6 Information Retrieval and Web Search

Web search needs no introduction. Due to its convenience and the richness of information on the Web, searching the Web is increasingly becoming the dominant information seeking method. People make fewer and fewer trips to libraries, but more and more searches on the Web. In fact, without effective search engines and rich Web contents, writing this book would have been much harder.

Web search has its root in **information retrieval** (or IR for short), a field of study that helps the user find needed information from a large collection of text documents. Traditional IR assumes that the basic information unit is a **document**, and a large collection of documents is available to form the text database. On the Web, the documents are **Web pages**.

Retrieving information simply means finding a set of documents that is relevant to the user query. A ranking of the set of documents is usually also performed according to their relevance scores to the query. The most commonly used query format is a list of **keywords**, which are also called **terms**. IR is different from data retrieval in databases using SQL queries because the data in databases are highly structured and stored in relational tables, while information in text is unstructured. There is no structured query language like SQL for text retrieval.

It is safe to say that Web search is the single most important application of IR. To a great extent, Web search also helped IR. Indeed, the tremendous success of search engines has pushed IR to the center stage. Search is, however, not simply a straightforward application of traditional IR models. It uses some IR results, but it also has its unique techniques and presents many new problems for IR research.

First of all, efficiency is a paramount issue for Web search, but is only secondary in traditional IR systems mainly due to the fact that document collections in most IR systems are not very large. However, the number of pages on the Web is huge. For example, Google indexed more than 8 billion pages when this book was written. Web users also demand very fast responses. No matter how effective an algorithm is, if the retrieval cannot be done efficiently, few people will use it.

Web pages are also quite different from conventional text documents used in traditional IR systems. First, Web pages have **hyperlinks** and **an-**

chor texts, which do not exist in traditional documents (except citations in research publications). Hyperlinks are extremely important for search and play a central role in search ranking algorithms as we will see in the next chapter. Anchor texts associated with hyperlinks too are crucial because a piece of anchor text is often a more accurate description of the page that its hyperlink points to. Second, Web pages are semi-structured. A Web page is not simply a few paragraphs of text like in a traditional document. A Web page has different fields, e.g., title, metadata, body, etc. The information contained in certain fields (e.g., the title field) is more important than in others. Furthermore, the content in a page is typically organized and presented in several structured blocks (of rectangular shapes). Some blocks are important and some are not (e.g., advertisements, privacy policy, copyright notices, etc). Effectively detecting the main content block(s) of a Web page is useful to Web search because terms appearing in such blocks are more important.

Finally, **spamming** is a major issue on the Web, but not a concern for traditional IR. This is so because the rank position of a page returned by a search engine is extremely important. If a page is relevant to a query but is ranked very low (e.g., below top 30), then the user is unlikely to look at the page. If the page sells a product, then this is bad for the business. In order to improve the ranking of some target pages, "illegitimate" means, called spamming, are often used to boost their rank positions. Detecting and fighting Web spam is a critical issue as it can push low quality (even irrelevant) pages to the top of the search rank, which harms the quality of the search results and the user's search experience.

In this chapter, we first study some information retrieval models and methods that are closely related to Web search. We then dive into some Web search specific issues.

6.1 Basic Concepts of Information Retrieval

Information retrieval (IR) is the study of helping users to find information that matches their information needs. Technically, IR studies the acquisition, organization, storage, retrieval, and distribution of information. Historically, IR is about document retrieval, emphasizing document as the basic unit. Fig. 6.1 gives a general architecture of an IR system.

In Figure 6.1, the user with information need issues a query (**user query**) to the **retrieval system** through the **query operations** module. The retrieval module uses the **document index** to retrieve those documents that contain some query terms (such documents are likely to be relevant to the query), compute relevance scores for them, and then rank the retrieved documents according to the scores. The ranked documents are then presented to the user. The **document collection** is also called the **text database**, which is indexed by the **indexer** for efficient retrieval.

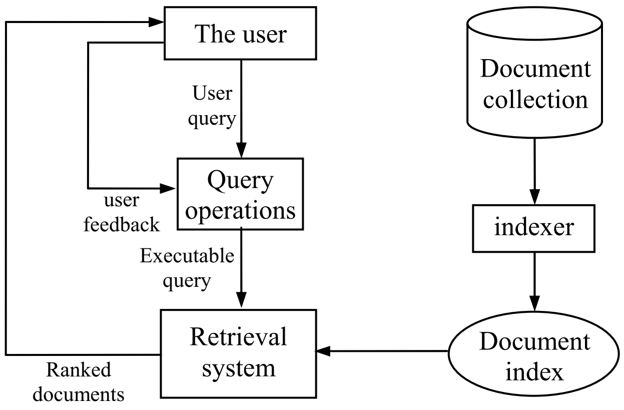


Fig. 6.1. A general IR system architecture

A user query represents the user's information needs, which is in one of the following forms:

1. **Keyword queries:** The user expresses his/her information needs with a list of (at least one) keywords (or **terms**) aiming to find documents that contain some (at least one) or all the query terms. The terms in the list are assumed to be connected with a "soft" version of the logical AND. For example, if one is interested in finding information about Web mining, one may issue the query 'Web mining' to an IR or search engine system. 'Web mining' is retreated as 'Web AND mining'. The retrieval system then finds those likely relevant documents and ranks them suitably to present to the user. Note that a retrieved document does not have to contain all the terms in the query. In some IR systems, the ordering of the words is also significant and will affect the retrieval results.
2. **Boolean queries:** The user can use Boolean operators, AND, OR, and NOT to construct complex queries. Thus, such queries consist of terms and Boolean operators. For example, 'data OR Web' is a Boolean query, which requests documents that contain the word 'data' or 'Web. A page is returned for a Boolean query if the query is logically true in the page (i.e., **exact match**). Although one can write complex Boolean queries using the three operators, users seldom write such queries. Search engines usually support a restricted version of Boolean queries.
3. **Phrase queries:** Such a query consists of a sequence of words that makes up a phrase. Each returned document must contain at least one

instance of the phrase. In a search engine, a phrase query is normally enclosed with double quotes. For example, one can issue the following phrase query (including the double quotes), "Web mining techniques and applications" to find documents that contain the exact phrase.
4. **Proximity queries:** The proximity query is a relaxed version of the phrase query and can be a combination of terms and phrases. Proximity queries seek the query terms within close proximity to each other. The closeness is used as a factor in ranking the returned documents or pages. For example, a document that contains all query terms close together is considered more relevant than a page in which the query terms are far apart. Some systems allow the user to specify the maximum allowed distance between the query terms. Most search engines consider both term proximity and term ordering in retrieval.
5. **Full document queries:** When the query is a full document, the user wants to find other documents that are similar to the query document. Some search engines (e.g., Google) allow the user to issue such a query by providing the URL of a query page. Additionally, in the returned results of a search engine, each snippet may have a link called "more like this" or "similar pages." When the user clicks on the link, a set of pages similar to the page in the snippet is returned.
6. **Natural language questions:** This is the most complex case, and also the ideal case. The user expresses his/her information need as a natural language question. The system then finds the answer. However, such queries are still hard to handle due to the difficulty of natural language understanding. Nevertheless, this is an active research area, called **question answering**. Some search systems are starting to provide question answering services on some specific types of questions, e.g., definition questions, which ask for definitions of technical terms. Definition questions are usually easier to answer because there are strong linguistic patterns indicating definition sentences, e.g., "defined as", "refers to", etc. Definitions can usually be extracted offline [339, 280].

The **query operations** module can range from very simple to very complex. In the simplest case, it does nothing but just pass the query to the retrieval engine after some simple pre-processing, e.g., removal of **stopwords** (words that occur very frequently in text but have little meaning, e.g., "the", "a", "in", etc). We will discuss text pre-processing in Sect. 6.5. In more complex cases, it needs to transform natural language queries into executable queries. It may also accept user feedback and use it to expand and refine the original queries. This is usually called **relevance feedback**, which will be discussed in Sect. 6.3.

The **indexer** is the module that indexes the original raw documents in some data structures to enable efficient retrieval. The result is the **docu-**

ment index. In Sect. 6.6, we study a particular type of indexing scheme, called the **inverted index**, which is used in search engines and most IR systems. An inverted index is easy to build and very efficient to search.

The **retrieval system** computes a relevance score for each indexed document to the query. According to their relevance scores, the documents are ranked and presented to the user. Note that it usually does not compare the user query with every document in the collection, which is too inefficient. Instead, only a small subset of the documents that contains at least one query term is first found from the index and relevance scores with the user query are then computed only for this subset of documents.

6.2 Information Retrieval Models

An IR model governs how a document and a query are represented and how the relevance of a document to a user query is defined. There are four main IR models: Boolean model, vector space model, language model and probabilistic model. The most commonly used models in IR systems and on the Web are the first three models, which we study in this section.

Although these three models represent documents and queries differently, they used the same framework. They all treat each document or query as a **"bag" of words** or **terms**. Term sequence and position in a sentence or a document are ignored. That is, a document is described by a set of distinctive terms. A term is simply a word whose semantics helps remember the document's main themes. We should note that the term here may not be a natural language word in a dictionary. Each term is associated with a weight. Given a collection of documents D, let $V = \{t_1, t_2, ..., t_{|V|}\}$ be the set of distinctive terms in the collection, where t_i is a term. The set V is usually called the **vocabulary** of the collection, and $|V|$ is its size, i.e., the number of terms in V. A weight $w_{ij} > 0$ is associated with each term t_i of a document $\mathbf{d}_j \in D$. For a term that does not appear in document $\mathbf{d}_j$, $w_{ij} = 0$. Each document $\mathbf{d}_j$ is thus represented with a term vector,

$$\mathbf{d}_j = (w_{1j}, w_{2j}, ..., w_{|V|j}),$$

where each weight w_{ij} corresponds to the term $t_i \in V$, and quantifies the level of importance of t_i in document $\mathbf{d}_j$. The sequence of the components (or terms) in the vector is not significant. Note that following the convention of this book, a bold lower case letter is used to represent a vector.

With this vector representation, a collection of documents is simply represented as a relational table (or a matrix). Each term is an attribute, and each weight is an attribute value. In different retrieval models, w_{ij} is computed differently.

6.2.1 Boolean Model

The Boolean model is one of the earliest and simplest information retrieval models. It uses the notion of exact matching to match documents to the user query. Both the query and the retrieval are based on Boolean algebra.

Document Representation: In the Boolean model, documents and queries are represented as sets of terms. That is, each term is only considered present or absent in a document. Using the vector representation of the document above, the weight w_{ij} ($\in \{0, 1\}$) of term t_i in document $\mathbf{d}_j$ is 1 if t_i appears in document $\mathbf{d}_j$, and 0 otherwise, i.e.,

$$w_{ij} = \begin{cases} 1 & \text{if } t_i \text{ appears in } \mathbf{d}_j \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

Boolean Queries: As we mentioned in Sect. 6.1, query terms are combined logically using the Boolean operators **AND**, **OR**, and **NOT**, which have their usual semantics in logic. Thus, a Boolean query has a precise semantics. For instance, the query, ((x AND y) AND (NOT z)) says that a retrieved document must contain both the terms x and y but not z. As another example, the query expression (x OR y) means that at least one of these terms must be in each retrieved document. Here, we assume that x, y and z are terms. In general, they can be Boolean expressions themselves.

Document Retrieval: Given a Boolean query, the system retrieves every document that makes the query logically true. Thus, the retrieval is based on the binary decision criterion, i.e., a document is either relevant or irrelevant. Intuitively, this is called **exact match**. There is no notion of partial match or ranking of the retrieved documents. This is one of the major disadvantages of the Boolean model, which often leads to poor retrieval results. It is quite clear that the frequency of terms and their proximity contribute significantly to the relevance of a document.

Due to this problem, the Boolean model is seldom used alone in practice. Most search engines support some limited forms of Boolean retrieval using explicit **inclusion** and **exclusion operators**. For example, the following query can be issued to Google, 'mining –data +"equipment price"', where + (inclusion) and – (exclusion) are similar to Boolean operators AND and NOT respectively. The operator OR may be supported as well.

6.2.2 Vector Space Model

This model is perhaps the best known and most widely used IR model.

Document Representation

A document in the vector space model is represented as a weight vector, in which each component weight is computed based on some variation of TF or TF-IDF scheme. The weight w_{ij} of term t_i in document $\mathbf{d}_j$ is no longer in {0, 1} as in the Boolean model, but can be any number.

Term Frequency (TF) Scheme: In this method, the weight of a term t_i in document $\mathbf{d}_j$ is the number of times that t_i appears in document $\mathbf{d}_j$, denoted by f_{ij}. Normalization may also be applied (see Equation (2)).

The shortcoming of the TF scheme is that it does not consider the situation where a term appears in many documents of the collection. Such a term may not be discriminative.

TF-IDF Scheme: This is the most well known weighting scheme, where TF still stands for the **term frequency** and IDF the **inverse document frequency**. There are several variations of this scheme. Here we only give the most basic one.

Let N be the total number of documents in the system or the collection and df_i be the number of documents in which term t_i appears at least once. Let f_{ij} be the raw frequency count of term t_i in document $\mathbf{d}_j$. Then, the **normalized term frequency** (denoted by tf_{ij}) of t_i in $\mathbf{d}_j$ is given by

$$tf_{ij} = \frac{f_{ij}}{\max\{f_{1j}, f_{2j}, \ldots, f_{|V|j}\}}, \tag{2}$$

where the maximum is computed over all terms that appear in document $\mathbf{d}_j$. If term t_i does not appear in $\mathbf{d}_j$ then $tf_{ij} = 0$. Recall that $|V|$ is the vocabulary size of the collection.

The inverse document frequency (denoted by idf_i) of term t_i is given by:

$$idf_i = \log \frac{N}{df_i}. \tag{3}$$

The intuition here is that if a term appears in a large number of documents in the collection, it is probably not important or not discriminative. The final TF-IDF term weight is given by:

$$w_{ij} = tf_{ij} \times idf_i. \tag{4}$$

Queries

A query $\mathbf{q}$ is represented in exactly the same way as a document in the document collection. The term weight w_{iq} of each term t_i in $\mathbf{q}$ can also be computed in the same way as in a normal document, or slightly differently. For example, Salton and Buckley [470] suggested the following:

$$w_{iq} = \left(0.5 + \frac{0.5 f_{iq}}{\max\{f_{1q}, f_{2q}, ..., f_{|V|q}\}}\right) \times \log \frac{N}{df_i}. \tag{5}$$

Document Retrieval and Relevance Ranking

It is often difficult to make a binary decision on whether a document is relevant to a given query. Unlike the Boolean model, the vector space model does not make such a decision. Instead, the documents are ranked according to their degrees of relevance to the query. One way to compute the degree of relevance is to calculate the similarity of the query **q** to each document $\mathbf{d}_j$ in the document collection *D*. There are many similarity measures. The most well known one is the **cosine similarity**, which is the cosine of the angle between the query vector **q** and the document vector $\mathbf{d}_j$,

$$cosine(\mathbf{d}_j, \mathbf{q}) = \frac{\langle \mathbf{d}_j \bullet \mathbf{q} \rangle}{\| \mathbf{d}_j \| \times \| \mathbf{q} \|} = \frac{\sum_{i=1}^{|V|} w_{ij} \times w_{iq}}{\sqrt{\sum_{i=1}^{|V|} w_{ij}^2} \times \sqrt{\sum_{i=1}^{|V|} w_{iq}^2}}. \tag{6}$$

Cosine similarity is also widely used in text/document clustering.

The dot product of the two vectors is another similarity measure,

$$sim(\mathbf{d}_j, \mathbf{q}) = \langle \mathbf{d}_j \bullet \mathbf{q} \rangle. \tag{7}$$

Ranking of the documents is done using their similarity values. The top ranked documents are regarded as more relevant to the query.

Another way to assess the degree of relevance is to directly compute a relevance score for each document to the query. The **Okapi** method and its variations are popular techniques in this setting. The Okapi retrieval formula given here is based on that in [465, 493]. It has been shown that Okapi variations are more effective than cosine for short query retrieval.

Since it is easier to present the formula directly using the "bag" of words notation of documents than vectors, document $\mathbf{d}_j$ will be denoted by d_j and query **q** will be denoted by *q*. Additional notations are as follows:

t_i is a term
f_{ij} is the raw frequency count of term t_i in document d_j
f_{iq} is the raw frequency count of term t_i in query q
N is the total number of documents in the collection
df_i is the number of documents that contain the term t_i
dl_j is the document length (in bytes) of d_j
$avdl$ is the average document length of the collection

The Okapi relevance score of a document d_j for a query q is:

$$okapi(d_j, q) = \sum_{t_i \in q, d_j} \ln \frac{N - df_i + 0.5}{df_i + 0.5} \times \frac{(k_1 + 1) f_{ij}}{k_1 (1 - b + b \frac{dl_j}{avdl}) + f_{ij}} \times \frac{(k_2 + 1) f_{iq}}{k_2 + f_{iq}}, \quad (8)$$

where k_1 (between 1.0-2.0), b (usually 0.75) and k_2 (between 1-1000) are parameters.

Yet another score function is the **pivoted normalization weighting** score function, denoted by pnw [493]:

$$pnw(d_j, q) = \sum_{t_i \in q, d_j} \frac{1 + \ln(1 + \ln(f_{ij}))}{(1 - s) + s \frac{dl_j}{avdl}} \times f_{iq} \times \ln \frac{N+1}{df_i}, \quad (9)$$

where s is a parameter (usually set to 0.2). Note that these are empirical functions based on intuitions and experimental evaluations. There are many variations of these functions used in practice.

6.2.3 Statistical Language Model

Statistical language models (or simply **language models**) are based on probability and have foundations in statistical theory. The basic idea of this approach to retrieval is simple. It first estimates a language model for each document, and then ranks documents by the likelihood of the query given the language model. Similar ideas have previously been used in natural language processing and speech recognition. The formulation and discussion in this section is based on those in [595, 596]. Information retrieval using language models was first proposed by Ponte and Croft [448].

Let the query q be a sequence of terms, $q = q_1q_2…q_m$ and the document collection D be a set of documents, $D = \{d_1, d_2, …, d_N\}$. In the language modeling approach, we consider the probability of a query q as being "generated" by a probabilistic model based on a document d_j, i.e., $\Pr(q|d_j)$. To rank documents in retrieval, we are interested in estimating the posterior probability $\Pr(d_j|q)$. Using the Bayes rule, we have

$$\Pr(d_j \mid q) = \frac{\Pr(q \mid d_j)\Pr(d_j)}{\Pr(q)} \quad (10)$$

For ranking, $\Pr(q)$ is not needed as it is the same for every document. $\Pr(d_j)$ is usually considered uniform and thus will not affect ranking. We only need to compute $\Pr(q|d_j)$.

The language model used in most existing work is based on unigram, i.e., only individual terms (words) are considered. That is, the model assumes that each term (word) is generated independently, which is essentially a multinomial distribution over words. The general case is the *n*-gram model, where the *n*th term is conditioned on the previous *n*-1 terms.

Based on the multinomial distribution and the unigram model, we have

$$\Pr(q = q_1 q_2 \ldots q_m \mid d_j) = \prod_{i=1}^{m} \Pr(q_i \mid d_j) = \prod_{i=1}^{|V|} \Pr(t_i \mid d_j)^{f_{iq}}, \tag{11}$$

where f_{iq} is the number of times that term t_i occurs in q, and $\sum_{i=1}^{|V|} \Pr(t_i \mid d_j) = 1$. The retrieval problem is reduced to estimating $\Pr(t_i|d_j)$, which can be the relative frequency,

$$\Pr(t_i \mid d_j) = \frac{f_{ij}}{|d_j|}. \tag{12}$$

Recall that f_{ij} is the number of times that term t_i occurs in document d_j. $|d_j|$ denotes the total number of words in d_j.

However, one problem with this estimation is that a term that does not appear in d_j has the probability of 0, which underestimates the probability of the unseen term in the document. This situation is similar to text classification using the naïve Bayesian model (see Sect. 3.7). A non-zero probability is typically assigned to each unseen term in the document, which is called **smoothing**. Smoothing adjusts the estimates of probabilities to produce more accurate probabilities. The name smoothing comes from the fact that these techniques tend to make distributions more uniform, by adjusting low probabilities such as zero probabilities upward, and high probabilities downward. Not only do smoothing methods aim to prevent zero probabilities, but they also attempt to improve the accuracy of the model as a whole. Traditional additive smoothing is

$$\Pr_{add}(t_i \mid d_j) = \frac{\lambda + f_{ij}}{\lambda |V| + |d_j|}. \tag{13}$$

When $\lambda = 1$, it is the **Laplace smoothing** and when $0 < \lambda < 1$, it is the **Lidstone smoothing**. Many other more sophisticated smoothing methods can be found in [97, 596].

6.3 Relevance Feedback

To improve the retrieval effectiveness, researchers have proposed many techniques. Relevance feedback is one of the effective ones. It is a process where the user identifies some relevant and irrelevant documents in the initial list of retrieved documents, and the system then creates an expanded query by extracting some additional terms from the sample relevant and irrelevant documents for a second round of retrieval. The system may also produce a classification model using the user-identified relevant and irrelevant documents to classify the documents in the document collection into relevant and irrelevant documents. The relevance feedback process may be repeated until the user is satisfied with the retrieved result.

The Rocchio Method

This is one of the early and effective relevance feedback algorithms. It is based on the first approach above. That is, it uses the user-identified relevant and irrelevant documents to expand the original query. The new (or expanded) query is then used to perform retrieval again.

Let the original query vector be **q**, the set of relevant documents selected by the user be D_r, and the set of irrelevant documents be D_{ir}. The expanded query $\mathbf{q}_e$ is computed as follows,

$$\mathbf{q}_e = \alpha\mathbf{q} + \frac{\beta}{|D_r|}\sum_{\mathbf{d}_r \in D_r}\mathbf{d}_r - \frac{\gamma}{|D_{ir}|}\sum_{\mathbf{d}_{ir} \in D_{ir}}\mathbf{d}_{ir}, \tag{14}$$

where α, β and γ are parameters. Equation (14) simply augments the original query vector **q** with additional terms from relevant documents. The original query **q** is still needed because it directly reflects the user's information need. Relevant documents are considered more important than irrelevant documents. The subtraction is used to reduce the influence of those terms that are not discriminative (i.e., they appear in both relevant and irrelevant documents), and those terms that appear in irrelevant documents only. The three parameters are set empirically. Note that a slight variation of the algorithm is one without the normalization of $|D_r|$ and $|D_{ir}|$. Both these methods are simple and efficient to compute, and usually produce good results.

Machine Learning Methods

Since we have a set of relevant and irrelevant documents, we can construct a classification model from them. Then the relevance feedback problem becomes a learning problem. Any supervised learning method may be used, e.g., naïve Bayesian classification and SVM. Similarity comparison with the original query is no longer needed.

In fact, a variation of the Rocchio method above, called the **Rocchio classification** method, can be used for this purpose too. Building a Rocchio classifier is done by constructing a prototype vector $\mathbf{c}_i$ for each class i, which is either *relevant* or *irrelevant* in this case (negative elements or components of the vector $\mathbf{c}_i$ are usually set to 0):

$$\mathbf{c}_i = \frac{\alpha}{|D_i|}\sum_{\mathbf{d}\in D_i}\frac{\mathbf{d}}{||\mathbf{d}||} - \frac{\beta}{|D - D_i|}\sum_{\mathbf{d}\in D-D_i}\frac{\mathbf{d}}{||\mathbf{d}||}, \tag{15}$$

where D_i is the set of documents of class i, and α and β are parameters. Using the TF-IDF term weighting scheme, $\alpha = 16$ and $\beta = 4$ usually work quite well.

In classification, cosine similarity is applied. That is, each test document $\mathbf{d}_t$ is compared with every prototype $\mathbf{c}_i$ based on cosine similarity. $\mathbf{d}_t$ is assigned to the class with the highest similarity value (Fig. 6.2).

```
   Algorithm
1    for each class $i$ do
2        construct its prototype vector $\mathbf{c}_i$ using Equation (15)
3    endfor
4    for each test document $\mathbf{d}_t$ do
5        the class of $\mathbf{d}_t$ is $\arg\max_i cosine(\mathbf{d}_t, \mathbf{c}_i)$
6    endfor
```

Fig. 6.2. Training and testing of a Rocchio classifier

Apart from the above classic methods, the following learning techniques are also applicable:

Learning from Labeled and Unlabeled Examples (LU Learning): Since the number of user-selected relevant and irrelevant documents may be small, it can be difficult to build an accurate classifier. However, unlabeled examples, i.e., those documents that are not selected by the user, can be utilized to improve learning to produce a more accurate classifier. This fits the LU learning model exactly (see Sect. 5.1). The user-selected relevant and irrelevant documents form the small labeled training set.

Learning from Positive and Unlabeled Examples (PU Learning): The two learning models mentioned above assume that the user can confidently identify both relevant and irrelevant documents. However, in some cases, the user only selects (or clicks) documents that he/she feels relevant based on the title or summary information (e.g., snippets in Web search), which are most likely to be true relevant documents, but does not indicate irrelevant documents. Those documents that are not selected by the user may not be treated as irrelevant because he/she has not seen them. Thus, they can only be regarded as unlabeled documents. This is called **implicit feedback**. In order to learn in this case, we can use PU learning, i.e., learning from positive and unlabeled examples (see Sect. 5.2). We regard the user-selected documents as positive examples, and unselected documents as unlabeled examples. Researchers have experimented with this approach in the Web search context and obtained good results [128].

Using Ranking SVM and Language Models: In the implicit feedback setting, a technique called **ranking SVM** is proposed in [260] to rank the unselected documents based on the selected documents. A language model based approach is also proposed in [487].

Pseudo-Relevance Feedback

Pseudo-relevance feedback is another technique used to improve retrieval effectiveness. Its basic idea is to extract some terms (usually frequent terms) from the top-ranked documents and add them to the original query to form a new query for a second round of retrieval. Again, the process can be repeated until the user is satisfied with the final results. The main difference between this method and the relevance feedback method is that in this method, the user is not involved in the process. The approach simply assumes that the top-ranked documents are likely to be relevant. Through query expansion, some relevant documents missed in the initial round can be retrieved to improve the overall performance. Clearly, the effectiveness of this method relies on the quality of the selected expansion terms.

6.4 Evaluation Measures

Precision and recall measures have been described in Chap. 3 on supervised learning, where each document is classified to a specific class. In IR and Web search, usually no decision is made on whether a document is relevant or irrelevant to a query. Instead, a ranking of the documents is produced for the user. This section studies how to evaluate such rankings.

Again, let the collection of documents in the database be D, and the total number of documents in D be N. Given a user query $\mathbf{q}$, the retrieval algorithm first computes relevance scores for all documents in D and then produce a ranking R_q of the documents based on the relevance scores, i.e.,

$$R_q: \ <\mathbf{d}_1^q, \mathbf{d}_2^q, ..., \mathbf{d}_N^q>, \tag{16}$$

where $\mathbf{d}_1^q \in D$ is the most relevant document to query $\mathbf{q}$ and $\mathbf{d}_N^q \in D$ is the most irrelevant document to query $\mathbf{q}$.

Let D_q ($\subseteq D$) be the set of actual relevant documents of query $\mathbf{q}$ in D. We can compute the precision and recall values at each $\mathbf{d}_i^q$ in the ranking.

Recall at rank position i or document $\mathbf{d}_i^q$ (denoted by $r(i)$) is the fraction of relevant documents from $\mathbf{d}_1^q$ to $\mathbf{d}_i^q$ in R_q. Let the number of relevant documents from $\mathbf{d}_1^q$ to $\mathbf{d}_i^q$ in R_q be s_i ($\leq |D_q|$) ($|D_q|$ is the size of D_q). Then,

$$r(i) = \frac{s_i}{|D_q|}. \tag{17}$$

Precision at rank position i or document $\mathbf{d}_i^q$ (denoted by $p(i)$) is the fraction of documents from $\mathbf{d}_1^q$ to $\mathbf{d}_i^q$ in R_q that are relevant:

$$p(i) = \frac{s_i}{i} \tag{18}$$

Example 1: We have a document collection D with 20 documents. Given a query $\mathbf{q}$, we know that eight documents are relevant to $\mathbf{q}$. A retrieval algorithm produces the ranking (of all documents in D) shown in Fig. 6.3.

Rank *i*	+/–	***p*(*i*)**	***r*(*i*)**
1	+	1/1 = 100%	1/8 = 13%
2	+	2/2 = 100%	2/8 = 25%
3	+	3/3 = 100%	3/8 = 38%
4	–	3/4 = 75%	3/8 = 38%
5	+	4/5 = 80%	4/8 = 50%
6	–	4/6 = 67%	4/8 = 50%
7	+	5/7 = 71%	5/8 = 63%
8	–	5/8 = 63%	5/8 = 63%
9	+	6/9 = 67%	6/8 = 75%
10	+	7/10 = 70%	7/8 = 88%
11	–	7/11 = 63%	7/8 = 88%
12	–	7/12 = 58%	7/8 = 88%
13	+	8/13 = 62%	8/8 = 100%
14	–	8/14 = 57%	8/8 = 100%
15	–	8/15 = 53%	8/8 = 100%
16	–	8/16 = 50%	8/8 = 100%
17	–	8/17 = 53%	8/8 = 100%
18	–	8/18 = 44%	8/8 = 100%
19	–	8/19 = 42%	8/8 = 100%
20	–	8/20 = 40%	8/8 = 100%

Fig. 6.3. Precision and recall values at each rank position

In column 1 of Fig. 6.3, 1 represents the highest rank and 20 represents the lowest rank. "+" and "–" in column 2 indicate a relevant document and an irrelevant document respectively. The precision ($p(i)$) and recall ($r(i)$) values at each position i are given in columns 3 and 4. ■

Average Precision: Sometimes we want a single precision to compare different retrieval algorithms on a query **q**. An average precision (p_{avg}) can be computed based on the precision at each relevant document in the ranking,

$$p_{avg} = \frac{\sum_{d_i^q \in D_q} p(i)}{|D_q|}. \quad (19)$$

For the ranking in Fig. 6.3 of Example 1, the average precision is 81%:

$$p_{avg} = \frac{100\% + 100\% + 100\% + 80\% + 71\% + 67\% + 70\% + 62\%}{8} = 81\%. \quad (20)$$

Precision–Recall Curve: Based on the precision and recall values at each rank position, we can draw a precision–recall curve where the *x*-axis is the recall and the *y*-axis is the precision. Instead of using the precision and recall at each rank position, the curve is commonly plotted using 11 standard recall levels, 0%, 10%, 20%, …, 100%.

Since we may not obtain exactly these recall levels in the ranking, interpolation is needed to obtain the precisions at these recall levels, which is done as follows: Let r_i be a recall level, $i \in \{0, 1, 2, \ldots, 10\}$, and $p(r_i)$ be the precision at the recall level r_i. $p(r_i)$ is computed with

$$p(r_i) = \max_{r_i \leq r \leq r_{10}} p(r). \quad (21)$$

That is, to interpolate precision at a particular recall level r_i, we take the maximum precision of all recalls between level r_i and level r_{10}.

Example 2: Following Example 1, we obtain the interpolated precisions at all 11 recall levels in the table of Fig. 6.4. The precision-recall curve is shown on the right.

i	$p(r_i)$	r_i
0	100%	0%
1	100%	10%
2	100%	20%
3	100%	30%
4	80%	40%
5	80%	50%
6	71%	60%
7	70%	70%
8	70%	80%
9	62%	90%
10	62%	100%

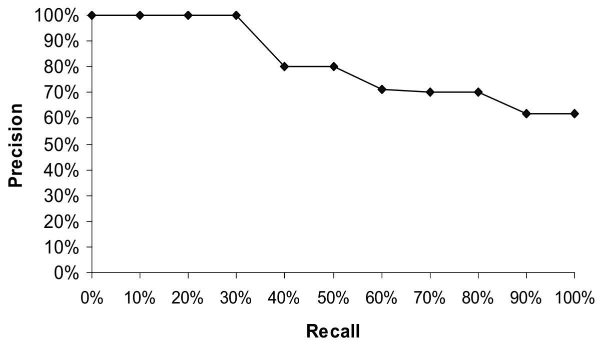


Fig. 6.4. The precision-recall curve ■

Comparing Different Algorithms: Frequently, we need to compare the retrieval results of different algorithms. We can draw their precision-recall curves together in the same figure for comparison. Figure 6.5 shows the curves of two algorithms on the same query and the same document collection. We observe that the precisions of one algorithm are better than those of the other at low recall levels, but are worse at high recall levels.

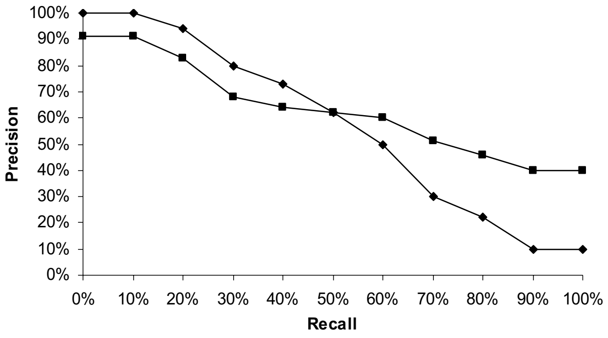


Fig. 6.5. Comparison of two retrieval algorithms based on their precision-recall curves

Evaluation Using Multiple Queries: In most retrieval evaluations, we are interested in the performance of an algorithm on a large number of queries. The overall precision (denoted by $\overline{p}(r_i)$) at each recall level r_i is computed as the average of individual precisions at that recall level, i.e.,

$$\overline{p}(r_i) = \frac{1}{|Q|}\sum_{j=1}^{|Q|} p_j(r_i), \tag{22}$$

where Q is the set of all queries and $p_j(r_i)$ is the precision of query j at the recall level r_i. Using the average precision at each recall level, we can also draw a precision-recall curve.

Although in theory precision and recall do not depend on each other, in practice a high recall is almost always achieved at the expense of precision, and a high precision is achieved at the expense of recall. Thus, precision and recall has a trade-off. Depending on the application, one may want a high precision or a high recall.

One problem with precision and recall measures is that, in many applications, it can be very hard to determine the set of relevant documents D_q for each query **q**. For example, on the Web, D_q is almost impossible to determine because there are simply too many pages to manually inspect. Without D_q, the recall value cannot be computed. In fact, recall does not make much sense for Web search because the user seldom looks at pages ranked below 30. However, precision is critical, and it can be estimated for top ranked documents. Manual inspection of only the top 30 pages is reasonable. The following precision computation is commonly used.

Rank Precision: We compute the precision values at some selected rank positions. For a Web search engine, we usually compute precisions for the top 5, 10, 15, 20, 25 and 30 returned pages (as the user seldom looks at more than 30 pages). We assume that the number of relevant pages is more than 30. Following Example 1, we have $p(5) = 80\%$, $p(10) = 70\%$, $p(15) = 53\%$, and $p(20) = 40\%$.

We should note that precision is not the only measure for evaluating search ranking, reputation or quality of the top ranked pages are also very important as we will see later in this chapter and also in Chap. 7.

F-score: Another often used evaluation measure is the F-score, which we have used in Chap. 3. Here we can compute the F-score at each rank position *i*. Recall that F-score is the harmonic mean of precision and recall:

$$F(i) = \frac{2}{\frac{1}{r(i)} + \frac{1}{p(i)}} = \frac{2p(i)r(i)}{p(i)+r(i)}. \quad (23)$$

Finally, the precision and recall **breakeven point** is also a commonly used measure, which we have discussed in Sect. 3.3.2 in Chap. 3.

6.5 Text and Web Page Pre-Processing

Before the documents in a collection are used for retrieval, some pre-processing tasks are usually performed. For traditional text documents (no HTML tags), the tasks are stopword removal, stemming, and handling of digits, hyphens, punctuations, and cases of letters. For Web pages, additional tasks such as HTML tag removal and identification of main content blocks also require careful considerations. We discuss them in this section.

6.5.1 Stopword Removal

Stopwords are frequently occurring and insignificant words in a language that help construct sentences but do not represent any content of the documents. Articles, prepositions and conjunctions and some pronouns are natural candidates. Common stopwords in English include:

a, about, an, are, as, at, be, by, for, from, how, in, is, of, on, or, that, the, these, this, to, was, what, when, where, who, will, with

Such words should be removed before documents are indexed and stored. Stopwords in the query are also removed before retrieval is performed.

6.5.2 Stemming

In many languages, a word has various syntactical forms depending on the contexts that it is used. For example, in English, nouns have plural forms, verbs have gerund forms (by adding "*ing*"), and verbs used in the past tense are different from the present tense. These are considered as syntactic variations of the same root form. Such variations cause low recall for a retrieval system because a relevant document may contain a variation of a query word but not the exact word itself. This problem can be partially dealt with by **stemming**.

Stemming refers to the process of reducing words to their stems or roots. A **stem** is the portion of a word that is left after removing its prefixes and suffixes. In English, most variants of a word are generated by the introduction of suffixes (rather than prefixes). Thus, stemming in English usually means **suffix removal,** or **stripping**. For example, "computer", "computing", and "compute" are reduced to "comput". "walks", "walking" and "walker" are reduced to "walk". Stemming enables different variations of the word to be considered in retrieval, which improves the recall. There are several stemming algorithms, also known as **stemmers**. In English, the most popular stemmer is perhaps the Martin Porter's stemming algorithm [449], which uses a set of rules for stemming.

Over the years, many researchers evaluated the advantages and disadvantages of using stemming. Clearly, stemming increases the recall and reduces the size of the indexing structure. However, it can hurt precision because many irrelevant documents may be considered relevant. For example, both "cop" and "cope" are reduced to the stem "cop". However, if one is looking for documents about police, a document that contains only "cope" is unlikely to be relevant. Although many experiments have been conducted by researchers, there is still no conclusive evidence one way or the other. In practice, one should experiment with the document collection at hand to see whether stemming helps.

6.5.3 Other Pre-Processing Tasks for Text

Digits: Numbers and terms that contain digits are removed in traditional IR systems except some specific types, e.g., dates, times, and other pre-specified types expressed with regular expressions. However, in search engines, they are usually indexed.

Hyphens: Breaking hyphens are usually applied to deal with inconsistency of usage. For example, some people use "state-of-the-art", but others use "state of the art". If the hyphens in the first case are removed, we eliminate the inconsistency problem. However, some words may have a hyphen as an integral part of the word, e.g., "Y-21". Thus, in general, the system can follow a general rule (e.g., removing all hyphens) and also have some exceptions. Note that there are two types of removal, i.e., (1) each hyphen is replaced with a space and (2) each hyphen is simply removed without leaving a space so that "state-of-the-art" may be replaced with "state of the art" or "stateoftheart". In some systems both forms are indexed as it is hard to determine which is correct, e.g., if "pre-processing" is converted to "pre processing", then some relevant pages will not be found if the query term is "preprocessing".

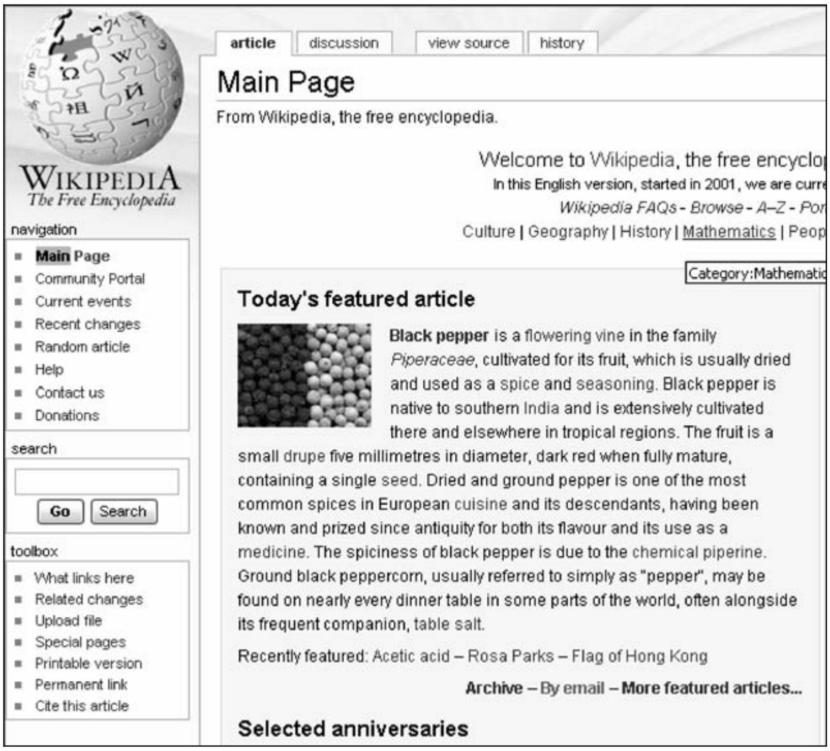
Punctuation Marks: Punctuation can be dealt with similarly as hyphens.

Case of Letters: All the letters are usually converted to either the upper or lower case.

6.5.4 Web Page Pre-Processing

We have indicated at the beginning of the section that Web pages are different from traditional text documents. Thus, additional pre-processing is needed. We describe some important ones below.

1. **Identifying different text fields:** In HTML, there are different text fields, e.g., title, metadata, and body. Identifying them allows the retrieval system to treat terms in different fields differently. For example, in search engines terms that appear in the title field of a page are regarded as more important than terms that appear in other fields and are assigned higher weights because the title is usually a concise description of the page. In the body text, those emphasized terms (e.g., under header tags <h1>, <h2>, …, bold tag <b>, etc.) are also given higher weights.
2. **Identifying anchor text:** Anchor text associated with a hyperlink is treated specially in search engines because the anchor text often represents a more accurate description of the information contained in the page pointed to by its link. In the case that the hyperlink points to an external page (not in the same site), it is especially valuable because it is a summary description of the page given by other people rather than the author/owner of the page, and is thus more trustworthy.
3. **Removing HTML tags:** The removal of HTML tags can be dealt with similarly to punctuation. One issue needs careful consideration, which affects proximity queries and phrase queries. HTML is inherently a vis-


WIKIPEDIA
The Free Encyclopedia

article | discussion | view source | history

Main Page

From Wikipedia, the free encyclopedia.

Welcome to Wikipedia, the free encyclo
In this English version, started in 2001, we are curr
Wikipedia FAQs - Browse - A–Z - Por
Culture | Geography | History | Mathematics | Peop

navigation
- Main Page
- Community Portal
- Current events
- Recent changes
- Random article
- Help
- Contact us
- Donations

search
Go | Search

toolbox
- What links here
- Related changes
- Upload file
- Special pages
- Printable version
- Permanent link
- Cite this article

Category:Mathemati

Today's featured article

Black pepper is a flowering vine in the family *Piperaceae*, cultivated for its fruit, which is usually dried and used as a spice and seasoning. Black pepper is native to southern India and is extensively cultivated there and elsewhere in tropical regions. The fruit is a small drupe five millimetres in diameter, dark red when fully mature, containing a single seed. Dried and ground pepper is one of the most common spices in European cuisine and its descendants, having been known and prized since antiquity for both its flavour and its use as a medicine. The spiciness of black pepper is due to the chemical piperine. Ground black peppercorn, usually referred to simply as "pepper", may be found on nearly every dinner table in some parts of the world, often alongside its frequent companion, table salt.

Recently featured: Acetic acid – Rosa Parks – Flag of Hong Kong

Archive – By email – **More featured articles...**

Selected anniversaries

Fig. 6.6. An example of a Web page from Wikipedia

ual presentation language. In a typical commercial page, information is presented in many rectangular blocks (see Fig. 6.6). Simply removing HTML tags may cause problems by joining text that should not be joined. For example, in Fig. 6.6, "cite this article" at the bottom of the left column will join "Main Page" on the right, but they should not be joined. They will cause problems for phrase queries and proximity queries. This problem had not been dealt with satisfactorily by search engines at the time when this book was written.

4. **Identifying main content blocks:** A typical Web page, especially a commercial page, contains a large amount of information that is not part of the main content of the page. For example, it may contain banner ads, navigation bars, copyright notices, etc., which can lead to poor results for search and mining. In Fig. 6.6, the main content block of the page is the block containing "Today's featured article." It is not desirable to index anchor texts of the navigation links as a part of the content of this page. Several researchers have studied the problem of identifying main content blocks. They showed that search and data mining results can be

improved significantly if only the main content blocks are used. We briefly discuss two techniques for finding such blocks in Web pages.

Partitioning based on visual cues: This method uses visual information to help find main content blocks in a page. Visual or rendering information of each HTML element in a page can be obtained from the Web browser. For example, Internet Explorer provides an API that can output the *X* and *Y* coordinates of each element. A machine learning model can then be built based on the location and appearance features for identifying main content blocks of pages. Of course, a large number of training examples need to be manually labeled (see [77, 495] for details).

Tree matching: This method is based on the observation that in most commercial Web sites pages are generated by using some fixed templates. The method thus aims to find such hidden templates. Since HTML has a nested structure, it is thus easy to build a tag tree for each page. **Tree matching** of multiple pages from the same site can be performed to find such templates. In Chap. 9, we will describe a tree matching algorithm for this purpose. Once a template is found, we can identify which blocks are likely to be the main content blocks based on the following observation: the text in main content blocks are usually quite different across different pages of the same template, but the non-main content blocks are often quite similar in different pages. To determine the text similarity of corresponding blocks (which are sub-trees), the **shingle method** described in the next section can be used.

6.5.5 Duplicate Detection

Duplicate documents or pages are not a problem in traditional IR. However, in the context of the Web, it is a significant issue. There are different types of duplication of pages and contents on the Web.

Copying a page is usually called **duplication** or **replication**, and copying an entire site is called **mirroring**. **Duplicate pages** and **mirror sites** are often used to improve efficiency of browsing and file downloading worldwide due to limited bandwidth across different geographic regions and poor or unpredictable network performances. Of course, some duplicate pages are the results of plagiarism. Detecting such pages and sites can reduce the index size and improve search results.

Several methods can be used to find duplicate information. The simplest method is to hash the whole document, e.g., using the MD5 algorithm, or computing an aggregated number (e.g., checksum). However, these methods are only useful for detecting exact duplicates. On the Web, one seldom finds exact duplicates. For example, even different mirror sites may have different URLs, different Web masters, different contact information, different advertisements to suit local needs, etc.

One efficient duplicate detection technique is based on **n-grams** (also called **shingles**). An *n*-gram is simply a consecutive sequence of words of a fixed window size *n*. For example, the sentence, "John went to school with his brother," can be represented with five 3-gram phrases "John went to", "went to school", "to school with", "school with his", and "with his brother". Note that 1-gram is simply the individual words.

Let $S_n(d)$ be the set of distinctive *n*-grams (or shingles) contained in document *d*. Each *n*-gram may be coded with a number or a MD5 hash (which is usually a 32-digit hexadecimal number). Given the *n*-gram representations of the two documents d_1 and d_2, $S_n(d_1)$ and $S_n(d_2)$, the **Jaccard coefficient** can be used to compute the similarity of the two documents,

$$sim(d_1, d_2) = \frac{|S_n(d_1) \cap S_n(d_2)|}{|S_n(d_1) \cup S_n(d_2)|}. \tag{24}$$

A threshold is used to determine whether d_1 and d_2 are likely to be duplicates of each other. For a particular application, the window size *n* and the similarity threshold are chosen through experiments.

6.6 Inverted Index and Its Compression

The basic method of Web search and traditional IR is to find documents that contain the terms in the user query. Given a user query, one option is to scan the document database sequentially to find the documents that contain the query terms. However, this method is obviously impractical for a large collection, such as the Web. Another option is to build some data structures (called **indices**) from the document collection to speed up retrieval or search. There are many index schemes for text [31]. The **inverted index**, which has been shown superior to most other indexing schemes, is a popular one. It is perhaps the most important index method used in search engines. This indexing scheme not only allows efficient retrieval of documents that contain query terms, but also very fast to build.

6.6.1 Inverted Index

In its simplest form, the inverted index of a document collection is basically a data structure that attaches each distinctive term with a list of all documents that contains the term. Thus, in retrieval, it takes constant time to find the documents that contains a query term. Finding documents containing multiple query terms is also easy as we will see later.

Given a set of documents, $D = \{d_1, d_2, \ldots, d_N\}$, and each document has a unique identifier (ID). An inverted index consists of two parts: a vocabulary V, containing all the distinct terms in the document set, and for each distinct term t_i an **inverted list** of postings. Each **posting** stores the ID (denoted by id_j) of the document d_j that contains term t_i and other pieces of information about term t_i in document d_j. Depending on the need of the retrieval or ranking algorithm, different pieces of information may be included. For example, to support phrase and proximity search, a posting for a term t_i usually consists of the following,

$$<id_j, f_{ij}, [o_1, o_2, \ldots, o_{|f_{ij}|}]>$$

where id_j is the ID of document d_j that contains the term t_i, f_{ij} is the frequency count of t_i in d_j, and o_k are the offsets (or positions) of term t_i in d_j. Postings of a term are sorted in increasing order based on the id_j's and so are the offsets in each posting (see Example 3). This facilitates compression of the inverted index as we will see in Sect. 6.6.4.

Example 3: We have three documents of id_1, id_2, and id_3:

```
id1: Web mining is useful.
     1   2      3  4
id2: Usage mining applications.
     1     2      3
id3: Web structure mining studies the Web hyperlink structure.
     1   2         3      4       5   6   7         8
```

The numbers below each document are the offset position of each word. The vocabulary is the set:

{Web, mining, useful, applications, usage, structure, studies, hyperlink}

Stopwords "is" and "the" have been removed, but no stemming is applied. Figure 6.7 shows two inverted indices.

Applications:	id_2
Hyperlink:	id_3
Mining:	id_1, id_2, id_3
Structure:	id_3
Studies:	id_3
Usage:	id_2
Useful:	id_1
Web:	id_1, id_3

(A)

Applications:	<id_2, 1, [3]>
Hyperlink:	<id_3, 1, [7]>
Mining:	<id_1, 1, [2]>, <id_2, 1, [2]>, <id_3, 1, [3]>
Structure:	<id_3, 2, [2, 8]>
Studies:	<id_3, 1, [4]>
Usage:	<id_2, 1, [1]>
Useful:	<id_1, 1, [4]>
Web:	<id_1, 1, [1]>, <id_3, 2, [1, 6]>

(B)

Fig. 6.7. Two inverted indices: a simple version and a more complex version

Figure 6.7(A) is a simple version, where each term is attached with only an inverted list of IDs of the documents that contain the term. Each inverted list in Fig. 6.7(B) is more complex as it contains additional information, i.e., the frequency count of the term and its positions in each document. Note that we use id_i as the document IDs to distinguish them from offsets. In an actual implementation, they may also be positive integers. Note also that a posting can contain other types of information depending on the need of the retrieval or search algorithm (see Sect. 6.8). ■

6.6.2 Search Using an Inverted Index

Queries are evaluated by first fetching the inverted lists of the query terms, and then processing them to find the documents that contain all (or some) terms. Specifically, given the query terms, searching for relevant documents in the inverted index consists of three main steps:

Step 1 (**vocabulary search**): This step finds each query term in the vocabulary, which gives the inverted list of each term. To speed up the search, the vocabulary usually resides in the main memory. Various indexing methods, e.g., hashing, tries or B-tree, can be used to speed up the search. Lexicographical ordering may also be employed due to its space efficiency. Then the binary search method can be applied. The complexity is $O(\log|V|)$, where $|V|$ is the vocabulary size.

If the query contains only a single term, this step gives all the relevant documents and the algorithm then goes to step 3. If the query contains multiple terms, the algorithm proceeds to step 2.

Step 2 (**results merging**): After the inverted list of each term is found, merging of the lists is performed to find their intersection, i.e., the set of documents containing all query terms. Merging simply traverses all the lists in synchronization to check whether each document contains all query terms. One main heuristic is to use the shortest list as the base to merge with the other longer lists. For each posting in the shortest list, a binary search may be applied to find it in each longer list. Note that partial match (i.e., documents containing only some of the query terms) can be achieved as well in a similar way, which is more useful in practice.

Usually, the whole inverted index cannot fit in memory, so part of it is cached in memory for efficiency. Determining which part to cache involves analysis of query logs to find frequent query terms. The inverted lists of these frequent query terms can be cached in memory.

Step 3 (**Rank score computation**): This step computes a rank (or relevance) score for each document based on a relevance function (e.g., okapi or cosine), which may also consider the phrase and term proximity information. The score is then used in the final ranking.

Example 4: Using the inverted index built in Fig. 6.7(B), we want to search for "web mining" (the query). In step 1, two inverted lists are found:

Mining:	<id_1, 1, [2]>, <id_2, 1, [2]>, <id_3, 1, [3]>
Web:	<id_1, 1, [1]>, <id_3, 2, [1, 6]>

In step 2, the algorithm traverses the two lists and finds documents containing both words (documents id_1 and id_3). The word positions are also retrieved. In step 3, we compute the rank scores. Considering the proximity and the sequence of words, we give id_1 a higher rank (or relevance) score than id_3 as "web" and "mining" are next to each other in id_1 and in the same sequence as that in the query. Different search engines may use different algorithms to combine these factors. ■

6.6.3 Index Construction

The construction of an inverted index is quite simple and can be done efficiently using a trie data structure among many others. The time complexity of the index construction is $O(T)$, where T is the number of all terms (including duplicates) in the document collection (after pre-processing).

For each document, the algorithm scans it sequentially and for each term, it finds the term in the trie. If it is found, the document ID and other information (e.g., the offset of the term) are added to the inverted list of the term. If the term is not found, a new leaf is created to represent the term.

Example 5: Let us build an inverted index for the three documents in Example 3, which are reproduced below for easy reference. Figure 6.8 shows the vocabulary trie and the inverted lists for all terms.

```
id1: Web mining is useful.
     1    2     3  4
id2: Usage mining applications.
     1     2      3
id3: Web structure mining studies the Web hyperlink structure
     1   2         3      4       5   6   7         8
```
■

To build the index efficiently, the trie is usually stored in memory. However, in the context of the Web, the whole index will not fit in the main memory. The following technique can be applied.

We follow the above algorithm to build the index until the memory is full. The partial index I_1 obtained so far is written on the disk. Then, we process the subsequent documents and build the partial index I_2 in memory, and so on. After all documents have been processed, we have k partial indices, I_1, I_2, …, I_k, on disk. We then merge the partial indices in a hierarchical manner. That is, we first perform pair-wise merges of I_1 and I_2, I_3 and I_4, and so on. This gives us larger indices $I_{1\text{-}2}$, $I_{3\text{-}4}$ and so on. After the first level merging is complete, we proceed to the second level merging, i.e., we merge $I_{1\text{-}2}$ and $I_{3\text{-}4}$, $I_{5\text{-}6}$ and $I_{7\text{-}8}$ and so on. This process continues until all the partial indices are merged into a single index. Each merge is fairly straightforward because the vocabulary in each partial index is sorted by the trie construction. The complexity of each merge is thus linear in the number of terms in both partial indices. Since each level needs a linear process of the whole index, the complete merging process takes O($k\log k$) time. To reduce the disk space requirement, whenever a new partial index is generated, we can merge it with a previously merged index. That is, when we have I_1 and I_2, we can merge them immediately to produce $I_{1\text{-}2}$, and when I_3 is produced, it is merged with $I_{1\text{-}2}$ to produce $I_{1\text{-}2\text{-}3}$ and so on.

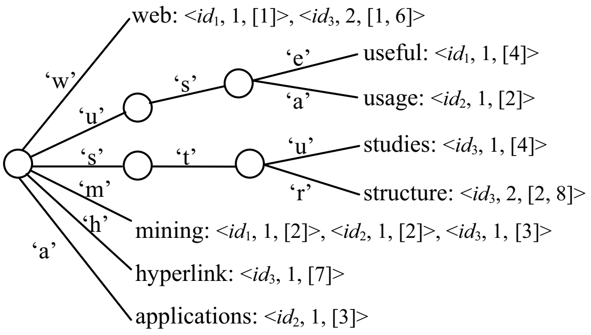

web: <id1, 1, [1]>, <id3, 2, [1, 6]>
'w'
'u'
's'
'e'
'a'
useful: <id1, 1, [4]>
usage: <id2, 1, [2]>
's'
't'
'u'
'r'
studies: <id3, 1, [4]>
structure: <id3, 2, [2, 8]>
'm'
'h'
'a'
mining: <id1, 1, [2]>, <id2, 1, [2]>, <id3, 1, [3]>
hyperlink: <id3, 1, [7]>
applications: <id2, 1, [3]>

Fig. 6.8. The vocabulary trie and the inverted lists

Instead of using a trie, an alternative method is to use an in-memory hash table (or other data structures) for terms. The algorithm is quite straightforward and will not be discussed further.

On the Web, an important issue is that pages are constantly added, modified or deleted. It may be quite inefficient to modify the main index because a single page change can require updates to a large number of records of the index. One simple solution is to construct two additional indices, one for added pages and one for deleted pages. Modification can be regarded as a deletion and then an addition. Given a user query, it is searched in the main index and also in the two auxiliary indices. Let the pages returned from the search in the main index be D_0, the pages returned from the search in the index of added pages be D_+ and the pages returned from the search in the index of deleted pages be $D_–$. Then, the final results returned to the user is $(D_0 \cup D_+) - D$. When the two auxiliary indices become too large, they can be merged into the main index.

6.6.4 Index Compression

An inverted index can be very large. In order to speed up the search, it should reside in memory as much as possible to avoid disk I/O. Because of this, reducing the index size becomes an important issue. A natural solution to this is **index compression**, which aims to represent the same information with fewer bits or bytes. Using compression, the size of an inverted index can be reduced dramatically. In the lossless compression, the original index can also be reconstructed exactly using the compressed version. Lossless compression methods are the focus of this section.

The inverted index is quite amiable to compression. Since the main space used by an inverted index is for the storage of document IDs and offsets of each term, we thus want to reduce this space requirement. Since all the information is represented with positive integers, we only discuss **integer compression** techniques in this section.

Without compression, on most architectures an integer has a fixed-size representation of four bytes (32 bits). However, few integers need 4 bytes to represent, so a more compact representation (compression) is clearly possible. There are generally two classes of compression schemes for inverted lists: the **variable-bit** scheme and the **variable-byte** scheme.

In the variable-bit (also called **bitwise**) scheme, an integer is represented with an integral number of bits. Well known bitwise methods include **unary coding**, **Elias gamma coding** and **delta coding** [161], and **Golomb coding** [202]. In the variable-byte scheme, an integer is stored in an integral number of bytes, where each byte has 8 bits. A simple bytewise scheme is the variable-byte coding [547]. These coding schemes basically map integers onto self-delimiting binary codewords (bits), i.e., the start bit and the end bit of each integer can be detected with no additional delimiters or markers.

An interesting feature of the inverted index makes compression even more effective. Since document IDs in each inverted list are sorted in increasing order, we can store the difference between any two adjacent document IDs, id_i and id_{i+1}, where $id_{i+1} > id_i$, instead of the actual IDs. This difference is called the **gap** between id_i and id_{i+1}. The gap is a smaller number than id_{i+1} and thus requires fewer bits. In search, if the algorithm linearly traverses each inverted list, document IDs can be recovered easily. Since offsets in each posting are also sorted, they can be stored similarly.

For example, the sorted document IDs are: 4, 10, 300, and 305. They can be represented with gaps, 4, 6, 290 and 5. Given the gap list 4, 6, 290 and 5, it is easy to recover the original document IDs, 4, 10, 300, and 305. We note that for frequent terms (which appear in a large number of documents) the gaps are small and can be encoded with short codes (fewer bits). For infrequent or rare terms, the gaps can be large, but they do not use up much space due to the fact that only a small number of documents contain them. Storing gaps can significantly reduce the index size.

We now discuss each of the coding schemes in detail. Each scheme includes a method for **coding** (or **compression**) and a method for **decoding** (**decompression**).

Unary Coding

Unary coding is simple. It represents a number x with x–1 bits of zeros followed by a bit of one. For example, 5 is represented as 00001. The one bit is simply the delimitor. Decoding is also straightforward. This scheme is effective for very small numbers, but wasteful for large numbers. It is thus seldom used alone in practice.

Table 6.1 shows example codes of different coding schemes for 10 decimal integers. Column 2 shows the unary code for each integer.

Table 6.1: Example codes for integers of different coding schemes: Spacing in the Elias, Golomb, and variable-byte codes separates the prefix of the code from the suffix.

Decimal	Unary	Elias Gamma	Elias Delta	Golomb (b = 3)	Golomb (b = 10)	Variable byte
1	1	1	1	1 10	1 001	0000001 0
2	01	0 10	0 100	1 11	1 010	0000010 0
3	001	0 11	0 101	01 0	1 011	0000011 0
4	0001	00 100	0 1100	01 10	1 100	0000100 0
5	00001	00 101	0 1101	01 11	1 101	0000101 0
6	000001	00 110	0 1110	001 0	1 1100	0000110 0
7	0000001	00 111	0 1111	001 10	1 1101	0000111 0
8	00000001	000 1000	00 100000	001 11	1 1110	0001000 0
9	000000001	000 1001	00 100001	0001 0	1 1111	0001001 0
10	0000000001	000 1010	00 100010	0001 10	01 000	0001010 0

Elias Gamma Coding

Coding: In the Elias gamma coding, a positive integer x is represented by: $1+\lfloor \log_2 x \rfloor$ in unary (i.e., $\lfloor \log_2 x \rfloor$ 0-bits followed by a 1-bit), followed by the binary representation of x without its most significant bit. Note that $1+\lfloor \log_2 x \rfloor$ is simply the number of bits of x in binary. The coding can also be described with the following two steps:

1. Write x in binary.
2. Subtract 1 from the number of bits written in step 1 and prepend that many zeros.

Example 6: The number 9 is represented by 0001001, since $1+\lfloor\log_2 9\rfloor = 4$, or 0001 in unary, and 9 is 001 in binary with the most significant bit removed. Alternatively, we first write 9 in binary, which is 1001 with 4 bits, and then prepend three zeros. In this way, 1 is represented by 1 (in one bit), and 2 is represented by 010. Additional examples are shown in column 3 of Table 6.1. ■

Decoding: We decode an Elias gamma-coded integer in two steps:

1. Read and count zeroes from the stream until we reach the first one. Call this count of zeroes K.
2. Consider the one that was reached to be the first digit of the integer, with a value of 2^K, read the remaining K bits of the integer.

Example 7: To decompress 0001001, we first read all zero bits from the beginning until we see a bit of 1. We have $K = 3$ zero bits. We then include the 1 bit with the following 3 bits, which give us 1001 (binary for 9). ■

Gamma coding is efficient for small integers but is not suited to large integers for which the parameterized Golomb code or the Elias delta code is more suitable.

Elias Delta Coding

Elias delta codes are somewhat longer than gamma codes for small integers, but for larger integers such as document numbers in an index of Web pages, the situation is reversed.

Coding: In the Elias delta coding, a positive integer x is stored with the gamma code representation of $1+\lfloor\log_2 x\rfloor$, followed by the binary representation of x less the most significant bit.

Example 8: Let us code the number 9. Since $1+\lfloor\log_2 x\rfloor = 4$, we have its gamma code 00100 for 4. Since 9's binary representation less the most significant bit is 001, we have the delta code of 00100001 for 9. Additional examples are shown in column 4 of Table 6.1. ■

Decoding: To decode an Elias delta-coded integer x, we first decode the gamma-code part $1+\lfloor\log_2 x\rfloor$ as the magnitude M (the number of bits of x in binary), and then retrieve the binary representation of x less the most significant bit. Specifically, we use the following steps:

1. Read and count zeroes from the stream until you reach the first one. Call this count of zeroes L.
2. Considering the one that was reached to be the first bit of an integer, with a value of 2^L, read the remaining L digits of the integer. This is the

integer M.
3. Put a one in the first place of our final output, representing the value 2^M. Read and append the following M-1 bits.

Example 9: We want to decode 00100001. We can see that $L = 2$ after step 1, and after step 2, we have read and consumed 5 bits. We also obtain $M = 4$ (100 in binary). Finally, we prepend 1 to the M-1 bits (which is 001) to give 1001, which is 9 in binary. ■

While Elias codes yield acceptable compression and fast decoding, a better performance in both aspects is possible with the Golomb coding.

Golomb Coding

The Golomb coding is a form of parameterized coding in which integers to be coded are stored as values relative to a constant b. Several variations of the original Golomb scheme exist, which save some bits in coding compared to the original scheme. We describe one version here.

Coding: A positive integer x is represented in two parts:
1. The first part is a unary representation of q+1, where q is the quotient $\lfloor(x/b)\rfloor$, and
2. The second part is a special binary representation of the remainder $r = x-qb$. Note that there are b possible remainders. For example, if $b = 3$, the possible remainders will be 0, 1, and 2.

The binary representation of a remainder requires $\lfloor\log_2 b\rfloor$ or $\lceil\log_2 b\rceil$ bits. Clearly, it is not possible to write every remainder in $\lfloor\log_2 b\rfloor$ bits in binary. To save space, we want to write the first few remainders using $\lfloor\log_2 b\rfloor$ bits and the rest using $\lceil\log_2 b\rceil$ bits. We must do so such that the decoder knows when $\lfloor\log_2 b\rfloor$ bits are used and when $\lceil\log_2 b\rceil$ bits are used. Let $i = \lfloor\log_2 b\rfloor$. We code the first d remainders using i bits,

$$d = 2^{i+1} - b. \qquad (25)$$

It is worth noting that these d remainders are all less than d. The rest of the remainders are coded with $\lceil\log_2 b\rceil$ bits and are all greater than or equal to d. They are coded using a special binary code (also called a **fixed prefix code**) with $\lceil\log_2 b\rceil$ (or i+1) bits.

Example 10: For $b = 3$, to code $x = 9$, we have the quotient $q = \lfloor 9/3\rfloor = 3$. For remainder, we have $i = \lfloor\log_2 3\rfloor = 1$ and $d = 1$. Note that for $b = 3$, there are three remainders, i.e., 0, 1, and 2, which are coded as 0, 10, and 11 respectively. The remainder for 9 is $r = 9 - 3 \times 3 = 0$. The final code for 9 is 00010. Additional examples for $b = 3$ are shown in column 5 of Table 6.1.

For $b = 10$, to code $x = 9$, we have the quotient $q = \lfloor 9/10 \rfloor = 0$. For remainder, we have $i = \lfloor \log_2 10 \rfloor = 3$ and $d = 6$. Note that for $b = 10$, there are 10 remainders, i.e., 0, 1, 2, …, 10, which are coded as 000, 001, 010, 011, 100, 101, 1100, 1101, 1110, 1111 respectively. The remainder of 9 is $r = 9 - 0 \times 5 = 9$. The final code for 9 is 11111. Additional examples for $b = 10$ are shown in column 6 of Table 6.1. ■

We can see that the first d remainders are standard binary codes, but the rest are not. They are generated using a tree instead. Figure 6.9 shows an example based on $b = 5$. The leaves are the five remainders. The first three remainders (0, 1, 2) are in the standard binary code, and the rest (3 and 4) have an additional bit. It is important to note that the first 2 bits ($i = 2$) of the remainder 3 (the first remainder coded in i+1 bits) is 11, which is 3 (i.e., d) in binary. This information is crucial for decoding because it enables the algorithm to know when i+1 bits are used. We also notice that d is completely determined by b, which helps decoding.

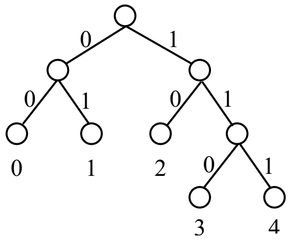


Fig. 6.9. The coding tree for $b = 5$

If b is a power of 2 (called **Golomb–Rice coding**), i.e., $b = 2^k$ for integer $k \geq 0$, every remainder is coded with the same number of bits because $\lfloor \log_2 b \rfloor = \lceil \log_2 b \rceil$. This is also easy to see from Equation (25), i.e., $d = 2^k$.

Decoding: To decode a Golomb-coded integer x, we use the following steps:

```
1. Decode unary-coded quotient $q$ (the relevant bits are comsumed).
2. Compute $i = \lfloor \log_2 b \rfloor$ and $d = 2^{i+1} - b$.
3. Retrieve the next $i$ bits and assign it to $r$.
4. If $r \geq d$ then
       retrieve one more bit and append it to $r$ at the end;
       $r = r - d$.
5. Return $x = qb + r$.
```

Some explanation is in order for step 4. As we discussed above, if $r \geq d$ we need i+1 bits to code the remainder. The first line of step 4 retrieves the additional bit and appends it to r. The second line obtains the true value of the remainder r.

Example 11: We want to decode 11111 for $b = 10$. We see that $q = 0$ because there is no zero at the beginning. The first bit is consumed. We know that $i = \lfloor \log_2 10 \rfloor = 3$ and $d = 6$. We then retrieve the next three bits, 111, which is 7 in decimal, and assign it to r (= 111). Since 7 > 6 (which is d), we retrieve one more bit, which is 1, and r is now 1111 (15 in decimal). The new $r = r - d = 15 - 6 = 9$. Finally, $x = qb + r = 0 + 9 = 9$. ■

Now we discuss the selection of b for each term. For gap compression, Witten et al. [551] reported that a suitable b is

$$b \approx 0.69\left(\frac{N}{n_t}\right), \tag{26}$$

where N is the total number of documents and n_t is the number of documents that contain term t.

Variable-Byte Coding

Coding: In this method, seven bits in each byte are used to code an integer, with the least significant bit set to 0 in the last byte, or to 1 if further bytes follow. In this way, small integers are represented efficiently. For example, 135 is represented in two bytes, since it lies in the range 2^7 and 2^{14}, as 00000011 00001110. Additional examples are shown in column 6 of Table 6.1.

Decoding: Decoding is performed in two steps:

1. Read all bytes until a byte with the zero last bit is seen.
2. Remove the least significant bit from each byte read so far and concatenate the remaining bits.

For example, 00000011 00001110 is decoded to 00000010000111, which is 135.

Finally, experimental results in [547] show that non-parameterized Elias coding is generally not as space-efficient or as fast as parameterized Golomb coding for retrieval. Gamma coding does not work well. Variable-byte integers are often faster than variable-bit integers, despite having higher storage costs, because fewer CPU operations are required to decode variable-byte integers and they are byte-aligned on disk. A suitable compression technique can allow retrieval to be up to twice as fast than without compression, while the space requirement averages 20% – 25% of the cost of storing uncompressed integers.

6.7 Latent Semantic Indexing

The retrieval models discussed so far are based on keyword or term matching, i.e., matching terms in the user query with those in the documents. However, many concepts or objects can be described in multiple ways (using different words) due to the context and people's language habits. If a user query uses different words from the words used in a document, the document will not be retrieved although it may be relevant because the document uses some symonyms of the words in the user query. This causes low recall. For example, "picture", "image" and "photo" are **synonyms** in the context of digital cameras. If the user query only has the word "picture", relevant documents that contain "image" or "photo" but not "picture" will not be retrieved.

Latent semantic indexing (LSI), proposed by Deerwester et al. [125], aims to deal with this problem through the identification of statistical associations of terms. It is assumed that there is some underlying latent semantic structure in the data that is partially obscured by the randomness of word choice. It then uses a statistical technique, called **singular value decomposition** (SVD) [203], to estimate this latent structure, and to remove the "noise". The results of this decomposition are descriptions of terms and documents based on the latent semantic structure derived from SVD. This structure is also called the **hidden "concept" space**, which associates syntactically different but semantically similar terms and documents. These transformed terms and documents in the "concept" space are then used in retrieval, not the original terms or documents. Furthermore, the query is also transformed into the "concept" space before retrieval.

Let D be the text collection, the number of distinctive words in D be m and the number of documents in D be n. LSI starts with an $m \times n$ term-document matrix $\boldsymbol{A}$. Each row of $\boldsymbol{A}$ represents a term and each column represents a document. The matrix may be computed in various ways, e.g., using term frequency or TF-IDF values. We use term frequency as an example in this section. Thus, each entry or cell of the matrix $\boldsymbol{A}$, denoted by A_{ij}, is the number of times that term i occurs in document j.

6.7.1 Singular Value Decomposition

What SVD does is to factor matrix $\boldsymbol{A}$ (a $m \times n$ matrix) into the product of three matrices, i.e.,

$$A = U\Sigma V^T, \tag{27}$$

where

U is a $m \times r$ matrix and its columns, called **right singular vectors***,* are eigenvectors associated with the r non-zero eigenvalues of $\boldsymbol{AA}^T$. Furthermore, the columns of ***U*** are unit orthogonal vectors, i.e., $\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{I}$ (identity matrix).

V is an $n \times r$ matrix and its columns, called **right singular vectors***,* are eigenvectors associated with the r non-zero eigenvalues of $\boldsymbol{A}^T\boldsymbol{A}$. The columns of ***V*** are also unit orthogonal vectors, i.e., $\boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{I}$.

$\boldsymbol{\Sigma}$ is a $r \times r$ diagonal matrix, $\boldsymbol{\Sigma} = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r)$, $\sigma_i > 0$. $\sigma_1, \sigma_2, \ldots$, and σ_r, called **singular values**, are the non-negative square roots of the r (non-zero) eigenvalues of $\boldsymbol{AA}^T$. They are arranged in decreasing order, i.e., $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$.

We note that initially ***U*** is in fact an $m \times m$ matrix and ***V*** an $n \times n$ matrix and $\boldsymbol{\Sigma}$ an $m \times n$ diagonal matrix. $\boldsymbol{\Sigma}$'s diagonal consists of nonnegative eigenvalues of $\boldsymbol{AA}^T$ or $\boldsymbol{A}^T\boldsymbol{A}$. However, due to zero eigenvalues, $\boldsymbol{\Sigma}$ has zero-valued rows and columns. Matrix multiplication tells us that those zero-valued rows and columns from $\boldsymbol{\Sigma}$ can be dropped. Then, the last $m-r$ columns in ***U*** and the last $n-r$ columns in ***V*** can also be dropped.

m is the number of row (terms) in ***A*,** representing the number of terms.
n is the number of columns in ***A***, representing the number of documents.
r is the **rank** of ***A****,* $r \leq \min(m, n)$.

The singular value decomposition of ***A*** always exists and is unique up to

1. allowable permutations of columns of ***U*** and ***V*** and elements of $\boldsymbol{\Sigma}$ leaving it still diagonal; that is, columns i and j of $\boldsymbol{\Sigma}$ may be interchanged *iff* row i and j of $\boldsymbol{\Sigma}$ are interchanged, and columns i and j of ***U*** and ***V*** are interchanged.
2. sign (+/–) flip in ***U*** and ***V***.

An important feature of SVD is that we can delete some insignificant dimensions in the transformed (or "concept") space to optimally (in the least square sense) approximate matrix ***A***. The significance of the dimensions is indicated by the magnitudes of the singular values in $\boldsymbol{\Sigma}$, which are already sorted. In the context of information retrieval, the insignificant dimensions may represent "noisy" in the data, and should be removed. Let us use only the k largest singular values in $\boldsymbol{\Sigma}$ and set the remaining small ones to zero. The approximated matrix of ***A*** is denoted by $\boldsymbol{A}_k$. We can also reduce the size of the matrices $\boldsymbol{\Sigma}$, ***U*** and ***V*** by deleting the last $r-k$ rows and columns from $\boldsymbol{\Sigma}$, the last $r-k$ columns in ***U*** and the last $r-k$ columns in ***V***. We then obtain

$$A_k = U_k \Sigma_k V_k^T, \tag{28}$$

which means that we use the k-largest singular triplets to approximate the original (and somewhat "noisy") term-document matrix A. The new space is called the ***k*-concept space**. Figure 6.10 shows the original matrices and the reduced matrices schematically.

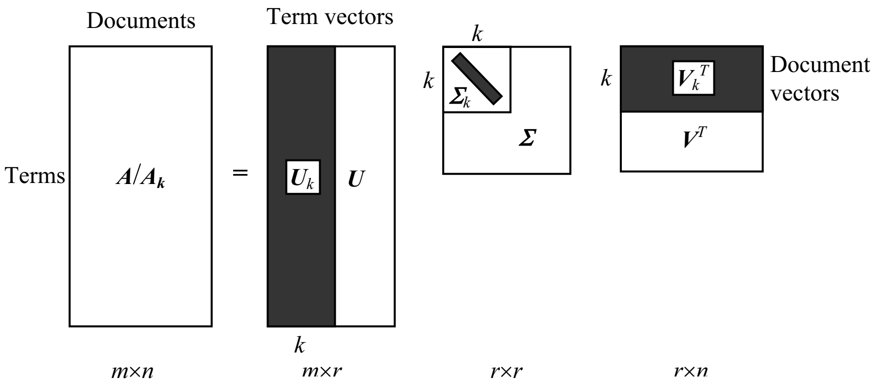


Fig. 6.10. The schematic representation of A and A_k

It is critical that the LSI method does not re-construct the original term-document matrix A perfectly. The truncated SVD captures most of the important underlying structures in the association of terms and documents, yet at the same time removes the noise or variability in word usage that plagues keyword matching retrieval methods.

Intuitive Idea of LSI: The intuition of LSI is that SVD rotates the axes of m-dimensional space of A such that the first axis runs along the largest variation (variance) among the documents, the second axis runs along the second largest variation (variance) and so on. Figure 6.11 shows an example.

The original x-y space is mapped to the x'-y' space generated by SVD. We can see that x and y are clearly correlated. In our retrieval context, each data point represents a document and each axis (x or y) in the original space represents a term. Hence, the two terms are correlated or co-occur frequently. In the SVD, the direction of x' in which the data has the largest variation is represented by the first column vector of U, and the direction of y' is represented by the second column vector of U. ΣV^T represents the documents in the transformed "concept" space. The singular values in Σ are simply scaling factors.

We observe that y' direction is insignificant, and may represent some "noise", so we can remove it. Then, every data point (document) is projected to x'. We have an outlier document $\mathbf{d}_i$ that contains term x, but not term y. However, if it is projected to x', it becomes closer to other points.

Let us see what happens if we have a query $\mathbf{q}$ represented with a star in Fig. 6.11, which contains only a single term "y". Using the traditional exact term matching, $\mathbf{d}_i$ is not relevant because "y" does not appear in $\mathbf{d}_i$. However, in the new space after projection, they are quite close or similar.

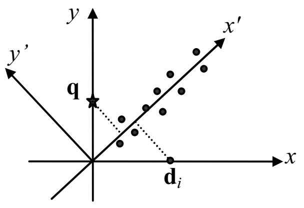


Fig. 6.11. Intuition of the LSI.

6.7.2 Query and Retrieval

Given a user query $\mathbf{q}$ (represented by a column vector as those in $\boldsymbol{A}$), it is first converted into a document in the k-concept space, denoted by $\mathbf{q}_k$. This transformation is necessary because SVD has transformed the original documents into the k-concept space and stored them in $\boldsymbol{V}_k$. The idea is that $\mathbf{q}$ is treated as a new document in the original space represented as a column in $\boldsymbol{A}$, and then mapped to $\mathbf{q}_k$ as an additional document (or column) in $\boldsymbol{V}_k^T$. From Equation (28), it is easy to see that

$$\mathbf{q} = \boldsymbol{U}_k \boldsymbol{\Sigma}_k \mathbf{q}_k^T. \tag{29}$$

Since the columns in $\boldsymbol{U}$ are unit orthogonal vectors, $\boldsymbol{U}_k^T \boldsymbol{U}_k = \boldsymbol{I}$. Thus,

$$\boldsymbol{U}_k^T \mathbf{q} = \boldsymbol{\Sigma}_k \mathbf{q}_k^T. \tag{30}$$

As the inverse of a diagonal matrix is still a diagonal matrix, and each entry on the diagonal is $1/\sigma_i$ ($1 \leq i \leq k$), if it is multiplied on both sides of Equation (30), we obtain,

$$\boldsymbol{\Sigma}_k^{-1} \boldsymbol{U}_k^T \mathbf{q} = \mathbf{q}_k^T. \tag{31}$$

Finally, we get the following (notice that the transpose of a diagonal matrix is itself),

$$\mathbf{q}_k = \mathbf{q}^T \boldsymbol{U}_k \boldsymbol{\Sigma}_k^{-1}. \tag{32}$$

For retrieval, we simply compare $\mathbf{q}_k$ with each document (row) in V_k using a similarity measure, e.g., the cosine similarity. Recall that each row of V_k (or each column of V_k^T) corresponds to a document (column) in A. This method has been used traditionally.

Alternatively, since $\Sigma_k V_k^T$ (not V_k^T) represents the documents in the transformed k-concept space, we can compare the similarity of the query document in the transformed space, which is $\Sigma_k\mathbf{q}_k^T$, and each transformed document in $\Sigma_k V_k^T$ for retrieval. The difference between the two methods is obvious. This latter method considers scaling effects of the singular values in Σ_k, but the former does not. However, it is not clear which method performs better as I know of no reported study on this alternative method.

6.7.3 An Example

Example 12: We will use the example in [125] to illustrate the process. The document collection has the following nine documents. The first five documents are related to human computer interaction, and the last four documents are related to graphs. To reduce the size of the problem, only the underlined terms are used in our computation.

c_1:	Human machine interface for Lab ABC computer applications
c_2:	A survey of user opinion of computer system response time
c_3:	The EPS user interface management system
c_4:	System and human system engineering testing of EPS
c_5:	Relation of user-perceived response time to error measurement
m_1:	The generation of random, binary, unordered trees
m_2:	The intersection graph of paths in trees
m_3:	Graph minors IV: Widths of trees and well-quasi-ordering
m_4:	Graph minors: A survey

The term-document matrix A is given below, which is a 9×12 matrix.

$$A = \begin{array}{ccccccccc|l}
c_1 & c_2 & c_3 & c_4 & c_5 & m_1 & m_2 & m_3 & m_4 & \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \textit{human} \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \textit{interface} \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \textit{computer} \\
0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & \textit{user} \\
0 & 1 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & \textit{system} \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \textit{response} \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \textit{time} \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & \textit{EPS} \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \textit{survey} \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & \textit{trees} \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \textit{graph} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & \textit{minors}
\end{array}$$

After performing SVD, we obtain three matrices, $\boldsymbol{U}$, $\boldsymbol{\Sigma}$ and $\boldsymbol{V}^T$, which are given below. Singular values on the diagonal of $\boldsymbol{\Sigma}$ are in decreasing order.

$$\boldsymbol{U} = \begin{pmatrix} 0.22 & -0.11 & 0.29 & -0.41 & -0.11 & -0.34 & 0.52 & -0.06 & -0.41 \\ 0.20 & -0.07 & 0.14 & -0.55 & 0.28 & 0.50 & -0.07 & -0.01 & -0.11 \\ 0.24 & 0.04 & -0.16 & -0.59 & -0.11 & -0.25 & -0.30 & 0.06 & 0.49 \\ 0.40 & 0.06 & -0.34 & 0.10 & 0.33 & 0.38 & 0.00 & 0.00 & 0.01 \\ 0.64 & -0.17 & 0.36 & 0.33 & -0.16 & -0.21 & -0.17 & 0.03 & 0.27 \\ 0.27 & 0.11 & -0.43 & 0.07 & 0.08 & -0.17 & 0.28 & -0.02 & -0.05 \\ 0.27 & 0.11 & -0.43 & 0.07 & 0.08 & -0.17 & 0.28 & -0.02 & -0.05 \\ 0.30 & -0.14 & 0.33 & 0.19 & 0.11 & 0.27 & 0.03 & -0.02 & -0.17 \\ 0.21 & 0.27 & -0.18 & -0.03 & -0.54 & 0.08 & -0.47 & -0.04 & -0.58 \\ 0.01 & 0.49 & 0.23 & 0.03 & 0.59 & -0.39 & -0.29 & 0.25 & -0.23 \\ 0.04 & 0.62 & 0.22 & 0.00 & -0.07 & 0.11 & 0.16 & -0.68 & 0.23 \\ 0.03 & 0.45 & 0.14 & -0.01 & -0.30 & 0.28 & 0.34 & 0.68 & 0.18 \end{pmatrix}$$

$$\boldsymbol{\Sigma} = \begin{pmatrix} 3.34 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2.54 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2.35 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.64 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.50 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.31 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.85 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.56 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.36 \end{pmatrix}$$

$$\boldsymbol{V}^T = \begin{pmatrix} 0.20 & -0.06 & 0.11 & -0.95 & 0.05 & -0.08 & 0.18 & -0.01 & -0.06 \\ 0.61 & 0.17 & -0.50 & -0.03 & -0.21 & -0.26 & -0.43 & 0.05 & 0.24 \\ 0.46 & -0.13 & 0.21 & 0.04 & 0.38 & 0.72 & -0.24 & 0.01 & 0.02 \\ 0.54 & -0.23 & 0.57 & 0.27 & -0.21 & -0.37 & 0.26 & -0.02 & -0.08 \\ 0.28 & 0.11 & -0.51 & 0.15 & 0.33 & 0.03 & 0.67 & -0.06 & -0.26 \\ 0.00 & 0.19 & 0.10 & 0.02 & 0.39 & -0.30 & -0.34 & 0.45 & -0.62 \\ 0.01 & 0.44 & 0.19 & 0.02 & 0.35 & -0.21 & -0.15 & -0.76 & 0.02 \\ 0.02 & 0.62 & 0.25 & 0.01 & 0.15 & 0.00 & 0.25 & 0.45 & 0.52 \\ 0.08 & 0.53 & 0.08 & -0.03 & -0.60 & 0.36 & 0.04 & -0.07 & -0.45 \end{pmatrix}$$

Now let us choose only two largest singular values from $\boldsymbol{\Sigma}$, i.e., $k = 2$. Thus, the concept space has only two dimensions. The other two matrices are also truncated accordingly. We obtain the 3 matrix $\boldsymbol{U}_k$, $\boldsymbol{\Sigma}_k$ and $\boldsymbol{V}_k^T$:

$$\boldsymbol{A}_k = \overset{\boldsymbol{U}_k}{\begin{bmatrix} 0.22 & -0.11 \\ 0.20 & -0.07 \\ 0.24 & 0.04 \\ 0.40 & 0.06 \\ 0.64 & -0.17 \\ 0.27 & 0.11 \\ 0.27 & 0.11 \\ 0.30 & -0.14 \\ 0.21 & 0.27 \\ 0.01 & 0.49 \\ 0.04 & 0.62 \\ 0.03 & 0.45 \end{bmatrix}} \overset{\boldsymbol{\Sigma}_k}{\begin{bmatrix} 3.34 & 0 \\ 0 & 2.54 \end{bmatrix}} \overset{\boldsymbol{V}_k^T}{\begin{bmatrix} 0.20 & 0.61 & 0.46 & 0.54 & 0.28 & 0.00 & 0.02 & 0.02 & 0.08 \\ -0.06 & 0.17 & -0.13 & -0.23 & 0.11 & 0.19 & 0.44 & 0.62 & 0.53 \end{bmatrix}}$$

Now we issue a search query **q**, "user interface", to find relevant documents. The transformed query document $\mathbf{q}_k$ of query **q** in the *k*-concept space is computed below using Equation (26), which is (0.179 -0.004).

$$\mathbf{q}_k = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}^T \begin{pmatrix} 0.22 & -0.11 \\ 0.20 & -0.07 \\ 0.24 & 0.04 \\ 0.40 & 0.06 \\ 0.64 & -0.17 \\ 0.27 & 0.11 \\ 0.27 & 0.11 \\ 0.30 & -0.14 \\ 0.21 & 0.27 \\ 0.01 & 0.49 \\ 0.04 & 0.62 \\ 0.03 & 0.45 \end{pmatrix} \begin{bmatrix} 3.34 & 0 \\ 0 & 2.54 \end{bmatrix}^{-1} = (0.179 \quad -0.004)$$

$\mathbf{q}_k$ is then compared with every document vector in $\boldsymbol{V}_k$ using the cosine similarity. The similarity values are as follows:

c_1:	0.964
c_2:	0.957
c_3:	0.968
c_4:	0.928
c_5:	0.922
m_1:	–0.022
m_2:	0.023
m_3:	0.010
m_4:	0.127

We obtain the final ranking of (c_3, c_1, c_2, c_4, c_5, m_4, m_2, m_3, m_1). ■

6.7.4 Discussion

LSI has been shown to perform better than traditional keywords based methods. The main drawback is the time complexity of the SVD, which is $O(m^2n)$. It is thus difficult to use for a large document collection such as the Web. Another drawback is that the concept space is not interpretable as its description consists of all numbers with little semantic meaning.

Determining the optimal number of dimensions *k* of the concept space is also a major difficulty. There is no general consensus for an optimal number of dimensions. The original paper [125] of LSI suggests 50–350 dimensions. In practice, the value of *k* needs to be determined based on the specific document collection via trial and error, which is a very time consuming process due to the high time complexity of the SVD.

To close this section, one can imagine that association rules may be able to approximate the results of LSI and avoid its shortcomings. Association rules represent term correlations or co-occurrences. Association rule mining has two advantages. First, its mining algorithm is very efficient. Since we may only need rules with 2-3 terms, which are sufficient for practical purposes, the mining algorithm only needs to scan the document collection 2-3 times. Second, rules are easy to understand. However, little research has been done in this direction so far.

6.8 Web Search

We now put it all together and describe the working of a search engine. Since it is difficult to know the internal details of a commercial search engine, most contents in this section are based on research papers, especially the early Google paper [68]. Due to the efficiency problem, latent semantic indexing is probably not used in Web search yet. Current search algorithms are still mainly based on the vector space model and term matching.

A search engine starts with the crawling of pages on the Web. The crawled pages are then parsed, indexed, and stored. At the query time, the index is used for efficient retrieval. We will not discuss crawling here. Its details can be found in Chap. 8. The subsequent operations of a search engine are described below:

Parsing: A parser is used to parse the input HTML page, which produces a stream of tokens or terms to be indexed. The parser can be constructed using a lexical analyzer generator such as YACC and Flex (which is from the GNU project). Some pre-processing tasks described in Sect. 6.5 may also be performed before or after parsing.

Indexing: This step produces an inverted index, which can be done using any of the methods described in Sect. 6.6. For retrieval efficiency, a search engine may build multiple inverted indices. For example, since the titles and anchor texts are often very accurate descriptions of the pages, a small inverted index may be constructed based on the terms appeared in them alone. Note that here the anchor text is for indexing the page that its link points to, not the page containing it. A full index is then built based on all the text in each page, including anchor texts (a piece of anchor text is indexed both for the page that contains it, and for the page that its link points to). In searching, the algorithm may search in the small index first and then the full index. If a sufficient number of relevant pages are found in the small index, the system may not search in the full index.

Searching and Ranking: Given a user query, searching involves the following steps:

1. pre-processing the query terms using some of the methods described in Sect. 6.5, e.g., stopword removal and stemming;
2. finding pages that contain all (or most of) the query terms in the inverted index;
3. ranking the pages and returning them to the user.

The ranking algorithm is the heart of a search engine. However, little is known about the algorithms used in commercial search engines. We give a general description based on the algorithm in the early Google system.

As we discussed earlier, traditional IR uses cosine similarity values or any other related measures to rank documents. These measures only consider the content of each document. For the Web, such content based methods are not sufficient. The problem is that on the Web there are too many relevant documents for almost any query. For example, using "web mining" as the query, the search engine Google estimated that there were 46,500,000 relevant pages. Clearly, there is no way that any user will look at this huge number of pages. Therefore, the issue is how to rank the pages and present the user the "best" pages at the top.

An important ranking factor on the Web is the quality of the pages, which was hardly studied in traditional IR because most documents used in IR evaluations are from reliable sources. However, on the Web, anyone can publish almost anything, so there is no quality control. Although a page may be 100% relevant, it may not be a quality page due to several reasons. For example, the author may not be an expert of the query topic, the information given in the page may be unreliable or biased, etc.

However, the Web does have an important mechanism, the hyperlinks (links), that can be used to assess the quality of each page to some extent. A link from page *x* to page *y* is an implicit conveyance of authority of page *x* to page *y*. That is, the author of page *x* believes that page *y* contains quality or **authoritative information**. One can also regard the fact that page *x* points to page *y* as a vote of page *x* for page *y*. This democratic nature of the Web can be exploited to assess the quality of each page. In general, the more votes a page receives, the more likely it is a **quality page**. The actual algorithms are more involved than simply counting the number of votes or links pointing to a page (called **in-links**). We will describe the algorithms in the next chapter. **PageRank** is the most well known such algorithm (see Sect. 7.3). It makes use of the link structure of Web pages to compute a quality or reputation score for each page. Thus, a Web page can be evaluated based on both its content factors and its reputation. Content-based evaluation depends on two kinds of information:

Occurrence Type: There are several types of occurrences of query terms in a page:

Title: a query term occurs in the title field of the page.

Anchor text: a query term occurs in the anchor text of a page pointing to the current page being evaluated.

URL: a query term occurs in the URL of the page. Many URL addresses contain some descriptions of the page. For example, a page on Web mining may have the URL http://www.domain.edu/Web-mining.html.

Body: a query term occurs in the body field of the page. In this case, the prominence of each term is considered. Prominence means whether the term is emphasized in the text with a large font, or bold and/or italic tags. Different prominence levels can be used in a system. Note that anchor texts in the page can be treated as plain texts for the evaluation of the page.

Count: The number of occurrences of a term of each type. For example, a query term may appear in the title field of the page 2 times. Then, the title count for the term is 2.

Position: This is the position of each term in each type of occurrence. The information is used in proximity evaluation involving multiple query terms. Query terms that are near to each other are better than those that are far apart. Furthermore, query teams appearing in the page in the same sequence as they are in the query are also better.

For the computation of the content based score (also called the **IR score**), each occurrence type is given an associated weight. All **type weights** form a fixed vector. Each raw term count is converted to a **count weight**, and all count weights also form a vector.

The quality or reputation of a page is usually computed based on the link structure of Web pages, which we will study in Chap. 7. Here, we assume that a reputation score has been computed for each page.

Let us now look at two kinds of queries, **single word queries** and **multi-word queries**. A single word query is the simplest case with only a single term. After obtaining the pages containing the term from the inverted index, we compute the dot product of the **type weight vector** and the **count weight vector** of each page, which gives us the IR score of the page. The IR score of each page is then combined with its **reputation score** to produce the final score of the page.

For a multi-word query, the situation is similar, but more complex since there is now the issue of considering term proximity and ordering. Let us simplify the problem by ignoring the term ordering in the page. Clearly, terms that occur close to each other in a page should be weighted higher than those that occur far apart. Thus multiple occurrences of terms need to be matched so that nearby terms are identified. For every matched set, a proximity value is calculated, which is based on how far apart the terms are in the page. Counts are also computed for every type and proximity. Each type and proximity pair has a type-proximity-weight. The counts are converted into count-weights. The dot product of the count-weights and the type-proximity-weights gives an IR score to the page. Term ordering can be considered similarly and included in the IR score, which is then combined with the page reputation score to produce the final rank score.

6.9 Meta-Search and Combining Multiple Rankings

In the last section, we described how an individual search engine works. We now discuss how several search engines can be used together to produce a **meta-search engine**, which is a search system that does not have its own database of Web pages. Instead, it answers the user query by combining the results of some other search engines which normally have their databases of Web pages. Figure 6.12 shows a meta-search architecture.

After receiving a query from the user through the **search interface**, the meta-search engine submits the query to the underlying search engines (called its **component search engines**). The returned results from all these search engines are then combined (**fused** or **merged**) and sent to the user.

A meta-search engine has some intuitive appeals. First of all, it increases the search coverage of the Web. The Web is a huge information source, and each individual search engine may only cover a small portion of it. If we use only one search engine, we will never see those relevant pages that are not covered by the search engine.

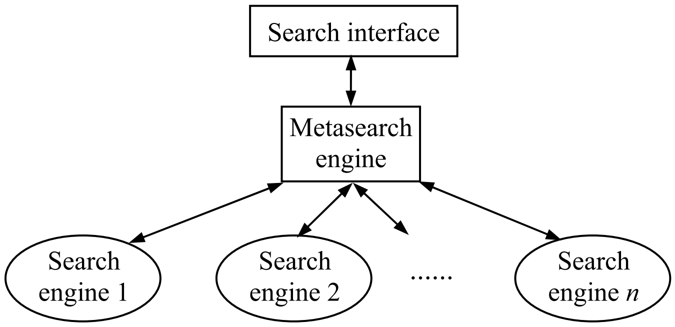


Fig. 6.12. A meta-search architecture

Meta-search may also improve the search effectiveness. Each component search engine has its ranking algorithm to rank relevant pages, which is often biased, i.e., it works well for certain types of pages or queries but not for others. By combining the results from multiple search engines, their biases can be reduced and thus the search precision can be improved.

The key operation in meta-search is to combine the ranked results from the component search engines to produce a single ranking. The first task is to identify whether two pages from different search engines are the same, which facilitates combination and **duplicate removal**. Without downloading the full pages (which is too time consuming), this process is not simple due to aliases, symbolic links, redirections, etc. Typically, several heuristics are used for the purpose, e.g., comparing domain names of URLs, titles of the pages, etc.

The second task is to combine the ranked results from individual search engines to produce a single ranking, i.e., to fuse individual rankings. There are two main classes of meta-search combination (or fusion) algorithms: ones that use similarity scores returned by each component system and ones that do not. Some search engines return a similarity score (with the query) for each returned page, which can be used to produce a better combined ranking. We discuss these two classes of algorithms below.

It is worth noting that the first class of algorithms can also be used to combine scores from different similarity functions in a single IR system or in a single search engine. Indeed, the algorithms below were originally proposed for this purpose. It is likely that search engines already use some such techniques (or their variations) within their ranking mechanisms because a ranking algorithm needs to consider multiple factors.

6.9.1 Combination Using Similarity Scores

Let the set of candidate documents to be ranked be $D = \{d_1, d_2, \ldots, d_N\}$. There are k underlying systems (component search engines or ranking techniques). The ranking from system or technique i gives document d_j the similarity score, s_{ij}. Some popular and simple combination methods are defined by Fox and Shaw in [184].

CombMIN: The combined similarity score for each document d_j is the minimum of the similarities from all underlying search engine systems:

$$\text{CombMIN}(d_j) = \min(s_{1j}, s_{2j}, \ldots, s_{kj}). \tag{33}$$

CombMAX: The combined similarity score for each document d_j is the maximum of the similarities from all underlying search engine systems:

$$\text{CombMAX}(d_j) = \max(s_{1j}, s_{2j}, \ldots, s_{kj}). \tag{34}$$

CombSUM: The combined similarity score for each document d_j is the sum of the similarities from all underlying search engine systems.

$$\text{CombSUM}(d_j) = \sum_{i=1}^{k} s_{ij}. \quad (35)$$

CombANZ: It is defined as

$$\text{CombANZ}(d_j) = \frac{\text{CombSUM}(d_j)}{r_j}, \quad (36)$$

where r_j is the number of non-zero similarities, or the number of systems that retrieved d_j.

CombMNZ: It is defined as

$$\text{CombMNZ}(d_j) = \text{CombSUM}(d_j) \times r_j \quad (37)$$

where r_j is the number of non-zero similarities, or the number of systems that retrieved d_j.

It is a common practice to normalize the similarity scores from each ranking using the maximum score before combination. Researchers have shown that, in general, CombSUM and CombMNZ perform better. CombMNZ outperforms CombSUM slightly in most cases.

6.9.2 Combination Using Rank Positions

We now discuss some popular rank combination methods that use only rank positions of each search engine. In fact, there is a field of study called the **social choice theory** [273] that studies voting algorithms as techniques to make group or social decisions (choices). The algorithms discussed below are based on voting in elections.

In 1770 Jean-Charles de Borda proposed "election by order of merit". Each voter announces a (linear) preference order on the candidates. For each voter, the top candidate receives n points (if there are n candidates in the election), the second candidate receives $n-1$ points, and so on. The points from all voters are summed up to give the final points for each candidate. If there are candidates left unranked by a voter, the remaining points are divided evenly among the unranked candidates. The candidate with the most points wins. This method is called the **Borda ranking**.

An alternative method was proposed by Marquis de Condorcet in 1785. The **Condorcet ranking** algorithm is a majoritarian method where the winner of the election is the candidate(s) that beats each of the other candidates in a pair-wise comparison. If a candidate is not ranked by a voter, the candidate loses to all other ranked candidates. All unranked candidates tie with one another.

Yet another simple method, called the **reciprocal ranking**, sums one over the rank of each candidate across all voters. For each voter, the top candidate has the score of 1, the second ranked candidate has the score of 1/2, and the third ranked candidate has the score of 1/3 and so on. If a candidate is not ranked by a voter, it is skipped in the computation for this voter. The candidates are then ranked according to their final total scores. This rank strategy gives much higher weight than Borda ranking to candidates that are near the top of a list.

Example 13: We use an example in the context of meta-search to illustrate the working of these methods. Consider a meta-search system with five underlying search engine systems, which have ranked four candidate documents or pages, *a*, *b*, *c*, and *d* as follows:

system 1:	*a*, *b*, *c*, *d*
system 2:	*b*, *a*, *d*, *c*
system 3:	*c*, *b*, *a*, *d*
system 4:	*c*, *b*, *d*
system 5:	*c*, *b*

Let us denote the score of each candidate *x* by Score(*x*).

Borda Ranking: The score for each page is as follows:

$$\text{Score}(a) = 4 + 3 + 2 + 1 + 1.5 = 11.5$$
$$\text{Score}(b) = 3 + 4 + 3 + 3 + 3 = 16$$
$$\text{Score}(c) = 2 + 1 + 4 + 4 + 4 = 15$$
$$\text{Score}(d) = 1 + 2 + 1 + 2 + 1.5 = 7.5$$

Thus the final ranking is: *b, c, a, d.*

Condorcet Ranking: We first build an $n \times n$ matrix for all pair-wise comparisons, where n is the number of pages. Each non-diagonal entry (i, j) of the matrix shows the number of wins, loses, and ties of page i over page j, respectively. For our example, the matrix is as follows:

	a	*b*	*c*	*d*
a	-	1:4:0	2:3:0	3:1:1
b	4:1:0	-	2:3:0	5:0:0
c	3:2:0	3:2:0	-	4:1:0
d	1:3:1	0:5:0	1:4:0	-

Fig. 6.13. The pair-wise comparison matrix for the four candidate pages

After the matrix is constructed, pair-wise winners are determined, which produces a win, lose and tie table. Each pair in Fig. 6.13 is compared, and the winner receives one point in its “win” column and the loser receives one point in its “lose” column. For a pair-wise tie, both receive one point in the “tie” column. For example, for page *a*, it only beats *d* because *a* is ranked ahead of *d* three times out of 5 ranks (Fig. 6.13). The win, lose and tie table for Fig. 6.13 is given in Fig. 6.14 below.

	win	*lose*	*tie*
a	1	2	0
b	2	1	0
c	3	0	0
d	0	3	0

Fig. 6.14. The win, lose and tie table for the comparison matrix in Fig. 6.13

To rank the pages, we use their win and lose values. If the number of wins that a page *i* has is higher than another page *j*, then *i* wins over *j*. If their win property is equal, we consider their lose scores, and the page which has a lower lose score wins. If both their win and lose scores are the same, then the pages are tied. The final ranks of the tied pages are randomly assigned. Clearly *c* is the Condorcet winner in our example. The final ranking is: *c*, *b*, *a*, *d*.

Using Reciprocal Ranking:

Score(a) = 1 + 1/2 + 1/3 = 1.83
Score(b) = 1/2 + 1 + 1/2 + 1/2 + 1/2 = 3
Score(c) = 1/3 + 1/4 + 1 + 1 + 1 = 3.55
Score(d) = 1/4 + 1/3 + 1/4 + 1/3= 1.17

The final ranking is: *c*, *b*, *a*, *d*. ■

6.10 Web Spamming

Web search has become very important in the information age. Increased exposure of pages on the Web can result in significant financial gains and/or fames for organizations and individuals. The rank positions of Web pages in search are perhaps the single most important indicator of such exposures of pages. If a user searches for information that is relevant to your pages but your pages are ranked low by search engines, then the user may not see the pages because one seldom clicks a large number of returned pages. This is not acceptable for businesses, organizations, and even individuals. Thus, it has become very important to understand search engine ranking algorithms and to present the information in one’s pages in such a way that the pages will be ranked high when terms related to the contents of the pages are searched. Unfortunately, this also results in **spamming**, which refers to human activities that deliberately mislead search engines to rank some pages higher than they deserve.

There is a gray area between spamming and legitimate page optimization. It is difficult to define precisely what are justifiable and unjustifiable actions aimed at boosting the importance and consequently the rank positions of one's pages.

Assume that, given a user query, each page on the Web can be assigned an information value. All the pages are then ranked according to their information values. Spamming refers to actions that do not increase the information value of a page, but dramatically increase its rank position by misleading search algorithms to rank it high. Due to the fact that search engine algorithms do not understand the content of each page, they use syntactic or surface features to assess the information value of the page. Spammers exploit this weakness to boost the ranks of their pages.

Spamming is annoying for users because it makes it harder to find truly useful information and leads to frustrating search experiences. Spamming is also bad for search engines because spam pages consume crawling bandwidth, pollute the Web, and distort search ranking.

There are in fact many companies that are in the business of helping others improve their page ranking. These companies are called **Search Engine Optimization** (SEO) companies, and their businesses are thriving. Some SEO activities are ethical and some, which generate spam, are not.

As we mentioned earlier, search algorithms consider both content based factors and reputation based factors in scoring each page. In this section, we briefly describe some spam methods that exploit these factors. The section is mainly based on [214] by Gyongyi and Garcia-Molina.

6.10.1 Content Spamming

Most search engines use variations of TF-IDF based measures to assess the relevance of a page to a user query. Content-based spamming methods basically tailor the contents of the text fields in HTML pages to make spam pages more relevant to some queries. Since TF-IDF is computed based on terms, **content spamming** is also called **term spamming**. Term spamming can be placed in any text field:

Title: Since search engines usually give higher weights to terms in the title of a page due to the importance of the title to a page, it is thus common to spam the title.

Meta-Tags: The HTML meta-tags in the page header enable the owner to include some meta information of the page, e.g., author, abstract, keywords, content language, etc. However, meta-tags are very heavily spammed. Search engines now give terms within these tags very low weights or completely ignore their contents.

Body: Clearly spam terms can be placed within the page body to boost the page ranking.

Anchor Text: As we discussed in Sect. 6.8, the anchor text of a hyperlink is considered very important by search engines. It is indexed for the page containing it and also for the page that it points to, so anchor text spam affects the ranking of both pages.

URL: Some search engines break down the URL of a page into terms and consider them in ranking. Thus, spammers can include spam terms in the URL. For example, a URL may be http://www.xxx.com/cheap-MP3-player-case-battery.html

There are two main term spam techniques, which simply create synthetic contents containing spam terms.

1. **Repeating some important terms:** This method increases the TF scores of the repeated terms in a document and thus increases the relevance of the document to these terms. Since plain repetition can be easily detected by search engines, the spam terms can be weaven into some sentences, which may be copied from some other sources. That is, the spam terms are randomly placed in these sentences. For example, if a spammer wants to repeat the word "mining", it may add it randomly in an unrelated (or related) sentence, e.g., "the picture mining quality of this camera mining is amazing," instead of repeating it many times consecutively (next to each other), which is easy to detect.
2. **Dumping of many unrelated terms:** This method is used to make the page relevant to a large number of queries. In order to create the spam content quickly, the spammer may simply copy sentences from related pages on the Web and glue them together.

   Advertisers may also take advantage of some frequently searched terms on the Web and put them in the target pages so that when users search for the frequently search terms, the target pages become relevant. For example, to advertise cruise liners or cruise holiday packages, spammers put "Tom Cruise" in their advertising pages as "Tom Cruise" is a popular film actor in USA and is searched very frequently.

6.10.2 Link Spamming

Since hyperlinks play an important role in determining the reputation score of a page, spammers also spam on hyperlinks.

Out-Link Spamming: It is quite easy to add out-links in one's pages pointing to some **authoritative pages** to boost the hub cores of one's pages. A page is a **hub page** if it points to many authoritative (or quality) pages. The concepts of authority and hub will be formally studied in the next chapter (Sect. 7.4). To create massive out-links, spammers may use a technique called **directory cloning**. There are many directories, e.g., Yahoo!, DMOZ Open Directory, on the Web which contain a large number of links to other Web pages that are organized according to some pre-specified topic hierarchies. Spammers simply replicate a large portion of a directory in the spam page to create a massive out-link structure quickly.

In-Link Spamming: In-link spamming is harder to achieve because it is not easy to add hyperlinks on the Web pages of others. Spammers typically use one or more of the following techniques.

1. *Creating a honey pot*: If a page wants to have a high reputation/quality score, it needs quality pages pointing to it (see Sect. 7.3 in the next chapter). This method basically tries to create some important pages that contain links to target spam pages. For example, the spammer can create a set of pages that contains some very useful information, e.g., glossary of Web mining terms, or Java FAQ and help pages. The honey pots attract people pointing to them because they contain useful information, and consequently have high reputation scores (high quality pages). Such honey pots contain (hidden) links to target spam pages that the spammers want to promote. This strategy can significantly boost the spam pages.
2. *Adding links to Web directories*: Many Web directories allow the user to submit URLs. Spammers can submit the URLs of spam pages at multiple directory sites. Since directory pages often have high quality (or authority) and hub scores, they can boost reputation scores of spam pages significantly.
3. *Posting links to the user-generated content* (reviews, forum discussions, blogs, etc): There are numerous sites on the Web that allow the user to freely post messages, which are called the **user-generated content**. Spammers can add links pointing to their pages to the seemly innocent messages that they post.
4. *Participating in link exchange*: In this case, many spammers form a group and set up a link exchange scheme so that their sites point to each other in order to promote the pages of all the sites.
5. *Creating own spam farm*: In this case, the spammer needs to control a large number of sites. Then, any link structure can be created to boost the ranking of target spam pages.

6.10.3 Hiding Techniques

In most situations, spammer wants to conceal or to hide the spamming sentences, terms and links so that the Web users do not see them. They can use a number of techniques.

Content Hiding: Spam items are made invisible. One simple method is to make the spam terms the same color as the background color. For example, one may use the following for hiding,

```
<body background = white>
   <font color = white> spam items</font>
   …
</body>
```

To hide a hyperlink, one can also use a very small image and a blank image. For example, one may use

```
<a href = target.html"><img src="blank.gif"> </a>
```

A spammer can also use scripts to hide some of the visual elements on the page, for instance, by setting the visible HTML style attribute to false.

Cloaking: Spam Web servers return a HTML document to the user and a different document to a Web crawler. In this way, the spammer can present the Web user with the intended content and send a spam page to the search engine for indexing.

Spam Web servers can identify Web crawlers in one of the two ways:

1. It maintains a list of IP addresses of search engines and identifies search engine crawlers by matching IP addresses.
2. It identifies Web browsers based on the **user–agent field** in the HTTP request message. For instance, the user–agent name of the following HTTP request message is the one used by the Microsoft Internet Explorer 6 browser:

```
GET /pub/WWW/TheProject.html HTTP/1.1
Host: www.w3.org
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)
```

User–agent names are not standard, so it is up to the requesting application what to include in the corresponding message field. However, search engine crawlers usually identify themselves by names distinct from normal Web browsers in order to allow well-intended, and legitimate optimization. For example, some sites serve search engines a version of their pages that is free of navigation links and advertisements.

Redirection: Spammers can also hide a spammed page by automatically redirecting the browser to another URL as soon as the page is loaded. Thus, the spammed page is given to the search engine for indexing (which the user will never see), and the target page is presented to the Web user through redirection. One way to achieve redirection is to use the "refresh" meta-tag, and set the refresh time to zero. Another way is to use scripts.

6.10.4 Combating Spam

Some spamming activities, like redirection using refresh meta-tag, are easy to detect. However, redirections by using scripts are hard to identify because search engine crawlers do not execute scripts. To prevent cloaking, a search engine crawler may identify itself as a regular Web browser.

Using the terms of anchor texts of links that point to a page to index the page is able to fight content spam to some extent because anchor texts from other pages are more trustworthy. This method was originally proposed to index pages that were not fetched by search engine crawlers [364]. It is now a general technique used by search engines as we have seen in Sect. 6.8, i.e., search engines give terms in such anchor texts higher weights. In fact, the terms near a piece of anchor text also offer good editorial judgment about the target page.

The PageRank algorithm [68] is able to combat content spam to a certain degree as it is based on links that point to the target pages, and the pages that point to the target pages need to be reputable or with high PageRank scores as well (see Chap. 7). However, it does not deal with the in-link based spamming methods discussed above.

Instead of combating each individual type of spam, a method (called TrustRank) is proposed in [216] to combat all kinds of spamming methods at the same time. It takes advantage of the approximate isolation of reputable and non-spam pages, i.e., reputable Web pages seldom pointing to spam pages, and spam pages often link to many reputable pages (in an attempt to improve their hub scores). Link analysis methods are used to separate reputable pages and any form of spam without dealing with each spam technique individually.

Combating spam can also be seen as a classification problem, i.e., predicting whether a page is a spam page or not. One can use any supervised learning algorithm to train a spam classifier. The key issue is to design features used in learning. The following are some example features used in [417] to detect content spam.

1. Number of words in the page: A spam page tends to contain more words than a non-spam page so as to cover a large number of popular words.

2. Average word length: The mean word length for English prose is about 5 letters. Average word length of synthetic content is often different.
3. Number of words in the page title: Since search engines usually give extra weights to terms appearing in page titles, spammers often put many keywords in the titles of the spam pages.
4. Fraction of visible content: Spam pages often hide spam terms by making them invisible to the user.

Other features used include the amount of anchor text, compressibility, fraction of page drawn from globally popular words, independent *n*-gram likelihoods, conditional *n*-gram likelihoods, etc. Details can be found in [417]. Its spam detection classifier gave very good results. Testing on 2364 spam pages and 14806 non-spam pages (17170 pages in total), the classifier was able to correctly identify 2,037 (86.2%) of the 2364 spam pages, while misidentifying only 526 spam and non-spam pages.

Another interesting technique for fighting spam is to partition each Web page into different blocks using techniques discussed in Sect. 6.5. Each block is given an importance level automatically. To combat link spam, links in less important blocks are given lower transition probabilities to be used in the PageRank computation. The original PageRank algorithm assigns every link in a page an equal transition probability (see Sect. 7.3). The non-uniform probability assignment results in lower PageRank scores for pages pointed to by links in less important blocks. This method is effective because in the link exchange scheme and the honey pot scheme, the spam links are usually placed in unimportant blocks of the page, e.g., at the bottom of the page. The technique may also be used to fight term spam in a similar way, i.e., giving terms in less important blocks much lower weights in rank score computation. This method is proposed in [78].

However, sophisticated spam is still hard to detect. Combating spam is an on-going process. Once search engines are able to detect certain types of spam, spammers invent more sophisticated spamming methods.

Bibliographic Notes

Information retrieval (IR) is a major research field. This chapter only gives a brief introduction to some commonly used models and techniques. There are several text books that have a comprehensive coverage of the field, e.g., those by Baeza-Yates and Ribeiro-Neto [31], Grossman and Frieder [209], Salton and McGill [471], van Rijsbergen (an online book at http://www.dcs.gla.ac.uk/Keith/Preface.html), Witten et al. [551], and Yu and Meng [581].

A similar chapter in the book by Chakrabarti [85] also discusses many Web specific issues and has influenced the writing of this chapter. Below, we discuss some further readings related to Web search and mining.

On index compression, Elias coding was introduced by Elias [161] and Golomb coding was introduced by Golomb [202]. Their applications to index compression was studied by several researchers, e.g., Witten et al. [551], Bell et al. [45], Moffat et al. [392], and Williams and Zobel [547]. Wikipedia is a great source of information on this topic as well.

Latent semantic index (LSI) was introduced by Deerwester et al. [125], which uses the singular value decomposition technique (SVD) [203]. Additional information about LSI and/or SVD can be found in [48, 581, 288]. Telcordia Technologies, where LSI was developed, maintains a LSI page at http://lsi.research.telcordia.com/ with more references.

On Web page pre-processing, the focus has been on identifying the main content blocks of each page because a typical Web page contains a large amount of noise, which can adversely affect the search or mining accuracy. Several researchers have attempted the task, e.g., Bar-Yossef et al. [38], Debnath et al. [124], Gibson, et al. [199], Li et al. [324], Lin and Ho [336], Ma et al. [355], Ramaswamy et al. [456], Song et al. [495], Yi et al. [576], Yin and Lee [579], etc.

Although search is probably the biggest application on the Web, little is known about the actual implementation of a search engine except some principal ideas. Sect. 6.8 is largely based on the Google paper by Brin and Page [68], and bits and pieces in various other sources. Over the years, a large number of researchers have studied Web search. More recent studies on various aspects of search can be found in [37, 79, 89, 262, 289, 297, 451, 460, 508, 567, 569, 611].

For metasearch, the combination methods in Sect. 6.9.1 were proposed by Fox and Shaw [184]. Aslam and Montague [28], Montague and Aslam [394], and Nuray and Can [418] provide good descriptions of Borda ranking and Condorcet ranking. In addition to ranking, Meng et al. [378] discussed many other metasearch issues.

On Web spam, Gyongyi and Garcia-Molina gave an excellent taxonomy of different types of spam [214]. The TrustRank algorithm is also due to them [216]. An improvement to TrustRank was proposed by Wu et al. [557]. General link spam detection was studied by Adali et al. [1], Amitay et al. [19], Baeza-Yates et al. [30], Gyongyi and Garcia-Molina [215], Wu and Davison [555], Zhang et al. [604], etc. Content spam detection was studied by Fetterly et al. [176, 177], and Ntoulas et al. [417]. A cloaking detection algorithm is reported in [556].