4

Processing Text

"I was trying to comprehend the meaning of the words."

Spock, *Star Trek: The Final Frontier*

4.1 From Words to Terms

After gathering the text we want to search, the next step is to decide whether it should be modified or restructured in some way to simplify searching. The types of changes that are made at this stage are called *text transformation* or, more often, *text processing*. The goal of text processing is to convert the many forms in which words can occur into more consistent *index terms*. Index terms are the representation of the content of a document that are used for searching.

The simplest decision about text processing would be to not do it at all. A good example of this is the "find" feature in your favorite word processor. By the time you use the find command, the text you wish to search has already been gathered: it's on the screen. After you type the word you want to find, the word processor scans the document and tries to find the exact sequence of letters that you just typed. This feature is extremely useful, and nearly every text editing program can do this because users demand it.

The trouble is that exact text search is rather restrictive. The most annoying restriction is case-sensitivity: suppose you want to find "computer hardware", and there is a sentence in the document that begins with "Computer hardware". Your search query does not exactly match the text in the sentence, because the first letter of the sentence is capitalized. Fortunately, most word processors have an option for ignoring case during searches. You can think of this as a very rudimentary form of online text processing. Like most text processing techniques, ignoring case increases the probability that you will find a match for your query in the document.

Many search engines do not distinguish between uppercase and lowercase letters. However, they go much further. As we will see in this chapter, search engines can strip punctuation from words to make them easier to find. Words are split apart in a process called *tokenization*. Some words may be ignored entirely in order to make query processing more effective and efficient; this is called *stopping*. The system may use *stemming* to allow similar words (like "run" and "running") to match each other. Some documents, such as web pages, may have formatting changes (like bold or large text), or explicit *structure* (like titles, chapters, and captions) that can also be used by the system. Web pages also contain *links* to other web pages, which can be used to improve document ranking. All of these techniques are discussed in this chapter.

These text processing techniques are fairly simple, even though their effects on search results can be profound. None of these techniques involves the computer doing any kind of complex reasoning or understanding of the text. Search engines work because much of the meaning of text is captured by counts of word occurrences and co-occurrences,[1] especially when that data is gathered from the huge text collections available on the Web. Understanding the statistical nature of text is fundamental to understanding retrieval models and ranking algorithms, so we begin this chapter with a discussion of *text statistics*. More sophisticated techniques for *natural language processing* that involve syntactic and semantic analysis of text have been studied for decades, including their application to information retrieval, but to date have had little impact on ranking algorithms for search engines. These techniques are, however, being used for the task of question answering, which is described in Chapter 11. In addition, techniques involving more complex text processing are being used to identify additional index terms or features for search. *Information extraction* techniques for identifying people's names, organization names, addresses, and many other special types of features are discussed here, and *classification*, which can be used to identify semantic categories, is discussed in Chapter 9.

Finally, even though this book focuses on retrieving English documents, information retrieval techniques can be used with text in many different languages. In this chapter, we show how different languages require different types of text representation and processing.

[1] Word co-occurrence measures the number of times groups of words (usually pairs) occur together in documents. A *collocation* is the name given to a pair, group, or sequence of words that occur together more often than would be expected by chance. The *term association measures* that are used to find collocations are discussed in Chapter 6.

4.2 Text Statistics

Although language is incredibly rich and varied, it is also very predictable. There are many ways to describe a particular topic or event, but if the words that occur in many descriptions of an event are counted, then some words will occur much more frequently than others. Some of these frequent words, such as "and" or "the," will be common in the description of any event, but others will be characteristic of that particular event. This was observed as early as 1958 by Luhn, when he proposed that the significance of a word depended on its frequency in the document. Statistical models of word occurrences are very important in information retrieval, and are used in many of the core components of search engines, such as the ranking algorithms, query transformation, and indexing techniques. These models will be discussed in later chapters, but we start here with some of the basic models of word occurrence.

One of the most obvious features of text from a statistical point of view is that the distribution of word frequencies is very *skewed*. There are a few words that have very high frequencies and many words that have low frequencies. In fact, the two most frequent words in English ("the" and "of") account for about 10% of all word occurrences. The most frequent six words account for 20% of occurrences, and the most frequent 50 words are about 40% of all text! On the other hand, given a large sample of text, typically about one half of all the unique words in that sample occur only once. This distribution is described by *Zipf's law*,[2] which states that the frequency of the rth most common word is inversely proportional to r or, alternatively, the *rank* of a word times its frequency (f) is approximately a constant (k):

$$r \cdot f = k$$

We often want to talk about the probability of occurrence of a word, which is just the frequency of the word divided by the total number of word occurrences in the text. In this case, Zipf's law is:

$$r \cdot P_r = c$$

where P_r is the probability of occurrence for the rth ranked word, and c is a constant. For English, $c \approx 0.1$. Figure 4.1 shows the graph of Zipf's law with this constant. This clearly shows how the frequency of word occurrence falls rapidly after the first few most common words.

[2] Named after the American linguist George Kingsley Zipf.

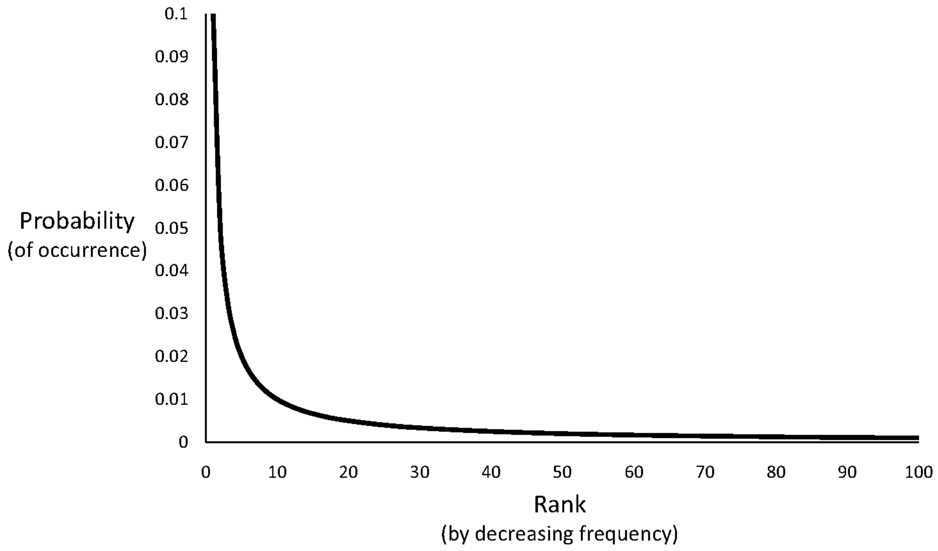


Fig. 4.1. Rank versus probability of occurrence for words assuming Zipf's law (rank × probability = 0.1)

To see how well Zipf's law predicts word occurrences in actual text collections, we will use the Associated Press collection of news stories from 1989 (called AP89) as an example. This collection was used in TREC evaluations for several years. Table 4.1 shows some statistics for the word occurrences in AP89. The vocabulary size is the number of unique words in the collection. Even in this relatively small collection, the vocabulary size is quite large (nearly 200,000 unique words). A large proportion of these words (70,000) occur only once. Words that occur once in a text corpus or book have long been regarded as important in text analysis, and have been given the special name of *Hapax Legomena*.[3]

Table 4.2 shows the 50 most frequent words from the AP89 collection, together with their frequencies, ranks, probability of occurrence (converted to a percentage of total occurrences), and the $r.P_r$ value. From this table, we can see

[3] The name was created by scholars studying the Bible. Since the 13th century, people have studied the word occurrences in the Bible and, of particular interest, created *concordances*, which are indexes of where words occur in the text. Concordances are the ancestors of the inverted files that are used in modern search engines. The first concordance was said to have required 500 monks to create.

Total documents	84,678
Total word occurrences	39,749,179
Vocabulary size	198,763
Words occurring > 1000 times	4,169
Words occurring once	70,064

Table 4.1. Statistics for the AP89 collection

that Zipf's law is quite accurate, in that the value of $r.P_r$ is approximately constant, and close to 0.1. The biggest variations are for some of the most frequent words. In fact, it is generally observed that Zipf's law is inaccurate for low and high ranks (high-frequency and low-frequency words). Table 4.3 gives some examples for lower-frequency words from AP89.

Figure 4.2 shows a log-log plot[4] of the $r.P_r$ values for all words in the AP89 collection. Zipf's law is shown as a straight line on this graph since $\log P_r = \log(c \cdot r^{-1}) = \log c - \log r$. This figure clearly shows how the predicted relationship breaks down at high ranks (approximately rank 10,000 and above). A number of modifications to Zipf's law have been proposed,[5] some of which have interesting connections to cognitive models of language.

It is possible to derive a simple formula for predicting the proportion of words with a given frequency from Zipf's law. A word that occurs n times has rank $r_n = k/n$. In general, more than one word may have the same frequency. We assume that the rank r_n is associated with the *last* of the group of words with the same frequency. In that case, the number of words with the same frequency n will be given by $r_n - r_{n+1}$, which is the rank of the last word in the group minus the rank of the last word of the previous group of words with a higher frequency (remember that higher-frequency words have lower ranks). For example, Table 4.4 has an example of a ranking of words in decreasing order of their frequency. The number of words with frequency 5,099 is the rank of the last member of that

[4] The x and y axes of a log-log plot show the logarithm of the values of x and y, not the values themselves.

[5] The most well-known is the derivation by the mathematician Benoit Mandelbrot (the same person who developed fractal geometry), which is $(r + \beta)^\alpha \cdot P_r = \gamma$, where β, α, and γ are parameters that can be tuned for a particular text. In the case of the AP89 collection, however, the fit for the frequency data is not noticeably better than the Zipf distribution.

Word	*Freq.*	*r*	$P_r(\%)$	$r.P_r$	*Word*	*Freq*	*r*	$P_r(\%)$	$r.P_r$
the	2,420,778	1	6.49	0.065	has	136,007	26	0.37	0.095
of	1,045,733	2	2.80	0.056	are	130,322	27	0.35	0.094
to	968,882	3	2.60	0.078	not	127,493	28	0.34	0.096
a	892,429	4	2.39	0.096	who	116,364	29	0.31	0.090
and	865,644	5	2.32	0.120	they	111,024	30	0.30	0.089
in	847,825	6	2.27	0.140	its	111,021	31	0.30	0.092
said	504,593	7	1.35	0.095	had	103,943	32	0.28	0.089
for	363,865	8	0.98	0.078	will	102,949	33	0.28	0.091
that	347,072	9	0.93	0.084	would	99,503	34	0.27	0.091
was	293,027	10	0.79	0.079	about	92,983	35	0.25	0.087
on	291,947	11	0.78	0.086	i	92,005	36	0.25	0.089
he	250,919	12	0.67	0.081	been	88,786	37	0.24	0.088
is	245,843	13	0.65	0.086	this	87,286	38	0.23	0.089
with	223,846	14	0.60	0.084	their	84,638	39	0.23	0.089
at	210,064	15	0.56	0.085	new	83,449	40	0.22	0.090
by	209,586	16	0.56	0.090	or	81,796	41	0.22	0.090
it	195,621	17	0.52	0.089	which	80,385	42	0.22	0.091
from	189,451	18	0.51	0.091	we	80,245	43	0.22	0.093
as	181,714	19	0.49	0.093	more	76,388	44	0.21	0.090
be	157,300	20	0.42	0.084	after	75,165	45	0.20	0.091
were	153,913	21	0.41	0.087	us	72,045	46	0.19	0.089
an	152,576	22	0.41	0.090	percent	71,956	47	0.19	0.091
have	149,749	23	0.40	0.092	up	71,082	48	0.19	0.092
his	142,285	24	0.38	0.092	one	70,266	49	0.19	0.092
but	140,880	25	0.38	0.094	people	68,988	50	0.19	0.093

Table 4.2. Most frequent 50 words from AP89

Word	*Freq.*	*r*	$P_r(\%)$	$r.P_r$
assistant	5,095	1,021	.013	0.13
sewers	100	17,110	.000256	0.04
toothbrush	10	51,555	.000025	0.01
hazmat	1	166,945	.000002	0.04

Table 4.3. Low-frequency words from AP89

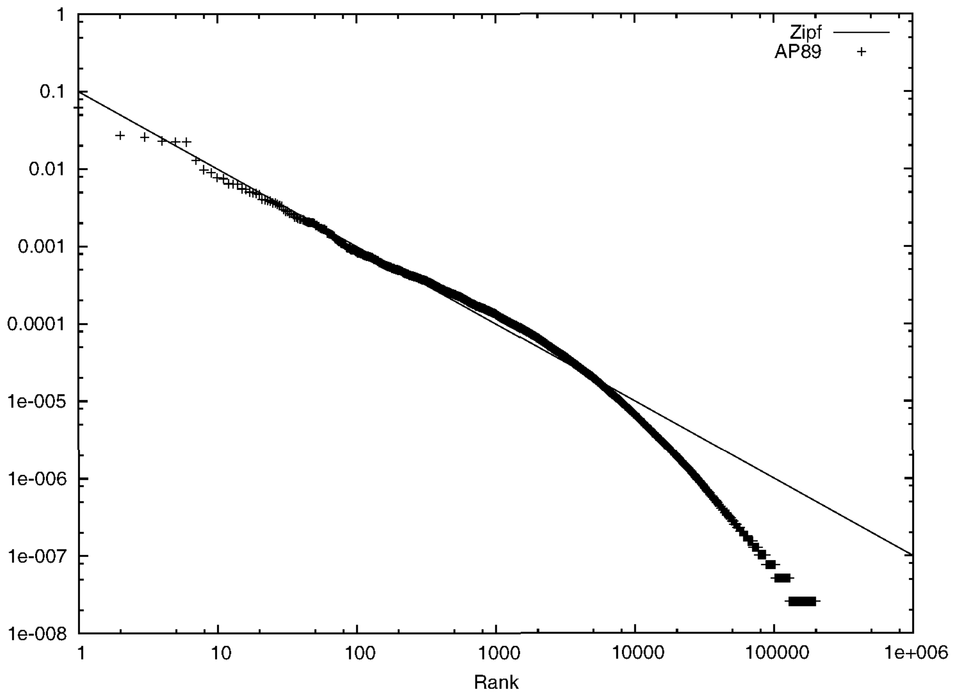


Fig. 4.2. A log-log plot of Zipf's law compared to real data from AP89. The predicted relationship between probability of occurrence and rank breaks down badly at high ranks.

group ("chemical") minus the rank of the last member of the previous group with higher frequency ("summit"), which is $1006 - 1002 = 4$.

Rank	*Word*	*Frequency*
1000	concern	5,100
1001	spoke	5,100
1002	summit	5,100
1003	bring	5,099
1004	star	5,099
1005	immediate	5,099
1006	chemical	5,099
1007	african	5,098

Table 4.4. Example word frequency ranking

Given that the number of words with frequency n is $r_n - r_{n+1} = k/n - k/(n+1) = k/n(n+1)$, then the proportion of words with this frequency can be found by dividing this number by the total number of words, which will be the rank of the last word with frequency 1. The rank of the last word in the vocabulary is $k/1 = k$. The proportion of words with frequency n, therefore, is given by $1/n(n+1)$. This formula predicts, for example, that $1/2$ of the words in the vocabulary will occur once. Table 4.5 compares the predictions of this formula with real data from a different TREC collection.

Number of Occurrences (n)	*Predicted Proportion (1/n(n+1))*	*Actual Proportion*	*Actual Number of Words*
1	0.500	0.402	204,357
2	0.167	0.132	67,082
3	0.083	0.069	35,083
4	0.050	0.046	23,271
5	0.033	0.032	16,332
6	0.024	0.024	12,421
7	0.018	0.019	9,766
8	0.014	0.016	8,200
9	0.011	0.014	6,907
10	0.009	0.012	5,893

Table 4.5. Proportions of words occurring n times in 336,310 documents from the TREC Volume 3 corpus. The total vocabulary size (number of unique words) is 508,209.

4.2.1 Vocabulary Growth

Another useful prediction related to word occurrence is *vocabulary growth*. As the size of the corpus grows, new words occur. Based on the assumption of a Zipf distribution for words, we would expect that the number of new words that occur in a given amount of new text would decrease as the size of the corpus increases. New words will, however, always occur due to sources such as invented words (think of all those drug names and start-up company names), spelling errors, product numbers, people's names, email addresses, and many others. The relationship between the size of the corpus and the size of the vocabulary was found empirically by Heaps (1978) to be:

$$v = k \cdot n^{\beta}$$

where v is the vocabulary size for a corpus of size n words, and k and β are parameters that vary for each collection. This is sometimes referred to as *Heaps' law*. Typical values for k and β are often stated to be $10 \leq k \leq 100$ and $\beta \approx 0.5$. Heaps' law predicts that the number of new words will increase very rapidly when the corpus is small and will continue to increase indefinitely, but at a slower rate for larger corpora. Figure 4.3 shows a plot of vocabulary growth for the AP89 collection compared to a graph of Heaps' law with k = 62.95 and β = 0.455. Clearly, Heaps' law is a good fit. The parameter values are similar for many of the other TREC news collections. As an example of the accuracy of this prediction, if the first 10,879,522 words of the AP89 collection are scanned, Heaps' law predicts that the number of unique words will be 100,151, whereas the actual value is 100,024. Predictions are much less accurate for small numbers of words ($<$ 1,000).

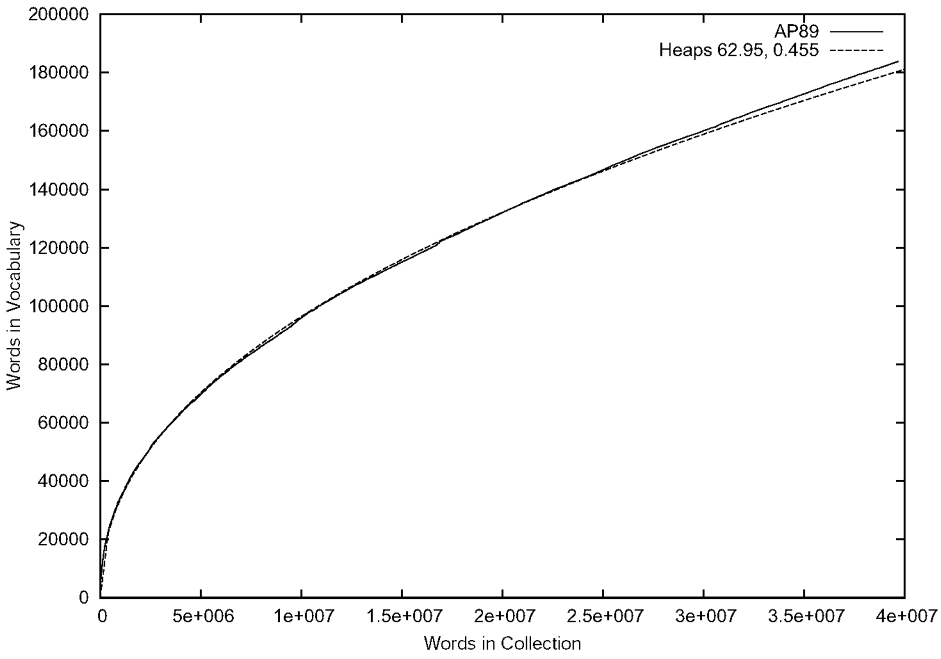


Fig. 4.3. Vocabulary growth for the TREC AP89 collection compared to Heaps' law

Web-scale collections are considerably larger than the AP89 collection. The AP89 collection contains about 40 million words, but the (relatively small) TREC Web collection GOV2[6] contains more than 20 *billion* words. With that many words, it seems likely that the number of new words would eventually drop to near zero and Heaps' law would not be applicable. It turns out this is not the case. Figure 4.4 shows a plot of vocabulary growth for GOV2 together with a graph of Heaps' law with $k = 7.34$ and $\beta = 0.648$. This data indicates that the number of unique words continues to grow steadily even after reaching 30 million. This has significant implications for the design of search engines, which will be discussed in Chapter 5. Heaps' law provides a good fit for this data, although the parameter values are very different than those for other TREC collections and outside the boundaries established as typical with these and other smaller collections.

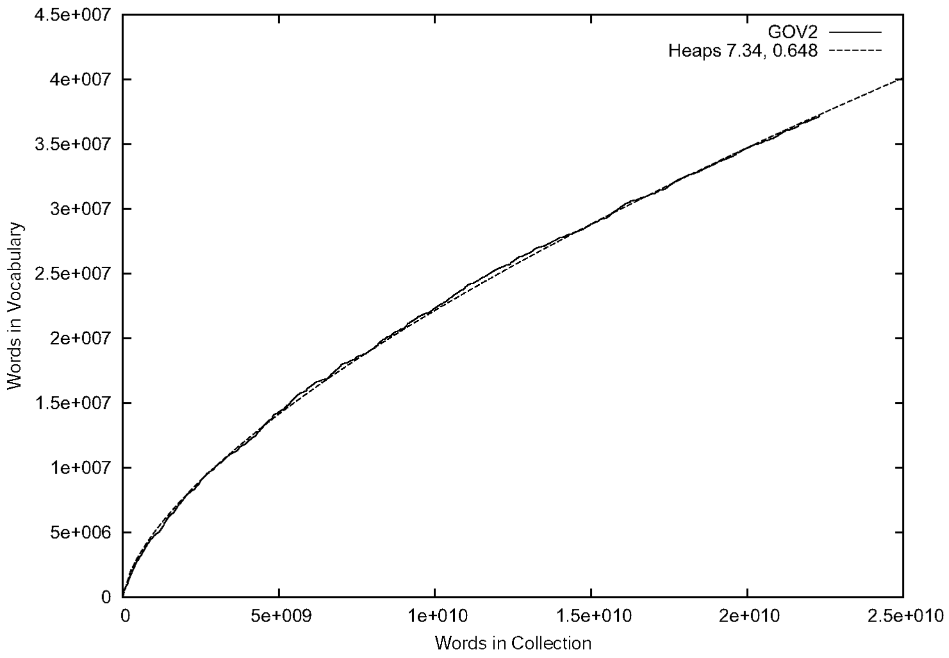


Fig. 4.4. Vocabulary growth for the TREC GOV2 collection compared to Heaps' law

[6] Web pages crawled from websites in the .gov domain during early 2004. See section 8.2 for more details.

4.2.2 Estimating Collection and Result Set Sizes

Word occurrence statistics can also be used to estimate the size of the results from a web search. All web search engines have some version of the query interface shown in Figure 4.5, where immediately after the query ("tropical fish aquarium" in this case) and before the ranked list of results, an estimate of the total number of results is given. This is typically a very large number, and descriptions of these systems always point out that it is just an estimate. Nevertheless, it is always included.

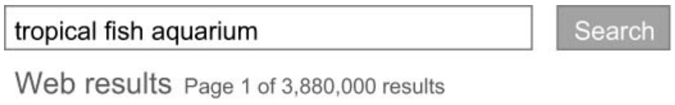


Fig. 4.5. Result size estimate for web search

To estimate the size of a result set, we first need to define "results." For the purposes of this estimation, a result is any document (or web page) that contains *all* of the query words. Some search applications will rank documents that do not contain all the query words, but given the huge size of the Web, this is usually not necessary. If we assume that words occur *independently* of each other, then the probability of a document containing all the words in the query is simply the product of the probabilities of the individual words occurring in a document. For example, if there are three query words a, b, and c, then:

$$P(a \cap b \cap c) = P(a) \cdot P(b) \cdot P(c)$$

where $P(a \cap b \cap c)$ is the joint probability, or the probability that all three words occur in a document, and $P(a)$, $P(b)$, and $P(c)$ are the probabilities of each word occurring in a document. A search engine will always have access to the number of documents that a word occurs in (f_a, f_b, and f_c),[7] and the number of documents in the collection (N), so these probabilities can easily be estimated as $P(a) = f_a/N$, $P(b) = f_b/N$, and $P(c) = f_c/N$. This gives us

$$f_{abc} = N \cdot f_a/N \cdot f_b/N \cdot f_c/N = (f_a \cdot f_b \cdot f_c)/N^2$$

where f_{abc} is the estimated size of the result set.

[7] Note that these are *document occurrence frequencies*, not the total number of word occurrences (there may be many occurrences of a word in a document).

Word(s)	*Document Frequency*	*Estimated Frequency*
tropical	120,990	
fish	1,131,855	
aquarium	26,480	
breeding	81,885	
tropical fish	18,472	5,433
tropical aquarium	1,921	127
tropical breeding	5,510	393
fish aquarium	9,722	1,189
fish breeding	36,427	3,677
aquarium breeding	1,848	86
tropical fish aquarium	1,529	6
tropical fish breeding	3,629	18

Table 4.6. Document frequencies and estimated frequencies for word combinations (assuming independence) in the GOV2 Web collection. Collection size (N) is 25,205,179.

Table 4.6 gives document occurrence frequencies for the words "tropical", "fish", "aquarium", and "breeding", and for combinations of those words in the TREC GOV2 Web collection. It also gives the estimated size of the frequencies of the combinations based on the independence assumption. Clearly, this assumption does not lead to good estimates for result size, especially for combinations of three words. The problem is that the words in these combinations do not occur independently of each other. If we see the word "fish" in a document, for example, then the word "aquarium" is more likely to occur in this document than in one that does not contain "fish".

Better estimates are possible if word co-occurrence information is also available from the search engine. Obviously, this would give exact answers for two-word queries. For longer queries, we can improve the estimate by not assuming independence. In general, for three words

$$P(a \cap b \cap c) = P(a \cap b) \cdot P(c|(a \cap b))$$

where $P(a \cap b)$ is the probability that the words a and b co-occur in a document, and $P(c|(a \cap b))$ is the probability that the word c occurs in a document *given* that the words a and b occur in the document.[8] If we have co-occurrence information,

[8] This is called a *conditional* probability.

we can approximate this probability using either $P(c|a)$ or $P(c|b)$, whichever is the largest. For the example query "tropical fish aquarium" in Table 4.6, this means we estimate the result set size by multiplying the number of documents containing both "tropical" and "aquarium" by the probability that a document contains "fish" given that it contains "aquarium", or:

$$\begin{aligned} f_{tropical \cap fish \cap aquarium} &= f_{tropical \cap aquarium} \cdot f_{fish \cap aquarium} / f_{aquarium} \\ &= 1921 \cdot 9722/26480 = 705 \end{aligned}$$

Similarly, for the query "tropical fish breeding":

$$\begin{aligned} f_{tropical \cap fish \cap breeding} &= f_{tropical \cap breeding} \cdot f_{fish \cap breeeding} / f_{breeding} \\ &= 5510 \cdot 36427/81885 = 2451 \end{aligned}$$

These estimates are much better than the ones produced assuming independence, but they are still too low. Rather than storing even more information, such as the number of occurrences of word triples, it turns out that reasonable estimates of result size can be made using just word frequency and the size of the *current* result set. Search engines estimate the result size because they do not rank all the documents that contain the query words. Instead, they rank a much smaller subset of the documents that are likely to be the most relevant. If we know the proportion of the total documents that have been ranked (s) and the number of documents found that contain all the query words (C), we can simply estimate the result size as C/s, which assumes that the documents containing all the words are distributed uniformly.[9] The proportion of documents processed is measured by the proportion of the documents containing the least frequent word that have been processed, since all results must contain that word.

For example, if the query "tropical fish aquarium" is used to rank GOV2 documents in the Galago search engine, after processing 3,000 out of the 26,480 documents that contain "aquarium", the number of documents containing all three words is 258. This gives a result size estimate of 258/(3,000 ÷ 26,480) = 2,277. After processing just over 20% of the documents, the estimate is 1,778 (compared to the actual figure of 1,529). For the query "tropical fish breeding", the estimates after processing 10% and 20% of the documents that contain "breeding" are 4,076

[9] We are also assuming *document-at-a-time* processing, where the inverted lists for all query words are processed at the same time, giving complete document scores (see Chapter 5).

and 3,762 (compared to 3,629). These estimates, as well as being quite accurate, do not require knowledge of the total number of documents in the collection.

Estimating the total number of documents stored in a search engine is, in fact, of significant interest to both academia (how big is the Web?) and business (which search engine has better coverage of the Web?). A number of papers have been written about techniques to do this, and one of these is based on the concept of word independence that we used before. If a and b are two words that occur independently, then

$$f_{ab}/N = f_a/N \cdot f_b/N$$

and

$$N = (f_a \cdot f_b)/f_{ab}$$

To get a reasonable estimate of N, the two words should be independent and, as we have seen from the examples in Table 4.6, this is often not the case. We can be more careful about the choice of query words, however. For example, if we use the word "lincoln" (document frequency 771,326 in GOV2), we would expect the words in the query "tropical lincoln" to be more independent than the word pairs in Table 4.6 (since the former are less semantically related). The document frequency of "tropical lincoln" in GOV2 is 3,018, which means we can estimate the size of the collection as N = (120,990 · 771,326)/3,018 = 30,922,045. This is quite close to the actual number of 25,205,179.

4.3 Document Parsing

4.3.1 Overview

Document parsing involves the recognition of the content and structure of text documents. The primary content of most documents is the words that we were counting and modeling using the Zipf distribution in the previous section. Recognizing each word occurrence in the sequence of characters in a document is called *tokenizing* or *lexical analysis*. Apart from these words, there can be many other types of content in a document, such as metadata, images, graphics, code, and tables. As mentioned in Chapter 2, metadata is information about a document that is not part of the text content. Metadata content includes document attributes such as date and author, and, most importantly, the *tags* that are used by *markup languages* to identify document components. The most popular markup languages are HTML (Hypertext Markup Language) and XML (Extensible Markup Language).

The parser uses the tags and other metadata recognized in the document to interpret the document's structure based on the syntax of the markup language (*syntactic analysis*) and to produce a representation of the document that includes both the structure and content. For example, an HTML parser interprets the structure of a web page as specified using HTML tags, and creates a Document Object Model (DOM) representation of the page that is used by a web browser. In a search engine, the output of a document parser is a representation of the content and structure that will be used for creating indexes. Since it is important for a search index to represent every document in a collection, a document parser for a search engine is often more tolerant of syntax errors than parsers used in other applications.

In the first part of our discussion of document parsing, we focus on the recognition of the tokens, words, and phrases that make up the content of the documents. In later sections, we discuss separately the important topics related to document structure, namely markup, links, and extraction of structure from the text content.

4.3.2 Tokenizing

Tokenizing is the process of forming words from the sequence of characters in a document. In English text, this appears to be simple. In many early systems, a "word" was defined as any sequence of alphanumeric characters of length 3 or more, terminated by a space or other special character. All uppercase letters were also converted to lowercase.[10] This means, for example, that the text

Bigcorp's 2007 bi-annual report showed profits rose 10%.

would produce the following sequence of tokens:

bigcorp 2007 annual report showed profits rose

Although this simple tokenizing process was adequate for experiments with small test collections, it does not seem appropriate for most search applications or even experiments with TREC collections, because too much information is discarded. Some examples of issues involving tokenizing that can have significant impact on the effectiveness of search are:

[10] This is sometimes referred to as *case folding*, *case normalization*, or *downcasing*.

- Small words (one or two characters) can be important in some queries, usually in combinations with other words. For example, xp, ma, pm, ben e king, el paso, master p, gm, j lo, world war II.[11]
- Both hyphenated and non-hyphenated forms of many words are common. In some cases the hyphen is not needed. For example, e-bay, wal-mart, active-x, cd-rom, t-shirts. At other times, hyphens should be considered either as part of the word or a word separator. For example, winston-salem, mazda rx-7, e-cards, pre-diabetes, t-mobile, spanish-speaking.
- Special characters are an important part of the tags, URLs, code, and other important parts of documents that must be correctly tokenized.
- Capitalized words can have different meaning from lowercase words. For example, "Bush" and "Apple".
- Apostrophes can be a part of a word, a part of a possessive, or just a mistake. For example, rosie o'donnell, can't, don't, 80's, 1890's, men's straw hats, master's degree, england's ten largest cities, shriner's.
- Numbers can be important, including decimals. For example, nokia 3250, top 10 courses, united 93, quicktime 6.5 pro, 92.3 the beat, 288358 (yes, this was a real query; it's a patent number).
- Periods can occur in numbers, abbreviations (e.g., "I.B.M.", "Ph.D."), URLs, ends of sentences, and other situations.

From these examples, tokenizing seems more complicated than it first appears. The fact that these examples come from queries also emphasizes that the text processing for queries *must* be the same as that used for documents. If different tokenizing processes are used for queries and documents, many of the index terms used for documents will simply not match the corresponding terms from queries. Mistakes in tokenizing become obvious very quickly through retrieval failures.

To be able to incorporate the range of language processing required to make matching effective, the tokenizing process should be both simple and flexible. One approach to doing this is for the first pass of tokenizing to focus entirely on identifying markup or tags in the document. This could be done using a tokenizer and parser designed for the specific markup language used (e.g., HTML), but it should accommodate syntax errors in the structure, as mentioned previously. A second pass of tokenizing can then be done on the appropriate parts of the document structure. Some parts that are not used for searching, such as those containing HTML code, will be ignored in this pass.

[11] These and other examples were taken from a small sample of web queries.

Given that nearly everything in the text of a document can be important for some query, the tokenizing rules have to convert most of the content to searchable tokens. Instead of trying to do everything in the tokenizer, some of the more difficult issues, such as identifying word variants or recognizing that a string is a name or a date, can be handled by separate processes, including stemming, information extraction, and query transformation. Information extraction usually requires the full form of the text as input, including capitalization and punctuation, so this information must be retained until extraction has been done. Apart from this restriction, capitalization is rarely important for searching, and text can be reduced to lowercase for indexing. This does not mean that capitalized words are not used in queries. They are, in fact, used quite often, but in queries where the capitalization does not reduce ambiguity and so does not impact effectiveness. Words such as "Apple" that are often used in examples (but not so often in real queries) can be handled by query reformulation techniques (Chapter 6) or simply by relying on the most popular pages (section 4.5).

If we take the view that complicated issues are handled by other processes, the most general strategy for hyphens, apostrophes, and periods would be to treat them as word terminators (like spaces). It is important that all the tokens produced are indexed, including single characters such as "s" and "o". This will mean, for example, that the query[12] `"o'connor"` is equivalent to `"o connor"`, `"bob's"` is equivalent to `"bob s"`, and `"rx-7"` is equivalent to `"rx 7"`. Note that this will also mean that a word such as "rx7" will be a different token than "rx-7" and therefore will be indexed separately. The task of relating the queries `rx 7`, `rx7`, and `rx-7` will then be handled by the query transformation component of the search engine.

On the other hand, if we rely entirely on the query transformation component to make the appropriate connections or inferences between words, there is the risk that effectiveness could be lowered, particularly in applications where there is not enough data for reliable query expansion. In these cases, more rules can be incorporated into the tokenizer to ensure that the tokens produced by the query text will match the tokens produced from document text. For example, in the case of TREC collections, a rule that tokenizes all words containing apostrophes by the string without the apostrophe is very effective. With this rule, "O'Connor" would be tokenized as "oconnor" and "Bob's" would produce the token "bobs". Another effective rule for TREC collections is to tokenize all abbreviations con-

[12] We assume the common syntax for web queries where `"<words>"` means match exactly the phrase contained in the quotes.

taining periods as the string without periods. An abbreviation in this case is any string of alphabetic single characters separated by periods. This rule would tokenize "I.B.M." as "ibm", but "Ph.D." would still be tokenized as "ph d".

In summary, the most general tokenizing process will involve first identifying the document structure and then identifying words in text as any sequence of alphanumeric characters, terminated by a space or special character, with everything converted to lowercase. This is not much more complicated than the simple process we described at the start of the section, but it relies on information extraction and query transformation to handle the difficult issues. In many cases, additional rules are added to the tokenizer to handle some of the special characters, to ensure that query and document tokens will match.

4.3.3 Stopping

Human language is filled with function words: words that have little meaning apart from other words. The most popular—"the," "a," "an," "that," and "those"—are *determiners*. These words are part of how we describe nouns in text, and express concepts like location or quantity. Prepositions, such as "over," "under," "above," and "below," represent relative position between two nouns.

Two properties of these function words cause us to want to treat them in a special way in text processing. First, these function words are extremely common. Table 4.2 shows that nearly all of the most frequent words in the AP89 collection fall into this category. Keeping track of the quantity of these words in each document requires a lot of disk space. Second, both because of their commonness and their function, these words rarely indicate anything about document relevance on their own. If we are considering individual words in the retrieval process and not phrases, these function words will help us very little.

In information retrieval, these function words have a second name: *stopwords*. We call them stopwords because text processing stops when one is seen, and they are thrown out. Throwing out these words decreases index size, increases retrieval efficiency, and generally improves retrieval effectiveness.

Constructing a stopword list must be done with caution. Removing too many words will hurt retrieval effectiveness in particularly frustrating ways for the user. For instance, the query "to be or not to be" consists entirely of words that are usually considered stopwords. Although not removing stopwords may cause some problems in ranking, removing stopwords can cause perfectly valid queries to return no results.

A stopword list can be constructed by simply using the top n (e.g., 50) most frequent words in a collection. This can, however, lead to words being included that are important for some queries. More typically, either a standard stopword list is used,[13] or a list of frequent words and standard stopwords is manually edited to remove any words that may be significant for a particular application. It is also possible to create stopword lists that are customized for specific parts of the document structure (also called *fields*). For example, the words "click", "here", and "privacy" may be reasonable stopwords to use when processing anchor text.

If storage space requirements allow, it is best to at least index all words in the documents. If stopping is required, the stopwords can always be removed from queries. By keeping the stopwords in the index, there will be a number of possible ways to execute a query with stopwords in it. For instance, many systems will remove stopwords from a query unless the word is preceded by a plus sign (+). If keeping stopwords in an index is not possible because of space requirements, as few as possible should be removed in order to maintain maximum flexibility.

4.3.4 Stemming

Part of the expressiveness of natural language comes from the huge number of ways to convey a single idea. This can be a problem for search engines, which rely on matching words to find relevant documents. Instead of restricting matches to words that are identical, a number of techniques have been developed to allow a search engine to match words that are semantically related. *Stemming*, also called *conflation*, is a component of text processing that captures the relationships between different variations of a word. More precisely, stemming reduces the different forms of a word that occur because of *inflection* (e.g., plurals, tenses) or *derivation* (e.g., making a verb into a noun by adding the suffix -ation) to a common stem.

Suppose you want to search for news articles about Mark Spitz's Olympic swimming career. You might type "mark spitz swimming" into a search engine. However, many news articles are usually summaries of events that have already happened, so they are likely to contain the word "swam" instead of "swimming." It is the job of the stemmer to reduce "swimming" and "swam" to the same stem (probably "swim") and thereby allow the search engine to determine that there is a match between these two words.

[13] Such as the one distributed with the Lemur toolkit and included with Galago.

In general, using a stemmer for search applications with English text produces a small but noticeable improvement in the quality of results. In applications involving highly inflected languages, such as Arabic or Russian, stemming is a crucial part of effective search.

There are two basic types of stemmers: *algorithmic* and *dictionary-based*. An algorithmic stemmer uses a small program to decide whether two words are related, usually based on knowledge of word suffixes for a particular language. By contrast, a dictionary-based stemmer has no logic of its own, but instead relies on pre-created dictionaries of related terms to store term relationships.

The simplest kind of English algorithmic stemmer is the *suffix-s* stemmer. This kind of stemmer assumes that any word ending in the letter "s" is plural, so cakes $\rightarrow$ cake, dogs $\rightarrow$ dog. Of course, this rule is not perfect. It cannot detect many plural relationships, like "century" and "centuries". In very rare cases, it detects a relationship where it does not exist, such as with "I" and "is". The first kind of error is called a *false negative*, and the second kind of error is called a *false positive*.[14]

More complicated algorithmic stemmers reduce the number of false negatives by considering more kinds of suffixes, such as -ing or -ed. By handling more suffix types, the stemmer can find more term relationships; in other words, the false negative rate is reduced. However, the false positive rate (finding a relationship where none exists) generally increases.

The most popular algorithmic stemmer is the *Porter stemmer*.[15] This has been used in many information retrieval experiments and systems since the 1970s, and a number of implementations are available. The stemmer consists of a number of steps, each containing a set of rules for removing suffixes. At each step, the rule for the longest applicable suffix is executed. Some of the rules are obvious, whereas others require some thought to work out what they are doing. As an example, here are the first two parts of step 1 (of 5 steps):

Step 1a:

- Replace *sses* by *ss* (e.g., stresses $\rightarrow$ stress).
- Delete *s* if the preceding word part contains a vowel not immediately before the *s* (e.g., gaps $\rightarrow$ gap but gas $\rightarrow$ gas).
- Replace *ied* or *ies* by *i* if preceded by more than one letter, otherwise by *ie* (e.g., ties $\rightarrow$ tie, cries $\rightarrow$ cri).

[14] These terms are used in any binary decision process to describe the two types of errors. This includes evaluation (Chapter 8) and classification (Chapter 9).

[15] http://tartarus.org/martin/PorterStemmer/

- If suffix is ***us*** or ***ss*** do nothing (e.g., stress → stress).

Step 1b:

- Replace ***eed***, ***eedly*** by ***ee*** if it is in the part of the word after the first non-vowel following a vowel (e.g., agreed → agree, feed → feed).
- Delete ***ed***, ***edly***, ***ing***, ***ingly*** if the preceding word part contains a vowel, and then if the word ends in ***at***, ***bl***, or ***iz*** add ***e*** (e.g., fished → fish, pirating → pirate), or if the word ends with a double letter that is not ***ll***, ***ss***, or ***zz***, remove the last letter (e.g., falling→ fall, dripping → drip), or if the word is short, add ***e*** (e.g., hoping → hope).
- Whew!

The Porter stemmer has been shown to be effective in a number of TREC evaluations and search applications. It is difficult, however, to capture all the subtleties of a language in a relatively simple algorithm. The original version of the Porter stemmer made a number of errors, both false positives and false negatives. Table 4.7 shows some of these errors. It is easy to imagine how confusing "execute" with "executive" or "organization" with "organ" could cause significant problems in the ranking. A more recent form of the stemmer (called Porter2)[16] fixes some of these problems and provides a mechanism to specify exceptions.

False positives	*False negatives*
organization/organ	european/europe
generalization/generic	cylinder/cylindrical
numerical/numerous	matrices/matrix
policy/police	urgency/urgent
university/universe	create/creation
addition/additive	analysis/analyses
negligible/negligent	useful/usefully
execute/executive	noise/noisy
past/paste	decompose/decomposition
ignore/ignorant	sparse/sparsity
special/specialized	resolve/resolution
head/heading	triangle/triangular

Table 4.7. Examples of errors made by the original Porter stemmer. False positives are pairs of words that have the same stem. False negatives are pairs that have different stems.

[16] http://snowball.tartarus.org

A dictionary-based stemmer provides a different approach to the problem of stemming errors. Instead of trying to detect word relationships from letter patterns, we can store lists of related words in a large dictionary. Since these word lists can be created by humans, we can expect that the false positive rate will be very low for these words. Related words do not even need to look similar; a dictionary stemmer can recognize that "is," "be," and "was" are all forms of the same verb. Unfortunately, the dictionary cannot be infinitely long, so it cannot react automatically to new words. This is an important problem since language is constantly evolving. It is possible to build stem dictionaries automatically by statistical analysis of a text corpus. Since this is particularly useful when stemming is used for query expansion, we discuss this technique in section 6.2.1.

Another strategy is to combine an algorithmic stemmer with a dictionary-based stemmer. Typically, irregular words such as the verb "to be" are the oldest in the language, while new words follow more regular grammatical conventions. This means that newly invented words are likely to work well with an algorithmic stemmer. A dictionary can be used to detect relationships between common words, and the algorithmic stemmer can be used for unrecognized words.

A well-known example of this hybrid approach is the *Krovetz stemmer* (Krovetz, 1993). This stemmer makes constant use of a dictionary to check whether the word is valid. The dictionary in the Krovetz stemmer is based on a general English dictionary but also uses exceptions that are generated manually. Before being stemmed, the dictionary is checked to see whether a word is present; if it is, it is either left alone (if it is in the general dictionary) or stemmed based on the exception entry. If the word is not in the dictionary, it is checked for a list of common inflectional and derivational suffixes. If one is found, it is removed and the dictionary is again checked to see whether the word is present. If it is not found, the ending of the word may be modified based on the ending that was removed. For example, if the ending -ies is found, it is replaced by -ie and checked in the dictionary. If it is found in the dictionary, the stem is accepted; otherwise the ending is replaced by *y*. This will result in calories → calorie, for example. The suffixes are checked in a sequence (for example, plurals before -ion endings), so multiple suffixes may be removed.

The Krovetz stemmer has a lower false positive rate than the Porter stemmer, but also tends to have a higher false negative rate, depending on the size of the exception dictionaries. Overall, the effectiveness of the two stemmers is comparable when used in search evaluations. The Krovetz stemmer has the additional advantage of producing stems that, in most cases, are full words, whereas the Porter

Original text:
Document will describe marketing strategies carried out by U.S. companies for their agricultural chemicals, report predictions for market share of such chemicals, or report market statistics for agrochemicals, pesticide, herbicide, fungicide, insecticide, fertilizer, predicted sales, market share, stimulate demand, price cut, volume of sales.

Porter stemmer:
document describ market strategi carri compani agricultur chemic report predict market share chemic report market statist agrochem pesticid herbicid fungicid insecticid fertil predict sale market share stimul demand price cut volum sale

Krovetz stemmer:
document describe marketing strategy carry company agriculture chemical report prediction market share chemical report market statistic agrochemic pesticide herbicide fungicide insecticide fertilizer predict sale stimulate demand price cut volume sale

Fig. 4.6. Comparison of stemmer output for a TREC query. Stopwords have also been removed.

stemmer often produces stems that are word fragments. This is a concern if the stems are used in the search interface.

Figure 4.6 compares the output of the Porter and Krovetz stemmers on the text of a TREC query. The output of the Krovetz stemmer is similar in terms of which words are reduced to the same stems, although "marketing" is not reduced to "market" because it was in the dictionary. The stems produced by the Krovetz stemmer are mostly words. The exception is the stem "agrochemic", which occurred because "agrochemical" was not in the dictionary. Note that text processing in this example has removed stopwords, including single characters. This resulted in the removal of "U.S." from the text, which could have significant consequences for some queries. This can be handled by better tokenization or information extraction, as we discuss in section 4.6.

As in the case of stopwords, the search engine will have more flexibility to answer a broad range of queries if the document words are not stemmed but instead are indexed in their original form. Stemming can then be done as a type of query expansion, as explained in section 6.2.1. In some applications, both the full words and their stems are indexed, in order to provide both flexibility and efficient query processing times.

We mentioned earlier that stemming can be particularly important for some languages, and have virtually no impact in others. Incorporating language-specific stemming algorithms is one of the most important aspects of customizing, or *internationalizing*, a search engine for multiple languages. We discuss other aspects of internationalization in section 4.7, but focus on the stemming issues here.

As an example, Table 4.8 shows some of the Arabic words derived from the same root. A stemming algorithm that reduced Arabic words to their roots would clearly not work (there are less than 2,000 roots in Arabic), but a broad range of prefixes and suffixes must be considered. Highly inflectional languages like Arabic have many word variants, and stemming can make a large difference in the accuracy of the ranking. An Arabic search engine with high-quality stemming can be more than 50% more effective, on average, at finding relevant documents than a system without stemming. In contrast, improvements for an English search engine vary from less than 5% on average for large collections to about 10% for small, domain-specific collections.

kitab	*a book*
kitabi	*my book*
alkitab	*the book*
kitabuki	*your book* (f)
kitabuka	*your book (m)*
kitabuhu	*his book*
kataba	*to write*
maktaba	*library, bookstore*
maktab	*office*

Table 4.8. Examples of words with the Arabic root **ktb**

Fortunately, stemmers for a number of languages have already been developed and are available as open source software. For example, the Porter stemmer is available in French, Spanish, Portuguese, Italian, Romanian, German, Dutch, Swedish, Norwegian, Danish, Russian, Finnish, Hungarian, and Turkish.[17] In addition, the statistical approach to building a stemmer that is described in section 6.2.1 can be used when only a text corpus is available.

[17] http://snowball.tartarus.org/

4.3.5 Phrases and N-grams

Phrases are clearly important in information retrieval. Many of the two- and three-word queries submitted to search engines are phrases, and finding documents that contain those phrases will be part of any effective ranking algorithm. For example, given the query "black sea", documents that contain that phrase are much more likely to be relevant than documents containing text such as "the sea turned black". Phrases are more precise than single words as topic descriptions (e.g., "tropical fish" versus "fish") and usually less ambiguous (e.g., "rotten apple" versus "apple"). The impact of phrases on retrieval can be complex, however. Given a query such as "fishing supplies", should the retrieved documents contain exactly that phrase, or should they get credit for containing the words "fish", "fishing", and "supplies" in the same paragraph, or even the same document? The details of how phrases affect ranking will depend on the specific retrieval model that is incorporated into the search engine, so we will defer this discussion until Chapter 7. From the perspective of text processing, the issue is whether phrases should be identified at the same time as tokenizing and stemming, so that they can be indexed for faster query processing.

There are a number of possible definitions of a phrase, and most of them have been studied in retrieval experiments over the years. Since a phrase has a grammatical definition, it seems reasonable to identify phrases using the syntactic structure of sentences. The definition that has been used most frequently in information retrieval research is that a phrase is equivalent to a simple *noun phrase*. This is often restricted even further to include just sequences of nouns, or adjectives followed by nouns. Phrases defined by these criteria can be identified using a *part-of-speech (POS) tagger*. A POS tagger marks the words in a text with labels corresponding to the part-of-speech of the word in that context. Taggers are based on statistical or rule-based approaches and are trained using large corpora that have been manually labeled. Typical tags that are used to label the words include NN (singular noun), NNS (plural noun), VB (verb), VBD (verb, past tense), VBN (verb, past participle), IN (preposition), JJ (adjective), CC (conjunction, e.g., "and", "or"), PRP (pronoun), and MD (modal auxiliary, e.g., "can", "will").

Figure 4.7 shows the output of a POS tagger for the TREC query text used in Figure 4.6. This example shows that the tagger can identify phrases that are sequences of nouns, such as "marketing/NN strategies/NNS", or adjectives followed by nouns, such as "agricultural/JJ chemicals/NNS". Taggers do, however, make mistakes. The words "predicted/VBN sales/NNS" would not be identified as a noun phrase, because "predicted" is tagged as a verb.

Original text:

Document will describe marketing strategies carried out by U.S. companies for their agricultural chemicals, report predictions for market share of such chemicals, or report market statistics for agrochemicals, pesticide, herbicide, fungicide, insecticide, fertilizer, predicted sales, market share, stimulate demand, price cut, volume of sales.

Brill tagger:

```
Document/NN will/MD describe/VB marketing/NN strategies/NNS carried/VBD out/IN by/IN U.S./NNP
companies/NNS for/IN their/PRP agricultural/JJ chemicals/NNS ,/, report/NN predictions/NNS for/IN
market/NN share/NN of/IN such/JJ chemicals/NNS ,/, or/CC report/NN market/NN statistics/NNS for/IN
agrochemicals/NNS ,/, pesticide/NN ,/, herbicide/NN ,/, fungicide/NN ,/, insecticide/NN ,/, fertilizer/NN
,/, predicted/VBN sales/NNS ,/, market/NN share/NN ,/, stimulate/VB demand/NN ,/, price/NN cut/NN
,/, volume/NN of/IN sales/NNS ./.
```

Fig. 4.7. Output of a POS tagger for a TREC query

Table 4.9 shows the high-frequency simple noun phrases from a TREC corpus consisting mainly of news stories and a corpus of comparable size consisting of all the 1996 patents issued by the United States Patent and Trademark Office (PTO). The phrases were identified by POS tagging. The frequencies of the example phrases indicate that phrases are used more frequently in the PTO collection, because patents are written in a formal, legal style with considerable repetition. There were 1,100,000 phrases in the TREC collection that occurred more than five times, and 3,700,000 phrases in the PTO collection. Many of the TREC phrases are proper nouns, such as “los angeles” or “european union”, or are topics that will be important for retrieval, such as “peace process” and “human rights”. Two phrases are associated with the format of the documents (“article type”, “end recording”). On the other hand, most of the high-frequency phrases in the PTO collection are standard terms used to describe all patents, such as“present invention” and “preferred embodiment”, and relatively few are related to the content of the patents, such as “carbon atoms” and “ethyl acetate”. One of the phrases, “group consisting”, was the result of a frequent tagging error.

Although POS tagging produces reasonable phrases and is used in a number of applications, in general it is too slow to be used as the basis for phrase indexing of large collections. There are simpler and faster alternatives that are just as effective. One approach is to store word position information in the indexes and use this information to identify phrases only when a query is processed. This provides considerable flexibility in that phrases can be identified by the user or by using POS tagging on the query, and they are not restricted to adjacent groups of

TREC data		Patent data	
Frequency	*Phrase*	*Frequency*	*Phrase*
65824	united states	975362	present invention
61327	article type	191625	u.s. pat
33864	los angeles	147352	preferred embodiment
18062	hong kong	95097	carbon atoms
17788	north korea	87903	group consisting
17308	new york	81809	room temperature
15513	san diego	78458	seq id
15009	orange county	75850	brief description
12869	prime minister	66407	prior art
12799	first time	59828	perspective view
12067	soviet union	58724	first embodiment
10811	russian federation	56715	reaction mixture
9912	united nations	54619	detailed description
8127	southern california	54117	ethyl acetate
7640	south korea	52195	example 1
7620	end recording	52003	block diagram
7524	european union	46299	second embodiment
7436	south africa	41694	accompanying drawings
7362	san francisco	40554	output signal
7086	news conference	37911	first end
6792	city council	35827	second end
6348	middle east	34881	appended claims
6157	peace process	33947	distal end
5955	human rights	32338	cross-sectional view
5837	white house	30193	outer surface

Table 4.9. High-frequency noun phrases from a TREC collection and U.S. patents from 1996

words. The identification of syntactic phrases is replaced by testing word proximity constraints, such as whether two words occur within a specified text window. We describe position indexing in Chapter 5 and retrieval models that exploit word proximity in Chapter 7.

In applications with large collections and tight constraints on response time, such as web search, testing word proximities at query time is also likely to be too slow. In that case, we can go back to identifying phrases in the documents during text processing, but use a much simpler definition of a phrase: any sequence of n words. This is also known as an *n-gram*. Sequences of two words are called *bigrams*, and sequences of three words are called *trigrams*. Single words are called *unigrams*. N-grams have been used in many text applications and we will mention them again frequently in this book, particularly in association with *language models* (section 7.3). In this discussion, we are focusing on *word* n-grams, but *character* n-grams are also used in applications such as OCR, where the text is "noisy" and word matching can be difficult (section 11.6). Character n-grams are also used for indexing languages such as Chinese that have no word breaks (section 4.7). N-grams, both character and word, are generated by choosing a particular value for n and then moving that "window" forward one unit (character or word) at a time. In other words, n-grams *overlap*. For example, the word "tropical" contains the following character bigrams: tr, ro, op, pi, ic, ca, and al. Indexes based on n-grams are obviously larger than word indexes.

The more frequently a word n-gram occurs, the more likely it is to correspond to a meaningful phrase in the language. N-grams of all lengths form a Zipf distribution, with a few common phrases occurring very frequently and a large number occurring with frequency 1. In fact, the rank-frequency data for n-grams (which includes single words) fits the Zipf distribution better than words alone. Some of the most common n-grams will be made up of stopwords (e.g., "and the", "there is") and could be ignored, although as with words, we should be cautious about discarding information. Our previous example query "to be or not to be" could certainly make use of n-grams. We could potentially index all n-grams in a document text up to a specific length and make them available to the ranking algorithm. This would seem to be an extravagant use of indexing time and disk space because of the large number of possible n-grams. A document containing 1,000 words, for example, would contain 3,990 instances of word n-grams of length $2 \leq n \leq 5$. Many web search engines, however, use n-gram indexing because it provides a fast method of incorporating phrase features in the ranking.

Google recently made available a file of n-grams derived from web pages.[18] The statistics for this sample are shown in Table 4.10. An analysis of n-grams on the Web (Yang et al., 2007) found that "all rights reserved" was the most frequent trigram in English, whereas "limited liability corporation" was the most frequent in Chinese. In both cases, this was due to the large number of corporate sites, but

[18] http://googleresearch.blogspot.com/2006/08/all-our-n-gram-are-belong-to-you.html

it also indicates that n-grams are not dominated by common patterns of speech such as "and will be".

Number of tokens:	1,024,908,267,229
Number of sentences:	95,119,665,584
Number of unigrams:	13,588,391
Number of bigrams:	314,843,401
Number of trigrams:	977,069,902
Number of fourgrams:	1,313,818,354
Number of fivegrams:	1,176,470,663

Table 4.10. Statistics for the Google n-gram sample

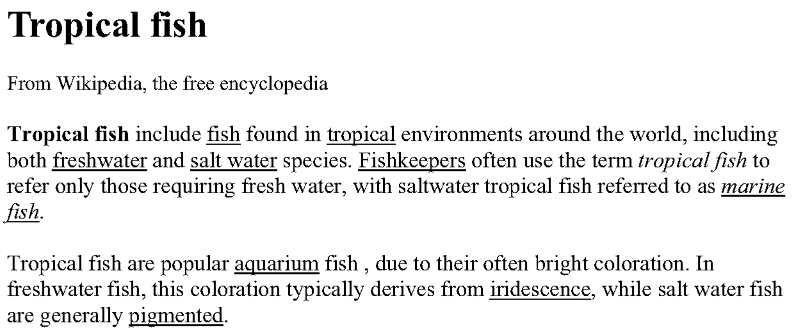
4.4 Document Structure and Markup

In database applications, the fields or attributes of database records are a critical part of searching. Queries are specified in terms of the required values of these fields. In some text applications, such as email or literature search, fields such as *author* and *date* will have similar importance and will be part of the query specification. In the case of web search, queries usually do not refer to document structure or fields, but that does not mean that this structure is unimportant. Some parts of the structure of web pages, indicated by HTML markup, are very significant features used by the ranking algorithm. The document parser must recognize this structure and make it available for indexing.

As an example, Figure 4.8 shows part of a web page for a Wikipedia[19] entry. The page has some obvious structure that could be used in a ranking algorithm. The main heading for the page, "tropical fish", indicates that this phrase is particularly important. The same phrase is also in bold and italics in the body of the text, which is further evidence of its importance. Other words and phrases are used as the *anchor text* for links and are likely to be good terms to represent the content of the page.

The HTML source for this web page (Figure 4.9) shows that there is even more structure that should be represented for search. Each field or *element* in HTML is indicated by a start tag (such as `<h1>`) and an optional end tag (e.g.,

[19] The Web encyclopedia, http://en.wikipedia.org/.



Tropical fish

From Wikipedia, the free encyclopedia

Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species. Fishkeepers often use the term *tropical fish* to refer only those requiring fresh water, with saltwater tropical fish referred to as *marine fish*.

Tropical fish are popular aquarium fish , due to their often bright coloration. In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

Fig. 4.8. Part of a web page from Wikipedia

`</h1>`).[20] Elements can also have attributes (with values), given by `attribute_name = "value"` pairs. The `<head>` element of an HTML document contains metadata that is not displayed by a browser. The metadata element for keywords (`<meta name="keywords"`) gives a list of words and phrases that can be used as additional content terms for the page. In this case, these are the titles of other Wikipedia pages. The `<title>` metadata element gives the title for the page (which is different from the main heading).

The `<body>` element of the document contains the content that is displayed. The main heading is indicated by the `<h1>` tag. Other headings, of different sizes and potentially different importance, would be indicated by `<h2>` through `<h6>` tags. Terms that should be displayed in bold or italic are indicated by `<b>` and `<i>` tags. Unlike typical database fields, these tags are primarily used for formatting and can occur many times in a document. They can also, as we have said, be interpreted as a tag indicating a word or phrase of some importance.

Links, such as `<a href="/wiki/Fish" title="Fish">fish</a>`, are very common. They are the basis of link analysis algorithms such as PageRank (Brin & Page, 1998), but also define the anchor text. Links and anchor text are of particular importance to web search and will be described in the next section. The `title` attribute for a link is used to provide extra information about that link, although in our example it is the words in the last part of the URL for the associated Wikipedia page. Web search engines also make use of the URL of a page as a source of additional metadata. The URL for this page is:

```
http://en.wikipedia.org/wiki/Tropical_fish
```

[20] In XML the end tag is not optional.

The fact that the words “tropical” and “fish” occur in the URL will increase the importance of those words for this page. The depth of a URL (i.e., the number of directories deep the page is) can also be important. For example, the URL www.ibm.com is more likely to be the home page for IBM than a page with the URL:

www.pcworld.com/businesscenter/article/698/ibm_buys_apt!

```
<html>
<head>
<meta name="keywords" content="Tropical fish, Airstone, Albinism, Algae eater,
Aquarium, Aquarium fish feeder, Aquarium furniture, Aquascaping, Bath treatment
(fishkeeping),Berlin Method, Biotope" />
...
<title>Tropical fish - Wikipedia, the free encyclopedia</title>
</head>
<body>
 ...
<h1 class="firstHeading">Tropical fish</h1>
...
<p><b>Tropical fish</b> include <a href="/wiki/Fish" title="Fish">fish</a> found in <a
href="/wiki/Tropics" title="Tropics">tropical</a> environments around the world,
including both <a href="/wiki/Fresh_water" title="Fresh water">freshwater</a> and <a
href="/wiki/Sea_water" title="Sea water">salt water</a> species. <a
href="/wiki/Fishkeeping" title="Fishkeeping">Fishkeepers</a> often use the term
<i>tropical fish</i> to refer only those requiring fresh water, with saltwater tropical fish
referred to as <i><a href="/wiki/List_of_marine_aquarium_fish_species" title="List of
marine aquarium fish species">marine fish</a></i>.</p>
<p>Tropical fish are popular <a href="/wiki/Aquarium" title="Aquarium">aquarium</a>
fish , due to their often bright coloration. In freshwater fish, this coloration typically
derives from <a href="/wiki/Iridescence" title="Iridescence">iridescence</a>, while salt
water fish are generally <a href="/wiki/Pigment" title="Pigment">pigmented</a>.</p>
...
</body></html>
```

Fig. 4.9. HTML source for example Wikipedia page

In HTML, the element types are predefined and are the same for all documents. XML, in contrast, allows each application to define what the element types are and what tags are used to represent them. XML documents can be described by a *schema*, similar to a database schema. XML elements, consequently, are more closely tied to the semantics of the data than HTML elements. Search applications often use XML to record *semantic annotations* in the documents that are produced by information extraction techniques, as described in section 4.6. A document parser for these applications would record the annotations, along with the other document structure, and make them available for indexing.

The query language XQuery[21] has been defined by the database community for searching structured data described using XML. XQuery supports queries that specify both structural and content constraints, which raises the issue of whether a database or information retrieval approach is better for building a search engine for XML data. We discuss this topic in more detail in section 11.4, but the general answer is that it will depend on the data, the application, and the user needs. For XML data that contains a substantial proportion of text, the information retrieval approach is superior. In Chapter 7, we will describe retrieval models that are designed for text documents that contain both structure and metadata.

4.5 Link Analysis

Links connecting pages are a key component of the Web. Links are a powerful navigational aid for people browsing the Web, but they also help search engines understand the relationships between the pages. These detected relationships help search engines rank web pages more effectively. It should be remembered, however, that many document collections used in search applications such as desktop or enterprise search either do not have links or have very little link structure. For these collections, link analysis will have no impact on search performance.

As we saw in the last section, a link in a web page is encoded in HTML with a statement such as:

```
For more information on this topic, please go to <a
href="http://www.somewhere.com">the somewhere page</a>.
```

When this page appears in your web browser, the words "the somewhere page" will be displayed differently than regular text, usually underlined or in a different color (or both). When you click on that link, your browser will then load the web page http://www.somewhere.com. In this link, "the somewhere page" is called the *anchor text*, and http://www.somewhere.com is the *destination*. Both components are useful in the ranking process.

[21] http://www.w3.org/XML/Query/

4.5.1 Anchor Text

Anchor text has two properties that make it particularly useful for ranking web pages. First, it tends to be very short, perhaps two or three words, and those words often succinctly describe the topic of the linked page. For instance, links to www.ebay.com are highly likely to contain the word "eBay" in the anchor text. Many queries are very similar to anchor text in that they are also short topical descriptions of web pages. This suggests a very simple algorithm for ranking pages: search through all links in the collection, looking for anchor text that is an exact match for the user's query. Each time there is a match, add 1 to the score of the destination page. Pages would then be ranked in decreasing order of this score. This algorithm has some glaring faults, not the least of which is how to handle the query "click here". More generally, the collection of all the anchor text in links pointing to a page can be used as an additional text field for that page, and incorporated into the ranking algorithm.

Anchor text is usually written by people who are not the authors of the destination page. This means that the anchor text can describe a destination page from a different perspective, or emphasize the most important aspect of the page from a community viewpoint. The fact that the link exists at all is a vote of importance for the destination page. Although anchor text is not mentioned as often as link analysis algorithms (for example, PageRank) in discussions of web search engines, TREC evaluations have shown that it is the most important part of the representation of a page for some types of web search. In particular, it is essential for searches where the user is trying to find a home page for a particular topic, person, or organization.

4.5.2 PageRank

There are tens of billions of web pages, but most of them are not very interesting. Many of those pages are spam and contain no useful content at all. Other pages are personal blogs, wedding announcements, or family picture albums. These pages are interesting to a small audience, but probably not broadly. On the other hand, there are a few pages that are popular and useful to many people, including news sites and the websites of popular companies.

The huge size of the Web makes this a difficult problem for search engines. Suppose a friend had told you to visit the site for eBay, and you didn't know that www.ebay.com was the URL to use. You could type "eBay" into a search engine, but there are millions of web pages that contain the word "eBay". How can the search engine choose the most popular (and probably the correct) one? One very effective approach is to use the links between web pages as a way to measure popularity. The most obvious measure is to count the number of *inlinks* (links pointing to a page) for each page and use this as a feature or piece of evidence in the ranking algorithm. Although this has been shown to be quite effective, it is very susceptible to spam. Measures based on link analysis algorithms are designed to provide more reliable ratings of web pages. Of these measures, PageRank, which is associated with the Google search engine, is most often mentioned.

PageRank is based on the idea of a *random surfer* (as in web surfer). Imagine a person named Alice who is using her web browser. Alice is extremely bored, so she wanders aimlessly between web pages. Her browser has a special "surprise me" button at the top that will jump to a random web page when she clicks it. Each time a web page loads, she chooses whether to click the "surprise me" button or whether to click one of the links on the web page. If she clicks a link on the page, she has no preference for any particular link; instead, she just picks one randomly. Alice is sufficiently bored that she intends to keep browsing the Web like this forever.[22]

To put this in a more structured form, Alice browses the Web using this algorithm:

```
1. Choose a random number $r$ between 0 and 1.
2. If $r < \lambda$:
   • Click the "surprise me" button.
3. If $r \geq \lambda$:
   • Click a link at random on the current page.
4. Start again.
```

Typically we assume that λ is fairly small, so Alice is much more likely to click a link than to pick the "surprise me" button. Even though Alice's path through the web pages is random, Alice will still see popular pages more often than unpopular ones. That's because Alice often follows links, and links tend to point to popular pages. So, we expect that Alice will end up at a university website, for example, more often than a personal website, but less often than the CNN website.

[22] The PageRank calculation corresponds to finding what is known as the stationary probability distribution of a *random walk* on the graph of the Web. A random walk is a special case of a *Markov chain* in which the next state (the next page visited) depends solely on the current state (current page). The transitions that are allowed between states are all equally probable and are given by the links.

Suppose that CNN has posted a story that contains a link to a professor's web page. Alice now becomes much more likely to visit that professor's page, because Alice visits the CNN website frequently. A single link at CNN might influence Alice's activity more than hundreds of links at less popular sites, because Alice visits CNN far more often than those less popular sites.

Because of Alice's special "surprise me" button, we can be guaranteed that eventually she will reach every page on the Internet.[23] Since she plans to browse the Web for a very long time, and since the number of web pages is finite, she will visit every page a very large number of times. It is likely, however, that she will visit a popular site thousands of times more often than an unpopular one. Note that if she did not have the "surprise me" button, she would get stuck on pages that did not have links, pages whose links no longer pointed to any page, or pages that formed a loop. Links that point to the first two types of pages, or pages that have not yet been crawled, are called *dangling links*.

Now suppose that while Alice is browsing, you happened to walk into her room and glance at the web page on her screen. What is the probability that she will be looking at the CNN web page when you walk in? That probability is CNN's PageRank. Every web page on the Internet has a PageRank, and it is uniquely determined by the link structure of web pages. As this example shows, PageRank has the ability to distinguish between popular pages (those with many incoming links, or those that have links from popular pages) and unpopular ones. The PageRank value can help search engines sift through the millions of pages that contain the word "eBay" to find the one that is most popular (www.ebay.com).

Alice would have to click on many billions of links in order for us to get a reasonable estimate of PageRank, so we can't expect to compute it by using actual people. Fortunately, we can compute PageRank in a much more efficient way.

Suppose for the moment that the Web consists of just three pages, A, B, and C. We will suppose that page A links to pages B and C, page B links to page C, and page C links to page A, as shown in Figure 4.10.

The PageRank of page C, which is the probability that Alice will be looking at this page, will depend on the PageRank of pages A and B. Since Alice chooses randomly between links on a given page, if she starts in page A, there is a 50% chance that she will go to page C (because there are two outgoing links). Another way of saying this is that the PageRank for a page is divided evenly between all the

[23] The "surprise button" makes the random surfer model an *ergodic* Markov chain, which guarantees that the iterative calculation of PageRank will converge.

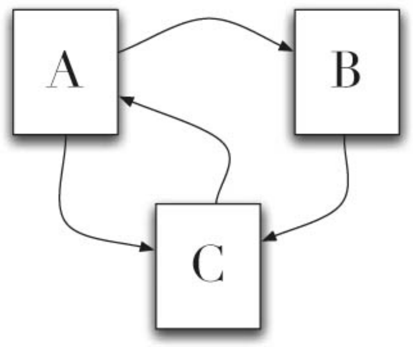


Fig. 4.10. A sample "Internet" consisting of just three web pages. The arrows denote links between the pages.

outgoing links. If we ignore the "surprise me" button, this means that the PageRank of page C, represented as $PR(C)$, can be calculated as:

$$PR(C) = \frac{PR(A)}{2} + \frac{PR(B)}{1}$$

More generally, we could calculate the PageRank for any page u as:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L_v}$$

where B_u is the set of pages that point to u, and L_v is the number of outgoing links from page v (not counting duplicate links).

There is an obvious problem here: we don't know the PageRank values for the pages, because that is what we are trying to calculate. If we start by assuming that the PageRank values for all pages are the same (1/3 in this case), then it is easy to see that we could perform multiple iterations of the calculation. For example, in the first iteration, $PR(C)$ = 0.33/2 + 0.33 = 0.5, $PR(A)$ = 0.33, and $PR(B)$ = 0.17. In the next iteration, $PR(C)$ = 0.33/2 + 0.17 = 0.33, $PR(A)$ = 0.5, and $PR(B)$ = 0.17. In the third iteration, $PR(C)$ = 0.42, $PR(A)$ = 0.33, and $PR(B)$ = 0.25. After a few more iterations, the PageRank values converge to the final values of $PR(C)$ = 0.4, $PR(A)$ = 0.4, and $PR(B)$ = 0.2.

If we take the "surprise me" button into account, part of the PageRank for page C will be due to the probability of coming to that page by pushing the button. Given that there is a 1/3 chance of going to any page when the button is pushed, the contribution to the PageRank for C for the button will be $\lambda/3$. This means that the total PageRank for C is now:

$$PR(C) = \frac{\lambda}{3} + (1-\lambda) \cdot (\frac{PR(A)}{2} + \frac{PR(B)}{1})$$

Similarly, the general formula for PageRank is:

$$PR(u) = \frac{\lambda}{N} + (1-\lambda) \cdot \sum_{v \in B_u} \frac{PR(v)}{L_v}$$

where N is the number of pages being considered. The typical value for λ is 0.15.

This can also be expressed as a matrix equation:

$$\mathbf{R} = \mathbf{TR}$$

where $\mathbf{R}$ is the vector of PageRank values and $\mathbf{T}$ is the matrix representing the transition probabilities for the random surfer model. The element $\mathbf{T}_{ij}$ represents the probability of going from page i to page j, and:

$$\mathbf{T}_{ij} = \frac{\lambda}{N} + (1-\lambda)\frac{1}{L_i}$$

Those of you familiar with linear algebra may recognize that the solution $\mathbf{R}$ is an *eigenvector* of the matrix $\mathbf{T}$.

Figure 4.11 shows some pseudocode for computing PageRank. The algorithm takes a graph G as input. Graphs are composed of vertices and edges, so $G = (V, E)$. In this case, the vertices are web pages and the edges are links, so the pseudocode uses the letters P and L instead. A link is represented as a pair (p, q), where p and q are the source and destination pages. Dangling links, which are links where the page q does not exist, are assumed to be removed. Pages with no outbound links are *rank sinks*, in that they accumulate PageRank but do not distribute it. In this algorithm, we assume that these pages link to all other pages in the collection.

The first step is to make a guess at the PageRank value for each page. Without any better information, we assume that the PageRank is the same for each page. Since PageRank values need to sum to 1 for all pages, we assign a PageRank of $1/|P|$ to each page in the input vector I. An alternative that may produce faster convergence would be to use a value based on the number of inlinks.

```
procedure PageRank($G$)
                ▷ $G$ is the web graph, consisting of vertices (pages) and edges (links).
    $(P, L) \leftarrow G$                        ▷ Split graph into pages and links
    $I \leftarrow$ a vector of length $|P|$              ▷ The current PageRank estimate
    $R \leftarrow$ a vector of length $|P|$        ▷ The resulting better PageRank estimate
    for all entries $I_i \in I$ do
        $I_i \leftarrow 1/|P|$                 ▷ Start with each page being equally likely
    end for
    while $R$ has not converged do
        for all entries $R_i \in R$ do
            $R_i \leftarrow \lambda/|P|$       ▷ Each page has a $\lambda/|P|$ chance of random selection
        end for
        for all pages $p \in P$ do
            $Q \leftarrow$ the set of pages $p$ such that $(p, q) \in L$ and $q \in P$
            if $|Q| > 0$ then
                for all pages $q \in Q$ do
                    $R_q \leftarrow R_q + (1 - \lambda)I_p/|Q|$ ▷ Probability $I_p$ of being at page $p$
                end for
            else
                for all pages $q \in P$ do
                    $R_p \leftarrow R_q + (1 - \lambda)I_p/|P|$
                end for
            end if
            $I \leftarrow R$                    ▷ Update our current PageRank estimate
        end for
    end while
    return $R$
end procedure
```

Fig. 4.11. Pseudocode for the iterative PageRank algorithm

In each iteration, we start by creating a result vector, R, and storing $\lambda/|P|$ in each entry. This is the probability of landing at any particular page because of a random jump. The next step is to compute the probability of landing on a page because of a clicked link. We do that by iterating over each web page in P. At each page, we retrieve the estimated probability of being at that page, I_p. From that page, the user has a λ chance of jumping randomly, or $1 - \lambda$ of clicking on a link. There are $|Q|$ links to choose from, so the probability of jumping to a page $q \in Q$ is $(1 - \lambda)I_p/|Q|$. We add this quantity to each entry R_q. In the event that there are no usable outgoing links, we assume that the user jumps randomly, and therefore the probability $(1 - \lambda)I_p$ is spread evenly among all $|P|$ pages.

To summarize, PageRank is an important example of query-independent metadata that can improve ranking for web search. Web pages have the same PageRank value regardless of what query is being processed. Search engines that use PageRank will prefer pages with high PageRank values instead of assuming that all web pages are equally likely to satisfy a query. PageRank is not, however, as important in web search as the conventional wisdom holds. It is just one of many features used in ranking. It does, however, tend to have the most impact on popular queries, which is a useful property.

The *HITS*[24] algorithm (Kleinberg, 1999) for link analysis was developed at about the same time as PageRank and has also been very influential. This algorithm estimates the value of the content of a page (the *authority* value) and the value of the links to other pages (the *hub* value). Both values are computed using an iterative algorithm based solely on the link structure, similar to PageRank. The HITS algorithm, unlike PageRank, calculates authority and hub values for a subset of pages retrieved by a given query.[25] This can be an advantage in terms of the impact of the HITS metadata on ranking, but may be computationally infeasible for search engines with high query traffic. In Chapter 10, we discuss the application of the HITS algorithm to finding web communities.

4.5.3 Link Quality

It is well known that techniques such as PageRank and anchor text extraction are used in commercial search engines, so unscrupulous web page designers may try to create useless links just to improve the search engine placement of one of their web pages. This is called *link spam*. Even typical users, however, can unwittingly fool simple search engine techniques. A good example of this is with blogs.

Many blog posts are comments about other blog posts. Suppose author A reads a post called b in author B's blog. Author A might write a new blog post, called a, which contains a link to post b. In the process of posting, author A may post a *trackback* to post b in author B's blog. A trackback is a special kind of comment that alerts author B that a reply has been posted in author A's blog.

[24] Hypertext Induced Topic Search

[25] Query-independent versions of HITS and topic-dependent versions of PageRank have also been defined.

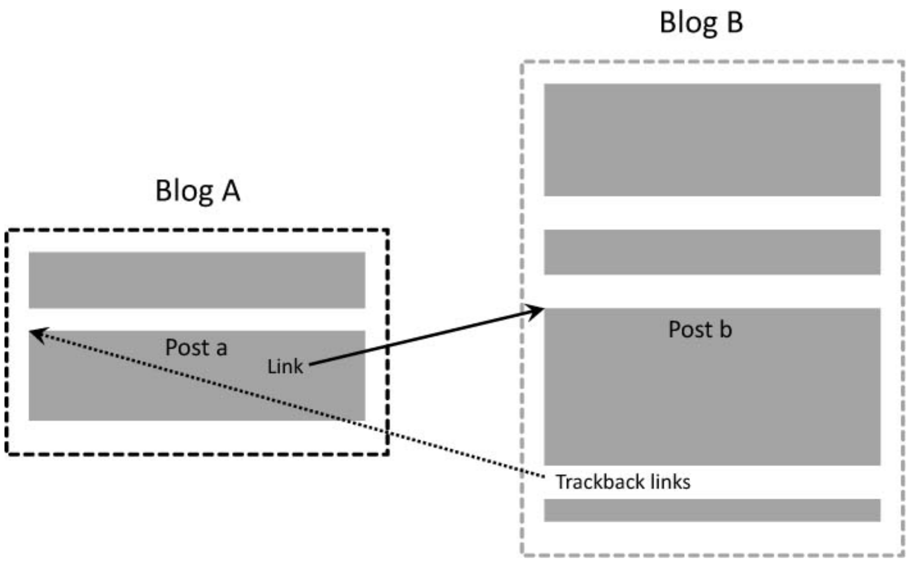


Fig. 4.12. Trackback links in blog postings

As Figure 4.12 shows, a cycle has developed between post a and post b. Post a links to post b, and post b contains a trackback link to post a. Intuitively we would say that post b is influential, because author A has decided to write about it. However, from the PageRank perspective, a and b have links to each other, and therefore neither is more influential than the other. The trouble here is that a trackback is a fundamentally different kind of link than one that appears in a post.

The comments section of a blog can also be a source of link spam. Page authors may try to promote their own websites by posting links to them in the comments section of popular blogs. Based on our discussion of PageRank, we know that a link from a popular website can make another website seem much more important. Therefore, this comments section is an attractive target for spammers.

In this case, one solution is for search engine companies to automatically detect these comment sections and effectively ignore the links during indexing. An even easier way to do this is to ask website owners to alter the unimportant links so that search engines can detect them. This is the purpose behind the `rel=nofollow` link attribute.

Most blog software is now designed to modify any link in a blog comment to contain the `rel=nofollow` attribute. Therefore, a post like this:

```
Come visit my <a href="http://www.page.com">web page</a>.
```

becomes something like this:

```
Come visit my <a rel=nofollow href="http://www.page.com">web page</a>.
```

The link still appears on the blog, but search engines are designed to ignore all links marked rel=nofollow. This helps preserve the integrity of PageRank calculation and anchor text harvesting.

4.6 Information Extraction

Information extraction is a language technology that focuses on extracting structure from text. Information extraction is used in a number of applications, and particularly for *text data mining*. For search applications, the primary use of information extraction is to identify features that can be used by the search engine to improve ranking. Some people have speculated that information extraction techniques could eventually transform text search into a database problem by extracting all of the important information from text and storing it in structured form, but current applications of these techniques are a very long way from achieving that goal.

Some of the text processing steps we have already discussed could be considered information extraction. Identifying noun phrases, titles, or even bolded text are examples. In each of these cases, a part of the text has been recognized as having some special property, and that property can be described using a markup language, such as XML. If a document is already described using HTML or XML, the recognition of some of the structural features (such as titles) is straightforward, but others, such as phrases, require additional processing before the feature can be *annotated* using the markup language. In some applications, such as when the documents in the collection are input through OCR, the document has no markup and even simple structures such as titles must be recognized and annotated.

These types of features are very general, but most of the recent research in information extraction has been concerned with features that have specific semantic content, such as *named entities*, *relationships*, and *events*. Although all of these features contain important information, *named entity recognition* has been used most often in search applications. A named entity is a word or sequence of words that is used to refer to something of interest in a particular application. The most common examples are people's names, company or organization names, locations, time and date expressions, quantities, and monetary values. It is easy to come up with other "entities" that would be important for specific applications. For an e-commerce application, for example, the recognition of product names and model numbers in web pages and reviews would be essential. In a pharmaceutical application, the recognition of drug names, dosages, and medical conditions may be important. Given the more specific nature of these features, the process of recognizing them and tagging them in text is sometimes called *semantic annotation*. Some of these recognized entities would be incorporated directly into the search using, for example, *facets* (see Chapter 6), whereas others may be used as part of browsing the search results. An example of the latter is the search engine feature that recognizes addresses in pages and provides links to the appropriate map.

Fred Smith, who lives at 10 Water Street, Springfield, MA, is a long-time collector of **tropical fish.**

```
<p ><PersonName><GivenName>Fred</GivenName> <Sn>Smith</Sn>
</PersonName>, who lives at <address><Street >10 Water Street</Street>,
<City>Springfield</City>, <State>MA</State></address>, is a long-time
collector of <b>tropical fish.</b></p>
```

Fig. 4.13. Text tagged by information extraction

Figure 4.13 shows a sentence and the corresponding XML markup after using information extraction. In this case, the extraction was done by a well-known word processing program.[26] In addition to the usual structure markup (`<p>` and `<b>`), a number of tags have been added that indicate which words are part of named entities. It shows, for example, that an address consisting of a street ("10 Water Street"), a city ("Springfield"), and a state ("MA") was recognized in the text.

Two main approaches have been used to build named entity recognizers: rule-based and statistical. A rule-based recognizer uses one or more *lexicons* (lists of words and phrases) that categorize names. Some example categories would be locations (e.g., towns, cities, states, countries, places of interest), people's names (given names, family names), and organizations (e.g., companies, government

[26] Microsoft Word

agencies, international groups). If these lists are sufficiently comprehensive, much of the extraction can be done simply by lookup. In many cases, however, rules or patterns are used to verify an entity name or to find new entities that are not in the lists. For example, a pattern such as "<number> <word> street" could be used to identify street addresses. Patterns such as "<street address>, <city>" or "in <city>" could be used to verify that the name found in the location lexicon as a city was indeed a city. Similarly, a pattern such as "<street address>, <city>, <state>" could also be used to identify new cities or towns that were not in the lexicon. New person names could be recognized by rules such as "<title> <name>", where <title> would include words such as "President", "Mr.", and "CEO". Names are generally easier to extract in mixed-case text, because capitalization often indicates a name, but many patterns will apply to all lower- or uppercase text as well. Rules incorporating patterns are developed manually, often by trial and error, although an initial set of rules can also be used as *seeds* in an automated learning process that can discover new rules.[27]

A statistical entity recognizer uses a probabilistic model of the words in and around an entity. A number of different approaches have been used to build these models, but because of its importance, we will briefly describe the *Hidden Markov Model* (HMM) approach. HMMs are used for many applications in speech and language. For example, POS taggers can be implemented using this approach.

4.6.1 Hidden Markov Models for Extraction

One of the most difficult parts of entity extraction is that words can have many different meanings. The word "Bush", for example, can describe a plant or a person. Similarly, "Marathon" could be the name of a race or a location in Greece. People tell the difference between these different meanings based on the *context* of the word, meaning the words that surround it. For instance, if "Marathon" is preceded by "Boston", the text is almost certainly describing a race. We can describe the context of a word mathematically by modeling the *generation*[28] of the sequence of words in a text as a process with the *Markov property*, meaning that the next word in the sequence depends on only a small number of the previous words.

[27] GATE (http://gate.ac.uk) is an example of an open source toolkit that provides both an information extraction component and an environment for customizing extraction for a specific application.

[28] We discuss *generative models* in more detail in Chapter 7.

More formally, a *Markov Model* describes a process as a collection of *states* with *transitions* between them. Each of the transitions has an associated probability. The next state in the process depends solely on the current state and the transition probabilities. In a Hidden Markov Model, each state has a set of possible outputs that can be generated. As with the transitions, each output also has a probability associated with it.

Figure 4.14 shows a *state diagram* representing a very simple model for sentence generation that could be used by a named entity recognizer. In this model, the words in a sentence are assumed to be either part of an entity name (in this case, either a person, organization, or location) or not part of one. Each of these entity categories is represented by a state, and after every word the system may stay in that state (represented by the arrow loops) or transition to another state. There are two special states representing the start and end of the sentence. Associated with each state representing an entity category, there is a probability distribution of the likely sequences of words for that category.

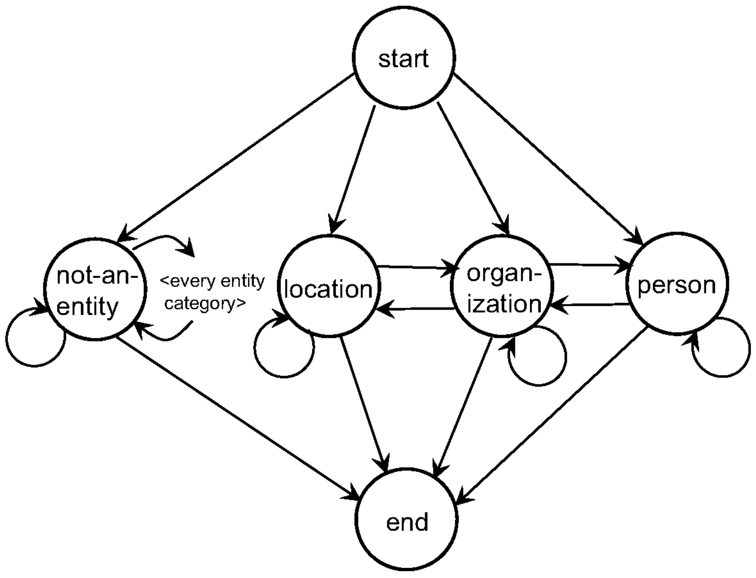


Fig. 4.14. Sentence model for statistical entity extractor

One possible use of this model is to construct new sentences. Suppose that we begin in the start state, and then the next state is randomly chosen according to the start state's transition probability table. For example, we may transition to the person state. Once we have entered the person state, we complete the transition by choosing an output according to the person state's output probability distribution. An example output may be the word "Thomas". This process would continue, with a new state being transitioned to and an output being generated during each step of the process. The final result is a set of states and their associated outputs.

Although such models can be used to generate new sentences, they are more commonly used to recognize entities in a sentence. To do this for a given sentence, a sequence of entity categories is found that gives the highest probability for the words in that sentence. Only the outputs generated by state transitions are visible (i.e., can be observed); the underlying states are "hidden." For the sentence in Figure 4.13, for example, the recognizer would find the sequence of states

<start><name><not-an-entity><location><not-an-entity><end>

to have the highest probability for that model. The words that were associated with the entity categories in this sequence would then be tagged. The problem of finding the most likely sequence of states in an HMM is solved by the *Viterbi algorithm*,[29] which is a *dynamic programming* algorithm.

The key aspect of this approach to entity recognition is that the probabilities in the sentence model must be estimated from training data. To estimate the transition and output probabilities, we generate training data that consists of text manually annotated with the correct entity tags. From this training data, we can directly estimate the probability of words associated with a given category (i.e., output probabilities), and the probability of transitions between categories. To build a more accurate recognizer, features that are highly associated with named entities, such as capitalized words and words that are all digits, would be included in the model. In addition, the transition probabilities could depend on the previous word as well as the previous category.[30] For example, the occurrence of the word "Mr." increases the probability that the next category is Person.

Although such training data can be useful for constructing accurate HMMs, collecting it requires a great deal of human effort. To generate approximately one million words of annotated text, which is the approximate size of training data required for accurate estimates, people would have to annotate the equivalent of

[29] Named after the electrical engineer Andrew Viterbi.

[30] Bikel et al. (1997) describe one of the first named entity recognizers based on the HMM approach.

more than 1,500 news stories. This may require considerably more effort than developing rules for a simple set of features. Both the rule-based and statistical approaches have recognition effectiveness of about 90%[31] for entities such as name, organization, and location, although the statistical recognizers are generally the best. Other entity categories, such as product names, are considerably more difficult. The choice of which entity recognition approach to use will depend on the application, the availability of software, and the availability of annotators.

Interestingly, there is little evidence that named entities are useful features for general search applications. Named entity recognition is a critical part of question-answering systems (section 11.5), and can be important in domain-specific or vertical search engines for accurately recognizing and indexing domain terms. Named entity recognition can also be useful for query analysis in applications such as local search, and as a tool for understanding and browsing search results.

4.7 Internationalization

The Web is used all over the world, and not just by English speakers. Although 65–70% of the Web is written in English, that percentage is continuing to decrease. More than half of the people who use the Web, and therefore search the Web, do not use English as their primary language. Other search applications, such as desktop search and corporate search, are being used in many different languages every day. Even an application designed for an environment that has mostly English-speaking users can have many non-English documents in the collection. Try using "poissons tropicaux" (tropical fish) as a query for your favorite web search engine and see how many French web pages are retrieved.[32]

A *monolingual* search engine is, as the name suggests, a search engine designed for a particular language.[33] Many of the indexing techniques and retrieval models we discuss in this book work well for any language. The differences between languages that have the most impact on search engine design are related to the text processing steps that produce the index terms for searching.

[31] By this we mean that about 9 out of 10 of the entities found are accurately identified, and 9 out of 10 of the existing entities are found. See Chapter 8 for details on evaluation measures.

[32] You would find many more French web pages, of course, if you used a French version of the search engine, such as http://fr.yahoo.com.

[33] We discuss *cross-language* search engines in section 6.4.

As we mentioned in the previous chapter, character encoding is a crucial issue for search engines dealing with non-English languages, and Unicode has become the predominant character encoding standard for the internationalization of software.

Other text processing steps also need to be customized for different languages. The importance of stemming for highly inflected languages has already been mentioned, but each language requires a customized stemmer. Tokenizing is also important for many languages, especially for the CJK family of languages. For these languages, the key problem is *word segmentation*, where the breaks corresponding to words or index terms must be identified in the continuous sequence of characters (spaces are generally not used). One alternative to segmenting is to index overlapping character bigrams (pairs of characters, see section 4.3.5). Figure 4.15 shows word segmentation and bigrams for the text "impact of droughts in China". Although the ranking effectiveness of search based on bigrams is quite good, word segmentation is preferred in many applications because many of the bigrams do not correspond to actual words. A segmentation technique can be implemented based on statistical approaches, such as a Hidden Markov Model, with sufficient training data. Segmentation can also be an issue in other languages. German, for example, has many compound words (such as "fischzuchttechniken" for "fish farming techniques") that should be segmented for indexing.

1. Original text

旱灾在中国造成的影响

(the impact of droughts in China)

2. Word segmentation

旱灾	在	中国	造成	的	影响
drought	at	china	make		impact

3. Bigrams

旱灾 灾在 在中 中国 国造

造成 成的 的影 影响

Fig. 4.15. Chinese segmentation and bigrams

In general, given the tools that are available, it is not difficult to build a search engine for the major languages. The same statement holds for any language that has a significant amount of online text available on the Web, since this can be used as a resource to build and test the search engine components. There are, however, a large number of other so-called "low-density" languages that may have many speakers but few online resources. Building effective search engines for these languages is more of a challenge.

References and Further Reading

The properties and statistics of text and document collections has been studied for some time under the heading of *bibliometrics*, which is part of the field of *library and information science*. Information science journals such as the *Journal of the American Society of Information Science and Technology* (JASIST) or *Information Processing and Management* (IPM) contain many papers in this general area. Information retrieval has, from the beginning, emphasized a statistical view of text, and researchers from IR and information science have always worked closely together. Belew (2000) contains a good discussion of the cognitive aspects of Zipf's law and other properties of text in relationship to IR. With the shift to statistical methods in the 1990s, *natural language processing* researchers also became interested in studying the statistical properties of text. Manning and Schütze (1999) is a good summary of text statistics from this perspective. Ha et al. (2002) give an interesting result showing that phrases (or n-grams) also generally follow Zipf's law, and that combining the phrases and words results in better predictions for frequencies at low ranks.

The paper by Anagnostopoulos et al. (2005) describes a technique for estimating query result size and also points to much of the relevant literature in this area. Similarly, Broder et al. (2006) show how to estimate corpus size and compare their estimation with previous techniques.

Not much is written about tokenizing or stopping. Both are considered sufficiently "well known" that they are hardly mentioned in papers. As we have pointed out, however, getting these basic steps right is crucial for the overall system's effectiveness. For many years, researchers used the stopword list published in van Rijsbergen (1979). When it became clear that this was not sufficient for the larger TREC collections, a stopword list developed at the University of Massachusetts and distributed with the Lemur toolkit has frequently been used. As mentioned previously, this list contains over 400 words, which will be too long for many applications.

The original paper describing the Porter stemmer was written in 1979, but was reprinted in Porter (1997). The paper by Krovetz (1993) describes his stemming algorithm but also takes a more detailed approach to studying the role of morphology in a stemmer.[34] The Krovetz stemmer is available on the Lemur website. Stemmers for other languages are available from various websites (including the Lemur website and the Porter stemmer website). A description of Arabic stemming techniques can be found in Larkey et al. (2002).

Research on the use of phrases in searching has a long history. Croft et al. (1991) describe retrieval experiments with phrases derived by both syntactic and statistical processing of the query, and showed that effectiveness was similar to phrases selected manually. Many groups that have participated in the TREC evaluations have used phrases as part of their search algorithms (Voorhees & Harman, 2005).

Church (1988) described an approach to building a statistical (or *stochastic*) part-of-speech tagger that is the basis for many current taggers. This approach uses manually tagged training data to train a probabilistic model of sequences of parts of speech, as well as the probability of a part of speech for a specific word. For a given sentence, the part-of-speech tagging that gives the highest probability for the whole sentence is used. This method is essentially the same as that used by a statistical entity extractor, with the states being parts of speech instead of entity categories. The Brill tagger (Brill, 1994) is a popular alternative approach that uses rules that are learned automatically from tagged data. Manning and Schütze (1999) provide a good overview of part-of-speech tagging methods.

Many variations of PageRank can be found in the literature. Many of these variations are designed to be more efficient to compute or are used in different applications. The topic-dependent version of PageRank is described in Haveliwala (2002). Both PageRank and HITS have their roots in the citation analysis algorithms developed in the field of bibliometrics.

The idea of enhancing the representation of a hypertext document (i.e., a web page) using the content of the documents that point to it has been around for some time. For example, Croft and Turtle (1989) describe a retrieval model based on incorporating text from related hypertext documents, and Dunlop and van Rijsbergen (1993) describe how documents with little text content (such as those containing images) could be retrieved using the text in linked documents. Re-

[34] Morphology is the study of the internal structure of words, and stemming is a form of *morphological processing*.

stricting the text that is incorporated to the anchor text associated with inlinks was first mentioned by McBryan (1994). Anchor text has been shown to be essential for some categories of web search in TREC evaluations, such as in Ogilvie and Callan (2003).

Techniques have been developed for applying link analysis in collections without explicit link structure (Kurland & Lee, 2005). In this case, the links are based on similarities between the content of the documents, calculated by a similarity measure such as the cosine correlation (see Chapter 7).

Information extraction techniques were developed primarily in research programs such as TIPSTER and MUC (Cowie & Lehnert, 1996). Using named entity extraction to provide additional features for search was also studied early in the TREC evaluations (Callan et al., 1992, 1995). One of the best-known rule-based information extraction systems is FASTUS (Hobbs et al., 1997). The BBN system Identifinder (Bikel et al., 1999), which is based on an HMM, has been used in many projects.

A detailed description of HMMs and the Viterbi algorithm can be found in Manning and Schütze (1999). McCallum (2005) provides an overview of information extraction, with references to more recent advances in the field. Statistical models that incorporate more complex features than HMMs, such as *conditional random fields*, have become increasingly popular for extraction (Sutton & McCallum, 2007).

Detailed descriptions of all the major encoding schemes can be found in Wikipedia. Fujii and Croft (1993) was one of the early papers that discussed the problems of text processing for search with CJK languages. An entire journal, *ACM Transactions on Asian Language Information Processing*,[35] has now been devoted to this issue. Peng et al. (2004) describe a statistical model for Chinese word segmentation and give references to other approaches.

Exercises

4.1. Plot rank-frequency curves (using a log-log graph) for words and bigrams in the Wikipedia collection available through the book website (http://www.search-engines-book.com). Plot a curve for the combination of the two. What are the best values for the parameter c for each curve?

[35] http://talip.acm.org/

4.2. Plot vocabulary growth for the Wikipedia collection and estimate the parameters for Heaps' law. Should the order in which the documents are processed make any difference?

4.3. Try to estimate the number of web pages indexed by two different search engines using the technique described in this chapter. Compare the size estimates from a range of queries and discuss the consistency (or lack of it) of these estimates.

4.4. Modify the Galago tokenizer to handle apostrophes or periods in a different way. Describe the new rules your tokenizer implements. Give examples of where the new tokenizer does a better job (in your opinion) and examples where it does not.

4.5. Examine the Lemur stopword list and list 10 words that you think would cause problems for some queries. Give examples of these problems.

4.6. Process five Wikipedia documents using the Porter stemmer and the Krovetz stemmer. Compare the number of stems produced and find 10 examples of differences in the stemming that could have an impact on ranking.

4.7. Use the GATE POS tagger to tag a Wikipedia document. Define a rule or rules to identify phrases and show the top 10 most frequent phrases. Now use the POS tagger on the Wikipedia queries. Are there any problems with the phrases identified?

4.8. Find the 10 Wikipedia documents with the most inlinks. Show the collection of anchor text for those pages.

4.9. Compute PageRank for the Wikipedia documents. List the 20 documents with the highest PageRank values together with the values.

4.10. Figure 4.11 shows an algorithm for computing PageRank. Prove that the entries of the vector I sum to 1 every time the algorithm enters the loop on line 9.

4.11. Implement a rule-based recognizer for cities (you can choose a subset of cities to make this easier). Create a test collection that you can scan manually to find cities mentioned in the text and evaluate your recognizer. Summarize the performance of the recognizer and discuss examples of failures.

4.12. Create a small test collection in some non-English language using web pages. Do the basic text processing steps of tokenizing, stemming, and stopping using tools from the book website and from other websites. Show examples of the index term representation of the documents.