)/\hat{P}(\bar{c})]$. So in log space, Naive Bayes is indeed a linear classifier.

Table 14.3 gives an example of a linear classifier for the category *interest* in Reuters-21578 (see Chapter 13, page 206). The document $\vec{x}_1$ "rate discount dlrs world" will be assigned to *interest* since $\vec{w}^T\vec{x}_1 = 0.67 \cdot 1 + 0.46 \cdot 1 + (-0.71) \cdot 1 + (-0.35) \cdot 1 = 0.05 > 0 = b$. For simplicity, we assume a simple binary vector representation in this example: 1 for occurring words, 0 for non-occurring words. Document $\vec{x}_2$ "prime dlrs" would be assigned to the other class (not in *interest*) since $\vec{w}^T\vec{x}_2 = -0.01 \leq b$.

LINEAR SEPARABILITY

Training a linear classifier amounts to identifying a separating hyperplane between the two classes. This is only possible if the two classes are *linearly separable*, that is, at least one separating hyperplane exists. In fact, if linear separability holds, then there is an infinite number of linear separators (Exercise 14.10). Five linear separators are shown in the example in Figure 14.7.

There are simple and efficient algorithms for finding a separating hyperplane, such as the perceptron algorithm (see Section 14.4), but they do not necessarily produce good classifiers. The challenge is to choose a hyperplane that has good classification effectiveness on the test set according to F_1, misclassification rate or a similar criterion. There are two broad classes of methods for making this selection:

- Methods that rely on the global distribution of documents. These methods are robust since atypical and noisy data are less likely to affect them. Naive Bayes, Rocchio and linear regression (see Section 14.4) are methods of this type.
- Methods that give more weight to documents close to the decision boundary. These methods are more accurate if enough training data of sufficient quality is available. Support vector machines (with linear kernel, see Chapter 15) and logistic regression (see Section 14.4) belong to this category.

It is perhaps surprising that so many of the best-known text classification algorithms are linear: perceptron, Naive Bayes, Rocchio, linear regression, logistic regression, and linear SVM. The simple neural network, a neural network without hidden layers, is also linear (see Section 14.4). Some of these methods, in particular regularized logistic regression and regularized linear regression, are among the best-performing text classifiers. Why is it that more complex models, models that can learn nonlinear decision boundaries, are not clearly superior to linear models?

BIAS-VARIANCE TRADEOFF
BIAS

The reason is the *bias-variance tradeoff*. Linear classifiers are high-bias and low-variance, nonlinear classifiers are low-bias and high-variance. *Bias* is the restrictiveness of the model of a classifier. Linear classifiers can only model one type of decision boundary, a hyperplane. The high-variance classifier kNN can – given enough training data – model an arbitrarily complex decision boundary.

VARIANCE

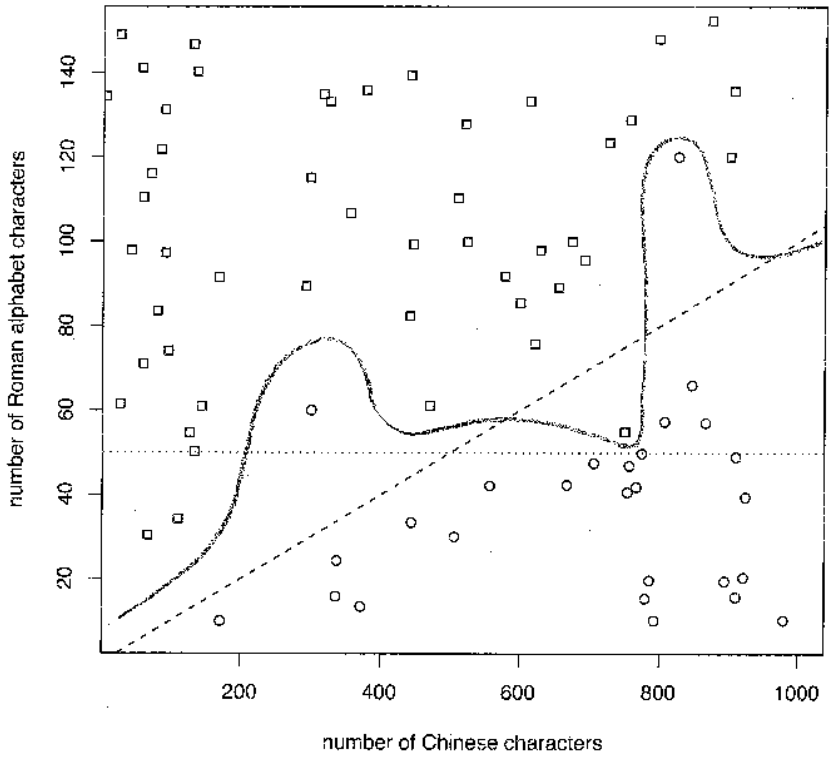
Variance refers to the variability of classification behavior when we train on different randomly generated training sets. NB's decision boundary is always a hyperplane and hence less variable. In contrast, kNN's more complex boundary will vary much more depending on the sample we are training on – especially if the distribution is noisy and the sample is small.

CAPACITY

But variance also corresponds to *capacity* – the number of independent parameters available to fit the data or, loosely speaking, the memory available to store information about the training set. Each kNN neighborhood S_k makes an independent classification decision. The parameter in this case is the estimate $\hat{P}(c|S_k)$ from Figure 14.5. Thus, kNN's capacity is unlimited in principle if enough training data is available. The capacity of NB is fixed and independent of the training set. It can only model a hyperplane. If the decision boundary is not linear and NB therefore incorrectly biased, then the greater capacity of kNN will carry the day (assuming a sufficiently large training set). Asymptotically, it will perfectly model the nonlinear decision boundary, something NB cannot do.

Figure 14.8 provides a (somewhat contrived) illustration of the tradeoff. Some Chinese text contains English words written in the Roman alphabet like CPU, ONLINE, and GPS. Consider the task of distinguishing Chinese-only web pages from mixed Chinese-English web pages. A search engine might offer Chinese users without knowledge of English (but who understand loanwords like CPU) the option of filtering out mixed pages. We use two features for this classification task: number of Roman alphabet characters and number of Chinese characters on the web page. In Figure 14.8, we see three classifiers:

- A high-bias low-variance classifier that only uses one feature, in this case the number of Roman alphabet characters. This classifier does not fit the distribution of the data and will not generalize well to new data, in par-



► **Figure 14.8** The bias-variance tradeoff. In this hypothetical web page classification scenario, we want to discriminate between Chinese-only web pages (circles) and mixed Chinese-English web pages (squares). The linear classifier (linear combination of two features, dashed line) offers the best tradeoff, with better generalization than the high-bias low-variance classifier (restricted to using one feature, dotted line) and the low-bias high-variance classifier (arbitrary decision boundary, solid line).

ticular, to long documents.

- A medium-bias medium-variance linear classifier using a linear combination of the two features. This classifier makes some mistakes on the training set, but captures the underlying generalization best: Pages with roughly more than 10% Roman characters are mixed pages.
- A low-bias high-variance classifier that selects a decision boundary that separates the two classes perfectly. Even though its accuracy on the training data is optimal, it is influenced by outliers and will not generalize well to new data.

OVERFITTING

The low-bias high-variance classifier *overfits* the training data. The goal in classification is to fit the training data to the extent that we capture true properties of the underlying distribution. In overfitting, the classifier also learns from the noise present in the training set. In Figure 14.8, this noise consists of the three outliers that do not fit into the general pattern of their respective classes. Overfitting decreases accuracy on new data and is a typical problem for high-variance classifiers.

The bias-variance tradeoff provides insight into why linear classifiers often perform best on text classification problems. Typical classes in text classification are complex and seem unlikely to be modelled well linearly. However, our intuitions are misleading for the high-dimensional spaces that we typically encounter in text applications. With increased dimensionality, the likelihood of linear separability increases rapidly (Exercise 14.12). Thus, linear models in high-dimensional spaces are quite powerful despite their linearity. More powerful nonlinear classifiers can model decision boundaries more complex than a hyperplane, but they are more sensitive to noise in the training data. Nonlinear classifiers may perform better if the training set is large, but by no means in all cases.

14.3.1 More than two classes

Linear classifiers are binary, that is, they distinguish two classes. What do we do if there are $J > 2$ different classes? This depends on whether the classes are mutually exclusive or not.

ANY-OF CLASSIFICATION

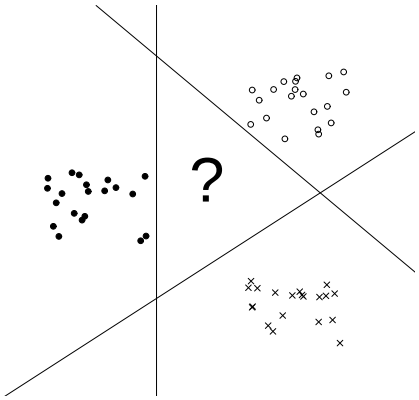
Classification for classes that are not mutually exclusive is called *any-of*, *multilabel*, or *multivalue classification*. In this case, a document can belong to several classes simultaneously or to a single class or to none of the classes. A decision on one class leaves all options open for the others. It is sometimes said that the classes are *independent* of each other, but this is misleading since the classes are rarely statistically independent. In terms of the formal definition of the classification problem in Equation (13.1) (page 191), we define J different classification functions γ_j for any-of classification, each returning either yes or no for its class: $\gamma_j(\vec{d}) \in \{0, 1\}$.

Solving an any-of classification task with linear classifiers is straightforward:

- Build a classifier for each class, where the training set consists of the set of documents in the class (positive labels) and its complement (negative labels).
- Given the test document, apply each classifier separately. The decision of one classifier has no influence on the decisions of the other classifiers.

ONE-OF CLASSIFICATION

The second type of classification with more than two classes is *one-of clas-*



► **Figure 14.9** Linear classification for one-of classification with $J > 2$. A combination method has to be applied in this case since J hyperplanes do not divide the space into J disjoint regions.

sification. Here, the classes are mutually exclusive. Each document must belong to exactly one of the classes. One-of classification is also called *multinomial*, *polytomous*[2], or *multiclass classification*. Formally, there is a single classification function γ in one-of classification whose range is C, i.e., $\gamma(\vec{d}) \in \{c_1, \ldots, c_J\}$. Naive Bayes, Rocchio and kNN are one-of classifiers.

True one-of problems are less common in text classification than any-of problems. In topic classification (with classes like *UK*, *China*, *poultry*, or *coffee*), a document can obviously be relevant to any of the topics – as when the prime minister of the UK visits China to talk about the coffee and poultry trade.

Nevertheless, we will often make a one-of assumption, as we did in Figure 14.1, even if classes are not really mutually exclusive. For language identification the one-of assumption is also approximately true as most text is written in only one language. If the one-of assumption holds to an approximation, then it simplifies the classification problem without a large decrease in classification accuracy.

Linear classifiers must be used with a combination method for one-of classification as illustrated in Figure 14.9. J hyperplanes do not divide $\mathbb{R}^{|V|}$ into the J different regions we would need for one-of classification.

CATEGORY RANKING

We need a combination method that performs some kind of *category ranking* and then selects the top-ranked class. Geometrically, the ranking can be with respect to the distances from the linear separators (one for each class).

2. A synonym of polytomous is polychotomous.

assigned class / true class	*money-fx*	*trade*	*interest*	*wheat*	*corn*	*grain*
money-fx	95	0	10	0	0	0
trade	1	1	90	0	1	0
interest	13	0	0	0	0	0
wheat	0	0	1	34	3	7
corn	1	0	2	13	26	5
grain	0	0	2	14	5	10

► **Table 14.4** A confusion matrix for Reuters-21578. Example: 14 documents from *grain* were incorrectly assigned to *wheat*. Adapted from Picca et al. (2006).

Documents close to a class's separator are more likely to be misclassified, so the greater the distance from the separator, the more plausible it is that a positive classification decision is correct. Alternatively, a direct measure of confidence can be used to rank classes, e.g., probability of class membership. The algorithm for one-of classification with linear classifiers can be summarized as follows:

- Build a classifier for each class, where the training set consists of the set of documents in the class (positive labels) and its complement (negative labels).
- Given the test document, apply each classifier separately.
- Assign the document to the class with
  - the maximum score,
  - the maximum confidence value,
  - or the maximum probability.

CONFUSION MATRIX

An important tool for analyzing the performance of a one-of classification system is the *confusion matrix*. The confusion matrix shows for each pair of classes $< c_1, c_2 >$, how many documents from c_1 were incorrectly assigned to c_2 and vice versa. In Table 14.4, the classifiers manage to distinguish the three financial classes *money-fx*, *trade*, and *interest* from the three agricultural classes *wheat*, *corn*, and *grain*, but make many errors within these two groups. The confusion matrix can help pinpoint opportunities for improving the accuracy of the system. For example, to address the largest error in Table 14.4, one could attempt to introduce features that distinguish *wheat* documents from *grain* documents.

14.4 References and further reading

As discussed in Chapter 9, Rocchio relevance feedback is due to Rocchio (Rocchio 1971). It was widely used as a classification method in the TREC competition in the 1990s (Buckley et al. 1994a, Voorhees and Harman 2005).

ROUTING

Initially, Rocchio classification was a form of *routing*. Routing merely ranks documents according to relevance to a class without assigning them. Early

FILTERING

work on *filtering*, a true classification approach that makes an assignment decision on each document, was published by Ittner et al. (1995) and Schapire et al. (1998).

A more detailed treatment of kNN can be found in (Hastie et al. 2001), including methods for tuning the parameter k. An example of an approximate fast kNN algorithm is locality-based hashing (Andoni et al. 2007). Kleinberg (1997) presents an approximate $O((M \log^2 M)(M + \log N))$ kNN algorithm (where M is the dimensionality of the space and N the number of data points), but at the cost of exponential storage requirements: $O((N \log M)^{2M})$. Yang (1994) uses an inverted index to speed up kNN classification. The optimality result for 1NN (twice the Bayes error asymptotically) is due to Cover and Hart (1967).

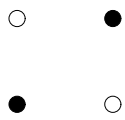
The concept of bias-variance tradeoff was introduced by Geman et al. (1992). See Schütze et al. (1995) and Lewis et al. (1996) for more discussion of linear text classifiers and Hastie et al. (2001) on linear classifiers in general. Readers interested in the algorithms mentioned, but not described in this chapter may wish to consult Bishop (2006) for neural networks, Hastie et al. (2001) for linear and logistic regression, and Minsky and Papert (1988) for the perceptron algorithm. Anagnostopoulos et al. (2006) show how to do linear classification with an inverted index.

We have only presented the simplest method for combining binary classifiers into a multiclass classifier. Another important method is error-correcting codes, where a vector of decisions of different binary classifiers is constructed for each document. A test document's vector is then "corrected" based on the distribution of decision vectors in the training set, a procedure that incorporates information from all binary classifiers and their correlations into the final classification decision (Dietterich and Bakiri 1995). Allwein et al. (2000) propose a general framework for combining binary classifiers.

14.5 Exercises

Exercise 14.1

Download Reuters-21578 and train and test Rocchio and kNN classifiers for the classes *acquisitions*, *corn*, *crude*, *earn*, *grain*, *interest*, *money-fx*, *ship*, *trade*, and *wheat*. Use the ModApte split. You may want to use one of a number of software packages that im-



► **Figure 14.10** A simple non-separable set of points.

plement Rocchio classification and kNN classification, for example, the Bow toolkit (McCallum 1996).

Exercise 14.2

Create a training set of 300 documents, 100 each from three different languages (e.g., English, French, Spanish). Create a test set by the same procedure and then add 100 documents from a fourth language. (i) Build a one-of classifier that identifies the language of a document and evaluate it on the test set. (ii) Build an any-of classifier that identifies the language of a document and evaluate it on the test set.

Exercise 14.3

Show that Rocchio classification can assign a label to a document that is different from its training set label.

Exercise 14.4

Show that the decision boundaries in Rocchio classification are, as in kNN, a subset of the Voronoi tessellation.

Exercise 14.5

kNN decision boundaries are concatenations of linear segments for $k = 1$. Is this also true for $k > 1$?

Exercise 14.6

Why are ties unlikely for odd k in kNN?

Exercise 14.7

Explain why kNN handles polymorphic classes better than Rocchio.

Exercise 14.8

Design an algorithm that performs an efficient kNN search in (i) 1 dimension, (ii) 2 dimensions.

Exercise 14.9

Can one design an exact efficient algorithm for kNN for very large k along the ideas you used to solve the last exercise?

Exercise 14.10

Prove that the number of linear separators of two classes is either infinite or zero.

Exercise 14.11

We can easily construct non-separable data sets in high dimensions by embedding a non-separable set like the one shown in Figure 14.10. Explain why such an embedded configuration is likely to become separable in high dimensions after adding noise.

Exercise 14.12

Make plausible that non-separable data sets become unlikely with increasing dimensionality by showing that the percentage of non-separable assignments of the vertices of a hypercube to two classes decreases with dimensionality for $d > 1$. For example, for $d = 1$ the number of non-separable assignments is 0, for $d = 2$, it is 2/16. One of the two non-separable cases for $d = 2$ is shown in Figure 14.10, the other is its mirror image. Solve the exercise either analytically or by simulation.

Exercise 14.13

Although we point out the similarities of Naive Bayes with linear vector space classifiers, it does not really make sense to represent count vectors (the document representations in NB) in a continuous vector space. There is however a formalization of NB that is analogous to Rocchio. Show that NB assigns a document to the class (represented as a parameter vector) whose KL-divergence to the document (represented as a count vector, normalized to be a probability distribution) is smallest.

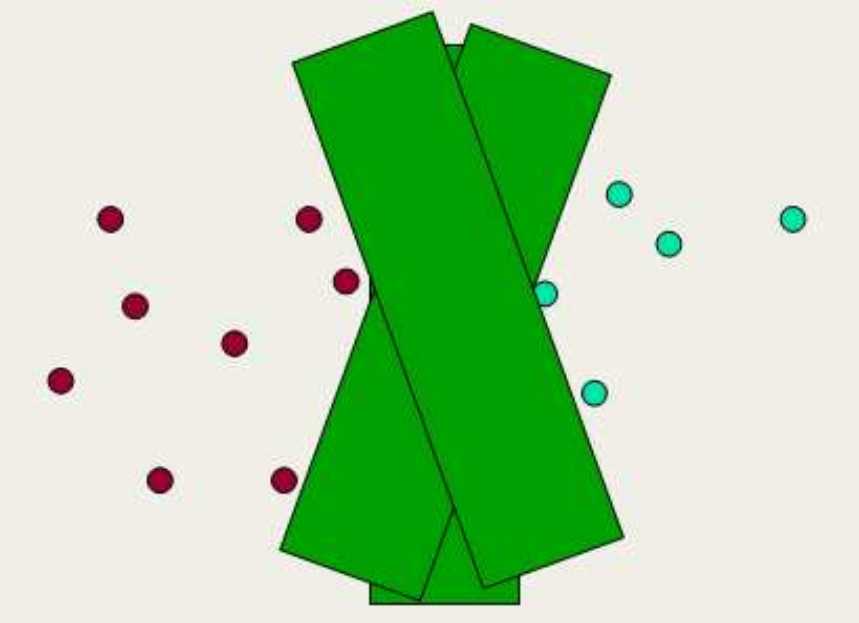
15 *Support vector machines and kernel functions*

Improving classifier performance has been an area of intensive machine-learning research for the last two decades, and this work has led to a new generation of state-of-the-art classifiers, such as support vector machines, boosted decision trees, regularized logistic regression, neural networks, and random forests. Many of these methods, including support vector machines (SVMs), the main topic of this chapter, have been applied with success to information retrieval problems, particularly text classification. We will initially motivate and develop SVMs for the case of two-class data sets that are separable by a linear classifier (Section 15.1), and then extend the model to non-separable data (Section 15.2) and nonlinear models (Section 15.3). The chapter then concludes with more general discussion of experimental results for text classification (Section 15.4) and system design choices and text-specific features to be exploited in all text categorization work (Section 15.5). Support vector machines, otherwise referred to as large-margin classifiers, are not necessarily better than other methods in the above group (except perhaps in low data situations), but they perform at the state-of-the-art level and have much current theoretical and empirical appeal. No one ever got fired for using an SVM.

15.1 Support vector machines: The linearly separable case

For some data sets, such as the one in Figure 14.7 (page 222), there are lots of possible linear separators. Intuitively, the gray/green one seems better than the black/red one because it draws the decision boundary in the middle of the void between the data. While some learning methods such as the perceptron algorithm just find any linear separator, others search for the best linear separator according to some criterion, and the SVM in particular defines this as looking for a decision surface that is maximally far away from any data points. This distance from the decision surface to the closest data point determines the *margin* of the classifier.

MARGIN



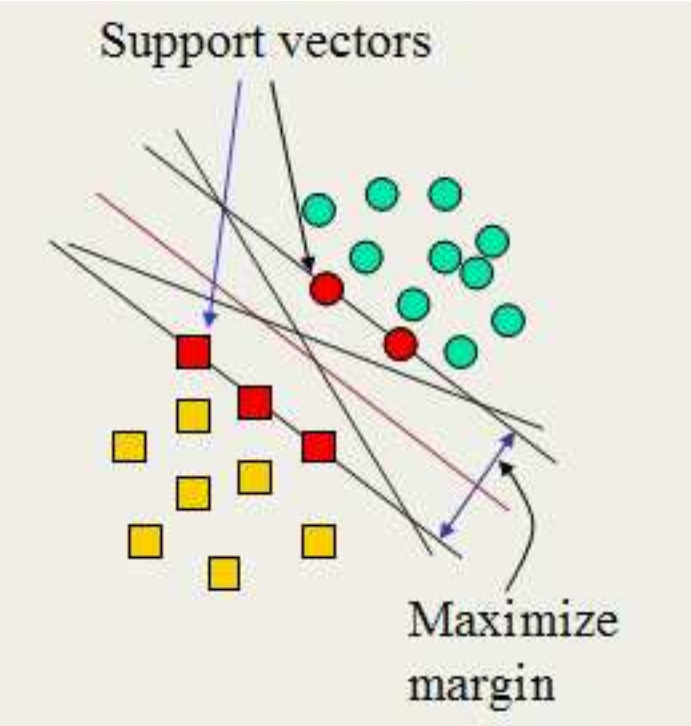
► **Figure 15.1** The intuition of large-margin classifiers. Insisting on a large margin reduces the capacity of the model: the range of angles at which the fat decision surface can be placed is smaller than for a decision hyperplane.

This seems like a good thing to do because points near the decision surface represent very uncertain classification decisions: there is almost a 50% chance of the classifier going either way. A classifier with a large margin makes no very uncertain classification decisions. Another intuition motivating SVMs is shown in Figure 15.1. By construction, an SVM classifier insists on a large margin around the decision boundary. Compared to a decision hyperplane, if you have to place a fat separator between classes, you have fewer choices of where it can be put. As a result of this, the capacity of the model has been decreased, and hence we expect that its ability to correctly generalize to test data is increased (cf. the discussion of the bias-variance tradeoff in Chapter 14, page 222).

SVMs are inherently two-class classifiers. To do multiclass classification, one has to use one of the methods discussed in Section 14.3.1 (page 227).

An SVM is constructed to maximize the margin around the separating hyperplane. This necessarily means that the decision function for an SVM is fully specified by a (usually small) subset of the data which defines the position of the separator. These points are referred to as the *support vectors*.

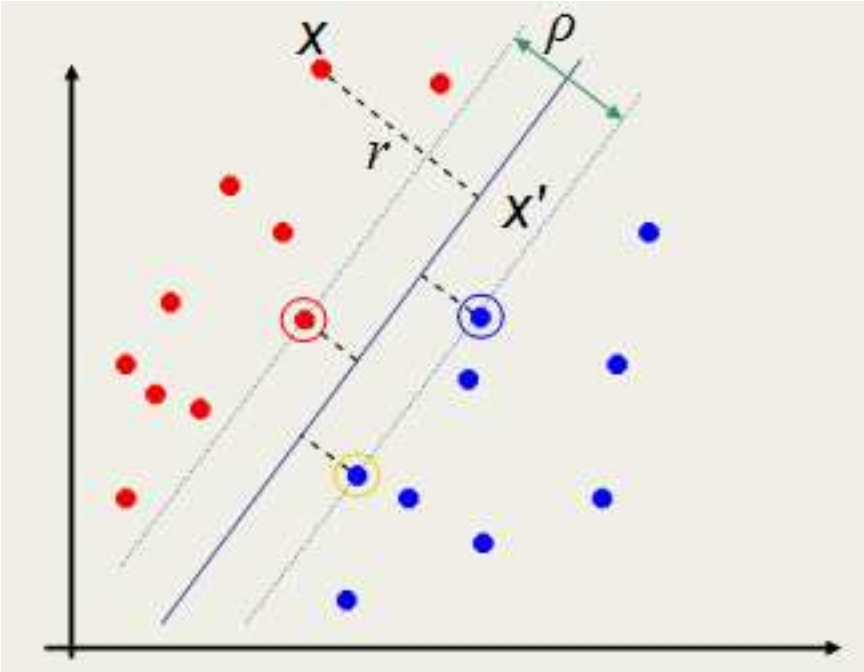
SUPPORT VECTOR



► **Figure 15.2** Support vectors are the points right up against the margin of the classifier.

Figure 15.2 shows the support vectors for a sample problem. Note that the other data points play no part in determining the decision surface that is chosen.

Let us try to formalize this notion with algebra. A decision hyperplane (page 223) can be defined by a decision hyperplane normal vector $\vec{w}$ which is perpendicular to the hyperplane. Because of this perpendicularity, all points $\vec{x}$ on the hyperplane satisfy $\vec{w}^{\mathrm{T}}\vec{x} = 0$. To choose among all the hyperplanes that are perpendicular to the normal vector, we also specify an intercept term b. Now suppose that we have a set of training data points $\{\vec{x}_i\}$ with corresponding classes $\{y_i\}$. For SVMs, the two data classes are always named $+1$ and -1 (rather than 1 and 0), and the intercept term is always explicitly represented as b rather than being folded into the weight vector by adding an extra always-on feature. It just turns out that the math works out much more cleanly if you do things this way, as we will see almost immediately in the



► **Figure 15.3** The geometric margin of a linear classifier.

definition of functional margin. Our linear classifier is then:

$$f(\vec{x}_i) = \text{sign}(\vec{w}^{\mathrm{T}}\vec{x}_i + b) \tag{15.1}$$

FUNCTIONAL MARGIN

We are confident in the classification of a point if it is far away from the decision boundary. For a given data set and decision hyperplane, we say that the *functional margin* of the i^{th} example $\vec{x}_i$ is $y_i(\vec{w}^{\mathrm{T}}\vec{x}_i + b)$. The functional margin of a dataset is then twice the minimal functional margin of any point in the data set (the factor of 2 comes from measuring across the whole width of the margin, as in Figure 15.2). However, there is a problem with this definition: we can always make the functional margin as big as we wish by simply scaling up $\vec{w}$ and b. For example, if we replace $\vec{w}$ by $5\vec{w}$ and b by $5b$ then the functional margin $y_i(5\vec{w}^{\mathrm{T}}\vec{x}_i + 5b)$ is five times as large. This suggests that we need to place some constraint on the size of the $\vec{w}$ vector. To get a sense of how to do that, let's look at the actual geometry.

What is the Euclidean distance from a point $\vec{x}$ to the decision boundary? Look at Figure 15.3. Let us call the distance we are looking for r. We know that the shortest distance between a point and a hyperplane is perpendicu-

lar to the plane, and hence, parallel to $\vec{w}$. A unit vector in this direction is $\vec{w}/\|\vec{w}\|$. Then, the dotted line in the diagram is a translation of the vector $r\vec{w}/\|\vec{w}\|$. Let us label the point on the hyperplane closest to $\vec{x}$ as $\vec{x}'$. Then we know from the above discussion that:

$$\vec{x}' = \vec{x} - yr\frac{\vec{w}}{\|\vec{w}\|} \tag{15.2}$$

where multiplying by y again just changes the sign for the two cases where $\vec{x}$ is on either side of the decision surface. Moreover, $\vec{x}'$ lies on the decision boundary and so satisfies that $\vec{w}\vec{x}' + b = 0$. Hence:

$$\vec{w}^{\mathrm{T}}(\vec{x} - yr\frac{\vec{w}}{\|\vec{w}\|}) + b = 0$$

Solving for r gives:[1]

$$r = y\frac{\vec{w}^{\mathrm{T}}\vec{x} + b}{\|\vec{w}\|} \tag{15.3}$$

GEOMETRIC MARGIN

Again, the points closest to the separating hyperplane are support vectors. The *geometric margin* of the classifier is the maximum width of the band that can be drawn separating the support vectors of the two classes. That is, it is twice the maximal r defined above, or the maximal width of one of the fat separators shown in Figure 15.1. The geometric margin is clearly invariant to scaling of parameters: if we replace $\vec{w}$ by $5\vec{w}$ and b by $5b$, then the geometric margin is the same, because it is scaled by the length of $\vec{w}$. This means that we can impose any scaling constraint we wish on $\vec{w}$ without affecting anything. Among other choices, requiring $\|\vec{w}\| = 1$ would make the geometric margin the same as the functional margin.

Since we can scale the functional margin as we please, let us require that the functional margin of all data points is at least 1 and that this bound is tight. Then for all items in the data:

$$y_i(\vec{w}^{\mathrm{T}}\vec{x}_i + b) \geq 1$$

and for the support vectors the inequality is an equality. Since each example's distance from the hyperplane is $r_i = y_i(\vec{w}^{\mathrm{T}}\vec{x}_i + b)/\|\vec{w}\|$, the geometric margin is $\rho = 2/\|\vec{w}\|$. Our desire is still to maximize this geometric margin. That is, we want to find $\vec{w}$ and b such that:

- $\rho = 2/\|\vec{w}\|$ is maximized
- For all $\{(\vec{x}_i, y_i)\}$, $y_i(\vec{w}^{\mathrm{T}}\vec{x}_i + b) \geq 1$

1. Recall that $\|\vec{w}\| = \sqrt{\vec{w}^{\mathrm{T}}\vec{w}}$.

Now noting that maximizing $2/\|\vec{w}\|$ is the same as minimizing $\|\vec{w}\|/2$, we have the standard final formulation as a minimization problem:

(15.4) Find $\vec{w}$ and b such that:

- $\frac{1}{2}\vec{w}^{\mathrm{T}}\vec{w}$ is minimized
- and for all $\{(\vec{x}_i, y_i)\}$, $y_i(\vec{w}^{\mathrm{T}}\vec{x}_i + b) \geq 1$

QUADRATIC PROGRAMMING

This is now optimizing a quadratic function subject to linear constraints. *Quadratic optimization* problems are a standard, well-known class of mathematical optimization problem, and many algorithms exist for solving them. We could in principle build our SVM using standard quadratic programming (QP) libraries, though in practice there are more specialized and much faster libraries available especially for building SVMs. There has been much research in this area and many fast but intricate algorithms exist for this problem. However, we will not present the details here. It is enough to understand how the problem is set up as a QP problem. Thereafter, there are only about 20 people in the world who don't use one of the standard SVM software packages to build models.

Nevertheless, to understand the mechanism for doing nonlinear classification with SVMs, we need to present a fraction more of how SVMs are solved although we will omit the details. The solution involves constructing a dual problem where a Lagrange multiplier α_i is associated with every constraint in the primary problem:

(15.5) Find $\alpha_1, \ldots \alpha_N$ such that $\sum \alpha_i - \frac{1}{2}\sum_i \sum_j \alpha_i \alpha_j y_i y_j \vec{x}_i^{\mathrm{T}} \vec{x}_j$ is maximized, and

- $\sum_i \alpha_i y_i = 0$
- $\alpha_i \geq 0$ for all $1 \leq i \leq N$

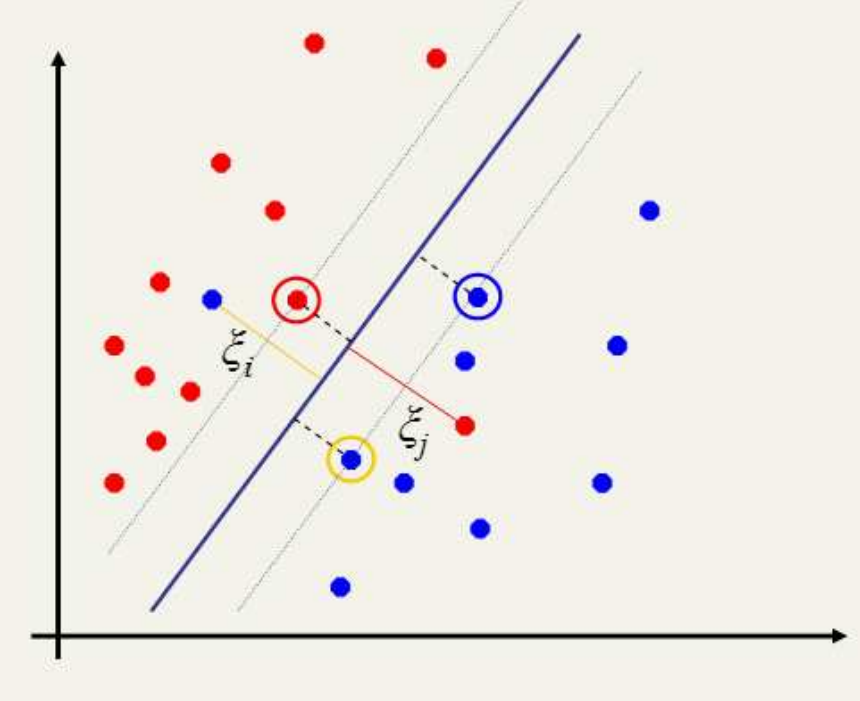
The solution is then of the form:

(15.6) $\vec{w} = \sum \alpha_i y_i \vec{x}_i$
$b = y_k - \vec{w}^{\mathrm{T}}\vec{x}_k$ for any $\vec{x}_k$ such that $\alpha_k \neq 0$

In the solution, most of the α_i are zero. Each non-zero α_i indicates that the corresponding $\vec{x}_i$ is a support vector. The classification function is then:

$$f(\vec{x}) = \text{sign}(\sum \alpha_i y_i \vec{x}_i^{\mathrm{T}} \vec{x} + b)$$

Notice that both the term to be maximized in the dual problem and the classifying function involve an *inner product* between pairs of points ($\vec{x}$ and $\vec{x}_i$ or $\vec{x}_i$ and $\vec{x}_j$), and that is the only way the data is used – we will return to the significance of this later.



► **Figure 15.4** Large margin classification with slack variables.

15.2 Soft margin classification

For the very high dimensional problems common in text classification, sometimes the data is linearly separable. But in the general case it is not, and even if it is, we might prefer a solution that better separates the bulk of the data while ignoring a couple of weird outlier points.

SLACK VARIABLES

If the training set is not linearly separable, the standard approach is to introduce *slack variables* ξ_i which allow misclassification of difficult or noisy examples. In this model, the fat decision margin is allowed to make a few mistakes, and we pay a cost for each misclassified example which depends on how far away from the decision surface it is. See Figure 15.4.

The formulation of the optimization problem with slack variables is then:

(15.7) Find $\vec{w}$, b, and $\xi_i \geq 0$ such that:

- $\frac{1}{2}\vec{w}^{\mathrm{T}}\vec{w} + C\sum_i \xi_i$ is minimized
- and for all $\{(\vec{x}_i, y_i)\}$, $y_i(\vec{w}^{\mathrm{T}}\vec{x}_i + b) \geq 1 - \xi_i$

The optimization problem is then trading off how fat it can make the margin versus how many points have to be moved around to allow this margin. The margin can be less than 1 for a point $\vec{x}_i$ by setting $\xi_i > 0$, but then one pays a penalty of $C\xi_i$ in the minimization for having done that. The parameter C is a *regularization* term, which provides a way to control overfitting: as C becomes large, it is unattractive to not respect the data at the cost of reducing the geometric margin, while when it is small, it is easy to account for some data points with the use of slack variables and to have the fat margin placed so it models the bulk of the data.

REGULARIZATION

The dual problem for soft margin classification becomes:

(15.8) Find $\alpha_1, \ldots \alpha_N$ such that $\sum \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \vec{x}_i^{\mathrm{T}} \vec{x}_j$ is maximized, and

- $\sum_i \alpha_i y_i = 0$
- $0 \leq \alpha_i \leq C$ for all $1 \leq i \leq N$

Note that neither the slack variables ξ_i nor Lagrange multipliers for them appear in the dual problem. All we are left with is the constant C bounding the possible size of the Lagrange multipliers for the support vector data points. Again, the $\vec{x}_i$ with non-zero α_i will be the support vectors, and typically they will be a small proportion of the data. The solution of the dual problem is of the form:

(15.9) $\vec{w} = \sum \alpha y_i \vec{x}_i$
$b = y_k(1 - \xi_k) - \vec{w}^{\mathrm{T}} \vec{x}_k$ for $k = \arg\max_k \alpha_k$

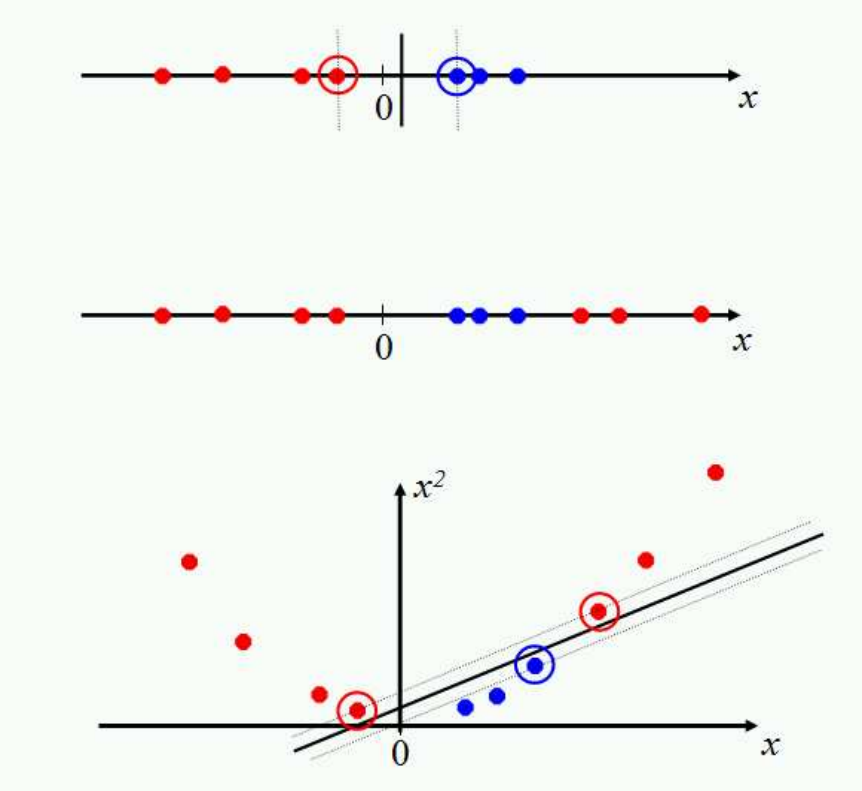
Again $\vec{w}$ is not needed explicitly for classification, which can be done simply in terms of the dot product of data points:

$$f(\vec{x}) = \sum_i \alpha_i y_i \vec{x}_i^{\mathrm{T}} \vec{x} + b$$

Given a new point $\vec{x}$ to classify, the classification function is computing the projection of the point onto the hyperplane normal. This will determine which class to assign to the point. If the point is within the margin of the classifier (or another confidence threshold t that we might have determined to avoid classification mistakes) then the classifier can return "don't know" rather than one of the two classes.

15.3 Nonlinear SVMs

With what we have presented so far, data sets that are linearly separable (perhaps with a few exceptions or some noise) are well-handled. But what are we going to do if the data set is just too hard – in the sense that it just doesn't allow classification by a linear classifier. Let us look at a one-dimensional case



► **Figure 15.5** Projecting nonlinearly separable data into a higher dimensional space can make it linearly separable.

for motivation. The top data set in Figure 15.5 is straightforwardly classified by a linear classifier but the middle data set is not. We instead need to be able to pick out an interval. One way to solve this problem is to map the data on to a higher dimensional space and then to use a linear classifier in the higher dimensional space. For example, the bottom part of the figure shows that a linear separator can easily classify the data if we use a quadratic function to map the data into two dimensions (a polar coordinates projection would be another possibility). The general idea is to map the original feature space to some higher-dimensional feature space where the training set is separable. Though, of course, we would want to do so in ways that preserve relevant notions of data point relatedness, so that the resultant classifier should still generalize well. Kernels can make a non-separable problem separable, and they can map data into a better representational space.

KERNEL TRICK

SVMs, and also a number of other linear classifiers, provide an easy and efficient way of doing this mapping to a higher dimensional space, which is referred to as "the *kernel trick*". It's not really a trick: it just exploits the math that we have seen. The SVM linear classifier relies on an inner product between data point vectors. Let $K(\vec{x}_i, \vec{x}_j) = \vec{x}_i^{\mathrm{T}} \vec{x}_j$. Then the classifier is

$$f(\vec{x}) = \sum_i \alpha_i y_i K(\vec{x}_i, \vec{x}) + b \tag{15.10}$$

KERNEL FUNCTION

Now consider if we decided to map every data point into a higher dimensional space via some transformation $\Phi: \vec{x} \mapsto \phi(\vec{x})$. Then the inner product becomes $\phi(\vec{x}_i)^{\mathrm{T}} \phi(\vec{x}_j)$. If it turned out that this inner product (which is just a real number) could be computed simply and efficiently in terms of the original data points, then we wouldn't have to actually map from $\vec{x} \mapsto \phi(\vec{x})$. Rather, we could simply compute the quantity $K(\vec{x}_i, \vec{x}_j) = \phi(\vec{x}_i)^{\mathrm{T}} \phi(\vec{x}_j)$, and then use the function's value in Equation (15.10). A *kernel function* K is such a function that corresponds to an inner product in some expanded feature space.

For example, for 2-dimensional vectors $\vec{x} = (x_1, x_2)$, let $K(\vec{x}_i, \vec{x}_j) = (1 + \vec{x}_i^{\mathrm{T}} \vec{x}_j)^2$. We wish to show that this is a kernel, i.e., that $K(\vec{x}_i, \vec{x}_j) = \phi(\vec{x}_i)^{\mathrm{T}} \phi(\vec{x}_j)$ for some ϕ. Consider $\phi(\vec{x}) = (1, x_1^2, \sqrt{2}x_1 x_2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2)$. Then:

$$\begin{aligned} K(\vec{x}_i, \vec{x}_j) = (1 + \vec{x}_i^{\mathrm{T}} \vec{x}_j)^2 &= 1 + x_{i1}^2 x_{j1}^2 + 2x_{i1}x_{j1}x_{i2}x_{j2} + x_{i2}^2 x_{j2}^2 + 2x_{i1}x_{j1} + 2x_{i2}x_{j2} \\ &= [1 \; x_{i1}^2 \; \sqrt{2}x_{i1}x_{i2} \; x_{i2}^2 \sqrt{2}x_{i1} \; \sqrt{2}x_{i2}]^{\mathrm{T}} [1 \; x_{j1}^2 \; \sqrt{2}x_{j1}x_{j2} \; x_{j2}^2 \sqrt{2}x_{j1} \; \sqrt{2}x_{j2}] \\ &= \phi(\vec{x}_i)^{\mathrm{T}} \phi(\vec{x}_j) \end{aligned}$$

KERNEL
MERCER KERNELS

What kinds of functions are valid *kernel functions* (sometimes more precisely referred to as *Mercer kernels*, because they satisfy Mercer's condition)? The function K must be continuous, symmetric, and have a positive definite gram matrix. Such a K means that there exists a mapping to a reproducing kernel Hilbert space (a Hilbert space is a vector space closed under dot products) such that the dot product there equals the value of K. If you know a fair amount of functional analysis, those last two sentences might have made sense; if you don't but would like to, you should consult the books on SVMs in the references; and if you're in neither of those two groups, you can content yourself with knowing that 90% of work with kernels uses one of two straightforward families of functions of two vectors, which do define valid kernels.

These two common families of kernels are polynomial kernels and radial basis functions. Polynomial kernels are of the form $K(\vec{x}, \vec{z}) = (1 + \vec{x}^{\mathrm{T}} \vec{z})^d$. The case of $d = 1$ is a linear kernel, which is what we had before we started talking about kernels (the constant 1 just changing the threshold). The case

of $d = 2$ gives a quadratic kernel, and is very commonly used. We illustrated the quadratic kernel above. The most common form of radial basis function is a Gaussian distribution, calculated as:

$$K(\vec{x}, \vec{z}) = e^{-\|\vec{x}-\vec{z}\|^2/(2\sigma^2)}$$

A radial basis function is equivalent to mapping the data into an infinite dimensional Hilbert space, and so we can't illustrate the radial basis function in the same way as we did for a quadratic kernel. Beyond these two families, there has been interesting work developing other kernels, some of which is promising for text applications. In particular, there has been investigation of string kernels.

The world of SVMs comes with its own language, which is rather different from the language otherwise used in machine learning. The terminology does have deep roots in mathematics, even though most people who use SVMs don't really understand those roots and just use the terminology because others do and it sounds cool. It's important not to be too awed by that terminology. Really, we are talking about some quite simple things. A polynomial kernel allows you to model feature conjunctions (up to the order of the polynomial). Simultaneously you also get the powers of the basic features – for most text applications, that probably isn't useful, but just comes along with the math and hopefully doesn't do harm. A radial basis function (RBF) allows one to have features that pick out circles (hyperspheres) – although the decision boundaries become much more complex as multiple such features interact. A string kernel lets you have features that are character subsequences of words. All of these are straightforward notions which have also been used in many other places under different names.

15.4 Experimental data

Experiments have shown SVMs to be a very effective text classifier. Dumais et al. (1998) compared a Rocchio variant, Naive Bayes, a more general Bayes Net classifier, Decision Trees, and SVM classifiers on the 10 largest classes in the Reuters-21578 Test Collection, which was introduced in Chapter 13 (page 206). Some of their results are shown in Table 15.1, with SVMs clearly performing best. This was one of several pieces of work that established the strong reputation of SVMs for text classification. Another comparison from around the same time was work by Joachims (1998). Some of his results are shown in Table 15.2. Joachims uses a large number of word features and reports notable gains from using higher order polynomial or RBF kernels. However, Dumais et al. (1998) used MI feature selection (Section 13.5.1, page 200) to build classifiers with a much more limited number of features (300) and got as good or better results than Joachims with just linear SVMs.

	Rocchio	NB	BayesNets	Trees	SVM (poly degree = 1)
earn	92.9%	95.9%	95.8%	97.8%	98.0%
acq	64.7%	87.8%	88.3%	89.7%	93.6%
money-fx	46.7%	56.6%	58.8%	66.2%	74.5%
grain	67.5%	78.8%	81.4%	85.0%	94.6%
crude	70.1%	79.5%	79.6%	85.0%	88.9%
trade	65.1%	63.9%	69.0%	72.5%	75.9%
interest	63.4%	64.9%	71.3%	67.1%	77.7%
ship	49.2%	85.4%	84.4%	74.2%	85.6%
wheat	68.9%	69.7%	82.7%	92.5%	91.8%
corn	48.2%	65.3%	76.4%	91.8%	90.3%
micro-Avg Top 10	64.6%	81.5%	85.0%	88.4%	92.0%
micro-Avg All Cat	61.7%	75.2%	80.0%	N/A	87.0%

► **Table 15.1** SVM classifier break-even performance from Dumais et al. (1998). Results are shown for the 10 largest categories and over all categories on the Reuters-21578 data set. Micro- and macro-averaging were defined in Table 13.6 (page 207).

					SVM (poly degree)					SVM (rbf width)			
	NB	Rocchio	Trees	kNN	1	2	3	4	5	0.6	0.8	1.0	1.2
earn	95.9	96.1	96.1	97.3	98.2	98.4	98.5	98.4	98.3	98.5	98.5	98.4	98.3
acq	91.5	92.1	85.3	92.0	92.6	94.6	95.2	95.2	95.3	95.0	95.3	95.3	95.4
money-fx	62.9	67.6	69.4	78.2	66.9	72.5	75.4	74.9	76.2	74.0	75.4	76.3	75.9
grain	72.5	79.5	89.1	82.2	91.3	93.1	92.4	91.3	89.9	93.1	91.9	91.9	90.6
crude	81.0	81.5	75.5	85.7	86.0	87.3	88.6	88.9	87.8	88.9	89.0	88.9	88.2
trade	50.0	77.4	59.2	77.4	69.2	75.5	76.6	77.3	77.1	76.9	78.0	77.8	76.8
interest	58.0	72.5	49.1	74.0	69.8	63.3	67.9	73.1	76.2	74.4	75.0	76.2	76.1
ship	78.7	83.1	80.9	79.2	82.0	85.4	86.0	86.5	86.0	85.4	86.5	87.6	87.1
wheat	60.6	79.4	85.5	76.6	83.1	84.5	85.2	85.9	83.8	85.2	85.9	85.9	85.9
corn	47.3	62.2	87.7	77.9	86.0	86.5	85.3	85.7	83.9	85.1	85.7	85.7	84.5
microavg.	72.0	79.9	79.4	82.3	84.2	85.1	85.9	86.2	85.9	86.4	86.5	86.3	86.2

► **Table 15.2** SVM classifier break-even performance from Joachims (1998). Results are shown for the 10 largest categories on the Reuters-21578 data set.

This mirrors the results discussed in Chapter 14 (page 222) on other linear approaches like Naive Bayes. At a minimum, it seems that working with simple word features can get one a long way. It is also noticeable the degree to which the two papers' results for other learning methods differ. In text classification, there's always more to know than simply which machine learning algorithm was used.

These and other results have shown that simple classifiers such as Naive Bayes classifiers are uncompetitive with classifiers like SVMs when trained

and tested on independent and identically distributed (i.i.d.) data, that is, uniform data with all the good properties of statistical sampling. However, these differences may often be invisible or even reverse themselves when working in the real world where, usually, the training sample is drawn from a subset of the data to which the classifier will be applied, the nature of the data drifts over time rather than being stationary, and there may well be errors in the data (among other problems). For general discussion of this issue see Hand (2006). Many practitioners have had the experience of being unable to build a fancy classifier for a certain problem that consistently performs as well as Naive Bayes.

15.5 Issues in the categorization of text documents

15.6 References and further reading

There are now a number of books dedicated to SVMs, large margin learning, and kernels of which currently the two best are probably Schölkopf and Smola (2001) and Shawe-Taylor and Cristianini (2004). Well-known, good article length introductions are Burges (1998) and Chen et al. (2005), the latter of which introduces the more recent ν-SVM, which provides an alternative parameterization for dealing with inseparable problems, whereby rather than specifying a penalty C, one specifies a parameter ν which bounds the number of examples which can appear on the wrong side of the decision surface. For the foundations by their originator, see Vapnik (1998). Other recent, more general books on statistical learning also give thorough coverage to SVMs, for example, Hastie et al. (2001).

The kernel trick was first presented in (Aizerman et al. 1964). For more about string kernels and other kernels for structured data, see (Lodhi et al. 2002, Gaertner et al. 2002). The Advances in Neural Information Processing (NIPS) conferences have become the premier venue for theoretical machine learning work, such as on SVMs. Other venues such as SIGIR are much stronger on experimental methodology and using text-specific features to leverage performance.

A recent comparison of most current machine learning classifiers (though on problems rather different from typical text problems) can be found in Caruana and Niculescu-Mizil (2006). Older examinations of a more limited set of classifiers on text classification problems can be found in (Yang 1999, Yang and Liu 1999, Dumais et al. 1998). Joachims (2002a) presents his work on SVMs applied to text problems in detail. Zhang and Oles (2001) have insightful comparisons of Naive Bayes, regularized logistic regression and SVM classifiers.

16 *Flat clustering*

CLUSTER

Clustering algorithms partition a set of documents into subsets or *clusters*. The goal is to create clusters that are coherent internally, but clearly different from each other. In other words, documents within a cluster should be as similar as possible; and documents in one cluster should be as dissimilar as possible from documents in other clusters.

UNSUPERVISED LEARNING

Clustering is the most common form of *unsupervised learning*. We don't have any labeled data as in supervised text classification (Chapter 13, page 192), where the classes are predefined and part of the input to the learning procedure. In clustering, it is the distribution and makeup of the data – instead of definitions of classes supplied by a human judge or labeler – that will determine cluster membership. However, as we will see in this chapter and the next, although there is no direct intervention through labeling, clustering depends a lot on the clustering framework that we choose, e.g., the clustering algorithm and the similarity measure.

FLAT CLUSTERING

Fundamentally, there are two types of clustering algorithms: flat and hierarchical clustering algorithms. *Flat clustering* creates a "flat" set of clusters without any explicit structure that would relate clusters to each other. *Hierachical algorithms* create a hierarchy of clusters and will be covered in Chapter 17. Chapter 17 also addresses the difficult problem of labeling clusters automatically.

This chapter motivates the use of clustering in information retrieval by introducing a number of applications (Section 16.1), defines the problem we are trying to solve in clustering (Section 16.2) and discusses measures for evaluating cluster quality (Section 16.3). It then describes two flat clustering algorithms, k-means (Section 16.4) and the Expectation-Maximization (or EM) algorithm (Section 16.5). K-means is perhaps the most widely used flat clustering algorithm due to its simplicity and efficiency. The EM algorithm can be viewed as a generalization of k-means and can be applied to a larger variety of document representations and distributions.

Application	What is clustered?	Benefit	Example
Result set clustering	result set	more effective information presentation to user	Figure 16.1
Collection clustering	collection	effective information presentation for exploratory browsing	McKeown et al. (2002), http://news.google.com
Scatter-Gather	(subsets of) collection	alternative user interface: "search without typing"	Figure 16.2
Language modeling	collection	increased precision and/or recall	Liu and Croft (2004)
Cluster-based retrieval	collection	higher efficiency: faster search	Salton (1975), Singitham et al. (2004)

► **Table 16.1** Some applications of clustering in information retrieval.

16.1 Clustering in information retrieval

CLUSTER HYPOTHESIS

The *cluster hypothesis* states the fundamental assumption we make when using clustering in information retrieval:

> **Cluster hypothesis.** Documents in the same cluster behave similarly with respect to relevance to queries.

The hypothesis states that if there is a document from a cluster that is relevant to a search request, then it is likely that other documents from the same cluster are also relevant. This is because clustering puts together documents that contain the same words. The cluster hypothesis is essentially identical to the contiguity hypothesis in Chapter 14 (page 213). In both cases, we posit that similar documents behave similarly with respect to relevance.

Table 16.1 shows some of the main applications of clustering in information retrieval. They differ in the set of documents that they cluster – result set, collection or subsets of the collection – and the aspect of an information retrieval system they try to improve – user experience, user interface, effectiveness or efficiency of the search system. But they are all based on the basic assumption stated by the cluster hypothesis – that documents within a cluster all have a similar degree of relevance to an information need.

RESULT SET CLUSTERING

The first application of clustering mentioned in Table 16.1 is *result set clustering*. The default presentation of search results in information retrieval is a simple list. Users scan the list from top to bottom until they have found the information they are looking for. Instead, result set clustering clusters the result set, so that similar documents appear together. It is often easier to scan coherent groups than individual documents. This is particularly use-

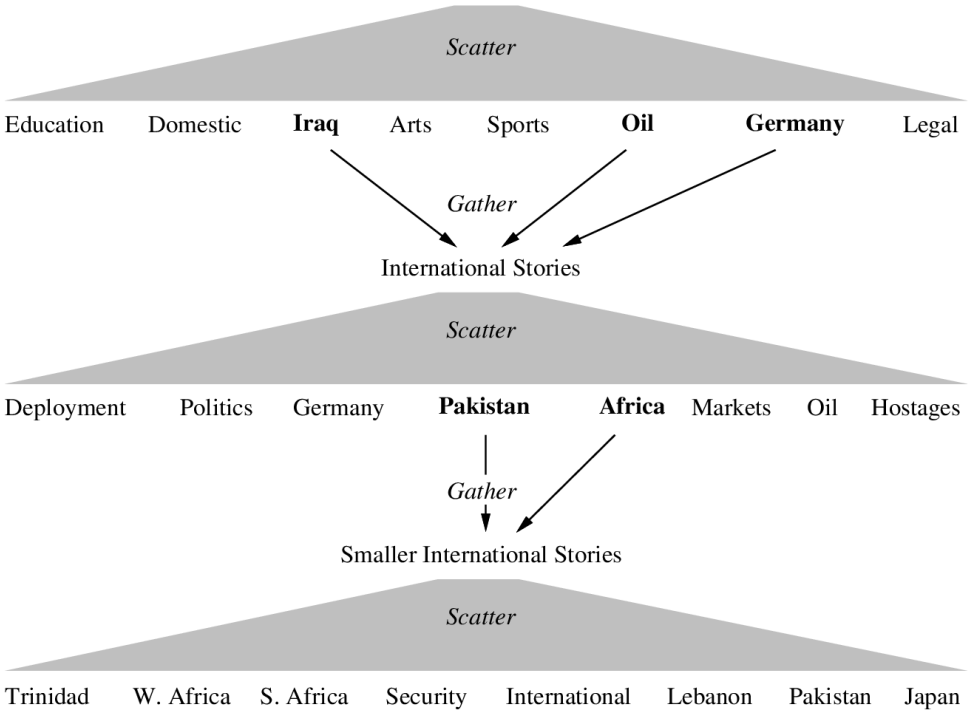
► **Figure 16.1** Clustering of search results to improve user recall. None of the top hits covers the animal sense of jaguar, but users can easily access it by clicking on the *cat* cluster in the *Clustered Results* panel on the left (third arrow from the top).

ful if a search term has different word senses. The example in Figure 16.1 is jaguar. Three frequent senses on the web refer to the car, the animal and an Apple operating system. Clearly, the *Clustered Results* panel returned by the Vivisimo search engine is a more effective user interface for understanding what's in the search result than a simple list of documents.

SCATTER-GATHER

A better user interface is also the goal of *Scatter-Gather*, the second application of clustering. Scatter-Gather clusters the whole collection to get groups of documents that the user can select ("gather"). The selected groups are merged and the resulting set is again clustered. This process is repeated until a cluster of interest is found. An example is shown in Figure 16.2.

Automatically generated clusters like those in Figure 16.2 will never be as neatly organized as a manually constructed hierarchical tree like the online Yahoo directory. Also, finding descriptive labels for clusters automatically is a difficult problem (Section 17.5, page 283). But cluster-based navigation is an interesting alternative to keyword searching, the standard information retrieval paradigm. This is especially true in scenarios where users prefer browsing over searching because they are unsure about which search terms



► **Figure 16.2** The Scatter-Gather user interface. A collection of New York Times news stories is clustered ("scattered") into eight clusters (top row). The user manually *gathers* three of these into a smaller collection *International Stories* and performs another scattering operation. This process repeats until a small cluster with relevant documents is found (e.g., *Trinidad*).

to use.

The third application of clustering attempts to exploit the cluster hypothesis directly for improving search results, based on a clustering of the entire collection. We use a standard inverted index to identify an initial set of documents that match the query, but we then add other documents from the same clusters even if they have low similarity to the query. For example, if the query is car and several car documents are taken from a cluster of automobile documents, then we can add documents from this cluster that use terms other than car (automobile, vehicle etc). This can increase recall since a group of documents with high mutual similarity is often relevant as a whole.

More recently this idea has been used for language modeling. Equation (12.5), page 182, showed that to avoid sparse data problems in the language mod-

eling approach to IR, the model of document d can be interpolated with a collection model. But the collection contains many documents with a word distribution untypical of d. By interpolating the document model with a model derived from d's cluster, we get more accurate estimates of the occurrence probabilities of words in d. Again, the underlying assumption is that a cluster groups similar documents together.

Clustering can also speed up search. As we saw in Section 7.1.2, page 100, search in the vector space model amounts to finding the nearest neighbors to the query. The inverted index supports fast nearest-neighbor search for the standard IR setting. However, sometimes we may not be able to use an inverted index efficiently, e.g., in latent semantic indexing (Chapter 18). In such cases, we could compute the similarity of the query to every document, but this is slow. The cluster hypothesis offers an alternative: Find the clusters that are closest to the query and only consider documents from these clusters. Within this much smaller set, we can compute similarities exhaustively and rank documents in the usual way. Since there are many fewer clusters than documents, finding the closest cluster is fast; and since the documents matching a query are all similar to each other, they tend to be in the same clusters. So the algorithm is inexact, but the expected decrease in search quality is small. This is essentially the application that was covered in Section 7.2.1 (page 104).

16.2 Problem statement

We can define the flat clustering problem as follows. Given are (i) a set of documents $D = \{d_1, \ldots, d_N\}$, (ii) a similarity measure (or distance metric), (iii) a partitioning criterion, and (iv) a desired number of clusters K. We then want to compute an assignment function $\gamma : D \rightarrow \{1, \ldots, K\}$ such that γ satisfies the partitioning criterion with respect to the similarity measure.

For most partitioning algorithms, the criterion is to find the clustering that optimizes an objective function. For example, k-means attempts to minimize the average (squared) distance from the cluster center (see next section). As in this example, the partitioning criterion often directly depends on the similarity measure.

The choice of underlying similarity measure is critical for getting meaningful clustering results. For documents, the type of similarity we want is usually *content similarity*, which we approximate using the now familiar vector space representation and cosine similarity (Chapter 7). If the partitioning criterion is something other than content, for example, language, then a different representation may be appropriate. When computing content similarity, stop words can be safely ignored, but they are important cues for separating clusters of, say, English and French documents.

Most similarity measures have a corresponding distance measure, which is often a distance metric. We observed earlier (Exercise 7.1, page 100) that there is a direct correspondence between cosine similarity and Euclidean distance for length-normalized vectors. In most cases, it doesn't matter whether the affinity of two documents is discussed in terms of similarity or distance and both views will be used interchangeably in this chapter and in Chapter 17.

CARDINALITY

One of the most difficult issues in clustering is to determine the *cardinality* of a clustering, the optimal number K of clusters. Often K is nothing more than a good guess based on experience and domain knowledge. But for k-means, we will also introduce a heuristic method for choosing K and an attempt to incorporate the selection of K into the objective function. Sometimes the application puts constraints on the range of K. For example, the Scatter-Gather interface in Figure 16.2 could not display more than about $K = 10$ clusters per layer because of the size of computer monitors in the 1990s and there is an analogous upper limit, perhaps $K = 20$ or $K = 30$, today.

If your goal is to optimize an objective function, then clustering is essentially a search problem. The brute force solution would be to enumerate all possible partitions and pick the best. However, for N documents and K clusters, there are $K^N/K!$ different partitions, so this approach is not feasible. For this reason, most partitioning algorithms refine an initial partitioning iteratively. If the search starts at an unfavorable initial point, we may miss the global optimum. Finding a good starting point is therefore another important problem we have to solve in flat clustering.

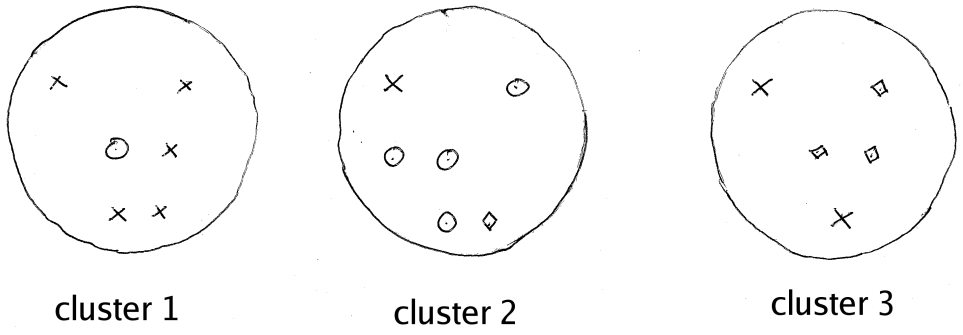
16.3 Evaluation of clustering

Partitioning criteria formalize the goal of attaining high intra-cluster similarity (documents within a cluster are similar) and low inter-cluster similarity (documents from different clusters are dissimilar). This is an *internal criterion* for the quality of a clustering. But good scores on an internal criterion don't necessarily translate into good performance in an application. An alternative to internal criteria is direct evaluation of the usefulness of clustering in the application of interest. For result set clustering, we may want to measure the time it takes users to find an answer with different clustering algorithms. This is the most direct evaluation, but it is expensive, especially if large user studies are necessary.

INTERNAL CRITERION OF QUALITY

As a surrogate for user judgments, we can use a set of classes in an evaluation benchmark or gold standard (see Section 8.3, page 119, and Section 13.6, page 206). The gold standard is ideally produced by human judges with a good level of inter-judge agreement (see Chapter 8, page 120). We can then compute an *external criterion* that evaluates how well the clustering matches the gold standard classes. For example, we may want to say that the optimal

EXTERNAL CRITERION OF QUALITY



► **Figure 16.3** Purity as an external evaluation criterion for cluster quality. Majority class and purity for the three clusters are: X, $\frac{5}{6}$ (cluster 1); circle, $\frac{4}{6}$ (cluster 2); and diamond, $\frac{3}{5}$ (cluster 3). Weighted average purity is $6/17 \times 5/6 + 6/17 \times 4/6 + 5/17 \times 3/5 \approx 0.71$.

clustering of the jaguar result set in Figure 16.1 consists of three classes corresponding to the three senses *car*, *animal*, and *operating system*. This section introduces four external criteria for evaluating how well a clustering reproduces the classes in the gold standard: purity, mutual information, Rand index and F measure. Purity is simple and transparent. Mutual information has the advantage of being information-theoretically motivated. These two measures are only appropriate if there is some type of constraint on the number of clusters or if the number of clusters is given. The Rand index evaluates both false positive and false negative decisions; the F measure in addition supports differential weighting of recall and precision by adjusting the parameter β. Which criterion is most appropriate depends on the characteristics of the clustering problem at hand.

PURITY The first measure is *purity*. Suppose we have K clusters $\omega_1, \omega_2, \ldots, \omega_K$ and J classes $c_1, c_2, \ldots, c_J$. Then the purity of a cluster is defined as follows:

$$\text{purity}(\omega_k) = \frac{1}{|\omega_k|} \max_j |\omega_k \cap c_j|$$

An overall measure of purity can be defined as a sum of cluster purities, weighted by the size of the clusters. An example of how to compute purity is presented in Figure 16.3. High purity is easy to achieve for a large number of clusters – in particular, purity is 1 if each document gets its own cluster. Thus, we cannot use purity to obtain a tradeoff between the quality of the clustering and the number of clusters.

An alternative to purity is mutual information (MI), which we are already

	same cluster	different clusters
same class	TP= 20	FN= 24
different classes	FP= 20	TN= 72

► **Table 16.2** The Rand index for pair-based evaluation of the clustering in Figure 16.3. The Rand index in this case is $(20 + 72)/(20 + 20 + 24 + 72) \approx 0.68$.

familiar with from Chapter 13 (page 200) as a feature selection method. MI measures the amount of information by which our knowledge about the classes increases when we are told what the clusters are:

$$I(\{\omega_1, \ldots, \omega_K\}; \{c_1, \ldots, c_J\}) = \sum_k \sum_j P(\omega_k c_j) \log \frac{P(\omega_k c_j)}{P(\omega_k)P(c_j)}$$

where $P(\omega_k)$, $P(c_j)$, and $P(\omega_k c_j)$ are the probabilities (or relative frequencies) of a document being in ω_k, c_j, and in the intersection of ω_k and c_j, respectively. Mutual information reaches a minimum of 0 if the clustering is random with respect to the classes. In that case, knowing that a document is in a particular cluster doesn't give us any new information about what class it might be in. Maximum mutual information is reached for a clustering Ω_{exact} that perfectly recreates the classes – but also if clusters in Ω_{exact} are further subdivided into smaller clusters (Exercise 16.4). So MI also has a problem with evaluating clusterings with a large number of clusters.

If we view our task in clustering as recognizing that a pair of similar documents belongs to the same cluster, then there are two types of errors we can commit. A *false positive* (FP) decision assigns two dissimilar documents to the same cluster. A *false negative* (FN) decision assigns two similar documents to different clusters. Purity and mutual information only directly penalize false positives.

RAND INDEX

A measure that penalizes both false positives and false negatives is the *Rand index* RI. It is defined as

$$\text{RI} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{FN} + \text{TN}}$$

where TP and TN are true positives and true negatives, respectively.

As an example for how to compute the numbers in Table 16.2, consider again Figure 16.3. The three clusters contain 6, 6, and 5 points, respectively, so the total number of pairs that are in the same cluster is:

$$\text{TP} + \text{FP} = \binom{6}{2} + \binom{6}{2} + \binom{5}{2} = 40$$

Of those, the X pairs in cluster 1, the circle pairs in cluster 2, the diamond

pairs in cluster 3, and the X pair in cluster 3 are true positives:

$$\text{TP} = \binom{5}{2} + \binom{4}{2} + \binom{3}{2} + \binom{2}{2} = 20$$

So FP $= 40 - 20 = 20$. FN and TN are computed similarly.

The Rand index gives equal weight to true positives and true negatives. In some situations, putting pairs of similar documents in the same cluster is more important than separating dissimilar documents. We can use the standard IR evaluation measures precision, recall and *F measure* (Section 8.1.2, page 111) to focus on true positives by selecting a value $\beta > 1$ (which puts higher weight on recall R):

F MEASURE

$$P = \frac{\text{TP}}{\text{TP} + \text{FP}}, R = \frac{\text{TP}}{\text{TP} + \text{FN}}, F_\beta = \frac{(\beta^2 + 1)PR}{\beta^2 P + R}$$

According to Table 16.2, $P = 20/40 = 0.5, R = 20/44 \approx 0.46$, and $F_1 \approx 0.48$. In information retrieval, evaluating clustering with F has the advantage that the measure is already familiar to the research community. Still, the Rand index currently seems to be the most widely used external criterion for cluster evaluation.

16.4 K-means

K-means is the most important flat clustering algorithm. Its objective is to minimize the average squared distance of documents from their cluster center where the cluster center is defined as the mean or *centroid* $\vec{\mu}$ of the documents in a cluster ω:

CENTROID

$$\vec{\mu}(\omega) = \frac{1}{|\omega|} \sum_{\vec{x} \in \omega} \vec{x}$$

This definition assumes that documents are represented as vectors in a real-valued space in the familiar way. We used centroids for Rocchio classification in Chapter 14 (page 215). They play a very similar role here once the clusters are known. But initially we do not have labeled training data – recall that clustering is *unsupervised* learning – , so computing centroids is not as simple as in the Rocchio case.

A measure of how well the centroids represent the members of their clusters is the *residual sum of squares* or *RSS*, the squared distance of each vector from its centroid summed over all vectors in an M-dimensional space:

RESIDUAL SUM OF SQUARES

$$\text{RSS}_k = \sum_{\vec{x} \in \omega_k} |\vec{x} - \vec{\mu}(\omega_k)|^2 = \sum_{m=1}^{M} (x_m - \vec{\mu}_m(\omega_k))^2$$

Given:
  X: a set of N vectors
  d: distance metric
  K: desired number of clusters

Select K random seeds $\{\vec{s}_1, \vec{s}_2, \ldots, \vec{s}_K\}$ from X
Let $\vec{\mu}(\omega_k) := \vec{s}_k$, $1 \leq k \leq K$
Repeat until stopping criterion is met:
  Reassignment step
    Assign each $\vec{x}_n$ to cluster ω_k s.t. $d(\vec{x}_n, \vec{\mu}(\omega_k))$ is minimal
  Recomputation step: For each ω_k:
    $\vec{\mu}(\omega_k) = \frac{1}{|\omega_k|} \sum_{\vec{x} \in \omega_k} \vec{x}$

► **Figure 16.4** The k-means algorithm.

$$\text{RSS} = \sum_{k=1}^{K} \text{RSS}_k \tag{16.1}$$

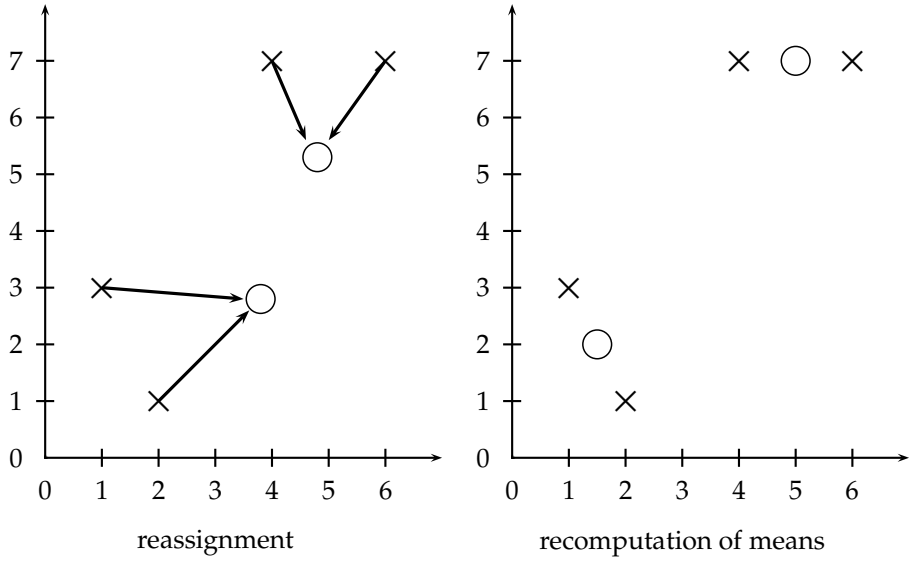
where $\vec{\mu}_m(\omega_k)$ is component m of $\vec{\mu}(\omega_k)$. This is the objective function in k-means and our goal is to minimize it. Since N is fixed, minimizing the average squared distance from the centroid and minimizing RSS are equivalent.

SEED K-means usually starts with K randomly selected documents, the *seeds*. It then moves the cluster centroids $\vec{\mu}_k$ around in space to minimize RSS. As shown in Figure 16.4, this is done iteratively by repeating two steps until a stopping criterion is met: reassigning documents to the cluster with the closest centroid; and recomputing each centroid based on the current members of its cluster. Figure 16.5 shows one iteration of the k-means algorithm for a set of four points.

Several termination conditions can be used as stopping criteria:

- A fixed number of iterations I has been completed.
- Assignment of documents to clusters (the partitioning function γ) does not change between iterations.
- Centroids $\vec{\mu}_k$ do not change between iterations.

Since RSS monotonically decreases in each iteration, k-means converges and will therefore terminate if we use the last two conditions. First, RSS decreases in the reassignment step since each vector is assigned to the closest centroid, so the distance it contributes to RSS decreases (or does not change). Secondly, it decreases in the recomputation step because the new centroid is



► **Figure 16.5** One iteration of the k-means algorithm in $\mathbb{R}^2$. The position of the two centroids (shown as circles) converges after one iteration.

the vector $\vec{v}$ for which RSS_k reaches its minimum.

$$\text{RSS}_k(\vec{v}) = \sum_{\vec{x} \in \omega_k} |\vec{v} - \vec{x}|^2 = \sum_{\vec{x} \in \omega_k} \sum_{m=1}^{M} (v_m - x_m)^2 \tag{16.2}$$

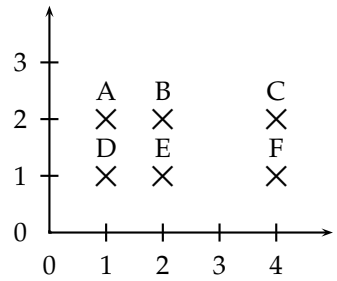
$$\frac{\partial \text{RSS}_k(\vec{v})}{\partial v_m} = \sum_{\vec{x} \in \omega_k} 2(v_m - x_m) \tag{16.3}$$

By setting the partial derivative to zero, we get:

$$v_m = \frac{1}{|\omega_k|} \sum_{\vec{x} \in \omega_k} x_m \tag{16.4}$$

which is the componentwise definition of the centroid. So RSS_k reaches its minimum when the centroid is recomputed and therefore RSS also decreases during recomputation.

Since there is only a finite set of possible clusterings, a monotonically decreasing algorithm will eventually arrive at a minimum. Take care, however,



► **Figure 16.6** The outcome of clustering in k-means depends on the initial seeds. For seeds B and E, k-means converges to $\{A, B, C\}, \{D, E, F\}$, a suboptimal clustering. For seeds D and F, it converges to $\{A, B, D, E\}, \{C, F\}$, the global optimum for $K = 2$.

to break ties consistently, e.g., by assigning a document to the cluster with the lowest index in case there are several with the same distance. Otherwise, the algorithm can cycle forever in a loop of clusterings that have the same cost.

This proves the convergence of k-means, but there is unfortunately no guarantee that a *global minimum* will be reached. This is a particular problem if a document set contains many outliers. If an outlier is chosen as an initial seed, then it may well happen that no other vector is assigned to it during subsequent iterations. We end up with a *singleton cluster* (a cluster with only one document) even though there is probably a clustering that is a better representation of the overall structure of the set. Figure 16.6 shows an example of a suboptimal clustering resulting from a bad choice of initial seeds.

SINGLETON CLUSTER

What is the time complexity of k-means? Most of the time is spent on computing vector distances. One such operation costs $O(M)$. The reassignment step computes $O(KN)$ distances, so its overall complexity is $O(KNM)$. In the recomputation step, each vector gets added to a centroid once, so the complexity of this step is $O(NM)$. For a fixed number of iterations I, the overall complexity is therefore $O(IKNM)$. So k-means is linear in all relevant factors: iterations, number of clusters, number of vectors and dimensionality of the space. This means that k-means is more efficient than the hierachical algorithms that are the topic of the next chapter. We had to fix the number of iterations I, but this rarely does harm in practice. In most cases, k-means converges quickly.

There is one subtlety in the preceding argument. Even a linear algorithm can be quite slow if the n in $O(n)$ is large, and M is usually large. A dimensionality of $M >$10,000 is common. High dimensionality is not a problem for computing the distance of two documents. Their vectors are sparse, so that,

on average, a small fraction of the theoretically possible M componentwise differences need to be computed. Centroids, however, are dense since they pool all terms that occur in any of the documents of their clusters. As a result, distance computations are time consuming in a naive implementation of k-means. But there are simple and effective heuristics for making centroid-document similarities as fast to compute as document-document similarities. Truncating centroids to the most significant k terms (e.g., $k = 1000$) hardly decreases cluster quality while achieving a significant speedup of the reassignment step (see references in Section 16.6).

K-MEDOIDS
MEDOID

The same problem is addressed by *k-medoids*, a variant of k-means that computes medoids instead of centroids as cluster centers. The *medoid* of a cluster is the document vector that is closest to the centroid. Since medoids are sparse document vectors, distance computations are fast.

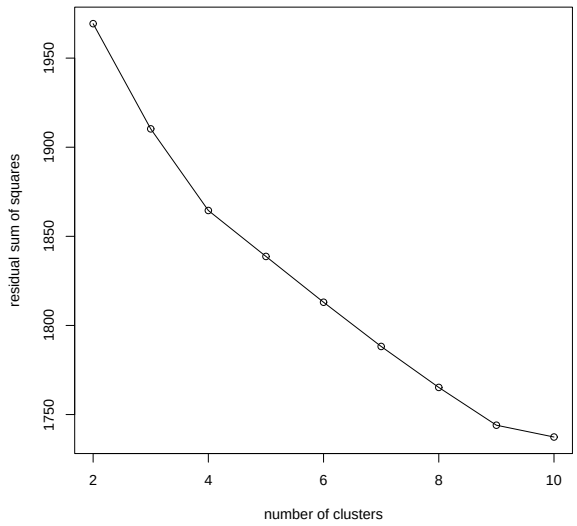
Another thorny implementation issue is the choice of seeds. We already mentioned that selecting outliers as seeds is bad. But there are effective heuristics for finding good seeds. We can remove outliers from the seed set; try out multiple starting points and choose the clustering with lowest cost; or obtain seeds from another method such as hierarchical clustering (see Chapter 17). Since hierarchical clustering methods are deterministic and therefore more predictable than k-means (although they are slower), a hierarchical clustering of a small random sample of size ℓK (e.g., for $\ell = 5$ or $\ell = 10$) will often provide good seeds. If seeds are provided by a method other than random selection, they can be arbitrary vectors and need not be members of the set of documents to be clustered.

16.4.1 Cluster cardinality in k-means

We stated in Section 16.2 that the number of clusters K is an input to most flat clustering algorithms. But what do we do if we cannot come up with a plausible guess for K? We could attempt to exploit the partitioning criterion minimizing RSS, for determining K. But just computing RSS for all possible K and then choosing the optimum won't work because RSS is a monotonically decreasing function in K. We would end up with each document being assigned its own cluster. Clearly, having as many clusters as documents does not constitute an optimal clustering.

A heuristic method that gets around this problem is to inspect the RSS values as the number of clusters increases and find the "knee" in the curve, the point where successive decreases in RSS become noticeably smaller. There are two such points in Figure 16.7, one at $K = 4$, where the gradient flattens slightly, and a clearer flattening at $K = 9$. This is fairly typical: there is seldom a single best number of clusters. We still need to employ an external constraint to choose from a number of possible values of K.

A second type of criterion for cluster cardinality imposes a penalty for each



► **Figure 16.7** Residual sum of squares (RSS) as a function of the number of clusters in k-means. In this clustering of 1203 Reuters-RCV1 documents, there are two points where the RSS curve flattens: at 4 clusters and at 9 clusters. The documents were selected from the categories *China*, *Germany*, *Russia* and *Sports*, so the $K = 4$ clustering is the closest to the Reuters categorization. Each data point is the average of 10 different trials with random seeds.

new cluster. To do this, we create an objective function that combines two elements: *fit*, a measure of how well the clustering models the data; and a measure of *model complexity* (often a function of the number of clusters). The two best known objective functions of this type are the *Akaike Information Criterion* or AIC and the *Bayesian Information Criterion* or BIC.

FIT
MODEL COMPLEXITY
AKAIKE INFORMATION CRITERION
BAYESIAN INFORMATION CRITERION

$$\textbf{AIC:} \quad K = \arg\min_K [-2L(K) + 2q(K)]$$
$$\textbf{BIC:} \quad K = \arg\min_K [-2L(K) + q(K)\ln(N)]$$

where $L(K)$ is the maximum log likelihood of the data for K clusters and $q(K)$ is the number of parameters of a model with K clusters.

It can be shown that for data with a normal distribution (which is implicitly assumed by optimizing RSS in k-means), these criteria are equivalent to:

(16.5) $$\textbf{AIC:} \quad K = \arg\min_K [N \ln \frac{\text{RSS}}{N} + 2q(K)]$$

(16.6) **BIC:** $K = \arg\min_K [N \ln \frac{\text{RSS}}{N} + q(K) \ln N]$

For k-means, $q(K) \approx KM$ since each element of the K centroids is a parameter that can be varied independently.

Normality is a crude approximation for the distribution of documents in a vector space. As a consequence, the two criteria can rarely be applied as is in text clustering. In Figure 16.7, the dimensionality of the vector space is $M \approx$ 50,000. Thus, $q(K) \approx 50{,}000K$ dominates the smaller RSS-based term (since RSS < 2000) and the minimum of the expression is reached for $K = 2$ in both cases. But as we know, $K = 4$ is a better choice than $K = 2$ here.

However, the general approach of penalizing each new cluster is still valid. This penalization can take the form $L(K) + \lambda q(K)$ (in analogy to AIC) or $L(K) + \lambda q(K) \log N$ (in analogy to BIC) where λ is the per-cluster penalty. If we can't trust the theoretically derived weighting in the two information criteria, then we must resort to an empirical method such as cross-validation or experience with the type of data we're dealing with to determine λ.

16.5 Model-based clustering

K-means searches for the clustering with the lowest RSS by attempting to minimize the average squared distance between vectors and their centroids. This captures the intuition that the centroid should be a good representative of its documents. A different way of posing the clustering problem is to formalize a clustering as a parameterized model Θ and then search for the parameters that maximize the likelihood of the data:

$$\textit{maximize} \quad L(D|\Theta)$$

This is the same approach we took in Chapter 13 for text classification where we also chose the model that was most likely to have generated the set of documents D.

HARD ASSIGNMENT

SOFT ASSIGNMENT

An important difference between k-means and model-based clustering is the way cluster membership works. K-means performs a *hard assignment*: each document is assigned to exactly one cluster. It cannot be a member of two or more clusters simultaneously. In general, model-based clustering performs a *soft assignment*. As with the generative models in Chapters 12 and 13, we use Bayes' rule to assign a document to a cluster by computing $P(\omega_k|d)$, the probability that d was generated by ω_k. The hidden variable here is the cluster ω_k. A document about Chinese cars may get soft assignments of 0.5 to each of the two clusters *China* and *automobiles*, reflecting the fact that both topics are pertinent. A hard clustering cannot model this simultaneous relevance to two topics.

Model-based clustering also provides a framework for incorporating our knowledge about a domain. K-means and the hierarchical algorithms in the

next chapter make fairly rigid assumptions about the data. For example, clusters in K-means are assumed to be spheres. Model-based clustering offers much more flexibility. The clustering model can be adapted to what we know about the underlying distribution of the data, be it binomial (as in the example below), Gaussian (another often occurring case in document clustering) or a member of a different family.

The result of model-based clustering is a model that defines γ, the cluster assignment function. It corresponds to the classification function γ in text classification. The crucial difference is that in classification we have a labeled training set, a definition of the classes by example. No labels are available in clustering and that makes the problem much harder.

EXPECTATION-MAXIMIZATION ALGORITHM
EM ALGORITHM

A commonly used algorithm for clustering in a generative framework is the *Expectation-Maximization algorithm* or *EM algorithm*. EM clustering is an iterative algorithm that maximizes $L(D|\Theta)$.

$$L(D|\Theta) = \log \prod_{n=1}^{N} P(d_n|\Theta) = \sum_{n=1}^{N} \log P(d_n|\Theta)$$

$L(D|\Theta)$ is the objective function that measures the goodness of the clustering. Given two clusterings with the same number of clusters, we prefer the one with higher $L(D|\Theta)$.

In principle, EM can be applied to any probabilistic modeling of the data. We will work with a mixture of multivariate binomials, the distribution we know from Section 11.3 (page 166) and Section 13.3 (page 198):

$$P(d|\omega_k, \Theta) = \prod_{m=1}^{M} P(X_m = I(w_m \in d)|\omega_k)$$

where $\Theta = \{\Theta_1, \ldots, \Theta_K\}$ and $\Theta_k = (\gamma_k, P(X_1 = e|\omega_k), P(X_2 = e|\omega_k), \ldots, P(X_M = e|\omega_k)), e \in \{0, 1\}$ are the parameters of the model and $I(w_m \in d_n)$ is 1 if word w_m occurs in document d_n and 0 otherwise. Example: for $P(X_1 = 1|\omega_2) = 0.3$, the probability of a document from cluster 2 containing word w_1 is 0.3.

The mixture model then is:

$$P(d|\Theta) = \sum_{k=1}^{K} \gamma_k \prod_{m=1}^{M} P(X_m = I(w_m \in d)|\omega_k) \tag{16.7}$$

In this model, we generate a document by first picking a cluster k with probability γ_k and then generating the words of the document according to the parameters $P(X_m = e|\omega_k)$. Whereas documents were vectors of M real-valued weights in k-means, the document representation of the multivariate binomial is a vector of M Boolean values.

How do we use EM to infer the parameters of the clustering from the data? That is, how do we choose parameters Θ that maximize $L(D|\Theta)$? EM is

EXPECTATION STEP
MAXIMIZATION STEP

actually quite similar to k-means in that it alternates between an *expectation step*, corresponding to reassignment, and a *maximization step*, corresponding to recomputation of the parameters of the models. (The parameters of the models in k-means are the centroids and the priors of the clusters.)

For the binomial mixture model, the maximization step recomputes the conditional parameters $q_{mk} = P(X_m = 1|\omega_k)$ and the priors γ_k as follows:

(16.8) **Maximization Step:** $$q_{mk} = \frac{\sum_{n=1}^{N} r_{nk} I(w_m \in d_n)}{\sum_{n=1}^{N} r_{nk}} \quad \gamma_k = \frac{\sum_{n=1}^{N} r_{nk}}{N}$$

r_{nk} is the soft assignment of document d_n to cluster k as computed in the preceding iteration. These are the maximum likelihood estimates for the parameters of the multivariate binomial from Table 13.2 (page 198) except that documents are assigned fractionally to clusters here. This guarantees that the maximization step computes the parameters that maximize the likelihood of the data given the model.

The expectation step computes the soft assignment of documents to clusters given the current parameters q_{mk} and γ_k:

(16.9) **ExpectationStep**: $$r_{nk} = \frac{\gamma_k (\prod_{w_m \in d_n} q_{mk})(\prod_{w_m \notin d_n} (1 - q_{mk}))}{\sum_{k=1}^{K} \gamma_k (\prod_{w_m \in d_n} q_{mk})(\prod_{w_m \notin d_n} (1 - q_{mk}))}$$

This expectation step applies 16.7 to computing the normalized likelihood that ω_k generated document d_n. It corresponds to the classification procedure for the multivariate Bernoulli in Table 13.2.

We clustered a set of 11 documents using EM in Table 16.3. After convergence in iteration 25, the first 5 documents are assigned to cluster 1 ($r_{i,1} = 1.00$) and the last 6 to cluster 2 ($r_{i,1} = 0.00$). Somewhat untypically, the final assignment is a hard assignment here. EM usually converges on a soft assignment. In iteration 25, the prior for cluster 1 is $5/11 \approx 0.45$ because 5 of the 11 documents are in cluster 1. Some words are quickly associated with one cluster because the initial assignment can "spread" to them unambiguously. For example, membership in cluster 2 spreads from sugar in document 7 to brazil in document 8 ($q_{\text{brazil},2} > 0$ starting in iteration 2). For words in ambiguous contexts, convergence takes longer. For example, both seed documents (6 and 7) contain sweet. As a result, it takes 25 iterations for the word to be unambiguously associated with one cluster.

Finding good seeds is even more critical for EM than for k-means. EM is prone to get stuck in local optima if the seeds are not chosen well. This is a general problem with EM which also occurs in applications of EM the reader may have encountered in other contexts: hidden markov models, probabilistic grammars, and machine translation. Therefore, as with k-means, the initial assignment of documents to clusters is often computed by a different

Documents

document	document text	seed for
1	hot chocolate cocoa beans	
2	cocoa ghana africa	
3	beans harvest ghana	
4	cocoa butter	
5	butter truffles	
6	sweet chocolate	cluster 1
7	sweet sugar	cluster 2
8	sugar cane brazil	
9	sweet sugar beet	
10	sweet cake icing	
11	cake black forest	

Clustering

	iteration							
	0	1	2	3	4	5	15	25
γ_1		0.50	0.45	0.52	0.56	0.55	0.53	0.45
$r_{1,1}$		1.00	1.00	1.00	1.00	1.00	1.00	1.00
$r_{2,1}$		0.50	0.73	0.97	1.00	1.00	1.00	1.00
$r_{3,1}$		0.50	0.80	0.99	1.00	1.00	1.00	1.00
$r_{4,1}$		0.50	0.71	0.87	0.98	1.00	1.00	1.00
$r_{5,1}$		0.50	0.53	0.61	0.78	0.96	1.00	1.00
$r_{6,1}$		1.00	1.00	1.00	1.00	1.00	0.74	0.00
$r_{7,1}$	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
$r_{8,1}$	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
$r_{9,1}$		0.00	0.00	0.00	0.00	0.00	0.00	0.00
$r_{10,1}$		0.50	0.42	0.19	0.02	0.00	0.00	0.00
$r_{11,1}$		0.50	0.56	0.53	0.28	0.02	0.00	0.00
$q_{\text{africa},1}$		0.000	0.036	0.046	0.057	0.061	0.064	0.071
$q_{\text{africa},2}$		0.000	0.031	0.019	0.002	0.000	0.000	0.000
$q_{\text{brazil},1}$		0.000	0.000	0.000	0.000	0.000	0.000	0.000
$q_{\text{brazil},2}$		0.000	0.062	0.071	0.077	0.074	0.070	0.062
$q_{\text{cocoa},1}$		0.000	0.143	0.152	0.167	0.181	0.191	0.214
$q_{\text{cocoa},2}$		0.000	0.063	0.040	0.012	0.001	0.000	0.000
$q_{\text{sugar},1}$		0.000	0.000	0.000	0.000	0.000	0.000	0.000
$q_{\text{sugar},2}$		0.500	0.187	0.214	0.231	0.221	0.210	0.187
$q_{\text{sweet},1}$		0.500	0.107	0.089	0.070	0.062	0.054	0.000
$q_{\text{sweet},2}$		0.500	0.156	0.185	0.216	0.219	0.220	0.250

► **Table 16.3** The EM clustering algorithm. The table shows a set of documents (above) and parameter values for selected iterations when clustering it with EM (below). Parameters shown are prior γ_1, soft assignment scores $r_{n,1}$ (both omitted for cluster 2), and lexical parameters $q_{m,k}$ for a few words. The random initialization (iteration 0) assigned document 6 to cluster 1 and document 7 to cluster 2 in this case. EM converges after 25 iterations. For smoothing, the r_{nk} in Equation 16.8 were replaced with $r_{nk} + \epsilon$ where $\epsilon = 0.0001$.

algorithm. For example, a hard k-means clustering may provide the initial assignment, which EM can then "soften up."

At the beginning of this section we motivated model-based clustering as a way of incorporating knowledge into clustering. The one structural property of EM that we cannot change is the fact that it is a mixture model. In reality, document collections are not generated by a simple mixture. Their composition involves a group of people who describe the world and share their views, a complex process for which a mixture is only a crude approximation. But at least we can get closer to an adequate formalization with model-based clustering. Insights about the domain can be built into the model and error analysis can reveal incorrect assumptions we have made. In comparison, non-model-based clustering, which is not based on explicit assumptions about the data, is harder to analyze and adapt.

16.6 References and further reading

A more general introduction to clustering, covering both k-means and EM, but without reference to text-specific issues, can be found in Duda et al. (2000). The EM algorithm was originally introduced by Dempster et al. (1977). Rasmussen (1992) gives an introduction to clustering from an information retrieval perspective. The cluster hypothesis is due to Jardine and van Rijsbergen (1971), where it was stated as follows: *Associations between documents convey information about the relevance of documents to requests.* Croft (1978) shows that cluster-based retrieval can be more accurate as well as more efficient than regular search. However, Voorhees (1985a) presents evidence that accuracy does not improve consistently across collections.

There is good evidence that clustering of result sets improves user experience and search result quality (Hearst and Pedersen 1996, Zamir and Etzioni 1999, Tombros et al. 2002, Käki 2005), although not as much as result set structuring based on carefully edited category hierarchies (Hearst 2006). The Scatter-Gather interface for browsing collections was presented by Cutting et al. (1992). The cluster-based language modeling approach was pioneered by Liu and Croft (2004). Schütze and Silverstein (1997) evaluate two methods for avoiding inefficient dense centroids in clustering.

The Columbia NewsBlaster system (McKeown et al. 2002), a forerunner to the now much more famous and refined Google News (http://news.google.com), used hierarchical clustering (Chapter 17) to give two levels of news topic granularity. See Hatzivassiloglou et al. (2000) for details. See also (Chen and Lin 2000, Radev et al. 2001). Other applications of clustering in information retrieval are duplicate detection (Yang and Callan 2006) and metadata discovery on the semantic web (Alonso et al. 2006).

The discussion of external evaluation measures is partially based on Strehl

(2002). Strehl also proposes a normalized mutual information (NMI) measure that penalizes high cardinalities (so that one cluster per document does not have maximum NMI). The Rand index is due to Rand (1971). Hubert and Arabie (1985) propose an *adjusted* Rand index that ranges between -1 and 1 and is 0 if there is only chance agreement between clusters and classes (similar to the kappa measure in Chapter 8, page 119).

AIC is due to Akaike (1974) and BIC to Schwarz (1978). BIC is applied to k-means by Pelleg and Moore (2000). Hamerly and Elkan (2003) propose an alternative to BIC that performs better in their experiments. Two methods of determining the number of clusters without external criteria are presented by Tibshirani et al. (2001).

We only have space here for classical completely unsupervised clustering. An important current topic of research is how to use prior knowledge to guide clustering (Ji and Xu 2006) and how to incorporate interactive feedback during clustering (Huang and Mitchell 2006). Fayyad et al. (1998) propose an initialization for EM clustering. For algorithms that can cluster very large data sets in one scan through the data see Bradley et al. (1998).

16.7 Exercises

Exercise 16.1

Define two documents to be associated if they have n content words in common (for, say, $n = 5$). What are some examples of documents for which the cluster hypothesis does not hold?

Exercise 16.2

Why would documents that do not use the same word for the concept *car* end up in the same cluster?

Exercise 16.3

Make up a simple example with points on a line in two clusters where the inexactness of cluster-based retrieval shows up, that is, retrieving clusters close to the query does worse than direct nearest neighbor search.

Exercise 16.4

Let Ω be a clustering that exactly reproduces a class structure C and Ω' a clustering that further subdivides some clusters. Show that $I(\Omega, C) = I(\Omega', C)$.

Exercise 16.5

Compute the mutual information between clusters and classes in Figure 16.3.

Exercise 16.6

Replace every point d in Figure 16.3 with two points in the same class as d. Compute F for this clustering. Is F higher or lower? Does this correspond to your intuition as to the relative difficulty of clustering a set twice as large?

Exercise 16.7

Two of the possible termination conditions for k-means were (1) assignment doesn't change, (2) centroids don't change (page 256). Do these two conditions imply each other?

Exercise 16.8

Compute RSS for the two clusterings in Figure 16.6.

Exercise 16.9

Synonymy is often a problem in keyword retrieval. Searching for car will miss documents on cars that use other words to describe the concept. Would you expect this type of phenomenon to be a problem in creating good clusters in k-means clustering? Why?

Exercise 16.10

Ambiguity is often a problem in keyword retrieval. Searching for suit will retrieve documents on lawsuits as well as on tailored suits. Would you expect ambiguity to be a problem in creating good clusters in k-means clustering? Why?

Exercise 16.11

K-means can also support soft clustering based on distance to centroid. How would you modify reassignment and recomputation for this soft version?

Exercise 16.12

In the last iteration in Table 16.3, document 6 is in cluster 2 even though it was the initial seed for cluster 1. Why does the document change membership?

Exercise 16.13

The values of the parameters q_{mk} in iteration 25 in Table 16.3 are rounded. What are the exact values that EM will converge to?

Exercise 16.14

Perform a k-means clustering for the documents in Table 16.3. After how many iterations does k-means converge? Compare the result to the EM clustering in Table 16.3 and discuss the differences.

Exercise 16.15

Modify the expectation and maximization steps of EM for a Gaussian mixture. As with Naive Bayes, the maximization step computes the maximum likelihood parameter estimates γ_k, $\vec{\mu}_k$, and Σ_k for each of the clusters. The expectation step computes for each vector a soft assignment to clusters (Gaussians) based on their current parameters.

Write down the corresponding equations.

Exercise 16.16

Show that k-means can be viewed as the limiting case of EM for Gaussian mixtures if variance is very small and all covariances are 0.

Exercise 16.17

We saw above that the time complexity of k-means is $O(IKNM)$. What is the time complexity of EM?

Exercise 16.18

K-means and Rocchio classification are closely related. Explain how.

Exercise 16.19

WITHIN-POINT SCATTER

The *within-point scatter* of a cluster ω is defined as

$$\frac{1}{2}\sum_{x_i\in\omega}\sum_{x_j\in\omega}|\vec{x}_i-\vec{x}_j|^2$$

Show that k-means finds a set of clusters with minimal within-point scatter.

Exercise 16.20

Write down the AIC and BIC criteria (the general form on page 260) for the multivariate binomial mixture in 16.7.

Exercise 16.21

Consider a mixed collection of English and French documents. (a) What result would you expect if this collection is clustered with k-means with $K = 2$? (b) We now want to cluster the collection with k-means with $K = 20$ with the goal of getting language-independent topical clusters (*government*, *arts*, *science* etc). How could one represent the documents to achieve this?

17 *Hierarchical clustering*

HIERARCHICAL CLUSTERING

Flat clustering is efficient and conceptually simple, but as we saw in Chapter 16 it has a number of drawbacks. It returns a flat unstructured set of clusters; it requires a prespecified number of clusters as input; and the flat clustering algorithms k-means and EM are not guaranteed to find the optimal set of clusters. *Hierarchical clustering* (or *hierarchic clustering*) outputs a hierarchy, a structure that is more informative than the unstructured set of clusters in flat clustering. It does not require us to prespecify the number of clusters. And most hierarchical algorithms find the optimal solution. These advantages of hierarchical clustering come at the cost of lower efficiency. Hierarchical clustering algorithms have a complexity that is at least quadratic in the number of documents compared to the linear complexity of k-means and EM.

This chapter first introduces *agglomerative* hierarchical clustering (Section 17.1). We then present four different agglomerative algorithms, in Sections 17.2–17.4, which differ in the similarity measure they employ: single-link, complete-link, group-average, and centroid clustering. Section 17.5 looks at automatic labeling of clusters, which is important whenever humans interact with the output of clustering. Finally, we discuss variants of hierarchical clustering algorithms and implementation issues in Sections 17.6 and 17.7.

In principle, the possible applications of flat and hierarchical clustering in information retrieval differ little. In particular, hierarchical clustering is appropriate for any of the applications shown in Table 16.1 (page 248) (see also Section 16.6, page 265). In fact, the example we gave for collection clustering is hierarchical. In general, we select flat clustering when efficiency is important and hierarchical clustering when one of the potential problems of flat clustering (not enough structure, predetermined number of clusters, no guarantee of optimality) is a concern. In addition, many researchers believe that hierarchical clustering produces better clusters than flat clustering on either internal or external criteria (Section 16.3, page 252). But there is no consensus on this issue (see references in Section 17.8).

17.1 Hierarchical agglomerative clustering

HIERARCHICAL AGGLOMERATIVE CLUSTERING
HAC

Hierarchical algorithms are either top-down or bottom-up. Bottom-up algorithms merge or *agglomerate* documents and clusters into larger and larger units. Bottom-up clustering is therefore called *hierarchical agglomerative clustering* or *HAC*. Top-down clustering requires a method for splitting a cluster and proceeds by splitting clusters recursively until individual documents are reached (see Section 17.6). HAC is more frequently used than top-down clustering and is the main subject of this chapter.

HAC treats each document as a singleton cluster at the outset and then successively merges pairs of clusters until all clusters have been merged into a single cluster that contains all documents.

DENDROGRAM

An HAC clustering is typically visualized as a *dendrogram* as shown in Figure 17.1. A merge of two clusters is represented as a horizontal line that connects the vertical lines of the two clusters (where documents are viewed as singleton clusters). The y-axis represents *combination similarity*, the similarity of the two clusters merged by the horizontal line at a particular y. By moving up from the bottom layer to the top node, we can reconstruct the history of mergers that resulted in the depicted clustering.

COMBINATION SIMILARITY

MONOTONICITY

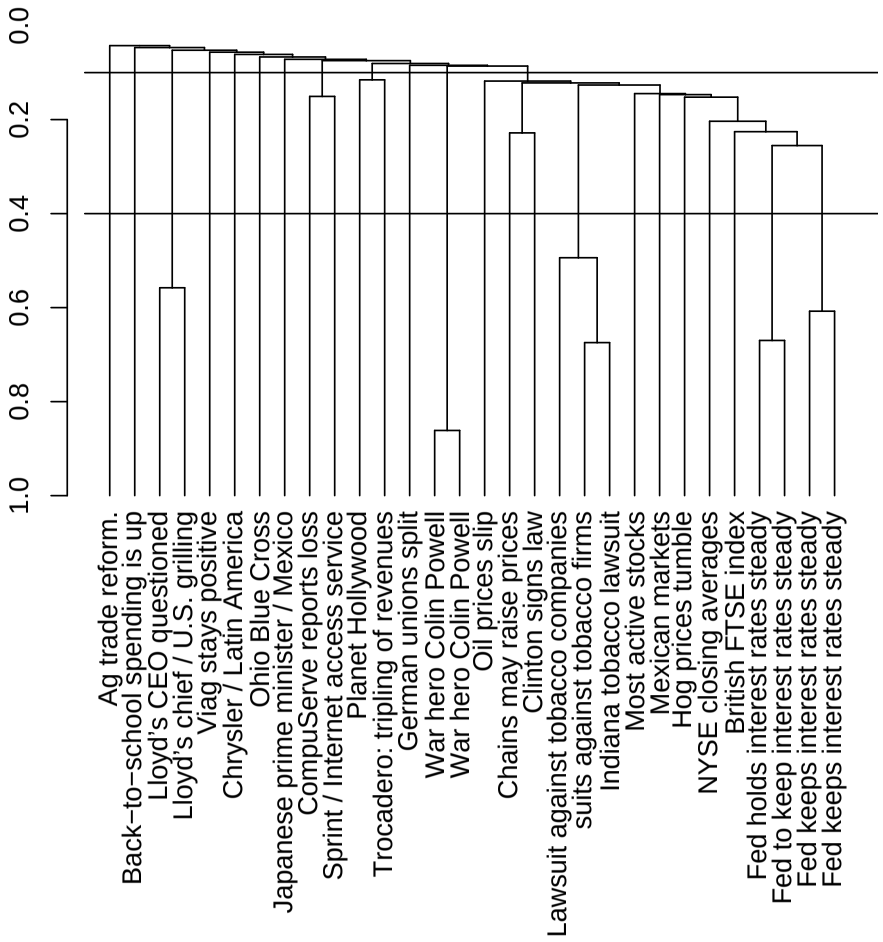
A fundamental assumption in HAC is that the merge operation is *monotonic*. If $s_1, s_2, \ldots, s_{K-1}$ are the successive combination similarities of an HAC clustering, then $s_1 \geq s_2 \geq \ldots \geq s_{K-1}$ must hold. A non-monotonic HAC clustering contains at least one *inversion* $s_i < s_{i+1}$ and contradicts the fundamental assumption that we found the best possible merger at each step. We will see an example of an inversion in Figure 17.11.

INVERSION

Hierarchical clustering does not require a prespecified number of clusters. But in some applications we want a partition of disjoint clusters just as in flat clustering. In those cases, the hierarchy needs to be cut at some point. A number of criteria can be used to determine the cutting point:

- Cut at a prespecified level of similarity. For example, we may want clusters with a minimum combination similarity of 0.4. In this case, we cut the dendrogram at 0.4. In Figure 17.1, cutting the diagram at $y = 0.4$ yields 24 clusters (grouping only documents with high similarity together) and cutting it at $y = 0.1$ yields 12 clusters (one large financial news cluster and 11 smaller clusters).

- Cut the dendrogram where the gap between two successive combination similarities is largest. Such large gaps arguably indicate "natural" clusterings. Adding one more cluster decreases the quality of the clustering significantly, so cutting before this steep decrease occurs can be viewed as optimal. This strategy is analogous to looking for the knee in the k-means graph in Figure 16.7 (page 260).



► **Figure 17.1** A dendrogram of a single-link clustering of 30 documents from Reuters-RCV1. The y-axis represents combination similarity, the similarity of the two component clusters that gave rise to the corresponding merge. For example, the combination similarity of *Lloyd's CEO questioned* and *Lloyd's chief / U.S. grilling* is ≈ 0.56. Two possible cuts of the dendrogram are shown: at 0.4 into 24 clusters and at 0.1 into 12 clusters.

Given: N one-document clusters
Compute similarity matrix
for $k = 1$ to N:
  for $\ell = 1$ to N:
    $C[k][\ell] = \text{sim}(d_k, d_\ell)$
Initialization
A=[] *(for collecting merge sequence)*
for $k = 1$ to N:
  $I[k] = 1$ *(keeps track of active clusters)*
Compute clustering
for $k = 1$ to $N - 1$:
  $(\ell, m) = \arg\max_{(\ell,m),\ell \neq m, I[\ell]=I[m]=1} C[\ell][m]$
  A.append($\langle \ell, m \rangle$)
  for $j = 1$ to N:
    $C[\ell][j] = C[j][\ell] = \text{sim}(j, \ell, m)$
  $I[m] = 0$ *(deactivate cluster)*

► **Figure 17.2** A simple, but inefficient HAC algorithm. In each iteration, the two most similar clusters are merged and the rows and columns of the merge cluster ℓ in C are updated. Ties in this algorithm and in HAC in general are broken randomly. The clustering is stored as a list of mergers in A. I indicates which clusters are still available to be merged. The function $\text{sim}(j, \ell, m)$ computes the similarity of cluster j with the merger of clusters ℓ and m. For some HAC algorithms $\text{sim}(j, \ell, m)$ is simply a function of $C[j][\ell]$ and $C[j][m]$, for example, the maximum of these two values for single-link.

- Evaluate each of the N clusterings with respect to a goodness measure g. Select the cutting point corresponding to the clustering with optimal goodness. We first need a goodness measure g for clusters. An example for g is ($-$ RSS) (Chapter 16 ,page 255). The goodness measure of the clustering is then the sum of (a) the weighted average of $g(\omega_1), \ldots, g(\omega_K)$ and (b) a penalty for the number of clusters. We need the penalty, because the goodness measure reaches its maximum (0.0 for ($-$ RSS)) for N one-document clusters. This strategy is similar to AIC and BIC in Chapter 16 (page 260).

- As in flat clustering, we can also prespecify the number of clusters K and select the cutting point that produces K clusters.

A simple, generic HAC algorithm is shown in Figure 17.2. It consists of $N - 1$ steps of merging the currently most similar clusters. We will now refine this algorithm for the different similarity measures of single-link, complete-link, group-average, and centroid clustering. Single-link clustering merges

method	combination similarity	time compl.	comment
single-link	max sim of any two points	$O(N^2)$	chaining effect
complete-link	min sim of any two points	$O(N^2 \log N)$	sensitive to outliers
centroid	similarity of centroids	$O(N^2 \log N)$	combination similarity not recoverable, inversions possible
group-average	avg sim of any two points	$O(N^2 \log N)$	optimal in most cases

► **Table 17.1** Comparison of HAC algorithms.

the two clusters that have the greatest *maximum* similarity between any two members; complete-link clustering merges the two clusters that have the greatest *minimum* similarity between any two members (Section 17.2). Group-average and centroid clustering average similarities between members and thereby avoid the long straggling clusters in single-link clustering (the chaining effect) and the sensitivity to outliers in complete-link clustering (Sections 17.3 and 17.4).

Table 17.1 summarizes the properties of the four HAC algorithms introduced in this chapter. We recommend GAAC because it is generally the method that produces the highest quality clustering. It does not suffer from chaining or sensitivity to outliers and has cleaner semantics than centroid clustering. The only exception to this recommendation is for non-vector representations. In that case, GAAC is not applicable and clustering should typically be performed with the complete-link method.

17.2 Single-link and complete-link clustering

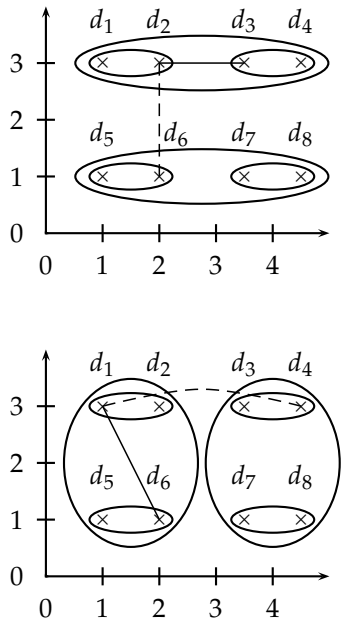
SINGLE-LINK CLUSTERING

In *single-link clustering* or *single-linkage clustering*, the similarity of two clusters is the similarity of their *most similar* members. This single-link merge criterion is *local*. We pay attention solely to the area where the two clusters come closest to each other. Other, more distant parts of the cluster and the clusters' overall structure are not taken into account.

COMPLETE-LINK CLUSTERING

In *complete-link clustering* or *complete-linkage clustering*, the similarity of two clusters is the similarity of their *most dissimilar* members. This is equivalent to choosing the cluster pair whose merger has the smallest diameter. This complete-link merge criterion is non-local: the entire structure of the clustering is taken into account. We prefer compact clusters with small diameters over long, straggly clusters. Complete-link clustering is sensitive to outliers. A single document far from the center can increase diameters of candidate merge clusters dramatically and completely change the final clustering.

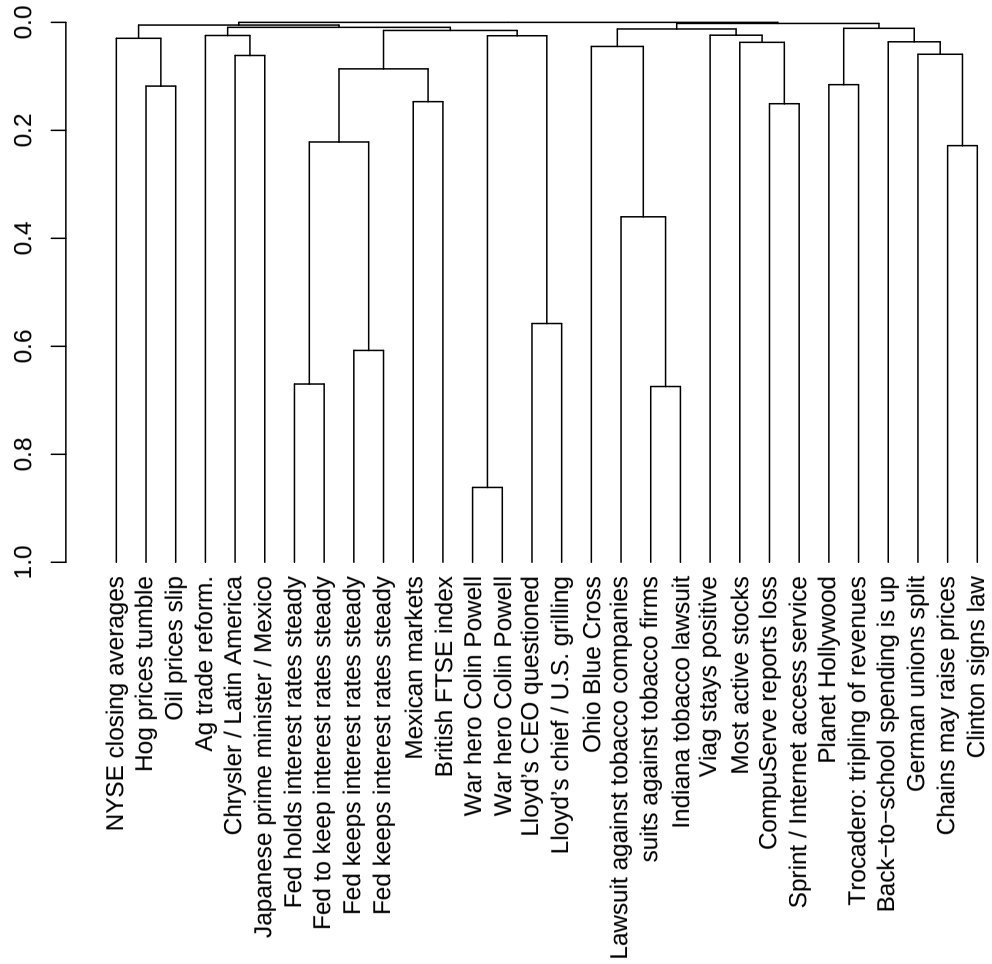
Figure 17.3 depicts a single-link and a complete-link clustering of eight



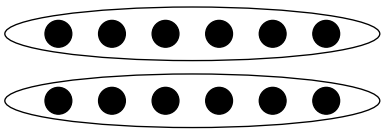
► **Figure 17.3** A single-link (top) and complete-link (bottom) clustering of eight documents. The ellipses correspond to successive clustering stages. Top: The single-link similarity of the two upper two-point clusters is the similarity (proximity) of d_2 and d_3 (solid line), which is greater than the single-link similarity of the two left pairs (dashed line). Bottom: The complete-link similarity of the two upper two-point clusters is the similarity (proximity) of d_1 and d_4 (dashed line), which is smaller than the complete-link similarity of the two left pairs (solid line).

documents. The first four steps, each producing a cluster consisting of a pair of two documents, are identical. Then single-link clustering joins the upper two pairs (and after that the lower two pairs) because on the maximum-similarity definition of cluster similarity, those two clusters are closest. Complete-link clustering joins the left two pairs (and then the right two pairs) because those are the closest pairs according to the minimum-similarity definition of cluster similarity.[1] Figure 17.1 is an example of a single-link clustering of a set of documents while Figure 17.4 is a complete-link clustering of the same

1. If you are bothered by the possibility of ties, assume that d_1 has coordinates $(1+\epsilon, 3-\epsilon)$ and that all other points have integer coordinates.



► **Figure 17.4** A dendrogram of a complete-link clustering of 30 documents from Reuters-RCV1. This complete-link clustering is more balanced than the single-link clustering of the same documents in Figure 17.1. When cutting the last merger, we obtain two clusters of similar size (documents 1–16 and documents 17–30). The y-axis represents combination similarity.



► **Figure 17.5** Chaining in single-link clustering. The local criterion in single-link clustering can cause undesirable elongated clusters.

set. Single-link clustering is used in Section 19.6.1 (page 315), as part of the union-find algorithm, to identify near duplicate pages on the web.

Both single-link and complete-link clustering have graph-theoretic interpretations. Define s_k to be the combination similarity of the two clusters merged in step k and $G(s_k)$ the graph that links all data points with a similarity of at least s_k. Then the clusters after step k in single-link clustering are the *connected components* of $G(s_k)$ and the clusters after step k in complete-link clustering are the maximum *cliques* of $G(s_k)$.[2]

CONNECTED COMPONENT
CLIQUE

These graph-theoretic interpretations motivate the terms single-link and complete-link clustering. Single-link clusters at step k are maximum sets of points that are linked via at least one link of similarity $s \geq s_k$; complete-link clusters at step k are maximum sets of points that are completely linked among each other via links of similarity $s \geq s_k$.

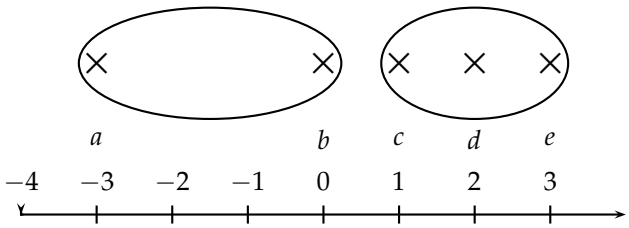
Single-link and complete-link clustering reduce the assessment of cluster quality to a single similarity between a pair of documents: the two most similar documents in single-link clustering, the two most dissimilar documents in complete-link clustering. A measurement based on one pair cannot fully reflect the distribution of documents in a cluster. It is therefore not surprising that both types of clustering are suboptimal. Single-link clustering can produce straggling clusters as shown in Figure 17.5. Since the merge criterion is strictly local, a chain of points can be extended for long distances without regard to the overall shape of the emerging cluster. This effect is called *chaining*.

CHAINING

The chaining effect is also apparent in Figure 17.1. The last 12 mergers of the single-link clustering (those above the $0.1(= 1 - 0.9)$ line) add on single documents or pairs of documents, corresponding to a chain. The complete-link clustering in Figure 17.4 avoids this problem. Documents are split into two groups of roughly equal size. In general, this is a more useful organization of the data than a clustering with chains.

But complete-link clustering has a different problem. It pays too much attention to outliers, points that do not fit well into the global structure of

2. A connected component is a maximum set of points such that there is a path connecting each pair. A clique is a set of points that are completely linked among each other.



► **Figure 17.6** Outliers in complete-link clustering. The four points have the coordinates $-3+2\times\epsilon, 0, 1+2\times\epsilon, 2$ and $3-\epsilon$. Complete-link clustering creates the two clusters shown as ellipses. Intuitively, $\{b, c, d, e\}$ should be one cluster, but it is split by outlier a.

the cluster. In the example in Figure 17.6 the four rightmost points are split because of the outlier at the left edge. Complete-link clustering does not find the intuitively correct cluster structure in this example.

17.2.1 Time complexity

The complexity of the generic HAC algorithm in Figure 17.2 is $O(N^3)$ because we exhaustively search matrix C for the largest similarity (which then gives us the next pair to merge). But in single-link clustering, we can do much better by keeping a next-best-merge array (NBM) as shown in Figure 17.7. NBM keeps track of what the best merge is for each cluster. Each of the three k-loops in Figure 17.7 and thus the overall complexity of single-link clustering is $O(N^2)$.

The time complexity of complete-link clustering is $O(N^2 \log N)$ as shown in Figure 17.8. The rows of the $N \times N$ matrix C are sorted in decreasing order of similarity in the priority queues P. $P[k]$.max() then returns the element of $C[k]$ that currently has the highest similarity with k. After creating the merged cluster of k_1 and k_2, k_1 is used as its representative. The function sim computes the similarity function for potential merger pairs: largest similarity for single-link, smallest similarity for complete-link, average similarity for GAAC (Section 17.3), and centroid similarity for centroid (Section 17.4). We give an example of how a row of C is processed (Figure 17.8, bottom panel). The first k-loop is $O(N^2)$, the second and third are $O(N^2 \log N)$ for an implementation of priority queues that supports deletion and insertion in $O(\log N)$. Overall complexity of the algorithm is therefore $O(N^2 \log N)$. In the definition of the merge functions, $\vec{v}_m$ and $\vec{v}_\ell$ are the vector sums of $\omega_{k_1} \cup \omega_{k_2}$ and ω_ℓ, respectively, and N_m and N_ℓ are the number of documents in $\omega_{k_1} \cup \omega_{k_2}$ and ω_ℓ, respectively.

Given: N one-document clusters
Compute similarity matrix
for $k = 1$ to N:
  for $\ell = 1$ to N:
    $C[k][\ell] = \text{sim}(d_k, d_\ell)$
Initialization
$A = [\,]$
for $k = 1$ to N:
  $I[k] = 1$
  $\text{NBM}[k].\text{index} = \arg\max_{i, i \neq k} C[k][i]$
  $\text{NBM}[k].\text{sim} = C[k][\text{NBM}[k].\text{index}]$
Compute clustering
for $k = 1$ to $N - 1$:
  $k_1 = \arg\max_{k, I[k]=1} \text{NBM}[k].\text{sim}$
  $k_2 = \text{NBM}[k_1].\text{index}$
  $A.\text{append}(\langle k_1, k_2 \rangle)$
  $k_{\text{min}} = \arg\min_{k_1, k_2}(\text{NBM}[k_1].\text{sim}, \text{NBM}[k_2].\text{sim})$
  $I[k_{\text{min}}] = 0$

► **Figure 17.7** Single-link clustering algorithm using an NBM array.

BEST-MERGE PERSISTENCE

What is the reason for the difference in time complexity between single-link and complete-link clustering? Single-link clustering is *best-merge persistent*. Suppose that the best merge cluster for ω_k is ω_j. Then after merging ω_j with a third cluster $\omega_i \neq \omega_k$, the merger of ω_i and ω_j will be ω_k's best merge cluster (Exercise 17.2). As a consequence, the best-merge candidate for the merged cluster is one of the two best-merge candidates of its components in single-link clustering. This means that no update of NBM is necessary after a merger in Figure 17.7.

Figure 17.9 demonstrates that best-merge persistence does not hold for complete-link clustering. After merging ω_k's best merge candidate ω_j with the third cluster ω_i, a completely different cluster ω_ℓ becomes the best merge candidate for ω_k. This is because the complete-link merge criterion is global and can be affected by points at a great distance from the area where two merge candidates meet.

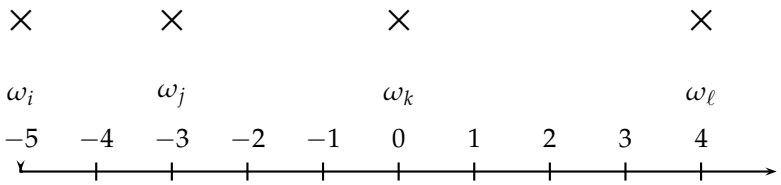
In practice, the performance penalty of the $O(N^2 \log N)$ algorithm is small compared to the $O(N^2)$ single-link algorithm since the computation of one dot product is an order of magnitude slower than the computation of a comparison in sorting. All the HAC algorithms we present are $O(N^2)$ with respect to dot product computations. So the difference in complexity is rarely a concern in practice when choosing one of the HAC algorithms.

Given: N normalized vectors $\vec{v}_i$
Compute matrix C
for $k = 1$ to N:
  for $\ell = 1$ to N:
    $C[k][\ell].\text{sim} = \vec{v}_k \cdot \vec{v}_\ell$
    $C[k][\ell].\text{index} = \ell$
for $k = 1$ to N:
  $P[k] :=$ priority queue for $C[k]$ sorted on sim
  Delete $C[k][k]$ from $P[k]$ *(don't want self-similarities)*
Initialization
$A = [\,]$
for $k = 1$ to N:
  $I[k] = 1$
Compute clustering
for $k = 1$ to $N - 1$:
  $k_1 = \arg\max_{k, I[k]=1} P[k].\max()$
  $k_2 = P[k_1].\max().\text{index}$
  $A.\text{append}(\langle k_1, k_2 \rangle)$
  $I[k_2] = 0$
  $P[k_1] = \emptyset$
  for all ℓ with $I[\ell] = 1, \ell \neq k_1$:
    $C[k_1][\ell].\text{sim} = C[\ell][k_1].\text{sim} = \text{sim}(\ell, k_1, k_2)$
    Delete $C[\ell][k_1]$ and $C[\ell][k_2]$ from $P[\ell]$
    Insert $C[\ell][k_1]$ in $P[\ell]$, $C[k_1][\ell]$ in $P[k_1]$

clustering algorithm	$\text{sim}(\ell, k_1, k_2)$
single-link	$\max(\text{sim}(\ell, k_1), \text{sim}(\ell, k_2))$
complete-link	$\min(\text{sim}(\ell, k_1), \text{sim}(\ell, k_2))$
centroid	$(\frac{1}{N_m}\vec{v}_m) \cdot (\frac{1}{N_\ell}\vec{v}_\ell)$
group-average	$\frac{1}{(N_m+N_\ell)(N_m+N_\ell-1)}[(\vec{v}_m + \vec{v}_\ell)^2 - (N_m + N_\ell)]$

compute $C[5]$	1 0.2	2 0.8	3 0.6	4 0.4
create $P[5]$ (by sorting)	2 0.8	3 0.6	4 0.4	1 0.2
merge 2 and 3, update similarity of 2, delete 3	2 0.3	4 0.4	1 0.2	
reinsert 2	4 0.4	2 0.3	1 0.2	

► **Figure 17.8** A generic HAC algorithm. Top: The algorithm. Center: Four different similarity measures. Bottom: An example of a sequence of processing steps for a priority queue. This is a made up example showing $P[5]$ for a 5×5 matrix C.



► **Figure 17.9** Complete-link clustering is not best-merge persistent. At first, ω_j is the best-merge cluster for ω_k. But after merging ω_i and ω_j, ω_ℓ becomes ω_k's best-merge candidate. In a best-merge persistent algorithm like single-link, the best-merge cluster would be $\omega_i \cup \omega_j$.

17.3 Group-average agglomerative clustering

GROUP-AVERAGE AGGLOMERATIVE CLUSTERING
GAAC

For clustering in a vector space, there is a clustering method that evaluates cluster quality based on *all* similarities between documents, thus avoiding the pitfalls of the single-link and complete-link criteria: *group-average agglomerative clustering* or *GAAC*. It is also called group-average clustering and average-link clustering. GAAC computes the average similarity sim-ga of all pairs, including pairs of documents from the same cluster. Self-similarities are not included in the average:

$$\text{sim-ga}(\omega_i, \omega_j) = \frac{1}{(N_i + N_j)(N_i + N_j - 1)} \sum_{d_k \in \omega_i \cup \omega_j} \sum_{d_\ell \in \omega_i \cup \omega_j, d_\ell \neq d_k} \vec{d}_k \cdot \vec{d}_\ell \tag{17.1}$$

where $\vec{d}$ is the normalized vector of document d, $\cdot$ denotes the scalar or dot product, and N_i and N_j are the number of documents in ω_i and ω_j, respectively.

The measure sim-ga can be computed efficiently because the sum of individual vector similarities is equal to the similarities of their sums:

$$\sum_{d_k \in \omega_i} \sum_{d_\ell \in \omega_j} (\vec{d}_k \cdot \vec{d}_\ell) = (\sum_{d_k \in \omega_i} \vec{d}_k) \cdot (\sum_{d_\ell \in \omega_j} \vec{d}_\ell) \tag{17.2}$$

With (17.2), we have:

$$\text{sim-ga}(\omega_i, \omega_j) = \frac{1}{(N_i + N_j)(N_i + N_j - 1)} [(\sum_{d_k \in \omega_i \cup \omega_j} \vec{d}_k)^2 - (N_i + N_j)] \tag{17.3}$$

The term $(N_i + N_j)$ is the sum of $N_i + N_j$ self-similarities of value 1.0. With this trick we can compute cluster similarity in constant time (assuming we

have available the two vector sums $\sum \vec{d}$) instead of in $O(N_i N_j)$. Note that for two single-document clusters, 17.3 is equivalent to the dot product.

Equation (17.2) relies on the distributivity of the scalar product with respect to vector addition. Since this is crucial for the efficient computation of a GAAC clustering, the method cannot be easily applied to non-vector representations of documents.

The merge algorithm for GAAC is the same as Figure 17.8 for complete-link except that we use as the similarity function 17.3. So the overall time complexity of GAAC is the same as for complete-link clustering: $O(N^2 \log N)$. Like complete-link clustering, GAAC is not best-merge persistent (Exercise 17.2). This means that there is no $O(N^2)$ algorithm for GAAC that would be analogous to the $O(N^2)$ algorithm for single-link.

We can also define group-average similarity as including self-similarities:

$$\text{(17.4)} \quad \text{sim-ga}'(\omega_i, \omega_j) = \frac{1}{(N_i+N_j)^2}\Big(\sum_{d_k \in \omega_i \cup \omega_j} \vec{d}_k\Big)^2 = \frac{1}{N_i+N_j} \sum_{d_k \in \omega_i \cup \omega_j} \vec{d}_k \cdot \vec{\mu}(\omega_i \cup \omega_j)$$

where the centroid $\mu(\omega)$ is defined as in Equation (14.1) (page 216). This definition is equivalent to the intuitive definition of cluster quality as average similarity of documents $\vec{d}_k$ to the cluster's centroid $\vec{\mu}$.

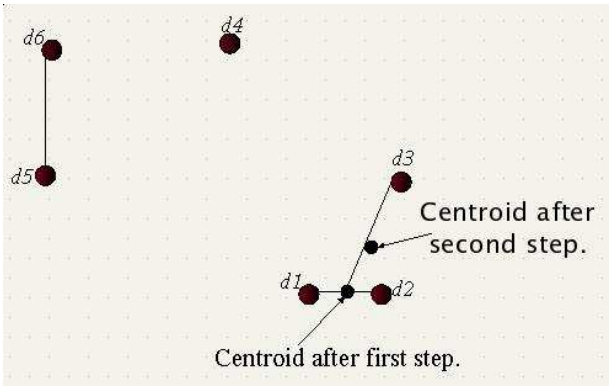
Self-similarities are always equal to 1.0, the maximum possible value for normalized vectors. The proportion of self-similarities in Equation (17.4) is $i/i^2 = 1/i$ for a cluster of size i. This gives an unfair advantage to small clusters since they will have proportionally more self-similarities. For two documents with a similarity s, this means that merging them generates a cluster with a similarity measure of $(1+s)/2$. The sim-ga similarity of the combined clusters according to Equation (17.3) is $s \leq (1+s)/2$, which is the same as in single-link, complete-link and centroid clustering. For these reasons, we prefer the definition in Equation (17.3), which excludes self-similarities from the average.

17.4 Centroid clustering

In centroid clustering, the similarity of two clusters is defined as the similarity of their centroids:

$$\text{(17.5)} \quad \text{sim-cent}(\omega_i, \omega_j) = \Big(\frac{1}{N_i}\sum_{d_k \in \omega_i} \vec{d}_k\Big) \cdot \Big(\frac{1}{N_j}\sum_{d_\ell \in \omega_j} \vec{d}_\ell\Big) = \frac{1}{N_i N_j} \sum_{d_k \in \omega_i} \sum_{d_\ell \in \omega_j} \vec{d}_k \cdot \vec{d}_\ell$$

The first part of the equation is centroid similarity, the second part shows that this is equivalent to average similarity of all pairs of documents from *different* clusters. Thus, the difference between GAAC and centroid clustering is



► **Figure 17.10** Centroid clustering. Each step merges the two clusters whose centroids are closest as measured by the dot product.

that GAAC considers all pairs of documents in computing average pairwise similarity whereas centroid clustering excludes pairs from the same cluster.

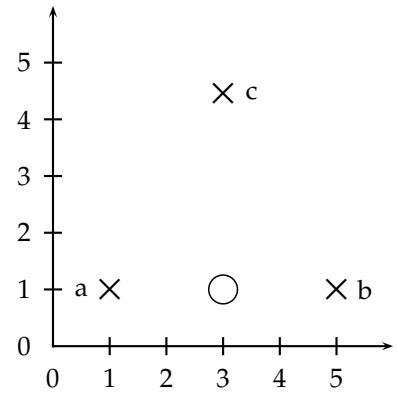
Figure 17.10 shows two iterations of centroid clustering.

Like GAAC, centroid clustering is not best-merge persistent and therefore $O(N^2 \log N)$ (Exercise 17.2). But centroid clustering has one unique property that makes it inferior to the other three HAC algorithms. A good HAC algorithm creates the optimal cluster ω in each step for some criterion of optimality. For single-link, complete-link and group-average clustering, these optimality criteria directly correspond to the combination similarity of the two components that were merged into ω: maximum distance of any point to its nearest neighbor in ω (single-link), maximum distance of any two points in ω (complete-link), and average of all pairwise similarities in ω (GAAC). So there is equivalence for these three methods between the combination similarity computed on the two components and an optimality criterion computed directly on ω (Exercise 17.3). But there is no such optimality score computable directly on the merged cluster ω for centroid clustering: each split of ω will in general produce a different similarity value of the centroids of the two parts. This is worrying because it means that centroid clustering makes no guarantees that the clusters it creates have good properties.

INVERSION

Centroid clustering does not have an optimality criterion because it is not monotonic. So-called *inversions* can occur: Similarity can increase during clustering as in the example in Figure 17.11. This means that we cannot be sure that after merging two clusters with similarity s in centroid clustering, we have found all clusters whose "quality" is better than s.

Despite this problem, centroid clustering is often used because its similarity measure – the similarity of two centroids – is conceptually simpler



► **Figure 17.11** Centroid clustering is not monotonic. The points a at $(1+\epsilon, 1)$, b at $(5, 1)$, and c at $(3, 1+2\sqrt{3})$ are almost equidistant, with a and b closer to each other than to c. In the first merger, the proximity of a and b is ≈ 4. In the second merger, the proximity of the centroid of a and b (the circle) and c is $\approx \cos(\pi/6) \times 4 = \sqrt{3}/2 \times 4 \approx 3.46 < 4$. This is an example of an inversion: similarity *increases* in this sequence of two clustering steps. In a monotonic HAC algorithm, similarity is monotonically *decreasing*.

than the average of all pairwise similarities in GAAC. Figure 17.10 is all one needs to understand centroid clustering. There is no equally simple graph that would explain how GAAC works.

17.5 Cluster labeling

In many applications of clustering, particularly in analysis tasks and in user interfaces (see applications in Table 16.1, page 248), human users interact with clusters. In such settings, we must label clusters, so that users can see what a cluster is about. In Scatter-Gather (Figure 16.2, page 250), each cluster is represented by titles of typical documents – those closest to the cluster centroid – and by a list of words with high weights in the centroid of the cluster. Titles are natural cluster labels because they are written by authors as one-line summaries that can be quickly scanned. Highly weighted words (or, even better, phrases, especially noun phrases) are often more representative of the cluster as a whole than a few titles can be. But a list of phrases takes more time to digest for users than a well crafted title. On the web, anchor text can play a role similar to a title since the anchor text pointing to a page can serve as a concise summary of its contents.

Consider the representation of the Germany cluster in one Scatter-Gather experiment:

	# docs	labeling method		
		centroid	mutual information	title
4	622	oil plant mexico production crude **power 000 refinery gas** bpd	plant oil production **barrels** crude bpd mexico **dolly capacity petroleum**	MEXICO: Hurricane Dolly heads for Mexico coast
9	1017	police security **russian** people military peace killed told **grozny court**	police killed military security peace told **troops forces rebels** people	RUSSIA: Russia's Lebed meets rebel chief in Chechnya
10	1259	00 000 tonnes traders futures wheat prices **cents september** tonne	**delivery** traders futures tonne tonnes **desk** wheat prices 000 00	USA: Export Business - Grain/oilseeds complex

► **Table 17.2** Automatically computed cluster labels for a k-means clustering ($K = 10$) of the first 10,000 documents in Reuters-RCV. Only three of the ten clusters (4, 9, and 10) are shown. The three last columns show cluster summaries computed by three labeling methods: most highly weigted words in centroid (centroid), mutual information, and the title of the document closest to the centroid of the cluster (title). Words selected by only one of the first two methods are in bold.

```
IN PUSH FOR UNIFICATION ...; LEADERS OF TWO GERMANYS ...
germany, east, west, year, soviet, unification
```

CLUSTER-INTERNAL LABELING

The two labeling methods applied here (titles of typical documents and words prominent in the centroid) are *cluster-internal*. Another cluster-internal method simply selects the most frequent words in the cluster. Cluster-internal methods are efficient, but they fail to distinguish words that are frequent in the collection as a whole from those that are frequent only in the cluster. The word year has a high frequency both in the cluster and in the collection and is therefore not helpful in understanding the contents of the Germany cluster.

DIFFERENTIAL CLUSTER LABELING

Differential cluster labeling selects cluster labels by comparing the distribution of words in one cluster with that of other clusters. The feature selection methods we introduced in Section 13.5 (page 200) can all be used for differential cluster labeling. In fact, selecting the most frequent words is also a feature selection technique we discussed in Section 13.5, albeit a non-differential one. In particular, mutual information (MI) (Section 13.5.1, page 200) or, equivalently, information gain and the χ^2-test (Section 13.5.2, page 202) will identify cluster labels that characterize one cluster in contrast to other clusters. A combination of a differential test with a penalty for rare words often gives the best labeling results because rare words are not necessarily representative of the cluster as a whole.

We apply three labeling methods to a k-means clustering in Table 17.2. In this example, there was almost no difference between MI and χ^2. We therefore omit the latter. The centroid method selects a few more uninformative words (000, court, cents, september) than MI (forces, desk), but most of the words selected by either method are good descriptors. We get a good sense of the documents in a cluster from scanning the selected words.

The title of the document closest to the centroid provides a complementary way of characterization. It is easier to read than a list of words. A full title can also contain important context that didn't make it into the top 10 terms selected by MI. The title for cluster 9 suggests that many of its documents are about the Chechnya conflict, a fact the MI words do not reveal. But a single document cannot be representative of all documents in a cluster as in the case of cluster 4, where the selected title is misleading. The main topic of the cluster is oil. Articles about hurricane Dolly only ended up in this cluster because of its effect on oil prices.

Additional complications in labeling clusters arise when we want to label a cluster hierarchy instead of a flat clustering. Not only do we need to distinguish an internal node from its siblings, but also from its parent and its children. Documents in child nodes are by definition also members of their parent node, so a naive differential method cannot be used to find labels that distinguish the parent from its children. However, more complex criteria, based on a combination of overall collection frequency and prevalence in a given cluster, can determine whether a term is a more informative label for a child node or a parent node.

17.6 Variants

So far we've only looked at agglomerative clustering, but a cluster hierarchy can also be generated top-down. This variant of hierarchical clustering is called *top-down clustering* or *divisive clustering*. We start at the top with all documents in one cluster. The cluster is split using a flat clustering algorithm. This procedure is applied recursively until the desired number of clusters has been computed or some other stopping criterion is met.

TOP-DOWN CLUSTERING
DIVISIVE CLUSTERING

Top-down clustering is conceptually more complex than bottom-up clustering since we need a second, flat clustering algorithm as a "subroutine". It has the advantage of being more efficient if we don't generate a hierarchy all the way down to individual document leaves. For a fixed number of top levels, using an efficient flat algorithm like k-means, top-down algorithms are linear in the number of documents and clusters. So they run much faster than $O(N^2)$ or $O(N^2 \log N)$ bottom-up algorithms.

There is evidence that divisive algorithms produce more accurate hierarchies than bottom-up algorithms in some circumstances (see reference to

bisecting k-means in Section 17.8). Bottom-up methods make clustering decisions based on local patterns without initially taking into account the global distribution. These early decisions cannot be undone. Top-down clustering benefits from complete information about the global distribution when making top-level partitioning decisions.

Another combination of flat and hierarchical clustering uses hierarchical clustering to create good seeds for k-means. K-means requires a set of seeds as initialization (Figure 16.4, page 256). If these seeds are badly chosen, then the resulting clustering will be of poor quality. If the HAC algorithm is applied to a subset of size $O(\sqrt{N})$ to generate K seeds for k-means, then the overall run time of k-means cum HAC seed generation is $O(N)$. This is because the application of a quadratic algorithm to a sample of size $O(\sqrt{N})$ has an overall complexity of $O(N)$. An appropriate adjustment can be made for an $O(N^2 \log N)$ algorithm to guarantee linearity. This algorithm is referred to as the *Buckshot algorithm*.

BUCKSHOT ALGORITHM

17.7 Implementation notes

Most problems that require the computation of a large number of dot products benefit from an inverted index. This is also the case for HAC clustering. Computational savings due to the inverted index are large if there are many zero similarities – either because many documents don't share any words or because an agressive stop list is used.

In low dimensions, more aggressive optimizations are possible that make the computation of most pairwise similarities unnecessary (Exercise 17.7). Because of the curse of dimensionality, no more efficient algorithms are known in higher dimensions. We confronted the same problem in kNN classification (see Section 14.4, page 230).

When computing a GAAC of a large document set in high dimensions, we have to take care to avoid dense centroids. For dense centroids, clustering can take time $O(|V|N^2 \log N)$ where $|V|$ is the size of the vocabulary whereas complete-link clustering is $O(M_d N^2 \log N)$ where M_d is the average vocabulary of a document. So for large vocabularies complete-link can be more efficient than an unoptimized implementation of GAAC. We discussed this problem in the context of k-means clustering in Chapter 16 (page 259) and suggested two solutions: truncating centroids (keeping only highly weighted terms) and representing clusters by means of sparse medoids instead of dense centroids. This optimization can also be applied to GAAC and centroid clustering.

17.8 References and further reading

An excellent general review of clustering is (Jain et al. 1999). Early references are (King 1967) (single-link), (Sneath and Sokal 1973) (complete-link, GAAC) and (Lance and Williams 1967) (discussing a large variety of hierarchical clustering algorithms). A reference for minimum spanning tree algorithms (as an alternative way of computing a single-link hierarchy, Exercise 17.1) is (Cormen et al. 1990).

It is often claimed that hierarchical clustering algorithms produce better clusterings than flat algorithms (Jain and Dubes 1988, p. 140), (Cutting et al. 1992, Larsen and Aone 1999) although more recently there have been experimental results suggesting the opposite (Zhao and Karypis 2002). Even without a consensus on average behavior, there is no doubt that results of EM and k-means are highly variable since they will occasionally converge to a local optimum of poor quality. Assuming ties are handled appropriately, the hierarchical algorithms we have presented here are deterministic and thus more reliable.

The complexity of complete-link, group-average and centroid clustering is sometimes given as $O(N^2)$ (Day and Edelsbrunner 1984, Voorhees 1985b) or even as less than quadratic (Murtagh 1983) because a document similarity computation is an order of magnitude more expensive than a simple comparison, the main operation executed in the merging steps after the $N \times N$ similarity matrix has been computed.

The centroid algorithm described here is due to Voorhees (1985b). Voorhees recommends complete-link and centroid clustering over single-link for a retrieval application. The Buckshot algorithm was originally published by Cutting et al. (1993).

WARD'S METHOD
MINIMUM VARIANCE CLUSTERING

An important HAC technique not discussed here is *Ward's method* (Ward 1963, El-Hamdouchi and Willett 1986), also called *minimum variance clustering*. In each step, it selects the merger with the smallest RSS (the sum of all squared distances of each point from its centroid), as defined in Chapter 16 (page 255). The merge criterion in Ward's method (a function of all individual distances from the centroid) is closely related to the merge criterion in GAAC (a function of all individusal similarities to the centroid).

Despite its importance for making the results of clustering useful, comparatively little work has been done on labeling clusters. Popescul and Ungar (2000) obtain good results from

a combination of χ^2 and term frequency. Glover et al. (2002b) use information gain for labeling clusters of web pages. The more complex problem of labeling nodes in a hierarchy (which requires distinguishing more general labels for parents from more specific labels for children) is tackled by Glover et al. (2002a) and Treeratpituk and Callan (2006). Some clustering algorithms attempt to find a set of labels first and then build (often overlapping) clusters

around the labels, thereby avoiding the problem of labeling altogether (Zamir and Etzioni 1999, Käki 2005). We know of no comprehensive study that evaluates the quality of such "label-based" clusters compared to the clustering algorithms discussed here and in Chapter 16. A problem related to cluster labeling occurs in marking areas of a self-organizing map (or Kohonen map) with typical words or phrases (Azcarraga and Yap Jr. 2001). In principle, work on multi-document summarization (McKeown and Radev 1995) is also applicable to cluster labeling, but multi-document summaries are usually longer than the short text needed when labeling clusters.

The bisecting k-means algorithm is due to Steinbach et al. (2000).

Although this introduction to hierarchical clustering focuses on document clustering, hierarchical clustering can be readily applied to clustering other objects in information retrieval. For example, terms can be clustered in document space, where documents are axes and terms are represented as vectors in document space (Qiu and Frei 1993).

Unlike k-means and EM, most hierarchical clustering algorithms do not have a probabilistic interpretation. Model-based hierarchical clustering (Vaithyanathan and Dom 2000) is an exception.

The evaluation methodology described in Section 16.3 (page 252) is also applicable to hierarchical clustering. Specialized evaluation measures for hierarchies are discussed by Larsen and Aone (1999) and Sahoo et al. (2006).

The R project environment (R Development Core Team 2005) offers good support for hierarchical clustering. The R function `hclust` implements single-link (`single`), complete-link (`complete`), group-average (`average`), and centroid (`centroid`) clustering; and Ward's method (`ward`). Another option provided is `median` clustering which represents each cluster by its medoid (cf. k-medoids in Chapter 16, page 259).

17.9 Exercises

Exercise 17.1

MINIMUM SPANNING TREE

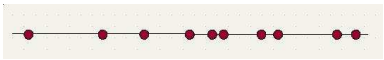
A single-link clustering can also be computed from the *minimum spanning tree* of a graph. The minimum spanning tree connects the vertices of a graph at the smallest possible cost, where cost is defined as the sum over all edges of the graph. In our case the cost of an edge is the distance between two documents. Show that if $\Delta_{k-1} > \Delta_k > \ldots > \Delta_1$ are the lengths of the edges of a minimum spanning tree, then these edges correspond to the $k-1$ merges in constructing a single-link clustering.

Exercise 17.2

Show that single-link clustering is best-merge persistent and that GAAC and centroid clustering are not best-merge persistent.

Exercise 17.3

Given a cluster ω that was produced in a) single-link, b) complete-link, c) group-average clustering, how can one compute the combination similarity of ω? (One way



► **Figure 17.12** Single-link clustering of points on a line. In low dimensions, single-link clustering can be implemented more efficiently than the general algorithm in Figure 17.2.

to do this is to restart the clustering and keep track of the combination similarities. The intent of the exercise is to read the combination similarity directly off of the cluster.)

Exercise 17.4

Apply group-average clustering to the points in Figures 17.5 and 17.6. Map them onto the surface of the unit sphere in a three-dimensional space to get normalized vectors. Is the group-average clustering different from the single-link and complete-link clusterings?

Exercise 17.5

a. Consider running 2-means clustering on a collection with documents from two different languages. What result would you expect?

b. Would you expect the same result when running an HAC algorithm?

Exercise 17.6

Suppose a run of HAC finds the clustering with $K = 7$ to have the highest value on some pre-chosen goodness measure of clustering. Have we found the highest-value clustering amongst all clusterings with $K = 7$?

Exercise 17.7

Show that for a single-link clustering of points on a line (Figure 17.12) only a linear number of similarities has to be computed. What is the overall complexity of single-link clustering of points on a line?

Exercise 17.8

Prove that single-link, complete-link, and group-average are monotonic in the sense defined on page 270.

Exercise 17.9

For N points, there are $K^N/K!$ different flat clusterings into K clusters. How many different hierarchical clusterings (or dendrograms) are there for N points? Which number is greater?

Exercise 17.10

For a set of N documents there are up to N^2 distinct similarities between clusters in single-link and complete-link clustering. How many distinct cluster similarities are there in GAAC and centroid clustering?

18 *Dimensionality reduction and Latent Semantic Indexing*

LOW-RANK APPROXIMATION

In Chapter 7 we introduced the notion of a *term-document matrix:* an $m \times n$ matrix M, each of whose rows represents a term and each of whose columns represents a document in the corpus. In this chapter we apply certain operations from linear algebra to this matrix, to compute so-called *low-rank approximations* to M. We then examine the application of such low-rank approximations to indexing and retrieving documents, a technique referred to as *latent semantic indexing*. Readers who do not require a refresher on linear algebra may skip Section 18.1 below.

18.1 Linear algebra review

RANK

We now briefly review some necessary background in linear algebra. Let M be an $m \times n$ matrix with real-valued entries; for term-document matrices, all entries are in fact non-negative. The *rank* of a matrix is the number of linearly independent rows (or columns) in it; thus, $rank(M) \leq \min\{m, n\}$. A square $r \times r$ matrix all of whose off-diagonal entries are zero is called a *diagonal matrix*; its rank is equal to the number of non-zero diagonal entries. If all the diagonal entries of such a diagonal matrix are 1, it is called the identity matrix of dimension r and represented by I_r.

Example 18.1: The 3×3 diagonal matrix below has rank 2, since the third row is the sum of (and thus linearly dependent on) the first two:

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 2 & 1 \end{pmatrix} \tag{18.1}$$

Next, for a square $m \times m$ matrix M, consider the equation

$$M\vec{x} = \lambda\vec{x}. \tag{18.2}$$

EIGENVALUES The values of λ satisfying Equation 18.2 are called the *eigenvalues*of M. The n-vector $\vec{x}$ satisfying (18.2) for an eigenvalue λ is the corresponding *right eigenvector*; thus eigenvalues and eigenvectors occur in pairs. The eigenvector corresponding to the largest-magnitude eigenvalue is called the *principal eigenvector.* In a similar fashion, the *left eigenvectors* of M are the m-vectors y such that

$$\vec{y}M = \lambda\vec{y}. \tag{18.3}$$

The number of non-zero eigenvalues of M is rank(M).

Example 18.2: Consider the 2×2 matrix

$$M = \begin{pmatrix} 6 & -2 \\ 4 & 0 \end{pmatrix} \tag{18.4}$$

Using this matrix in (18.2), we find that $\lambda = 2$ is an eigenvalue:

$$\begin{pmatrix} 6 & -2 \\ 4 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} = 2 \begin{pmatrix} 1 \\ 2 \end{pmatrix}. \tag{18.5}$$

The eigenvalues of a matrix are found by solving the *characteristic equation*, which is obtained by rewriting (18.2) in the form $(M - \lambda I_m)\vec{x} = 0$. This matrix equation has non-zero solutions if and only if $|(M - \lambda I_m)| = 0$, where $|S|$ denotes the determinant of a square matrix S. The equation $|(M - \lambda I_m)| = 0$ is an mth order polynomial equation in λ and can have at most m roots, which are the eigenvalues of M. These eigenvalues can in general be complex, even if all entries of M are real.

We now examine some further properties of eigenvalues and eigenvectors, to set up the central idea of singular value decompositions in Section 18.2 below. First, we look at the relationship between matrix-vector multiplication and eigenvalues. Consider the matrix

$$S = \begin{pmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{pmatrix}. \tag{18.6}$$

Clearly the matrix has rank 2, and therefore has 2 non-zero eigenvalues. It is easy to verify that the eigenvalues are $\lambda_1 = 3$, $\lambda_2 = 2$ and $\lambda_3 = 0$, with corresponding eigenvectors

$$\vec{v_1} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \vec{v_2} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \text{ and } \vec{v_3} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}. \tag{18.7}$$

Thus, for each of the eigenvectors, multiplication by S acts as if we were multiplying the eigenvector by a multiple of the identity matrix; the multiple

is different for each eigenvector. Now, consider an arbitrary vector, such as $\vec{x} = \begin{pmatrix} 2 \\ 4 \\ 6 \end{pmatrix}$. We may express $\vec{x}$ as a superposition of the three eigenvectors of S:

$$\vec{x} = \begin{pmatrix} 2 \\ 4 \\ 6 \end{pmatrix} = 2\vec{v_1} + 4\vec{v_2} + 6\vec{v_3}. \tag{18.8}$$

We now consider the effect of multiplying the arbitrary vector x by S:

$$\begin{aligned} S\vec{x} &= S(2\vec{v_1} + 4\vec{v_2} + 6\vec{v_3}) \\ &= 2S\vec{v_1} + 4S\vec{v_2} + 6S\vec{v_3} \\ &= 2\lambda_1\vec{v_1} + 4\lambda_2\vec{v_2} + 6\lambda_3\vec{v_3}. \end{aligned} \tag{18.9}$$

Thus, even though $\vec{x}$ is an arbitrary vector, the effect of multiplication by S is determined by the eigenvalues and eigenvectors of S. Furthermore, it is intuitively apparent from (18.9) that the product $S\vec{x}$ is relatively unaffected by terms arising from the small eigenvalues of S; in our example, since $\lambda_3 = 0$, the third term on the right hand side of (18.9) vanishes. This suggests that the effect of small eigenvalues (and their eigenvectors) on the matrix-vector product is small. We will carry forward this intuition when studying matrix decompositions in Section 18.2 below. Before doing so, we examine the eigenvectors and eigenvalues of special forms of matrices that will be of particular interest to us.

For a *symmetric* matrix S, the eigenvectors corresponding to distinct eigenvalues are *orthogonal*. Further, if S is both real and symmetric, the eigenvalues are all real. Finally, we recall that S is said to be *positive semidefinite* if for any real-valued column vector $\vec{w}$, we have $\vec{w}^T S\vec{w} \geq 0$. Then, we note that all the eigenvalues of S are non-negative.

Example 18.3: Consider the real, symmetric matrix

$$S = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}. \tag{18.10}$$

From the characteristic equation $|S - \lambda I| = 0$, we have the quadratic $(2 - \lambda)^2 - 1 = 0$, whose solutions yield the eigenvalues 3 and 1. The corresponding eigenvectors $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$ and $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ are orthogonal.

18.1.1 Matrix decompositions

In this section we examine ways in which a square matrix can be *factored* into the product of matrices derived from its eigenvectors. This will form the basis of our principal text-analysis technique in Section 18.3.

EIGEN DECOMPOSITION

Theorem 18.1 (Matrix diagonalization theorem) *Let S be a square real-valued $m \times m$ matrix with m linearly independent eigenvectors. Then there exists an* eigen decomposition

$$S = U\Lambda U^{-1}, \tag{18.11}$$

where the columns of U are the eigenvectors of S and Λ is a diagonal matrix whose diagonal entries are the eigenvalues of S in decreasing order

$$\begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \dots & \\ & & & \lambda_m \end{pmatrix}, \lambda_i \geq \lambda_{i+1}. \tag{18.12}$$

If the eigenvalues are distinct, then this decomposition is unique.

To understand how Theorem 18.1 works, we note that U has the eigenvectors of S as columns

$$U = (\vec{v_1}\ \vec{v_2} \cdots \vec{v_m}). \tag{18.13}$$

Then we have

$$\begin{aligned} SU &= S(\vec{v_1}\ \vec{v_2} \cdots \vec{v_m}) && (18.14)\\ &= (\lambda_1\vec{v_1}\ \lambda_2\vec{v_2} \cdots \lambda_m\vec{v_m}) && (18.15)\\ &= (\vec{v_1}\ \vec{v_2} \cdots \vec{v_m}) \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \dots & \\ & & & \lambda_m \end{pmatrix}. && (18.16) \end{aligned}$$

Thus, we have $SU = U\Lambda$, or $S = U\Lambda U^{-1}$.

Exercise 18.1
Compute the unique eigen decomposition of the 2×2 matrix in (18.10).

We consider an extension of the eigen decomposition above, for symmetric square matrices.

SYMMETRIC EIGEN DECOMPOSITION

Theorem 18.2 (Symmetric diagonalization theorem) *Let S be a square, symmetric real-valued $m \times m$ matrix with m linearly independent eigenvectors. Then there exists a* symmetric eigen decomposition

$$S = Q\Lambda Q^T, \tag{18.17}$$

where the columns of Q are the orthogonal and normalized (unit length) eigenvectors of S. Further, all entries of Q are real and we have $Q^{-1} = Q^T$.

We will build on this symmetric eigen decomposition to build low-rank approximations to term-document matrices.

18.2 Term-document matrices and singular value decompositions

The decompositions we have been studying thus far apply to square matrices. However, the matrix we are interested in is the $m \times n$ term-document matrix where (barring a rare coincidence) $m \neq n$. To this end we first describe an extension of the symmetric eigen decomposition known as the *singular value decomposition*. We then show in Section 18.3 how this can be used to construct an approximate version of M.

Theorem 18.3 *Let r be the rank of the $m \times n$ matrix M. Then, there is a* singular-value decomposition *(SVD for short) of M of the form*

$$M = U\Sigma V^T, \tag{18.18}$$

where

1. *U is an $m \times m$ matrix whose columns are the orthogonal eigenvectors of MM^T;*
2. *V is an $n \times n$ matrix whose columns are the orthogonal eigenvectors of M^TM;*
3. *The eigenvalues $\lambda_1, \ldots, \lambda_r$ of MM^T are the same as the eigenvalues of M^TM;*
4. *For $1 \leq i \leq r$, let $\sigma_i = \sqrt{\lambda_i}$, with $\lambda_i \geq \lambda_{i+1}$. Then the $m \times n$ matrix Σ is composed by setting $\Sigma_{ii} = \sigma_i$ for $1 \leq i \leq r$, and zero otherwise.*

The values σ_i are referred to as the *singular values* of M.

FIGURE

The following example illustrates the singular-value decomposition of a 3×2 matrix of rank 2; the singular values are $\Sigma_{11} = \sqrt{3}$ and $\Sigma_{22} = 1$.

$$\begin{pmatrix} 1 & -1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 2/\sqrt{6} & 0 & 1/\sqrt{3} \\ -1/\sqrt{6} & 1/\sqrt{2} & 1/\sqrt{3} \\ 1/\sqrt{6} & 1/\sqrt{2} & -1/\sqrt{3} \end{pmatrix} \begin{pmatrix} \sqrt{3} & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1/\sqrt{2} & -1/\sqrt{2} \\ 1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix}. \tag{18.19}$$

As with the other matrix decompositions defined above, the singular value decomposition of a matrix can be computed by a variety of algorithms (many of which have been publicly available software implementations); pointers to these are given in Section 18.4 at the end of this chapter.

18.3 Low rank approximations and latent semantic indexing

We next state a matrix approximation problem that at first seems to have little to do with information retrieval. Given an $m \times n$ matrix M and a positive integer k, we wish to find an $m \times n$ matrix M_k of rank $\leq k$, so as to minimize the *Frobenius norm* of the matrix difference $X = M - M_k$, defined to be

$$||X||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} X_{ij}^2}. \tag{18.20}$$

Thus, the Frobenius norm of X measures the discrepancy between M_k and M; our goal is to find a matrix M_k that minimizes this discrepancy, while constraining M_k to have rank at most k. If r is the rank of M, clearly $M_r = M$ and the Frobenius norm of the discrepancy is zero in this case.

We now describe how the singular value decomposition can be used to solve this matrix approximation problem, then derive from it an application to approximating term-document matrices. We invoke the following three-step procedure to this end:

1. Given M, construct its SVD in the form shown in (18.18); thus, $M = U\Sigma V^T$.
2. Derive from Σ the matrix Σ_k formed by replacing by zeros the $r - k + 1$ smallest singular values on the diagonal of Σ.
3. Compute and output $M_k = U\Sigma_k V^T$ as the rank-k approximation to M.

Note first that the rank of M_k is at most k: this follows from the fact that Σ_k has at most k non-zero values. Next, we recall the intuition of the example in (18.9): the effect of small eigenvalues on matrix products is small. Thus, it seems plausible that replacing these small eigenvalues by zero will not substantially alter the product, leaving it "close" to M. The following theorem due to Eckart and Young tells us that, in fact, this procedure yields the best possible result.

Theorem 18.4

$$\min_{X|\ rank(X)=k} ||M - X||_F = ||M - M_k||_F = \sigma_{k+1}. \tag{18.21}$$

Recalling that the singular values are in decreasing order $\sigma_1 \geq \sigma_2 \geq \cdots$, we learn from Theorem 18.4 that M_k is the best rank-k approximation to M, incurring an error (measured by the Frobenius norm of $M - M_k$) equal to σ_{k+1}. Thus, the larger k is the smaller this error (and in particular, for $k = r$, the error is $\sigma_{r+1} = 0$, since $\Sigma_r = \Sigma$ and thus $M_r = M$).

FIGURE

To derive further insight into why the process of truncating the smallest $r - k + 1$ singular values in Σ helps generate a rank-k approximation of low error, we examine the form of M_k:

$$M_k = U\Sigma_k V^T \tag{18.22}$$

$$= U \begin{pmatrix} \sigma_1 & 0 & 0 & 0 & 0 \\ 0 & \cdots & 0 & 0 & 0 \\ 0 & 0 & \sigma_k & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots \end{pmatrix} V^T \tag{18.23}$$

$$= \sum_{i=1}^{k} \sigma_i \vec{u}_i \vec{v}_i^T, \tag{18.24}$$

where $\vec{u}_i$ and $\vec{v}_i$ are the ith columns of U and V, respectively. Thus, $\vec{u}_i\vec{v}_i^T$ is a rank-1 matrix, so that we have just expressed M_k as the sum of k rank-1 matrices each weighted by a singular value. As i increases, the contribution of the rank-1 matrix $\vec{u}_i\vec{v}_i^T$ is weighted by a sequence of shrinking singular values σ_i.

Before discussing the approximation of a term-document matrix M by one of lower rank, we first motivate such an approximation. Recall the vector space representation of documents and queries introduced in Chapter 7. This vector space representation enjoys a number of advantages including the uniform treatment of queries and documents as vectors, the induced score computation based on cosine similarity, the ability to weight different terms differently, and its extension beyond document retrieval to such applications as clustering and classification. The vector space representation does, however, suffer from its inability to cope with two classic problems arising in natural languages: *synonymy* and *polysemy*. Synonymy refers to a case where two different words (say car and automobile) have the same meaning. Because the vector space representation fails to capture the relationship between synonymous terms such as car and automobile – according each a separate dimension in the vector space – we may have a situation where the computed similarity $q \cdot d$ between a query q (say, car) and a document d containing these terms underestimates the true similarity that a user would perceive. Polysemy on the other hand refers to the case where a term such as charge has multiple meanings, so that the computed similarity $q \cdot d$ overestimates the similarity that a user would perceive. Could we use the co-occurrences of terms (whether, for instance, charge occurs in a document containing steed versus in a document containing electron) to capture the latent semantic associations of terms and alleviate these problems?

Even for a corpus of modest size, the term-document matrix M is likely to have several tens of thousand of rows and columns, and a rank in the

LATENT SEMANTIC INDEXING

tens of thousands as well. In *latent semantic indexing* (generally abbreviated *LSI*), we use the SVD to construct a low-rank approximation M_k to the term-document matrix, for a value of k that is far smaller than the original rank of M. In the experimental work cited below, k is generally chosen to be in the low hundreds. We thus map each row/column (respectively corresponding to a term/document) to a k-dimensional space; this space is defined by the k principal eigenvectors (corresponding to the largest eigenvalues) of MM^T and M^TM. Note that the matrix M_k is itself still an $m \times n$ matrix, irrespective of k.

Next, we use the new k-dimensional LSI representation as we did the original representation – to compute similarities between vectors. A query vector $\vec{q}$ is mapped into its representation in the LSI space by the transformation

$$\vec{q}_k = \vec{q}^T U_k \Sigma_k^{-1}. \tag{18.25}$$

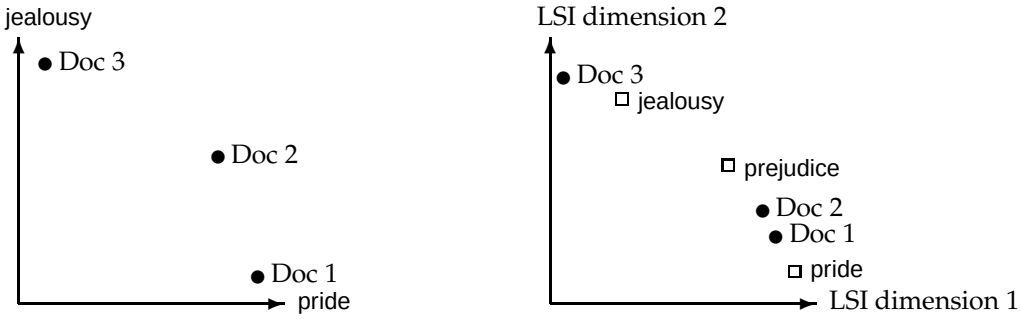
Now, we may use cosine similarities as in Chapter 7 to compute the similarity between a query and a document, or between two documents. We note again that this cosine computation, although occurring in $k \ll m$ dimensions, is computed as an inner product of two vectors each with m elements. Now that we have all the machinery for scoring and retrieval in place, we may ask: why bother?

First, the fidelity of the approximation of M_k to M could lead us to hope that the relative values of cosine similarities are preserved: if a query is close to a document in the original close, it remains relatively close in the k-dimensional space. But this in itself is not sufficiently interesting, especially given that the sparse query vector q turns into a dense query vector q_k in the low-dimensional space. This has a significant computational cost detailed in Problem 18.2 at the end of this chapter, when compared with the cost of processing q in its native form.

The low-rank approximation of M by M_k may be viewed as a *constrained optimization* problem: subject to the constraint that M_k have rank at most k, we seek a representation of the terms and documents comprising M with low Frobenius norm for the error $M - M_k$. When forced to squeeze the terms/documents down to a k-dimensional space, the intuition is that the SVD brings together terms with similar co-occurrences. This intuition suggests, then, that not only should retrieval quality not suffer too much from the dimension reduction, but in fact may *improve*.

Experiments with LSI tend to consistently bear out the following conclusions:

- The computational cost of the SVD is significant; at the time of this writing, we know of no successful experiment with over one million documents. This has been the biggest obstacle to the widespread adoption to LSI.



► **Figure 18.1** Original and LSI spaces. Only two of many axes are shown in each case.

- As we reduce k, recall tends to increase, as expected.
- Most surprisingly, a value of k in the low hundreds actually *increases* precision on many query benchmarks. This appears to confirm that for a suitable value of k, LSI addresses some of the challenges of synonymy and polysemy.

The experiments also documented some modes where LSI failed to match the performance of more traditional indexes and query languages. Most notably (and perhaps obviously), LSI shares two basic drawbacks of vector space retrieval: there is no good way of expressing negations (find documents that contain german but not shepherd), or Boolean conditions.

18.4 References and further reading

Pointers to SVD algorithms and software.

C. Eckart, G. Young, The approximation of a matrix by another of lower rank. Psychometrika, 1, 211-218, 1936.

http://www.cs.utk.edu/ berry/lsi++/

http://lsi.argreenhouse.com/lsi/LSIpapers.html

Dumais (1993) LSI meets TREC: A status report.

Dumais (1994) Latent Semantic Indexing (LSI) and TREC-2.

Dumais (1995) Using LSI for information filtering: TREC-3 experiments.

M. Berry, S. Dumais and G. O'Brien. Using linear algebra for intelligent information retrieval. SIAM Review, 37(4):573–595, 1995.

Hoffman.

Exercise 18.2

Stuff to be done

19 *Web search basics*

19.1 Background and history

The web is unprecedented in many ways: unprecedented in scale, unprecedented in the almost-complete lack of coordination in its creation, and unprecedented in the diversity of backgrounds and motives of its participants. Each of these contributes to making web search different – and generally far harder – than searching "traditional" documents.

The invention of hypertext (envisioned by Vannevar Bush in the 1940's and realized in working systems in the 1970's) significantly precedes the formation of the World Wide Web (which we will simply refer to the web), in the 1990's. The web attained widespread proliferation (to the point where it now claims a good fraction of humanity as participants) by relying on a simple, open client-server design: (1) the server communicated with the client via a protocol (the *http* or hypertext transfer protocol) that was lightweight and simple, asynchronously carrying a variety of payloads (text, images and – over time – richer media such as audio and video files) encoded in a simple markup language called *html* (for hypertext markup language); (2) the client – generally a *browser*, an application within a graphical user environment – can ignore what it does not understand. Each of these seemingly innocuous features contributed enormously to the growth of the web, so it is worthwhile to examine them further.

HTTP

HTML

The basic operation is as follows: a client (such as a browser) sends an *http request* to a *web server*. The browser specifies a *URL* (for *Universal Resource Locator*) such as `http://www.stanford.edu/home/atoz/contact.html`. In this example URL, the string `http` refers to the protocol to be used for transmitting the data. The string `www.stanford.edu` is known as the *domain* (sometimes the *top-level domain*) and specifies the root of a hierarchy of web pages (typically mirroring a filesystem hierarchy underlying the web server). In this example, `/home/atoz/contact.html` is a path in this hierarchy with a file `contact.html` that contains the information to be returned by the web server at `www.stanford.edu` in response to this request. The

URL

html-encoded file `contact.html` holds the content (in this instance, contact information for Stanford University) and the hyperlinks, as well as formatting rules for rendering this content in a browser. Such an http request thus allows us to fetch the content of a page, something that will prove to be useful to us for crawling and indexing documents (Chapter 20).

The designers of the first browsers made it easy to view the html markups on the content of a URL; this simple convenience allowed new users to create their own html content without extensive training and experience. As they did so, a second feature of browsers supported the rapid proliferation of web content creation and usage: browsers ignored what they did not understand. This did not, as one might fear, lead to the creation of numerous incompatible dialects of html. What it did promote was that amateur content creators could freely experiment with and learn from their newly created web pages without fear that a simple syntax error would "bring the system down". The result was that publishing on the web became a mass activity that was not limited to a few trained programmers, but rather open to tens and eventually hundreds of millions of individuals. This in turn meant that for most users and for most information needs, the web quickly became the best way to supply and consume information on everything from rare ailments to subway schedules.

Basic diagrams: html/http; search engine; taxonomy tree

The mass publishing of information on the web is essentially useless unless this wealth of information can be discovered and consumed by other users. Early attempts at making web information "discoverable" fell into two broad categories: (1) full-text index search engines such as Altavista, Excite and Infoseek and (2) taxonomies populated with web pages in categories, such as Yahoo! The former presented the user with a keyword search interface supported by inverted indexes and ranking mechanisms building on those introduced in earlier chapters. The latter allowed the user to browse through a hierarchical tree of category labels. While this is at first blush a convenient and intuitive metaphor for finding web pages, a number of challenges become apparent: (1) classifying web pages into taxonomy tree nodes is for the most part a manual editorial process, which is difficult to scale with the size of the web. Arguably, we only need to have "high-quality" web pages in the taxonomy, with only the best web pages for each category. However, just discovering these and classifying them accurately and consistently into the taxonomy entails significant human effort. (2) In order for a user to effectively discover web pages classified into the nodes of the taxonomy tree, the user's idea of what sub-tree(s) to seek for a particular topic should match that of the editors performing the classification. This quickly becomes challenging as the size of the taxonomy grows; the Yahoo! taxonomy tree surpassed 1000 distinct nodes fairly early on. Given these challenges, the popularity of taxonomies declined over time, even though variants (such as

About.com and the Open Directory Project) sprang up with subject-matter experts collecting and annotating web pages for each category.

The first generation of web search engines transported classical search techniques such as those in the preceding chapters to the web domain, focusing on the challenge of scale. The earliest web search engines had to contend with indexes containing millions of documents, which was a few orders of magnitude larger than any prior information retrieval system in the public domain. Indexing, query serving and ranking at this scale demanded harnessing together tens of machines to create highly available systems, again at scales not witnessed hitherto in a consumer-facing search application. The first generation of web search engines was largely successfully at solving these challenges while continually indexing a significant fraction of the web, all the while serving queries with sub-second response times. However, the quality and relevance of web search results left much to be desired owing to the idiosyncracies of content creation on the web. This necessitated the invention of new ranking and spam-fighting techniques in order to ensure the quality of the search results. To understand these, we next detail some of these characteristics of the web as a document corpus that demanded these new techniques.

19.2 Web characteristics

The essential feature that led to the explosive growth of the web – decentralized content publishing with essentially no central control of authorship – turned out to be the biggest challenge for web search engines in their quest to index and retrieve this content. Web page authors created content in dozens of (natural) languages and thousands of dialects, thus demanding many different forms of stemming and other linguistic operations. Because publishing was now open to tens of millions, web pages exhibited heterogeneity at a daunting scale, in many crucial aspects. First, content-creation was no longer the privy of editorially-trained writers; while this represented a tremendous democratization of content creation, it also resulted in a tremendous variation in grammar and style (and in many cases, no recognizable grammar or style). Indeed, web publishing in a sense unleashed the best and worst of desktop publishing on a planetary scale, so that pages quickly became riddled with wild variations in colors, fonts and structure. Indeed some web pages, including the professionally created home pages of some large corporations, consisted entirely of images (which, when clicked, led to richer textual content) – and therefore, no indexable text.

What about the substance of the text in web pages? The democratization of content creation on the web meant a new level of granularity in *opinion* on virtually any subject. This meant that the web contained truth, lies, contra-

dictions and suppositions on a grand scale. This gives rise to the question: which web pages does one trust? In a simplistic approach, one might argue that some publishers are trustworthy and others not – begging the question of how a search engine is to assign such a measure of trust to each website or web page. In Chapter 21 we will examine approaches to understanding this question. More subtly, there may be no universal, user-independent notion of trust; a web page whose contents are trustworthy to one user may not be so to another. In traditional (non-web) publishing this is not an issue: users self-select sources they find trustworthy. Thus one reader may find the reporting of *The New York Times* to be reliable, while another may prefer *The Wall Street Journal*. But when a search engine is the only viable means for a user to become aware of (let alone select) most content, this challenge becomes significant.

While the question "how big is the web?" has no easy answer (see Section 19.5 below), the question "how many web pages are in a search engine's index" is more precise (even this question has issues, as we will see below). By the end of 1995, Altavista reported that it had crawled and indexed approximately 30 million *static* web pages. Static web pages are those whose content does not vary from one request for that page to the next. For this purpose, a professor who manually updates his home page every week is considered to have a static web page, but an airport's flight status page is not. Since the number of static web pages was believed to be doubling every few months in 1995, engines such as Altavista had to constantly add hardware and bandwidth for crawling and indexing web pages.

WEB GRAPH

The set of static web pages together with the hyperlinks between them may be viewed as a directed *web graph*; Chapter 21 gives a more formal development and the applications of this view. As one might suspect, this directed graph is not *strongly connected*: there are pairs of pages such that one cannot proceed from one page of the pair to the other by following hyperlinks. The number of links going into (and out of) has averaged (over all web pages studied) from about 8 to 15, in a range of studies. There is ample evidence that these links are not randomly distributed; for one thing, the distribution of the number of links into a web page does not follow the Poisson distribution one would expect if every web page were to pick the destinations of its links uniformly at random. Rather, this distribution is widely reported to be a *power law*, in which the number of pages with *in-degree* (the number of links coming into a page) i is proportional to $1/i^{\alpha}$; the value of α typically reported by studies is 2.1.[1]

As important as it is to understand the creation and structure of the web, it is crucial that we understand as well the users of web search. This is

1. Cf. Zipf's law of the distribution of words in text in Chapter 5 (page 76), which is a power law with $\alpha = 1$.

again a significant change from traditional information retrieval, where users were typical professionals with at least some training in the art of phrasing queries over a well-authored corpus whose style and structure they understood well. In contrast, we search users tend to not know (or care) about the heterogeneity of web content, the syntax of query languages and the art of phrasing queries; indeed, a mainstream tool (as web search has come to become) should not place such onerous demands on billions of people. A range of studies has concluded that the average number of keywords in a web search is somewhere between 2 and 3. Syntax operators (Boolean connectives, wildcards, etc.) are seldom used, again a result of the composition of the audience – "normal" people, not trained information scientists.

19.2.1 Spam

Early in the history of web search, it became clear that web search engines were an important means for connecting advertisers to prospective buyers. A user searching for maui golf real estate is not merely seeking news or entertainment on the subject of housing on golf courses on the island of Maui, but instead likely to be seeking to purchase such a property. Sellers of such property and their agents, therefore, have a strong incentive to create web pages that rank highly on such a query. In a search engine whose scoring was based on term frequencies, a web page with numerous repetitions of maui golf real estate would rank highly. This led to the first generation of *spam*, which (in the context of web search) is the manipulation of web page content for the purpose of appearing high up in search results for selected keywords. To avoid irritating users with these repetitions, sophisticated *spammers* (the creators of spam) resorted to such tricks as rendering these repeated terms in the same color as the background. Despite these words being consequently invisible to the human user, a search engine indexer would parse the invisible words out of the html representation of the web page and index these words as being present in the page.

SPAM

At its root, spam stems from the heterogeneity of motives in content creation on the web. In particular, many web content creators have commercial motives and therefore stand to gain from manipulating search engine results. One might argue that this is no different from a company that uses large fonts to list its phone numbers in the yellow pages; but this generally costs the company more and is thus a fairer mechanism. A more apt analogy, perhaps, is the use of company names beginning with a long string of A's to be listed early in a yellow pages category. In fact, the yellow pages' model of companies paying for larger/darker fonts has been replicated in web search: in many engines, it is possible to pay to have one's web page included in the engine's search index – a model known as *paid inclusion*. Different engines

PAID INCLUSION

have different policies on whether to allow paid inclusion, and whether such a payment has any effect on search results.

Search engines soon became sophisticated enough in their spam detection to screen out an unusually large number of repetitions of particular keywords. Spammers responded with a richer set of spam techniques, the best known of which we now describe. The first of these techniques is *cloaking*: the spammer's web server returns different pages depending on whether the http request comes from a web search engine's crawler, or from a human user's browser. The former causes the web page to be indexed by the search engine under misleading keywords. When the user searches for these keywords and elects to view the page, he receives a web page that has altogether different content than that indexed by the engine. Such deception of search indexers was unknown in the traditional world of information retrieval; it stems from the fact that the web is partly collaborative but also partly competitive.

A *doorway page* contains text and meta-data carefully chosen to rank highly on selected search keywords. When a browser requests the doorway page, it is redirected to a page containing content of a more commercial nature. More complex spamming techniques involve manipulation of the meta-data related to a page including (for reasons we will see in Chapter 21) the links into a web page. Given that spamming is inherently an economically motivated activity, there has sprung around it an industry of *Search Engine Optimizers*, or SEO's. The business opportunity here is to provide consultancy services for clients who seek to have their web pages rank highly on selected keywords. Web search engines frown on this business of attempting to decipher and adapt to their proprietary ranking techniques and indeed announce policies on forms of SEO behavior they do not tolerate (and have been known to shut down search requests from certain SEO's for violation of these). Inevitably, the parrying between such SEO's (who gradually infer features of each web engine's ranking methods) and the web search engines (who adapt in response) is an unending struggle; indeed, the research sub-area of *adversarial information retrieval* has sprung up around this battle. One potent technique for addressing spammers fabricating the textual content of their web pages was the exploitation of the link structure of the web – a technique known as *link analysis*. The first web search engine to apply link analysis (to be detailed in Chapter 21) was Google, although all web search engines currently make use of it (and correspondingly, spammers invest considerable effort in subverting it as well).

SEARCH ENGINE OPTIMIZERS

ADVERSARIAL INFORMATION RETRIEVAL

19.3 Advertising as the economic model

We have seen in Section 19.2.1 that despite the web's somewhat utopian origins as the democratization of publishing, it soon assumed a commercial character. For corporations, the web was at the very least a marketing channel and in many cases a sales channel as well. In other words, the web became a mechanism for companies to reach their customers.

Early in the history of the web, companies used graphical banner advertisements on web pages at popular websites (such as news and entertainment sites such as MSN, America Online, Yahoo! and CNN). The primary purpose of these advertisements was *branding*: to convey to the viewer a positive feeling about the brand of the company placing the advertisement. Typically these advertisements were priced on a *cost per mil* (*CPM*) basis: the cost to the company of having its banner advertisement displayed 1000 times. Some websites struck contracts with their advertisers in which an advertisement was priced not by the number of times it is displayed (also known as *impressions*), but rather by the number of times it was *clicked on* by the user. This pricing model is known as the *cost per click* (*CPC*) model. In such cases, clicking on the advertisement led the user to a web page set up by the advertiser, where the user was induced to make a purchase. Here the goal of the advertisement was not so much brand promotion as to induce a transaction. These two distinct forms of advertising were already widely recognized in the context of conventional media such as broadcast and print. The interactivity of the web allowed the CPC billing model – clicks could be metered and monitored by the website and billed to the advertiser.

CPM

CPC

However, the user at the news/entertainment website was typically not there with an intent to make a purchase, as much as to consume news and entertainment. How could a company better target its audience? Companies did in fact achieve some measure of focus by carefully selecting the websites on which they advertised. For instance, a company selling golf clubs might wish to advertise on a web page containing sports news and even better on a web pages that discussed golf news, but perhaps not on a web page announcing recent breakthroughs on biochemistry.

Thus, demographic focusing of web advertising was already in use in the late 1990's, when web search engines were growing in usage (and therefore constantly in need of capital expenditures for hardware and bandwidth). The challenge for web search companies: how could they create a revenue stream that outweighed these expenditures? The pioneer in this direction was a company named Goto, which changed its name to Overture prior to eventual acquisition by Yahoo!. Goto was not, in the traditional sense, a search engine; rather, for every query term q it accepted *bids* from companies who wanted their web page shown on q. In response to the query q, Goto would return the pages of all advertisers who bid for q, ordered by their bids. Furthermore,

when the user clicked on one of the returned results, the corresponding advertiser would make a payment to Goto (in the initial implementation, this payment equalled the advertiser's bid for q).

Several aspects of Goto's model are worth highlighting. First, a user typing the query q into Goto's search interface was actively expressing an interest and intent related to the query q. For instance, a user typing golf clubs is more likely to be imminently purchasing a set, than one who is simply browsing news on golf. Second, Goto only got compensated when a user actually expressed interest in an advertisement – as evinced by the user clicking the advertisement. Taken together, these created a powerful mechanism by which to connect advertisers to consumers, quickly raising the annual revenues of Goto/Overture into hundreds of millions of dollars. This style of search engine came to be known variously as *sponsored search* or *paid placement*.

SPONSORED SEARCH
PAID PLACEMENT

Given these two kinds of search engines – the "pure" engines such as Google and Altavista, versus the sponsored search engines – the logical next step was to combine them into a single user experience. Current search engines follow precisely this model: they provide pure search results (generally known as *algorithmic search* results) as the primary response to a user's search, together with sponsored search results displayed separately and distinctively to the right of the algorithmic results. Retrieving sponsored search results and ranking them in response to a query has now become considerably more sophisticated than the simple Goto scheme described above; the process entails a blending of ideas from information retrieval and microeconomics, and is beyond the scope of this book. From the standpoint of advertisers, understanding how search engines do this ranking and how to allocate marketing campaign budgets to different sponsored search engines has become a profession known as *search engine marketing* (SEM).

ALGORITHMIC SEARCH

SEARCH ENGINE MARKETING

We close this section by mentioning an important pragmatic challenge to sponsored search. The inherently economic motives underlying sponsored search give rise to some participants attempting to subvert the system to their advantage. This can take many forms, one of which is known as *click spam*. There is currently no universally accepted definition of click spam. It refers (as the name suggests) to clicks on sponsored search results that are not from bona fide search users. For instance, a devious advertiser may attempt to exhaust the advertising budget of a competitor by clicking repeatedly (through the use of a robotic click generator) on that competitor's sponsored search advertisements. Search engines face the challenge of discerning which of the clicks they observe are part of a pattern of spam, to avoid charging their advertiser clients for such clicks.

CLICK SPAM

19.4 The search user experience

Given the sponsored search model, it is clear that the more user traffic a web search engine can attract, the more revenue it stands to earn from sponsored search. How do search engines differentiate themselves and grow their traffic? Here Google identified two principles that helped it grow at the expense of its competitors: (1) a focus on relevance, specifically precision (rather than recall) in the first few results; (2) a user experience that is lightweight, meaning that both the search query page and the search results page are uncluttered and almost entirely textual (with very few graphical elements). The effect of the first was simply to save users time in locating the information they sought; more on this below. The effect of the second is to provide a user experience that is extremely responsive, or at any rate not bottlenecked by the time to load the search query or results page.

19.4.1 User query needs

There appear to be three broad categories into which common web search queries can be grouped: (i) *informational*, (ii) *navigational* and (iii) *transactional*. We now explain these categories; it should be clear that some queries will fall in more than one of these categories, while others will fall outside them.

Informational queries seek general information on a broad topic, such as leukemia or Provence. There is typically not a single web page that contains all the information sought; indeed, users with informational queries typically try to assimilate information from multiple web pages.

Navigational queries seek the website or home page of a single entity that the user has in mind, say Lufthansa airlines. In such cases, the user's expectation is that the very first search result should be the home page of Lufthansa.

A transactional query is one that is a prelude to the user performing a transaction on the web – such as purchasing a product, downloading a file or making a reservation. In such cases, the engine should return results listing services that provide form interfaces for such transactions.

Just discerning which of these categories a query falls into can be challenging. The category not only governs the algorithmic search results, but the suitability of the query for sponsored search results (since the query may reveal an intent to purchase). For navigational queries, some have argued that the engine should return only a single result or even the target web page directly. Nevertheless, web search engines have historically engaged in a battle of bragging rights over which one indexes more web pages. Does the user really care? Perhaps not, but the media does highlight measurements (often statistically indefensible) of the sizes of various search engines. Users are influenced by these reports and thus, search engines do have to pay at-

tention to how their index sizes compare to competitors'. We now turn to a discussion of the size of a search engine index.

19.5 Index size and estimation

For informational (and to a lesser extent, transactional) queries, the user does care about the comprehensiveness of the engine. At a rough first cut, comprehensiveness grows with index size, although it does matter which specific pages an engine indexes – some are more informative than others. It is also difficult to reason about the fraction of the web indexed by a web page, because there is an infinite number of dynamic web pages; for instance, `http://www.yahoo.com/any_string` returns a valid html page rather than an error, politely informing the user that there is no such page at Yahoo! Such a "soft 404 error" is only one example of many web servers on the web that can generate an infinite number of valid web pages. Indeed, some of these are malicious *spider traps* devised to cause a search engine's crawler (the component that systematically fetches web pages for the engine's index, described in Chapter 20) to stay within a spammer's website and index many pages from that site.

One could ask the following better-defined question: given two search engines, what are the relative sizes of their indexes. Even this question turns out to be imprecise, for several reasons:

1. Search engines can return web pages in results whose contents they have not (fully or even partially) indexed. For one thing, engines generally index only the first few thousand words in a web page. In some cases, an engine is aware of a page p that is *linked to* by pages it has indexed, but has not indexed p itself. As we will see in Chapter 21, it is still possible to meaningfully return p in search results.

2. Search engines generally organize their indexes in various tiers and partitions, not all of which are examined on every search. For instance, a web page deep inside a website may be indexed but not retrieved on general web searches; it is however retrieved as a result on a search specific to that website.

Thus, search engine indexes include multiple classes of indexed pages, so that there is no single measure of index size. These issues notwithstanding, a number of techniques have been devised for estimating the ratio of the index sizes of two search engines, E_1 and E_2. The basic hypothesis underlying these techniques is that each engine indexes a fraction of the web chosen independently and uniformly at random. Notice that already, there are multiple questionable assumptions here. First, that there is a finite size

for the web from which each engine chooses a subset. Second, that each engine chooses an independent, uniformly chosen subset – as will be clear from the discussion of crawling in Chapter 20, this is far from true. However, if we begin with these assumptions, then we can invoke a classical estimation technique known as the *capture-recapture method*.

CAPTURE-RECAPTURE METHOD

Supposing that we could pick a random page from the index of E_1 and test whether it is in E_2's index; symmetrically, we test whether a random page from E_2 is in E_1. These experiments give us fractions x and y such that our estimate is that a fraction x of the pages in E_1 are in E_2, while a fraction y of the pages in E_2 are in E_1. Then, letting $|E_i|$ denote the size of the index of engine E_i, we have

$$x|E_1| = y|E_2| \Rightarrow \frac{|E_1|}{|E_2|} \approx \frac{y}{x}. \tag{19.1}$$

Note that if our assumption about E_1 and E_2 being independent and uniform random subsets of the web were true, and our sampling process unbiased, then Equation (19.1) gives us an unbiased estimator for $|E_1|/|E_2|$. We distinguish between two scenarios here: (1) the measurement is performed by someone with access to the index of one of the engines (say E_1); this would likely be an employee of E_1; (2) the measurement is performed by an independent party with no access to the innards of either engine. In the former case, we can simply pick a random document from one index. The latter case is more challenging; we begin with the sampling process by which we pick a random page from one engine *from outside the engine*, then describe the checking process by which we verify whether the random page is present in the other engine.

To implement the sampling phase, one strategy might be to generate a random page from the entire (idealized, finite) web and test it for presence in each engine. Unfortunately, picking a web page uniformly at random is a difficult problem. We briefly outline several attempts to achieve such a sample, pointing out the biases inherent to each; following this we describe in some detail one technique that much research has built on.

1. *Random searches:* Begin with a search log of web searches; send a random search from this log to E_1 and a random page from the results. Since such logs are not widely available outside a search engine, one implementation is to trap all search queries going out of a work group (say scientists in a research center) that agrees to have all its searches logged. This approach has a number of issues, including the bias from the types of searches made public by the work group. Further, a random document from the results of such a random search to E_1 is not the same as a random document from E_1.

2. *Random IP addresses:* A second approach is to generate random IP ad-

dresses and send a request to a web server residing at the random address, collecting all pages at that server. The biases here include the fact that many hosts might share one IP (due to a practice known as *virtual hosting*) or not accept http requests from the host where the experiment is conducted. Further, this technique is more likely to hit one of the many sites with few pages, skewing the document probabilities; this effect may be corrected if we understand the distribution of pages on a website.

3. *Random walks:* If the web were strongly connected, we could run a random walk starting at an arbitrary web page. This walk converges to a steady state distribution (see Chapter 21 for more background material on this), from which we can in principle pick a web page with a fixed probability. This method, too has a number of biases. First, the web is not strongly connected so that, even with various corrective rules, it is difficult to argue that we can reach a steady state distribution starting from any page. Second, the time it takes for the random walk to settle into this steady state is unknown and could exceed the length of the experiment.

Clearly each of these approaches is far from perfect. We now describe a fourth sampling approach, *random queries*. This approach is noteworthy for two reasons: (1) it has been successfully built upon for a series of increasingly refined estimates and (2) it has turned out to be the approach most likely to be misinterpreted and carelessly implemented, leading to misleading measurements. The idea is to pick a page (almost) uniformly at random from an engine's index by posing a random query to it. It should be clear that picking a set of random terms from (say) Webster's dictionary is not a good way of implementing this idea. For one thing, not all dictionary terms occur equally often, so this approach will not result in documents being chosen uniformly at random from the engine. For another, there are a great many "terms" in the web corpus that do not occur in a standard dictionary such as Webster's.

To address the problem of lexicon terms not in a standard dictionary, we begin by amassing a sample web lexicon. This could be done by crawling a limited portion of the web, or by crawling a manually-assembled representative subset of the web such as Yahoo! (as was done in the earliest experiments with this method). Consider a conjunctive query with two or more randomly chosen words from this lexicon; all current web search engines support such conjunctive queries. Consider the event whether a page (in either engine) is in the results set of such a random conjunctive query. The probability of this event induces a distribution over all pages in the union of the two engines. Then, we estimate $|E_1|/|E_2|$ by taking the ratio of the corresponding induced distributions. The estimate can be improved by repeating the experiment a large number of times.

Operationally, we proceed as follows: we use a random conjunctive query on E_1 and pick from the top 100 returned results a page p at random. We

then test p for presence in E_2 as follows: we choose 6-8 low-frequency terms in p and use them in a conjunctive query for E_2. Both the sampling process and the testing process have a number of issues.

1. Our sample is biased towards longer documents.
2. Picking from the top 100 results of E_1 induces a bias from the ranking algorithm of E_1. Picking from all the results of E_1 makes the experiment slower. This is particularly so because most web search engines put up defenses against excessive robotic querying.
3. During the checking phase, a number of additional biases are introduced: for instance, E_2 may not handle 8-word conjunctive queries properly.
4. Either E_1 or E_2 may refuse to respond to the test queries, treating them as robotic spam rather than as bona fide queries.
5. There could be operational problems like connection time-outs.

A sequence of research has built on this basic paradigm to eliminate some of these issues; there is no perfect solution yet, but the level of sophistication in statistics for understanding the biases is increasing.

19.6 Duplication and mirrors

One aspect we have ignored in the above discussion of index size is *duplication*: the web contains multiple copies of the same content. By some estimates, as many as 40% of the pages on the web are duplicates of other pages. Many of these are legitimate copies; for instance, certain information repositories are mirrored simply to provide redundancy and thus access reliability. Search engines try to avoid indexing multiple copies of the same content, to keep down storage and processing overheads.

The simplest approach to detecting duplicates is to compute, for each web page, a *fingerprint* that is a succinct (say 64-bit) digest of the sequence of characters on that page. Then, whenever the fingerprints of two web pages are equal, we test whether the pages themselves are equal and if so declare one of them to be a duplicate copy of the other. This simplistic approach fails to capture a crucial and widespread phenomenon on the web: *near duplication*. In many cases, the contents of a web page are identical to those of another, except for a few characters – say, a notation showing the date and time at which the page was last modified. Even in such cases, we want to be able to declare the two pages to be close enough that we only index one copy. Short of exhaustively comparing all pairs of web pages (an infeasible task at the scale of billions of pages), how can we detect and filter out such near duplicates?

19.6.1 Shingling

SHINGLING The answer lies in a technique known as *shingling*. Given a sequence of terms in a document d and a positive integer k, define the k-shingles of d to be the set of all consecutive sequences of k terms in d. As an example, consider the following text: a rose is a rose is a rose. The 4-shingles for this text are a rose is a, rose is a rose and is a rose is. Note that the first two of these shingles each occur twice in the text. In fact, $k = 4$ is a typical value used in the detection of near-duplicate web pages. Intuitively, two documents are near duplicates if the sets of shingles generated from them are nearly the same. We now make this intuition precise, then develop a method for efficiently computing and comparing the sets of shingles for all web pages.

Let $S(d_i)$ denote the set of shingles of document d_i. Recall the Jaccard coefficient (Chapter 3, page 46), which measures the degree of overlap between the sets $S(d_1)$ and $S(d_2)$ as $|S(d_1) \cap S(d_2)|/|S(d_1) \cup S(d_2)|$; denote this by $J(S(d_1), S(d_2))$. Our test for near duplication between d_1 and d_2 is to compute this Jaccard coefficient; if it exceeds a preset threshold (say, 0.9), we declare them near duplicates and eliminate one from indexing. However, this does not appear to have simplified matters: we still have to compute Jaccard coefficients pairwise.

To avoid this, we use a form of hashing. First, we map every shingle into a hash value over a large space, say 64 bits. For $i = 1, 2$, let $H(d_i)$ be the corresponding set of 64-bit hash values derived from $S(d_i)$. We now invoke the following trick to detect document pairs whose sets $H()$ have large Jaccard overlaps. Let π be a random permutation from the 64-bit integers to the 64-bit integers. Denote by $\Pi(d_i)$ the set of permuted hash values in $H(d_i)$; thus for each $h \in H(d_i)$, there is a corresponding value $\pi(h) \in \Pi(d_i)$.

Let x_i^π be the smallest integer in $\Pi(d_i)$. Then

Theorem 19.1

$$J(\Pi(d_1), \Pi(d_2)) = Pr[x_1^\pi = x_2^\pi].$$

SHOULD WE PROVE THIS?

Thus, our test for the Jaccard coefficient of the shingle sets is probabilistic: we compare the computed values x_i^π from different documents. If a pair coincides, we have candidate near duplicates. Consider repeating the test independently for many random permutations π (a choice suggested in practice is 200). Call the set of these 200 values of x_i^π the *sketch* $\psi(d_i)$ of d_i. We can then estimate the Jaccard coefficient for any pair of documents d_i, d_j to be $|\psi_i \cap \psi_j|/200$; if this exceeds a preset threshold, we declare that d_i and d_j are similar.

How can we quickly compute $|\psi_i \cap \psi_j|/200$ for all pairs i, j? Indeed, how do we represent all pairs of documents that are similar, without incurring a

blowup that is quadratic in the number of documents? To this end, we now present several implementation details. First, we use fingerprints to remove all but one copy of *identical* documents. We may also remove common html tags and integers from the shingle computation, to eliminate shingles that occur very commonly in documents without telling us anything about duplication. Next we use a *union-find* algorithm to create clusters that contain documents that are similar. To do this, we must accomplish a crucial step: going from the set of sketches to the set of pairs i, j such that d_i and d_j are similar.

To this end, we compute the number of shingles common for any pair of documents whose sketches have any members in common. For this, we begin with the list of sorted $< x_i^\pi, d_j >$ pairs and for each x_i^π, generate all pairs i, j for which x_i^π is present in both their sketches. We then merge these into counts for each pair i, j with non-zero sketch overlap; applying the preset threshold, we know which pairs i, j have heavily overlapping sketches. For instance, if the preset threshold were 80%, we need the merged count to be at least 160. As we identify such pairs, we run the union-find to group documents into near-duplicate "syntactic clusters". This is essentially a variant of the single-link clustering algorithm introduced in Section 17.2 (page 273).

One final trick cuts down the space needed in the computation of $|\psi_i \cap \psi_j|/200$ for pairs i, j, which in principle could still demand space quadratic in the number of documents. The idea is to remove from consideration those pairs i, j whose sketches have few shingles in common. To this end, we preprocess the sketch for each document as follows: sort the x_i^π in the sketch, then shingle this sorted sequence to generate a set of *super-shingles* for each document. If two documents have a super-shingle in common, we proceed to compute the precise value of $|\psi_i \cap \psi_j|/200$. This again is a heuristic but can be highly effective in cutting down the number of i, j pairs for which we accumulate the sketch overlap counts.

19.7 References and further reading

20 *Web crawling and indexes*

20.1 Overview

Web crawling is the process by which we gather pages from the web, for the primary purpose of indexing them and supporting a search engine. The objective of crawling is to quickly and efficiently gather as many useful web pages as possible, together with the link structure that interconnects them. In Chapter 19 we studied the complexities of the web stemming from its creation by millions of un-coordinated individuals. In this chapter we study the resulting difficulties for crawling the web.

Web crawling is performed with a variety of motivations ranging from small student projects to building mammoth commercial search engines. The goal of this chapter is not to describe how to build the crawler for a full-scale commercial search engine. We focus instead on a range of issues that are generic to crawling from the student project scale to substantial research projects. We begin by listing desiderata for web crawlers, leading to the discussion in Section 20.2 of how each of these issues is addressed. We list these desiderata in two categories: features that web crawlers *must* provide, followed by features they *should* provide.

20.1.1 Features a crawler *must* provide

Robust: The web contains servers that create *spider traps*, maliciously generating web pages that mislead crawlers into getting stuck fetching an infinite number of pages in a particular domain. Crawlers must be designed to be resilient to such traps. (Not all such traps are malicious; some are the inadvertent side-effect of faulty website development.)

Politeness: Web servers have both implicit and explicit policies regulating the rate at which a crawler can visit them. These *politeness* policies must be respected.

20.1.2 Features a crawler *should* provide

Distributed: The crawler should have the ability to execute in a distributed fashion across multiple machines.

Scalable: The crawler architecture should permit scaling up the crawl rate by adding extra machines and bandwidth.

Performance and efficiency: The crawl system should make maximum use of various system resources including processor, storage and network bandwidth.

Quality: Given that significant fractions of the web are of poor utility in terms of serving user query needs, the crawler should be biased towards fetching "useful" pages first.

Freshness: In many applications, the crawler should operate in continuous mode: it should obtain fresh copies of previously fetched pages. A search engine crawler, for instance, can thus ensure that the search engine's index contains a fairly faithful representation of each indexed web page. For such continuous crawling, a crawler should be able to crawl a page with a frequency that approximates the rate of change of that page.

Extensible: Crawlers should be designed to be extensible in many ways – to cope with new data formats, new fetch protocols, and so on. This demands that the crawler architecture be modular.

20.2 Crawling

The basic operation of any hypertext crawler (whether for a web, an intranet or other hypertext corpus) is as follows. The crawler begins with one or more URLs that constitute a *seed set*. It picks a URL from this seed set, then fetches the web page at that URL. The fetched page is then parsed, to extract both the text and the links from the page (each of which points to another URL). The extracted text is fed to a text indexer (described in Chapters 4 and 5). The extracted links (URLs) are then added to a *URL frontier*, which at all times consists of URLs whose corresponding pages have yet to be fetched by the crawler. (Initially, the URL frontier contains the seed set; as pages are fetched, the corresponding URLs are deleted from the URL frontier.) The entire process may be viewed as traversing the web graph (see Chapter 19).

This seemingly simple recursive traversal of the web graph is complicated by the demands above on a practical web crawling system: the crawler has to be distributed, scalable, efficient, polite, robust and extensible while fetching pages of high quality. We now examine the effects of each of these issues.

MERCATOR Our treatment follows the design of the *Mercator* crawler that has formed the basis of a number of research and commercial crawlers. As a reference point, fetching a billion pages (a small fraction of the static web at present) in a month-long crawl requires fetching several hundred pages each second. We will see how to use a multi-threaded design to address several bottlenecks in the overall crawler system in order to attain this fetch rate.

20.2.1 Crawler architecture

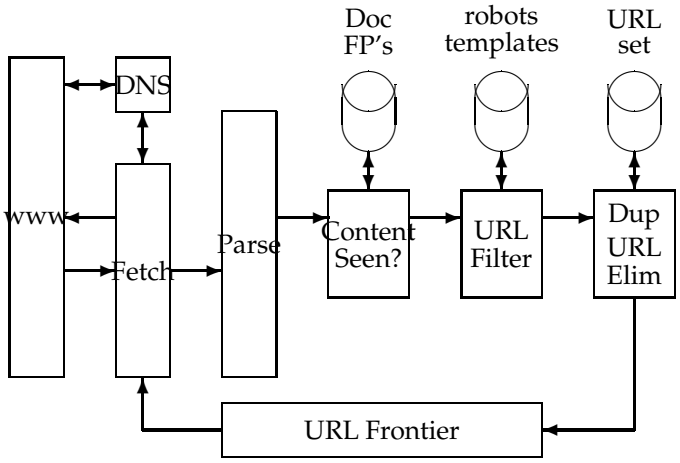
The simple scheme outlined above for crawling demands several modules:

1. The URL frontier, containing URLs yet to be fetched in the current crawl (in the case of continuous crawling, a URL may have been fetched previously but is back in the frontier for re-fetching).
2. A *DNS resolution* module that determines the web server from which to obtain a URL to be fetched. We describe this further in Section 20.2.2.
3. A fetch module that retrieves the web page at a URL.
4. A parsing module that extracts the set of links from a fetched web page.
5. A module that determines whether an extracted link is already in the URL frontier or has recently been fetched.

Figure 20.1 shows how these modules fit together. Crawling is typically performed by anywhere from one to potentially hundreds of threads, each of which loops through the logical cycle in Figure 20.1. These threads may be run in a single process, or be partitioned amongst multiple processes running at different nodes of a distributed system. We begin by assuming that the URL frontier is in place and non-empty; we defer our description of the implementation of the URL frontier.

A crawler thread begins by taking a URL from the frontier and fetching the web page at that URL, generally using the http protocol. The fetched page is then written into a temporary store, from which a number of operations are performed on it. First, the thread tests whether a web page with the same content has already been seen at another URL. The simplest implementation for this would use a checksum; a more sophisticated test would invoke shingling (Chapter 19).

Next, the page is parsed and the text as well as the links in it are extracted. The text (with any tag information – e.g., terms in boldface) is passed on to the indexer. Link information including anchor text is also passed on to the indexer for use in ranking in ways described in Chapter 21. In addition, each extracted link goes through a series of tests to determine whether the link should be added to the URL frontier.



► **Figure 20.1** The basic crawler architecture.

First, a *URL filter* is used to determine whether the extracted URL should be excluded from the frontier based on one of several tests. For instance, the crawl may seek to exclude certain domains (say, all .com URLs) – in this case the test would simply filter out the URL if it were from the .com domain. A similar test could be inclusive rather than exclusive. Many hosts on the web place certain portions of their websites off-limits to crawling, under a standard known as the *Robots Exclusion Protocol*. This is done by placing a file with the name robots.txt at the root of the URL hierarchy at the site. The robots.txt file must be fetched from a website in order to test whether the URL in consideration passes its restrictions and can therefore be added to the URL frontier. Rather than fetch it afresh for testing on each URL to be added to the frontier, a cache can be used to obtain a recently fetched copy of the file for the host. This is especially effective since many of the links extracted from a page fall within the host from which the page was fetched. Figure 20.2 shows an example robots.txt file that specifies that no robot should visit any URL starting with /yoursite/temp/, except the robot called "searchengine".

ROBOTS EXCLUSION PROTOCOL

At this point all URLs are *normalized*: often the html encoding of a link from a web page p indicates the target of that link relative to the page p. Thus, there is a relative link encoded thus in the html of the page en.wikipedia.org/wiki/Main_Page:

```
<a href="/wiki/Wikipedia:General_disclaimer" title="Wikipedia:General
disclaimer">Disclaimers</a>
```

```
User-agent: *
Disallow: /yoursite/temp/

User-agent: searchengine
Disallow:
```

► **Figure 20.2** An example robots.txt file.

points to the URL http://en.wikipedia.org/wiki/Wikipedia:General_disclaimer.

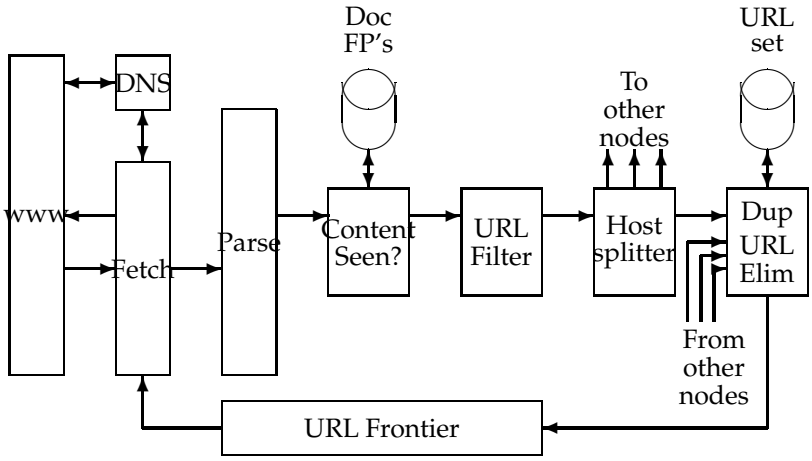
Finally, the URL is checked for *duplicate elimination*: if the URL is already in the frontier or (in the case of a non-continuous crawl) already crawled, we do not add it to the frontier. When the URL is added to the frontier, it is assigned a *priority* based on which it is eventually removed from the frontier for fetching. The details of this priority queuing are deferred to Section 20.2.3.

We conclude our description of the basic architecture by describing certain housekeeping tasks typically performed by a dedicated thread. The background thread is quiescent except that it wakes up once every few seconds to log crawl progress statistics (URLs crawled, frontier size, etc.), decide whether to terminate the crawl, or (once every few hours of crawling) checkpoint the crawl. In checkpointing, a snapshot of the crawler's state (say, the URL frontier) is committed to disk. In the event of a catastrophic crawler failure, the crawl is restarted from the most recent checkpoint.

Distributing the crawler

We have mentioned that the threads in a crawler as described above could run under different processes, each at a different node of a distributed crawling system. Such distribution is essential for scaling; it can additionally be of use in a geographically distributed crawler system where each node crawls hosts "near" it. The main idea is to partition the hosts being crawled amongst the crawler nodes. This partitioning can be done by (say) a hash function, or potentially some more specifically tailored policy (such as a crawler node in Europe to focus on European domains, although this isn't always dependable for several reasons – the routes that packets take through the internet do not always reflect geographic proximity, and in any case the domain of a host does not always reflect its physical location).

One detail remains: how do the various nodes of a crawler communicate and share URLs? The idea is to replicate the flow of Figure 20.1 at each node, with one essential difference: following the URL filter, we use a *host splitter* to despatch each surviving URL to the crawler node responsible for the URL. This modified flow is shown in Figure 20.3. The output of the host



► **Figure 20.3** Distributing the basic crawl architecture.

splitter goes into the Duplicate URL Eliminator block of each other node in the distributed system.

Exercise 20.1

Why is it better to partition hosts (rather than individual URLs) between the nodes of a distributed crawl system?

20.2.2 DNS resolution

IP ADDRESS

DNS RESOLUTION

DNS SERVER

Each web server (and indeed any host connected to the internet) has a unique *IP address*: a sequence of four bytes generally represented as four integers separated by dots; for instance 207.142.131.248 is the numerical IP address associated with the host www.wikipedia.org. Given a URL such as www.wikipedia.org in textual form, translating it to an IP address (in this case, 207.142.131.248) is a process known as *DNS resolution* or DNS lookup; here DNS stands for *Domain Name Service*. During DNS resolution, the program that wishes to perform this translation (in our case, a component of the web crawler) contacts a *DNS server* that returns the translated IP address. (In practice the entire translation may not occur at a single DNS server; rather, the DNS server contacted initially may recursively call upon other DNS servers to complete the translation.) For a more complex URL such as en.wikipedia.org/wiki/Domain_Name_System, the crawler component responsible for DNS resolution strips out the host

name – in this case en.wikipedia.org – and looks up the IP address for the host en.wikipedia.org.

DNS resolution is a well-known bottleneck in web crawling: due to the distributed nature of the Domain Name Service, DNS resolution may entail multiple requests and round-trips across the internet, requiring seconds and sometimes even longer. Right away, this puts in jeopardy our goal of fetching several hundred documents a second. A standard remedy is to introduce caching: URLs for which we have recently performed DNS lookups are likely to be found in the DNS cache, avoiding the need to go to the DNS servers on the internet. However, obeying politeness constraints (described further below) limits the rate of cache hits.

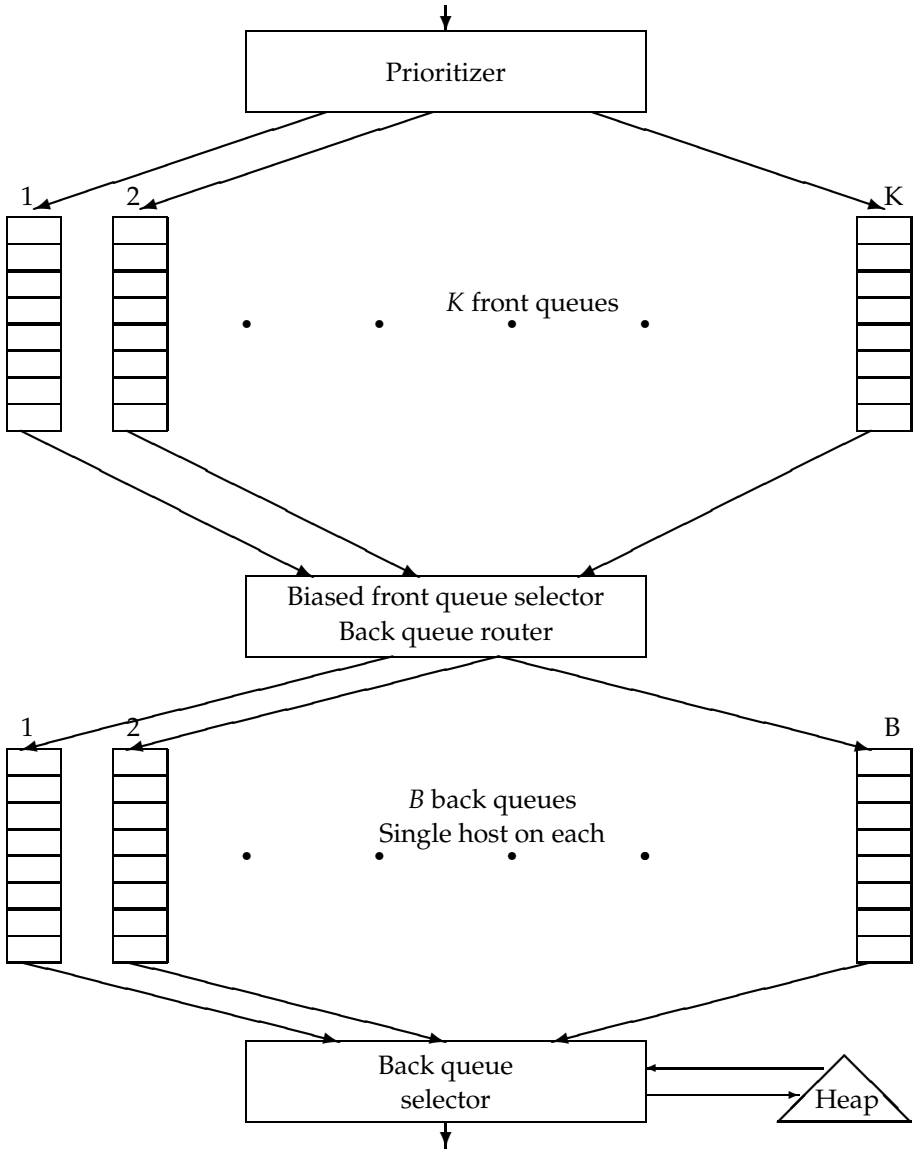
There is a another important difficulty in DNS resolution; the lookup implementations in standard libraries (likely to be used by anyone developing a crawler) are generally synchronous. This means that once a request is made to the Domain Name Service, other crawler threads are blocked until the first request is completed. To circumvent this, most web crawlers implement their own multi-threaded DNS resolver, as a component of the crawler. This consists of a local DNS server that collects and batches requests, then forwards a group of these to a DNS server on the internet.

20.2.3 The URL frontier

We now describe an implementation of the URL frontier. Fundamentally, the URL frontier is given a URL by its crawl process (or by the host splitter of another crawl process). It maintains the URLs in the frontier and regurgitates them in some order whenever a crawler thread seeks a URL. Two important considerations govern the order in which URLs are returned by the frontier. First, pages that change frequently should be prioritized for frequent crawling. The second consideration is called *politeness*: we must avoid repeated fetch requests to a host within a short time span. The likelihood of this is exacerbated because of a form of locality of reference: many URLs link to other URLs at the same host. As a result, a URL frontier implemented as a simple priority queue might result in a burst of fetch requests to a host. This may occur even if we were to constrain the crawler so that at most one thread could fetch from any single host at any time. A common heuristic is to insert a gap between successive fetch requests to a host that is an order of magnitude larger than the time taken for the most recent fetch from that host.

Figure 20.4 shows a polite and prioritizing implementation of a URL frontier. Its goals are to ensure that (i) only one connection be open at a time to any host; (ii) a waiting time of a few seconds occurs between successive requests to a host and (iii) robots.txt restrictions are obeyed.

The two major sub-modules are a set of *K front queues* in the upper por-



► **Figure 20.4** The URL frontier. The URL frontier. URL's extracted from already crawled pages flow in at the top of the figure. A crawl thread requesting a URL extracts it from the bottom of the figure. En route, a URL flows through one of several *front queues* that manage its priority for crawling, followed by one of several *back queues* that manage the crawler's politeness.

Host	Back queue
stanford.edu	23
microsoft.com	47
acm.org	12

► **Figure 20.5** Example of an auxiliary hosts-to-back queues table.

tion of the figure, and a set of *back queues* in the lower part; all of these are FIFO queues. The front queues implement the prioritization, while the back queues implement politeness. We now describe the flow of a URL added to the frontier as it makes its way through the front and back queues. First, a *prioritizer* assigns to the URL an integer priority i between 1 and K based on its download history (taking into account the rate at which the web page at this URL has changed between previous crawls). For instance, a document that has exhibited frequent change would be assigned a higher priority. Other heuristics could be application-dependent and explicit – for instance, URLs from news services may always be assigned the highest priority. The URL is now appended to the ith of the front queues.

Each of the B back queues maintains the following invariants: (i) it is non-empty while the crawl is in progress and (ii) it only contains URLs from a single host. An auxiliary table T (Figure 20.5) is used to maintain the mapping from hosts to back queues. In addition, we maintain a heap with one entry for each back queue, the entry being the earliest time t_e at which the host corresponding to that queue can be contacted again.

A crawler thread requesting a URL from the frontier extracts the root of this heap and (if necessary) waits until the corresponding time entry t_e. It then takes the URL u at the head of the back queue q corresponding to the extracted heap root, and proceeds to fetch the URL u. After fetching u, the calling thread checks whether q is empty. If so, it picks a front queue and extracts from its head a URL v. The choice of front queue is biased (usually by a random process) towards queues of higher priority, ensuring that URLs of high priority flow more quickly into the back queues. We examine v to check whether there is already a back queue holding URLs from its host. If so, v is added to that queue and we reach back to the front queues to find another candidate URL for insertion into the now-empty queue q. This process continues until q is non-empty again, at which point the thread inserts a heap entry for q with a new earliest time t_e based on the properties of v (such as when its host was last contacted), then continues with its processing.

The number of front queues, together with the policy of assigning priorities and picking queues, determines the priority properties we wish to build into the system. The number of back queues governs the extent to which we

can keep all crawl threads busy while respecting politeness. The designers of Mercator recommend a rough rule of three times as many back queues as crawler threads.

On a web-scale crawl, the URL frontier grows to the point where it demands more memory at a node than is available. The solution is to let most of the URL frontier reside on disk. A portion of each queue is kept in memory, with more brought in from disk as it is drained in memory.

20.3 Distributing indexes

In Chapter 4 we considered distributed indexing (Section 4.2). We now consider the distribution of the index across a large cluster of machines for supporting querying. Two obvious alternative index implementations suggest themselves: *partitioning by terms*, also known as global index organization, and *partitioning by documents*, also know as local index organization. In the former, the dictionary of index terms is partitioned into subsets, each subset residing at a set of nodes. Along with the terms at a node, we keep the postings for those terms. A query is routed to the nodes corresponding to its query terms. In principle, this allows greater concurrency since a stream of queries with different query terms would in principle hit different sets of machines.

TERM PARTITIONING
DOCUMENT PARTITIONING

In practice, partitioning indexes by dictionary terms turns out to be unwieldy. Multi-word queries require the sending of long postings lists between sets of nodes for merging, and the cost of this easily outweighs the greater concurrency. Load balancing the partitioning is governed not by an a priori analysis of relative term frequencies, but rather by the distribution of query terms and their co-occurrences. Achieving good partitions is a function of the co-occurrences of query terms and entails the clustering of terms to optimize objectives that are not yet fully understood. Finally, this strategy makes implementation of dynamic indexing more difficult.

A more common implementation is to partition by documents: each node contains the index for a subset of all documents. Each query is distributed to all nodes, with the results from various nodes being merged before presentation to the user. This strategy trades more local disk seeks for less inter-node communication. One difficulty in this approach is that global statistics used in scoring – such as idf – must be computed across the entire corpus even though the index at any single node only contains a subset of the documents. These are computed by distributed "background" processes that periodically refresh the node indexes with fresh global statistics.

How does one decide the partition of documents to nodes? Based on our development of crawler architecture in Section 20.2.1 above, one simple approach would be to assign all pages from a host to a single node. In partic-

ular, this partitioning could follow the partitioning of hosts to crawler nodes described above. A danger of such partitioning is that on many queries, a preponderance of the results for a query would come from a small number of hosts (and hence a small number of index nodes). A hash of each URL into the space of index nodes results in a more uniform distribution of query-time computation across nodes.

At query time, the query is broadcast to each of the nodes, with the top k results from each node being merged to find the top k documents for the query. A common implementation heuristic is to partition the document collection into indexes of documents that are more likely to score highly on most queries (using, for instance, techniques in Chapter 21) and low-scoring indexes with the remaining documents. We only search the low-scoring indexes when there are too few matches in the high-scoring indexes.

20.4 Connectivity servers

CONNECTIVITY SERVER

For reasons to become clearer in Chapter 21, web search engines require a *connectivity server* that supports fast *connectivity queries* on the web graph. Typical connectivity queries are *which URLs link to a given URL?* and *which URLs does a given URL link to?* To this end, we wish to store mappings in memory from URL to outlinks, and from URL to inlinks. Applications include crawl control, web graph analysis, connectivity, crawl optimization and *link analysis* (to be covered in Chapter 21).

It is instructive to consider a web with four billion pages, each with ten links to other pages. In the simplest form, we would require 32 bits or 4 bytes to specify each end (source and destination) of each link, requiring a total of

$$4 \times 10^9 \times 10 \times 8 = 3.2 \times 10^{11}$$

bytes of memory. We now show how some basic properties of the web graph can be exploited to use well under 10% of this memory requirement. At first sight, we appear to have a data compression problem – which is amenable to a variety of standard solutions. However, our goal is not to simply squeeze the web graph to fit into memory; we must do so in a way that supports connectivity queries.

We assume that each web page is represented by a unique integer; the specific scheme used to assign these integers is described below. We build an *adjacency table* that resembles an inverted index: it has a row for each web page, ordered by the corresponding integers. The row for any page p contains a sorted list of integers, each corresponding to a web page that links to p. This table permits us to respond to queries of the form *which pages link to p?* In similar fashion we build a table whose entries are the pages linked to by p.

```
www.stanford.edu/alchemy
www.stanford.edu/biology
www.stanford.edu/biology/plant
www.stanford.edu/biology/plant/copyright
www.stanford.edu/biology/plant/people
www.stanford.edu/chemistry
```

► **Figure 20.6** A segment of the lexicographic ordering of all URLs.

This table representation cuts the space taken by the naive representation (in which we explicitly represent each link by its two end points, each a 32-bit integer) by 50%. Our description below will focus on the table for the links *from* each page; it should be clear that the techniques apply just as well to the table of links to each page. To further reduce the storage for the table, we exploit several ideas:

1. Similarity between lists: Many rows of the table have many entries in common. Thus, if we enumerate one version of several similar rows, the remainder can be succinctly expressed in terms of the prototypical version.

2. Locality: many links from a page go to "nearby" pages. This suggests that in encoding the destination of a link, we can often use small integers and thereby save space.

3. We use gap encodings in sorted lists: rather than store the destination of each link, we store the offset from the previous entry in the row.

We now develop each of these techniques.

Consider a *lexicographic* ordering of all URLs: we treat each URL as an alphanumeric string and sort these strings. For instance, a segment of this sorted order may look like Figure 20.6. For a lexicographic sort to truly model web locality, the domain name part of the URL should be inverted, so that www.stanford.edu becomes edu.stanford.www, but this is not necessary here since we are mainly concerned with links local to a single host.

To each URL, we assign its position in this ordering as the unique identifying integer. Figure 20.7 shows an example of such a numbering and the resulting table. We next exploit a property that stems from the way most websites are structured, resulting in the similarity and locality ideas noted above. Most website have a template with a set of links from each page in the site to a fixed set of pages on the site (such as its copyright notice, terms of use, and so on). Assuming this were true, the rows corresponding to pages in a website will have many table entries in common. Moreover, under the

```
1, 2, 4, 8, 16, 32, 64
1, 4, 9, 16, 25, 36, 49, 64
1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144
1, 4, 8, 16, 25, 36, 49, 64
```

► **Figure 20.7** A four-row segment of the table of links.

lexicographic ordering of URLs, it is very likely that the pages from a website appear as contiguous rows in the table.

Accordingly, we adopt the following strategy: we walk down the table, encoding each table row in terms of the seven preceding rows. In the example of Figure 20.7, we could encode the fourth row as "the same as the row at offset 2 (meaning, two rows earlier in the table), with 9 replaced by 8". This requires the specification of the offset, the integer(s) dropped (in this case 9) and the integer(s) added (in this case 8). The use of only the seven preceding rows has two advantages: (i) the offset can be expressed with only 3 bits; this choice is optimized empirically (for we could well used the previous 15 rows and used 4 bits to express the offset) and (ii) fixing the maximum offset to a small value like seven avoids having to perform an expensive search among all candidate prototypes in terms of which to express the current row.

What if none of the preceding seven rows is a good prototype for expressing the current row? This would happen, for instance, at each boundary between different websites as we walk down the rows of the table. In this case we simply express the row as starting from the empty set and "adding in" each integer in that row. Additionally, by using gap encodings to store the gaps (rather than the actual integers) in each row, and encoding these gaps tightly based on the distribution of their values, we obtain further space reduction. In experiments mentioned in Section 20.5, the series of techniques outlined here appears to use as few as 3 bits per link, on average – a dramatic reduction from the 64 required in the naive representation.

While these ideas give us a representation of sizeable web graphs that comfortably fits in memory, we still need to support connectivity queries. What is entailed in retrieving from this representation the set of links from a page? First, we need an index from a hash of the URL to its row number in the table. Next, we need to reconstruct these entries, which may be encoded in terms of entries in other rows. This entails following the offsets to reconstruct these other rows – a process that in principle could lead through many levels of indirection. In practice however, this does not happen very often. A heuristic for controlling this can be introduced into the construction of the table: when examining the preceding seven rows as candidates from which to model the current row, we demand a threshold of similarity between the current row

and the candidate prototype. This threshold must be chosen with care. If the threshold is set too high, we seldom use prototypes and express many rows afresh. If the threshold is too low, most rows get expressed in terms of prototypes, so that at query time the reconstruction of a row leads to many levels of indirection through preceding prototypes.

20.5 References and further reading

The first web crawler appears to be Matthew Gray's Wanderer, written in the spring of 1993. The Mercator crawler is due to Najork and Heydon (Najork and Heydon 2001; 2002); the treatment in this chapter follows their work. Other classic early descriptions of web crawling include Burner (1997), Brin and Page (1998), Cho et al. (1998) and the creators of the Webbase system at Stanford (Hirai et al. 2000). The Robots Exclusion Protocol standard is described at http://www.robotstxt.org/wc/exclusion.html. Boldi et al. (2002) and Shkapenyuk and Suel (2002) provide more recent details of implementing large-scale distributed web crawlers.

Our discussion of DNS resolution (Section 20.2.2) uses the current convention for internet addresses, known as IPv4 (for Internet Protocol version 4) – each IP address is a sequence of four bytes. In the future, the convention for addresses (collectively known as the internet *address space*) is likely to use a new standard known as IPv6 (http://www.ipv6.org/).

Tomasic and Garcia-Molina (1993) and Jeong and Omiecinski (1995) are key early papers evaluating term partitioning versus document partitioning for distributed indexes. Document partitioning is found to be superior, at least when the distribution of terms is skewed, as it typically is in practice. This result has generally been confirmed in more recent work (MacFarlane et al. 2000). But the outcome depends on the details of the distributed system; at least one thread of work has reached the opposite conclusion (Ribeiro-Neto and Barbosa 1998, Badue et al. 2001).

The first implementation of a connectivity server was described by Bharat et al. (1998). The scheme discussed in this chapter, currently believed to be the best published scheme (achieving as few as 3 bits per link for encoding), is described in a series of papers by Boldi and Vigna (2004b;a).

21 *Link analysis*

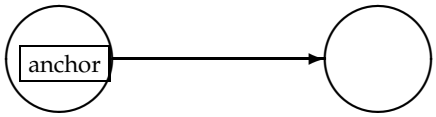
The analysis of hyperlinks and the graph structure of the web has been instrumental in the development of web search. In this chapter we focus on the use of hyperlinks for ranking web search results. Such *link analysis* is one of many factors considered by web search engines in computing a composite score for a web page on any given query. We begin by reviewing some basics of the web as a graph in Section 21.1, then proceed to the technical development of the elements of link analysis for ranking.

Link analysis for web search has intellectual antecedents in the field of *citation analysis*, aspects of which overlap with an area known as *bibliometrics*. These disciplines seek to quantify the influence of scholarly articles by analyzing the pattern of citations amongst them. Much as citations are a form by which a scholarly article confers authority on other articles, link analysis on the web views hyperlinks from a web page to another as a form of conferral of authority. Clearly, not every citation or hyperlink implies such authority conferral; for this reason, simply measuring the quality of a web page by the number of in-links (citations) is not robust enough. For instance, one may contrive to set up multiple web pages pointing to a target web page, with the intent of artificially boosting the latter's tally of in-links. Nevertheless, the phenomenon of citation is prevalent and dependable enough that it is feasible for web search engines to derive useful signals for ranking from more sophisticated link analysis.

21.1 The web as a graph

Consider the *static* web consisting of static html pages together with the hyperlinks between them. We view this static web as a directed graph in which each web page is a node and each hyperlink a directed edge.

Figure 21.1 shows two nodes A and B from the web graph, each corresponding to a web page, with a hyperlink from A to B. We refer to the set of all such nodes and directed edges as *the web graph*.



► **Figure 21.1** Two nodes of the web graph joined by a link.

```
<a href="http://www.acm.org/jacm/">Journal of the ACM.</a>
```

► **Figure 21.2** A fragment of html code.

Figure 21.1 also shows that (as is the case with most links on web pages) there is some text surrounding the origin of the hyperlink on page A. This text is generally encapsulated in the `href` tag that encodes the hyperlink in the html code of page A, and is referred to as *anchor text*. Our study of link analysis builds on two intuitions:

ANCHOR TEXT

1. The hyperlink from A to B connotes a conferral of authority on page B, by the creator of page A.
2. The anchor text describes the page B.

We begin by examining the second intuition, before proceeding to the first.

21.1.1 Anchor text

Figure 21.2 shows a fragment of html code from a web page, showing a hyperlink pointing to the home page of the Journal of the ACM. In this case, the link points to the page `http://www.acm.org/jacm/` and the anchor text is *Journal of the ACM.* Clearly, in this example the anchor is descriptive of the target page. But then the target page (B = `http://www.acm.org/jacm/`) itself contains the same description as well as considerable additional information on the journal. So what use is the anchor text?

The web is full of instances where the page B does not provide an accurate description of itself. In many cases this is a matter of how the publishers of page B choose to present themselves; this is especially common with corporate web pages, where a web presence is a marketing statement.

For example, at the time of the writing of this book the home page of the IBM corporation (`http://www.ibm.com`) did not contain the term computer anywhere in its html code, despite the fact that IBM is widely viewed as the world's largest computer maker. Similarly, the html code for the home page of Yahoo! (`http://www.yahoo.com`) does not at this time contain the word portal.

Thus, there is often a gap between how a web page presents itself and how many users of the web (and web search engines) would describe – and therefore search for – that web page. This represents an important gap that cannot be bridged by conventional inverted indexes, namely, web searchers need not use the same terms to describe a page they seek as the target page itself. In addition, many web pages are rich in graphics and images, and/or embed their text in these images; in such cases, the html parsing performed when crawling such web pages will not extract text that is useful for indexing these pages.

In such cases, the fact that the anchors of many hyperlinks pointing to `http://www.ibm.com` include the word computer can be exploited by web search engines. For instance, the anchor text terms can be included as terms under which to index the target web page. Thus, the postings for the term computer would include the document `http://www.ibm.com` and that for the term portal would include the document `http://www.yahoo.com`. A special indicator is used to show that these terms occur as anchor (rather than in-page) text. As with in-page terms, anchor text terms are generally weighted based on frequency, with a penalty for terms that occur very often (the most common terms in anchor text across the web are Click and here).

The use of anchor text has some interesting side-effects. Searching for big blue on most web search engines returns the home page of the IBM corporation as the top hit; this is consistent with the popular nickname that many people use to refer to IBM. On the other hand, there have been (and continue to be) many instances where derogatory anchor text such as evil empire leads to somewhat unexpected results on querying for these terms on web search engines. This phenomenon has been exploited in orchestrated campaigns against specific sites. More generally, orchestrated anchor text may be viewed as a form of spamming, since a website can create misleading anchor text pointing to itself, to boost its ranking on selected query terms – the inverse of derogatory anchor text. In other words, whereas anchor text indexing relies on the spontaneous, freeform text descriptions of many web page creators in describing a target page, it is subject to misuse. Detecting and combating such systematic abuse of anchor text is another form of spam detection that web search engines perform.

Exercise 21.1

Is it always possible to follow directed edges (hyperlinks) in the web graph from any node (web page) to any other? Why or why not?

Exercise 21.2

Find an instance of misleading anchor-text on the web.

Exercise 21.3

Given the collection of anchor-text phrases for a web page x, suggest a heuristic for one term or phrase from this collection that is most descriptive of x.

Exercise 21.4

Does your heuristic in the previous exercise take into account a single domain D repeating anchor text for x from multiple pages in D?

21.2 Pagerank

We now study our first technique for using links in the web graph for ranking. The technique assigns to every node in the web graph a numerical score between 0 and 1, known as its *pagerank*. The pagerank score of node will depend on the link structure of the web graph. Given a query, a web search engine constructs a composite score for each web page that combines a text-based score such as cosine similarity (Chapter 7) together with the pagerank score. This composite score is used to provide a ranked list of results for the query.

PAGERANK

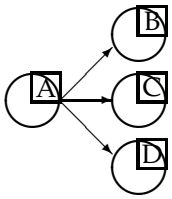
RANDOM SURFER

Consider a *random surfer* who begins at a web page (a node of the web graph) and executes a *random walk* on the web: at each of a succession of time steps, the surfer proceeds from the page A he is at to a randomly chosen web page that A hyperlinks to. Figure 21.3 shows the surfer at a node A, out of which there are three hyperlinks to nodes B, C and D; the surfer proceeds at the next time step to one of these three nodes, each being chosen with probability 1/3.

As the surfer proceeds in this fashion from node to node in his random walk, he visits some nodes more often than others; intuitively, these are nodes with many links coming in from other frequently visited nodes. To formalize this, we will invoke the theory of *Markov chains*; Section 21.2.1 reviews the basics of this theory. But before we review this material, we have a technicality to consider: what if the current location of the surfer, the node A, has no outgoing links?

MARKOV CHAINS

To address this we introduce an additional operation for our random surfer: the *teleport* operation. Whereas in the foregoing the surfer always moved from a node to other nodes it links to, in the teleport operation the surfer jumps from a node to any other node in the web graph, chosen uniformly at random. In other words, if n is the total number of nodes in the web graph,



► **Figure 21.3** The random surfer at node A proceeds with probability 1/3 to each of B, C and D.

the teleport operation takes the surfer to each node with probability $1/n$. Thus, the surfer could teleport back to his present position (with probability $1/n$).

In assigning a pagerank score to each node of the web graph, we use the teleport operation in two ways:

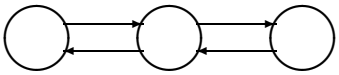
1. When at a node with no outgoing links, the surfer invokes the teleport operation.
2. At any other node (that does have outgoing links), the surfer invokes the teleport operation with probability $0 < \alpha < 1$ and the standard random walk (follow an outgoing link chosen uniformly at random as in Figure 21.3) with probability $1 - \alpha$, where α is a fixed parameter chosen in advance. Typically, α might be 0.1.

In Section 21.2.1 below, we will argue that when the surfer follows this combined process (random walk plus teleport) he visits each node v of the web graph a fixed fraction of the time $\pi(v)$ that depends on (1) the structure of the web graph and (2) the value of α. We call this value $\pi(v)$ the pagerank of v and will show how to compute this value in Section 21.2.2.

21.2.1 Markov chain review

A Markov chain is a *discrete-time stochastic process:* a process that occurs in a series of time-steps, at each of which a random choice is made. A Markov chain consists of n *states*; we re-use the symbol N here (already used for the number of web pages/nodes in our web graph) because in fact each web page will correspond to a state in the Markov chain we will formulate.

In addition, a Markov chain is characterized by an $n \times n$ *transition probability matrix* P each of whose entries is in the interval $[0, 1]$; further, the entries in each row of P add up to 1. The Markov chain is said to be in one of the n



► **Figure 21.4** A simple Markov chain with three states; the numbers on the links indicate the transition probabilities.

states at any given time-step; then, the entry P_{ij} tells us the probability that the state at the next time-step is j, conditioned on the current state being i. Each entry P_{ij} is known as a transition probability. It is clear then that

$$P_{ij} \in [0, 1], \forall i, j$$

and

$$\sum_{j=1}^{n} P_{ij} = 1, \forall i.$$

Fundamentally, the distribution of next states for a Markov chain depends only on the current state, and not on how the Markov chain arrived at the current state. Figure 21.4 shows a simple Markov chain with three states.

PROBABILITY VECTOR

A *probability vector* is a vector all of whose entries are in the interval $[0, 1]$, and the entries add up to 1. An n-dimensional probability vector each of whose components corresponds to one of the n states of a Markov chain can be viewed as a probability distribution over its states.

A random surfer on the web graph may be viewed as a Markov chain, with one state for each web page and each transition probability representing the probability of moving from one web page to another. The teleport operation contributes to these transition probabilities. This Markov chain can be readily derived from the adjacency matrix of the web graph, which encodes the hyperlinks (edges) between web pages. The probability distribution over the position of the surfer at any time step can be characterized by a probability vector $\vec{x}$. At $t = 0$ the surfer may begin at a state whose corresponding entry in $\vec{x}$ is 1 while all others are zero. By definition, the surfer's distribution at $t = 1$ is given by the probability vector $\vec{x}P$; at $t = 2$ by $(\vec{x}P)P = \vec{x}P^2$, and so on. We can thus compute a priori the surfer's distribution over the states at any time, given only the initial distribution and the transition probability matrix P.

Exercise 21.5

Write down the transition probability matrix for the example in Figure 21.4.

Exercise 21.6

Consider a web graph with three nodes 1, 2 and 3. The links are as follows: $1 \rightarrow 2, 3 \rightarrow 2, 2 \rightarrow 1, 2 \rightarrow 3$. Write down the transition probability matrices for the surfer's walk with teleporting, for the following three values of the teleport probability: (a) $\alpha = 0$; (b) $\alpha = 0.5$ and (c) $\alpha = 1$.

Exercise 21.7

A user of a browser can, in addition to clicking a hyperlink on the page x he is currently browsing, use the *back button* to go back to the page from which he arrived at x. Can such a user of back buttons be modeled as a Markov chain?

Exercise 21.8

Can one model the use of back buttons by a Markov chain that has a state for every hyperlink on the web, since a back button generally takes the user backwards on a link?

Exercise 21.9

What is the effect of repeated invocations of the back button?

If a Markov chain is allowed to run for many time steps, intuitively, each state is visited at a (different) frequency that depends on the structure of the Markov chain. In our running analogy, the surfer visits certain web pages (say, popular news home pages) more often than other pages. We now make this intuition precise, establishing conditions under which such a steady-state visit frequency exists. Following this, we set the pagerank of each node v to this steady-state visit frequency and show how it can be computed.

ERGODIC MARKOV CHAIN

Definition: A Markov chain is said to be *ergodic* if the following two conditions hold.

1. For any two states i, j, there is an integer $k \geq 2$ such that we have a sequence of k states $s_1 = i, s_2, \ldots, s_k = j$ such that $\forall 1 \leq \ell \leq k-1$, the transition probability $P_{s_\ell, s_{\ell+1}} > 0$.
2. There exists a time T_0 such that for all states j in the Markov chain, for all choices of the state i in which it is started at time step $t = 0$ and for all $t > T_0$, the probability of being in state j at time t is > 0.

Theorem 21.1 *For any ergodic Markov chain, there is a unique steady-state probability distribution over the states, $\vec{\pi}$, such that if $N(i,t)$ is the number of visits to state i in t steps, then*

$$\lim_{t \to \infty} \frac{N(i,t)}{t} = \pi(i),$$

where $\pi(i) > 0$ is the steady-state probability for state i.

Exercise 21.10

Consider a Markov chain with three states A, B and C, and transition probabilities as follows. From state A, the next state is B with probability 1. From B, the next state is either A with probability p_A, or state C with probability $1 - p_A$. From C the next state is A with probability 1. For what values of $p_A \in [0, 1]$ is this Markov chain ergodic?

Exercise 21.11

Show that for any directed graph, the Markov chain induced by a random walk with the teleport operation is ergodic.

Exercise 21.12

Show that the pagerank of every page is at least α/n. What does this imply about the difference in pagerank values (over the various pages) as α becomes close to 1?

It follows from Theorem 21.1 that the random walk with teleporting results in a unique distribution of steady-state probabilities over the states of the induced Markov chain. This steady-state probability for a state is the pagerank of the corresponding web page.

21.2.2 The Pagerank computation

How do we compute these pagerank values? Recall the definition of a left eigenvector (Equation 18.3) from Chapter 18; the left eigenvectors of the transition probability matrix P are N-vectors $\vec{\pi}$ such that

$$\vec{\pi} P = \lambda \vec{\pi}. \tag{21.1}$$

The N entries in the eigenvector $\vec{\pi}$ are the steady-state probabilities of the random walk with teleporting, and thus the pagerank values for the corresponding web pages. Indeed, one may interpret Equation 21.1 as telling us that if $\vec{\pi}$ is the probability distribution of the surfer across the web pages, he remains in the distribution $\vec{\pi}$ – thus, $\vec{\pi}$ is the steady-state distribution. If we were to compute the dominant left eigenvector of the matrix P – the one with eigenvalue 1 – we would have computed the pagerank values.

There are many algorithms available for computing left eigenvectors; the references at the end of Chapter 18 and the present chapter are a guide to these. To develop the reader's intuition, we give here a rather elementary method, sometimes known as *power iteration*. Recall from the discussion above that if $\vec{x}$ is the initial distribution over the states, then the distribution at time t is $\vec{x}P^t$. As t grows large, we would expect that the distribution $\vec{x}P^t$ is very similar to the distribution $\vec{x}P^{t+1}$, since for large t we would expect the Markov chain to attain its steady state. Recall, moreover, that by Theorem 21.1 this is independent of the initial distribution $\vec{x}$. Thus the power iteration method is simply to simulate the surfer's walk: begin at a state and run the walk for a large number of steps t, keeping track of the visit

frequencies at each of the states. After a large number of steps t, these frequencies "settle down" so that the variation in the computed frequencies is below some predetermined threshold. We declare these to be the computed pagerank values.

To aid the reader's intuition, we consider the web graph in Exercise 21.6 with $\alpha = 0.5$. The transition probability matrix of the surfer's walk with teleporting is then

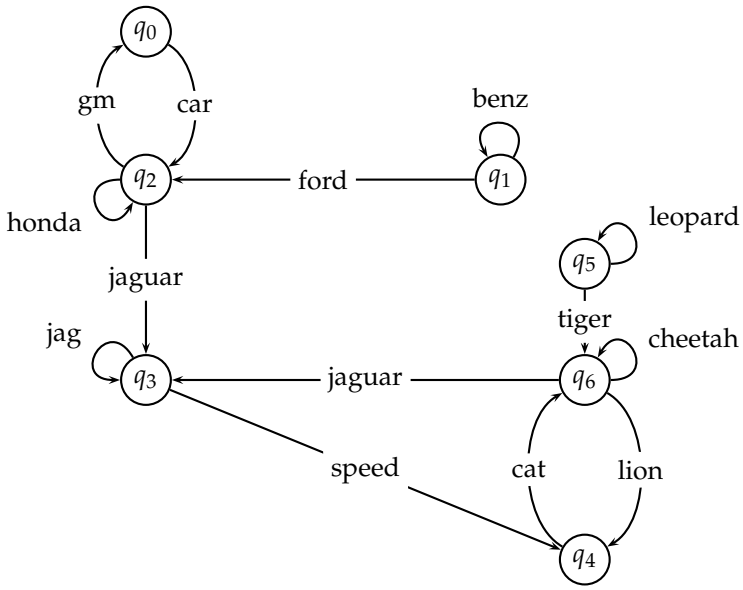
$$P = \begin{pmatrix} 1/6 & 2/3 & 1/6 \\ 5/12 & 1/6 & 5/12 \\ 1/6 & 2/3 & 1/6 \end{pmatrix}. \tag{21.2}$$

Next, imagine that the surfer starts in state 1, corresponding to the initial probability distribution vector $\vec{x} = (1\ 0\ 0)$. Then, after one step the distribution is

$$\vec{x}P = \begin{pmatrix} 1/6 & 2/3 & 1/6 \end{pmatrix}. \tag{21.3}$$

After two steps the distribution $\vec{x}P^2$ is $\vec{x} = (1/3 \quad 1/3 \quad 1/3)$, after three steps $\vec{x} = (7/24 \quad 5/12 \quad 7/24)$. Continuing for several steps, we see that the distribution converges to the steady state of $\vec{x} = (5/18 \quad 4/9 \quad 5/18)$. In this simple example, one may in fact directly calculate this steady-state probability distribution by observing the symmetry of the Markov chain: states 1 and 3 are symmetric, as evident from the fact that the first and third rows of the transition probability matrix in Equation (21.2) are identical. Postulating, then, that they both have the same steady-state probability and denoting this probability by p, we know that the steady-state distribution is of the form $\vec{\pi} = (p\ (1-2p)\ p)$. Now, using the identity $\vec{\pi} = \vec{\pi}P$, we solve a simple linear equation to obtain $p = 5/18$ and consequently, $\vec{\pi} = (5/18\ 4/9\ 5/18)$.

Note that the pagerank values of pages (and the implicit ordering amongst them) are independent of any query a user might pose; pagerank is thus a *query-independent* measure of the absolute quality of each web page. On the other hand, the relative ordering of pages should, intuitively, depend on the query being served. For this reason, search engines use absolute quality measures such as pagerank as just one of many factors in scoring a web page on a query.



► **Figure 21.5** A small web graph. Arcs are annotated with the word that occurs in the anchor text of the corresponding link.

Example 21.1: Consider the graph in Figure 21.5. For a teleportation rate of 0.1 its stochastic transition matrix is:

$$\begin{array}{ccccccc}
0.02 & 0.02 & 0.88 & 0.02 & 0.02 & 0.02 & 0.02 \\
0.02 & 0.45 & 0.45 & 0.02 & 0.02 & 0.02 & 0.02 \\
0.31 & 0.02 & 0.31 & 0.31 & 0.02 & 0.02 & 0.02 \\
0.02 & 0.02 & 0.02 & 0.45 & 0.45 & 0.02 & 0.02 \\
0.02 & 0.02 & 0.02 & 0.02 & 0.02 & 0.02 & 0.88 \\
0.02 & 0.02 & 0.02 & 0.02 & 0.02 & 0.45 & 0.45 \\
0.02 & 0.02 & 0.02 & 0.31 & 0.31 & 0.02 & 0.31
\end{array}$$

The pagerank vector of this matrix is:

$$\vec{x} = (0.05 \quad 0.04 \quad 0.11 \quad 0.25 \quad 0.21 \quad 0.04 \quad 0.31)$$

q_2, q_3, q_4 and q_6 are the nodes with at least two inlinks. Of these, q_2 has the lowest pagerank since activity is draining out of the top part of the graph – the walker can only return there through teleportation.

21.2.3 Topic-specific Pagerank

Thus far, we have discussed the pagerank computation with a teleport operation in which the surfer jumps to a random web page chosen uniformly at random. We now consider teleporting to a random web page chosen *non-uniformly*. To motivate this, consider a sports afficionado who would expect pages on sports to be ranked higher than non-sports pages. Let us imagine that the sports pages are "near" one another in the web graph. Then, a random surfer who frequently finds himself on random sports pages is likely (in the course of the random walk) to spend most of his time at sports pages, so that the steady-state distribution of sports pages is boosted.

Specifically, consider again a random surfer, endowed with a teleport operation as before. The difference is that instead of teleporting to a uniformly chosen random web page, the surfer teleports to *a random web page on the topic of sports.* We will not focus on how we collect all web pages on the topic of sports; in fact, we only need a non-zero subset S of sports-related web pages. This may be obtained, for instance, from a manually built directory of sports pages such as the open directory project (http://www.dmoz.org/) or that of Yahoo!

Provided the set S of sports-related pages is non-empty, it follows that there is a non-empty set of web pages $Y \supseteq S$ over which the random walk has a steady-state distribution; let us denote this *sports pagerank* distribution by $\vec{\pi}_s$. For web pages not in Y, we set the pagerank values to zero. We call $\vec{\pi}_s$ the *topic-specific pagerank* for sports.

TOPIC-SPECIFIC PAGERANK

Exercise 21.13

How does the set Y relate to S?

Exercise 21.14

Is the set Y always the set of all web pages? Why or why not?

Exercise 21.15

Is the sports pagerank of any page in S at least as large as its pagerank?

The above discussion does not demand that teleporting takes the random surfer to a uniformly chosen sports page; the distribution over teleporting targets S could in fact be arbitrary.

In like manner we can envision topic-specific pagerank distributions for each of several topics such as science, religion, politics and so on. Each of these distributions assigns to each web page a pagerank value in the interval $[0, 1)$. For a user interested in only a single topic from among these topics, we may invoke the corresponding pagerank distribution when scoring and ranking search results. This gives us the potential of considering settings in which the search engine knows what topics a user is interested in. This may

arise from users who either explicitly register their interests, or the system learns by observing the user's behavior and page access patterns over time.

Within this realm where a user's topics of interest are known to the engine, the above discussion leads to a pagerank distribution that is tailored to a single topic. But if, for instance, a user is known to have a mixture of interests from multiple topics, it is no longer clear how to adapt the methods described above. For instance, a user may have an interest mixture (or *profile*) that is 60% sports and 40% politics; can we compute a *personalized pagerank* for this user? At first glance, this appears daunting: how could we possibly compute a different pagerank distribution for each user profile (with, potentially, infinitely many possible profiles)? The first idea is to note that a user with this mixture of interests could teleport as follows: determine first whether to teleport to the set S of known sports pages, or to the set of known politics pages. This choice is made at random, choosing sports pages 60% of the time and politics pages 40% of the time. Once we choose (say) that a particular teleport step is to a random sports page, we choose a web page in S uniformly at random to teleport to. This in turn leads (see Exercise 21.16) to an ergodic Markov chain with a steady-state distribution that is personalized to this user's preferences over topics.

PERSONALIZED PAGERANK

While this idea has intuitive appeal, its implementation appears cumbersome: it seems to demand that for each user, we compute a transition probability matrix and compute its steady-state distribution. We are rescued by the fact that the evolution of the probability distribution over the states of a Markov chain is governed by a linear system. In Exercise 21.16 we show that it is not necessary to compute a pagerank vector for every distinct combination of user interests over topics; the personalized pagerank vector for any user can be expressed as a linear combination of the underlying topic-specific pageranks. For instance, the personalized pagerank vector for the user whose interests are 60% sports and 40% politics can be computed as

$$0.6\vec{\pi}_s + 0.4\vec{\pi}_p, \tag{21.4}$$

where $\vec{\pi}_s$ and $\vec{\pi}_p$ are the topic-specific pagerank vectors for sports and for politics, respectively.

Exercise 21.16

Consider a setting where we have two topic-specific pagerank values for each web page: a sports pagerank $\vec{\pi}_s$, and a politics pagerank $\vec{\pi}_p$. Let α be the (common) teleportation probability used in computing both sets of topic-specific pageranks. For $q \in [0,1]$, consider a user whose interest profile is divided between a fraction q in sports and a fraction $1-q$ in politics. Show that the user's personalized pagerank is the steady state distribution of a random walk in which – on a teleport step – the walk teleports to a sports page with probability q and to a politics page with probability $1-q$.

Exercise 21.17

Show that the Markov chain corresponding to this walk is ergodic and hence the user's personalized pagerank can be obtained by computing the steady state distribution of this Markov chain.

Exercise 21.18

Show that in this steady state distribution, the steady state probability for any web page x equals $q\pi_s(x) + (1-q)\pi_p(x)$.

21.3 Hubs and Authorities

HUB SCORE
AUTHORITY SCORE

The pagerank measure studied above assigns to each web page a score in the interval $(0, 1)$. We now develop a scheme in which every web page is assigned *two* scores, one called its *hub score* and the other its *authority score*. The idea is that for any query, we compute not one but two ranked lists of results – one ranking induced by the hub scores and the other by the authority scores.

This approach stems from a particular insight into the creation of web pages, reasoning that there are two primary kinds of web pages useful as results for *broad-topic searches*. By a broad topic search we mean not a navigational query (give me the home page of Air China), but rather a query such as "I wish to learn about leukemia". First, there are authoritative sources of information on the topic; in this case, the National Cancer Institute's page on leukemia would be such a page. We will call such pages *authorities*; in the computation we are about to describe, they are the pages that will emerge with high authority scores.

On the other hand, there are many pages on the web that are hand-compiled lists of links to authoritative web pages on a specific topic. These *hub* pages are not in themselves authoritative sources of topic-specific information, but rather compilations that someone with an interest in the topic has spent time putting together. The approach we will take, then, is to use these hub pages to discover the authority pages. In the computation we now develop, these hub pages are the pages that will emerge with high hub scores.

A good hub page is one that points to many good authorities. A good authority page is one that is pointed to by many good hub pages. We thus appear to have a circular definition of hubs and authorities; we will turn this into an iterative computation. Suppose that we have a subset of the web graph (meaning, a subset of all web pages, together with the hyperlinks amongst them); we will defer the discussion of how we pick this subset until Section 21.3.1. We will iteratively compute a hub score and an authority score for every web page in this subset.

For a web page x (a node in our chosen subset of the web graph) we use $h(x)$ to denote its hub score and $a(x)$ its authority score; initially, we set

$h(x) = a(x) = 1$ for all nodes x. We also denote by $x \mapsto y$ the existence of a hyperlink from x to y. The core of the iterative algorithm is the following pair of updates to the hub and authority scores of all pages, which capture the intuitive notions that good hubs point to good authorities and that good authorities are pointed to by good hubs.

$$\begin{aligned} h(x) &\leftarrow \sum_{x \mapsto y} a(y) \\ a(x) &\leftarrow \sum_{y \mapsto x} h(y). \end{aligned} \tag{21.5}$$

Thus, the first line of (21.5) sets the hub score of page x to the authority scores of the pages it links to. In other words, if x links to pages with high authority scores, its hub score increases. The second line plays the reverse role; if page x is linked to by good hubs, its authority score increases.

Exercise 21.19

If all the hub and authority scores are initialized to 1, what is the hub/authority score of a node after one iteration?

What happens as these updates are performed iteratively: we recompute hub scores for all nodes, then new authority scores based on these recomputed hub scores, and so on? To understand this, we recast the equations (21.5) into matrix-vector form. Let $\vec{h}$ and $\vec{a}$ denote the vectors of all hub and all authority scores respectively, for the pages in our subset of the web graph. Let A denote the *adjacency matrix* of the subset of the web graph that we are dealing with: A is a square matrix with one row and one column for each web page at hand. The entry A_{ij} is 1 if there is a hyperlink from page i to page j, and 0 otherwise. Then, we may write (21.5)

$$\begin{aligned} \vec{h} &\leftarrow A\vec{a} \\ \vec{a} &\leftarrow A^T\vec{h}, \end{aligned} \tag{21.6}$$

where A^T denotes the transpose of the matrix A. Notice now that the right hand side of each line of (21.6) is a vector that is the left hand side of the other line of (21.6). Substituting these into one another, we may rewrite (21.6) as

$$\begin{aligned} \vec{h} &\leftarrow AA^T\vec{h} \\ \vec{a} &\leftarrow A^TA\vec{a}. \end{aligned} \tag{21.7}$$

Now, (21.7) bears an uncanny resemblance to a pair of eigenvector equations; indeed, if we were to replace the $\leftarrow$ symbols by $=$ symbols and introduce the (unknown) eigenvalue, the first line of (21.7) becomes the equation for the

eigenvectors of AA^T, while the second becomes the equation for the eigenvectors of A^TA:

$$\begin{aligned} \vec{h} &= \lambda_h AA^T\vec{h} \\ \vec{a} &= \lambda_a A^TA\vec{a}. \end{aligned} \tag{21.8}$$

Here we have used λ_h to denote the eigenvalue of AA^T and λ_a to denote the eigenvalue of A^TA.

This observation leads to some key consequences:

1. The iterative updates in (21.5) (or equivalently, (21.6)), if scaled by the appropriate eigenvalues, are equivalent to the power iteration method for computing the eigenvectors of AA^T and A^TA. Thus, the iteratively computed entries of $\vec{h}$ and $\vec{a}$ settle into unique steady state values determined by the entries of A and hence the link structure of the graph.

2. In computing these eigenvector entries, we are not restricted to using the power iteration method; indeed, we could use any fast method for computing the eigenvectors of a matrix.

Exercise 21.20

How would one interpret the entries of the matrices AA^T and A^TA?

Exercise 21.21

What are the principal eigenvalues of AA^T and A^TA?

The resulting computation thus takes the following form:

1. Assemble the target subset of web pages, form the graph induced by their hyperlinks and compute AA^T and A^TA.

2. Compute the principal eigenvectors of AA^T and A^TA to form the vector of hub scores $\vec{h}$ and authority scores $\vec{a}$.

3. Output the top-scoring hubs and the top-scoring authorities.

The idea would be that since the iterative updates captured the intuition of good hubs and good authorities, the high-scoring pages we output would give us good hubs and authorities from the target subset of web pages. We now turn to the remaining detail: how do we gather a target subset of web pages around a topic such as leukemia?

Example 21.2: Assuming the query jaguar and double-weighting of links whose anchors contain the query word, the matrix A for Figure 21.5 is as follows:

$$\begin{array}{ccccccc} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 2 & 1 & 0 & 1 \end{array}$$

The hub and authority vectors are:

$$\vec{h} = (0.03 \quad 0.04 \quad 0.33 \quad 0.18 \quad 0.04 \quad 0.04 \quad 0.35)$$

$$\vec{a} = (0.10 \quad 0.01 \quad 0.12 \quad 0.47 \quad 0.16 \quad 0.01 \quad 0.13)$$

Here, q_3 is the main authority – two hubs (q_2 and q_6) are pointing to it via highly weighted jaguar links.

21.3.1 Choosing the subset of the web

In assembling a subset of web pages around a topic such as leukemia, we must cope with the fact that good authority pages may not contain the specific query term leukemia. This is especially true, as noted above in Section 21.1.1, when an authority page is using its web page to project a certain marketing presence. For instance, many pages on the IBM website are authoritative sources of information on computer hardware, even though these pages may not contain the term computer or hardware. However, a hub compiling computer hardware resources is likely to use these terms and also link to the relevant pages on the IBM website.

Building on these observations, the following procedure has been suggested for compiling the subset of the web for which to compute hub and authority scores.

1. Given a query (say leukemia), use a text index to get all pages containing leukemia. Call this the *root set* of pages.
2. Build the *base set* of pages, to include the root set as well as any page that either links to a page in the root set, or is linked to by a page in the root set.

We then use the base set for computing hub and authority scores. Why is the base set constructed in this manner? We identify three reasons:

1. First, as already observed, a good authority page may not contain the query text (such as computer hardware).

2. Second, if the text query manages to capture a good hub page P_h in the root set, then the inclusion of all pages linked to by any page in the root set will capture all the good authorities linked to by P_h in the base set.

3. Conversely, if the text query manages to capture a good authority page P_a in the root set, then the inclusion of pages points to P_a will bring other good hubs into the base set. In other words, the "expansion" of the root set into the best set enriches the common pool of good hubs and authorities.

Running this algorithm across a variety of queries reveals some interesting insights about link analysis. Frequently, the documents that emerge as top hubs and authorities include languages other than the language of the query. These pages were presumably drawn into the base set, following the assembly of the root set. Thus, some elements of *cross-language retrieval* (where a query in one language retrieves documents in another) are evident here; interestingly, this cross-language effect resulted purely from link analysis, with no linguistic translation taking place.

HITS

We conclude this section with some notes on implementing this algorithm, which is known as *HITS*, which is an acronym for *Hyperlink-Induced Topic Search*. First, we said that the root set consists of all pages matching the text query; in fact, implementations (see the references below) suggest that it suffices to use some 200 or so web pages for the root set, rather than all pages matching the text query. Second, we noted that any algorithm for computing eigenvectors may be used for computing the hub/authority score vector. In fact, we need not compute the exact values of these scores; it suffices to know the relative values of the scores so that we may identify the top hubs and authorities. To this end, it is possible that a small number of iterations of the power iteration method yield the relative ordering of the top hubs and authorities. Experiments have suggested that in practice, some five iterations of (21.5) yield fairly good results; moreover, since the link structure of the web graph is fairly sparse (the average web page links to about ten others), we do not perform these as matrix-vector products but rather as additive weight propagations along the hyperlinks.

21.4 References and further reading

The use of anchor text as an aid to searching and ranking stems from the work of McBryan (1994). The pagerank measure was developed in Brin and Page (1998) and in Page et al. (1998). A number of methods for the fast computation of pagerank values are surveyed in Berkhin (2005). The effect of the

teleport probability α has been studied by Baeza-Yates et al. (2005) and by Boldi et al. (2005). Topic-sensitive pagerank and variants were developed in Haveliwala (2002), Haveliwala (2003) and in Jeh and Widom (2003).

The HITS algorithm is due to Kleinberg (1999). Chakrabarti et al. (1998) developed variants that weighted links in the iterative computation based on the presence of query terms in the pages being linked and compared these to results from several web search engines. Bharat and Henzinger (1998) further developed these and other heuristics, showing that certain combinations outperformed the basic HITS algorithm. Borodin et al. (2001) provides a systematic study of several variants of the HITS algorithm. Ng et al. (2001) introduces a notion of *stability* for link analysis, arguing that small changes to link topology should not lead to significant changes in the ranked list of results for a query. Numerous other variants of HITS have been developed by a number of authors, the best know of which is perhaps SALSA (Lempel and Moran 2000).

Bibliography

Aizerman, M., E. Braverman, and L. Rozonoer. 1964. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control* 25:821–837.

Akaike, Hirotugu. 1974. A new look at the statistical model identification. *IEEE Transactions on automatic control* 19:716–723.

Allan, James. 2005. HARD track overview in TREC 2005: High accuracy retrieval from documents. In *The Fourteenth Text REtrieval Conference (TREC 2005) Proceedings*.

Allwein, Erin L., Robert E. Schapire, and Yoram Singer. 2000. Reducing multiclass to binary: A unifying approach for margin classifiers. *Journal of Machine Learning Research* 1:113–141.

Alonso, Omar, Sandeepan Banerjee, and Mark Drake. 2006. Gio: a semantic web application using the information grid framework. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pp. 857–858, New York, NY, USA. ACM Press.

Anagnostopoulos, Aris, Andrei Z. Broder, and Kunal Punera. 2006. Effective and efficient classification on a search-engine model. In *CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management*, pp. 208–217, New York, NY, USA. ACM Press.

Andoni, A., N. Immorlica, P. Indyk, and V. Mirrokni. 2007. Locality-sensitive hashing using stable distributions. In *Nearest Neighbor Methods in Learning and Vision: Theory and Practice*. MIT Press.

Anh, Vo Ngoc, Owen de Kretser, and Alistair Moffat. 2001. Vector-space ranking with effective early termination. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 35–42, New York, NY, USA. ACM Press.

Anh, Vo Ngoc, and Alistair Moffat. 2005. Inverted index compression using word-aligned binary codes. *Inf. Retr.* 8:151–166.

Anh, Vo Ngoc, and Alistair Moffat. 2006a. Improved word-aligned binary compression for text indexing. *IEEE Transactions on Knowledge and Data Engineering* 18: 857–861.

Anh, Vo Ngoc, and Alistair Moffat. 2006b. Pruned query evaluation using pre-computed impacts. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 372–379, New York, NY, USA. ACM Press.

Anh, Vo Ngoc, and Alistair Moffat. 2006c. Structured index organizations for high-throughput text querying. In *Proc. 13th Int. Symp. String Processing and Information Retrieval*, volume 4209 of *Lecture Notes in Computer Science*, pp. 304–315. Springer.

Azcarraga, Arnulfo P., and Teddy N. Yap Jr. 2001. Extracting meaningful labels for WEBSOM text archives. In *CIKM*, pp. 41–48.

Badue, Claudine Santos, Ricardo A. Baeza-Yates, Berthier Ribeiro-Neto, and Nivio Ziviani. 2001. Distributed query processing using partitioned inverted files. In *Eighth Symposium on String Processing and Information Retrieval (SPIRE 2001)*, pp. 10–20.

Baeza-Yates, Ricardo, Paolo Boldi, and Carlos Castillo. 2005. The choice of a damping function for propagating importance in link-based ranking. Technical report, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano.

Baeza-Yates, Ricardo, and Berthier Ribeiro-Neto. 1999. *Modern Information Retrieval*. Pearson Education.

Bahle, Dirk, Hugh E. Williams, and Justin Zobel. 2002. Efficient phrase querying with an auxiliary index. In *SIGIR 2002*, pp. 215–221.

Barzilay, Regina, and Micahel Elhadad. 1997. Using lexical chains for text summarization. In *Worksho on Intelligent Scalable Text Summarization*, pp. 10–17.

Beesley, Kenneth R. 1998. Language identifier: A computer program for automatic natural-language identification of on-line text. In *Languages at Crossroads: Proceedings of the 29th Annual Conference of the American Translators Association*, pp. 47–54.

Beesley, Kenneth R., and Lauri Karttunen. 2003. *Finite State Morphology*. Stanford, CA: CSLI Publications.

Berger, Adam, and John Lafferty. 1999. Information retrieval as statistical translation. In *SIGIR 22*, pp. 222–229.

Berkhin, P. 2005. A survey on pagerank computing. *Internet Mathematics* 2:73–120.

Bharat, Krishna, Andrei Broder, Monika Henzinger, Puneet Kumar, and Suresh Venkatasubramanian. 1998. The connectivity server: Fast access to linkage information on the web. In *Proceedings of the Seventh International World Wide Web Conference*, pp. 469–477.

Bharat, Krishna, and Monika R. Henzinger. 1998. Improved algorithms for topic distillation in a hyperlinked environment. In *Proceedings of SIGIR-98, 21st ACM International Conference on Research and Development in Information Retrieval*, pp. 104–111, Melbourne, AU.

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning*. Springer.

Boldi, P., B. Codenotti, M. Santini, and S. Vigna, 2002. Ubicrawler: A scalable fully distributed web crawler.

Boldi, P., M. Santini, and S. Vigna, 2005. Pagerank as a function of the damping factor.

Boldi, Paolo, and Sebastiano Vigna. 2004a. Codes for the world-wide web. *Internet Mathematics* 2:405–427.

Boldi, Paolo, and Sebastiano Vigna. 2004b. The webgraph framework i: Compression techniques. In *Proceedings of the 14th International World Wide Web Conference*, pp. 595–601. ACM press.

Boldi, Paolo, and Sebastiano Vigna. 2005. Compressed perfect embedded skip lists for quick inverted-index lookups. In *Proceedings of String Processing and Information Retrieval (SPIRE 2005)*, Lecture Notes in Computer Science. Springer-Verlag.

Borodin, A., G. O. Roberts, J. S. Rosenthal, and P. Tsaparas. 2001. Finding authorities and hubs from link structures on the world wide web. In *Proceedings of the 10th International World Wide Web Conference*, pp. 415–429.

Bourne, Charles P., and Donald F. Ford. 1961. A study of methods for systematically abbreviating english words and names. *Journal of the ACM* 8:538–552.

Bradley, Paul S., Usama M. Fayyad, and Cory Reina. 1998. Scaling clustering algorithms to large databases. In *KDD*, pp. 9–15.

Brill, Eric, and Robert C. Moore. 2000. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting of the ACL*, pp. 286–293.

Brin, Sergey, and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International World Wide Web Conference*, pp. 107–Ű117.

Brisaboa, Nieves R., Antonio Fari na, Gonzalo Navarro, and José R. Paramá. 2007. Lightweight natural language text compression. *Information Retrieval*. To appear.

Buckley, Chris, James Allan, and Gerard Salton. 1994a. Automatic routing and ad-hoc retrieval using smart: Trec 2. In *Proc. of the 2nd Text Retrieval Conference (TREC-2)*, pp. 45–55.

Buckley, Chris, Gerard Salton, and James Allan. 1994b. The effect of adding relevance information in a relevance feedback environment. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 292–300.

Buckley, Chris, Amit Singhal, Mandar Mitra, and Gerard Salton. 1996. New retrieval approaches using SMART: TREC 4. In D. K. Harman (ed.), *The Second Text REtrieval Conference (TREC-2)*, pp. 25–48.

Burges, Christopher J. C. 1998. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery* 2:121–167.

Burner, Mike. 1997. Crawling towards eternity: Building an archive of the world wide web. *Web Techniques Magazine* 2.

Bush, Vannevar. 1945. As we may think. *The Atlantic*.

Büttcher, Stefan, and Charles L. A. Clarke. 2005. Indexing time vs. query time: tradeoffs in dynamic information retrieval systems. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pp. 317–318, New York, NY, USA. ACM Press.

Büttcher, Stefan, and Charles L. A. Clarke. 2005. A security model for full-text file system search in multi-user environments. In *FAST*.

Büttcher, Stefan, Charles L. A. Clarke, and Brad Lushman. 2006. Hybrid index maintenance for growing text collections. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 356–363, New York, NY, USA. ACM Press.

Cao, Guihong, Jian-Yun Nie, and Jing Bai. 2005. Integrating word relationships into language models. In *SIGIR 2005*, pp. 298–305.

Carbonell, J., and J. Goldstein. 1998. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*.

Carletta, Jean. 1996. Assessing agreement on classification tasks: The kappa statistic. *Computational Linguistics* 22:249–254.

Carmel, David, Eitan Farchi, Yael Petruschka, and Aya Soffer. 2002. Automatic query refinement using lexical affinities with maximal information gain. In *SIGIR*.

Carmel, David, Yoelle S. Maarek, Matan Mandelbrod, Yosi Mass, and Aya Soffer. 2003. Searching xml documents via xml fragments. In *SIGIR '03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, pp. 151–158, New York, NY, USA. ACM Press.

Caruana, Rich, and Alexandru Niculescu-Mizil. 2006. An empirical comparison of supervised learning algorithms. In *ICML 2006*.

Cavnar, W. B., and J. M. Trenkle. 1994. N-gram-based text categorization. In *Proceedings of Third Annual Symposium on Document Analysis and Information Retrieval*, pp. 161–175.

Chakrabarti, S., B. Dom, D. Gibson, J. Kleinberg, P. Raghavan, and S. Rajagopalan. 1998. Automatic resource list compilation by analyzing hyperlink structure and associated text. In *Proceedings of the 7th International World Wide Web Conference*.

Chen, Hsin-Hsi, and Chuan-Jie Lin. 2000. A muitilingual news summarizer. In *COLING 2000*, pp. 159–165.

Chen, P.-H., C.-J. Lin, and B. Schölkopf. 2005. A tutorial on ν-support vector machines. *Applied Stochastic Models in Business and Industry* 21:111–136.

Cho, Junghoo, Hector Garcia-Molina, and Lawrence Page. 1998. Efficient crawling through url ordering. In *Proceedings of the Seventh International World Wide Web Conference*, pp. 161Ű–172.

Clarke, Charles L.A., Gordon V. Cormack, and Elizabeth A. Tudhope. 2000. Relevance ranking for one to three term queries. *Information Processing and Management* 36: 291–311.

Cleverdon, Cyril W. 1991. The significance of the Cranfield tests on index languages. In *Proceedings of the 14th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 3–12.

Cohen, William W., Robert E. Schapire, and Yoram Singer. 1998. Learning to order things. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla (eds.), *Advances in Neural Information Processing Systems*, volume 10. The MIT Press.

Cooper, Wm. S., Aitao Chen, and Fredric C. Gey. 1994. Full text retrieval based on probabilistic equations with coefficients fitted by logistic regression. In *The Second Text REtrieval Conference (TREC-2)*, pp. 57–66.

Cormen, Thomas H., Charles Eric Leiserson, and Ronald L. Rivest. 1990. *Introduction to Algorithms*. Cambridge MA: MIT Press.

Cover, Thomas M., and Peter E. Hart. 1967. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory* 13:21–27.

Crestani, F., M. Lalmas, C. J. Rijsbergen, and I. Campbell, 1998. Is this document relevant? ... probably: A survey of probabilistic models in information retrieval.

Croft, W. Bruce. 1978. A file organization for cluster-based retrieval. In *SIGIR '78: Proceedings of the 1st annual international ACM SIGIR conference on information storage and retrieval*, pp. 65–82, New York, NY, USA. ACM Press.

Croft, W. B., and D. J. Harper. 1979. Using probabilistic models of document retrieval without relevance information. *Journal of Documentation* 35:285–295.

Croft, W. Bruce, and John Lafferty (eds.). 2003. *Language Modeling for Information Retrieval*. New York: Springer.

Cutting, Douglas R., David R. Karger, and Jan O. Pedersen. 1993. Constant interaction-time scatter/gather browsing of very large document collections. In *SIGIR '93*, pp. 126–134.

Cutting, Douglas R., Jan O. Pedersen, David Karger, and John W. Tukey. 1992. Scatter/gather: A cluster-based approach to browsing large document collections. In *SIGIR '92*, pp. 318–329.

Damerau, Fred J. 1964. A technique for computer detection and correction of spelling errors. *Commun. ACM* 7:171–176.

Day, William H., and Herbert Edelsbrunner. 1984. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification* 1:1–24.

de Moura, Edleno Silva, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. 2000. Fast and flexible word searching on compressed text. *ACM Trans. Inf. Syst.* 18:113–139.

Dean, Jeffrey, and Sanjay Ghemawat. 2004. Mapreduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA.

Dempster, A.P., N.M. Laird, and D.B. Rubin. 1977. Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Statistical Society Series B* 39:1–38.

Di Eugenio, Barbara, and Michael Glass. 2004. The kappa statistic: A second look. *Computational Linguistics* 30:95–101.

Dietterich, Thomas G., and Ghulum Bakiri. 1995. Solving multiclass learning problems via error-correcting output codes. *J. Artif. Intell. Res. (JAIR)* 2:263–286.

Domingos, Pedro, and Michael Pazzani. 1997. On the optimality of the simple Bayesian classifier under zero-one loss. *Mach. Learn.* 29:103–130.

Duda, Richard O., Peter E. Hart, and David G. Stork. 2000. *Pattern Classification (2nd Edition)*. Wiley-Interscience.

Dumais, Susan T., and Hao Chen. 2000. Hierarchical classification of Web content. In *Proceedings of SIGIR-00, 23rd ACM International Conference on Research and Development in Information Retrieval*, pp. 256–263.

Dumais, S. T., J. Platt, D. Heckerman, and M Sahami. 1998. Inductive learning algorithms and representations for text categorization. In *Proceedings of the Seventh International Conference on Information and Knowledge Management (CIKM-98)*.

Dunning, Ted. 1993. Accurate methods for the statistics of surprise and coincidence. *Computational Linguistics* 19:61–74.

Dunning, Ted. 1994. Statistical identification of language. Technical Report 94-273, Computing Research Laboratory, New Mexico State University.

El-Hamdouchi, A., and P. Willett. 1986. Hierarchic document classification using ward's clustering method. In *SIGIR '86: Proceedings of the 9th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 149–156, New York, NY, USA. ACM Press.

Elias, Peter. 1975. Universal code word sets and representations of the integers. *IEEE Transactions on Information Theory* 21:194–203.

Fallows, Deborah, 2004. The internet and daily life. Pew Internet and American Life Project.

Fayyad, Usama M., Cory Reina, and Paul S. Bradley. 1998. Initialization of iterative refinement clustering algorithms. In *KDD*, pp. 194–198.

Forman, George. 2006. Tackling concept drift by temporal inductive transfer. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 252–259, New York, NY, USA. ACM Press.

Friedman, Jerome H. 1997. On bias, variance, 0/1–loss, and the curse-of-dimensionality. *Data Mining and Knowledge Discovery* 1:55–77.

Friedman, Nir, and Moises Goldszmidt. 1996. Building classifiers using bayesian networks. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 1277–1284.

Fuhr, Norbert. 1989. Optimum polynomial retrieval funcions based on the probability ranking principle. *ACM Transactions on Information Systems* 7:183–204.

Fuhr, Norbert. 1992. Probabilistic models in information retrieval. *The Computer Journal* 35:243–255.

Norbert Fuhr, Norbert Gövert, Gabriella Kazai, and Mounia Lalmas (eds.). 2003. *INitiative for the Evaluation of XML Retrieval (INEX). Proceedings of the First INEX Workshop. Dagstuhl, Germany, December 8–11, 2002*, ERCIM Workshop Proceedings, Sophia Antipolis, France. ERCIM. http://www.ercim.org/publication/ws-proceedings/INEX2002.pdf.

Fuhr, Norbert, and Kai Großjohann. 2001. Xirql: a query language for information retrieval in xml documents. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 172–180, New York, NY, USA. ACM Press.

Fuhr, Norbert, and Ulrich Pfeifer. 1994. Probabilistic information retrieval as a combination of abstraction, inductive learning, and probabilistic assumptions. *ACM Transactions on Information Systems* 12.

Gaertner, Thomas, John W. Lloyd, and Peter A. Flach. 2002. Kernels for structured data. In *12th International Conference on Inductive Logic Programming (ILP 2002)*, pp. 66–83.

Gao, Jianfeng, Mu Li, Chang-Ning Huang, and Andi Wu. 2005. Chinese word segmentation and named entity recognition: A pragmatic approach. *Computational Linguistics* 31:531–574.

Gao, Jianfeng, Jian-Yun Nie, Guangyuan Wu, and Guihong Cao. 2004. Dependence language model for information retrieval. In *SIGIR 2004*, pp. 170–177.

Garfield, Eugene. 1976. The permuterm subject index: An autobiographic review. *Journal of the American Society for Information Science* 27:288–291.

Geman, Stuart, Elie Bienenstock, and René Doursat. 1992. Neural networks and the bias/variance dilemma. *Neural Comput.* 4:1–58.

Glover, Eric, David M. Pennock, Steve Lawrence, and Robert Krovetz. 2002a. Inferring hierarchical descriptions. In *CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management*, pp. 507–514, New York, NY, USA. ACM Press.

Glover, Eric J., Kostas Tsioutsiouliklis, Steve Lawrence, David M. Pennock, and Gary W. Flake. 2002b. Using web structure for classifying and describing web pages. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pp. 562–569, New York, NY, USA. ACM Press.

Gövert, Norbert, and Gabriella Kazai. 2003. Overview of the INitiative for the Evaluation of XML retrieval (INEX) 2002. In Fuhr et al. (2003), pp. 1–17. In Fuhr et al. (2003).

Greiff, Warren R. 1998. A theory of term weighting based on exploratory data analysis. In *SIGIR 21*, pp. 11–19.

Grinstead, Charles M., and J. Laurie Snell. 1997. *Introduction to Probability*, 2 edition. Providence, RI: American Mathematical Society.

Hamerly, Greg, and Charles Elkan. 2003. Learning the k in k-means. In *NIPS*.

Hand, David J. 2006. Classifier technology and the illusion of progress. *Statistical Science* 21:1–14.

Harman, Donna. 1991. How effective is suffixing? *Journal of the American Society for Information Science* 42:7–15.

Harman, Donna. 1992. Relevance feedback revisited. In *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 1–10.

Harman, Donna, and Gerald Candela. 1990. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science* 41:581–589.

Harold, Elliotte Rusty, and Scott W. Means. 2004. *XML in a Nutshell, Third Edition*. O'Reilly Media, Inc.

Harter, Stephen P. 1998. Variations in relevance assessments and the measurement of retrieval effectiveness. *Journal of the American Society for Information Science* 47: 37–49.

Hastie, Trevor, Robert Tibshirani, and Jerome H. Friedman. 2001. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer Verlag.

Hatzivassiloglou, Vasileios, Luis Gravano, and Ankineedu Maganti. 2000. An investigation of linguistic features and clustering algorithms for topical document clustering. In *SIGIR '00: Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 224–231, New York, NY, USA. ACM Press.

Haveliwala, T., 2002. Topic-sensitive pagerank.

Haveliwala, T., 2003. Topic-sensitive pagerank: A context-sensitive ranking algorithm for web search.

Heaps, Harold S. 1978. *Information Retrieval: Computational and Theoretical Aspects*. New York: Academic Press.

Hearst, Marti A. 2006. Clustering versus faceted categories for information exploration. *Commun. ACM* 49:59–61.

Hearst, Marti A., and Jan O. Pedersen. 1996. Reexamining the cluster hypothesis. In *Proc. of SIGIR '96*, pp. 76–84, Zurich.

Heinz, Steffen, and Justin Zobel. 2003. Efficient single-pass index construction for text databases. *J. Am. Soc. Inf. Sci. Technol.* 54:713–729.

Heinz, Steffen, Justin Zobel, and Hugh E. Williams. 2002. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.* 20:192–223.

Hersh, W. R., C. Buckley, T. J. Loene, and D. Hicham. 1994. OHSUMED: An interactive retrieval evaluation and new large test collection for research. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information*, pp. 192–201.

Hiemstra, Djoerd. 1998. A linguistically motivated probabilistic model of information retrieval. In *ECDL 2*, pp. 569–584.

Hiemstra, Djoerd. 2000. A probabilistic justification for using tf.idf term weighting in information retrieval. *International Journal on Digital Libraries* 3:131–139.

Hiemstra, Djoerd, and Wessel Kraaij. 2005. A language-modeling approach to TREC. In Ellen M. Voorhees and Donna K. Harman (eds.), *TREC: Experiment and Evaluation in Information Retrieval*, pp. 373–395. Cambridge, MA: MIT Press.

Hirai, Jun, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. 2000. Efficient crawling through url ordering. In *Proceedings of the Ninth International World Wide Web Conference*, pp. 277–Ű293.

Huang, Yifen, and Tom M. Mitchell. 2006. Text clustering with extended user feedback. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 413–420, New York, NY, USA. ACM Press.

Hubert, Lawrence, and Phipps Arabie. 1985. Comparing partitions. *Journal of Classification* 2:193–218.

Hughes, Baden, Timothy Baldwin, Steven Bird, Jeremy Nicholson, and Andrew MacKinlay. 2006. Reconsidering language identification for written language resources. In *Proceedings 5th International Conference on Language Resources and Evaluation (LREC2006)*, pp. 485–488.

Hull, David. 1996. Stemming algorithms – A case study for detailed evaluation. *Journal of the American Society for Information Science* 47:70–84.

Ide, E. 1971. New experiments in relevance feedback. In Gerard Salton (ed.), *The SMART Retrieval System – Experiments in Automatic Document Processing*, pp. 337–354. Englewood Cliffs, NJ: Prentice-Hall.

Ittner, David J., David D. Lewis, and David D. Ahn. 1995. Text categorization of low quality images. In *Proceedings of SDAIR-95, 4th Annual Symposium on Document Analysis and Information Retrieval*, pp. 301–315, Las Vegas, US.

Jain, Anil K., and Richard C. Dubes. 1988. *Algorithms for Clustering Data*. Englewood Cliffs, NJ: Prentice Hall.

Jain, A. K., M. N. Murty, and P. J. Flynn. 1999. Data clustering: a review. *ACM Comput. Surv.* 31:264–323.

Jardine, N., and C. J. van Rijsbergen. 1971. The use of hierarchic clustering in information retrieval. *Information Storage and Retrieval* 7:217–240.

Jeh, G., and J. Widom. 2003. Scaling personalized web search. In *Proceedings of the 12th International World Wide Web Conference*.

Jensen, Finn V., and Finn B. Jensen. 2001. *Bayesian Networks and Decision Graphs*. Berlin: Springer Verlag.

Jeong, Byeong-Soo, and Edward Omiecinski. 1995. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel Distributed Systems* 6:142–153.

Ji, Xiang, and Wei Xu. 2006. Document clustering with prior knowledge. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 405–412, New York, NY, USA. ACM Press.

Jing, Hongyan. 2000. Sentence reduction for automatic text summarization. In *Proceedings of the sixth conference on Applied natural language processing*, pp. 310–315.

Joachims, Thorsten. 1998. Text categorization with support vector machines: learning with many relevant features. In Claire Nédellec and Céline Rouveirol (eds.), *Proceedings of ECML-98, 10th European Conference on Machine Learning*, number 1398 in Lecture Notes in Artificial Intelligence, pp. 137–142, Heidelberg. Springer Verlag.

Joachims, Thorsten. 2002a. *Learning to Classify Text using Support Vector Machines*. Kluwer.

Joachims, T. 2002b. Optimizing search engines using clickthrough data. In *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD)*.

Joachims, T., L. Granka, B. Pang, H. Hembrooke, and G. Gay. 2005. Accurately interpreting clickthrough data as implicit feedback. In *Proceedings of the Conference on Research and Development in Information Retrieval (SIGIR)*.

Jurafsky, Dan, and James H. Martin. 2000. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Englewood Cliffs, NJ: Prentice Hall.

Käki, Mika. 2005. Findex: search result categories help users when document ranking fails. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 131–140, New York, NY, USA. ACM Press.

Kamps, Jaap, Maarten Marx, Maarten de Rijke, and Börkur Sigurbjörnsson. 2002. The importance of morphological normalization for xml retrieval. In Fuhr et al. (2003), pp. 41–48. http://www.ercim.org/publication/ws-proceedings/INEX2002.pdf.

Kekäläinen, Jaana. 2005. Binary and graded relevance in IR evaluations—comparison of the effects on ranking of ir systems. *Information Processing and Management* 41: 1019–1033.

Kernighan, Mark D., Kenneth W. Church, and William A. Gale. 1990. A spelling correction program based on a noisy channel model. In *Proceedings of the 13th conference on International Conference On Computational Linguistics*, volume 2, pp. 205–210.

King, B. 1967. Step-wise clustering procedures. *J. Am. Stat. Assoc.* 69:86–101.

Kleinberg, Jon M. 1997. Two algorithms for nearest-neighbor search in high dimensions. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pp. 599–608, New York, NY, USA. ACM Press.

Kleinberg, Jon M. 1999. Authoritative sources in a hyperlinked environment. *Journal of the ACM* 46:604–632.

Knuth, Donald E. 1997. *The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition*. Addison-Wesley.

Koller, Daphne, and Mehran Sahami. 1997. Hierarchically classifying documents using very few words. In *Proceedings of the Fourteenth International Conference on Machine Learning (ICML-97)*, pp. 170–178.

Konheim, Alan G. 1981. *Cryptography: A Primer*. John Wiley & Sons.

Krippendorff, Klaus. 2003. *Content Analysis: An Introduction to its Methodology*. Sage.

Krovetz, Bob. 1995. *Word sense disambiguation for large text databases*. PhD thesis, University of Massachusetts Amherst.

Kukich, Karen. 1992. Technique for automatically correcting words in text. *ACM Comput. Surv.* 24:377–439.

Kupiec, Julian, Jan Pedersen, and Francine Chen. 1995. A trainable document summarizer. In *SIGIR '95*, pp. 68–73.

Lafferty, John, and Chengxiang Zhai. 2001. Document language models, query models, and risk minimization for information retrieval. In *SIGIR 2001*, pp. 111–119.

Lance, G. N., and W. T. Williams. 1967. A general theory of classificatory sorting strategies 1. Hierarchical systems. *Computer Journal* 9:373–380.

Larsen, Bjornar, and Chinatsu Aone. 1999. Fast and effective text mining using linear-time document clustering. In *KDD '99: Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 16–22, New York, NY, USA. ACM Press.

Lempel, R., and S. Moran. 2000. The stochastic approach for link-structure analysis (SALSA) and the TKC effect. *Computer Networks (Amsterdam, Netherlands: 1999)* 33: 387–401.

Lester, Nicholas, Alistair Moffat, and Justin Zobel. 2005. Fast on-line index construction by geometric partitioning. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pp. 776–783, New York, NY, USA. ACM Press.

Lester, Nicholas, Justin Zobel, and Hugh E. Williams. 2006. Efficient online index maintenance for contiguous inverted lists. *Information Processing & Management* 42: 916–933.

Levenshtein, V. 1965. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission* 1:8–17.

Lewis, David D. 1998. Naive (bayes) at forty: The independence assumption in information retrieval. In *ECML '98: Proceedings of the 10th European Conference on Machine Learning*, pp. 4–15, London, UK. Springer-Verlag.

Lewis, David D., Robert E. Schapire, James P. Callan, and Ron Papka. 1996. Training algorithms for linear text classifiers. In *SIGIR '96: Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 298–306, New York, NY, USA. ACM Press.

Lewis, David D., Yiming Yang, Tony G. Rose, and Fan Li. 2004. RCV1: A new benchmark collection for text categorization research. *J. Mach. Learn. Res.* 5:361–397.

Li, Fan, and Yiming Yang. 2003. A loss function analysis for classification methods in text categorization. In *ICML*, pp. 472–479.

Liddy, Elizabeth D. 2005. Automatic document retrieval, 2nd edition edition. In *Encyclopedia of Language and Linguistics*. Elsevier Press.

Lita, Lucian Vlad, Abe Ittycheriah, Salim Roukos, and Nanda Kambhatla. 2003. tRuEcasIng. In *ACL 41*, pp. 152–159.

Liu, Tie-Yan, Yiming Yang, Hao Wan, Hua-Jun Zeng, Zheng Chen, and Wei-Ying Ma. 2005. Support vector machines classification with very large scale taxonomy. *SIGKDD Explorations* 7:36–43.

Liu, Xiaoyong, and W. Bruce Croft. 2004. Cluster-based retrieval using language models. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 186–193, New York, NY, USA. ACM Press.

Lodhi, Huma, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. 2002. Text classification using string kernels. *Journal of Machine Learning Research* 2:419–444.

Lombard, M., J. Snyder-Duch, and C. C. Bracken. 2002. Content analysis in mass communication: Assessment and reporting of intercoder reliability. *Human Communication Research* 28:587–604.

Lovins, Julie Beth. 1968. Development of a stemming algorithm. *Translation and Computational Linguistics* 11:22–31.

Luhn, H.P. 1957. A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of Research and Development* 1.

Luhn, H.P. 1958. The automatic creation of literature abstracts. *IBM Journal of Research and Development* 2:159–165, 317.

Lunde, Ken. 1998. *CJKV Information Processing*. O'Reilly.

MacFarlane, A., J.A. McCann, and S.E. Robertson. 2000. Parallel search using partitioned inverted files. In *7th International Symposium on String Processing and Information Retrieval (SPIRE 2000)*, pp. 209–220.

Manning, Christopher D., and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. Cambridge, MA: MIT Press.

Mass, Yosi, Matan Mandelbrod, Einat Amitay, David Carmel, Yoëlle S. Maarek, and Aya Soffer. 2002. Juruxml - an xml retrieval system at inex'02. In Fuhr et al. (2003), pp. 73–80. http://www.ercim.org/publication/ws-proceedings/INEX2002.pdf.

McBryan, O. A. 1994. Genvl and wwww: Tools for taming the web. In *Proceedings of the First International World Wide Web Conference*.

McCallum, Andrew, and Kamal Nigam. 1998. A comparison of event models for Naive Bayes text classification. In *Working Notes of the 1998 AAAI/ICML Workshop on Learning for Text Categorization*, pp. 41–48.

McCallum, Andrew Kachites. 1996. Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering. http://www.cs.cmu.edu/ mccallum/bow.

McKeown, Kathleen, and Dragomir R. Radev. 1995. Generating summaries of multiple news articles. In *SIGIR '95: Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 74–82, New York, NY, USA. ACM Press.

McKeown, Kathleen R., Regina Barzilay, David Evans, Vasileios Hatzivassiloglou, Judith L. Klavans, Ani Nenkova, Carl Sable, Barry Schiffman, and Sergey Sigelman. 2002. Tracking and summarizing news on a daily basis with Columbia's Newsblaster. In *Proceedings of 2002 Human Language Technology Conference (HLT)*.

Melnik, Sergey, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. 2001. Building a distributed full-text index for the web. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pp. 396–406, New York, NY, USA. ACM Press.

Miller, David R. H., Tim Leek, and Richard M. Schwartz. 1999. A hidden Markov model information retrieval system. In *SIGIR 22*, pp. 214–221.

Minsky, Marvin Lee, and Seymour Papert (eds.). 1988. *Perceptrons: An introduction to computational geometry*. Cambridge, MA: MIT Press. Expanded edition.

Moffat, Alistair, and Justin Zobel. 1996. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.* 14:349–379.

Mooers, Calvin. 1961. From a point of view of mathematical etc. techniques. In R. A. Fairthorne (ed.), *Towards information retrieval*, pp. xvii–xxiii. London: Butterworths.

Murtagh, Fionn. 1983. A survey of recent advances in hierarchical clustering algorithms. *Computer Journal* 26:354–359.

Najork, Marc, and Allan Heydon. 2001. High-performanceweb crawling. Technical Report 173, Compaq Systems Research Center.

Najork, Marc, and Allan Heydon. 2002. High-performance web crawling. In Panos Pardalos James Abello and Mauricio Resende (eds.), *Handbook of Massive Data Sets*, chapter 2. Kluwer Academic Publishers.

Newsam, S., B. Sumengen, and B. S. Manjunath. 2001. Category-based image retrieval. In *IEEE International Conference on Image Processing, Special Session on Multimedia Indexing, Browsing and Retrieval*, volume 3, pp. 596–599.

Ng, Andrew Y., and Michael I. Jordan. 2001. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *NIPS*, pp. 841–848.

Ng, Andrew Y., Alice X. Zheng, and Michael I. Jordan. 2001. Link analysis, eigenvectors and stability. In *IJCAI*, pp. 903–910.

Page, Lawrence, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1998. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project.

Paice, Chris D. 1990. Another stemmer. *SIGIR Forum* 24:56–61.

Papineni, Kishore. 2001. Why inverse document frequency? In *NAACL 2*, pp. 1–8.

Pelleg, Dan, and Andrew Moore. 2000. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pp. 727–734, San Francisco. Morgan Kaufmann.

Persin, Michael, Justin Zobel, and Ron Sacks-Davis. 1996. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.* 47:749–764.

Peterson, James L. 1980. Computer programs for detecting and correcting spelling errors. *Commun. ACM* 23:676–687.

Picca, Davide, Benoît Curdy, and François Bavaud. 2006. Non-linear correspondence analysis in text retrieval: a kernel view. In *Proceedings of JADT*.

Ponte, Jay M., and W. Bruce Croft. 1998. A language modeling approach to information retrieval. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 275–281.

Popescul, Alexandrin, and Lyle H. Ungar. 2000. Automatic labeling of document clusters. unpublished.

Porter, Martin F. 1980. An algorithm for suffix stripping. *Program* 14:130–137.

Pugh, William. 1990. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM* 33:668–676.

Qiu, Yonggang, and H.P. Frei. 1993. Concept based query expansion. In *SIGIR 16*, pp. 160–169.

R Development Core Team. 2005. *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Radev, Dragomir R., Sasha Blair-Goldensohn, Zhu Zhang, and Revathi Sundara Raghavan. 2001. Interactive, domain-independent identification and summarization of topically related news articles. In *Proceedings, 5th European Conference on Research and Advanced Technology for Digital Libraries (ECDL 2001)*, pp. 225–238.

Rahm, Erhard, and Philip A. Bernstein. 2001. A survey of approaches to automatic schema matching. *VLDB Journal: Very Large Data Bases* 10:334–350.

Rand, William M. 1971. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association* 66:846–850.

Rasmussen, Edie. 1992. Clustering algorithms. In William B. Frakes and Ricardo Baeza-Yates (eds.), *Information Retrieval: Data Structures and Algorithms*, pp. 419–442. Englewood Cliffs, NJ: Prentice Hall.

Ribeiro-Neto, Berthier, Edleno S. Moura, Marden S. Neubert, and Nivio Ziviani. 1999. Efficient distributed algorithms to build inverted files. In *SIGIR '99: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 105–112, New York, NY, USA. ACM Press.

Ribeiro-Neto, Berthier A., and Ramurti A. Barbosa. 1998. Query performance for tightly coupled distributed digital libraries. In *ACM conference on Digital Libraries*, pp. 182–190.

Ripley, B. D. 1996. *Pattern Recognition and Neural Networks*. Cambridge: Cambridge University Press.

Robertson, Stephen. 2005. How Okapi came to TREC. In E.M. Voorhees and D.K. Harman (eds.), *TREC: Experiments and Evaluation in Information Retrieval*, pp. 287–299. MIT Press.

Robertson, S.E., and K. Spärck Jones. 1976a. Relevance weighting of search terms. *Journal of the American Society for Information Science* 27:129–146.

Robertson, S.E., and K. Spärck Jones. 1976b. Relevance weighting of search terms. *Journal of the American Society for Information Science* 27:129–146.

Rocchio, J. J. 1971. Relevance feedback in information retrieval. In Gerard Salton (ed.), *The SMART Retrieval System – Experiments in Automatic Document Processing*, pp. 313–323. Englewood Cliffs, NJ: Prentice-Hall.

Ruthven, Ian, and Mounia Lalmas. 2003. A survey on the use of relevance feedback for information access systems. *Knowledge Engineering Review* 18.

Sahoo, Nachiketa, Jamie Callan, Ramayya Krishnan, George Duncan, and Rema Padman. 2006. Incremental hierarchical clustering of text documents. In *CIKM '06: Proceedings of the 15th ACM international conference on Information and knowledge management*, pp. 357–366, New York, NY, USA. ACM Press.

Salton, Gerard (ed.). 1971. *The SMART Retrieval System – Experiments in Automatic Document Processing*. Englewood Cliffs, NJ: Prentice-Hall.

Salton, Gerard. 1975. *Dynamic information and library processing*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Salton, Gerard. 1989. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*. Reading, MA: Addison Wesley.

Salton, Gerald. 1991. The smart project in automatic document retrieval. In *Proceedings of the 14th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 356–358.

Salton, Gerard, and Chris Buckley. 1990. Improving retrieval performance by relevance feedback. *Journal of the American Society for Information Science* 41:288–297.

Salton, Gerard, and Chris Buckley. 1996. Term weighting approaches in automatic text retrieval. *Information Processing and Management* 32:431–443. Technical Report TR87-881, Department of Computer Science, Cornell University, 1987.

Saracevic, Tefko, and Paul Kantor. 1988. A study of information seeking and retrieving. ii: Users, questions and effectiveness. *Journal of the American Society for Information Science* 39:177–196.

Saracevic, Tefko, and Paul Kantor. 1996. A study of information seeking and retrieving iii: Searchers, searches, overlap. *Journal of the American Society for Information Science* 39:197–216.

Schapire, Robert E., Yoram Singer, and Amit Singhal. 1998. Boosting and Rocchio applied to text filtering. In *SIGIR '98*, pp. 215–223.

Scholer, Falk, Hugh E. Williams, John Yiannis, and Justin Zobel. 2002. Compression of inverted indexes for fast query evaluation. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 222–229, New York, NY, USA. ACM Press.

Schölkopf, Bernhard, and Alexander J. Smola. 2001. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press.

Schütze, Hinrich. 1998. Automatic word sense discrimination. *Computational Linguistics* 24:97–124.

Schütze, Hinrich, David A. Hull, and Jan O. Pedersen. 1995. A comparison of classifiers and document representations for the routing problem. In *SIGIR*, pp. 229–237.

Schütze, Hinrich, and Craig Silverstein. 1997. Projections for efficient document clustering. In *Proc. of SIGIR '97*, pp. 74–81.

Schwarz, Gideon. 1978. Estimating the dimension of a model. *Annals of Statistics* 6: 461–464.

Sebastiani, Fabrizio. 2002. Machine learning in automated text categorization. *ACM Computing Surveys* 34:1–47.

Shawe-Taylor, John, and Nello Cristianini. 2004. *Kernel Methods for Pattern Analysis*. Cambridge University Press.

Shkapenyuk, Vladislav, and Torsten Suel. 2002. Design and implementation of a high-performance distributed web crawler. In *ICDE*.

Singhal, Amit, Chris Buckley, and Mandar Mitra. 1996. Pivoted document length normalization. In *ACM SIGIR*, pp. 21–29.

Singhal, Amit, Mandar Mitra, and Chris Buckley. 1997. Learning routing queries in a query zone. In *Proc. of SIGIR '97*, pp. 25–32.

Singitham, Pavan Kumar C., Mahathi S. Mahabhashyam, and Prabhakar Raghavan. 2004. Efficiency-quality tradeoffs for vector score aggregation. In *VLDB*, pp. 624–635.

Sneath, Peter H.A., and Robert R. Sokal. 1973. *Numerical Taxonomy: The Principles and Practice of Numerical Classification*. San Francisco: W.H. Freeman.

Snedecor, George Waddel, and William G. Cochran. 1989. *Statistical methods*. Iowa State University Press.

Song, Ruihua, Ji-Rong Wen, and Wei-Ying Ma. 2005. Viewing term proximity from a different perspective. Technical Report MSR-TR-2005-69, Microsoft Research.

Spärck Jones, Karen. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation* 28:11–21.

Spärck Jones, Karen. 2004. Language modelling's generative model: Is it rational? MS, Computer Laboratory, University of Cambridge. http://www.cl.cam.ac.uk/~ksj21/langmodnote4.pdf.

Spärck Jones, Karen, S. Walker, and Stephen E. Robertson. 2000. A probabilistic model of information retrieval: Development and comparative experiments. *Information Processing and Management* pp. 779–808, 809–840.

Spink, Amanda, Bernard J. Jansen, and H. Cenk Ozmultu. 2000. Use of query reformulation and relevance feedback by Excite users. *Internet Research: Electronic Networking Applications and Policy* 10:317–328.

Sproat, Richard, and Thomas Emerson. 2003. The first international Chinese word segmentation bakeoff. In *The Second SIGHAN Workshop on Chinese Language Processing*.

Sproat, Richard, William Gale, Chilin Shih, and Nancy Chang. 1996. A stochastic finite-state word-segmentation algorithm for Chinese. *Computational Linguistics* 22: 377–404.

Sproat, Richard William. 1992. *Morphology and computation*. Cambridge, MA: MIT Press.

Steinbach, Michael, George Karypis, and Vipin Kumar. 2000. A comparison of document clustering techniques. In *KDD Workshop on Text Mining*.

Strehl, Alexander. 2002. *Relationship-based Clustering and Cluster Ensembles for High-dimensional Data Mining*. PhD thesis, The University of Texas at Austin.

Taube, M., and H. Wooster (eds.). 1958. *Information storage and retrieval: Theory, systems, and devices*. New York: Columbia University Press.

Tibshirani, Robert, Guenther Walther, and Trevor Hastie. 2001. Estimating the number of clusters in a data set via the gap statistic. *J. Roy. Statist. Soc. Ser. B* 63:411–423.

Tomasic, Anthony, and Hector Garcia-Molina. 1993. Query processing and inverted indices in shared-nothing document information retrieval systems. *VLDB Journal* 2:243–275.

Tombros, Anastasios, Robert Villa, and C. J. Van Rijsbergen. 2002. The effectiveness of query-specific hierarchic clustering in information retrieval. *Inf. Process. Manage.* 38:559–582.

Toutanova, Kristina, and Robert C. Moore. 2002. Pronunciation modeling for improved spelling correction. In *ACL 2002*, pp. 144–151.

Treeratpituk, Pucktada, and Jamie Callan. 2006. An experimental study on automatically labeling hierarchical clusters using statistical features. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 707–708, New York, NY, USA. ACM Press.

Tseng, Huihsin, Pichuan Chang, Galen Andrew, Daniel Jurafsky, and Christopher Manning. 2005. A conditional random field word segmenter. In *Fourth SIGHAN Workshop on Chinese Language Processing*.

Turtle, Howard. 1994. Natural language vs. Boolean query evaluation: a comparison of retrieval performance. In *SIGIR 17*, pp. 212–220.

Turtle, Howard, and W. Bruce Croft. 1989. Inference networks for document retrieval. In *Proceedings of the 13th annual international ACM SIGIR conference on research and development in information retrieval*, pp. 1–24.

Turtle, Howard, and W. Bruce Croft. 1991. Evaluation of an inference network-based retrieval model. *ACM Transactions on Information Systems* 9:187–222.

Vaithyanathan, Shivakumar, and Byron Dom. 2000. Model-based hierarchical clustering. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pp. 599–608, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

van Rijsbergen, C. J. 1979. *Information Retrieval*. London: Butterworths. Second Edition.

Vapnik, Vladimir N. 1998. *Statistical Learning Theory*. Wiley-Interscience.

Voorhees, Ellen. 1998. Variations in relevance judgements and the measurement of retrieval effectiveness. In *Proceedings of the 21st annual international ACM SIGIR conference on research and development in information retrieval*, pp. 315–323.

Voorhees, Ellen M. 1985a. The cluster hypothesis revisited. In *Proc. of SIGIR '85*, pp. 188–196.

Voorhees, Ellen M. 1985b. The effectiveness and efficiency of agglomerative hierarchic clustering in document retrieval. Technical Report TR 85-705, Cornell.

Voorhees, Ellen M., and Donna Harman (eds.). 2005. *TREC: Experiment and Evaluation in Information Retrieval*. MIT press.

Ward, Jr., J. H. 1963. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association* 58:236–244.

Weigend, Andreas S., E. D. Wiener, and Jan O. Pedersen. 1999. Exploiting hierarchy in text categorization. *Information Retrieval* 1:193–216.

Willams, Hugh E., Justin Zobel, and Dirk Bahle. 2004. Fast phrase querying with combined indexes. *ACM Transactions on Information Systems* 22:573–594.

Williams, Hugh E., and Justin Zobel. 2005. Searchable words on the web. *Int. J. on Digital Libraries* 5:99–105.

Witten, Ian H., Alistair Moffat, and Timothy C. Bell. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd edition. San Francisco, CA: Morgan Kaufmann.

Xu, J., and W. B. Croft. 1996. Query expansion using local and global document analysis. In *SIGIR 19*, pp. 4–11.

Yang, Hui, and Jamie Callan. 2006. Near-duplicate detection by instance-level constrained clustering. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 421–428, New York, NY, USA. ACM Press.

Yang, Yiming. 1994. Expert network: effective and effcient learning from human decisions in text categorization and retrieval. In *SIGIR*, pp. 13–22.

Yang, Yiming. 1999. An evaluation of statistical approaches to text categorization. *Information Retrieval* 1:69–90.

Yang, Yiming, and Xin Liu. 1999. A re-examination of text categorization methods. In *SIGIR 22*, pp. 42–49.

Yang, Yiming, and Jan Pedersen. 1997. Feature selection in statistical learning of text categorization. In *ICML*.

Yu, Cong, Hong Qi, and H. V. Jagadish. 2002. Integration of ir into an xml database. In Fuhr et al. (2003), pp. 162–169. http://www.ercim.org/publication/ws-proceedings/INEX2002.pdf.

Zamir, Oren, and Oren Etzioni. 1999. Grouper: a dynamic clustering interface to web search results. In *WWW '99: Proceeding of the eighth international conference on World Wide Web*, pp. 1361–1374, New York, NY, USA. Elsevier North-Holland, Inc.

Zaragoza, Hugo, Djoerd Hiemstra, Michael Tipping, and Stephen Robertson. 2003. Bayesian extension to the language model for ad hoc information retrieval. In *SIGIR 2003*, pp. 4–9.

Zhai, Chengxiang, and John Lafferty. 2001a. Model-based feedback in the language modeling approach to information retrieval. In *Proceedings of the Tenth International Conference on Information and Knowledge Management (CIKM 2001)*.

Zhai, Chengxiang, and John Lafferty. 2001b. A study of smoothing methods for language models applied to ad hoc information retrieval. In *SIGIR 2001*, pp. 334–342.

Zhang, Tong, and Frank J. Oles. 2001. Text categorization based on regularized linear classification methods. *Information Retrieval* 4:5–31.

Zhao, Ying, and George Karypis. 2002. Evaluation of hierarchical clustering algorithms for document datasets. In *CIKM '02: Proceedings of the eleventh international conference on Information and knowledge management*, pp. 515–524, New York, NY, USA. ACM Press.

Zipf, George Kingsley. 1949. *Human Behavior and the Principle of Least Effort*. Cambridge MA: Addison-Wesley Press.

Zobel, J., and P. Dart, 1995. Finding approximate matches in large lexicons.

Index