UNIT-I

Introduction to Web Data Mining and Data Mining Foundations

1 Introduction

Already know what the **World Wide Web** is and have used it extensively. The World Wide Web (or the **Web** for short) has impacted on almost every aspect of our lives. It is the biggest and most widely known information source that is easily accessible and searchable. It consists of billions of interconnected documents (called **Web pages**) which are authored by millions of people. Since its inception, the Web has dramatically changed our information seeking behavior. Before the Web, finding information means asking a friend or an expert, or buying/borrowing a book to read. However, with the Web, everything is only a few clicks away from the comfort of our homes or offices. Not only can we find needed information on the Web, but we can also easily share our information and knowledge with others.

The Web has also become an important channel for conducting businesses. We can buy almost anything from online stores without needing to go to a physical shop. The Web also provides convenient means for us to communicate with each other, to express our views and opinions on anything, and to discuss with people from anywhere in the world. The Web is truly a **virtual society**. In this first chapter, we introduce the Web, its history, and the topics that we will study in the book.

1.1 What is the World Wide Web?

The World Wide Web is officially defined as a "wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents." In simpler terms, the Web is an Internet-based computer network that allows users of one computer to access information stored on another through the world-wide network called the **Internet**.

The Web's implementation follows a standard **client-server** model. In this model, a user relies on a program (called the **client**) to connect to a remote machine (called the **server**) where the data is stored. Navigating through the Web is done by means of a client program called the **browser**, e.g., Netscape, Internet Explorer, Firefox, etc. Web browsers work by sending requests to remote servers for information and then interpreting

the returned documents written in HTML and laying out the text and graphics on the user's computer screen on the client side.

The operation of the Web relies on the structure of its **hypertext** documents. Hypertext allows Web page authors to link their documents to other related documents residing on computers anywhere in the world. To view these documents, one simply follows the links (called **hyperlinks**).

The idea of hypertext was invented by Ted Nelson in 1965 [403], who also created the well known hypertext system Xanadu (http://xanadu.com/). Hypertext that also allows other media (e.g., image, audio and video files) is called **hypermedia**.

1.2 A Brief History of the Web and the Internet

Creation of the Web: The Web was invented in 1989 by **Tim Berners-Lee**, who, at that time, worked at CERN (Centre European pour la Recherche Nucleaire, or European Laboratory for Particle Physics) in Switzerland. He coined the term "World Wide Web," wrote the first World Wide Web server, httpd, and the first client program (a browser and editor), "**WorldWideWeb**".

It began in March 1989 when Tim Berners-Lee submitted a proposal titled "Information Management: A Proposal" to his superiors at CERN. In the proposal, he discussed the disadvantages of hierarchical information organization and outlined the advantages of a hypertext-based system. The proposal called for a simple protocol that could request information stored in remote systems through networks, and for a scheme by which information could be exchanged in a common format and documents of individuals could be linked by hyperlinks to other documents. It also proposed methods for reading text and graphics using the display technology at CERN at that time. The proposal essentially outlined a **distributed hypertext system**, which is the basic architecture of the Web.

Initially, the proposal did not receive the needed support. However, in 1990, Berners-Lee re-circulated the proposal and received the support to begin the work. With this project, Berners-Lee and his team at CERN laid the foundation for the future development of the Web as a distributed hypertext system. They introduced their server and browser, the protocol used for communication between clients and the server, the HyperText Transfer Protocol (**HTTP**), the HyperText Markup Language (**HTML**) used for authoring Web documents, and the Universal Resource Locator (**URL**). And so it began.

Mosaic and Netscape Browsers: The next significant event in the development of the Web was the arrival of **Mosaic**. In February of 1993, Marc Andreesen from the University of Illinois' NCSA (National Center for Supercomputing Applications) and his team released the first "Mosaic for X" graphical Web browser for UNIX. A few months later, different versions of Mosaic were released for Macintosh and Windows operating systems. This was an important event. For the first time, a Web client, with a consistent and simple point-and-click graphical user interface, was implemented for the three most popular operating systems available at the time. It soon made big splashes outside the academic circle where it had begun. In mid-1994, Silicon Graphics founder Jim Clark collaborated with Marc Andreessen, and they founded the company **Mosaic Communications** (later renamed as **Netscape Communications**). Within a few months, the **Netscape** browser was released to the public, which started the explosive growth of the Web. The **Internet Explorer** from Microsoft entered the market in August, 1995 and began to challenge Netscape.

The creation of the World Wide Web by Tim Berners-Lee followed by the release of the Mosaic browser are often regarded as the two most significant contributing factors to the success and popularity of the Web.

Internet: The Web would not be possible without the Internet, which provides the communication network for the Web to function. The **Internet** started with the computer network **ARPANET** in the Cold War era. It was produced as the result of a project in the United States aiming at maintaining control over its missiles and bombers after a nuclear attack. It was supported by Advanced Research Projects Agency (ARPA), which was part of the Department of Defense in the United States. The first ARPANET connections were made in 1969, and in 1972, it was demonstrated at the First International Conference on Computers and Communication, held in Washington D.C. At the conference, ARPA scientists linked computers together from 40 different locations.

In 1973, Vinton Cerf and Bob Kahn started to develop the protocol later to be called **TCP/IP** (**Transmission Control Protocol/Internet Protocol**). In the next year, they published the paper "Transmission Control Protocol", which marked the beginning of TCP/IP. This new protocol allowed diverse computer networks to interconnect and communicate with each other. In subsequent years, many networks were built, and many competing techniques and protocols were proposed and developed. However, ARPANET was still the backbone to the entire system. During the period, the network scene was chaotic. In 1982, the TCP/IP was finally adopted, and the **Internet**, which is a connected set of networks using the TCP/IP protocol, was born.

Search Engines: With information being shared worldwide, there was a need for individuals to find information in an orderly and efficient manner. Thus began the development of search engines. The search system **Excite** was introduced in 1993 by six Stanford University students. **EINet Galaxy** was established in 1994 as part of the MCC Research Consortium at the University of Texas. Jerry Yang and David Filo created **Yahoo!** in 1994, which started out as a listing of their favorite Web sites, and offered directory search. In subsequent years, many search systems emerged, e.g., **Lycos**, **Inforseek**, **AltaVista**, **Inktomi**, **Ask Jeeves**, **Northernlight**, etc.

Google was launched in 1998 by Sergey Brin and Larry Page based on their research project at Stanford University. Microsoft started to commit to search in 2003, and launched the **MSN** search engine in spring 2005. It used search engines from others before. **Yahoo!** provided a general search capability in 2004 after it purchased Inktomi in 2003.

W3C (**The World Wide Web Consortium**): W3C was formed in the December of 1994 by MIT and CERN as an international organization to lead the development of the Web. W3C's main objective was "to promote standards for the evolution of the Web and interoperability between WWW products by producing specifications and reference software." The first **International Conference on World Wide Web** (**WWW**) was also held in 1994, which has been a yearly event ever since.

From 1995 to 2001, the growth of the Web boomed. Investors saw commercial opportunities and became involved. Numerous businesses started on the Web, which led to irrational developments. Finally, the bubble burst in 2001. However, the development of the Web was not stopped, but has only become more rational since.

1.3 Web Data Mining

The rapid growth of the Web in the last decade makes it the largest publicly accessible data source in the world. The Web has many unique characteristics, which make mining useful information and knowledge a fascinating and challenging task. Let us review some of these characteristics.

1. The amount of data/information on the Web is huge and still growing. The coverage of the information is also very wide and diverse. One can find information on almost anything on the Web.
2. Data of all types exist on the Web, e.g., structured tables, semi-structured Web pages, unstructured texts, and multimedia files (images, audios, and videos).

3. Information on the Web is **heterogeneous**. Due to the diverse authorship of Web pages, multiple pages may present the same or similar information using completely different words and/or formats. This makes integration of information from multiple pages a challenging problem.
4. A significant amount of information on the Web is linked. Hyperlinks exist among Web pages within a site and across different sites. Within a site, hyperlinks serve as information organization mechanisms. Across different sites, hyperlinks represent implicit conveyance of authority to the target pages. That is, those pages that are linked (or pointed) to by many other pages are usually high quality pages or **authoritative pages** simply because many people trust them.
5. The information on the Web is noisy. The **noise** comes from two main sources. First, a typical Web page contains many pieces of information, e.g., the **main content** of the page, navigation links, advertisements, copyright notices, privacy policies, etc. For a particular application, only part of the information is useful. The rest is considered noise. To perform fine-grain Web information analysis and data mining, the noise should be removed. Second, due to the fact that the Web does not have quality control of information, i.e., one can write almost anything that one likes, a large amount of information on the Web is of low quality, erroneous, or even misleading.
6. The Web is also about services. Most commercial Web sites allow people to perform useful operations at their sites, e.g., to purchase products, to pay bills, and to fill in forms.
7. The Web is dynamic. Information on the Web changes constantly. Keeping up with the change and monitoring the change are important issues for many applications.
8. The Web is a virtual society. The Web is not only about data, information and services, but also about interactions among people, organizations and automated systems. One can communicate with people anywhere in the world easily and instantly, and also express one's views on anything in Internet forums, blogs and review sites.

All these characteristics present both challenges and opportunities for mining and discovery of information and knowledge from the Web. In this book, we only focus on mining textual data. For mining of images, videos and audios, please refer to [143, 441].

To explore information mining on the Web, it is necessary to know data mining, which has been applied in many Web mining tasks. However, Web mining is not entirely an application of data mining. Due to the richness and diversity of information and other Web specific characteristics discussed above, Web mining has developed many of its own algorithms.

1.3.1 What is Data Mining?

Data mining is also called **knowledge discovery in databases** (**KDD**). It is commonly defined as the process of discovering useful **patterns** or knowledge from data sources, e.g., databases, texts, images, the Web, etc. The patterns must be valid, potentially useful, and understandable. Data mining is a multi-disciplinary field involving machine learning, statistics, databases, artificial intelligence, information retrieval, and visualization.

There are many data mining tasks. Some of the common ones are **supervised learning** (or **classification**), **unsupervised learning** (or **clustering**), **association rule mining**, and **sequential pattern mining**. We will study all of them in this book.

A data mining application usually starts with an understanding of the application domain by **data analysts** (**data miners**), who then identify suitable data sources and the target data. With the data, data mining can be performed, which is usually carried out in three main steps:

- **Pre-processing**: The raw data is usually not suitable for mining due to various reasons. It may need to be cleaned in order to remove noises or abnormalities. The data may also be too large and/or involve many irrelevant attributes, which call for data reduction through sampling and attribute selection. Details about data pre-processing can be found in any standard data mining textbook.
- **Data mining**: The processed data is then fed to a data mining algorithm which will produce patterns or knowledge.
- **Post-processing**: In many applications, not all discovered patterns are useful. This step identifies those useful ones for applications. Various evaluation and visualization techniques are used to make the decision.

The whole process (also called the **data mining process**) is almost always iterative. It usually takes many rounds to achieve final satisfactory results, which are then incorporated into real-world operational tasks.

Traditional data mining uses structured data stored in relational tables, spread sheets, or flat files in the tabular form. With the growth of the Web and text documents, **Web mining** and **text mining** are becoming increasingly important and popular. Web mining is the focus of this book.

1.3.2 What is Web Mining?

Web mining aims to discover useful information or knowledge from the **Web hyperlink structure**, **page content**, and **usage data**. Although Web mining uses many data mining techniques, as mentioned above it is not

purely an application of traditional data mining due to the heterogeneity and semi-structured or unstructured nature of the Web data. Many new mining tasks and algorithms were invented in the past decade. Based on the primary kinds of data used in the mining process, Web mining tasks can be categorized into three types: Web structure mining, Web content mining and Web usage mining.

- **Web structure mining**: Web structure mining discovers useful knowledge from hyperlinks (or links for short), which represent the structure of the Web. For example, from the links, we can discover important Web pages, which, incidentally, is a key technology used in search engines. We can also discover communities of users who share common interests. Traditional data mining does not perform such tasks because there is usually no link structure in a relational table.
- **Web content mining**: Web content mining extracts or mines useful information or knowledge from Web page contents. For example, we can automatically classify and cluster Web pages according to their topics. These tasks are similar to those in traditional data mining. However, we can also discover patterns in Web pages to extract useful data such as descriptions of products, postings of forums, etc, for many purposes. Furthermore, we can mine customer reviews and forum postings to discover consumer sentiments. These are not traditional data mining tasks.
- **Web usage mining**: Web usage mining refers to the discovery of user access patterns from Web usage logs, which record every click made by each user. Web usage mining applies many data mining algorithms. One of the key issues in Web usage mining is the pre-processing of clickstream data in usage logs in order to produce the right data for mining.

In this book, we will study all these three types of mining. However, due to the richness and diversity of information on the Web, there are a large number of Web mining tasks. We will not be able to cover them all. We will only focus on some important tasks and their algorithms.

The **Web mining process** is similar to the data mining process. The difference is usually in the data collection. In traditional data mining, the data is often already collected and stored in a data warehouse. For Web mining, data collection can be a substantial task, especially for Web structure and content mining, which involves crawling a large number of target Web pages. We will devote a whole chapter on crawling.

Once the data is collected, we go through the same three-step process: data pre-processing, Web data mining and post-processing. However, the techniques used for each step can be quite different from those used in traditional data mining.

2 Association Rules and Sequential Patterns

Association rules are an important class of regularities in data. Mining of association rules is a fundamental data mining task. It is perhaps the most important model invented and extensively studied by the database and data mining community. Its objective is to find all **co-occurrence** relationships, called **associations**, among data items. Since it was first introduced in 1993 by Agrawal et al. [9], it has attracted a great deal of attention. Many efficient algorithms, extensions and applications have been reported.

The classic application of association rule mining is the **market basket** data analysis, which aims to discover how items purchased by customers in a supermarket (or a store) are associated. An example association rule is

$$\text{Cheese} \rightarrow \text{Beer} \qquad [\text{support} = 10\%, \text{confidence} = 80\%].$$

The rule says that 10% customers buy Cheese and Beer together, and those who buy Cheese also buy Beer 80% of the time. Support and confidence are two measures of rule strength, which we will define later.

This mining model is in fact very general and can be used in many applications. For example, in the context of the Web and text documents, it can be used to find word co-occurrence relationships and Web usage patterns as we will see in later chapters.

Association rule mining, however, does not consider the sequence in which the items are purchased. Sequential pattern mining takes care of that. An example of a sequential pattern is "5% of customers buy bed first, then mattress and then pillows". The items are not purchased at the same time, but one after another. Such patterns are useful in Web usage mining for analyzing **clickstreams** in server logs. They are also useful for finding **language** or **linguistic patterns** from natural language texts.

2.1 Basic Concepts of Association Rules

The problem of mining association rules can be stated as follows: Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of **items**. Let $T = (t_1, t_2, \ldots, t_n)$ be a set of **transactions** (the database), where each transaction t_i is a set of items such that $t_i \subseteq I$. An **association rule** is an implication of the form,

$X \rightarrow Y$, where $X \subset I$, $Y \subset I$, and $X \cap Y = \emptyset$.

X (or Y) is a set of items, called an **itemset**.

Example 1: We want to analyze how the items sold in a supermarket are related to one another. I is the set of all items sold in the supermarket. A transaction is simply a set of items purchased in a basket by a customer. For example, a transaction may be:

{Beef, Chicken, Cheese},

which means that a customer purchased three items in a basket, Beef, Chicken, and Cheese. An association rule may be:

Beef, Chicken $\rightarrow$ Cheese,

where {Beef, Chicken} is X and {Cheese} is Y. For simplicity, brackets "{" and "}" are usually omitted in transactions and rules. ■

A transaction $t_i \in T$ is said to **contain** an itemset X if X is a subset of t_i (we also say that the itemset X **covers** t_i). The **support count** of X in T (denoted by $X.count$) is the number of transactions in T that contain X. The strength of a rule is measured by its **support** and **confidence**.

Support: The support of a rule, $X \rightarrow Y$, is the percentage of transactions in T that contains $X \cup Y$, and can be seen as an estimate of the probability, $\Pr(X \cup Y)$. The rule support thus determines how frequent the rule is applicable in the transaction set T. Let n be the number of transactions in T. The support of the rule $X \rightarrow Y$ is computed as follows:

$$support = \frac{(X \cup Y).count}{n}. \tag{1}$$

Support is a useful measure because if it is too low, the rule may just occur due to chance. Furthermore, in a business environment, a rule **covering** too few cases (or transactions) may not be useful because it does not make business sense to act on such a rule (not profitable).

Confidence: The confidence of a rule, $X \rightarrow Y$, is the percentage of transactions in T that contain X also contain Y. It can be seen as an estimate of the conditional probability, $\Pr(Y \mid X)$. It is computed as follows:

$$confidence = \frac{(X \cup Y).count}{X.count}. \tag{2}$$

Confidence thus determines the **predictability** of the rule. If the confidence of a rule is too low, one cannot reliably infer or predict Y from X. A rule with low predictability is of limited use.

Objective: Given a transaction set *T*, the problem of mining association rules is to discover all association rules in *T* that have support and confidence greater than or equal to the user-specified **minimum support** (denoted by **minsup**) and **minimum confidence** (denoted by **minconf**).

The keyword here is "all", i.e., association rule mining is complete. Previous methods for rule mining typically generate only a subset of rules based on various heuristics (see Chap. 3).

Example 2: Figure 2.1 shows a set of seven transactions. Each transaction t_i is a set of items purchased in a basket in a store by a customer. The set *I* is the set of all items sold in the store.

t_1: Beef, Chicken, Milk
t_2: Beef, Cheese
t_3: Cheese, Boots
t_4: Beef, Chicken, Cheese
t_5: Beef, Chicken, Clothes, Cheese, Milk
t_6: Chicken, Clothes, Milk
t_7: Chicken, Milk, Clothes

Fig. 2.1. An example of a transaction set

Given the user-specified minsup = 30% and minconf = 80%, the following association rule (**sup** is the support, and **conf** is the confidence)

Chicken, Clothes $\rightarrow$ Milk [sup = 3/7, conf = 3/3]

is valid as its support is 42.84% (> 30%) and its confidence is 100% (> 80%). The rule below is also valid, whose consequent has two items:

Clothes $\rightarrow$ Milk, Chicken [sup = 3/7, conf = 3/3].

Clearly, more association rules can be discovered, as we will see later. ■

We note that the data representation in the transaction form of Fig. 2.1 is a simplistic view of shopping baskets. For example, the quantity and price of each item are not considered in the model.

We also note that a text document or even a sentence in a single document can be treated as a transaction without considering word sequence and the number of occurrences of each word. Hence, given a set of documents or a set of sentences, we can find word co-occurrence relations.

A large number of association rule mining algorithms have been reported in the literature, which have different mining efficiencies. Their resulting sets of rules are, however, all the same based on the definition of association rules. That is, given a transaction data set *T*, a minimum support and a minimum confidence, the set of association rules existing in *T* is

uniquely determined. Any algorithm should find the same set of rules although their computational efficiencies and memory requirements may be different. The best known mining algorithm is the **Apriori** algorithm proposed in [11], which we study next.

2.2 Apriori Algorithm

The Apriori algorithm works in two steps:

1. **Generate all frequent itemsets:** A frequent itemset is an itemset that has transaction support above minsup.
2. **Generate all confident association rules from the frequent itemsets:** A confident association rule is a rule with confidence above minconf.

We call the number of items in an itemset its **size**, and an itemset of size k a k-itemset. Following Example 2 above, {Chicken, Clothes, Milk} is a frequent 3-itemset as its support is 3/7 (minsup = 30%). From the itemset, we can generate the following three association rules (minconf = 80%):

Rule 1:	Chicken, Clothes $\rightarrow$ Milk	[sup = 3/7, conf = 3/3]
Rule 2:	Clothes, Milk $\rightarrow$ Chicken	[sup = 3/7, conf = 3/3]
Rule 3:	Clothes $\rightarrow$ Milk, Chicken	[sup = 3/7, conf = 3/3].

Below, we discuss the two steps in turn.

2.2.1 Frequent Itemset Generation

The Apriori algorithm relies on the ***apriori*** or **downward closure** property to efficiently generate all frequent itemsets.

Downward Closure Property: If an itemset has minimum support, then every non-empty subset of this itemset also has minimum support.

The idea is simple because if a transaction contains a set of items X, then it must contain any non-empty subset of X. This property and the minsup threshold prune a large number of itemsets that cannot be frequent.

To ensure efficient itemset generation, the algorithm assumes that the items in I are sorted in **lexicographic order** (a total order). The order is used throughout the algorithm in each itemset. We use the notation $\{w[1], w[2], \ldots, w[k]\}$ to represent a k-itemset w consisting of items $w[1]$, $w[2]$, …, $w[k]$, where $w[1] < w[2] < \ldots < w[k]$ according to the total order.

The Apriori algorithm for frequent itemset generation, which is given in Fig. 2.2, is based on **level-wise search**. It generates all frequent itemsets by

```
Algorithm Apriori(T)
1   C_1 ← init-pass(T);                          // the first pass over T
2   F_1 ← {f | f ∈ C_1, f.count/n ≥ minsup};     // n is the no. of transactions in T
3   for (k = 2; F_{k-1} ≠ ∅; k++) do             // subsequent passes over T
4     C_k ← candidate-gen(F_{k-1});
5     for each transaction t ∈ T do              // scan the data once
6       for each candidate c ∈ C_k do
7         if c is contained in t then
8           c.count++;
9       endfor
10    endfor
11    F_k ← {c ∈ C_k | c.count/n ≥ minsup}
12  endfor
13  return F ← ⋃_k F_k;
```

Fig. 2.2. The Apriori algorithm for generating frequent itemsets

```
Function candidate-gen(F_{k-1})
1   C_k ← ∅;                                     // initialize the set of candidates
2   forall f_1, f_2 ∈ F_{k-1}                    // find all pairs of frequent itemsets
3     with f_1 = {i_1, … , i_{k-2}, i_{k-1}}     // that differ only in the last item
4     and  f_2 = {i_1, … , i_{k-2}, i'_{k-1}}
5     and i_{k-1} < i'_{k-1} do                  // according to the lexicographic order
6       c ← {i_1, …, i_{k-1}, i'_{k-1}};         // join the two itemsets f_1 and f_2
7       C_k ← C_k ∪ {c};                         // add the new itemset c to the candidates
8       for each (k–1)-subset s of c do
9         if (s ∉ F_{k-1}) then
10          delete c from C_k;                   // delete c from the candidates
11      endfor
12  endfor
13  return C_k;                                  // return the generated candidates
```

Fig. 2.3. The candidate-gen function

making multiple passes over the data. In the first pass, it counts the supports of individual items (line 1) and determines whether each of them is frequent (line 2). F_1 is the set of frequent 1-itemsets. In each subsequent pass k, there are three steps:

1. It starts with the seed set of itemsets F_{k-1} found to be frequent in the (k–1)-th pass. It uses this seed set to generate **candidate itemsets** C_k (line 4), which are possible frequent itemsets. This is done using the candidate-gen() function.
2. The transaction database is then scanned and the actual support of each candidate itemset c in C_k is counted (lines 5–10). Note that we do not need to load the whole data into memory before processing. Instead, at

any time, only one transaction resides in memory. This is a very important feature of the algorithm. It makes the algorithm scalable to huge data sets, which cannot be loaded into memory.
3. At the end of the pass or scan, it determines which of the candidate itemsets are actually frequent (line 11).

The final output of the algorithm is the set F of all frequent itemsets (line 13). The candidate-gen() function is discussed below.

Candidate-gen function: The candidate generation function is given in Fig. 2.3. It consists of two steps, the **join step** and the **pruning step**.

Join step (lines 2–6 in Fig. 2.3): This step joins two frequent (k–1)-itemsets to produce a possible candidate c (line 6). The two frequent itemsets f_1 and f_2 have exactly the same items except the last one (lines 3–5). c is added to the set of candidates C_k (line 7).

Pruning step (lines 8–11 in Fig. 2.3): A candidate c from the join step may not be a final candidate. This step determines whether all the k–1 subsets (there are k of them) of c are in F_{k-1}. If anyone of them is not in F_{k-1}, c cannot be frequent according to the downward closure property, and is thus deleted from C_k.

The correctness of the candidate-gen() function is easy to show (see [11]). Here, we use an example to illustrate the working of the function.

Example 3: Let the set of frequent itemsets at level 3 be

$$F_3 = \{\{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\}, \{1, 3, 5\}, \{2, 3, 4\}\}.$$

For simplicity, we use numbers to represent items. The join step (which generates candidates for level 4) will produce two candidate itemsets, {1, 2, 3, 4} and {1, 3, 4, 5}. {1, 2, 3, 4} is generated by joining the first and the second itemsets in F_3 as their first and second items are the same respectively. {1, 3, 4, 5} is generated by joining {1, 3, 4} and {1, 3, 5}.

After the pruning step, we have only:

$$C_4 = \{\{1, 2, 3, 4\}\}$$

because {1, 4, 5} is not in F_3 and thus {1, 3, 4, 5} cannot be frequent.

Example 4: Let us see a complete running example of the Apriori algorithm based on the transactions in Fig. 2.1. We use minsup = 30%.

F_1: {{Beef}:4, {Cheese}:4, {Chicken}:5, {Clothes}:3, {Milk}:4}

Note: the number after each frequent itemset is the support count of the itemset, i.e., the number of transactions containing the itemset. A minimum support count of 3 is sufficient because the support of 3/7 is greater than 30%, where 7 is the total number of transactions.

C_2: {{Beef, Cheese}, {Beef, Chicken}, {Beef, Clothes}, {Beef, Milk}, {Cheese, Chicken}, {Cheese, Clothes}, {Cheese, Milk}, {Chicken, Clothes}, {Chicken, Milk}, {Clothes, Milk}}

F_2: {{Beef, Chicken}:3, {Beef, Cheese}:3, {Chicken, Clothes}:3, {Chicken, Milk}:4, {Clothes, Milk}:3}

C_3: {{Chicken, Clothes, Milk}}

Note: {Beef, Cheese, Chicken} is also produced in line 6 of Fig. 2.3. However, {Cheese, Chicken} is not in F_2, and thus the itemset {Beef, Cheese, Chicken} is not included in C_3.

F_3: {{Chicken, Clothes, Milk}:3}. ■

Finally, some remarks about the Apriori algorithm are in order:

- Theoretically, this is an exponential algorithm. Let the number of items in I be m. The space of all itemsets is $O(2^m)$ because each item may or may not be in an itemset. However, the mining algorithm exploits the sparseness of the data and the high minimum support value to make the mining possible and efficient. The **sparseness** of the data in the context of market basket analysis means that the store sells a lot of items, but each shopper only purchases a few of them.
- The algorithm can scale up to large data sets as it does not load the entire data into the memory. It only scans the data K times, where K is the size of the largest itemset. In practice, K is often small (e.g., < 10). This scale-up property is very important in practice because many real-world data sets are so large that they cannot be loaded into the main memory.
- The algorithm is based on level-wise search. It has the flexibility to stop at any level. This is useful in practice because in many applications, long frequent itemsets or rules are not needed as they are hard to use.
- As mentioned earlier, once a transaction set T, a minsup and a minconf are given, the set of frequent itemsets that can be found in T is uniquely determined. Any algorithm should find the same set of frequent itemsets. This property about association rule mining does not hold for many other data mining tasks, e.g., classification or clustering, for which different algorithms may produce very different results.
- The main problem with association rule mining is that it often produces a huge number of itemsets (and rules), tens of thousands, or more, which makes it hard for the user to analyze them to find those useful ones. This is called the **interestingness** problem. Researchers have proposed several methods to tackle this problem (see Bibliographic Notes).

An efficient implementation of the Apriori algorithm involves sophisticated data structures and programming techniques, which are beyond the

scope of this book. Apart from the Apriori algorithm, there is a large number of other algorithms, e.g., FP-growth [220] and many others.

2.2.2 Association Rule Generation

In many applications, frequent itemsets are already useful and sufficient. Then, we do not need to generate association rules. In applications where rules are desired, we use frequent itemsets to generate all association rules.

Compared with frequent itemset generation, rule generation is relatively simple. To generate rules for every frequent itemset f, we use all non-empty subsets of f. For each such subset α, we output a rule of the form

$$(f - \alpha) \rightarrow \alpha, \text{ if}$$

$$confidence = \frac{f.count}{(f - \alpha).count} \geq minconf, \tag{3}$$

where $f.count$ (or $(f-\alpha).count$) is the support count of f (or $(f - \alpha)$). The support of the rule is $f.count/n$, where n is the number of transactions in the transaction set T. All the support counts needed for confidence computation are available because if f is frequent, then any of its non-empty subsets is also frequent and its support count has been recorded in the mining process. Thus, no data scan is needed in rule generation.

This exhaustive rule generation strategy is, however, inefficient. To design an efficient algorithm, we observe that the support count of f in the above confidence computation does not change as α changes. It follows that for a rule $(f - \alpha) \rightarrow \alpha$ to hold, all rules of the form $(f - \alpha_{sub}) \rightarrow \alpha_{sub}$ must also hold, where α_{sub} is a non-empty subset of α, because the support count of $(f - \alpha_{sub})$ must be less than or equal to the support count of $(f - \alpha)$. For example, given an itemset {A, B, C, D}, if the rule (A, B $\rightarrow$ C, D) holds, then the rules (A, B, C $\rightarrow$ D) and (A, B, D $\rightarrow$ C) must also hold.

Thus, for a given frequent itemset f, if a rule with consequent α holds, then so do rules with consequents that are subsets of α. This is similar to the downward closure property that, if an itemset is frequent, then so are all its subsets. Therefore, from the frequent itemset f, we first generate all rules with one item in the consequent. We then use the consequents of these rules and the function candidate-gen() (Fig. 2.3) to generate all possible consequents with two items that can appear in a rule, and so on. An algorithm using this idea is given in Fig. 2.4. Note that all 1-item consequent rules (rules with one item in the consequent) are first generated in line 2 of the function genRules(). The confidence is computed using (3).

```
Algorithm genRules(F)                    // F is the set of all frequent itemsets
1   for each frequent k-itemset f_k in F, k ≥ 2 do
2       output every 1-item consequent rule of f_k with confidence ≥ minconf and
            support ← f_k.count / n     // n is the total number of transactions in T
3       H_1 ← {consequents of all 1-item consequent rules derived from f_k above};
4       ap-genRules(f_k, H_1);
5   endfor

Procedure ap-genRules(f_k, H_m)          // H_m is the set of m-item consequents
1   if (k > m + 1) AND (H_m ≠ ∅) then
2     H_{m+1} ← candidate-gen(H_m);
3     for each h_{m+1} in H_{m+1} do
4         conf ← f_k.count / (f_k − h_{m+1}).count;
5         if (conf ≥ minconf) then
6             output the rule (f_k − h_{m+1}) → h_{m+1} with confidence = conf and
              support = f_k.count / n;     // n is the total number of transactions in T
7         else
8           delete h_{m+1} from H_{m+1};
9     endfor
10    ap-genRules(f_k, H_{m+1});
11  endif
```

Fig. 2.4. The association rule generation algorithm

Example 5: We again use transactions in Fig. 2.1, minsup = 30% and minconf = 80%. The frequent itemsets are as follows (see Example 4):

F_1: {{Beef}:4, {Cheese}:4, {Chicken}:5, {Clothes}:3, {Milk}:4}
F_2: {{Beef, Cheese}:3, {Beef, Chicken}:3, {Chicken, Clothes}:3, {Chicken, Milk}:4, {Clothes, Milk}:3}
F_3: {{Chicken, Clothes, Milk}:3}.

We use only the itemset in F_3 to generate rules (generating rules from each itemset in F_2 can be done in the same way). The itemset in F_3 generates the following possible 1-item consequent rules:

Rule 1: Chicken, Clothes → Milk [sup = 3/7, conf = 3/3]
Rule 2: Chicken, Milk → Clothes [sup = 3/7, conf = 3/4]
Rule 3: Clothes, Milk → Chicken [sup = 3/7, conf = 3/3].

Due to the minconf requirement, only Rule 1 and Rule 3 are output in line 2 of the algorithm genRules(). Thus, H_1 = {{Chicken}, {Milk}}. The function ap-genRules() is then called. Line 2 of ap-genRules() produces H_2 = {{Chicken, Milk}}. The following rule is then generated:

Rule 4: Clothes → Milk, Chicken [sup = 3/7, conf = 3/3].

Thus, three association rules are generated from the frequent itemset {Chicken, Clothes, Milk} in F_3, namely Rule 1, Rule 3 and Rule 4. ■

2.3 Data Formats for Association Rule Mining

So far, we have used only transaction data for mining association rules. Market basket data sets are naturally of this format. Text documents can be seen as transaction data as well. Each document is a transaction, and each distinctive word is an item. Duplicate words are removed.

However, mining can also be performed on relational tables. We just need to convert a table data set to a transaction data set, which is fairly straightforward if each attribute in the table takes **categorical** values. We simply change each value to an **attribute–value** pair.

Example 6: The table data in Fig. 2.5(A) can be converted to the transaction data in Fig. 2.5(B). Each attribute–value pair is considered an **item**. Using only values is not sufficient in the transaction form because different attributes may have the same values. For example, without including attribute names, value a's for Attribute1 and Attribute2 are not distinguishable. After the conversion, Fig. 2.5(B) can be used in mining. ■

If an attribute takes numerical values, it becomes complex. We need to first discretize its value range into intervals, and treat each interval as a categorical value. For example, an attribute's value range is from 1–100. We may want to divide it into 5 equal-sized intervals, 1–20, 21–40, 41–60, 61–80, and 81–100. Each interval is then treated as a categorical value. Discretization can be done manually based on expert knowledge or automatically. There are several existing algorithms [151, 501].

A point to note is that for a table data set, the join step of the candidate generation function (Fig. 2.3) needs to be slightly modified in order to ensure that it does not join two itemsets to produce a candidate itemset containing two items from the same attribute.

Clearly, we can also convert a transaction data set to a table data set using a binary representation and treating each item in I as an attribute. If a transaction contains an item, its attribute value is 1, and 0 otherwise.

2.4 Mining with Multiple Minimum Supports

The key element that makes association rule mining practical is the minsup threshold. It is used to prune the search space and to limit the number of frequent itemsets and rules generated. However, using only a single min-

Attribute1	Attribute2	Atribute3
a	a	x
b	n	y

(A) Table data

t_1: (Attribute1, a), (Attribute2, a), (Attribute3, x)
t_2: (Attribute1, b), (Attribute2, n), (Attribute3, y)

(B) Transaction data

Fig. 2.5. From a table data set to a transaction data set

sup implicitly assumes that all items in the data are of the same nature and/or have similar frequencies in the database. This is often not the case in real-life applications. In many applications, some items appear very frequently in the data, while some other items rarely appear. If the frequencies of items vary a great deal, we will encounter two problems [344]:

1. If the minsup is set too high, we will not find rules that involve infrequent items or **rare items** in the data.
2. In order to find rules that involve both frequent and rare items, we have to set the minsup very low. However, this may cause combinatorial explosion and make mining impossible because those frequent items will be associated with one another in all possible ways.

Let us use an example to illustrate the above problem with a very low minsup, which will actually introduce another problem.

Example 7: In a supermarket transaction data set, in order to find rules involving those infrequently purchased items such as FoodProcessor and CookingPan (they generate more profits per item), we need to set the minsup very low. Let us use only frequent itemsets in this example as they are generated first and rules are produced from them. They are also the source of all the problems. Now assume we set a very low minsup of 0.005%. We find the following meaningful frequent itemset:

{FoodProcessor, CookingPan} [sup = 0.006%].

However, this low minsup may also cause the following two meaningless itemsets being discovered:

f_1: {Bread, Cheese, Egg, Bagel, Milk, Sugar, Butter} [sup = 0.007%],

f_2: {Bread, Egg, Milk, CookingPan} [sup = 0.006%].

Knowing that 0.007% of the customers buy the seven items in f_1 together is useless because all these items are so frequently purchased in a supermar-

ket. Worst still, they will almost certainly cause combinatorial explosion! For itemsets involving such items to be useful, their supports have to be much higher. Similarly, knowing that 0.006% of the customers buy the four items in f_2 together is also meaningless because Bread, Egg and Milk are purchased on almost every grocery shopping trip. ■

This dilemma is called the **rare item problem**. Using a single minsup for the whole data set is inadequate because it cannot capture the inherent natures and/or frequency differences of items in the database. By the natures of items we mean that some items, by nature, appear more frequently than others. For example, in a supermarket, people buy FoodProcessor and CookingPan much less frequently than Bread and Milk. The situation is the same for online stores. In general, those durable and/or expensive goods are bought less often, but each of them generates more profit. It is thus important to capture rules involving less frequent items. However, we must do so without allowing frequent items to produce too many meaningless rules with very low supports and cause combinatorial explosion [344].

One common solution to this problem is to partition the data into several smaller blocks (subsets), each of which contains only items of similar frequencies. Mining is then done separately for each block using a different minsup. This approach is, however, not satisfactory because itemsets or rules that involve items across different blocks will not be found.

A better solution is to allow the user to specify multiple minimum supports, i.e., to specify a different **minimum item support** (**MIS**) to each item. Thus, different itemsets need to satisfy different minimum supports depending on what items are in the itemsets. This model thus enables us to achieve our objective of finding itemsets involving rare items without causing frequent items to generate too many meaningless itemsets. This method helps solve the problem of f_1. To deal with the problem of f_2, we prevent itemsets that contain both very frequent items and very rare items from being generated. A **constraint** will be introduced to realize this.

An **interesting by-product** of this extended model is that it enables the user to easily instruct the algorithm to generate only itemsets that contain certain items but not itemsets that contain only the other items. This can be done by setting the MIS values to more than 100% (e.g., 101%) for these other items. This capability is very useful in practice because in many applications the user is only interested in certain types of itemsets or rules.

2.4.1 Extended Model

To allow multiple minimum supports, the original model in Sect. 2.1 needs to be extended. In the extended model, the minimum support of a rule is

expressed in terms of **minimum item supports** (**MIS**) of the items that appear in the rule. That is, each item in the data can have a MIS value specified by the user. By providing different MIS values for different items, the user effectively expresses different support requirements for different rules. It seems that specifying a MIS value for each item is a difficult task. This is not so as we will see at the end of Sect. 2.4.2.

Let MIS(i) be the MIS value of item i. The **minimum support** of a rule R is the lowest MIS value among the items in the rule. That is, a rule R,

$$i_1, i_2, \ldots, i_k \rightarrow i_{k+1}, \ldots, i_r,$$

satisfies its minimum support if the rule's actual support in the data is greater than or equal to:

$$min(\text{MIS}(i_1), \text{MIS}(i_2), \ldots, \text{MIS}(i_r)).$$

Minimum item supports thus enable us to achieve the goal of having higher minimum supports for rules that involve only frequent items, and having lower minimum supports for rules that involve less frequent items.

Example 8: Consider the set of items in a data set, {Bread, Shoes, Clothes}. The user-specified MIS values are as follows:

MIS(Bread) = 2% MIS(Clothes) = 0.2% MIS(Shoes) = 0.1%.

The following rule doesn't satisfy its minimum support:

Clothes $\rightarrow$ Bread [sup = 0.15%, conf = 70%].

This is so because *min*(MIS(Bread), MIS(Clothes)) = 0.2%. The following rule satisfies its minimum support:

Clothes $\rightarrow$ Shoes [sup = 0.15%, conf = 70%].

because *min*(MIS(Clothes), MIS(Shoes)) = 0.1%. ■

As we explained earlier, the **downward closure property** holds the key to pruning in the Apriori algorithm. However, in the new model, if we use the Apriori algorithm to find all frequent itemsets, the downward closure property no longer holds.

Example 9: Consider the four items 1, 2, 3 and 4 in a data set. Their minimum item supports are:

MIS(1) = 10% MIS(2) = 20% MIS(3) = 5% MIS(4) = 6%.

If we find that itemset {1, 2} has a support of 9% at level 2, then it does not satisfy either MIS(1) or MIS(2). Using the Apriori algorithm, this itemset is discarded since it is not frequent. Then, the potentially frequent itemsets {1, 2, 3} and {1, 2, 4} will not be generated for level 3. Clearly, itemsets {1,

2, 3} and {1, 2, 4} may be frequent because MIS(3) is only 5% and MIS(4) is 6%. It is thus wrong to discard {1, 2}. However, if we do not discard {1, 2}, the downward closure property is lost. ■

Below, we present an algorithm to solve this problem. The essential idea is to sort the items according to their MIS values in ascending order to avoid the problem.

Note that MIS values prevent low support itemsets involving only frequent items from being generated because their individual MIS values are all high. To prevent very frequent items and very rare items from appearing in the same itemset, we introduce the **support difference constraint**.

Let $sup(i)$ be the actual support of item i in the data. For each itemset s, the support difference constraint is as follows:

$$max_{i \in s}\{sup(i)\} - min_{i \in s}\{sup(i)\} \leq \varphi,$$

where $0 \leq \varphi \leq 1$ is the user-specified **maximum support difference**, and it is the same for all itemsets. The constraint basically limits the difference between the largest and the smallest actual supports of items in itemset s to φ. This constraint can reduce the number of itemsets generated dramatically, and it does not affect the downward closure property.

2.4.2 Mining Algorithm

The new algorithm generalizes the Apriori algorithm for finding frequent itemsets. We call the algorithm, **MS-Apriori**. When there is only one MIS value (for all items), it reduces to the Apriori algorithm.

Like Apriori, MS-Apriori is also based on level-wise search. It generates all frequent itemsets by making multiple passes over the data. However, there is an exception in the second pass as we will see later.

The key operation in the new algorithm is the sorting of the items in I in ascending order of their MIS values. This order is fixed and used in all subsequent operations of the algorithm. The items in each itemset follow this order. For example, in Example 9 of the four items 1, 2, 3 and 4 and their given MIS values, the items are sorted as follows: 3, 4, 1, 2. This order helps solve the problem identified above.

Let F_k denote the set of frequent k-itemsets. Each itemset w is of the following form, $\{w[1], w[2], \ldots, w[k]\}$, which consists of items, $w[1]$, $w[2]$, …, $w[k]$, where $MIS(w[1]) \leq MIS(w[2]) \leq \ldots \leq MIS(w[k])$. The algorithm MS-Apriori is given in Fig. 2.6. Line 1 performs the sorting on I according to the MIS value of each item (stored in MS). Line 2 makes the first pass over the data using the function init-pass(), which takes two arguments, the

```
Algorithm MS-Apriori(T, MS, φ)                    // MS stores all MIS values
1   M ← sort(I, MS);                              // according to MIS(i)'s stored in MS
2   L ← init-pass(M, T);                          // make the first pass over T
3   F1 ← {{l} | l ∈ L, l.count/n ≥ MIS(l)};       // n is the size of T
4   for (k = 2; Fk−1 ≠ ∅; k++) do
5     if k = 2 then
6       Ck ← level2-candidate-gen(L, φ)           // k = 2
7     else Ck ← MScandidate-gen(Fk−1, φ)
8     endif;
9     for each transaction t ∈ T do
10      for each candidate c ∈ Ck do
11        if c is contained in t then             // c is a subset of t
12          c.count++
13        if c − {c[1]} is contained in t then    // c without the first item
14          (c − {c[1]}).count++
15      endfor
16    endfor
17    Fk ← {c ∈ Ck | c.count/n ≥ MIS(c[1])}
18  endfor
19  return F ← ⋃k Fk;
```

Fig. 2.6. The MS-Apriori algorithm

data set T and the sorted items M, to produce the seeds L for generating candidate itemsets of length 2, i.e., C_2. init-pass() has two steps:

1. It first scans the data once to record the support count of each item.
2. It then follows the sorted order to find the first item i in M that meets MIS(i). i is inserted into L. For each subsequent item j in M after i, if $j.count/n \geq$ MIS(i), then j is also inserted into L, where $j.count$ is the support count of j, and n is the total number of transactions in T.

Frequent 1-itemsets (F_1) are obtained from L (line 3). It is easy to show that all frequent 1-itemsets are in F_1.

Example 10: Let us follow Example 9 and the given MIS values for the four items. Assume our data set has 100 transactions (not limited to the four items). The first pass over the data gives us the following support counts: {3}.*count* = 6, {4}.*count* = 3, {1}.*count* = 9 and {2}.*count* = 25. Then,

$$L = \{3, 1, 2\}, \text{ and } F_1 = \{\{3\}, \{2\}\}.$$

Item 4 is not in L because 4.*count*/n < MIS(3) (= 5%), and {1} is not in F_1 because 1.*count* / n < MIS(1) (= 10%). ■

For each subsequent pass (or data scan), say pass k, the algorithm performs three operations.

1. The frequent itemsets in F_{k-1} found in the $(k-1)th$ pass are used to generate the candidates C_k using the MScandidate-gen() function (line 7). However, there is a special case, i.e., when $k = 2$ (line 6), for which the candidate generation function is different, i.e., level2-candidate-gen().
2. It then scans the data and updates various support counts of the candidates in C_k (line 9–16). For each candidate c, we need to update its support count (lines 11–12) and also the support count of c without the first item (lines 13–14), i.e., $c - \{c[1]\}$, which is used in rule generation and will be discussed in Sect. 2.4.3. If rule generation is not required, lines 13 and 14 can be deleted.
3. The frequent itemsets (F_k) for the pass are identified in line 17.

We present candidate generation functions level2-candidate-gen() and MScandidate-gen() below.

Level2-candidate-gen function: It takes an argument L, and returns a superset of the set of all frequent 2-itemsets. The algorithm is given in Fig. 2.7. Note that in line 5, we use $|sup(h) - sup(l)| \leq \varphi$ because $sup(l)$ may not be lower than $sup(h)$, although $MIS(l) \leq MIS(h)$.

Example 11: Let us continue with Example 10. We set $\varphi = 10\%$. Recall the MIS values of the four items are (in Example 9):

MIS(1) = 10% MIS(2) = 20%
MIS(3) = 5% MIS(4) = 6%.

The level2-candidate-gen() function in Fig. 2.7 produces

$C_2 = \{\{3, 1\}\}$.

{1, 2} is not a candidate because the support count of item 1 is only 9 (or 9%), less than MIS(1) (= 10%). Hence, {1, 2} cannot be frequent. {3, 2} is not a candidate because $sup(3) = 6\%$ and $sup(2) = 25\%$ and their difference is greater than $\varphi = 10\%$ ■

Note that we must use L rather than F_1 because F_1 does not contain those items that may satisfy the MIS of an earlier item (in the sorted order) but not the MIS of itself, e.g., item 1 in the above example. Using L, the problem discussed in Sect. 2.4.1 is solved for C_2.

MScandidate-gen function: The algorithm is given in Fig. 2.8, which is similar to the candidate-gen function in the Apriori algorithm. It also has two steps, the **join step** and the **pruning step**. The join step (lines 2–6) is the same as that in the candidate-gen() function. The pruning step (lines 8–12) is, however, different.

For each $(k-1)$-subset s of c, if s is not in F_{k-1}, c can be deleted from C_k. However, there is an exception, which is when s does not include $c[1]$

```
Function level2-candidate-gen(L, φ)
1  C2 ← ∅;                                  // initialize the set of candidates
2  for each item l in L in the same order do
3     if l.count/n ≥ MIS(l) then
4        for each item h in L that is after l do
5           if h.count/n ≥ MIS(l) and |sup(h) − sup(l)| ≤ φ then
6              C2 ← C2 ∪ {{l, h}};   // insert the candidate {l, h} into C2
```

Fig. 2.7. The level2-candidate-gen function

```
Function MScandidate-gen(F_{k−1}, φ)
1   C_k ← ∅;                                  // initialize the set of candidates
2   forall f1, f2 ∈ F_k                       // find all pairs of frequent itemsets
3     with f1 = {i1, … , i_{k−2}, i_{k−1}}    // that differ only in the last item
4     and  f2 = {i1, … , i_{k−2}, i'_{k−1}}
5     and i_{k-1} < i'_{k−1} and |sup(i_{k-1}) − sup(i'_{k−1})| ≤ φ  do
6        c ← {i1, …, i_{k−1}, i'_{k−1}};    // join the two itemsets f1 and f2
7        C_k ← C_k ∪ {c};                   // insert the candidate itemset c into C_k
8        for each (k−1)-subset s of c do
9           if (c[1] ∈ s) or (MIS(c[2]) = MIS(c[1])) then
10             if (s ∉ F_{k−1}) then
11                delete c from C_k;         // delete c from the set of candidates
12       endfor
13  endfor
14  return C_k;                               // return the generated candidates
```

Fig. 2.8. The MScandidate-gen function

(there is only one such s). That is, the first item of c, which has the lowest MIS value, is not in s. Even if s is not in F_{k-1}, we cannot delete c because we cannot be sure that s does not satisfy MIS(c[1]), although we know that it does not satisfy MIS(c[2]), unless MIS(c[2]) = MIS(c[1]) (line 9).

Example 12: Let F_3 = {{1, 2, 3}, {1, 2, 5}, {1, 3, 4}, {1, 3, 5}, {1, 4, 5}, {1, 4, 6}, {2, 3, 5}}. Items in each itemset are in the sorted order. The join step produces (we ignore the support difference constraint here)

{1, 2, 3, 5}, {1, 3, 4, 5} and {1, 4, 5, 6}.

The pruning step deletes {1, 4, 5, 6} because {1, 5, 6} is not in F_3. We are then left with C_4 = {{1, 2, 3, 5}, {1, 3, 4, 5}}. {1, 3, 4, 5} is not deleted although {3, 4, 5} is not in F_3 because the minimum support of {3, 4, 5} is MIS(3), which may be higher than MIS(1). Although {3, 4, 5} does not satisfy MIS(3), we cannot be sure that it does not satisfy MIS(1). However, if MIS(3) = MIS(1), then {1, 3, 4, 5} can also be deleted. ■

The problem discussed in Sect. 2.4.1 is solved for C_k ($k > 2$) because, due to the sorting, we do not need to extend a frequent (k–1)-itemset with any item that has a lower MIS value. Let us see a complete example.

Example 13: Given the following seven transactions,

Beef, Bread
Bread, Clothes
Bread, Clothes, Milk
Cheese, Boots
Beef, Bread, Cheese, Shoes
Beef, Bread, Cheese, Milk
Bread, Milk, Clothes

and MIS(Milk) = 50%, MIS(Bread) = 70%, and 25% for all other items. Again, the support difference constraint is not used. The following frequent itemsets are produced:

F_1 = {{Beef}, {Cheese}, {Clothes}, {Bread}}
F_2 = {{Beef, Cheese}, {Beef, Bread}, {Cheese, Bread}
{Clothes, Bread}, {Clothes, Milk}}
F_3 = {{Beef, Cheese, Bread}, {Clothes, Milk, Bread}}. ■

To conclude this sub-section, let us further discuss two important issues:

1. Specify MIS values for items: This is usually done in two ways:
   - Assign a MIS value to each item according to its actual support/frequency in the data set T. For example, if the actual support of item i in T is $sup(i)$, then the MIS value for i may be computed with $\lambda \times sup(i)$, where λ is a parameter ($0 \leq \lambda \leq 1$) and is the same for all items in T.
   - Group items into clusters (or blocks). Items in each cluster have similar frequencies. All items in the same cluster are given the same MIS value. We should note that in the extended model frequent itemsets involving items from different clusters will be found.
2. Generate itemsets that must contain certain items: As mentioned earlier, the extended model enables the user to instruct the algorithm to generate itemsets that must contain certain items, or not to generate any itemsets consisting of only the other items. Let us see an example.

   Example 14: Given the data set in Example 13, if we want to generate frequent itemsets that must contain at least one item in {Boots, Bread, Cheese, Milk, Shoes}, or not to generate itemsets involving only Beef and/or Clothes, we can simply set

   MIS(Beef) = 101%, and MIS(Clothes) = 101%

Then the algorithm will not generate the itemsets, {Beef}, {Clothes} and {Beef, Clothes}. However, it will still generate such frequent itemsets as {Cheese, Beef} and {Cheese, Bread, Beef}. ■

In many applications, this feature comes quite handy because the user is often only interested in certain types of itemsets or rules.

2.4.3 Rule Generation

Association rules are generated using frequent itemsets. In the case of a single minsup, if f is a frequent itemset and f_{sub} is a subset of f, then f_{sub} must also be a frequent itemset. All their support counts are computed and recorded by the Apriori algorithm. Then, the confidence of each possible rule can be easily calculated without seeing the data again.

However, in the case of MS-Apriori, if we only record the support count of each frequent itemset, it is not sufficient. Let us see why.

Example 15: Recall in Example 8, we have

MIS(Bread) = 2% MIS(Clothes) = 0.2% MIS(Shoes) = 0.1%.

If the actual support for the itemset {Clothes, Bread} is 0.15%, and for the itemset {Shoes, Clothes, Bread} is 0.12%, according to MS-Apriori, {Clothes, Bread} is not a frequent itemset since its support is less than MIS(Clothes). However, {Shoes, Clothes, Bread} is a frequent itemset as its actual support is greater than

min(MIS(Shoes), MIS(Clothes), MIS(Bread)) = MIS(Shoes)).

We now have a problem in computing the confidence of the rule,

Clothes, Bread → Shoes

because the itemset {Clothes, Bread} is not a frequent itemset and thus its support count is not recorded. In fact, we may not be able to compute the confidences of the following rules either:

Clothes → Shoes, Bread
Bread → Shoes, Clothes

because {Clothes} and {Bread} may not be frequent. ■

Lemma: The above problem may occur only when the item that has the lowest MIS value in the itemset is in the consequent of the rule (which may have multiple items). We call this problem the **head-item problem**.

Proof by contradiction: Let f be a frequent itemset, and $a \in f$ be the item with the lowest MIS value in f (a is called the **head item**). Thus, f uses

MIS(a) as its minsup. We want to form a rule, $X \rightarrow Y$, where X, $Y \subset f$, $X \cup Y = f$ and $X \cap Y = \emptyset$. Our examples above already show that the head-item problem may occur when $a \in Y$. Now assume that the problem can also occur when $a \in X$. Since $a \in X$ and $X \subset f$, a must have the lowest MIS value in X and X must be a frequent itemset, which is ensured by the MS-Apriori algorithm. Hence, the support count of X is recorded. Since f is a frequent itemset and its support count is also recorded, then we can compute the confidence of $X \rightarrow Y$. This contradicts our assumption. ■

The lemma indicates that we need to record the support count of $f - \{a\}$. This is achieved by lines 13–14 in MS-Apriori (Fig. 2.6). All problems in Example 15 are solved. A similar rule generation function as genRules() in Apriori can be designed to generate rules with multiple minimum supports.

2.5 Mining Class Association Rules

The mining models studied so far do not use any targets. That is, any item can appear as a consequent or condition of a rule. However, in some applications, the user is interested in only rules with some fixed **target items** on the right-hand side. For example, the user has a collection of text documents from some topics (target items), and he/she wants to know what words are correlated with each topic. In [352], a data mining system based entirely on such rules (called **class association rules)** is reported, which is in production use in Motorola for many different applications. In the Web environment, class association rules are also useful because many types of Web data are in the form of transactions, e.g., search queries issued by users, and pages clicked by visitors. There are often target items as well, e.g., advertisements. Web sites want to know how user activities are associated with advertisements that they may like to view. This touches the issue of classification or prediction, which we will study in the next chapter.

2.5.1 Problem Definition

Let T be a transaction data set consisting of n transactions. Each transaction is labeled with a class y. Let I be the set of all items in T, Y be the set of all **class labels** (or target items) and $I \cap Y = \emptyset$. A **class association rule** (**CAR**) is an implication of the form

$$X \rightarrow y, \text{ where } X \subseteq I, \text{ and } y \in Y.$$

The definitions of **support** and **confidence** are the same as those for nor-

mal association rules. In general, a class association rule is different from a normal association rule in two ways:

1. The consequent of a CAR has only a single item, while the consequent of a normal association rule can have any number of items.
2. The consequent y of a CAR can only be from the class label set Y, i.e., $y \in Y$. No item from I can appear as the consequent, and no class label can appear as a rule condition. In contrast, a normal association rule can have any item as a condition or a consequent.

Objective: The problem of mining CARs is to generate the complete set of CARs that satisfies the user-specified minimum support (minsup) and minimum confidence (minconf) constraints.

Example 16: Figure 2.9 shows a data set which has seven text documents. Each document is a transaction and consists of a set of keywords. Each transaction is also labeled with a topic class (education or sport).

I = {Student, Teach, School, City, Game, Baseball, Basketball, Team, Coach, Player, Spectator}
Y = {Education, Sport}.

	Transactions	**Class**
doc 1:	Student, Teach, School	: Education
doc 2:	Student, School	: Education
doc 3:	Teach, School, City, Game	: Education
doc 4:	Baseball, Basketball	: Sport
doc 5:	Basketball, Player, Spectator	: Sport
doc 6:	Baseball, Coach, Game, Team	: Sport
doc 7:	Basketball, Team, City, Game	: Sport

Fig. 2.9. An example of a data set for mining class association rules

Let minsup = 20% and minconf = 60%. The following are two examples of class association rules:

Student, School → Education [sup= 2/7, conf = 2/2]
Game → Sport [sup= 2/7, conf = 2/3]. ■

A question that one may ask is: can we mine the data by simply using the Apriori algorithm and then perform a post-processing of the resulting rules to select only those class association rules? In principle, the answer is yes because CARs are a special type of association rules. However, in practice this is often difficult or even impossible because of combinatorial explosion, i.e., the number of rules generated in this way can be huge.

2.5.2 Mining Algorithm

Unlike normal association rules, CARs can be mined directly in a single step. The key operation is to find all **ruleitems** that have support above minsup. A **ruleitem** is of the form:

(*condset*, *y*),

where **condset** $\subseteq I$ is a set of items, and $y \in Y$ is a class label. The support count of a condset (called **condsupCount**) is the number of transactions in T that contain the condset. The support count of a ruleitem (called **rulesupCount**) is the number of transactions in T that contain the condset and are labeled with class y. Each ruleitem basically represents a rule:

$condset \rightarrow y$,

whose **support** is (*rulesupCount* / *n*), where *n* is the total number of transactions in *T*, and whose **confidence** is (*rulesupCount / condsupCount*).

Ruleitems that satisfy the minsup are called **frequent** ruleitems, while the rest are called infrequent ruleitems. For example, ({Student, School}, Education) is a ruleitem in T of Fig. 2.9. The support count of the condset {Student, School} is 2, and the support count of the ruleitem is also 2. Then the support of the ruleitem is 2/7 (= 28.6%), and the confidence of the ruleitem is 100%. If minsup = 10%, then the ruleitem satisfies the minsup threshold. We say that it is frequent. If minconf = 80%, then the ruleitem satisfies the minconf threshold. We say that the ruleitem is **confident**. We thus have the class association rule:

Student, School $\rightarrow$ Education [sup= 2/7, conf = 2/2].

The rule generation algorithm, called **CAR-Apriori**, is given in Fig. 2.10, which is based on the Apriori algorithm. Like the Apriori algorithm, CAR-Apriori generates all the frequent ruleitems by making multiple passes over the data. In the first pass, it computes the support count of each 1-ruleitem (containing only one item in its condset) (line 1). The set of all 1-candidate ruleitems considered is:

$C_1 = \{(\{i\}, y) \mid i \in I, \text{and } y \in Y\}$,

which basically associates each item in I (or in the transaction data set T) with every class label. Line 2 determines whether the candidate 1-ruleitems are frequent. From frequent 1-ruleitems, we generate 1-condition CARs (rules with only one condition) (line 3). In a subsequent pass, say k, it starts with the seed set of (k–1)-ruleitems found to be frequent in the (k–1)-th pass, and uses this seed set to generate new possibly frequent k-ruleitems, called **candidate *k*-ruleitems** (C_k in line 5). The actual support

Algorithm CAR-Apriori(T)
1 $C_1 \leftarrow$ init-pass(T); // the first pass over T
2 $F_1 \leftarrow \{f \mid f \in C_1, f.\ rulesupCount / n \geq minsup\}$;
3 $CAR_1 \leftarrow \{f \mid f \in F_1, f.rulesupCount / f.condsupCount \geq minconf\}$;
4 **for** ($k = 2$; $F_{k-1} \neq \emptyset$; k++) **do**
5 $C_k \leftarrow$ CARcandidate-gen(F_{k-1});
6 **for** each transaction $t \in T$ **do**
7 **for** each candidate $c \in C_k$ **do**
8 **if** $c.condset$ is contained in t **then** // c is a subset of t
9 $c.condsupCount$++;
10 **if** $t.class = c.class$ **then**
11 $c.rulesupCount$++
12 **endfor**
13 **end-for**
14 $F_k \leftarrow \{c \in C_k \mid c.rulesupCount / n \geq minsup\}$;
15 $CAR_k \leftarrow \{f \mid f \in F_k, f.rulesupCount / f.$condsupCount $\geq minconf\}$;
16 **endfor**
17 **return** $CAR \leftarrow \bigcup_k CAR_k$;

Fig. 2.10. The CAR-Apriori algorithm

counts, both *condsupCount* and *rulesupCount*, are updated during the scan of the data (lines 6–13) for each candidate k-ruleitem. At the end of the data scan, it determines which of the candidate k-ruleitems in C_k are actually frequent (line 14). From the frequent k-ruleitems, line 15 generates k-condition CARs (class association rules with k conditions).

One interesting note about ruleitem generation is that if a ruleitem/rule has a confidence of 100%, then extending the ruleitem with more conditions (adding items to its condset) will also result in rules with 100% confidence although their supports may drop with additional items. In some applications, we may consider these subsequent rules **redundant** because additional conditions do not provide any more information. Then, we should not extend such ruleitems in candidate generation for the next level, which can reduce the number of generated rules substantially. If desired, redundancy handling can be added in the CAR-Apriori algorithm easily.

The CARcandidate-gen() function is very similar to the candidate-gen() function in the Apriori algorithm, and it is thus omitted. The only difference is that in CARcandidate-gen() ruleitems with the same class are joined by joining their condsets.

Example 17: Let us work on a complete example using our data in Fig. 2.9. We set minsup = 15%, and minconf = 70%

F_1: {({School}, Education):(3, 3), ({Student}, Education):(2, 2),
({Teach}, Education):(2, 2), ({Baseball}, Sport):(2, 2),

({Basketball}, Sport):(3, 3), ({Game}, Sport):(3, 2),
({Team}, Sport):(2, 2)}

Note: The two numbers within the parentheses after each ruleitem are its *condSupCount* and *ruleSupCount* respectively.

CAR_1: School → Education [sup = 3/7, conf = 3/3]
Student → Education [sup = 2/7, conf = 2/2]
Teach → Education [sup = 2/7, conf = 2/2]
Baseball → Sport [sup = 2/7, conf = 2/2]
Basketball → Sport [sup = 3/7, conf = 3/3]
Game → Sport [sup = 2/7, conf = 2/3]
Team → Sport [sup = 2/7, conf = 2/2]

Note: We do not deal with rule redundancy in this example.

C_2: {({School, Student}, Education), ({School, Teach}, Education),
({Student, Teach}, Education), ({Baseball, Basketball}, Sport),
({Baseball, Game}, Sport), ({Baseball, Team}, Sport),
({Basketball, Game}, Sport), ({Basketball, Team}, Sport),
({Game, Team}, Sport)}

F_2: {({School, Student}, Education):(2, 2),
({School, Teach}, Education):(2, 2), ({Game, Team}, Sport):(2, 2)}

CAR_2: School, Student → Education [sup = 2/7, conf = 2/2]
School, Teach → Education [sup = 2/7, conf = 2/2]
Game, Team → Sport [sup = 2/7, conf = 2/2] ■

We note that for many applications involving target items, the data sets used are relational tables. They need to be converted to transaction forms before mining. We can use the method in Sect. 2.3 for the purpose.

Example 18: In Fig. 2.11(A), the data set has three data attributes and a class attribute with two possible values, positive and negative. It is converted to the transaction data in Fig. 2.11(B). Notice that for each class, we only use its original value. There is no need to attach the attribute "Class"

Attribute1	Attribute2	Atribute3	Class
a	a	x	positive
b	n	y	negative

(A) Table data

t_1: (Attribute1, a), (Attribute2, a), (Attribute3, x) : Positive
t_2: (Attribute1, b), (Attribute2, n), (Attribute3, y) : negative

(B) Transaction data

Fig. 2.11. Converting a table data set (A) to a transaction data set (B)

because there is no ambiguity. As discussed in Sect. 2.3, for each numeric attribute, its value range needs to be discretized into intervals either manually or automatically before conversion and rule mining. There are many discretization algorithms. Interested readers are referred to [151]. ■

2.5.3 Mining with Multiple Minimum Supports

The concept of mining with multiple minimum supports discussed in Sect. 2.4 can be incorporated in class association rule mining in two ways:

1. **Multiple minimum class supports:** The user can specify different minimum supports for different classes. For example, the user has a data set with two classes, Yes and No. Based on the application requirement, he/she may want all rules of class Yes to have the minimum support of 5% and all rules of class No to have the minimum support of 20%.
2. **Multiple minimum item supports:** The user can specify a minimum item support for every item (either a class item/label or a non-class item). This is more general and is similar to normal association rule mining discussed in Sect. 2.4.

For both approaches, similar mining algorithms to that given in Sect. 2.4 can be devised. The *support difference constraint* in Sect. 2.4.1 can be incorporated as well. Like normal association rule mining with multiple minimum supports, by setting minimum class and/or item supports to more than 100% for some items, the user effectively instructs the algorithm not to generate rules involving only these items.

Finally, although we have discussed only multiple minimum supports so far, we can easily use different minimum confidences for different classes as well, which provides an additional flexibility in applications.

2.6 Basic Concepts of Sequential Patterns

Association rule mining does not consider the order of transactions. However, in many applications such orderings are significant. For example, in market basket analysis, it is interesting to know whether people buy some items in sequence, e.g., buying bed first and then buying bed sheets some time later. In Web usage mining, it is useful to find navigational patterns in a Web site from sequences of page visits of users (see Chap. 12). In text mining, considering the ordering of words in a sentence is vital for finding linguistic or language patterns (see Chap. 11). For these applications, association rules will not be appropriate. Sequential patterns are needed. Be-

low, we define the problem of mining sequential patterns and introduce the main concepts involved.

Let $I = \{i_1, i_2, ..., i_m\}$ be a set of items. A **sequence** is an ordered list of itemsets. Recall an **itemset** X is a non-empty set of items $X \subseteq I$. We denote a sequence s by $\langle a_1 a_2 ... a_r \rangle$, where a_i is an itemset, which is also called an **element** of s. We denote an element (or an itemset) of a sequence by $\{x_1, x_2, \ldots, x_k\}$, where $x_j \in I$ is an item. We assume without loss of generality that items in an element of a sequence are in **lexicographic order**. An item can occur only once in an element of a sequence, but can occur multiple times in different elements. The **size** of a sequence is the number of elements (or itemsets) in the sequence. The **length** of a sequence is the number of items in the sequence. A sequence of length k is called a ***k*-sequence**. If an item occurs multiple times in different elements of a sequence, each occurrence contributes to the value of k. A sequence $s_1 = \langle a_1 a_2 ... a_r \rangle$ is a **subsequence** of another sequence $s_2 = \langle b_1 b_2 ... b_v \rangle$, or s_2 is a **supersequence** of s_1, if there exist integers $1 \leq j_1 < j_2 < ... < j_{r-1} < j_r \leq v$ such that $a_1 \subseteq b_{j_1}$, $a_2 \subseteq b_{j_2}, ..., a_r \subseteq b_{j_r}$. We also say that s_2 **contains** s_1.

Example 19: Let $I = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. The sequence $\langle \{3\}\{4, 5\}\{8\} \rangle$ is contained in (or is a subsequence of) $\langle \{6\}\{3, 7\}\{9\}\{4, 5, 8\}\{3, 8\} \rangle$ because $\{3\} \subseteq \{3, 7\}$, $\{4, 5\} \subseteq \{4, 5, 8\}$, and $\{8\} \subseteq \{3, 8\}$. However, $\langle \{3\}\{8\} \rangle$ is not contained in $\langle \{3, 8\} \rangle$ or vice versa. The size of the sequence $\langle \{3\}\{4, 5\}\{8\} \rangle$ is 3, and the length of the sequence is 4. ■

Objective: Given a set S of input **data sequences** (or **sequence database**), the problem of mining sequential patterns is to find all sequences that have a user-specified **minimum support**. Each such sequence is called a **frequent sequence**, or a **sequential pattern**. The **support** for a sequence is the fraction of total data sequences in S that contains this sequence.

Example 20: We use the market basket analysis as an example. Each sequence in this context represents an ordered list of transactions of a particular customer. A transaction is a set of items that the customer purchased at a time (called the transaction time). Then transactions in the sequence are ordered by increasing transaction time. Table 2.1 shows a transaction database which is already sorted according to customer ID (the major key) and transaction time (the minor key). Table 2.2 gives the data sequences (also called **customer sequences**). Table 2.3 gives the output sequential patterns with the minimum support of 25%, i.e., two customers. ■

Table 2.1. A set of transactions sorted by customer ID and transaction time

Customer ID	Transaction Time	Transaction (items bought)
1	July 20, 2005	30
1	July 25, 2005	90
2	July 9, 2005	10, 20
2	July 14, 2005	30
2	July 20, 2005	10, 40, 60, 70
3	July 25, 2005	30, 50, 70, 80
4	July 25, 2005	30
4	July 29, 2005	30, 40, 70, 80
4	August 2, 2005	90
5	July 12, 2005	90

Table 2.2. The sequence database produced from the transactions in Table 2.1.

Customer ID	Data Sequence
1	⟨{30} {90}⟩
2	⟨{10, 20} {30} {10, 40, 60, 70}⟩
3	⟨{30, 50, 70, 80}⟩
4	⟨{30} {30, 40, 70, 80} {90}⟩
5	⟨{90}⟩

Table 2.3. The final output sequential patterns

	Sequential Patterns with Support ≥ 25%
1-sequences	⟨{30}⟩, ⟨{40}⟩, ⟨{70}⟩, ⟨{80}⟩, ⟨{90}⟩
2-sequences	⟨{30} {40}⟩, ⟨{30} {70}⟩, ⟨{30}, {90}⟩, ⟨{30, 70}⟩, ⟨{30, 80}⟩, ⟨{40, 70}⟩, ⟨{70, 80}⟩
3-sequences	⟨{30} {40, 70}⟩, ⟨{30, 70, 80}⟩

2.7 Mining Sequential Patterns Based on GSP

This section describes two algorithms for mining sequential patterns based on the GSP algorithm in [500]: the original GSP, which uses a single minimum support, and MS-GSP, which uses multiple minimum supports.

2.7.1 GSP Algorithm

GSP works in almost the same way as the Apriori algorithm. We still use F_k to store the set of all frequent k-sequences, and C_k to store the set of all

```
Algorithm GSP(S)
1   C_1 ← init-pass(S);                          // the first pass over S
2   F_1 ← {⟨{f}⟩| f ∈ C_1, f.count/n ≥ minsup};  // n is the number of sequences in S
3   for (k = 2; F_{k-1} ≠ ∅; k++) do             // subsequent passes over S
4     C_k ← candidate-gen-SPM(F_{k-1});
5     for each data sequence s ∈ S do            // scan the data once
6       for each candidate c ∈ C_k do
7         if c is contained in s then
8           c.count++;                           // increment the support count
9       endfor
10    endfor
11    F_k ← {c ∈ C_k | c.count/n ≥ minsup}
12  endfor
13  return ⋃_k F_k;
```

Fig. 2.12. The GSP Algorithm for generating sequential patterns

Function candidate-gen-SPM(F_{k-1}) // SPM: Sequential Pattern Mining

1. **Join step.** Candidate sequences are generated by joining F_{k-1} with F_{k-1}. A sequence s_1 joins with s_2 if the subsequence obtained by dropping the first item of s_1 is the same as the subsequence obtained by dropping the last item of s_2. The candidate sequence generated by joining s_1 with s_2 is the sequence s_1 extended with the last item in s_2. There are two cases:
   - the added item forms a separate element if it was a separate element in s_2, and is appended at the end of s_1 in the merged sequence, and
   - the added item is part of the last element of s_1 in the merged sequence otherwise.

   When joining F_1 with F_1, we need to add the item in s_2 both as part of an itemset and as a separate element. That is, joining $\langle\{x\}\rangle$ with $\langle\{y\}\rangle$ gives us both $\langle\{x, y\}\rangle$ and $\langle\{x\}\{y\}\rangle$. Note that x and y in $\{x, y\}$ are ordered.
2. **Prune step.** A candidate sequence is pruned if any one of its (k–1)-subsequences is infrequent (without minimum support).

Fig. 2.13. The candidate-gen-SPM function

candidate k-sequences. The algorithm is given in Fig. 2.12. The main difference is in the candidate generation, candidate-gen-SPM(), which is given in Fig. 2.13. We use an example to illustrate the function.

Example 21: Table 2.4 shows F_3, and C_4 after the join and prune steps. In the join step, the sequence ⟨{1, 2}{4}⟩ joins with ⟨{2}{4, 5}⟩ to produce ⟨{1, 2}{4, 5}⟩, and joins with ⟨{2}{4}{6}⟩ to produce ⟨{1, 2}{4} {6}⟩. The other sequences cannot be joined. For instance, ⟨{1}{4, 5}⟩ does not join with any sequence since there is no sequence of the form ⟨{4, 5}{x}⟩ or ⟨{4, 5, x}⟩. In the prune step, ⟨{1, 2}{4} {6}⟩ is removed since ⟨{1}{4} {6}⟩ is not in F_3. ■

Table 2.4. Candidate generation: an example

Frequent 3-sequences	Candidate 4-sequences	
	after joining	after pruning
⟨{1, 2} {4}⟩	⟨{1, 2} {4, 5}⟩	⟨{1, 2} {4, 5}⟩
⟨{1, 2} {5}⟩	⟨{1, 2} {4} {6}⟩	
⟨{1} {4, 5}⟩		
⟨{1, 4} {6}⟩		
⟨{2} {4, 5}⟩		
⟨{2} {4} {6}⟩		

2.7.2 Mining with Multiple Minimum Supports

As in association rule mining, using a single minimum support in sequential pattern mining is also a limitation for many applications because some items appear very frequently in the data, while some others appear rarely.

Example 22: One of the Web mining tasks is the mining of comparative sentences such as "*the picture quality of camera X is better than that of camera Y.*" from product reviews, forum postings and blogs (see Chap. 11). Such a sentence usually contains a comparative indicator word such as better in the example. We want to discover linguistic patterns involving a set of given comparative indicators, e.g., better, more, less, ahead, win, superior, etc. Some of these indicators (e.g., more and better) appear very frequently in natural language sentences, while some others (e.g., win and ahead) appear rarely. In order to find patterns that contain such rare indicators, we have to use a very low minsup. However, this causes patterns involving frequent indicators to generate a huge number of spurious patterns. Moreover, we need a way to tell the algorithm that we only want patterns that contain at least one comparative indicator. Using GSP with a single minsup is no longer appropriate. The multiple minimum supports model solves both problems nicely. ■

We again use the concept of **minimum item supports** (**MIS**). The user is allowed to assign each item a MIS value. By providing different MIS values for different items, the user essentially expresses different support requirements for different sequential patterns. To ease the task of specifying many MIS values by the user, the same strategies as those for mining association rules can also be applied here (see Sect. 2.4.2).

Let MIS(i) be the MIS value of item i. The **minimum support** of a sequential pattern P is the lowest MIS value among the items in the pattern. Let the set of items in P be: $i_1, i_2, \ldots, i_r$. The minimum support for P is:

```
Algorithm MS-GSP(S, MS)                          // MS stores all MIS values
1   M ← sort(I, MS);                             // according to MIS(i)'s stored in MS
2   L ← init-pass(M, S);                         // make the first pass over S
3   F1 ← {⟨{l}⟩ | l ∈ L, l.count/n ≥ MIS(l)};    // n is the size of S
4   for (k = 2; Fk–1 ≠ ∅; k++) do
5     if k = 2 then
6       Ck ← level2-candidate-gen-SPM(L)
7     else Ck ← MScandidate-gen-SPM(Fk–1)
8     endif
9     for each data sequence s ∈ S do
10      for each candidate c ∈ Ck do
11        if c is contained in s then
12          c.count++
13        if c' is contained in s, where c' is c after an occurrence of
              c.minMISItem is removed from c then
14          c.rest.count++          // c.rest: c without c.minMISItem
15      endfor
16    endfor
17    Fk ← {c ∈ Ck | c.count/n ≥ MIS(c.minMISItem)}
18  endfor
19  return F ← ⋃k Fk;
```

Fig. 2.14. The MS-GSP algorithm

$$\text{minsup}(P) = min(\text{MIS}(i_1), \text{MIS}(i_2), \ldots, \text{MIS}(i_r)).$$

The new algorithm, called **MS-GSP**, is given in Fig. 2.14. It generalizes the GSP algorithm in Fig. 2.12. Like GSP, MS-GSP is also based on level-wise search. Line 1 sorts the items in ascending order according to their MIS values stored in *MS*. Line 2 makes the first pass over the sequence data using the function init-pass(), which performs the same function as that in MS-Apriori to produce the seeds set L for generating the set of candidate sequences of length 2, i.e., C_2. Frequent 1-sequences (F_1) are obtained from L (line 3).

For each subsequent pass, the algorithm works similarly to MS-Apriori. The function level2-candidate-gen-SPM() can be designed based on level2-candidate-gen in MS-Apriori and the join step in Fig. 2.13. MScandidate-gen-SPM() is, however, complex, which we will discuss shortly.

In line 13, *c.minMISItem* gives the item that has the lowest MIS value in the candidate sequence c. Unlike that in MS-Apriori, where the first item in each itemset has the lowest MIS value, in sequential pattern mining the item with the lowest MIS value may appear anywhere in a sequence. Similar to those in MS-Apriori, lines 13 and 14 are used to ensure that all sequential rules can be generated after MS-GSP without scanning the original data. Note that in traditional sequential pattern mining, sequential rules are not defined. We will define several types in Sect. 2.9.

Let us now discuss MScandidate-gen-SPM(). In MS-Apriori, the ordering of items is not important and thus we put the item with the lowest MIS value in each itemset as the first item of the itemset, which simplifies the join step. However, for sequential pattern mining, we cannot artificially put the item with the lowest MIS value as the first item in a sequence because the ordering of items is significant. This causes problems for joining.

Example 23: Assume we have a sequence $s_1 = \langle\{1, 2\}\{4\}\rangle$ in F_3, from which we want to generate candidate sequences for the next level. Suppose that item 1 has the lowest MIS value in s_1. We use the candidate generation function in Fig. 2.13. Assume also that the sequence $s_2 = \langle\{2\}\{4, 5\}\rangle$ is not in F_3 because its minimum support is not satisfied. Then we will not generate the candidate $\langle\{1, 2\}\{4, 5\}\rangle$. However, $\langle\{1, 2\}\{4, 5\}\rangle$ can be frequent because items 2, 4, and 5 may have higher MIS values than item 1. ■

To deal with this problem, let us make an observation. The problem only occurs when the first item in the sequence s_1 or the last item in the sequence s_2 is the only item with the lowest MIS value, i.e., no other item in s_1 (or s_2) has the same lowest MIS value. If the item (say x) with the lowest MIS value is not the first item in s_1, then s_2 must contain x, and the candidate generation function in Fig. 2.13 will still be applicable. The same reasoning goes for the last item of s_2. Thus, we only need special treatment for these two cases.

Let us see how to deal with the first case, i.e., the first item is the only item with the lowest MIS value. We use an example to develop the idea. Assume we have the frequent 3-sequence of $s_1 = \langle\{1, 2\}\{4\}\rangle$. Based on the algorithm in Fig. 2.13, s_1 may be extended to generate two possible candidates using $\langle\{2\}\{4\}\{x\}\rangle$ and $\langle\{2\}\{4, x\}\rangle$

$c_1 = \langle\{1, 2\}\{4\}\{x\}\rangle$ and $c_2 = \langle\{1, 2\}\{4, x\}\rangle$,

where x is an item. However, $\langle\{2\}\{4\}\{x\}\rangle$ and $\langle\{2\}\{4, x\}\rangle$ may not be frequent because items 2, 4, and x may have higher MIS values than item 1, but we still need to generate c_1 and c_2 because they can be frequent. A different join strategy is thus needed.

We observe that for c_1 to be frequent, the subsequence $s_2 = \langle\{1\}\{4\}\{x\}\rangle$ must be frequent. Then, we can use s_1 and s_2 to generate c_1. c_2 can be generated in a similar manner with $s_2 = \langle\{1\}\{4, x\}\rangle$. s_2 is basically the subsequence of c_1 (or c_2) without the second item. Here we assume that the MIS value of x is higher than item 1. Otherwise, it falls into the second case.

Let us see the same problem for the case where the last item has the only lowest MIS value. Again, we use an example to illustrate. Assume we have the frequent 3-sequence $s_2 = \langle\{3, 5\}\{1\}\rangle$. It can be extended to produce two possible candidates based on the algorithm in Fig. 2.13,

Function MScandidate-gen-SPM(F_{k-1})

1 **Join Step**. Candidate sequences are generated by joining F_{k-1} with F_{k-1}.

2 **if** the MIS value of the first item in a sequence (denoted by s_1) is less than (<) the MIS value of every other item in s_1 **then** // s_1 and s_2 can be equal
Sequence s_1 joins with s_2 if (1) the subsequences obtained by dropping the second item of s_1 and the last item of s_2 are the same, and (2) the MIS value of the last item of s_2 is greater than that of the first item of s_1. Candidate sequences are generated by extending s_1 with the last item of s_2:
- **if** the last item l in s_2 is a separate element **then**
  $\{l\}$ is appended at the end of s_1 as a separate element to form a candidate sequence c_1.
  if (the length and the size of s_1 are both 2) AND (the last item of s_2 is greater than the last item of s_1) **then** // maintain lexicographic order
  l is added at the end of the last element of s_1 to form another candidate sequence c_2.
- **else if** ((the length of s_1 is 2 and the size of s_1 is 1) AND (the last item of s_2 is greater than the last item of s_1)) OR (the length of s_1 is greater than 2) **then**
  the last item in s_2 is added at the end of the last element of s_1 to form the candidate sequence c_2.

3 **elseif** the MIS value of the last item in a sequence (denoted by s_2) is less than (<) the MIS value of every other item in s_2 **then**
A similar method to the one above can be used in the reverse order.

4 **else** use the **Join Step** in Fig. 2.13

5 **Prune step**: A candidate sequence is pruned if any one of its (k–1)-subsequences is infrequent (without minimum support) except the subsequence that does not contain the item with strictly the lowest MIS value.

Fig. 2.15. The MScandidate-gen-SPM function

$c_1 = \langle\{x\}\{3, 5\}\{1\}\rangle$, and $c_2 = \langle\{x, 3, 5\}\{1\}\rangle$.

For c_1 to be frequent, the subsequence $s_1 = \langle\{x\}\{3\}\{1\}\rangle$ has to be frequent (we assume that the MIS value of x is higher than that of item 1). Thus, we can use s_1 and s_2 to generate c_1. c_2 can be generated with $s_1 = \langle\{x, 3\}\{1\}\rangle$. s_1 is basically the subsequence of c_1 (or c_2) without the second last item.

The MScandidate-gen-SPM() function is given in Fig. 2.15, which is self-explanatory. Some special treatments are needed for 2-sequences because the same s_1 (or s_2) may generate two candidate sequences. We use two examples to show the working of the function.

Example 24: Consider the items 1, 2, 3, 4, 5, and 6 with their MIS values,

MIS(1) = 0.03 MIS(2) = 0.05 MIS(3) = 0.03
MIS(4) = 0.07 MIS(5) = 0.08 MIS(6) = 0.09.

The data set has 100 sequences. The following frequent 3-sequences are in F_3 with their actual support counts attached after ":":

(a). ⟨{1}{4}{5}⟩:4 (b). ⟨{1}{4}{6}⟩:5 (c). ⟨{1}{5}{6}⟩:6
(d). ⟨{1}{5, 6}⟩:5 (e). ⟨{1}{6}{3}⟩:4 (f). ⟨{6}{3}{6}⟩:9
(g). ⟨{5, 6}{3}⟩:5 (h). ⟨{5}{4}{3}⟩:4 (i). ⟨{4}{5}{3}⟩:7.

For sequence (a) (= s_1), item 1 has the lowest MIS value. It cannot join with sequence (b) because condition (1) in Fig. 2.15 is not satisfied. However, (a) can join with (c) to produce the candidate sequence, ⟨{1}{4}{5}{6}⟩. (a) can also join with (d) to produce ⟨{1}{4}{5, 6}⟩. (b) can join with (e) to produce ⟨{1}{4}{6}{3}⟩, which is pruned subsequently because ⟨{1}{4}{3}⟩ is infrequent. (d) and (e) can be joined to give ⟨{1}{5, 6}{3}⟩, but it is pruned because ⟨{1}{5}{3}⟩ does not exist. (e) can join with (f) to produce ⟨{1}{6}{3}{6}⟩ which is done in line 4 because both item 1 and item 3 in (e) have the same MIS value. However, it is pruned because ⟨{1}{3}{6}⟩ is infrequent. We do not join (d) and (g), although they can be joined based on the algorithm in Fig. 2.13, because the first item of (d) has the lowest MIS value and we use a different join method for such sequences.

Now we look at 3-sequences whose last item has strictly the lowest MIS value. (i) (= s_1) can join with (h) (= s_2) to produce ⟨{4}{5}{4}{3}⟩. However, it is pruned because ⟨{4}{4}{3}⟩ is not in F_3. ■

Example 25: Now we consider generating candidates from frequent 2-sequences, which is special as we noted earlier. We use the same items and MIS values in Example 24. The following frequent 2-sequences are in F_2 with their actual support counts attached after ":":

(a). ⟨{1}{5}⟩:6 (b). ⟨{1}{6}⟩:7 (c) ⟨{5}{4}⟩:8
(d). ⟨{1, 5}⟩:6 (e). ⟨{1, 6}⟩:6.

(a) can join with (b) to produce both ⟨{1}{5}{6}⟩ and ⟨{1}{5, 6}⟩. (b) can join with (d) to produce ⟨{1, 5}{6}⟩. (e) can join with (a) to produce ⟨{1, 6}{5}⟩. Clearly, there are other joins. Again, (a) will not join with (c). ■

Note that the **support difference constraint** in Sect. 2.4.1 can also be included. We omitted it to simplify the algorithm as it is already complex. Also, the user can instruct the algorithm to generate only certain sequential patterns or not to generate others by setting the MIS values suitably.

2.8 Mining Sequential Patterns Based on PrefixSpan

We now introduce another sequential pattern mining algorithm, called PrefixSpan [439], which does not generate candidates. Different from the GSP

algorithm [500], which can be regarded as performing breadth-first search to find all sequential patterns, PrefixSpan performs depth-first search.

2.8.1 PrefixSpan Algorithm

It is easy to introduce the original PrefixSpan algorithm using an example.

Example 26: Consider again mining sequential patterns from Table 2.2 with minsup = 25%. PrefixSpan first sorts all items in each element (or itemset) as shown in the table. Then, by one scan of the sequence database, it finds all frequent items, i.e., 30, 40, 70, 80 and 90. The corresponding length one sequential patterns are ⟨{30}⟩, ⟨{40}⟩, ⟨{70}⟩, ⟨{80}⟩ and ⟨{90}⟩.

We notice that the complete set of sequential patterns can actually be divided into five mutually exclusive subsets: the subset with prefix ⟨{30}⟩, the subset with prefix ⟨{40}⟩, the subset with prefix ⟨{70}⟩, the subset with prefix ⟨{80}⟩, and the subset with prefix ⟨{90}⟩. We only need to find the five subsets one by one.

To find sequential patterns having prefix ⟨{30}⟩, the algorithm extends the prefix by adding items to it one at a time. To add the next item x, there are two possibilities, i.e., x joining the last itemset of the prefix (i.e., ⟨{30, x}⟩) and x forming a separate itemset (i.e., ⟨{30}{x}⟩). PrefixSpan performs the task by first forming the ⟨{30}⟩-projected database and then finding all the cases of the two types in the projected database. The projected database is produced as follows: If a sequence contains item 30, then the suffix following the first 30 is extracted as a sequence in the projected database. Furthermore, since infrequent items cannot appear in a sequential pattern, all infrequent items are removed from the projection. The first sequence in our example, ⟨{30}{90}⟩, is projected to ⟨{90}⟩. The second sequence, ⟨{10, 20}{30}{10, 40, 60, 70}⟩, is projected to ⟨{40, 70}⟩, where the infrequent items 10 and 60 are removed. The third sequence ⟨{30, 50, 70, 80}⟩ is projected to ⟨{_, 70, 80}⟩, where the infrequent item 50 is removed. Note that the underline symbol "_" in this projection denotes that the items (only 30 in this case) in the last itemset of the prefix are in the same itemset as items 50, 70 and 80 in the sequence. The fourth sequence is projected to ⟨{30, 40, 70, 80}{90}⟩. The projection of the last sequence is empty since it does not contain item 30. The final projected database for prefix ⟨{30}⟩ contains the following sequences:

⟨{90}⟩, ⟨{40, 70}⟩, ⟨{_, 70, 80}⟩, and ⟨{30, 40, 70, 80}{90}⟩

By scanning the projected database once, PrefixSpan finds all possible one item extensions to the prefix, i.e., all x's for ⟨{30, x}⟩ and all x's for ⟨{30}{x}⟩. Let us discuss the details.

Find All Frequent Patterns of the Form ⟨{30, x}⟩: Two templates {_, x} and {30, x} are used to match each projected sequence to accumulate the support count for each possible x (here x matches any item). If in the same sequence multiple matches are found with the same x, they are only counted once. Note that in general, the second template should use the last itemset in the prefix rather than only its last item. In our example, they are the same because there is only one item in the last itemset of the prefix.

Find All Frequent Patterns of the Form ⟨{30}{x}⟩: In this case, x's are frequent items in the projected database that are not in the same itemset as the last item of the prefix.

Let us continue with our example. It is easy to check that both items 70 and 80 are in the same itemset as 30. That is, we have two frequent sequences ⟨{30, 70}⟩ and ⟨{30, 80}⟩. The support count of ⟨{30, 70}⟩ is 2 based on the projected database; one from the projected sequence ⟨{_, **70**, 80}⟩ (a {_, x} match) and one from the projected sequence ⟨{**30**, 40, **70**, 80}{90}⟩ (a {30, x} match). In both cases, the x's are the same, i.e., 70. Similarly, the support count of ⟨{30, 80}⟩ is 2 as well and thus frequent.

It is also easy to check that items 40, 70, and 90 are also frequent but not in the same itemset as 30. Thus, ⟨{30}{40}⟩, ⟨{30}{70}⟩, and ⟨{30}{90}⟩ are three sequential patterns. The set of sequential patterns having prefix ⟨{30}⟩ can be further divided into five mutually exclusive subsets: the ones with prefixes ⟨{30, 70}⟩, ⟨{30, 80}⟩, ⟨{30}{40}⟩, ⟨{30}{70}⟩, and ⟨{30}{90}⟩.

We can recursively find the five subsets by forming their corresponding projected databases. For example, to find sequential patterns having prefix ⟨{30}{40}⟩, we can form the ⟨{30}{40}⟩-projected database containing projections ⟨{_, 70}⟩ and ⟨{_, 70, 80}{90}⟩. Template ⟨{_, x}⟩ has two matches and in both cases x is 70. Thus, ⟨{30}{40, 70}⟩ is output as a sequential pattern. Since there is no other frequent item in this projected database, the prefix cannot grow longer. The depth-first search returns from this branch.

After completing the mining of the ⟨{30}⟩-projected database, we find all sequential patterns with prefix ⟨{30}⟩, i.e., ⟨{30}⟩, ⟨{30}{40}⟩, ⟨{30}{40, 70}⟩, ⟨{30}{70}⟩, ⟨{30}{90}⟩, ⟨{30, 70}⟩, ⟨{30, 80}⟩ and ⟨{30, 70, 80}⟩

By forming and mining the ⟨{40}⟩-, ⟨{70}⟩-, ⟨{80}⟩- and ⟨{90}⟩-projected databases, the remaining sequential patterns can be found. ■

The pseudo code of PrefixSpan can be found in [439]. Comparing to the breadth-first search of GSP, the key advantage of PrefixSpan is that it does not generate any candidates. It only counts the frequency of local items. With a low minimum support, a huge number of candidates can be generated by GSP, which can cause memory and computational problems.

2.8.2 Mining with Multiple Minimum Supports

The PrefixSpan algorithm can be adapted to mine with multiple minimum supports. Again, let MIS(i) be the user-specified **minimum item support** of item i. Let φ be the user-specified support difference threshold in the **support difference constraint** (Sect. 2.4.1), i.e., $|sup(i) - sup(j)| \leq \varphi$, where i and j are items in the same sequential pattern, and $sup(x)$ is the actual support of item x in the sequence database S. PrefixSpan can be modified as follows. We call the modified algorithm **MS-PS**.

1. Find every item i whose actual support in the sequence database S is at least MIS(i). i is called a frequent item.
2. Sort all the discovered frequent items in ascending order according to their MIS values. Let $i_1, \ldots, i_u$ be the frequent items in the sorted order.
3. For each item i_k in the above sorted order,
   (i) identify all the data sequences in S that contain i_k and at the same time remove every item j in each sequence that does not satisfy $|sup(j) - sup(i_k)| \leq \varphi$. The resulting set of sequences is denoted by S_k. Note that we are not using i_k as the prefix to project the database S.
   (ii) call the function r-PrefixSpan(i_k, S_k, count(MIS(i_k))) (*r*estricted *PrefixSpan*), which finds all sequential patterns that contain i_k, i.e., no pattern that does not contain i_k should be generated. r-PrefixSpan() uses count(MIS(i_k)) (the minimum support count in terms of the number of sequences) as the only minimum support for mining in S_k. The sequence count is easier to use than the MIS value in percentage, but they are equivalent. Once the complete set of such patterns is found from S_k, All occurrences of i_k are removed from S.

r-PrefixSpan() is almost the same as PrefixSpan with one important difference. During each recursive call, either the prefix or every sequence in the projected database must contain i_k because, as we stated above, this function finds only those frequent sequences that contain i_k. Another minor difference is that the support difference constraint needs to be checked during each projection as $sup(i_k)$ may not be the lowest in the pattern.

Example 27: Consider mining sequential patterns from Table 2.5. Let MIS(20) = 30% (3 sequences in minimum support count), MIS(30) = 20% (2 sequences), MIS(40) = 30% (3 sequences), and the MIS values for the rest of the items be 15% (2 sequences). We ignore the support difference constraint as it is simple. In step 1, we find three frequent items, 20, 30 and 40. After sorting in step 2, we have (30, 20, 40). We then go to step 3.

In the first iteration of step 3, we work on $i_1 = 30$. Step 3(i) gives us the second, fourth and sixth sequences in Table 2.5, i.e.,

Table 2.5. An example of a sequence database

Sequence ID	Data Sequence
1	⟨{20, 50}⟩
2	⟨{40}{30}{40, 60}⟩
3	⟨{40, 90, 120}⟩
4	⟨{30}{20, 40}{40, 100}⟩
5	⟨{20, 40}{10}⟩
6	⟨{40}{30}{110}⟩
7	⟨{20}{80}{70}⟩

S_1 = {⟨{40}{30}{40, 60}⟩, ⟨{30}{20, 40}{40, 100}⟩, ⟨{40}{30}{110}⟩}.

We then run r-PrefixSpan(30, S_1, 3) in step 3(ii). The frequent items in S_1 are 30, and 40. They both have the support of 3 sequences. The length one frequent sequence is only ⟨{30}⟩. ⟨{40}⟩ is not included because we require that every frequent sequence must contain 30. We next find frequent sequences having prefix ⟨{30}⟩. The database S_1 is projected to give ⟨{40}⟩ and ⟨{40}{40}⟩. 20, 60 and 100 have been removed because their supports in S_1 are less than the required support for item 30 (i.e., 3 sequences). For the same reason, the projection of ⟨{40}{30}{110}⟩ is empty. Thus, we find a length two frequent sequence ⟨{30}{40}⟩. In this case, there is no item in the same itemset as 30 to form a frequent sequence of the form ⟨{30, x}⟩.

Next, we find frequent sequences with prefix ⟨{40}⟩. We again project S_1, which gives us only ⟨{30}{40}⟩ and ⟨{30}⟩. ⟨{40, 100}⟩ is not included because it does not contain 30. This projection gives us another length two frequent sequence ⟨{40}{30}⟩. The first iteration of step 3 ends.

In the second iteration of step 3, we work on i_2 = 20. Step 3(i) gives us the first, fourth, fifth and seventh sequences in Table 2.5 with item 30 removed, S_2 = {⟨{20, 50}⟩, ⟨{20, 40}{40, 100}⟩, ⟨{20, 40}{10}⟩, ⟨{20}{80}{70}⟩}. It is easy to see that only item 20 is frequent, and thus only a length one frequent sequence is generated, ⟨{20}⟩.

In the third iteration of step 3, we work on i_3 = 40. We can verify that again only one frequent sequence, i.e., ⟨{40}⟩, is found.

The final set of sequential patterns generated from the sequence database in Table 2.5 is {⟨{30}⟩, ⟨{20}⟩, ⟨{40}⟩, ⟨{40}{30}⟩, ⟨{30}{40}⟩}. ■

2.9 Generating Rules from Sequential Patterns

In classic sequential pattern mining, no rules are generated. It is, however, possible to define and generate many types of rules. This section intro-

duces only three types, **sequential rules**, **label sequential rules** and **class sequential rules**, which have been used in Web usage mining and Web content mining (see Chaps. 11 and 12).

2.9.1 Sequential Rules

A **sequential rule** (**SR**) is an implication of the form, $X \rightarrow Y$, where Y is a sequence and X is a **proper subsequence** of Y, i.e., X is a subsequence of Y and the length Y is greater than the length of X. The **support** of a sequential rule, $X \rightarrow Y$, in a sequence database S is the fraction of sequences in S that contain Y. The **confidence** of a sequential rule, $X \rightarrow Y$, in S is the proportion of sequences in S that contain X also contain Y.

Given a minimum support and a minimum confidence, according to the downward closure property, all the rules can be generated from frequent sequences without going to the original sequence data. Let us see an example of a sequential rule found from the data sequences in Table 2.6.

Table 2.6. An example of a sequence database for mining sequential rules

	Data Sequence
1	⟨{1}{3}{5}{7, 8, 9}⟩
2	⟨{1}{3}{6}{7, 8}⟩
3	⟨{1, 6}{7}⟩
4	⟨{1}{3}{5, 6}⟩
5	⟨{1}{3}{4}⟩

Example 28: Given the sequence database in Table 2.6, the minimum support of 30% and the minimum confidence of 60%, one of the sequential rules found is the following,

⟨{1}{7}⟩ → ⟨{1}{3}{7, 8}⟩ [sup = 2/5, conf = 2/3]

Data sequences 1, 2 and 3 contain ⟨{1}{7}⟩, and data sequences 1 and 2 contain ⟨{1}{3}{7, 8}⟩. ■

If multiple minimum supports are used, we can employ the results of multiple minimum support pattern mining to generate all the rules.

2.9.2 Label Sequential Rules

Sequential rules may not be restrictive enough in some applications. We introduce a special kind of sequential rules called **label sequential rules**. A label sequential rule (LSR) is of the form, $X \rightarrow Y$, where Y is a sequence

and X is a sequence produced from Y by replacing some of its items with wildcards. A wildcard is denoted by an "*" which matches any item. These replaced items are usually very important and are called **labels**. The labels are a small subset of all the items in the data.

Example 29: Given the sequence database in Table 2.6, the minimum support of 30% and the minimum confidence of 60%, one of the label sequential rules found is the following,

$$\langle\{1\}\{*\}\{7, *\}\rangle \rightarrow \langle\{1\}\{3\}\{7, 8\}\rangle \quad [\text{sup} = 2/5, \text{conf} = 2/2].$$

Notice the confidence change compared to the rule in Example 28. The supports of the two rules are the same. In this case, data sequences 1 and 2 contain $\langle\{1\}\{*\}\{7, *\}\rangle$, and they also contain $\langle\{1\}\{3\}\{7, 8\}\rangle$. Items 3 and 8 are labels. ■

LSRs are useful because in some applications we need to predict the labels in an input sequence, e.g., items 3 and 8 above. The confidence of the rule simply gives us the estimated probability that the two "*"s are 3 and 8 given that an input sequence contains $\langle\{1\}\{*\}\{7, *\}\rangle$. We will see an application of LSRs in Chap. 11, where we want to predict whether a word in a comparative sentence is an entity (e.g., a product name), which is a label.

Note that due to the use of wildcards, frequent sequences alone are not sufficient for computing rule confidences. Scanning the data is needed. Notice also that the same pattern may appear in a data sequence multiple times. Rule confidences thus can be defined in different ways according to application needs. The wildcards may also be restricted to match only certain types of items to make the label prediction meaningful and unambiguous (see some examples in Chap. 11).

2.9.3 Class Sequential Rules

Class sequential rules (CSR) are analogous to class association rules (CAR). Let S be a set of data sequences. Each sequence is also labeled with a class y. Let I be the set of all items in S, and Y be the set of all class labels, $I \cap Y = \varnothing$. Thus, the input data D for mining is represented with $\{(s_1, y_1), (s_2, y_2), \ldots, (s_n, y_n)\}$, where s_i is a sequence in S and $y_i \in Y$ is its class. A **class sequential rule** (**CSR**) is of the form

$$X \rightarrow y, \text{ where } X \text{ is a sequence, and } y \in Y.$$

A data instance (s_i, y_i) is said to **cover** a CSR, $X \rightarrow y$, if X is a subsequence of s_i. A data instance (s_i, y_i) is said to **satisfy** a CSR if X is a subsequence of s_i and $y_i = y$.

Example 30: Table 2.7 gives an example of a sequence database with five data sequences and two classes, $c1$ and $c2$. Using the minimum support of 30% and the minimum confidence of 60%, one of the discovered CSRs is: $\{1\}\{3\}\{7, 8\} \rightarrow c_1$ [sup = 2/5, conf = 2/3].

Data sequences 1 and 2 satisfy the rule, and data sequences 1, 2 and 5 cover the rule.

Table 2.7. An example of a sequence database for mining CSRs

	Data Sequence	Class
1	⟨{1}{3}{5}{7, 8, 9}⟩	c_1
2	⟨{1}{3}{6}{7, 8}⟩	c_1
3	⟨{1, 6}{9}⟩	c_2
4	⟨{3}{5, 6}⟩	c_2
5	⟨{1}{3}{4}{7, 8}⟩	c_2

As in class association rule mining, we can modify the GSP and Prefix- Span algorithms to produce algorithms for mining all CSRs. Similarly, we can also use multiple minimum class supports and/or multiple minimum item supports as in class association rule mining.

UNIT - II
Supervised and Unsupervised Learning

3 Supervised Learning

Supervised learning has been a great success in real-world applications. It is used in almost every domain, including text and Web domains. Supervised learning is also called **classification** or **inductive learning** in machine learning. This type of learning is analogous to human learning from past experiences to gain new knowledge in order to improve our ability to perform real-world tasks. However, since computers do not have "experiences", machine learning learns from data, which are collected in the past and represent past experiences in some real-world applications.

There are several types of supervised learning tasks. In this chapter, we focus on one particular type, namely, learning a target function that can be used to predict the values of a discrete class attribute. This type of learning has been the focus of the machine learning research and is perhaps also the most widely used learning paradigm in practice. This chapter introduces a number of such supervised learning techniques. They are used in almost every Web mining application. We will see their uses from Chaps. 6–12.

3.1 Basic Concepts

A data set used in the learning task consists of a set of data records, which are described by a set of attributes $A = \{A_1, A_2, \ldots, A_{|A|}\}$, where $|A|$ denotes the number of attributes or the size of the set A. The data set also has a special target attribute C, which is called the **class** attribute. In our subsequent discussions, we consider C separately from attributes in A due to its special status, i.e., we assume that C is not in A. The class attribute C has a set of discrete values, i.e., $C = \{c_1, c_2, \ldots, c_{|C|}\}$, where $|C|$ is the number of classes and $|C| \geq 2$. A class value is also called a **class label**. A data set for learning is simply a relational table. Each data record describes a piece of "past experience". In the machine learning and data mining literature, a data record is also called an **example**, an **instance**, a **case** or a **vector**. A data set basically consists of a set of examples or instances.

Given a data set D, the objective of learning is to produce a **classification/prediction function** to relate values of attributes in A and classes in C. The function can be used to predict the class values/labels of the future

data. The function is also called a **classification model**, a **predictive model** or simply a **classifier**. We will use these terms interchangeably in this book. It should be noted that the function/model can be in any form, e.g., a decision tree, a set of rules, a Bayesian model or a hyperplane.

Example 1: Table 3.1 shows a small loan application data set. It has four attributes. The first attribute is Age, which has three possible values, young, middle and old. The second attribute is Has_Job, which indicates whether an applicant has a job. Its possible values are true (has a job) and false (does not have a job). The third attribute is Own_house, which shows whether an applicant owns a house. The fourth attribute is Credit_rating, which has three possible values, fair, good and excellent. The last column is the Class attribute, which shows whether each loan application was approved (denoted by Yes) or not (denoted by No) in the past.

Table 3.1. A loan application data set

ID	Age	Has_job	Own_house	Credit_rating	Class
1	young	false	false	fair	**No**
2	young	false	false	good	**No**
3	young	true	false	good	**Yes**
4	young	true	true	fair	**Yes**
5	young	false	false	fair	**No**
6	middle	false	false	fair	**No**
7	middle	false	false	good	**No**
8	middle	true	true	good	**Yes**
9	middle	false	true	excellent	**Yes**
10	middle	false	true	excellent	**Yes**
11	old	false	true	excellent	**Yes**
12	old	false	true	good	**Yes**
13	old	true	false	good	**Yes**
14	old	true	false	excellent	**Yes**
15	old	false	false	fair	**No**

We want to learn a classification model from this data set that can be used to classify future loan applications. That is, when a new customer comes into the bank to apply for a loan, after inputting his/her age, whether he/she has a job, whether he/she owns a house, and his/her credit rating, the classification model should predict whether his/her loan application should be approved. ■

Our learning task is called **supervised learning** because the class labels (e.g., Yes and No values of the class attribute in Table 3.1) are provided in

the data. It is as if some teacher tells us the classes. This is in contrast to the **unsupervised learning**, where the classes are not known and the learning algorithm needs to automatically generate classes. Unsupervised learning is the topic of the next chapter.

The data set used for learning is called the **training data** (or **the training set)**. After a **model** is learned or built from the training data by a **learning algorithm**, it is evaluated using a set of **test data** (or **unseen data)** to assess the model accuracy.

It is important to note that the test data is not used in learning the classification model. The examples in the test data usually also have class labels. That is why the test data can be used to assess the accuracy of the learned model because we can check whether the class predicted for each test case by the model is the same as the actual class of the test case. In order to learn and also to test, the available data (which has classes) for learning is usually split into two disjoint subsets, the training set (for learning) and the test set (for testing). We will discuss this further in Sect. 3.3.

The accuracy of a classification model on a test set is defined as:

$$Accuracy = \frac{\text{Number of correct classifications}}{\text{Total number of test cases}}, \tag{1}$$

where a correct classification means that the learned model predicts the same class as the original class of the test case. There are also other measures that can be used. We will discuss them in Sect. 3.3.

We pause here to raises two important questions:

1. What do we mean by learning by a computer system?
2. What is the relationship between the training and the test data?

We answer the first question first. Given a data set D representing past "experiences", a task T and a performance measure M, a computer system is said to **learn** from the data to perform the task T if after learning the system's performance on the task T improves as measured by M. In other words, the learned model or knowledge helps the system to perform the task better as compared to no learning. Learning is the process of building the model or extracting the knowledge.

We use the data set in Example 1 to explain the idea. The task is to predict whether a loan application should be approved. The performance measure M is the accuracy in Equation (1). With the data set in Table 3.1, if there is no learning, all we can do is to guess randomly or to simply take the majority class (which is the `Yes` class). Suppose we use the majority class and announce that every future instance or case belongs to the class `Yes`. If the future data are drawn from the same distribution as the existing training data in Table 3.1, the estimated classification/prediction accuracy

on the future data is 9/15 = 0.6 as there are 9 `Yes` class examples out of the total of 15 examples in Table 3.1. The question is: can we do better with learning? If the learned model can indeed improve the accuracy, then the learning is said to be effective.

The second question in fact touches the **fundamental assumption of machine learning**, especially the theoretical study of machine learning. The assumption is that the distribution of training examples is identical to the distribution of test examples (including future unseen examples). In practical applications, this assumption is often violated to a certain degree. Strong violations will clearly result in poor classification accuracy, which is quite intuitive because if the test data behave very differently from the training data then the learned model will not perform well on the test data. To achieve good accuracy on the test data, training examples must be sufficiently representative of the test data.

We now illustrate the steps of learning in Fig. 3.1 based on the preceding discussions. In step 1, a learning algorithm uses the training data to generate a classification model. This step is also called the **training step** or **training phase**. In step 2, the learned model is tested using the test set to obtain the classification accuracy. This step is called the **testing step** or **testing phase**. If the accuracy of the learned model on the test data is satisfactory, the model can be used in real-world tasks to predict classes of new cases (which do not have classes). If the accuracy is not satisfactory, we need to go back and choose a different learning algorithm and/or do some further processing of the data (this step is called **data pre-processing**, not shown in the figure). A practical learning task typically involves many iterations of these steps before a satisfactory model is built. It is also possible that we are unable to build a satisfactory model due to a high degree of randomness in the data or limitations of current learning algorithms.

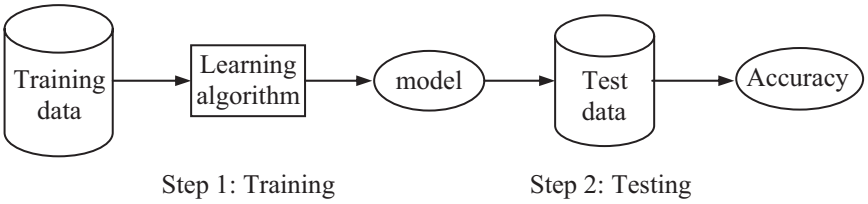


Fig. 3.1. The basic learning process: training and testing

From the next section onward, we study several supervised learning algorithms, except Sect. 3.3, which focuses on model/classifier evaluation.

We note that throughout the chapter we assume that the training and test data are available for learning. However, in many text and Web page related learning tasks, this is not true. Usually, we need to collect raw data,

design attributes and compute attribute values from the raw data. The reason is that the raw data in text and Web applications are often not suitable for learning either because their formats are not right or because there are no obvious attributes in the raw text documents or Web pages.

3.2 Decision Tree Induction

Decision tree learning is one of the most widely used techniques for classification. Its classification accuracy is competitive with other learning methods, and it is very efficient. The learned classification model is represented as a tree, called a **decision tree**. The techniques presented in this section are based on the C4.5 system from Quinlan [453].

Example 2: Figure 3.2 shows a possible decision tree learnt from the data in Table 3.1. The tree has two types of nodes, **decision nodes** (which are internal nodes) and **leaf nodes**. A decision node specifies some test (i.e., asks a question) on a single attribute. A leaf node indicates a class.

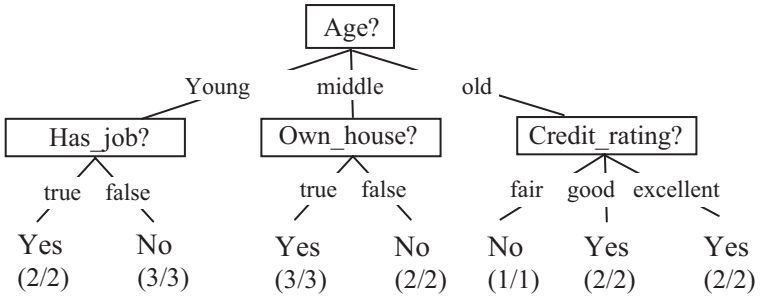


Fig. 3.2. A decision tree for the data in Table 3.1

The root node of the decision tree in Fig. 3.2 is Age, which basically asks the question: what is the age of the applicant? It has three possible answers or **outcomes**, which are the three possible values of Age. These three values form three tree branches/edges. The other internal nodes have the same meaning. Each leaf node gives a class value (Yes or No). (x/y) below each class means that x out of y training examples that reach this leaf node have the class of the leaf. For instance, the class of the left most leaf node is Yes. Two training examples (examples 3 and 4 in Table 3.1) reach here and both of them are of class Yes. ■

To use the decision tree in **testing**, we traverse the tree top-down according to the attribute values of the given test instance until we reach a leaf node. The class of the leaf is the predicted class of the test instance.

Example 3: We use the tree to predict the class of the following new instance, which describes a new loan applicant.

Age	Has_job	Own_house	Credit-rating	Class
young	false	false	good	?

Going through the decision tree, we find that the predicted class is No as we reach the second leaf node from the left. ■

A decision tree is constructed by partitioning the training data so that the resulting subsets are as pure as possible. A **pure subset** is one that contains only training examples of a single class. If we apply all the training data in Table 3.1 on the tree in Fig. 3.2, we will see that the training examples reaching each leaf node form a subset of examples that have the same class as the class of the leaf. In fact, we can see that from the x and y values in (x/y). We will discuss the decision tree building algorithm in Sect. 3.2.1.

An interesting question is: Is the tree in Fig. 3.2 unique for the data in Table 3.1? The answer is no. In fact, there are many possible trees that can be learned from the data. For example, Fig. 3.3 gives another decision tree, which is much smaller and is also able to partition the training data perfectly according to their classes.

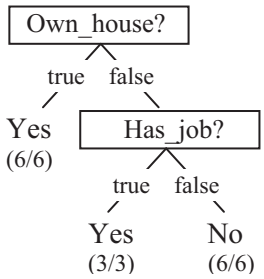


Fig. 3.3. A smaller tree for the data set in Table 3.1

In practice, one wants to have a small and accurate tree for many reasons. A smaller tree is more general and also tends to be more accurate (we will discuss this later). It is also easier to understand by human users. In many applications, the user understanding of the classifier is important. For example, in some medical applications, doctors want to understand the model that classifies whether a person has a particular disease. It is not satisfactory to simply produce a classification because without understanding why the decision is made the doctor may not trust the system and/or does not gain useful knowledge.

It is useful to note that in both Fig. 3.2 and Fig. 3.3, the training examples that reach each leaf node all have the same class (see the values of

(x/y) at each leaf node). However, for most real-life data sets, this is usually not the case. That is, the examples that reach a particular leaf node are not of the same class, i.e., $x \leq y$. The value of x/y is, in fact, the **confidence** (conf) value used in association rule mining, and x is the **support count**. This suggests that a decision tree can be converted to a set of if-then rules.

Yes, indeed. The conversion is done as follows: Each path from the root to a leaf forms a rule. All the decision nodes along the path form the conditions of the rule and the leaf node or the class forms the consequent. For each rule, a support and confidence can be attached. Note that in most classification systems, these two values are not provided. We add them here to see the connection of association rules and decision trees.

Example 4: The tree in Fig. 3.3 generates three rules. "," means "and".

Own_house = true → Class =Yes [sup=6/15, conf=6/6]
Own_house = false, Has_job = true → Class = Yes [sup=3/15, conf=3/3]
Own_house = false, Has_job = false → Class = No [sup=6/15, conf=6/6].

We can see that these rules are of the same format as association rules. However, the rules above are only a small subset of the rules that can be found in the data of Table 3.1. For instance, the decision tree in Fig. 3.3 does not find the following rule:

Age = young, Has_job = false → Class = No [sup=3/15, conf=3/3].

Thus, we say that a decision tree only finds a subset of rules that exist in data, which is sufficient for classification. The objective of association rule mining is to find all rules subject to some minimum support and minimum confidence constraints. Thus, the two methods have different objectives. We will discuss these issues again in Sect. 3.5 when we show that association rules can be used for classification as well, which is obvious.

An interesting and important property of a decision tree and its resulting set of rules is that the tree paths or the rules are **mutually exclusive** and **exhaustive**. This means that every data instance is **covered** by a single rule (a tree path) and a single rule only. By **covering** a data instance, we mean that the instance satisfies the conditions of the rule.

We also say that a decision tree **generalizes** the data as a tree is a smaller (more compact) description of the data, i.e., it captures the key regularities in the data. Then, the problem becomes building the best tree that is small and accurate. It turns out that finding the best tree that models the data is a NP-complete problem [248]. All existing algorithms use heuristic methods for tree building. Below, we study one of the most successful techniques.

. **Algorithm** decisionTree(D, A, T)
1 **if** D contains only training examples of the same class $c_j \in C$ **then**
2 make T a leaf node labeled with class c_j;
3 **elseif** $A = \emptyset$ **then**
4 make T a leaf node labeled with c_j, which is the most frequent class in D
5 **else** // D contains examples belonging to a mixture of classes. We select a single
6 // attribute to partition D into subsets so that each subset is purer
7 p_0 = impurityEval-1(D);
8 **for** each attribute $A_i \in A$ (=$\{A_1, A_2, \ldots, A_k\}$) **do**
9 p_i = impurityEval-2(A_i, D)
10 **endfor**
11 Select $A_g \in \{A_1, A_2, \ldots, A_k\}$ that gives the biggest impurity reduction, computed using $p_0 - p_i$;
12 **if** $p_0 - p_g < threshold$ **then** // A_g does not significantly reduce impurity p_0
13 make T a leaf node labeled with c_j, the most frequent class in D.
14 **else** // A_g is able to reduce impurity p_0
15 Make T a decision node on A_g;
16 Let the possible values of A_g be $v_1, v_2, \ldots, v_m$. Partition D into m disjoint subsets $D_1, D_2, \ldots, D_m$ based on the m values of A_g.
17 **for** each D_j in $\{D_1, D_2, \ldots, D_m\}$ **do**
18 **if** $D_j \neq \emptyset$ **then**
19 create a branch (edge) node T_j for v_j as a child node of T;
20 decisionTree(D_j, $A-\{A_g\}$, T_j) // A_g is removed
21 **endif**
22 **endfor**
23 **endif**
24 **endif**

Fig. 3.4. A decision tree learning algorithm

3.2.1 Learning Algorithm

As indicated earlier, a decision tree T simply partitions the training data set D into disjoint subsets so that each subset is as pure as possible (of the same class). The learning of a tree is typically done using the **divide-and-conquer** strategy that recursively partitions the data to produce the tree. At the beginning, all the examples are at the root. As the tree grows, the examples are sub-divided recursively. A decision tree learning algorithm is given in Fig. 3.4. For now, we assume that every attribute in D takes discrete values. This assumption is not necessary as we will see later.

The **stopping criteria** of the recursion are in lines 1–4 in Fig. 3.4. The algorithm stops when all the training examples in the current data are of the same class, or when every attribute has been used along the current tree

path. In tree learning, each successive recursion chooses the **best attribute** to partition the data at the current node according to the values of the attribute. The best attribute is selected based on a function that aims to minimize the impurity after the partitioning (lines 7–11). In other words, it maximizes the purity. The key in decision tree learning is thus the choice of the **impurity function**, which is used in lines 7, 9 and 11 in Fig. 3.4. The recursive recall of the algorithm is in line 20, which takes the subset of training examples at the node for further partitioning to extend the tree.

This is a greedy algorithm with no backtracking. Once a node is created, it will not be revised or revisited no matter what happens subsequently.

3.2.2 Impurity Function

Before presenting the impurity function, we use an example to show what the impurity function aims to do intuitively.

Example 5: Figure 3.5 shows two possible root nodes for the data in Table 3.1.

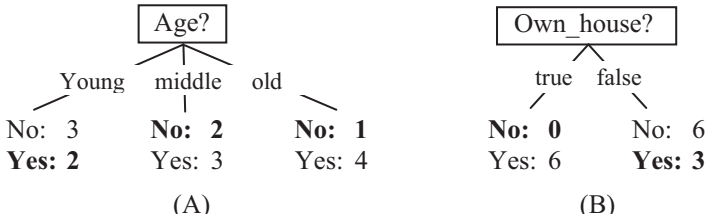


Fig. 3.5. Two possible root nodes or two possible attributes for the root node

Fig. 3.5(A) uses Age as the root node, and Fig. 3.5(B) uses Own_house as the root node. Their possible values (or outcomes) are the branches. At each branch, we listed the number of training examples of each class (No or Yes) that land or reach there. Fig. 3.5(B) is obviously a better choice for the root. From a prediction or classification point of view, Fig. 3.5(B) makes fewer mistakes than Fig. 3.5(A). In Fig. 3.5(B), when Own_house = true every example has the class Yes. When Own_house = false, if we take majority class (the most frequent class), which is No, we make three mistakes/errors. If we look at Fig. 3.5(A), the situation is worse. If we take the majority class for each branch, we make five mistakes (marked in bold). Thus, we say that the impurity of the tree in Fig. 3.5(A) is higher than the tree in Fig. 3.5(B). To learn a decision tree, we prefer Own_house to Age to be the root node. Instead of counting the number of mistakes or errors, C4.5 uses a more principled approach to perform this evaluation on every attribute in order to choose the best attribute to build the tree. ■

The most popular impurity functions used for decision tree learning are **information gain** and **information gain ratio**, which are used in C4.5 as two options. Let us first discuss information gain, which can be extended slightly to produce information gain ratio.

The information gain measure is based on the **entropy** function from **information theory** [484]:

$$entropy(D) = -\sum_{j=1}^{|C|} \Pr(c_j) \log_2 \Pr(c_j) \tag{2}$$

$$\sum_{j=1}^{|C|} \Pr(c_j) = 1,$$

where $\Pr(c_j)$ is the probability of class c_j in data set D, which is the number of examples of class c_j in D divided by the total number of examples in D. In the entropy computation, we define $0\log 0 = 0$. The unit of entropy is **bit**. Let us use an example to get a feeling of what this function does.

Example 6: Assume we have a data set D with only two classes, positive and negative. Let us see the entropy values for three different compositions of positive and negative examples:

1. The data set D has 50% positive examples ($\Pr(positive) = 0.5$) and 50% negative examples ($\Pr(negative) = 0.5$).

$$entropy(D) = -0.5 \times \log_2 0.5 - 0.5 \times \log_2 0.5 = 1.$$

2. The data set D has 20% positive examples ($\Pr(positive) = 0.2$) and 80% negative examples ($\Pr(negative) = 0.8$).

$$entropy(D) = -0.2 \times \log_2 0.2 - 0.8 \times \log_2 0.8 = 0.722.$$

3. The data set D has 100% positive examples ($\Pr(positive) = 1$) and no negative examples, ($\Pr(negative) = 0$).

$$entropy(D) = -1 \times \log_2 1 - 0 \times \log_2 0 = 0.$$

We can see a trend: When the data becomes purer and purer, the entropy value becomes smaller and smaller. In fact, it can be shown that for this binary case (two classes), when $\Pr(positive) = 0.5$ and $\Pr(negative) = 0.5$ the entropy has the maximum value, i.e., 1 bit. When all the data in D belong to one class the entropy has the minimum value, 0 bit. ■

It is clear that the entropy measures the amount of impurity or disorder in the data. That is exactly what we need in decision tree learning. We now describe the information gain measure, which uses the entropy function.

Information Gain

The idea is the following:

1. Given a data set D, we first use the entropy function (Equation 2) to compute the impurity value of D, which is *entropy*(D). The **impurityEval-1** function in line 7 of Fig. 3.4 performs this task.
2. Then, we want to know which attribute can reduce the impurity most if it is used to partition D. To find out, every attribute is evaluated (lines 8–10 in Fig. 3.4). Let the number of possible values of the attribute A_i be v. If we are going to use A_i to partition the data D, we will divide D into v disjoint subsets D_1, D_2, ..., D_v. The entropy after the partition is

$$entropy_{A_i}(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times entropy(D_j). \tag{3}$$

   The **impurityEval-2** function in line 9 of Fig. 3.4 performs this task.
3. The information gain of attribute A_i is computed with:

$$gain(D, A_i) = entropy(D) - entropy_{A_i}(D). \tag{4}$$

Clearly, the gain criterion measures the reduction in impurity or disorder. The *gain* measure is used in line 11 of Fig. 3.4, which chooses attribute A_g resulting in the largest reduction in impurity. If the gain of A_g is too small, the algorithm stops for the branch (line 12). Normally a threshold is used here. If choosing A_g is able to reduce impurity significantly, A_g is employed to partition the data to extend the tree further, and so on (lines 15–21 in Fig. 3.4). The process goes on recursively by building sub-trees using D_1, D_2, …, D_m (line 20). For subsequent tree extensions, we do not need A_g any more, as all training examples in each branch has the same A_g value.

Example 7: Let us compute the gain values for attributes Age, Own_house and Credit_Rating using the whole data set D in Table 3.1, i.e., we evaluate for the root node of a decision tree.

First, we compute the entropy of D. Since D has 6 No class training examples, and 9 Yes class training examples, we have

$$entropy(D) = -\frac{6}{15} \times \log_2 \frac{6}{15} - \frac{9}{15} \times \log_2 \frac{9}{15} = 0.971.$$

We then try Age, which partitions the data into 3 subsets (as Age has three possible values) D_1 (with Age=young), D_2 (with Age=middle), and D_3 (with Age=old). Each subset has five training examples. In Fig. 3.5, we also see the number of No class examples and the number of Yes examples in each subset (or in each branch).

$$entropy_{Age}(D) = -\frac{5}{15} \times entropy(D_1) - \frac{5}{15} \times entropy(D_2) - \frac{5}{15} \times entropy(D_3)$$
$$= \frac{5}{15} \times 0.971 + \frac{5}{15} \times 0.971 + \frac{5}{15} \times 0.722 = 0.888.$$

Likewise, we compute for Own_house, which partitions D into two subsets, D_1 (with Own_house=true) and D_2 (with Own_house=false).

$$entropy_{Own\_house}(D) = -\frac{6}{15} \times entropy(D_1) - \frac{9}{15} \times entropy(D_2)$$
$$= \frac{6}{15} \times 0 + \frac{9}{15} \times 0.918 = 0.551.$$

Similarly, we obtain $entropy_{Has\_job}(D) = 0.647$, and $entropy_{Credit\_rating}(D) = 0.608$. The gains for the attributes are:

$gain(D, \text{Age}) = 0.971 - 0.888 = 0.083$
$gain(D, \text{Own\_house}) = 0.971 - 0.551 = 0.420$
$gain(D, \text{Has\_job}) = 0.971 - 0.647 = 0.324$
$gain(D, \text{Credit\_rating}) = 0.971 - 0.608 = 0.363.$

Own_house is the best attribute for the root node. Figure 3.5(B) shows the root node using Own_house. Since the left branch has only one class (Yes) of data, it results in a leaf node (line 1 in Fig. 3.4). For Own_house = false, further extension is needed. The process is the same as above, but we only use the subset of the data with Own_house = false, i.e., D_2. ■

Information Gain Ratio

The gain criterion tends to favor attributes with many possible values. An extreme situation is that the data contain an *ID* attribute that is an identification of each example. If we consider using this *ID* attribute to partition the data, each training example will form a subset and has only one class, which results in $entropy_{ID}(D) = 0$. So the gain by using this attribute is maximal. From a prediction point of review, such a partition is useless.

Gain ratio (Equation 5) remedies this bias by normalizing the gain using the entropy of the data with respect to the values of the attribute. Our previous entropy computations are done with respect to the class attribute:

$$gainRatio(D, A_i) = \frac{gain(D, A_i)}{-\sum_{j=1}^{s}\left(\frac{|D_j|}{|D|} \times \log_2 \frac{|D_j|}{|D|}\right)} \quad (5)$$

where s is the number of possible values of A_i, and D_j is the subset of data

that has the jth value of A_i. $|D_j|/|D|$ corresponds to the probability of Equation (2). Using Equation (5), we simply choose the attribute with the highest gainRatio value to extend the tree.

This method works because if A_i has too many values the denominator will be large. For instance, in our above example of the *ID* attribute, the denominator will be $\log_2|D|$. The denominator is called the **split info** in C4.5. One note is that the split info can be 0 or very small. Some heuristic solutions can be devised to deal with it (see [453]).

3.2.3 Handling of Continuous Attributes

It seems that the decision tree algorithm can only handle discrete attributes. In fact, continuous attributes can be dealt with easily as well. In a real life data set, there are often both discrete attributes and continuous attributes. Handling both types in an algorithm is an important advantage.

To apply the decision tree building method, we can divide the value range of attribute A_i into intervals at a particular tree node. Each interval can then be considered a discrete value. Based on the intervals, gain or gainRatio is evaluated in the same way as in the discrete case. Clearly, we can divide A_i into any number of intervals at a tree node. However, two intervals are usually sufficient. This **binary split** is used in C4.5. We need to find a **threshold** value for the division.

Clearly, we should choose the threshold that maximizes the gain (or gainRatio). We need to examine all possible thresholds. This is not a problem because although for a continuous attribute A_i the number of possible values that it can take is infinite, the number of actual values that appear in the data is always finite. Let the set of distinctive values of attribute A_i that occur in the data be $\{v_1, v_2, \ldots, v_r\}$, which are sorted in ascending order. Clearly, any threshold value lying between v_i and v_{i+1} will have the same effect of dividing the training examples into those whose value of attribute A_i lies in $\{v_1, v_2, \ldots, v_i\}$ and those whose value lies in $\{v_{i+1}, v_{i+2}, \ldots, v_r\}$. There are thus only $r-1$ possible splits on A_i, which can all be evaluated.

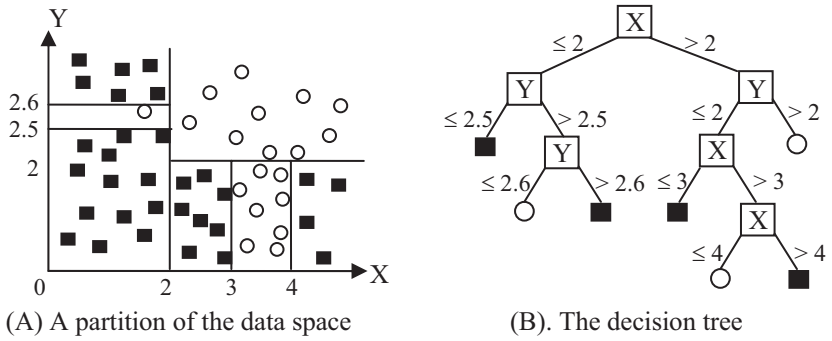
The threshold value can be the middle point between v_i and v_{i+1}, or just on the "right side" of value v_i, which results in two intervals $A_i \leq v_i$ and $A_i > v_i$. This latter approach is used in C4.5. The advantage of this approach is that the values appearing in the tree actually occur in the data. The threshold value that maximizes the gain (gainRatio) value is selected. We can modify the algorithm in Fig. 3.4 (lines 8–11) easily to accommodate this computation so that both discrete and continuous attributes are considered.

A change to line 20 of the algorithm in Fig. 3.4 is also needed. For a continuous attribute, we do not remove attribute A_g because an interval can

be further split recursively in subsequent tree extensions. Thus, the same continuous attribute may appear multiple times in a tree path (see Example 9), which does not happen for a discrete attribute.

From a geometric point of view, a decision tree built with only continuous attributes represents a partitioning of the data space. A series of splits from the root node to a leaf node represents a hyper-rectangle. Each side of the hyper-rectangle is an axis-parallel hyperplane.

Example 8: The hyper-rectangular regions in Fig. 3.6(A), which partitions the space, are produced by the decision tree in Fig. 3.6(B). There are two classes in the data, represented by empty circles and filled rectangles. ■



(A) A partition of the data space (B). The decision tree

Fig. 3.6. A partitioning of the data space and its corresponding decision tree

Handling of continuous (numeric) attributes has an impact on the efficiency of the decision tree algorithm. With only discrete attributes the algorithm grows linearly with the size of the data set D. However, sorting of a continuous attribute takes $|D|\log|D|$ time, which can dominate the tree learning process. Sorting is important as it ensures that gain or gainRatio can be computed in one pass of the data.

3.2.4 Some Other Issues

We now discuss several other issues in decision tree learning.

Tree Pruning and Overfitting: A decision tree algorithm recursively partitions the data until there is no impurity or there is no attribute left. This process may result in trees that are very deep and many tree leaves may cover very few training examples. If we use such a tree to predict the training set, the accuracy will be very high. However, when it is used to classify unseen test set, the accuracy may be very low. The learning is thus not effective, i.e., the decision tree does not **generalize** the data well. This

phenomenon is called **overfitting**. More specifically, we say that a classifier f_1 **overfits** the data if there is another classifier f_2 such that f_1 achieves a higher accuracy on the training data than f_2, but a lower accuracy on the unseen test data than f_2 [385].

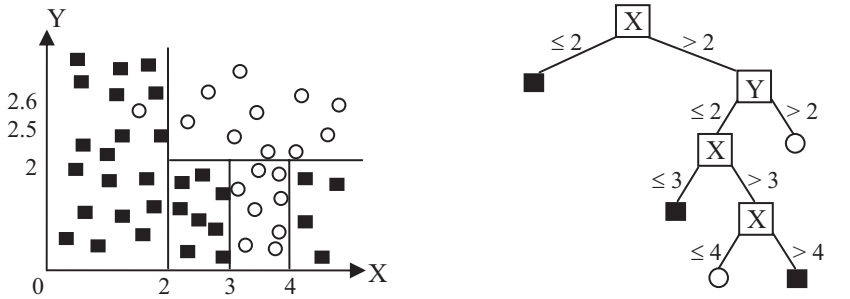
Overfitting is usually caused by noise in the data, i.e., wrong class values/labels and/or wrong values of attributes, but it may also be due to the complexity and randomness of the application domain. These problems cause the decision tree algorithm to refine the tree by extending it to very deep using many attributes.

To reduce overfitting in the context of decision tree learning, we perform pruning of the tree, i.e., to delete some branches or sub-trees and replace them with leaves of majority classes. There are two main methods to do this, **stopping early** in tree building (which is also called **pre-pruning**) and **pruning** the tree after it is built (which is called **post-pruning**). Post-pruning has been shown more effective. Early-stopping can be dangerous because it is not clear what will happen if the tree is extended further (without stopping). Post-pruning is more effective because after we have extended the tree to the fullest, it becomes clearer which branches/sub-trees may not be useful (overfit the data). The general idea of post-pruning is to estimate the error of each tree node. If the estimated error for a node is less than the estimated error of its extended sub-tree, then the sub-tree is pruned. Most existing tree learning algorithms take this approach. See [453] for a technique called the pessimistic error based pruning.

Example 9: In Fig. 3.6(B), the sub-tree representing the rectangular region

$$X \leq 2,\ Y > 2.5,\ Y \leq 2.6$$

in Fig. 3.6(A) is very likely to be overfitting. The region is very small and contains only a single data point, which may be an error (or noise) in the data collection. If it is pruned, we obtain Fig. 3.7(A) and (B). ■



(A) A partition of the data space (B). The decision tree

Fig. 3.7. The data space partition and the decision tree after pruning

Another common approach to pruning is to use a separate set of data called the **validation set**, which is not used in training and neither in testing. After a tree is built, it is used to classify the validation set. Then, we can find the errors at each node on the validation set. This enables us to know what to prune based on the errors at each node.

Rule Pruning: We noted earlier that a decision tree can be converted to a set of rules. In fact, C4.5 also prunes the rules to simplify them and to reduce overfitting. First, the tree (C4.5 uses the unpruned tree) is converted to a set of rules in the way discussed in Example 4. Rule pruning is then performed by removing some conditions to make the rules shorter and fewer (after pruning some rules may become redundant). In most cases, pruning results in a more accurate rule set as shorter rules are less likely to overfit the training data. Pruning is also called **generalization** as it makes rules more **general** (with fewer conditions). A rule with more conditions is more **specific** than a rule with fewer conditions.

Example 10: The sub-tree below $X \leq 2$ in Fig. 3.6(B) produces these rules:

Rule 1: $X \leq 2$, $Y > 2.5$, $Y > 2.6 \rightarrow$ ■
Rule 2: $X \leq 2$, $Y > 2.5$, $Y \leq 2.6 \rightarrow$ O
Rule 3: $X \leq 2$, $Y \leq 2.5 \rightarrow$ ■

Note that $Y > 2.5$ in Rule 1 is not useful because of $Y > 2.6$, and thus Rule 1 should be

Rule 1: $X \leq 2$, $Y > 2.6 \rightarrow$ ■

In pruning, we may be able to delete the conditions $Y > 2.6$ from Rule 1 to produce:

$X \leq 2 \rightarrow$ ■

Then Rule 2 and Rule 3 become redundant and can be removed. ■

A useful point to note is that after pruning the resulting set of rules may no longer be **mutually exclusive** and **exhaustive**. There may be data points that satisfy the conditions of more than one rule, and if inaccurate rules are discarded, of no rules. An ordering of the rules is thus needed to ensure that when classifying a test case only one rule will be applied to determine the class of the test case. To deal with the situation that a test case does not satisfy the conditions of any rule, a **default class** is used, which is usually the majority class.

Handling Missing Attribute Values: In many practical data sets, some attribute values are missing or not available due to various reasons. There are many ways to deal with the problem. For example, we can fill each

missing value with the special value "unknown" or the most frequent value of the attribute if the attribute is discrete. If the attribute is continuous, use the mean of the attribute for each missing value.

The decision tree algorithm in C4.5 takes another approach. At a tree node, distribute the training example with missing value for the attribute to each branch of the tree proportionally according to the distribution of the training examples that have values for the attribute.

Handling Skewed Class Distribution: In many applications, the proportions of data for different classes can be very different. For instance, in a data set of intrusion detection in computer networks, the proportion of intrusion cases is extremely small (< 1%) compared with normal cases. Directly applying the decision tree algorithm for classification or prediction of intrusions is usually not effective. The resulting decision tree often consists of a single leaf node "normal", which is useless for intrusion detection. One way to deal with the problem is to over sample the intrusion examples to increase its proportion. Another solution is to rank the new cases according to how likely they may be intrusions. The human users can then investigate the top ranked cases.

3.3 Classifier Evaluation

After a classifier is constructed, it needs to be evaluated for accuracy. Effective evaluation is crucial because without knowing the approximate accuracy of a classifier, it cannot be used in real-world tasks.

There are many ways to evaluate a classifier, and there are also many measures. The main measure is the classification **accuracy** (Equation 1), which is the number of correctly classified instances in the test set divided by the total number of instances in the test set. Some researchers also use the **error rate**, which is 1 – *accuracy*. Clearly, if we have several classifiers, the one with the highest accuracy is preferred. Statistical significance tests may be used to check whether one classifier's accuracy is significantly better than that of another given the same training and test data sets. Below, we first present several common methods for classifier evaluation, and then introduce some other evaluation measures.

3.3.1 Evaluation Methods

Holdout Set: The available data D is divided into two disjoint subsets, the **training set** D_{train} and the **test set** D_{test}, $D = D_{train} \cup D_{test}$ and $D_{train} \cap D_{test} =$

∅. The test set is also called the holdout set. This method is mainly used when the data set D is large. Note that the examples in the original data set D are all labeled with classes.

As we discussed earlier, the training set is used for learning a classifier while the test set is used for evaluating the resulting classifier. The training set should not be used to evaluate the classifier as the classifier is biased toward the training set. That is, the classifier may overfit the training set, which results in very high accuracy on the training set but low accuracy on the test set. Using the unseen test set gives an unbiased estimate of the classification accuracy. As for what percentage of the data should be used for training and what percentage for testing, it depends on the data set size. 50–50 and two thirds for training and one third for testing are commonly used.

To partition D into training and test sets, we can use a few approaches:

1. We randomly sample a set of training examples from D for learning and use the rest for testing.
2. If the data is collected over time, then we can use the earlier part of the data for training/learning and the later part of the data for testing. In many applications, this is a more suitable approach because when the classifier is used in the real-world the data are from the future. This approach thus better reflects the dynamic aspects of applications.

Multiple Random Sampling: When the available data set is small, using the above methods can be unreliable because the test set would be too small to be representative. One approach to deal with the problem is to perform the above random sampling n times. Each time a different training set and a different test set are produced. This produces n accuracies. The final estimated accuracy on the data is the average of the n accuracies.

Cross-Validation: When the data set is small, the ***n*-fold cross-validation** method is very commonly used. In this method, the available data is partitioned into n equal-size disjoint subsets. Each subset is then used as the test set and the remaining $n-1$ subsets are combined as the training set to learn a classifier. This procedure is then run n times, which gives n accuracies. The final estimated accuracy of learning from this data set is the average of the n accuracies. 10-fold and 5-fold cross-validations are often used.

A special case of cross-validation is the **leave-one-out cross-validation**. In this method, each fold of the cross validation has only a single test example and all the rest of the data is used in training. That is, if the original data has m examples, then this is m-fold cross-validation. This method is normally used when the available data is very small. It is not efficient for a large data set as m classifiers need to be built.

In Sect. 3.2.4, we mentioned that a validation set can be used to prune a decision tree or a set of rules. If **a validation set** is employed for that purpose, it should not be used in testing. In that case, the available data is divided into three subsets, a training set, a validation set and a test set. Apart from using a validation set to help tree or rule pruning, a validation set is also used frequently to estimate parameters in learning algorithms. In such cases, the values that give the best accuracy on the validation set are used as the final values of the parameters. Cross-validation can be used for parameter estimating as well. Then a separate validation set is not needed. Instead, the whole training set is used in cross-validation.

3.3.2 Precision, Recall, F-score and Breakeven Point

In some applications, we are only interested in one class. This is particularly true for text and Web applications. For example, we may be interested in only the documents or web pages of a particular topic. Also, in classification involving skewed or highly imbalanced data, e.g., network intrusion and financial fraud detection, we are typically interested in only the minority class. The class that the user is interested in is commonly called the **positive class**, and the rest **negative classes** (the negative classes may be combined into one negative class). Accuracy is not a suitable measure in such cases because we may achieve a very high accuracy, but may not identify a single intrusion. For instance, 99% of the cases are normal in an intrusion detection data set. Then a classifier can achieve 99% accuracy without doing anything by simply classifying every test case as "not intrusion". This is, however, useless.

Precision and **recall** are more suitable in such applications because they measure how precise and how complete the classification is on the positive class. It is convenient to introduce these measures using a **confusion matrix** (Table 3.2). A confusion matrix contains information about actual and predicted results given by a classifier.

Table 3.2. Confusion matrix of a classifier

	Classified positive	Classified negative
Actual positive	TP	FN
Actual negative	FP	TN

where

TP: the number of correct classifications of the positive examples (**true positive**)
FN: the number of incorrect classifications of positive examples (**false negative**)
FP: the number of incorrect classifications of negative examples (**false positive**)
TN: the number of correct classifications of negative examples (**true negative**)

Based on the confusion matrix, the precision (p) and recall (r) of the positive class are defined as follows:

$$p = \frac{TP}{TP + FP}. \qquad r = \frac{TP}{TP + FN}. \tag{6}$$

In words, precision p is the number of correctly classified positive examples divided by the total number of examples that are classified as positive. Recall r is the number of correctly classified positive examples divided by the total number of actual positive examples in the test set. The intuitive meanings of these two measures are quite obvious.

However, it is hard to compare classifiers based on two measures, which are not functionally related. For a test set, the precision may be very high but the recall can be very low, and vice versa.

Example 11: A test data set has 100 positive examples and 1000 negative examples. After classification using a classifier, we have the following confusion matrix (Table 3.3),

Table 3.3. Confusion matrix of a classifier

	Classified positive	Classified negative
Actual positive	1	99
Actual negative	0	1000

This confusion matrix gives the precision p = 100% and the recall r = 1% because we only classified one positive example correctly and classified no negative examples wrongly. ■

Although in theory precision and recall are not related, in practice high precision is achieved almost always at the expense of recall and high recall is achieved at the expense of precision. In an application, which measure is more important depends on the nature of the application. If we need a single measure to compare different classifiers, the **F-score** is often used:

$$F = \frac{2pr}{p + r} \tag{7}$$

The F-score (also called the **F_1-score**) is the harmonic mean of precision and recall.

$$F = \frac{2}{\frac{1}{p} + \frac{1}{r}} \tag{8}$$

The harmonic mean of two numbers tends to be closer to the smaller of the two. Thus, for the F-score to be high, both p and r must be high.

There is also another measure, called **precision and recall breakeven point,** which is used in the information retrieval community. The breakeven point is when the precision and the recall are equal. This measure assumes that the test cases can be ranked by the classifier based on their likelihoods of being positive. For instance, in decision tree classification, we can use the confidence of each leaf node as the value to rank test cases.

Example 12: We have the following ranking of 20 test documents. 1 represents the highest rank and 20 represents the lowest rank. "+" ("–") represents an actual positive (negative) documents.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
+	+	+	–	+	–	+	–	+	+	–	–	+	–	–	–	+	–	–	+

Assume that the test set has 10 positive examples.

At rank 1:	$p = 1/1 = 100\%$	$r = 1/10 = 10\%$
At rank 2:	$p = 2/2 = 100\%$	$r = 2/10 = 20\%$
…	…	…
At rank 9:	$p = 6/9 = 66.7\%$	$r = 6/10 = 60\%$
At rank 10:	$p = 7/10 = 70\%$	$r = 7/10 = 70\%$

The breakeven point is $p = r = 70\%$. Note that interpolation is needed if such a point cannot be found. ■

3.4 Rule Induction

In Sect. 3.2, we showed that a decision tree can be converted to a set of rules. Clearly, the set of rules can be used for classification as the tree. A natural question is whether it is possible to learn classification rules directly. The answer is yes. The process of learning such rules is called **rule induction** or **rule learning**. We study two approaches in the section.

3.4.1 Sequential Covering

Most rule induction systems use an algorithm called **sequential covering**. A classifier built with this algorithm consists of a list of rules, which is also called a **decision list** [463]. In the list, the ordering of the rules is significant.

The basic idea of sequential covering is to learn a list of rules sequentially, one at a time, to cover the training data. After each rule is learned,

the training examples covered by the rule are removed. Only the remaining data are used to find subsequent rules. Recall that a rule covers an example if the example satisfies the conditions of the rule. We study two specific algorithms based on this general strategy. The first algorithm is based on the CN2 system [104], and the second algorithm is based on the ideas in FOIL [452], I-REP [189], REP [70], and RIPPER [106] systems. Many ideas are also taken from [385].

Algorithm 1 (*Ordered Rules*)

This algorithm learns each rule without pre-fixing a class. That is, in each iteration, a rule of any class may be found. Thus rules of different classes may intermix in the final rule list. The ordering of rules is important.

This algorithm is given in Fig. 3.8. D is the training data. *RuleList* is the list of rules, which is initialized to empty set (line 1). *Rule* is the best rule found in each iteration. The function learn-one-rule-1() learns the *Rule* (lines 2 and 6). The stopping criteria for the while-loop can be of various kinds. Here we use $D = \emptyset$ or *Rule* is NULL (a rule is not learned). Once a rule is learned from the data, it is inserted into *RuleList* at the end (line 4). All the training examples that are covered by the rule are removed from the data (line 5). The remaining data is used to find the next rule and so on. After rule learning ends, a **default class** is inserted at the end of *RuleList.* This is because there may still be some training examples that are not covered by any rule as no good rule can be found from them, or because some test cases may not be covered by any rule and thus cannot be classified. The final list of rules is as follows:

$$<r_1, r_2, \ldots, r_k, \textit{default-class}> \tag{9}$$

where r_i is a rule.

Algorithm 2 (*Ordered Classes*)

This algorithm learns all rules for each class together. After rule learning for one class is completed, it moves to the next class. Thus all rules for each class appear together in the rule list. The sequence of rules for each class is unimportant, but the rule subsets for different classes are ordered. Typically, the algorithm finds rules for the least frequent class first, then the second least frequent class and so on. This ensures that some rules are learned for rare classes. Otherwise, they may be dominated by frequent classes and end up with no rules if considered after frequent classes.

The algorithm is given in Fig. 3.9. The data set D is split into two subsets, *Pos* and *Neg*, where *Pos* contains all the examples of class c from D,

```
Algorithm sequential-covering-1(D)
1   RuleList ← ∅;
2   Rule ← learn-one-rule-1(D);
3   while Rule is not NULL AND D ≠ ∅ do
4       RuleList ← insert Rule at the end of RuleList;
5       Remove from D the examples covered by Rule;
6       Rule ← learn-one-rule-1(D)
7   endwhile
8   insert a default class c at the end of RuleList, where c is the majority class
      in D;
9   return RuleList
```

Fig. 3.8. The first rule learning algorithm based on sequential covering

```
Algorithm sequential-covering-2(D, C)
1   RuleList ← ∅;                          // empty rule set at the beginning
2   for each class c ∈ C do
3       prepare data (Pos, Neg), where Pos contains all the examples of class
          c from D, and Neg contains the rest of the examples in D;
4       while Pos ≠ ∅ do
5           Rule ← learn-one-rule-2(Pos, Neg, c);
6           if Rule is NULL then
7             exit-while-loop
8           else RuleList ← insert Rule at the end of RuleList;
9                Remove examples covered by Rule from (Pos, Neg)
10          endif
11      endwhile
12  endfor
13  return RuleList
```

Fig. 3.9. The second rule learning algorithm based on sequential covering

and *Neg* the rest of the examples in *D* (line 3). *c* is the class that the algorithm is working on now. Two stopping conditions for rule learning of each class are in line 4 and line 6. The other parts of the algorithm are quite similar to those of the first algorithm in Fig. 3.8. Both learn-one-rule-1() and learn-one-rule-2() functions are described in Sect. 3.4.2.

Use of Rules for Classification

To use a list of rules for classification is straightforward. For a test case, we simply try each rule in the list sequentially. The class of the first rule that covers this test case is assigned as the class of the test case. Clearly, if no rule applies to the test case, the default class is used.

3.4.2 Rule Learning: Learn-One-Rule Function

We now present the function learn-one-rule(), which works as follows: It starts with an empty set of conditions. In the first iteration, one condition is added. In order to find the best condition to add, all possible conditions are tried, which form **candidate rules**. A **condition** is of the form A_i *op* v, where A_i is an attribute and v is a value of A_i. We also called it an **attribute-value** pair. For a discrete attribute, *op* is "=". For a continuous attribute, $op \in \{>, \leq\}$. The algorithm evaluates all the candidates to find the best one (the rest are discarded). After the first best condition is added, it tries to add the second condition and so on in the same fashion until some stopping condition is satisfied. Note that we omit the rule class here because it is implied, i.e., the majority class of the data covered by the conditions.

This is a heuristic and greedy algorithm in that after a condition is added, it will not be changed or removed through backtracking. Ideally, we would want to try all possible combinations of attributes and values. However, this is not practical as the number of possibilities grows exponentially. Hence, in practice, the above greedy algorithm is used. However, instead of keeping only the best set of conditions, we can improve the function a little by keeping k best sets of conditions ($k > 1$) in each iteration. This is called the **beam search** (k beams), which ensures that a larger space is explored. Below, we present two specific implementations of the algorithm, namely learn-one-rule-1() and learn-one-rule-2(). learn-one-rule-1() is used in the sequential-covering-1 algorithm, and learn-one-rule-2() is used in the sequential-covering-2 algorithm.

Learn-One-Rule-1

This function uses beam search (Fig. 3.10). The number of beams is k. *BestCond* stores the conditions of the rule to be returned. The class is omitted as it is the majority class of the data covered by *BestCond*. *candidateCondSet* stores the current best condition sets (which are the frontier beams) and its size is less than or equal to k. Each condition set contains a set of conditions connected by "and" (conjunction). *newCandidateCondSet* stores all the new candidate condition sets after adding each attribute-value pair (a possible condition) to every candidate in *candidateCondSet* (lines 5–11). Lines 13–17 update the *BestCond*. Specifically, an evaluation function is used to assess whether each new candidate condition set is better than the existing best condition set *BestCond* (line 14). If so, it replaces the current *BestCond* (line 15). Line 18 updates *candidateCondSet*, which selects k new best condition sets (new beams).

Once the final *BestCond* is found, it is evaluated to see if it is significantly better than without any condition (∅) using a *threshold* (line 20). If

```
Function learn-one-rule-1(D)
1   BestCond ← ∅;                          // rule with no condition.
2   candidateCondSet ← {BestCond};
3   attributeValuePairs ← the set of all attribute-value pairs in D of the form
      (A_i op v), where A_i is an attribute and v is a value or an interval;
4   while candidateCondSet ≠ ∅ do
5       newCandidateCondSet ← ∅;
6       for each candidate cond in candidateCondSet do
7           for each attribute-value pair a in attributeValuePairs do
8               newCond ← cond ∪ {a};
9               newCandidateCondSet ← newCandidateCondSet ∪ {newCond}
10          endfor
11      endfor
12      remove duplicates and inconsistencies, e.g., {A_i = v_1, A_i = v_2};
13      for each candidate newCond in newCandidateCondSet do
14          if evaluation(newCond, D) > evaluation(BestCond, D) then
15              BestCond ← newCond
16          endif
17      endfor
18      candidateCondSet ← the k best members of newCandidateCondSet
            according to the results of the evaluation function;
19  endwhile
20  if evaluation(BestCond, D) – evaluation(∅, D) > threshold then
21     return the rule: "BestCond → c" where is c the majority class of the data
           covered by BestCond
22  else return NULL
23  endif
```

Fig. 3.10. The learn-one-rule-1 function

```
Function evaluation(BestCond, D)
1   D′ ← the subset of training examples in D covered by BestCond;
2   entropy (D') = −∑_{j=1}^{|C|} Pr(c_j) log_2 Pr(c_j);
3   return – entropy(D′)                   // since entropy measures impurity.
```

Fig. 3.11. The entropy based evaluation function

yes, a rule will be formed using *BestCond* and the most frequent (or the majority) class of the data covered by *BestCond* (line 21). If not, NULL is returned to indicate that no significant rule is found.

The evaluation() function (Fig. 3.11) uses the entropy function as in the decision tree learning. Other evaluation functions are possible too. Note that when $BestCond = \emptyset$, it covers every example in D, i.e., $D = D'$.

```
   Function learn-one-rule-2(Pos, Neg, class)
1    split (Pos, Neg) into (GrowPos, GrowNeg) and (PrunePos, PruneNeg)
2    BestRule ← GrowRule(GrowPos, GrowNeg, class)        // grow a new rule
3    BestRule ← PruneRule(BestRule, PrunePos, PruneNeg)  // prune the rule
4    if the error rate of BestRule on (PrunePos, PruneNeg) exceeds 50% then
5       return NULL
6    endif
7    return BestRule
```

Fig. 3.12. The learn-one-rule-2() function

Learn-One-Rule-2

In the learn-one-rule-2() function (Fig. 3.12), a rule is first generated and then it is pruned. This method starts by splitting the positive and negative training data *Pos* and *Neg*, into growing and pruning sets. The growing sets, *GrowPos* and *GrowNeg*, are used to generate a rule, called *BestRule*. The pruning sets, *PrunePos* and *PruneNeg* are used to prune the rule because *BestRule* may overfit the data. Note that *PrunePos* and *PruneNeg* are actually validation sets discussed in Sects. 3.2.4 and 3.3.1.

growRule() function: growRule() generates a rule (called *BestRule*) by repeatedly adding a condition to its condition set that maximizes an evaluation function until the rule covers only some positive examples in *GrowPos* but no negative examples in *GrowNeg*. This is basically the same as lines 4–17 in Fig. 3.10, but without beam search (i.e., only the best rule is kept in each iteration). Let the current partially developed rule be R:

$$R: \quad av_1, .., av_k \rightarrow class$$

where each av_j is a condition (an attribute-value pair). By adding a new condition av_{k+1}, we obtain the rule R^+: $av_1, .., av_k, av_{k+1} \rightarrow class$. The evaluation function for R^+ is the following **information gain** criterion (which is different from the gain function used in decision tree learning):

$$gain(R, R^+) = p_1 \times \left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0} \right) \tag{10}$$

where p_0 (respectively, n_0) is the number of positive (negative) examples covered by R in *Pos* (*Neg*), and p_1 (n_1) is the number of positive (negative) examples covered by R^+ in *Pos* (*Neg*). The GrowRule() function simply returns the rule R^+ that maximizes the gain.

PruneRule() function: To prune a rule, we consider deleting every subset of conditions from the *BestRule*, and choose the deletion that maximizes:

$$v(BestRule, PrunePos, PruneNeg) = \frac{p-n}{p+n}, \tag{11}$$

where p (respectively n) is the number of examples in *PrunePo*s (*Prune-Neg*) covered by the current rule (after a deletion).

3.4.3 Discussion

Separate-and-Conquer vs. Divide-and-Conquer: Decision tree learning is said to use the *divide-and-conquer* strategy. At each step, all attributes are evaluated and one is selected to partition/divide the data into m disjoint subsets, where m is the number of values of the attribute. Rule induction discussed in this section is said to use the *separate-and-conquer* strategy, which evaluates all attribute-value pairs (conditions) (which are much larger in number than the number of attributes) and selects only one. Thus, each step of divide-and-conquer expands m rules, while each step of separate-and-conquer expands only one rule. Due to both effects, the separate-and-conquer strategy is much slower than the divide-and-conquer strategy.
Rule Understandability: If-then rules are easy to understand by human users. However, a word of caution about rules generated by sequential covering is in order. Such rules can be misleading because the covered data are removed after each rule is generated. Thus the rules in the rule list are not independent of each other. A rule r may be of high quality in the context of the data D' from which r was generated. However, it may be a weak rule with a very low accuracy (confidence) in the context of the whole data set D ($D' \subseteq D$) because many training examples that can be covered by r have already been removed by rules generated before r. If you want to understand the rules and possibly use them in some real-world tasks, you should be aware of this fact.

3.5 Classification Based on Associations

In Sect. 3.2, we showed that a decision tree can be converted to a set of rules, and in Sect. 3.4, we saw that a set of rules may also be found directly for classification. It is thus only natural to expect that association rules, in particular **class association rules** (**CAR**), may be used for classification too. Yes, indeed! In fact, normal association rules can be employed for classification as well as we will see in Sect. 3.5.3. CBA, which stands for *C*lassification *B*ased on *A*ssociations, is the first reported system that uses

association rules for classification [343]. In this section, we describe three approaches to employing association rules for classification:

1. Using class association rules for classification directly.
2. Using class association rules as features or attributes.
3. Using normal (or classic) association rules for classification.

The first two approaches can be applied to tabular data or transactional data. The last approach is usually employed for transactional data only. All these methods are useful in the Web environment as many types of Web data are in the form of transactions, e.g., search queries issued by users, and Web pages clicked by visitors. Transactional data sets are difficult to handle by traditional classification techniques, but are very natural for association rules. Below, we describe the three approaches in turn. We should note that various sequential rules can be used for classification in similar ways as well if sequential data sets are involved.

3.5.1 Classification Using Class Association Rules

Recall that a class association rule (CAR) is an association rule with only a class label on the right-hand side of the rule as its consequent (Sect. 2.5). For instance, from the data in Table 3.1, the following rule can be found:

$$\text{Own\_house} = \text{false}, \text{Has\_job} = \text{true} \rightarrow \text{Class} = \text{Yes} \quad [\text{sup}=3/15, \text{conf}=3/3],$$

which was also a rule from the decision tree in Fig. 3.3. In fact, there is no difference between rules from a decision tree (or a rule induction system) and CARs if we consider only categorical (or discrete) attributes (more on this later). The differences are in the mining processes and the final rule sets. CAR mining finds all rules in data that satisfy the user-specified minimum support (minsup) and minimum confidence (minconf) constraints. A decision tree or a rule induction system finds only a **subset** of the rules (expressed as a tree or a list of rules) for classification.

Example 13: Recall that the decision tree in Fig. 3.3 gives the following three rules:

$$\begin{aligned}
&\text{Own_house} = \text{true} \rightarrow \text{Class} = \text{Yes} && [\text{sup}=6/15, \text{conf}=6/6]\\
&\text{Own_house} = \text{false}, \text{Has_job} = \text{true} \rightarrow \text{Class} = \text{Yes} && [\text{sup}=3/15, \text{conf}=3/3]\\
&\text{Own_house} = \text{false}, \text{Has_job} = \text{false} \rightarrow \text{Class} = \text{No} && [\text{sup}=6/15, \text{conf}=6/6].
\end{aligned}$$

However, there are many other rules that exist in data, e.g.,

$$\begin{aligned}
&\text{Age} = \text{young}, \text{Has_job} = \text{true} \rightarrow \text{Class} = \text{Yes} && [\text{sup}=2/15, \text{conf}=2/2]\\
&\text{Age} = \text{young}, \text{Has_job} = \text{false} \rightarrow \text{Class} = \text{No} && [\text{sup}=3/15, \text{conf}=3/3]\\
&\text{Credit_rating} = \text{fair} \rightarrow \text{Class} = \text{No} && [\text{sup}=4/15, \text{conf}=4/5]
\end{aligned}$$

and many more, if we use minsup = 2/15 = 13.3% and minconf = 70%. ■

In many cases, rules that are not in the decision tree (or a rule list) may be able to perform classification more accurately. Empirical comparisons reported by several researchers show that classification using CARs can perform more accurately on many data sets than decision trees and rule induction systems (see Bibliographic Notes for references).

The complete set of rules from CAR mining is also beneficial from a rule usage point of view. In some applications, the user wants to act on some interesting rules. For example, in an application for finding causes of product problems, more rules are preferred to fewer rules because. With more rules, the user is more likely to find rules that indicate causes of the problems. Such rules may not be generated by a decision tree or a rule induction system. A deployed data mining system based on CARs is reported in [352]. We should, however, also bear in mind of the following:

1. Decision tree learning and rule induction do not use the minsup or minconf constraint. Thus, some rules that they find can have very low supports, which, of course, are likely to be pruned because the chance that they overfit the training data is high. Although we can set a low minsup for CAR mining, it may cause combinatorial explosion. In practice, in addition to minsup and minconf, a limit on the total number of rules to be generated may be used to further control the CAR generation process. When the number of generated rules reaches the limit, the algorithm stops. However, with this limit, we may not be able to generate long rules (with many conditions). Recall that the Apriori algorithm works in a level-wise fashion, i.e., short rules are generated before long rules. In some applications, this might not be an issue as short rules are often preferred and are sufficient for classification or for action. Long rules normally have very low supports and tend to overfit the data. However, in some other applications, long rules can be useful.
2. CAR mining does not use continuous (numeric) attributes, while decision trees deal with continuous attributes naturally. Rule induction can use continuous attributes as well. There is still no satisfactory method to deal with such attributes directly in association rule mining. Fortunately, many attribute discretization algorithms exist that can automatically discretize the value range of a continuous attribute into suitable intervals [e.g., 151, 172], which are then considered as discrete values.

Mining Class Association Rules for Classification

There are many techniques that use CARs to build classifiers. Before describing them, let us first discuss some issues related to CAR mining for

classification. Since a CAR mining algorithm has been discussed in Sect. 2.5, we will not repeat it here.

Rule Pruning: CAR rules are highly redundant, and many of them are not statistically significant (which can cause overfitting). Rule pruning is thus needed. The idea of pruning CARs is basically the same as that in decision tree building or rule induction. Thus, we will not discuss it further (see [343, 328] for some of the pruning methods).

Multiple Minimum Class Supports: As discussed in Sect. 2.5.3, a single minsup is inadequate for mining CARs because many practical classification data sets have uneven class distributions, i.e., some classes cover a large proportion of the data, while others cover only a very small proportion (which are called **rare** or **infrequent classes**).

Example 14: Suppose we have a dataset with two classes, *Y* and *N*. 99% of the data belong to the *Y* class, and only 1% of the data belong to the *N* class. If we set minsup = 1.5%, we will not find any rule for class *N*. To solve the problem, we need to lower down the minsup. Suppose we set minsup = 0.2%. Then, we may find a huge number of overfitting rules for class *Y* because minsup = 0.2% is too low for class *Y*. ■

Multiple minimum class supports can be applied to deal with the problem. We can assign a different **minimum class support** $minsup_i$ for each class c_i, i.e., all the rules of class c_i must satisfy $minsup_i$. Alternatively, we can provide one single total minsup, denoted by *t_minsup*, which is then distributed to each class according to the class distribution:

$$minsup_i = t\_minsup \times sup(c_i) \tag{12}$$

where $sup(c_i)$ is the support of class c_i in training data. The formula gives frequent classes higher minsups and infrequent classes lower minsups.

Parameter Selection: The parameters used in CAR mining are the minimum supports and the minimum confidences. Note that a different minimum confidence may also be used for each class. However, minimum confidences do not affect the classification much because classifiers tend to use high confidence rules. One minimum confidence is sufficient as long as it is not set too high. To determine the best $minsup_i$ for each class c_i, we can try a range of values to build classifiers and then use a validation set to select the final value. Cross-validation may be used as well.

Data Formats: The algorithm for CAR mining given in Sect. 2.5.2 is for mining transaction data sets. However, many classification data sets are in the table format. As we discussed in Sect. 2.3, a tabular data set can be easily converted to a transaction data set.

Classifier Building

After all CAR rules are found, a classifier is built using the rules. There are many existing methods, which can be grouped into three categories.

Use the Strongest Rule: This is perhaps the simplest strategy. It simply uses CARs directly for classification. For each test instance, it finds the strongest rule that covers the instance. Recall that a rule **covers** an instance if the instance satisfies the conditions of the rule. The class of the strongest rule is then assigned as the class of the test instance. The strength of a rule can be measured in various ways, e.g., based on confidence, χ^2 test, or a combination of both support and confidence values.

Select a Subset of the Rules to Build a Classifier: The representative method of this category is the one used in the CBA system. The method is similar to the sequential covering method, but applied to class association rules with additional enhancements as discussed above.

Let the set of all discovered CARs be S. Let the training data set be D. The basic idea is to select a subset L ($\subseteq S$) of high confidence rules to cover the training data D. The set of selected rules, including a default class, is then used as the classifier. The selection of rules is based on a total order defined on the rules in S.

Definition: Given two rules, r_i and r_j, $r_i \succ r_j$ (also called r_i precedes r_j or r_i has a higher precedence than r_j) if

1. the confidence of r_i is greater than that of r_j, or
2. their confidences are the same, but the support of r_i is greater than that of r_j, or
3. both the confidences and supports of r_i and r_j are the same, but r_i is generated earlier than r_j.

A CBA classifier L is of the form:

$$L = <r_1, r_2, \ldots, r_k, \textit{default-class}>$$

where $r_i \in S$, $r_a \succ r_b$ if $b > a$. In classifying a test case, the first rule that satisfies the case classifies it. If no rule applies to the case, it takes the default class (*default-class*). A simplified version of the algorithm for building such a classifier is given in Fig. 3.13. The classifier is the *RuleList*.

This algorithm can be easily implemented by making one pass through the training data for every rule. However, this is extremely inefficient for large data sets. An efficient algorithm that makes at most two passes over the data is given in [343].

Combine Multiple Rules: Like the first approach, this approach does not take any additional step to build a classifier. At the classification time, for

```
Algorithm CBA(S, D)
1  S = sort(S);                    // sorting is done according to the precedence ≻
2  RuleList = ∅;                   // the rule list classifier
3  for each rule r ∈ S in sequence do
4      if D ≠ ∅ AND r classifies at least one example in D correctly then
5          delete from D all training examples covered by r;
6          add r at the end of RuleList
7      endif
8  endfor
9  add the majority class as the default class at the end of RuleList
```

Fig. 3.13. A simple classifier building algorithm

each test instance, the system first finds the subset of rules that covers the instance. If all the rules in the subset have the same class, the class is assigned to the test instance. If the rules have different classes, the system divides the rules into groups according to their classes, i.e., all rules of the same class are in the same group. The system then compares the aggregated effects of the rule groups and finds the strongest group. The class label of the strongest group is assigned to the test instance. To measure the strength of each rule group, there again can be many possible techniques. For example, the CMAR system uses a weighted χ^2 measure [328].

3.5.2 Class Association Rules as Features

In the above two methods, rules are directly used for classification. In this method, rules are used as features to augment the original data or simply form a new data set, which is then fed to a traditional classification algorithm, e.g., decision trees or the naïve Bayesian method.

To use CARs as features, only the conditional part of each rule is needed, and it is often treated as a Boolean feature/attribute. If a data instance in the original data contains the conditional part, the value of the feature/attribute is set to 1, and otherwise it is set to 0. Several applications of this method have been reported [23, 131, 255, 314]. The reason that this approach is helpful is that CARs capture multi-attribute or multi-item correlations with class labels. Many classification algorithms do not find such correlations (e.g., naïve Bayesian), but they can be quite useful.

3.5.3 Classification Using Normal Association Rules

Not only can class association rules be used for classification, but also normal association rules. For example, association rules are commonly

used in e-commerce Web sites for product recommendations, which work as follows: When a customer purchases some products, the system will recommend him/her some other related products based on what he/she has already purchased (see Chap. 12).

Recommendation is essentially a classification or prediction problem. It predicts what a customer is likely to buy. Association rules are naturally applicable to such applications. The classification process is the following:

1. The system first uses previous purchase transactions (the same as market basket transactions) to mine association rules. In this case, there are no fixed classes. Any item can appear on the left-hand side or the right-hand side of a rule. For recommendation purposes, usually only one item appears on the right-hand side of a rule.
2. At the prediction (e.g., recommendation) time, given a transaction (e.g., a set of items already purchased by a customer), all the rules that cover the transaction are selected. The strongest rule is chosen and the item on the right-hand side of the rule (i.e., the consequent) is then the predicted item and recommended to the user. If multiple rules are very strong, multiple items can be recommended.

This method is basically the same as the "**use the strongest rule**" method described in Sect. 3.5.1. Again, the rule strength can be measured in various ways, e.g., confidence, χ^2 test, or a combination of both support and confidence. For example, in [337], the product of support and confidence is used as the rule strength. Clearly, the other two methods discussed in Sect. 3.5.1 can be applied as well.

The key advantage of using association rules for recommendation is that they can predict any item since any item can be the class item on the right-hand side. Traditional classification algorithms only work with a single fixed class attribute, and are not easily applicable to recommendations.

Finally, we note that multiple minimum supports (Sect. 2.4) can be of significant help. Otherwise, **rare items** will never be recommended, which causes the **coverage** problem (see Sect. 12.3.3). It is shown in [389] that using multiple minimum supports can dramatically increase the coverage.

3.6 Naïve Bayesian Classification

Supervised learning can be naturally studied from a probabilistic point of view. The task of classification can be regarded as estimating the class **posterior** probabilities given a test example d, i.e.,

$$\Pr(C = c_j \mid d). \tag{13}$$

We then see which class c_j is more probable. The class with the highest probability is assigned to the example d.

Formally, let A_1, A_2, …, $A_{|A|}$ be the set of attributes with discrete values in the data set D. Let C be the class attribute with $|C|$ values, c_1, c_2, …, $c_{|C|}$. Given a test example d with observed attribute values a_1 through $a_{|A|}$, where a_i is a possible value of A_i (or a member of the domain of A_i), i.e.,

$$d = <A_1=a_1, ..., A_{|A|}=a_{|A|}>.$$

The prediction is the class c_j such that $\Pr(C=c_j \mid A_1=a_1, ..., A_{|A|}=a_{|A|})$ is maximal. c_j is called a **maximum *a posteriori*** (MAP) hypothesis.

By Bayes' rule, the above quantity (13) can be expressed as

$$\begin{aligned} &\Pr(C = c_j \mid A_1 = a_1,..., A_{|A|} = a_{|A|}) \\ &= \frac{\Pr(A_1 = a_1,..., A_{|A|} = a_{|A|} \mid C = c_j)\Pr(C = c_j)}{\Pr(A_1 = a_1,..., A_{|A|} = a_{|A|})} \\ &= \frac{\Pr(A_1 = a_1,..., A_{|A|} = a_{|A|} \mid C = c_j)\Pr(C = c_j)}{\sum_{k=1}^{|C|} \Pr(A_1 = a_1,..., A_{|A|} = a_{|A|} \mid C = c_k)\Pr(C = c_k)}. \end{aligned} \tag{14}$$

$\Pr(C=c_j)$ is the class **prior** probability of c_j, which can be estimated from the training data. It is simply the fraction of the data in D with class c_j.

If we are only interested in making a classification, $\Pr(A_1=a_1, ..., A_{|A|}=a_{|A|})$ is irrelevant for decision making because it is the same for every class. Thus, only $\Pr(A_1=a_1 \wedge ... \wedge A_{|A|}=a_{|A|} \mid C=c_j)$ needs to be computed, which can be written as

$$\begin{aligned} &\Pr(A_1=a_1, ..., A_{|A|}=a_{|A|} \mid C=c_j) \\ &= \Pr(A_1=a_1 \mid A_2=a_2, ..., A_{|A|}=a_{|A|}, C=c_j) \times \Pr(A_2=a_2, ..., A_{|A|}=a_{|A|} \mid C=c_j). \end{aligned} \tag{15}$$

Recursively, the second term above (i.e., $\Pr(A_2=a_2, ..., A_{|A|}=a_{|A|}|C=c_j)$) can be written in the same way (i.e., $\Pr(A_2=a_2|A_3=a_3 ..., A_{|A|}=a_{|A|}, C=c_j) \times \Pr(A_3=a_3, ..., A_{|A|}=a_{|A|}|C=c_j)$), and so on. However, to further our derivation, we need to make an important assumption.

Conditional independence assumption: We assume that all attributes are conditionally independent given the class $C = c_j$. Formally, we assume,

$$\Pr(A_1=a_1 \mid A_2=a_2, ..., A_{|A|}=a_{|A|}, C=c_j) = \Pr(A_1=a_1 \mid C=c_j) \tag{16}$$

and similarly for A_2 through $A_{|A|}$. We then obtain

$$\Pr(A_1 = a_1,..., A_{|A|} = a_{|A|} \mid C = c_j) = \prod_{i=1}^{|A|} \Pr(A_i = a_i \mid C = c_j) \tag{17}$$

$$\Pr(C = c_j \mid A_1 = a_1,..., A_{|A|} = a_{|A|}) = \frac{\Pr(C = c_j)\prod_{i=1}^{|A|} \Pr(A_i = a_i \mid C = c_j)}{\sum_{k=1}^{|C|} \Pr(C = c_k)\prod_{i=1}^{|A|} \Pr(A_i = a_i \mid C = c_k)}. \quad (18)$$

Next, we need to estimate the *prior* probabilities $\Pr(C=c_j)$ and the conditional probabilities $\Pr(A_i=a_i \mid C=c_j)$ from the training data, which are straightforward.

$$\Pr(C = c_j) = \frac{\text{number of examples of class } c_j}{\text{total number of examples in the data set}} \quad (19)$$

$$\Pr(A_i = a_i \mid C = c_j) = \frac{\text{number of examples with } A_i = a_i \text{ and class } c_j}{\text{number of examples of class } c_j}. \quad (20)$$

If we only need a decision on the most probable class for each test instance, we only need the numerator of Equation (18) since the denominator is the same for every class. Thus, given a test case, we compute the following to decide the most probable class for the test case:

$$c = \arg\max_{c_j} \Pr(C = c_j)\prod_{i=1}^{|A|} \Pr(A_i = a_i \mid C = c_j) \quad (21)$$

Example 15: Suppose that we have the training data set in Fig. 3.14, which has two attributes *A* and *B*, and the class *C*. We can compute all the probability values required to learn a naïve Bayesian classifier.

A	B	C
m	b	t
m	s	t
g	q	t
h	s	t
g	q	t
g	q	f
g	s	f
h	b	f
h	q	f
m	b	f

Fig. 3.14. An example of a training data set

$\Pr(C = t) = 1/2$,	$\Pr(C = f) = 1/2$	
$\Pr(A=m \mid C=t) = 2/5$	$\Pr(A=g \mid C=t) = 2/5$	$\Pr(A=h \mid C=t) = 1/5$
$\Pr(A=m \mid C=f) = 1/5$	$\Pr(A=g \mid C=f) = 2/5$	$\Pr(A=h \mid C=f) = 2/5$
$\Pr(B=b \mid C=t) = 1/5$	$\Pr(B=s \mid C=t) = 2/5$	$\Pr(B=q \mid C=t) = 2/5$
$\Pr(B=b \mid C=f) = 2/5$	$\Pr(B=s \mid C=f) = 1/5$	$\Pr(B=q \mid C=f) = 2/5$

Now we have a test example:

$$A = m \quad B = q \quad C = ?$$

We want to know its class. Equation (21) is applied. For $C = t$, we have

$$\Pr(C = t)\prod_{j=1}^{2}\Pr(A_j = a_j \mid C = t) = \frac{1}{2}\times\frac{2}{5}\times\frac{2}{5} = \frac{2}{25}.$$

For class $C = f$, we have

$$\Pr(C = f)\prod_{j=1}^{2}\Pr(A_j = a_j \mid C = f) = \frac{1}{2}\times\frac{1}{5}\times\frac{2}{5} = \frac{1}{25}.$$

Since $C = t$ is more probable, t is the predicted class of the test case. ■

It is easy to see that the probabilities (i.e., $\Pr(C=c_j)$ and $\Pr(A_i=a_i \mid C=c_j)$) required to build a naïve Bayesian classifier can be found in one scan of the data. Thus, the algorithm is linear in the number of training examples, which is one of the great strengths of the naïve Bayes, i.e., it is extremely efficient. In terms of classification accuracy, although the algorithm makes the strong assumption of conditional independence, several researchers have shown that its classification accuracies are surprisingly strong. See experimental comparisons of various techniques in [148, 285, 349].

To learn practical naïve Bayesian classifiers, we still need to address some additional issues: how to handle numeric attributes, zero counts, and missing values. Below, we deal with each of them in turn.

Numeric Attributes: The above formulation of the naïve Bayesian learning assumes that all attributes are categorical. However, most real-life data sets have numeric attributes. Therefore, in order to use the naïve Bayeisan algorithm, each numeric attribute needs to be discretized into intervals. This is the same as for class association rule mining. Existing discretization algorithms in [e.g., 151, 172] can be used.

Zero Counts: It is possible that a particular attribute value in the test set never occurs together with a class in the training set. This is problematic because it will result in a 0 probability, which wipes out all the other probabilities $\Pr(A_i=a_i \mid C=c_j)$ when they are multiplied according to Equation

(21) or Equation (18). A principled solution to this problem is to incorporate a small-sample correction into all probabilities.

Let n_{ij} be the number of examples that have both $A_i = a_i$ and $C = c_j$. Let n_j be the total number of examples with $C=c_j$ in the training data set. The uncorrected estimate of $\Pr(A_i=a_i \mid C=c_j)$ is n_{ij}/n_j, and the corrected estimate is

$$\Pr(A_i = a_i \mid C = c_j) = \frac{n_{ij} + \lambda}{n_j + \lambda m_i} \tag{22}$$

where m_i is the number of values of attribute A_i (e.g., 2 for a Boolean attribute), and λ is a multiplicative factor, which is commonly set to $\lambda = 1/n$, where n is the total number of examples in the training set D [148, 285]. When $\lambda = 1$, we get the well known **Laplace's law of succession** [204]. The general form of correction (also called **smoothing**) in Equation (22) is called the **Lidstone's law of succession** [330]. Applying the correction $\lambda = 1/n$, the probabilities of Example 15 are revised. For example,

$\Pr(A=m \mid C=t) = (2+1/10) / (5 + 3*1/10) = 2.1/5.3 = 0.396$
$\Pr(B=b \mid C=t) = (1+1/10) / (5 + 3*1/10) = 1.1/5.3 = 0.208.$

Missing Values: Missing values are ignored, both in computing the probability estimates in training and in classifying test instances.

3.7 Naïve Bayesian Text Classification

Text classification or categorization is the problem of learning classification models from training documents labeled with pre-defined classes. That learned models are then used to classify future documents. For example, we have a set of news articles of three classes or topics, Sport, Politics, and Science. We want to learn a classifier that is able to classify future news articles into these classes.

Due to the rapid growth of online documents in organizations and on the Web, automated document classification is an important problem. Although the techniques discussed in the previous sections can be applied to text classification, it has been shown that they are not as effective as the methods presented in this section and in the next two sections. In this section, we study a naïve Bayesian learning method that is specifically formulated for texts, which makes use of text specific characteristics. However, the ideas are similar to those in Sect. 3.6. Below, we first present a probabilistic framework for texts, and then study the naïve Bayesian equations for their classification. There are several slight variations of this model. This section is mainly based on the formulation given in [365].

3.7.1 Probabilistic Framework

The naïve Bayesian learning method for text classification is derived based on a probabilistic **generative model**. It assumes that each document is generated by a **parametric distribution** governed by a set of **hidden parameters**. Training data is used to estimate these parameters. The parameters are then applied to classify each test document using Bayes' rule by calculating the **posterior probability** that the distribution associated with a class (represented by the unobserved class variable) would have generated the given document. Classification then becomes a simple matter of selecting the most probable class.

The generative model is based on two assumptions:

1. The data (or the text documents) are generated by a mixture model.
2. There is one-to-one correspondence between mixture components and document classes.

A **mixture model** models the data with a number of statistical distributions. Intuitively, each distribution corresponds to a data cluster and the parameters of the distribution provide a description of the corresponding cluster. Each distribution in a mixture model is also called a **mixture component** (the distribution can be of any kind). Figure 3.15 plots two **probability density functions** of a mixture of two Gaussian distributions that generate a 1-dimensional data set of two classes, one distribution per class, whose parameters (denoted by θ_i) are the mean (μ_i) and the standard deviation (σ_i), i.e., $\theta_i = (\mu_i, \sigma_i)$.

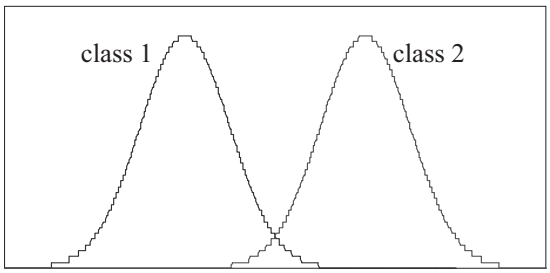


Fig. 3.15. Probability density functions of the two distributions in the mixture model

Let the number of mixture components (or distributions) in a mixture model be K, and the jth distribution have the parameters θ_j. Let Θ be the set of parameters of all components, $\Theta = \{\varphi_1, \varphi_2, \ldots, \varphi_K, \theta_1, \theta_2, \ldots, \theta_K\}$, where φ_j is the **mixture weight** (or **mixture probability**) of the mixture component j and θ_j is the set of parameters of component j. The mixture

weights are subject to the constraint $\sum_{j=1}^{K} \varphi_j = 1$. The meaning of mixture weights (or probabilities) will be clear below.

Let us see how the mixture model generates a collection of documents. Recall the classes C in our classification problem are c_1, c_2, …, $c_{|C|}$. Since we assume that there is one-to-one correspondence between mixture components and classes, each class corresponds to a mixture component. Thus $|C| = K$, and the jth mixture component can be represented by its corresponding class c_j and is parameterized by θ_j. The mixture weights are **class prior probabilities**, i.e., $\varphi_j = \Pr(c_j|\Theta)$. The mixture model generates each document d_i by:

1. first selecting a mixture component (or class) according to class prior probabilities (i.e., mixture weights), $\varphi_j = \Pr(c_j|\Theta)$;
2. then having this selected mixture component (c_j) generate a document d_i according to its parameters, with distribution $\Pr(d_i|c_j; \Theta)$ or more precisely $\Pr(d_i|c_j; \theta_j)$.

The probability that a document d_i is generated by the mixture model can be written as the sum of total probability over all mixture components. Note that to simplify the notation, we use c_j instead of $C = c_j$ as in the previous section:

$$\Pr(d_i \mid \Theta) = \sum_{j=1}^{|C|} \Pr(c_j \mid \Theta) \Pr(d_i \mid c_j; \Theta). \tag{23}$$

Since each document is attached with its class label, we can now derive the naïve Bayesian model for text classification. Note that in the above probability expressions, we include Θ to represent their dependency on Θ as we employ a generative model. In an actual implementation, we need not be concerned with Θ, i.e., it can be ignored.

3.7.2 Naïve Bayesian Model

A text document consists of a sequence of sentences, and each sentence consists of a sequence of words. However, due to the complexity of modeling word sequence and their relationships, several assumptions are made in the derivation of the Bayesian classifier. That is also why we call the final classification model, *naïve Bayesian* classification.

Specifically, the naïve Bayesian classification treats each document as a "bag" of words. The generative model makes the following assumptions:

1. Words of a document are generated independently of the context, that is, independently of the other words in the same document given the class label. This is the familiar naïve Bayesian assumption used before.
2. The probability of a word is independent of its position in the document. For example, the probability of seeing the word "student" in the first position of the document is the same as seeing it in any other position. The document length is chosen independent of its class.

With these assumptions, each document can be regarded as generated by a **multinomial distribution**. In other words, each document is drawn from a multinomial distribution of words with as many independent trials as the length of the document. The words are from a given vocabulary $V = \{w_1, w_2, \ldots, w_{|V|}\}$, $|V|$ being the number of words in the vocabulary. To see why this is a multinomial distribution, we give a short introduction to the multinomial distribution.

A **multinomial trial** is a process that can result in any of k outcomes, where $k \geq 2$. Each outcome of a multinomial trial has a probability of occurrence. The probabilities of the k outcomes are denoted by $p_1, p_2, \ldots, p_k$. For example, the rolling of a die is a multinomial trial, with six possible outcomes 1, 2, 3, 4, 5, 6. For a fair die, $p_1 = p_2 = \ldots = p_k = 1/6$.

Now assume n independent trials are conducted, each with the k possible outcomes and the k probabilities, $p_1, p_2, \ldots, p_k$. Let us number the outcomes 1, 2, 3, …, k. For each outcome, let X_t denote the number of trials that result in that outcome. Then, $X_1, X_2, \ldots, X_k$ are discrete random variables. The collection of $X_1, X_2, \ldots, X_k$ is said to have the **multinomial distribution** with parameters, $n, p_1, p_2, \ldots, p_k$.

In our context, n corresponds to the length of a document, and the outcomes correspond to all the words in the vocabulary V ($k = |V|$). $p_1, p_2, \ldots, p_k$ correspond to the probabilities of occurrence of the words in V in a document, which are $\Pr(w_t|c_j; \Theta)$. X_t is a random variable representing the number of times that word w_t appears in a document. We can thus directly apply the probability function of the multinomial distribution to find the probability of a document given its class (including the probability of document length, $\Pr(|d_i|)$, which is assumed to be independent of class):

$$\Pr(d_i \mid c_j; \Theta) = \Pr(|d_i|) \, |d_i|! \prod_{t=1}^{|V|} \frac{\Pr(w_t \mid c_j; \Theta)^{N_{ti}}}{N_{ti}!} \tag{24}$$

where N_{ti} is the number of times that word w_t occurs in document d_i and

$$\sum_{t=1}^{|V|} N_{ti} = |d_i|, \text{ and } \sum_{t=1}^{|V|} \Pr(w_t \mid c_j; \Theta) = 1. \tag{25}$$

The parameters θ_j of the generative component for each class c_j are the probabilities of all words w_t in V, written as $\Pr(w_t|c_j; \Theta)$, and the probabilities of document lengths, which are the same for all classes (or mixture components) due to our assumption.

Parameter Estimation: The parameters can be estimated from the training data $D = \{D_1, D_2, ..., D_{|C|}\}$, where D_j is the subset of data for class c_j (recall $|C|$ is the number of classes). The vocabulary V is the set of all distinctive words in D. Note that we do not need to estimate the probability of each document length as it is not used in our final classifier. The estimate of Θ is written as $\hat{\Theta}$. The parameters are estimated based on empirical counts.

The estimated probability of word w_t given class c_j is simply the number of times that w_t occurs in the training data D_j (of class c_j) divided by the total number of word occurrences in the training data for that class:

$$\Pr(w_t \mid c_j; \hat{\Theta}) = \frac{\sum_{i=1}^{|D|} N_{ti} \Pr(c_j \mid d_i)}{\sum_{s=1}^{|V|} \sum_{i=1}^{|D|} N_{si} \Pr(c_j \mid d_i)}. \quad (26)$$

In Equation (26), we do not use D_j explicitly. Instead, we include $\Pr(c_j|d_i)$ to achieve the same effect because $\Pr(c_j|d_i) = 1$ for each document in D_j and $\Pr(c_j|d_i) = 0$ for documents of other classes. Again, N_{ti} is the number of times that word w_t occurs in document d_i.

In order to handle 0 counts for infrequently occurring words that do not appear in the training set, but may appear in the test set, we need to smooth the probability to avoid probabilities of 0 or 1. This is the same problem as in Sect. 3.6. The standard way of doing this is to augment the count of each distinctive word with a small quantity λ ($0 \le \lambda \le 1$) or a fraction of a word in both the numerator and denominator. Thus, any word will have at least a very small probability of occurrence.

$$\Pr(w_t \mid c_j; \hat{\Theta}) = \frac{\lambda + \sum_{i=1}^{|D|} N_{ti} \Pr(c_j \mid d_i)}{\lambda |V| + \sum_{s=1}^{|V|} \sum_{i=1}^{|D|} N_{si} \Pr(c_j \mid d_i)}. \quad (27)$$

This is called the **Lidstone smoothing** (Lidsone's law of succession). When $\lambda = 1$, the smoothing is known as the **Laplace smoothing**. Many experiments have shown that $\lambda < 1$ works better for text classification [7]. The best λ value for a data set can be found through experiments using a validation set or through cross-validation.

Finally, class prior probabilities, which are mixture weights φ_j, can be easily estimated using the training data as well.

$$\Pr(c_j \mid \hat{\Theta}) = \frac{\sum_{i=1}^{|D|} \Pr(c_j \mid d_i)}{|D|}. \tag{28}$$

Classification: Given the estimated parameters, at the classification time, we need to compute the probability of each class c_j for the test document d_i. That is, we compute the probability that a particular mixture component c_j generated the given document d_i. Using Bayes rule and Equations (23), (24), (27), and (28), we have

$$\begin{aligned} \Pr(c_j \mid d_i; \hat{\Theta}) &= \frac{\Pr(c_j \mid \hat{\Theta}) \Pr(d_i \mid c_j; \hat{\Theta})}{\Pr(d_i \mid \hat{\Theta})} \\ &= \frac{\Pr(c_j \mid \hat{\Theta}) \prod_{k=1}^{|d_i|} \Pr(w_{d_i,k} \mid c_j; \hat{\Theta})}{\sum_{r=1}^{|C|} \Pr(c_r \mid \hat{\Theta}) \prod_{k=1}^{|d_i|} \Pr(w_{d_i,k} \mid c_r; \hat{\Theta})}, \end{aligned} \tag{29}$$

where $w_{d_i,k}$ is the word in position k of document d_i (which is the same as using w_t and N_{ti}). If the final classifier is to classify each document into a single class, the class with the highest posterior probability is selected:

$$\arg\max_{c_j \in C} \Pr(c_j \mid d_i; \hat{\Theta}). \tag{30}$$

3.7.3 Discussion

Most assumptions made by naïve Bayesian learning are violated in practice. For example, words in a document are clearly not independent of each other. The mixture model assumption of one-to-one correspondence between classes and mixture components may not be true either because a class may contain documents from multiple topics. Despite such violations, researchers have shown that naïve Bayesian learning produces very accurate models.

Naïve Bayesian learning is also very efficient. It scans the training data only once to estimate all the probabilities required for classification. It can be used as an incremental algorithm as well. The model can be updated easily as new data comes in because the probabilities can be conveniently revised. Naïve Bayesian learning is thus widely used for text classification.

The naïve Bayesian formulation presented here is based on a mixture of **multinomial distributions**. There is also a formulation based on **multivariate Bernoulli distributions** in which each word in the vocabulary is a binary feature, i.e., it either appears or does not appear in the document.

Thus, it does not consider the number of times that a word occurs in a document. Experimental comparisons show that multinomial formulation consistently produces more accurate classifiers [365].

3.8 Support Vector Machines

Support vector machines (SVM) is another type of learning system [525], which has many desirable qualities that make it one of most popular algorithms. It not only has a solid theoretical foundation, but also performs classification more accurately than most other algorithms in many applications, especially those applications involving very high dimensional data. For instance, it has been shown by several researchers that SVM is perhaps the most accurate algorithm for text classification. It is also widely used in Web page classification and bioinformatics applications.

In general, SVM is a **linear learning system** that builds two-class classifiers. Let the set of training examples D be

$$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\},$$

where $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{ir})$ is a r-dimensional **input vector** in a real-valued space $X \subseteq \Re^r$, y_i is its **class label** (output value) and $y_i \in \{1, -1\}$. 1 denotes the positive class and −1 denotes the negative class. Note that we use slightly different notations in this section. For instance, we use y instead of c to represent a class because y is commonly used to represent classes in the SVM literature. Similarly, each data instance is called an **input vector** and denoted by a bold face letter. In the following, we use bold face letters for all vectors.

To build a classifier, SVM finds a linear function of the form

$$f(\mathbf{x}) = \langle \mathbf{w} \cdot \mathbf{x} \rangle + b \tag{31}$$

so that an input vector $\mathbf{x}_i$ is assigned to the positive class if $f(\mathbf{x}_i) \geq 0$, and to the negative class otherwise, i.e.,

$$y_i = \begin{cases} 1 & \text{if } \langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b \geq 0 \\ -1 & \text{if } \langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b < 0 \end{cases} \tag{32}$$

Hence, $f(\mathbf{x})$ is a real-valued function $f: X \subseteq \Re^r \rightarrow \Re$. $\mathbf{w} = (w_1, w_2, \ldots, w_r) \in \Re^r$ is called the **weight vector**. $b \in \Re$ is called the **bias**. $\langle \mathbf{w} \cdot \mathbf{x} \rangle$ is the **dot product** of $\mathbf{w}$ and $\mathbf{x}$ (or **Euclidean inner product**). Without using vector notation, Equation (31) can be written as:

$$f(x_1, x_2, \ldots, x_r) = w_1x_1 + w_2x_2 + \ldots + w_rx_r + b,$$

where x_i is the variable representing the ith coordinate of the vector **x**. For convenience, we will use the vector notation from now on.

In essence, SVM finds a hyperplane

$$\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0 \tag{33}$$

that separates positive and negative training examples. This hyperplane is called the **decision boundary** or **decision surface**.

Geometrically, the hyperplane $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$ divides the input space into two half spaces: one half for positive examples and the other half for negative examples. Recall that a hyperplane is commonly called **a line** in a 2-dimensional space and **a plane** in a 3-dimensional space.

Fig. 3.16(A) shows an example in a 2-dimensional space. Positive instances (also called positive data points or simply positive points) are represented with small filled rectangles, and negative examples are represented with small empty circles. The thick line in the middle is the decision boundary hyperplane (a line in this case), which separates positive (above the line) and negative (below the line) data points. Equation (31), which is also called the **decision rule** of the SVM classifier, is used to make classification decisions on test instances.

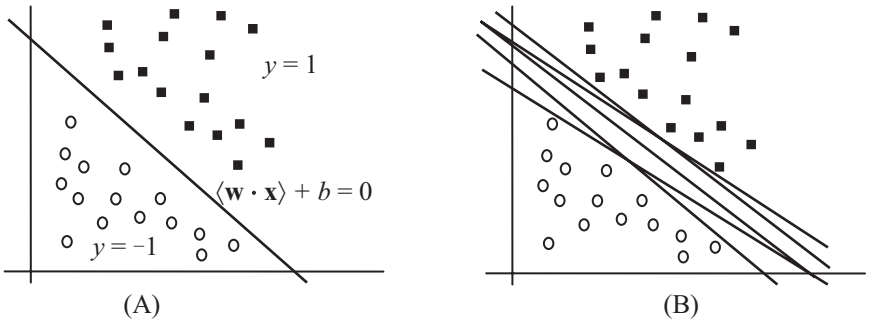


Fig. 3.16. (A) A linearly separable data set and (B) possible decision boundaries

Fig. 3.16(A) raises two interesting questions:

1. There are an infinite number of lines that can separate the positive and negative data points as illustrated by Fig. 3.16(B). Which line should we choose?
2. A hyperplane classifier is only applicable if the positive and negative data can be linearly separated. How can we deal with nonlinear separations or data sets that require nonlinear decision boundaries?

The SVM framework provides good answers to both questions. Briefly, for question 1, SVM chooses the hyperplane that maximizes the margin (the

gap) between positive and negative data points, which will be defined formally shortly. For question 2, SVM uses **kernel** functions. Before we dive into the details, we should note that SVM requires numeric data and only builds two-class classifiers. At the end of the section, we will discuss how these limitations may be addressed.

3.8.1 Linear SVM: Separable Case

This sub-section studies the simplest case of linear SVM. It is assumed that the positive and negative data points are linearly separable.

From linear algebra, we know that in $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$, $\mathbf{w}$ defines a direction perpendicular to the hyperplane (see Fig. 3.17). $\mathbf{w}$ is also called the **normal vector** (or simply **normal**) of the hyperplane. Without changing the normal vector $\mathbf{w}$, varying b moves the hyperplane parallel to itself. Note also that $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$ has an inherent degree of freedom. We can rescale the hyperplane to $\langle \lambda\mathbf{w} \cdot \mathbf{x} \rangle + \lambda b = 0$ for $\lambda \in \Re^+$ (positive real numbers) without changing the function/hyperplane.

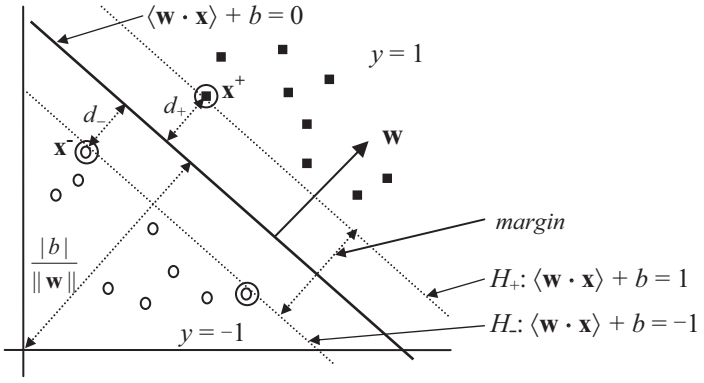


Fig. 3.17. Separating hyperplanes and margin of SVM: Support vectors are circled

Since SVM maximizes the margin between positive and negative data points, let us find the margin. Let d_+ (respectively d_-) be the shortest distance from the separating hyperplane ($\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$) to the closest positive (negative) data point. The **margin** of the separating hyperplane is $d_+ + d_-$. SVM looks for the separating hyperplane with the largest margin, which is also called the **maximal margin hyperplane**, as the final **decision boundary**. The reason for choosing this hyperplane to be the decision boundary is because theoretical results from *structural risk minimization* in

computational learning theory show that maximizing the margin minimizes the upper bound of classification errors.

Let us consider a positive data point ($\mathbf{x}^+$, 1) and a negative ($\mathbf{x}^-$, −1) that are closest to the hyperplane $<\mathbf{w} \cdot \mathbf{x}> + b = 0$. We define two parallel hyperplanes, H_+ and H_-, that pass through $\mathbf{x}^+$ and $\mathbf{x}^-$ respectively. H_+ and H_- are also parallel to $<\mathbf{w} \cdot \mathbf{x}> + b = 0$. We can rescale $\mathbf{w}$ and b to obtain

$$H_+: \quad \langle \mathbf{w} \cdot \mathbf{x}^+ \rangle + b = 1 \tag{34}$$

$$H_-: \quad \langle \mathbf{w} \cdot \mathbf{x}^- \rangle + b = -1 \tag{35}$$

such that $\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b \geq 1$ if $y_i = 1$

$\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b \leq -1$ if $y_i = -1$,

which indicate that no training data fall between hyperplanes H_+ and H_-.

Now let us compute the distance between the two **margin hyperplanes** H_+ and H_-. Their distance is the **margin** ($d_+ + d_-$). Recall from vector space in linear algebra that the (perpendicular) Euclidean distance from a point $\mathbf{x}_i$ to a hyperplane $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$ is:

$$\frac{|\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b|}{\|\mathbf{w}\|}, \tag{36}$$

where $\|\mathbf{w}\|$ is the Euclidean norm of $\mathbf{w}$,

$$\|\mathbf{w}\| = \sqrt{<\mathbf{w} \cdot \mathbf{w}>} = \sqrt{w_1^2 + w_2^2 + \ldots + w_r^2} \tag{37}$$

To compute d_+, instead of computing the distance from $\mathbf{x}^+$ to the separating hyperplane $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$, we pick up any point $\mathbf{x}_s$ on $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$ and compute the distance from $\mathbf{x}_s$ to $\langle \mathbf{w} \cdot \mathbf{x}^+ \rangle + b = 1$ by applying Equation 36 and noticing that $\langle \mathbf{w} \cdot \mathbf{x}_s \rangle + b = 0$,

$$d_+ = \frac{|\langle \mathbf{w} \cdot \mathbf{x}_s \rangle + b - 1|}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|} \tag{38}$$

Likewise, we can compute the distance of $\mathbf{x}_s$ to $\langle \mathbf{w} \cdot \mathbf{x}^+ \rangle + b = -1$ to obtain $d_- = 1/\|\mathbf{w}\|$. Thus, the decision boundary $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$ lies half way between H_+ and H_-. The margin is thus

$$margin = d_+ + d_- = \frac{2}{\|\mathbf{w}\|} \tag{39}$$

In fact, we can compute the margin in many ways. For example, it can be computed by finding the distances from the origin to the three hyperplanes, or by projecting the vector ($\mathbf{x}_2^- - \mathbf{x}_1^+$) to the normal vector $\mathbf{w}$.

Since SVM looks for the separating hyperplane that maximizes the margin, this gives us an optimization problem. Since maximizing the margin is the same as minimizing $||\mathbf{w}||^2/2 = \langle \mathbf{w} \cdot \mathbf{w} \rangle/2$. We have the following linear separable SVM formulation.

Definition (Linear SVM: Separable Case): Given a set of linearly separable training examples,

$$D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\},$$

learning is to solve the following constrained minimization problem,

$$\begin{aligned} &\text{Minimize}: \ \frac{\langle \mathbf{w} \cdot \mathbf{w} \rangle}{2} \\ &\text{Subject to}: \ y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) \geq 1, \quad i = 1, 2, ..., n \end{aligned} \tag{40}$$

Note that the constraint $y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b \geq 1, \quad i = 1, 2, ..., n$ summarizes:

$$\begin{aligned} \langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b &\geq 1 \qquad \text{for } y_i = 1 \\ \langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b &\leq -1 \qquad \text{for } y_i = -1. \end{aligned}$$

Solving the problem (40) will produce the solutions for $\mathbf{w}$ and b, which in turn give us the maximal margin hyperplane $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$ with the margin $2/||\mathbf{w}||$.

A full description of the solution method requires a significant amount of optimization theory, which is beyond the scope of this book. We will only use those relevant results from optimization without giving formal definitions, theorems or proofs.

Since the objective function is quadratic and convex and the constraints are linear in the parameters $\mathbf{w}$ and b, we can use the standard Lagrangian multiplier method to solve it.

Instead of optimizing only the objective function (which is called unconstrained optimization), we need to optimize the Lagrangian of the problem, which considers the constraints at the same time. The need to consider constraints is obvious because they restrict the feasible solutions. Since our inequality constraints are expressed using "≥", the **Lagrangian** is formed by the constraints multiplied by positive Lagrange multipliers and subtracted from the objective function, i.e.,

$$L_P = \frac{1}{2}\langle \mathbf{w} \cdot \mathbf{w} \rangle - \sum_{i=1}^{n} \alpha_i [y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) - 1] \tag{41}$$

where $\alpha_i \geq 0$ are the **Lagrange multipliers**.

The optimization theory says that an optimal solution to (41) must satisfy certain conditions, called **Kuhn–Tucker conditions**, which play a

central role in constrained optimization. Here, we give a brief introduction to these conditions. Let the general optimization problem be

$$\begin{aligned} &\text{Minimize} : \ f(\mathbf{x}) \\ &\text{Subject to} : \ g_i(\mathbf{x}) \le b_i, \ \ i = 1, 2, ..., n \end{aligned} \tag{42}$$

where f is the objective function and g_i is a constraint function (which is different from y_i in (40) as y_i is not a function but a class label of 1 or −1). The Lagrangian of (42) is,

$$L_P = f(\mathbf{x}) + \sum_{i=1}^{n} \alpha_i [g_i(\mathbf{x}) - b_i)] \tag{43}$$

An optimal solution to the problem in (42) must satisfy the following **necessary** (but **not sufficient**) conditions:

$$\frac{\partial L_P}{\partial x_j} = 0, \ \ j = 1, 2, ..., r \tag{44}$$

$$g_i(\mathbf{x}) - b_i \le 0, \ \ i = 1, 2, ..., n \tag{45}$$

$$\alpha_i \ge 0, \ \ i = 1, 2, ..., n \tag{46}$$

$$\alpha_i (b_i - g_i(\mathbf{x}_i)) = 0, \ \ i = 1, 2, ..., n \tag{47}$$

These conditions are called the **Kuhn–Tucker conditions**. Note that (45) is simply the original set of constraints in (42). The condition (47) is called the **complementarity condition**, which implies that at the solution point,

$$\begin{aligned} &\text{If} \ \ \alpha_i > 0 \qquad \text{then} \quad g_i(\mathbf{x}) = b_i. \\ &\text{If} \ \ g_i(\mathbf{x}) > b_i \quad \text{then} \quad \alpha_i = 0. \end{aligned}$$

These mean that for active constraints, $\alpha_i > 0$, whereas for inactive constraints $\alpha_i = 0$. As we will see later, they give some very desirable properties to SVM.

Let us come back to our problem. For the minimization problem (40), the Kuhn–Tucker conditions are (48)–(52):

$$\frac{\partial L_P}{\partial w_j} = w_j - \sum_{i=1}^{n} y_i \alpha_i x_{ij} = 0, \ \ j = 1, 2, ..., r \tag{48}$$

$$\frac{\partial L_P}{\partial b} = -\sum_{i=1}^{n} y_i \alpha_i = 0 \tag{49}$$

$$y_i (\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) - 1 \ge 0, \ \ i = 1, 2, ..., n \tag{50}$$

$$\alpha_i \geq 0, \quad i = 1, 2, ..., n \tag{51}$$

$$\alpha_i (y_i (\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) - 1) = 0, \quad i = 1, 2, ..., n \tag{52}$$

Inequality (50) is the original set of constraints. We also note that although there is a Lagrange multiplier α_i for each training data point, the complementarity condition (52) shows that only those data points on the margin hyperplanes (i.e., H_+ and H_-) can have $\alpha_i > 0$ since for them $y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) - 1 = 0$. These data points are called **support vectors**, which give the name to the algorithm, *support vector machines*. All the other data points have $\alpha_i = 0$.

In general, Kuhn–Tucker conditions are necessary for an optimal solution, but not sufficient. However, for our minimization problem with a convex objective function and a set of linear constraints, the Kuhn–Tucker conditions are both **necessary** and **sufficient** for an optimal solution.

Solving the optimization problem is still a difficult task due to the inequality constraints. However, the Lagrangian treatment of the convex optimization problem leads to an alternative **dual** formulation of the problem, which is easier to solve than the original problem, which is called the **primal** problem (L_P is called the **primal Lagrangian**).

The concept of duality is widely used in the optimization literature. The aim is to provide an alternative formulation of the problem which is more convenient to solve computationally and/or has some theoretical significance. In the context of SVM, the dual problem is not only easy to solve computationally, but also crucial for using **kernel functions** to deal with nonlinear decision boundaries as we do not need to compute **w** explicitly (which will be clear later).

Transforming from the primal to its corresponding dual can be done by setting to zero the partial derivatives of the Lagrangian (41) with respect to the **primal variables** (i.e., **w** and b), and substituting the resulting relations back into the Lagrangian. This is to simply substitute (48), which is

$$w_j = \sum_{i=1}^{n} y_i \alpha_i x_{ij}, \quad j = 1, 2, ..., r \tag{53}$$

and (49), which is

$$\sum_{i=1}^{n} y_i \alpha_i = 0, \tag{54}$$

into the original Lagrangian (41) to eliminate the primal variables, which gives us the dual objective function (denoted by L_D),

$$L_D = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_i \cdot \mathbf{x}_j \rangle. \tag{55}$$

L_D contains only **dual variables** and must be maximized under the simpler constraints, (48) and (49), and $\alpha_i \geq 0$. Note that (48) is not needed as it has already been substituted into the objective function L_D. Hence, the **dual** of the primal Equation (40) is

$$\begin{aligned} &\text{Maximize: } L_D = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_i \cdot \mathbf{x}_j \rangle. \\ &\text{Subject to: } \sum_{i=1}^{n} y_i \alpha_i = 0 \\ &\qquad\qquad \alpha_i \geq 0, \quad i = 1, 2, ..., n. \end{aligned} \tag{56}$$

This dual formulation is called the **Wolfe dual**. For our convex objective function and linear constraints of the primal, it has the property that the α_i's at the maximum of L_D gives **w** and b occurring at the minimum of L_P (the primal).

Solving (56) requires numerical techniques and clever strategies beyond the scope of this book. After solving (56), we obtain the values for α_i, which are used to compute the weight vector **w** and the bias b using Equations (48) and (52) respectively. Instead of depending on one support vector ($\alpha_i > 0$) to compute b, in practice all support vectors are used to compute b, and then take their average as the final value for b. This is because the values of α_i are computed numerically and can have numerical errors. Our final **decision boundary** (**maximal margin hyperplane)** is

$$\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = \sum_{i \in sv} y_i \alpha_i \langle \mathbf{x}_i \cdot \mathbf{x} \rangle + b = 0 \tag{57}$$

where sv is the set of indices of the support vectors in the training data.

Testing: We apply (57) for classification. Given a test instance **z**, we classify it using the following:

$$sign(\langle \mathbf{w} \cdot \mathbf{z} \rangle + b) = sign\left(\sum_{i \in sv} y_i \alpha_i \langle \mathbf{x}_i \cdot \mathbf{z} \rangle + b \right). \tag{58}$$

If (58) returns 1, then the test instance **z** is classified as positive; otherwise, it is classified as negative.

3.8.2 Linear SVM: Non-separable Case

The linear separable case is the ideal situation. In practice, however, the training data is almost always noisy, i.e., containing errors due to various reasons. For example, some examples may be labeled incorrectly. Furthermore, practical problems may have some degree of randomness. Even for two identical input vectors, their labels may be different.

For SVM to be useful, it must allow noise in the training data. However, with noisy data the linear separable SVM will not find a solution because the constraints cannot be satisfied. For example, in Fig. 3.18, there is a negative point (circled) in the positive region, and a positive point in the negative region. Clearly, no solution can be found for this problem.

Recall that the primal for the linear separable case was:

$$\begin{aligned} &\text{Minimize}: \ \frac{\langle \mathbf{w} \cdot \mathbf{w} \rangle}{2} \\ &\text{Subject to}: \ y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) \geq 1, \quad i = 1, 2, ..., n. \end{aligned} \tag{59}$$

To allow errors in data, we can relax the margin constraints by introducing **slack** variables, ξ_i (≥ 0) as follows:

$$\begin{aligned} \langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b &\geq 1 - \xi_i \quad \text{for } y_i = 1 \\ \langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b &\leq -1 + \xi_i \quad \text{for } y_i = -1. \end{aligned}$$

Thus we have the new constraints:

$$\begin{aligned} \text{Subject to:} \quad & y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) \geq 1 - \xi_i, \ i = 1, 2, \ldots, n, \\ & \xi_i \geq 0, \ \ i = 1, 2, \ldots, n. \end{aligned}$$

The geometric interpretation is shown in Fig. 3.18, which has two error data points $\mathbf{x}_a$ and $\mathbf{x}_b$ (circled) in wrong regions.

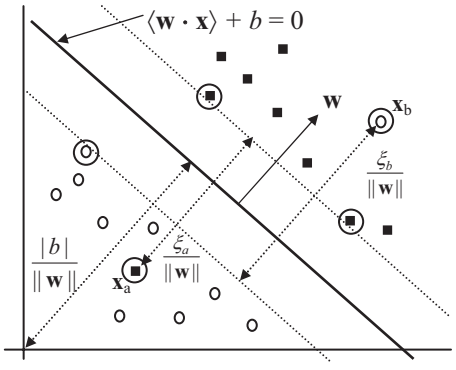


Fig. 3.18. The non-separable case: $\mathbf{x}_a$ and $\mathbf{x}_b$ are error data points

We also need to penalize the errors in the objective function. A natural way is to assign an extra cost for errors to change the objective function to

$$\text{Minimize}: \frac{\langle \mathbf{w} \cdot \mathbf{w} \rangle}{2} + C\left(\sum_{i=1}^{n} \xi_i\right)^k \tag{60}$$

where $C \geq 0$ is a user specified parameter. The resulting optimization problem is still a convex programming problem. $k = 1$ is commonly used, which has the advantage that neither ξ_i nor its Lagrangian multipliers appear in the dual formulation. We only discuss the $k = 1$ case below.

The new optimization problem becomes:

$$\begin{aligned} &\text{Minimize}: \frac{\langle \mathbf{w} \cdot \mathbf{w} \rangle}{2} + C\sum_{i=1}^{n} \xi_i \\ &\text{Subject to}: y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) \geq 1 - \xi_i, \quad i = 1, 2, ..., n \\ &\qquad\qquad \xi_i \geq 0, \quad i = 1, 2, ..., n. \end{aligned} \tag{61}$$

This formulation is called the **soft-margin SVM**. The primal Lagrangian (denoted by L_P) of this formulation is as follows

$$L_P = \frac{1}{2}\langle \mathbf{w} \cdot \mathbf{w} \rangle + C\sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \alpha_i [y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) - 1 + \xi_i] - \sum_{i=1}^{n} \mu_i \xi_i \tag{62}$$

where α_i, $\mu_i \geq 0$ are the **Lagrange multipliers**. The **Kuhn–Tucker conditions** for optimality are the following:

$$\frac{\partial L_P}{\partial w_j} = w_j - \sum_{i=1}^{n} y_i \alpha_i x_{ij} = 0, \quad j = 1, 2, ..., r \tag{63}$$

$$\frac{\partial L_P}{\partial b} = -\sum_{i=1}^{n} y_i \alpha_i = 0 \tag{64}$$

$$\frac{\partial L_P}{\partial \xi_i} = C - \alpha_i - \mu_i = 0, \quad i = 1, 2, ..., n \tag{65}$$

$$y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) - 1 + \xi_i \geq 0, \quad i = 1, 2, ..., n \tag{66}$$

$$\xi_i \geq 0, \quad i = 1, 2, ..., n \tag{67}$$

$$\alpha_i \geq 0, \quad i = 1, 2, ..., n \tag{68}$$

$$\mu_i \geq 0, \quad i = 1, 2, ..., n \tag{69}$$

$$\alpha_i(y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) - 1 + \xi_i) = 0, \quad i = 1, 2, ..., n \tag{70}$$

$$\mu_i \xi_i = 0, \quad i = 1, 2, ..., n \tag{71}$$

As the linear separable case, we then transform the primal to its dual by setting to zero the partial derivatives of the Lagrangian (62) with respect to the **primal variables** (i.e., **w**, b and ξ_i), and substituting the resulting relations back into the Lagrangian. That is, we substitute Equations (63), (64) and (65) into the primal Lagrangian (62). From Equation (65), $C - \alpha_i - \mu_i = 0$, we can deduce that $\alpha_i \le C$ because $\mu_i \ge 0$. Thus, the dual of (61) is

$$\begin{aligned} &\text{Maximize: } L_D(\boldsymbol{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_i \cdot \mathbf{x}_j \rangle \\ &\text{Subject to: } \sum_{i=1}^{n} y_i \alpha_i = 0 \\ &\qquad\qquad 0 \le \alpha_i \le C, \quad i = 1, 2, ..., n. \end{aligned} \tag{72}$$

Interestingly, ξ_i and its Lagrange multipliers μ_i are not in the dual and the objective function is identical to that for the separable case. The only difference is the constraint $\alpha_i \le C$ (inferred from $C - \alpha_i - \mu_i = 0$ and $\mu_i \ge 0$).

The dual problem (72) can also be solved numerically, and the resulting α_i values are then used to compute **w** and b. **w** is computed using Equation (63) and b is computed using the Kuhn–Tucker complementarity conditions (70) and (71). Since we do not have values for ξ_i, we need to get around it. From Equations (65), (70) and (71), we observe that if $0 < \alpha_i < C$ then both $\xi_i = 0$ and $y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) - 1 + \xi_i) = 0$. Thus, we can use any training data point for which $0 < \alpha_i < C$ and Equation (70) (with $\xi_i = 0$) to compute b:

$$b = \frac{1}{y_i} - \sum_{i=1}^{n} y_i \alpha_i \langle \mathbf{x}_i \cdot \mathbf{x}_j \rangle. \tag{73}$$

Again, due to numerical errors, we can compute all possible *b's* and then take their average as the final b value.

Note that Equations (65), (70) and (71) in fact tell us more:

$$\begin{aligned} \alpha_i = 0 \quad &\Rightarrow \quad y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) \ge 1 \text{ and } \xi_i = 0 \\ 0 < \alpha_i < C \quad &\Rightarrow \quad y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) = 1 \text{ and } \xi_i = 0 \\ \alpha_i = C \quad &\Rightarrow \quad y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) \le 1 \text{ and } \xi_i \ge 0 \end{aligned} \tag{74}$$

Similar to support vectors for the separable case, (74) shows one of the most important properties of SVM: the solution is sparse in α_i. Most training data points are outside the margin area and their α_i's in the solution are 0. Only those data points that are on the margin (i.e., $y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) = 1$, which are support vectors in the separable case), inside the margin (i.e., α_i

= C and $y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) < 1$), or errors are non-zero. Without this sparsity property, SVM would not be practical for large data sets.

The final decision boundary is (we note that many α_i's are 0)

$$\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = \sum_{i=1}^{n} y_i \alpha_i \langle \mathbf{x}_i \cdot \mathbf{x} \rangle + b = 0. \tag{75}$$

The decision rule for classification (testing) is the same as the separable case, i.e., $sign(\langle \mathbf{w} \cdot \mathbf{x} \rangle + b)$. We notice that for both Equations (75) and (73), $\mathbf{w}$ does not need to be explicitly computed. This is crucial for using kernel functions to handle nonlinear decision boundaries.

Finally, we still have the problem of determining the parameter C. The value of C is usually chosen by trying a range of values on the training set to build multiple classifiers and then to test them on a validation set before selecting the one that gives the best classification result on the validation set. Cross-validation is commonly used as well.

3.8.3 Nonlinear SVM: Kernel Functions

The SVM formulations discussed so far require that positive and negative examples can be linearly separated, i.e., the decision boundary must be a hyperplane. However, for many real-life data sets, the decision boundaries are nonlinear. To deal with nonlinearly separable data, the same formulation and solution techniques as for the linear case are still used. We only transform the input data from its original space into another space (usually of a much higher dimensional space) so that a linear decision boundary can separate positive and negative examples in the transformed space, which is called the **feature space**. The original data space is called the **input space**.

Thus, the basic idea is to map the data in the input space X to a feature space F via a nonlinear mapping ϕ,

$$\begin{aligned} &\phi : X \to F \\ &\mathbf{x} \mapsto \phi(\mathbf{x}). \end{aligned} \tag{76}$$

After the mapping, the original training data set $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$ becomes:

$$\{(\phi(\mathbf{x}_1), y_1), (\phi(\mathbf{x}_2), y_2), \ldots, (\phi(\mathbf{x}_n), y_n)\}. \tag{77}$$

The same linear SVM solution method is then applied to F. Figure 3.19 illustrates the process. In the input space (figure on the left), the training examples cannot be linearly separated. In the transformed feature space (figure on the right), they can be separated linearly.

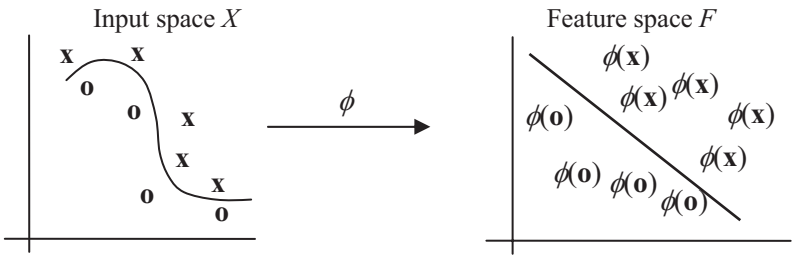


Fig. 3.19. Transformation from the input space to the feature space

With the transformation, the optimization problem in (61) becomes

$$\begin{aligned} &\text{Minimize}: \ \frac{\langle \mathbf{w} \cdot \mathbf{w} \rangle}{2} + C\sum_{i=1}^{n} \xi_i \\ &\text{Subject to}: \ y_i(\langle \mathbf{w} \cdot \phi(\mathbf{x}_i) \rangle + b) \geq 1 - \xi_i, \quad i = 1, 2, ..., n \\ &\qquad\qquad \xi_i \geq 0, \quad i = 1, 2, ..., n \end{aligned} \tag{78}$$

Its corresponding dual is

$$\text{Maximize: } L_D = \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{n} y_i y_j \alpha_i \alpha_j \langle \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j) \rangle. \tag{79}$$

$$\begin{aligned} &\text{Subject to:} \ \sum_{i=1}^{n} y_i \alpha_i = 0 \\ &\qquad\qquad 0 \leq \alpha_i \leq C, \quad i = 1, 2, ..., n. \end{aligned}$$

The final decision rule for classification (testing) is

$$\sum_{i=1}^{n} y_i \alpha_i \langle \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}) \rangle + b \tag{80}$$

Example 16: Suppose our input space is 2-dimensional, and we choose the following transformation (mapping):

$$(x_1, x_2) \mapsto (x_1^2, x_2^2, \sqrt{2}x_1 x_2) \tag{81}$$

The training example ((2, 3), −1) in the input space is transformed to the following training example in the feature space:

((4, 9, 8.5), −1). ■

The potential problem with this approach of transforming the input data explicitly to a feature space and then applying the linear SVM is that it

may suffer from the curse of dimensionality. The number of dimensions in the feature space can be huge with some useful transformations (see below) even with reasonable numbers of attributes in the input space. This makes it computationally infeasible to handle.

Fortunately, explicit transformations can be avoided if we notice that in the dual representation both the construction of the optimal hyperplane (79) in F and the evaluation of the corresponding decision/classification function (80) only require the evaluation of dot products $\langle \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \rangle$ and never the mapped vector $\phi(\mathbf{x})$ in its explicit form. This is a crucial point.

Thus, if we have a way to compute the dot product $\langle \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \rangle$ in the feature space F using the input vectors $\mathbf{x}$ and $\mathbf{z}$ directly, then we would not need to know the feature vector $\phi(\mathbf{x})$ or even the mapping function ϕ itself. In SVM, this is done through the use of **kernel functions**, denoted by K,

$$K(\mathbf{x}, \mathbf{z}) = \langle \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \rangle, \tag{82}$$

which are exactly the functions for computing dot products in the transformed feature space using input vectors $\mathbf{x}$ and $\mathbf{z}$. An example of a kernel function is the **polynomial kernel**,

$$K(\mathbf{x}, \mathbf{z}) = \langle \mathbf{x} \cdot \mathbf{z} \rangle^d. \tag{83}$$

Example 17: Let us compute this kernel with degree $d = 2$ in a 2-dimensional space. Let $\mathbf{x} = (x_1, x_2)$ and $\mathbf{z} = (z_1, z_2)$.

$$\begin{aligned} \langle \mathbf{x} \cdot \mathbf{z} \rangle^2 &= (x_1 z_1 + x_2 z_2)^2 \\ &= {x_1}^2 {z_1}^2 + 2 x_1 z_1 x_2 z_2 + {x_2}^2 {z_2}^2 \\ &= \langle ({x_1}^2, {x_2}^2, \sqrt{2} x_1 x_2) \cdot ({z_1}^2, {z_2}^2, \sqrt{2} z_1 z_2) \rangle \\ &= \langle \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \rangle, \end{aligned} \tag{84}$$

where $\phi(\mathbf{x}) = ({x_1}^2, {x_2}^2, \sqrt{2} x_1 x_2)$, which shows that the kernel $\langle \mathbf{x} \cdot \mathbf{z} \rangle^2$ is a dot product in the transformed feature space. The number of dimensions in the feature space is 3. Note that $\phi(\mathbf{x})$ is actually the mapping function used in Example 16. Incidentally, in general the number of dimensions in the feature space for the polynomial kernel function $K(\mathbf{x}, \mathbf{z}) = \langle \mathbf{x} \cdot \mathbf{z} \rangle^d$ is $\binom{r+d-1}{d}$, which is a huge number even with a reasonable number (r) of attributes in the input space. Fortunately, by using the kernel function in (83), the huge number of dimensions in the feature space does not matter. ■

The derivation in (84) is only for illustration purposes. We do not need to find the mapping function. We can simply apply the kernel function di-

rectly. That is, we replace all the dot products $\langle \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \rangle$ in (79) and (80) with the kernel function $K(\mathbf{x}, \mathbf{z})$ (e.g., the polynomial kernel in (83)). This strategy of directly using a kernel function to replace dot products in the feature space is called the **kernel trick**. We would never need to explicitly know what ϕ is.

However, the question is, how do we know whether a function is a kernel without performing the derivation such as that in (84)? That is, how do we know that a kernel function is indeed a dot product in some feature space? This question is answered by a theorem called the **Mercer's theorem**, which we will not discuss here. See [118] for details.

It is clear that the idea of kernel generalizes the dot product in the input space. The dot product is also a kernel with the feature map being the identity

$$K(\mathbf{x}, \mathbf{z}) = \langle \mathbf{x} \cdot \mathbf{z} \rangle. \tag{85}$$

Commonly used kernels include

$$\text{Polynomial:} \quad K(\mathbf{x}, \mathbf{z}) = (\langle \mathbf{x} \cdot \mathbf{z} \rangle + \theta)^d \tag{86}$$

$$\text{Gaussian RBF:} \quad K(\mathbf{x}, \mathbf{z}) = e^{-\|\mathbf{x}-\mathbf{z}\|^2/2\sigma} \tag{87}$$

where $\theta \in \Re$, $d \in N$, and $\sigma > 0$.

Summary

SVM is a linear learning system that finds the maximal margin decision boundary to separate positive and negative examples. Learning is formulated as a quadratic optimization problem. Nonlinear decision boundaries are found via a transformation of the original data to a much higher dimensional feature space. However, this transformation is never explicitly done. Instead, kernel functions are used to compute dot products required in learning without the need to even know the transformation function.

Due to the separation of the learning algorithm and kernel functions, kernels can be studied independently from the learning algorithm. One can design and experiment with different kernel functions without touching the underlying learning algorithm.

SVM also has some limitations:

1. It works only in real-valued space. For a categorical attribute, we need to convert its categorical values to numeric values. One way to do this is to create an extra binary attribute for each categorical value, and set the attribute value to 1 if the categorical value appears, and 0 otherwise.

2. It allows only two classes, i.e., binary classification. For multiple class classification problems, several strategies can be applied, e.g., one-against-rest, and error-correcting output coding [138].
3. The hyperplane produced by SVM is hard to understand by users. It is difficult to picture where the hyperplane is in a high-dimensional space. The matter is made worse by kernels. Thus, SVM is commonly used in applications that do not required human understanding.

3.9 K-Nearest Neighbor Learning

All the previous learning methods learn some kinds of models from the data, e.g., decision trees, sets of rules, posterior probabilities, and hyperplanes. These learning methods are often called **eager learning** methods as they learn models of the data before testing. In contrast, *k*-nearest neighbor (*k*NN) is a **lazy learning** method in the sense that no model is learned from the training data. Learning only occurs when a test example needs to be classified. The idea of *k*NN is extremely simple and yet quite effective in many applications, e.g., text classification.

It works as follows: Again let D be the training data set. Nothing will be done on the training examples. When a test instance d is presented, the algorithm compares d with every training example in D to compute the similarity or distance between them. The k most similar (closest) examples in D are then selected. This set of examples is called the ***k* nearest neighbors** of d. d then takes the most frequent class among the k nearest neighbors. Note that $k = 1$ is usually not sufficient for determining the class of d due to noise and outliers in the data. A set of nearest neighbors is needed to accurately decide the class. The general *k*NN algorithm is given in Fig. 3.20.

Algorithm *k*NN(D, d, k)
1 Compute the distance between d and every example in D;
2 Choose the k examples in D that are nearest to d, denote the set by P ($\subseteq D$);
3 Assign d the class that is the most frequent class in P (or the majority class).

Fig. 3.20. The *k*-nearest neighbor algorithm

The key component of a *k*NN algorithm is the **distance/similarity function**, which is chosen based on applications and the nature of the data. For relational data, the Euclidean distance is commonly used. For text documents, cosine similarity is a popular choice. We will introduce these distance functions and many others in the next chapter.

The number of nearest neighbors k is usually determined by using a validation set, or through cross validation on the training data. That is, a

range of k values are tried, and the k value that gives the best accuracy on the validation set (or cross validation) is selected. Figure 3.21 illustrates the importance of choosing the right k.

Example 18: In Fig. 3.21, we have two classes of data, positive (filled squares) and negative (empty circles). If 1-nearest neighbor is used, the test data point $\oplus$ will be classified as negative, and if 2-nearest neighbors are used, the class cannot be decided. If 3-nearest neighbors are used, the class is positive as two positive examples are in the 3-nearest neighbors.

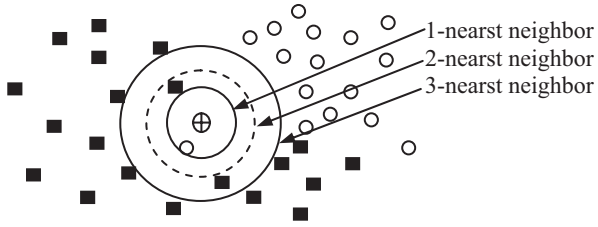


Fig. 3.21. An illustration of k-nearest neighbor classification

Despite its simplicity, researchers have showed that the classification accuracy of kNN can be quite strong and in many cases as accurate as those elaborated methods. For instance, it is showed in [574] that kNN performs equally well as SVM for some text classification tasks. kNN is also very flexible. It can work with any arbitrarily shaped decision boundaries.

kNN is, however, slow at the classification time. Due to the fact that there is no model building, each test instance is compared with every training example at the classification time, which can be quite time consuming especially when the training set D and the test set are large. Another disadvantage is that kNN does not produce an understandable model. It is thus not applicable if an understandable model is required in the application.

3.10 Ensemble of Classifiers

So far, we have studied many individual classifier building techniques. A natural question to ask is: can we build many classifiers and then combine them to produce a better classifier? Yes, in many cases. This section describes two well known ensemble techniques, **bagging** and **boosting**. In both these methods, many classifiers are built and the final classification decision for each test instance is made based on some forms of voting of the committee of classifiers.

3.10.1 Bagging

Given a training set D with n examples and a base learning algorithm, bagging (for *B*ootstrap *Agg*regat*ing*) works as follows [63]:

Training:

1. Create k bootstrap samples S_1, S_2, and S_k. Each sample is produced by drawing n examples at random from D with replacement. Such a sample is called a **bootstrap replicate** of the original training set D. On average, each sample S_i contains 63.2% of the original examples in D, with some examples appearing multiple times.
2. Build a classifier based on each sample S_i. This gives us k classifiers. All the classifiers are built using the same base learning algorithm.

Testing: Classify each test (or new) instance by voting of the k classifiers (equal weights). The majority class is assigned as the class of the instance.

Bagging can improve the accuracy significantly for unstable learning algorithms, i.e., a slight change in the training data resulting in a major change in the output classifier. Decision tree and rule induction methods are examples of unstable learning methods. k-nearest neighbor and naïve Bayesian methods are examples of stable techniques. For stable classifiers, Bagging may sometime degrade the accuracy.

3.10.2 Boosting

Boosting is a family of ensemble techniques, which, like bagging, also manipulates the training examples and produces multiple classifiers to improve the classification accuracy [477]. Here we only describe the popular **AdaBoost** algorithm given in [186]. Unlike bagging, AdaBoost assigns a weight to each training example.

Training: AdaBoost produces a sequence of classifiers (also using the same base learner). Each classifier is dependent on the previous one, and focuses on the previous one's errors. Training examples that are incorrectly classified by the previous classifiers are given higher weights.

Let the original training set D be $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$, where $\mathbf{x}_i$ is an input vector, y_i is its class label and $y_i \in Y$ (the set of class labels). With a weight attached to each example, we have, $\{(\mathbf{x}_1, y_1, w_1), (\mathbf{x}_2, y_2, w_2), \ldots, (\mathbf{x}_n, y_n, w_n)\}$, and $\sum_i w_i = 1$. The AdaBoost algorithm is given in Fig. 3.22.

The algorithm builds a sequence of k classifiers (k is specified by the user) using a base learner, called BaseLeaner in line 3. Initially, the weight

AdaBoost(D, Y, BaseLeaner, k)

1. Initialize $D_1(w_i) \leftarrow 1/n$ for all i; // initialize the weights
2. **for** $t = 1$ to k **do**
3. $f_t \leftarrow$ BaseLearner(D_t); // build a new classifier f_t
4. $e_t \leftarrow \sum_{i: f_t(D_t(\mathbf{x}_i)) \neq y_i} D_t(w_i)$; // compute the error of f_t
5. **if** $e_t > ½$ **then** // if the error is too large,
6. $k \leftarrow k - 1$; // remove the iteration and
7. exit-loop // exit
8. **else**
9. $\beta_t \leftarrow e_t / (1 - e_t)$;
10. $D_{t+1}(w_i) \leftarrow D_t(w_i) \times \begin{cases} \beta_t & \text{if } f_t(D_t(\mathbf{x}_i)) = y_i \\ 1 & \text{otherwise} \end{cases}$; // update the weights
11. $D_{t+1}(w_i) \leftarrow \frac{D_{t+1}(w_i)}{\sum_{i=1}^{n} D_{t+1}(w_i)}$ // normalize the weights
12. **endif**
13. **endfor**
14. $f_{final}(\mathbf{x}) \leftarrow \underset{y \in Y}{\text{argmax}} \sum_{t: f_t(\mathbf{x}) = y} \log \frac{1}{\beta_t}$ // the final output classifier

Fig. 3.22. The AdaBoost algorithm

for each training example is $1/n$ (line 1). In each iteration, the training data set becomes D_t, which is the same as D, but with different weights. Each iteration builds a new classifier f_t (line 3). The error of f_t is calculated in line 4. If it is too large, delete the iteration and exit (lines 5–7). Lines 9–11 update and normalize the weights for building the next classifier.

Testing: For each test case, the results of the series of classifiers are combined to determine the final class of the test case, which is shown in line 14 of Fig. 3.22 (a weighted voting).

Boosting works better than bagging in most cases as shown in [454]. It also tends to improve performance more when the base learner is unstable.

Bibliographic Notes

Supervised learning has been studied extensively by the machine learning community. The book by Mitchell [385] covers most learning techniques and is easy to read. Duda et al.'s pattern classification book is also a great

reference [155]. Most data mining books have one or two chapters on supervised learning, e.g., those by Han and Kamber [218], Hand et al. [221], Tan et al. [512], and Witten and Frank [549].

For decision tree induction, Quinlan's book [453] has all the details and the code of his popular decision tree system C4.5. Other well-known systems include CART by Breiman et al. [62] and CHAD by Kass [270]. Scaling up of decision tree algorithms was also studied in several papers. These algorithms can have the data on disk, and are thus able to run with huge data sets. See [195] for an algorithm and also additional references.

Rule induction algorithms generate rules directly from the data. Well-known systems include AQ by Michalski et al. [381], CN2 by Clark and Niblett [104], FOIL by Quinlan [452], FOCL by Pazzani et al. [438], I-REP by Furnkranz and Widmer [189], and RIPPER by Cohen [106].

Using association rules to build classifiers was proposed by Liu et al. in [343], which also reported the CBA system. CBA selects a small subset of class association rules as the classifier. Other classifier building techniques include combining multiple rules by Li et al. [328], using rules as features by Meretakis and Wüthrich [379], Antonie and Zaiane [23], Deshpande and Karpis [131], Jindal and Liu [255], and Lesh et al. [314], generating a subset of rules by Cong et al. [112, 113], Wang et al. [536], Yin and Han [578], and Zaki and Aggarwal [587]. Other systems include those by Dong et al. [149], Li et al. [319, 320], Yang et al. [570], etc.

The naïve Bayesian classification model described in Sect. 3.6 is based on the papers by Domingos and Pazzani [148], Kohavi et al. [285] and Langley et al [301]. The naïve Bayesian classification for text discussed in Sect. 3.7 is based on the multinomial formulation given by McCallum and Nigam [365]. This model was also used earlier by Lewis and Gale [317], Li and Yamanishi [318], and Nigam et al. [413]. Another formulation of naïve Bayes is based on the multivariate Bernoulli model, which was used in Lewis [316], and Robertson and Sparck-Jones [464].

Support vector machines (SVM) was first introduced by Vapnik and his colleagues in 1992 [59]. Further details were given in his 1995 book [525]. Two other books on SVM and kernel methods are those by Cristianini and Shawe-Taylor [118] and Scholkopf and Smola [479]. The discussion of SVM in this chapter is heavily influenced by Cristianini and Shawe-Taylor's book and the tutorial paper by Burges [74]. Two popular SVM systems are SVM^{Light} (available at http://svmlight.joachims.org/) and LIBSVM (available at http://www.csie.ntu.edu.tw/~cjlin/libsvm/).

Existing classifier ensemble methods include bagging by Breiman [63], boosting by Schapire [477] and Freund and Schapire [186], random forest also by Breiman [65], stacking by Wolpert [552], random trees by Fan [169], and many others.

4 Unsupervised Learning

Supervised learning discovers patterns in the data that relate data attributes to a class attribute. These patterns are then utilized to predict the values of the class attribute of future data instances. These classes indicate some real-world predictive or classification tasks such as determining whether a news article belongs to the category of sports or politics, or whether a patient has a particular disease. However, in some other applications, the data have no class attributes. The user wants to explore the data to find some intrinsic structures in them. Clustering is one technology for finding such structures. It organizes data instances into **similarity groups**, called **clusters** such that the data instances in the same cluster are similar to each other and data instances in different clusters are very different from each other. Clustering is often called **unsupervised learning**, because unlike supervised learning, class values denoting an *a priori* partition or grouping of the data are not given. Note that according to this definition, we can also say that association rule mining is an unsupervised learning task. However, due to historical reasons, clustering is closely associated and even synonymous with unsupervised learning while association rule mining is not. We follow this convention, and describe some main clustering techniques in this chapter.

Clustering has been shown to be one of the most commonly used data analysis techniques. It also has a long history, and has been used in almost every field, e.g., medicine, psychology, botany, sociology, biology, archeology, marketing, insurance, library science, etc. In recent years, due to the rapid increase of online documents and the expansion of the Web, text document clustering too has become a very important task. In Chap. 12, we will also see that clustering is very useful in Web usage mining.

4.1 Basic Concepts

Clustering is the process of organizing data instances into groups whose members are similar in some way. A **cluster** is therefore a collection of data instances which are "similar" to each other and are "dissimilar" to

data instances in other clusters. In the clustering literature, a data instance is also called an **object** as the instance may represent an object in the real-world. It is also called a **data point** as it can be seen as a point in an r-dimension space, where r is the number of attributes in the data.

Fig. 4.1 shows a 2-dimensional data set. We can clearly see three groups of data points. Each group is a cluster. The task of clustering is to find the three clusters hidden in the data. Although it is easy for a human to visually detect clusters in a 2-dimensional or even 3-demensional space, it becomes very hard, if not impossible, to detect clusters visually as the number of dimensions increases. Additionally, in many applications, clusters are not as clear-cut or well separated as the three clusters in Fig. 4.1. Automatic techniques are thus needed for clustering.

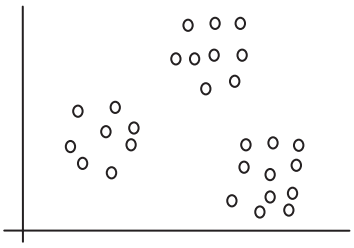


Fig. 4.1. Three natural groups or clusters of data points

After seeing the example in Fig. 4.1, you may ask the question: What is clustering for? To answer it, let us see some application examples from different domains.

Example 1: A company wants to conduct a marketing campaign to promote its products. The most effective strategy is to design a set of personalized marketing materials for each individual customer according to his/her profile and financial situation. However, this is too expensive for a large number of customers. At the other extreme, the company designs only one set of marketing materials to be used for all customers. This one-size-fits-all approach, however, may not be effective. The most cost-effective approach is to segment the customers into a small number of groups according to their similarities and design some targeted marketing materials for each group. This segmentation task is commonly done using clustering algorithms, which **partition** customers into similarity groups. In marketing research, clustering is often called **segmentation**. ■

Example 2: A company wants to produce and sell T-shirts. Similar to the case above, on one extreme, for each customer it can measure his/her size and have a T-shirt tailor-made for him/her. Obviously, this T-shirt is going to be expensive. On the other extreme, only one size of T-shirts is made.

Since this size may not fit most people, the company might not be able to sell as many T-shirts. Again, the most cost effective way is to group people based on their sizes and make a different generalized size of T-shirts for each group. This is why we see small, medium and large size T-shirts in shopping malls, and seldom see T-shirts with only a single size. The method used to group people according to their sizes is clustering. The process is usually as follows: The T-shirt manufacturer first samples a large number of people and measure their sizes to produce a measurement database. It then clusters the data, which **partitions** the data into some similarity subsets, i.e., clusters. For each cluster, it computes the average of the sizes and then uses the average to mass-produce T-shirts for all people of similar size. ■

Example 3: Everyday, news agencies around the world generate a large number of news articles. If a Web site wants to collect these news articles to provide an integrated news service, it has to organize the collected articles according to some topic hierarchy. The question is: What should the topics be, and how should they be organized? One possibility is to employ a group of human editors to do the job. However, the manual organization is costly and very time consuming, which makes it unsuitable for news and other time sensitive information. Throwing all the news articles to the readers with no organization is clearly not an option. Although classification is able to classify news articles according to predefined topics, it is not applicable here because classification needs training data, which have to be manually labeled with topic classes. Since news topics change constantly and rapidly, the training data would need to change constantly as well, which is infeasible via manual labeling. Clustering is clearly a solution for this problem because it automatically groups a stream of news articles based on their content similarities. **Hierarchical clustering algorithms** can also organize documents hierarchically, i.e., each topic may contain sub-topics and so on. Topic hierarchies are particularly useful for texts. ■

The above three examples indicate two types of clustering, **partitional** and **hierarchical**. Indeed, these are the two most important types of clustering approaches. We will study some specific algorithms of these two types of clustering.

Our discussion and examples above also indicate that clustering needs a similarity function to measure how similar two data points (or objects) are, or alternatively a **distance function** to measure the distance between two data points. We will use distance functions in this chapter. The goal of clustering is thus to discover the intrinsic grouping of the input data through the use of a clustering algorithm and a distance function.

```
Algorithm k-means(k, D)
1  choose k data points as the initial centroids (cluster centers)
2  repeat
3     for each data point x ∈ D do
4        compute the distance from x to each centroid;
5        assign x to the closest centroid          // a centroid represents a cluster
6     endfor
7     re-compute the centroid using the current cluster memberships
8  until the stopping criterion is met
```

Fig. 4.2. The *k*-means algorithm

4.2 K-means Clustering

The *k*-means algorithm is the best known **partitional clustering algorithm.** It is perhaps also the most widely used among all clustering algorithms due to its simplicity and efficiency. Given a set of data points and the required number of *k* clusters (*k* is specified by the user), this algorithm iteratively partitions the data into *k* clusters based on a distance function.

4.2.1 K-means Algorithm

Let the set of data points (or instances) D be

$$\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\},$$

where $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{ir})$ is a vector in a real-valued space $X \subseteq \Re^r$, and r is the number of attributes in the data (or the number of dimensions of the **data space**). The *k*-means algorithm partitions the given data into *k* clusters. Each cluster has a cluster **center**, which is also called the cluster **centroid**. The centroid, usually used to represent the cluster, is simply the mean of all the data points in the cluster, which gives the name to the algorithm, i.e., since there are *k* clusters, thus *k* means. Figure 4.2 gives the *k*-means clustering algorithm.

At the beginning, the algorithm randomly selects *k* data points as the **seed** centroids. It then computes the distance between each seed centroid and every data point. Each data point is assigned to the centroid that is closest to it. A centroid and its data points therefore represent a cluster. Once all the data points in the data are assigned, the centroid for each cluster is re-computed using the data points in the current cluster. This process repeats until a stopping criterion is met. The stopping (or convergence) criterion can be any one of the following:

1. no (or minimum) re-assignments of data points to different clusters.
2. no (or minimum) change of centroids.
3. minimum decrease in the **sum of squared error** (SSE),

$$SSE = \sum_{j=1}^{k} \sum_{\mathbf{x} \in C_j} dist(\mathbf{x}, \mathbf{m}_j)^2, \tag{1}$$

where k is the number of required clusters, C_j is the jth cluster, $\mathbf{m}_j$ is the centroid of cluster C_j (the mean vector of all the data points in C_j), and $dist(\mathbf{x}, \mathbf{m}_j)$ is the distance between data point $\mathbf{x}$ and centroid $\mathbf{m}_j$.

The k-means algorithm can be used for any application data set where the **mean** can be defined and computed. In **Euclidean space**, the mean of a cluster is computed with:

$$\mathbf{m}_j = \frac{1}{|C_j|} \sum_{\mathbf{x}_i \in C_j} \mathbf{x}_i, \tag{2}$$

where $|C_j|$ is the number of data points in cluster C_j. The distance from a data point $\mathbf{x}_i$ to a cluster mean (centroid) $\mathbf{m}_j$ is computed with

$$\begin{aligned} dist(\mathbf{x}_i, \mathbf{m}_j) &= \| \mathbf{x}_i - \mathbf{m}_j \| \\ &= \sqrt{(x_{i1} - m_{j1})^2 + (x_{i2} - m_{j2})^2 + ... + (x_{ir} - m_{jr})^2}. \end{aligned} \tag{3}$$

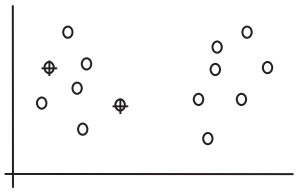
Example 4: Figure 4.3(A) shows a set of data points in a 2-dimensional space. We want to find 2 clusters from the data, i.e., $k = 2$. First, two data points (each marked with a cross) are randomly selected to be the initial centroids (or seeds) shown in Fig. 4.3(A). The algorithm then goes to the first iteration (the repeat-loop).

Iteration 1: Each data point is assigned to its closest centroid to form 2 clusters. The resulting clusters are given in Fig. 4.3(B). Then the centroids are re-computed based on the data points in the current clusters (Fig. 4.3(C)). This leads to iteration 2.

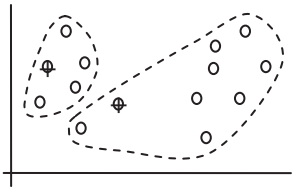
Iteration 2: Again, each data point is assigned to its closest new centroid to form two new clusters shown in Fig. 4.3(D). The centroids are then re-computed. The new centroids are shown in Fig. 4.3(E).

Iteration 3: The same operations are performed as in the first two iterations. Since there is no re-assignment of data points to different clusters in this iteration, the algorithm ends.

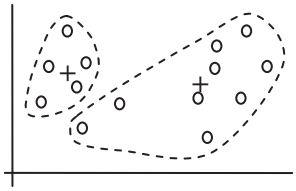
The final clusters are those given in Fig. 4.3(G). The set of data points in each cluster and its centroid are output to the user.



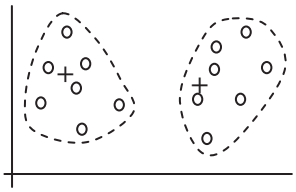
(A). Random selection of k seeds (or centroids)



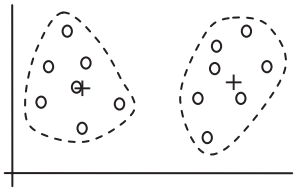
Iteration 1: (B). Cluster assignment



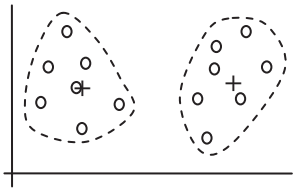
(C). Re-compute centroids



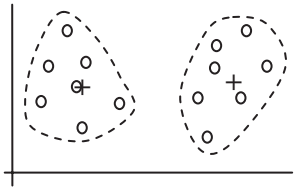
Iteration 2: (D). Cluster assignment



(E). Re-compute centroids



Iteration 3: (F). Cluster assignment



(G). Re-compute centroids

Fig. 4.3. The working of the k-means algorithm through an example ■

One problem with the k-means algorithm is that some clusters may become empty during the clustering process since no data point is assigned to them. Such clusters are called **empty clusters**. To deal with an empty cluster, we can choose a data point as the replacement centroid, e.g., a data point that is furthest from the centroid of a large cluster. If the sum of the squared error (SSE) is used as the stopping criterion, the cluster with the largest squared error may be used to find another centroid.

4.2.2 Disk Version of the K-means Algorithm

The *k*-means algorithm may be implemented in such a way that it does not need to load the entire data set into the main memory, which is useful for large data sets. Notice that the centroids for the *k* clusters can be computed incrementally in each iteration because the summation in Equation (2) can be calculated separately first. During the clustering process, the number of data points in each cluster can be counted incrementally as well. This gives us a disk based implementation of the algorithm (Fig. 4.4), which produces exactly the same clusters as that in Fig. 4.2, but with the data on disk. In each for-loop, the algorithm simply scans the data once.

The whole clustering process thus scans the data *t* times, where *t* is the number of iterations before convergence, which is usually not very large (< 50). In applications, it is quite common to set a limit on the number of iterations because later iterations typically result in only minor changes to the clusters. Thus, this algorithm may be used to cluster large data sets which cannot be loaded into the main memory. Although there are several special algorithms that scale-up clustering algorithms to large data sets, they all require sophisticated techniques.

Algorithm disk-*k*-means(*k*, *D*)
1 Choose *k* data points as the initial centriods $\mathbf{m}_j$, $j = 1, \ldots, k$;
2 **repeat**
3 initialize $\mathbf{s}_j \leftarrow \mathbf{0}$, $j = 1, \ldots, k$; // **0** is a vector with all 0's
4 initialize $n_j \leftarrow 0$, $j = 1, \ldots, k$; // n_j is the number of points in cluster j
5 **for** each data point $\mathbf{x} \in D$ **do**
6 $j \leftarrow \arg\min_{i \in \{1,2,\ldots k\}} dist(\mathbf{x}, \mathbf{m}_i)$;
7 assign **x** to the cluster *j*;
8 $\mathbf{s}_j \leftarrow \mathbf{s}_j + \mathbf{x}$;
9 $n_j \leftarrow n_j + 1$;
10 **endfor**
11 $\mathbf{m}_j \leftarrow \mathbf{s}_j / n_j$, $j = 1, \ldots, k$;
12 **until** the stopping criterion is met

Fig. 4.4. A simple disk version of the *k*-means algorithm

Let us give some explanations of this algorithm. Line 1 does exactly the same thing as the algorithm in Fig. 4.2. Line 3 initializes vector $\mathbf{s}_j$ which is used to incrementally compute the sum in Equation (2) (line 8). Line 4 initializes n_j which records the number of data points assigned to cluster *j* (line 9). Lines 6 and 7 perform exactly the same tasks as lines 4 and 5 in the original algorithm in Fig. 4.2. Line 11 re-computes the centroids,

which are used in the next iteration. Any of the three stopping criteria may be used here. If the sum of squared error is applied, we can modify the algorithm slightly to compute the sum of square error incrementally.

4.2.3 Strengths and Weaknesses

The main strengths of the *k*-means algorithm are its simplicity and efficiency. It is easy to understand and easy to implement. Its time complexity is $O(tkn)$, where n is the number of data points, k is the number of clusters, and t is the number of iterations. Since both k and t are normally much smaller than n. The *k*-means algorithm is considered a linear algorithm in the number of data points.

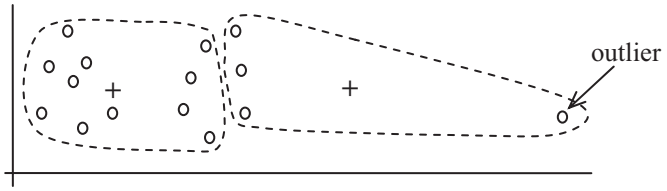
The weaknesses and ways to address them are as follows:

1. The algorithm is only applicable to data sets where the notion of the **mean** is defined. Thus, it is difficult to apply to categorical data sets. There is, however, a variation of the *k*-means algorithm called ***k*-modes**, which clusters categorical data. The algorithm uses the mode instead of the mean as the centroid. Assuming that the data instances are described by r categorical attributes, the mode of a cluster C_j is a tuple $\mathbf{m}_j = (m_{j1}, m_{j2}, \ldots, m_{jr})$ where m_{ji} is the most frequent value of the ith attribute of the data instances in cluster C_j. The similarity (or distance) between a data instance and a mode is the number of values that they match (or do not match).
2. The user needs to specify the number of clusters k in advance. In practice, several k values are tried and the one that gives the most desirable result is selected. We will discuss the evaluation of clusters later.
3. The algorithm is sensitive to **outliers**. Outliers are data points that are very far away from other data points. Outliers could be errors in the data recording or some special data points with very different values. For example, in an employee data set, the salary of the Chief-Executive-Officer (CEO) of the company may be considered as an outlier because its value could be many times larger than everyone else. Since the *k*-means algorithm uses the mean as the centroid of each cluster, outliers may result in undesirable clusters as the following example shows.

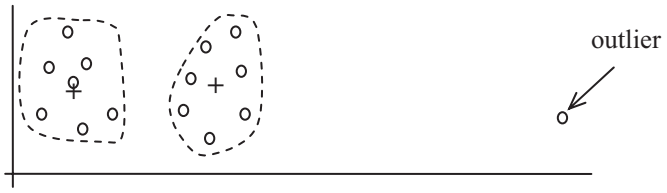
   Example 5: In Fig. 4.5(A), due to an outlier data point, the two resulting clusters do not reflect the natural groupings in the data. The ideal clusters are shown in Fig. 4.5(B). The outlier should be identified and reported to the user. ■

   There are several methods for dealing with outliers. One simple method is to remove some data points in the clustering process that are

much further away from the centroids than other data points. To be safe, we may want to monitor these possible outliers over a few iterations and then decide whether to remove them. It is possible that a very small cluster of data points may be outliers. Usually, a threshold value is used to make the decision.



(A): Undesirable clusters



(B): Ideal clusters

Fig. 4.5. Clustering with and without the effect of outliers ■

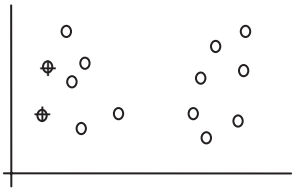
Another method is to perform random sampling. Since in sampling we only choose a small subset of the data points, the chance of selecting an outlier is very small. We can use the sample to do a pre-clustering and then assign the rest of the data points to these clusters, which may be done in any of the three ways below:

- Assign each remaining data point to the centroid closest to it. This is the simplest method.
- Use the clusters produced from the sample to perform supervised learning (classification). Each cluster is regarded as a class. The clustered sample is thus treated as the training data for learning. The resulting classifier is then applied to classify the remaining data points into appropriate classes or clusters.
- Use the clusters produced from the sample as seeds to perform **semi-supervised learning**. Semi-supervised learning is a new learning model that learns from a small set of labeled examples (with classes) and a large set of unlabeled examples (without classes). In our case, the clustered sample data are used as the labeled set and the remaining data points are used as the unlabeled set. The results of the learn-

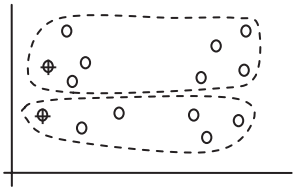
ing naturally cluster all the remaining data points. We will study this technique in the next chapter.

4. The algorithm is sensitive to **initial seeds**, which are the initially selected centroids. Different initial seeds may result in different clusters. Thus, if the sum of squared error is used as the stopping criterion, the algorithm only achieves **local optimal**. The global optimal is computationally infeasible for large data sets.

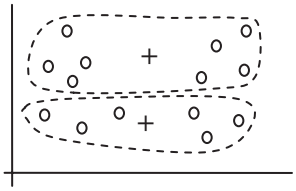
Example 6: Figure 4.6 shows the clustering process of a 2-dimensional data set. The goal is to find two clusters. The randomly selected initial seeds are marked with crosses in Fig. 4.6(A). Figure 4.6(B) gives the clustering result of the first iteration. Figure 4.6(C) gives the result of the second iteration. Since there is no re-assignment of data points, the algorithm stops.



(A). Random selection of seeds (centroids)



(B). Iteration 1



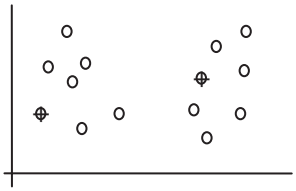
(C). Iteration 2

Fig. 4.6. Poor initial seeds (centroids)

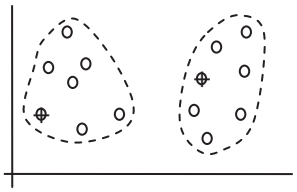
If the initial seeds are different, we may obtain entirely different clusters as Fig. 4.7 shows. Figure 4.7 uses the same data as Fig. 4.6, but different initial seeds (Fig. 4.7(A)). After two iterations, the algorithm ends, and the final clusters are given in Fig. 4.7(C). These two clusters are more reasonable than the two clusters in Fig. 4.6(C), which indicates that the choice of the initial seeds in Fig. 4.6(A) is poor.

To select good initial seeds, researchers have proposed several methods. One simple method is to first compute the mean **m** (the centroid) of the entire data set (any random data point rather than the mean can be

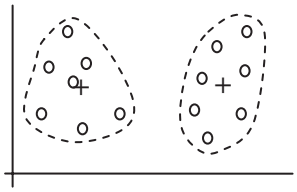
used as well). Then the first seed data point $\mathbf{x}_1$ is selected to be the furthest from the mean $\mathbf{m}$. The second data point $\mathbf{x}_2$ is selected to be the furthest from $\mathbf{x}_1$. Each subsequent data point $\mathbf{x}_i$ is selected such that the sum of distances from $\mathbf{x}_i$ to those already selected data points is the largest. However, if the data has outliers, the method will not work well. To deal with outliers, again, we can randomly select a small sample of the data and perform the same operation on the sample. As we discussed above, since the number of outliers is small, the chance that they show up in the sample is very small.



(A). Random selection of k seeds (centroids)



(B). Iteration 1



(C). Iteration 2

Fig. 4.7. Good initial seeds (centroids) ■

Another method is to sample the data and use the sample to perform hierarchical clustering, which we will discuss in Sect. 4.4. The centroids of the resulting k clusters are used as the initial seeds.

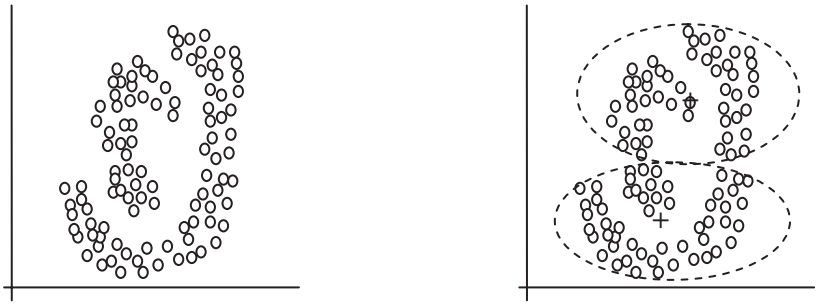
Yet another approach is to manually select seeds. This may not be a difficult task for text clustering applications because it is easy for human users to read some documents and pick some good seeds. These seeds may help improve the clustering result significantly and also enable the system to produce clusters that meet the user's needs.

5. The k-means algorithm is not suitable for discovering clusters that are not hyper-ellipsoids (or hyper-spheres).

Example 7: Figure 4.8(A) shows a 2-dimensional data set. There are two irregular shaped clusters. However, the two clusters are not hyper-

ellipsoids, which means that the *k*-means algorithm will not be able to find them. Instead, it may find the two clusters shown in Fig. 4.8(B).

The question is: are the two clusters in Fig. 4.8(B) necessarily bad? The answer is no. It depends on the application. It is not true that a clustering algorithm that is able to find arbitrarily shaped clusters is always better. We will discuss this issue in Sect. 4.3.2.



(A): Two natural clusters (B): *k*-means clusters

Fig. 4.8. Natural (but irregular) clusters and *k*-means clusters ■

Despite these weaknesses, *k*-means is still the most popular algorithm in practice due to its simplicity, efficiency and the fact that other clustering algorithms have their own lists of weaknesses. There is no clear evidence showing that any other clustering algorithm performs better than the *k*-means algorithm in general, although it may be more suitable for some specific types of data or applications than *k*-means. Note also that comparing different clustering algorithms is a very difficult task because unlike supervised learning, nobody knows what the correct clusters are, especially in high dimensional spaces. Although there are several cluster evaluation methods, they all have drawbacks. We will discuss the evaluation issue in Sect. 4.9.

4.3 Representation of Clusters

Once a set of clusters is found, the next task is to find a way to represent the clusters. In some applications, outputting the set of data points that makes up the cluster to the user is sufficient. However, in other applications that involve decision making, the resulting clusters need to be represented in a compact and understandable way, which also facilitates the evaluation of the resulting clusters.

4.3.1 Common Ways of Representing Clusters

There are three main ways to represent clusters:

1. Use the centroid of each cluster to represent the cluster. This is the most popular way. The centroid tells where the center of the cluster is. One may also compute the radius and standard deviation of the cluster to determine the spread in each dimension. The centroid representation alone works well if the clusters are of the hyper-spherical shape. If clusters are elongated or are of other shapes, centroids may not be suitable.
2. Use classification models to represent clusters. In this method, we treat each cluster as a class. That is, all the data points in a cluster are regarded as having the same class label, e.g., the cluster ID. We then run a supervised learning algorithm on the data to find a classification model. For example, we may use the decision tree learning to distinguish the clusters. The resulting tree or set of rules provide an understandable representation of the clusters.

   Figure 4.9 shows a partitioning produced by a decision tree algorithm. The original clustering gave three clusters. Data points in cluster 1 are represented by 1's, data points in cluster 2 are represented by 2's, and data points in cluster 3 are represented by 3's. We can see that the three clusters are separated and each can be represented with a rule.

   $x \leq 2 \rightarrow$ cluster 1
   $x > 2, y > 1.5 \rightarrow$ cluster 2
   $x > 2, y \leq 1.5 \rightarrow$ cluster 3

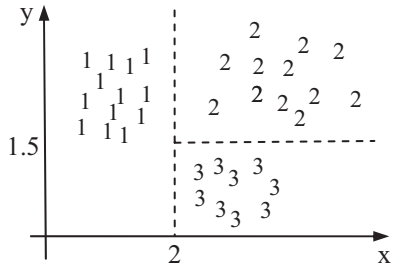


Fig. 4.9. Description of clusters using rules

We make two remarks about this representation method:

- The partitioning in Fig. 4.9 is an ideal case as each cluster is represented by a single rectangle (or rule). However, in most applications, the situation may not be so ideal. A cluster may be split into a few

hyper-rectangles or rules. However, there is usually a dominant or large rule which covers most of the data points in the cluster.
- One can use the set of rules to evaluate the clusters to see whether they conform to some existing domain knowledge or intuition.

3. Use frequent values in each cluster to represent it. This method is mainly for clustering of categorical data (e.g., in the *k*-modes clustering). It is also the key method used in text clustering, where a small set of frequent words in each cluster is selected to represent the cluster.

4.3.2 Clusters of Arbitrary Shapes

Hyper-elliptical and hyper-spherical clusters are usually easy to represent, using their centroids together with spreads (e.g., standard deviations), rules, or a combination of both. However, other arbitrary shaped clusters, like the natural clusters shown in Fig. 4.8(A), are hard to represent especially in high dimensional spaces.

A common criticism about an algorithm like *k*-means is that it is not able to find arbitrarily shaped clusters. However, this criticism may not be as bad as it sounds because whether one type of clustering is desirable or not depends on the application. Let us use the natural clusters in Fig. 4.8(A) to discuss this issue together with an artificial application.

Example 8: Assume that the data shown in Fig. 4.8(A) is the measurement data of people's physical sizes. We want to group people based on their sizes into only two groups in order to mass-produce T-shirts of only 2 sizes (say large and small). Even if the measurement data indicate two natural clusters as in Fig. 4.8(A), it is difficult to use the clusters because we need centroids of the clusters to design T-shirts. The clusters in Fig. 4.8(B) are in fact better because they provide us the centroids that are representative of the surrounding data points. If we use the centroids of the two natural clusters as shown in Fig. 4.10 to make T-shirts, it is clearly inappropriate because they are too near to each other in this case. In general, it does not make sense to define the concept of center or centroid for an irregularly shaped cluster. ■

Note that clusters of arbitrary shapes can be found by neighborhood search algorithms such as some hierarchical clustering methods (see the next section), and density-based clustering methods [164]. Due to the difficulty of representing an arbitrarily shaped cluster, an algorithm that finds such clusters may only output a list of data points in each cluster, which are not as easy to use. These kinds of clusters are more useful in spatial and image processing applications, but less useful in others.

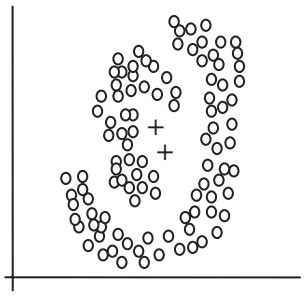


Fig. 4.10. Two natural clusters and their centroids

4.4 Hierarchical Clustering

Hierarchical clustering is another major clustering approach. It has a number of desirable properties which make it popular. It clusters by producing a nested sequence of clusters like a **tree** (also called a **dendrogram**). Singleton clusters (individual data points) are at the bottom of the tree and one root cluster is at the top, which covers all data points. Each internal cluster node contains child cluster nodes. Sibling clusters partition the data points covered by their common parent. Figure 4.11 shows an example.

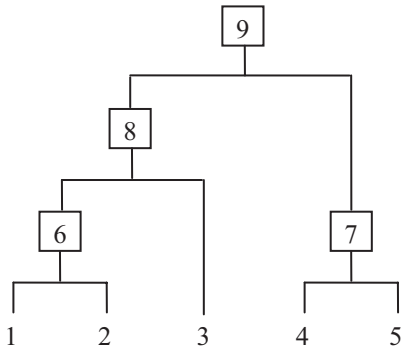


Fig. 4.11. An illustration of hierarchical clustering

At the bottom of the tree, there are 5 clusters (5 data points). At the next level, cluster 6 contains data points 1 and 2, and cluster 7 contains data points 4 and 5. As we move up the tree, we have fewer and fewer clusters. Since the whole clustering tree is stored, the user can choose to view clusters at any level of the tree.

There are two main types of hierarchical clustering methods:

Agglomerative (bottom up) clustering: It builds the dendrogram (tree) from the bottom level, and merges the most similar (or nearest) pair of clusters at each level to go one level up. The process continues until all the data points are merged into a single cluster (i.e., the root cluster).

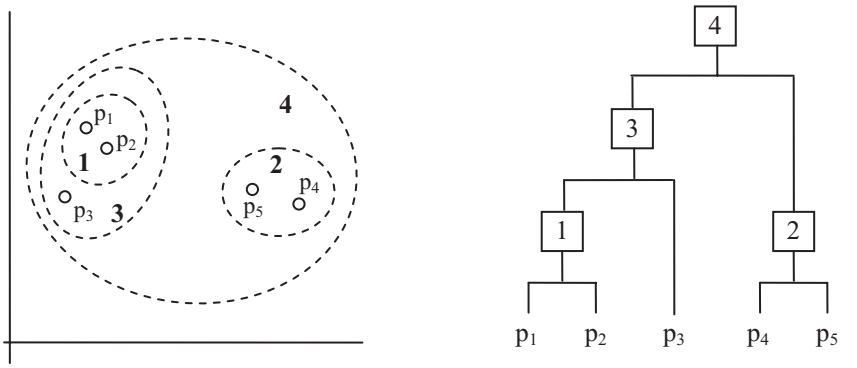
Divisive (top down) clustering: It starts with all data points in one cluster, the root. It then splits the root into a set of child clusters. Each child cluster is recursively divided further until only singleton clusters of individual data points remain, i.e., each cluster with only a single point.

Agglomerative methods are much more popular than divisive methods. We will focus on agglomerative hierarchical clustering. The general agglomerative algorithm is given in Fig. 4.12.

Algorithm Agglomerative(D)
1 Make each data point in the data set D a cluster,
2 Compute all pair-wise distances of $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in D$;
2 **repeat**
3 find two clusters that are nearest to each other;
4 merge the two clusters form a new cluster c;
5 compute the distance from c to all other clusters;
12 **until** there is only one cluster left

Fig. 4.12. The agglomerative hierarchical clustering algorithm

Example 9: Figure 4.13 illustrates the working of the algorithm. The data points are in a 2-dimensional space. Figure 4.13(A) shows the sequence of nested clusters, and Fig. 4.13(B) gives the dendrogram. ■



(A). Nested clusters (B) Dendrogram

Fig. 4.13. The working of an agglomerative hierarchical clustering algorithm

Unlike the *k*-means algorithm, which uses only the centroids in distance computation, hierarchical clustering may use anyone of several methods to determine the distance between two clusters. We introduce them next.

4.4.1 Single-Link Method

In **single-link** (or **single linkage**) hierarchical clustering, the distance between two clusters is the distance between two closest data points in the two clusters (one data point from each cluster). In other words, the single-link clustering merges the two clusters in each step whose two nearest data points (or members) have the smallest distance, i.e., the two clusters with the **smallest minimum** pair-wise distance. The single-link method is suitable for finding non-elliptical shape clusters. However, it can be sensitive to noise in the data, which may cause the **chain effect** and produce straggly clusters. Figure 4.14 illustrates this situation. The noisy data points (represented with filled circles) in the middle connect two natural clusters and split one of them.

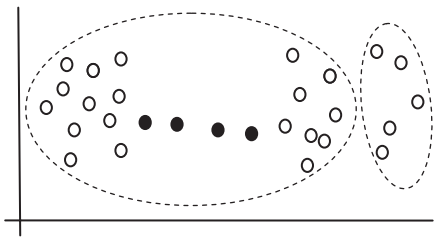


Fig. 4.14. The chain effect of the single-link method

With suitable data structures, single-link hierarchical clustering can be done in $O(n^2)$ time, where n is the number of data points. This is much slower than the *k*-means method, which performs clustering in linear time.

4.4.2 Complete-Link Method

In **complete-link** (or **complete linkage**) clustering, the distance between two clusters is the **maximum** of all pair-wise distances between the data points in the two clusters. In other words, the complete-link clustering merges the two clusters in each step whose two furthest data points have the smallest distance, i.e., the two clusters with the **smallest maximum** pair-wise distance. Figure 4.15 shows the clusters produced by complete-link clustering using the same data as in Fig. 4.14.

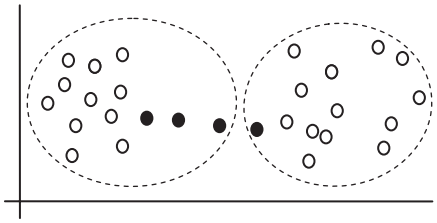


Fig. 4.15. Clustering using the complete-link method

Although the complete-link method does not have the problem of chain effects, it can be sensitive to outliers. Despite this limitation, it has been observed that the complete-link method usually produces better clusters than the single-link method. The worse case time complexity of the complete-link clustering is $O(n^2 \log n)$, where n is the number of data points.

4.4.3 Average-Link Method

This is a compromise between the sensitivity of complete-link clustering to outliers and the tendency of single-link clustering to form long chains that do not correspond to the intuitive notion of clusters as compact, spherical objects. In this method, the distance between two clusters is the average distance of all pair-wise distances between the data points in two clusters. The time complexity of this method is also $O(n^2 \log n)$.

Apart from the above three popular methods, there are several others. The following two methods are also commonly used:

Centroid method: In this method, the distance between two clusters is the distance between their centroids.

Ward's method: In this method, the distance between two clusters is defined as the increase in the sum of squared error (distances) from that of two clusters to that of one merged cluster. Thus, the clusters to be merged in the next step are the ones that will increase the sum the least. Recall that the sum of squared error (SSE) is one of the measures used in the *k*-means clustering (Equation (1)).

4.4.4. Strengths and Weaknesses

Hierarchical clustering has several advantages compared to the *k*-means and other partitioning clustering methods. It is able to take any form of distance or similarity function. Moreover, unlike the *k*-means algorithm which only gives *k* clusters at the end, the hierarchy of clusters from hier-

archical clustering enables the user to explore clusters at any level of detail (or granularity). In many applications, this resulting hierarchy can be very useful in its own right. For example, in text document clustering, the cluster hierarchy may represent a topic hierarchy in the documents.

Some studies have shown that agglomerative hierarchical clustering often produces better clusters than the *k*-means method. It can also find clusters of arbitrary shapes, e.g., using the single-link method.

Hierarchical clustering also has several weaknesses. As we discussed with the individual methods, the single-link method may suffer from the chain effect, and the complete-link method is sensitive to outliers. The main shortcomings of all hierarchical clustering methods are their computation complexities and space requirements, which are at least quadratic. Compared to the *k*-means algorithm, this is very inefficient and not practical for large data sets. One can use sampling to deal with the problems. A small sample is taken to do clustering and then the rest of the data points are assigned to each cluster either by distance comparison or by supervised learning (see Sect. 4.3.1). Some **scale-up methods** may also be applied to large data sets. The main idea of the scale-up methods is to find many small clusters first using an efficient algorithm, and then to use the centroids of these small clusters to represent the clusters to perform the final hierarchical clustering (see the BIRCH method in [610]).

4.5 Distance Functions

Distance or similarity functions play a central role in all clustering algorithms. Numerous distance functions have been reported in the literature and used in applications. Different distance functions are also used for different types of attributes (also called **variables**).

4.5.1 Numeric Attributes

The most commonly used distance functions for numeric attributes are the **Euclidean distance** and **Manhattan** (**city block**) **distance**. Both distance measures are special cases of a more general distance function called the **Minkowski distance**. We use $dist(\mathbf{x}_i, \mathbf{x}_j)$ to denote the distance between two data points of r dimensions. The Minkowski distance is:

$$dist(\mathbf{x}_i, \mathbf{x}_j) = (|x_{i1} - x_{j1}|^h + |x_{i2} - x_{j2}|^h + ... + |x_{ir} - x_{jr}|^h)^{\frac{1}{h}}, \quad (4)$$

where h is a positive integer.

If $h = 2$, it is the **Euclidean distance**,

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{(x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2 + ... + (x_{ir} - x_{jr})^2}. \quad (5)$$

If $h = 1$, it is the **Manhattan distance**,

$$dist(\mathbf{x}_i, \mathbf{x}_j) = | x_{i1} - x_{j1} | + | x_{i2} - x_{j2} | + ... + | x_{ir} - x_{jr} |. \quad (6)$$

Other common distance functions include:

Weighted Euclidean distance: A weight is associated with each attribute to express its importance in relation to other attributes.

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{w_1(x_{i1} - x_{j1})^2 + w_2(x_{i2} - x_{j2})^2 + ... + w_r(x_{ir} - x_{jr})^2}. \quad (7)$$

Squared Euclidean distance: the standard Euclidean distance is squared in order to place progressively greater weights on data points that are further apart. The distance is

$$dist(\mathbf{x}_i, \mathbf{x}_j) = (x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2 + ... + (x_{ir} - x_{jr})^2. \quad (8)$$

Chebychev distance: This distance measure is appropriate in cases where one wants to define two data points as "different" if they are different on any one of the attributes. The Chebychev distance is

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \max(| x_{i1} - x_{j1} |, | x_{i2} - x_{j2} |, ..., | x_{ir} - x_{jr} |). \quad (9)$$

4.5.2 Binary and Nominal Attributes

The above distance measures are only appropriate for numeric attributes. For **binary** and **nominal** attributes (also called **unordered categorical** attributes), we need different functions. Let us discuss binary attributes first.

A **binary attribute** has two states or values, usually represented by 1 and 0. The two states have no numerical ordering. For example, Gender has two values, male and female, which have no ordering relations but are just different. Existing distance functions for binary attributes are based on the proportion of value matches in two data points. A match means that, for a particular attribute, both data points have the same value. It is easy to use a **confusion matrix** to introduce these measures. Given the *i*th and *j*th data points, $\mathbf{x}_i$ and $\mathbf{x}_j$, we can construct the confusion matrix in Fig. 4.16.

To give the distance functions, we further divide binary attributes into **symmetric** and **asymmetric** attributes. For different types of attributes, different distance functions need to be used [271]:

		Data point $\mathbf{x}_j$ 1	0	
Data point $\mathbf{x}_i$	1	a	b	$a+b$
	0	c	d	$c+d$
		$a+c$	$b+d$	$a+b+c+d$

a: the number of attributes with the value of 1 for both data points.
b: the number of attributes for which $x_{if} = 1$ and $x_{jf} = 0$, where x_{if} (x_{jf}) is the value of the fth attribute of the data point $\mathbf{x}_i$ ($\mathbf{x}_j$).
c: the number of attributes for which $x_{if} = 0$ and $x_{jf} = 1$.
d: the number of attributes with the value of 0 for both data points.

Fig. 4.16. Confusion matrix of two data points with only binary attributes

Symmetric attributes: A binary attribute is **symmetric** if both of its states (0 and 1) have equal importance, and carry the same weight, e.g., male and female of the attribute Gender. The most commonly used distance function for symmetric attributes is the **simple matching distance**, which is the proportion of mismatches (Equation (10)) of their values. We assume that every attribute in the data set is a symmetric attribute.

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{b+c}{a+b+c+d} \tag{10}$$

We can also weight some components in Equation (10) according to application needs. For example, we may want mismatches to carry twice the weight of matches, or vice versa:

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{2(b+c)}{a+d+2(b+c)} \tag{11}$$

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{b+c}{2(a+d)+b+c} \tag{12}$$

Example 10: Given the following two data points, where each attribute is a symmetric binary attribute,

$\mathbf{x}_1$	1	1	1	0	1	0	0
$\mathbf{x}_2$	0	1	1	0	0	1	0

the distance computed based on the simple matching distance is

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{2+1}{2+2+1+2} = \frac{3}{7} = 0.429. \tag{13}$$

■

Asymmetric attributes: A binary attribute is asymmetric if one of the states is more important or valuable than the other. By convention, we use state 1 to represent the more important state, which is typically the rare or infrequent state. The most commonly used distance measure for asymmetric attributes is the **Jaccard distance**:

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{b+c}{a+b+c}. \tag{14}$$

Similarly, we can vary the Jaccard distance by giving more weight to (b+c) or more weight to a to express different emphases.

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{2(b+c)}{a+2(b+c)}. \tag{15}$$

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{b+c}{2a+b+c}. \tag{16}$$

Note that there is also a **Jaccard coefficient**, which measures similarity (rather than distance) and is defined as a / (a+b+c).

For general **nominal attributes** with more than two states or values, the commonly used distance measure is also based on the simple matching distance. Given two data points $\mathbf{x}_i$ and $\mathbf{x}_j$, let the number of attributes be r, and the number of values that match in $\mathbf{x}_i$ and $\mathbf{x}_j$ be q:

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{r-q}{r}. \tag{17}$$

As that for binary attributes, we can give higher weights to different components in Equation (17) according to different application characteristics.

4.5.3 Text Documents

Although a text document consists of a sequence of sentences and each sentence consists of a sequence of words, a document is usually considered as a "bag" of words in document clustering. The sequence and the position information of words are ignored. Thus a document can be represented as a vector just like a normal data point. However, we use similarity to compare two documents rather than distance. The most commonly used simi-

larity function is the **cosine similarity**. We will study this similarity measure in Sect. 6.2.2 when we discuss information retrieval and Web search.

4.6 Data Standardization

One of the most important steps in data pre-processing for clustering is to standardize the data. For example, using the Euclidean distance, standardization of attributes is highly recommended so that all attributes can have equal impact on the distance computation. This is to avoid obtaining clusters that are dominated by attributes with the largest amounts of variation.

Example 11: In a 2-dimensional data set, the value range of one attribute is from 0 to 1, while the value range of the other attribute is from 0 to 1000. Consider the following pair of data points $\mathbf{x}_i$: (0.1, 20) and $\mathbf{x}_j$: (0.9, 720). The Euclidean distance between the two points is

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{(0.9 - 0.1)^2 + (720 - 20)^2} = 700.000457, \tag{18}$$

which is almost completely dominated by (720–20) = 700. To deal with the problem, we standardize the attributes, e.g., to force the attributes to have a common value range. If both attributes are forced to have a scale within the range 0–1, the values 20 and 720 become 0.02 and 0.72. The distance on the first dimension becomes 0.8 and the distance on the second dimension 0.7, which are more equitable. Then, $dist(\mathbf{x}_i, \mathbf{x}_j) = 1.063$. ■

This example shows that standardizing attributes is important. In fact, different types of attributes require different treatments. We list these treatments below.

Interval-scaled attributes: These are numeric/continuous attributes. Their values are real numbers following a linear scale. Examples of such attributes are age, height, weight, cost, etc. The idea is that intervals keep the same importance through out the scale. For example, the difference in age between 10 and 20 is the same as that between 40 and 50.

There are two main approaches to standardize interval scaled attributes, **range** and ***z*-score**. The range method divides each value by the range of valid values of the attribute so that the transformed value ranges between 0 and 1. Given the value x_{if} of the fth attribute of the ith data point, the new value $rg(x_{if})$ is,

$$rg(x_{if}) = \frac{x_{if} - \min(f)}{\max(f) - \min(f)}, \tag{19}$$

where min(f) and max(f) are the minimum value and maximum value of attribute f respectively. max(f) − min(f) is the value range of the valid values of attribute f.

The z-score method transforms an attribute value based on the mean and the standard deviation of the attribute. That is, the z-score of the value indicates how far and in what direction the value deviates from the mean of the attribute, expressed in units of the standard deviation of the attribute. The standard deviation of attribute f, denoted by σ_f, is computed with:

$$\sigma_f = \sqrt{\frac{\sum_{i=1}^{n}(x_{if} - \mu_f)^2}{n-1}}, \tag{20}$$

where n is the number of data points in the data set, x_{if} is the same as above, and μ_f is the mean of attribute f, which is computed with:

$$\mu_f = \frac{1}{n}\sum_{i=1}^{n} x_{if}. \tag{21}$$

Given the value x_{if}, its z-score (the new value after transformation) is $z(x_{if})$,

$$z(x_{if}) = \frac{x_{if} - \mu_f}{\sigma_f}. \tag{22}$$

Ratio-Scaled Attributes: These are also numeric attributes taking real values. However, unlike interval-scaled attributes, their scales are not linear. For example, the total amount of microorganisms that evolve in a time t is approximately given by

$$Ae^{Bt},$$

where A and B are some positive constants. This formula is referred to as exponential growth. If we have such attributes in a data set for clustering, we have one of the following two options:

1. Treat it as an interval-scaled attribute. This is often not recommended due to scale distortion.
2. Perform logarithmic transformation to each value, x_{if}, i.e.,

$$\log(x_{if}). \tag{23}$$

   After the transformation, the attribute can be treated as an interval-scaled attribute.

Nominal (Unordered Categorical) Attributes: As we discussed in Sect. 4.5.2, the value of such an attribute can take anyone of a set of states (also

called categories). The states have no logical or numerical ordering. For example, the attribute *fruit* may have the possible values, Apple, Orange, and Pear, which have no ordering. A **binary attribute** is a special case of a nominal attribute with only two states or values.

Although nominal attributes are not standardized as numeric attributes, it is sometime useful to convert a nominal attribute to a set of binary attributes. Let the number of values of a nominal attribute be v. We can then create v binary attributes to represent them, i.e., one binary attribute for each value. If a data instance for the nominal attribute takes a particular value, the value of its corresponding binary attribute is set to 1, otherwise it is set to 0. The resulting binary attributes can be used as numeric attributes. We will discuss this again in Sect. 4.7.

Example 12: For the nominal attribute *fruit*, we create three binary attributes called, Apple, Orange, and Pear in the new data. If a particular data instance in the original data has Apple as the value for *fruit*, then in the transformed data, we set the value of the attribute Apple to 1, and the values of attributes Orange and Pear to 0. ■

Ordinal (Ordered Categorical) Attributes: An ordinal attribute is like a nominal attribute, but its values have a numerical ordering. For example, the age attribute may have the values, Young, Middle-Age and Old. The common approach to distance computation is to treat ordinal attributes as interval-scaled attributes and use the same methods as for interval-scaled attributes to standardize the values of ordinal attributes.

4.7 Handling of Mixed Attributes

So far, we have assumed that a data set contains only one type of attributes. However, in practice, a data set may contain mixed attributes. That is, it may contain any subset of the six types of attributes, **interval-scaled**, **symmetric binary**, **asymmetric binary**, **ratio-scaled**, **ordinal** and **nominal** attributes. Clustering a data set involving mixed attributes is a challenging problem.

One way to deal with such a data set is to choose a dominant attribute type and then convert the attributes of other types to this type. For example, if most attributes in a data set are interval-scaled, we can convert ordinal attributes and ratio-scaled attributes to interval-scaled attributes as discussed above. It is also appropriate to treat symmetric binary attributes as interval-scaled attributes. However, it does not make much sense to convert a nominal attribute with more than two values or an asymmetric binary attribute to an interval-scaled attribute, but it is still frequently done in

practice by assigning some numbers to them according to some hidden ordering. For instance, in the example of Apple, Orange, and Pear, one may order them according to their prices, and thus make the attribute *fruit* an ordinal attribute or even an interval-scaled attribute. In the previous section, we also saw that a nominal attribute can be converted to a set of (symmetric) binary attributes, which in turn can be regarded as interval-scaled attributes.

Another method of handling mixed attributes is to compute the distance of each attribute of the two data points separately and then combine all the individual distances to produce an overall distance. We describe one such method, which is due to Gower [205] and is also described in [218, 271]. We describe the combination formula first (Equation (24)) and then present the methods to compute individual distances.

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{\sum_{f=1}^{r} \delta_{ij}^f d_{ij}^f}{\sum_{f=1}^{r} \delta_{ij}^f}. \tag{24}$$

This distance value is between 0 and 1. r is the number of attributes in the data set. The indicator δ_{ij}^f is 1 if both values x_{if} and x_{jf} for attribute f are non-missing, and it is set to 0 otherwise. It is also set to 0 if attribute f is asymmetric and the match is 0–0. Equation (24) cannot be computed if all δ_{ij}^f's are 0. In such a case, some default value may be used or one of the data points is removed. d_{ij}^f is the distance contributed by attribute f, and it is in the range 0–1. If f is a binary or nominal attribute,

$$d_{ij}^f = \begin{cases} 1 & if\ x_{if} \neq x_{jf} \\ 0 & otherwise \end{cases} \tag{25}$$

If all the attributes are nominal, Equation (24) reduces to Equation (17). The same is true for symmetric binary attributes, in which we recover the simple matching distance (Equation (10)). When the attributes are all asymmetric, we obtain the Jaccard distance (Equation (14)).

If attribute f is interval-scaled, we use

$$d_{ij}^f = \frac{|\, x_{if} - x_{jf} \,|}{R_f} \tag{26}$$

where R_f is the value range of attribute f, which is

$$R_f = \max(f) - \min(f) \tag{27}$$

Ordinal attributes and ratio-scaled attributes are handled in the same way after conversion.

If all the attributes are interval-scaled, Equation (24) becomes the Manhattan distance assuming that all attribute values are standardized by dividing their values with the ranges of their corresponding attributes.

4.8 Which Clustering Algorithm to Use?

Clustering research and application has a long history. Over the years, a vast collection of clustering algorithms has been designed. This chapter only introduced several of the main algorithms.

Given an application data set, choosing the "best" clustering algorithm to cluster the data is a challenge. Every clustering algorithm has limitations and works well with only certain data distributions. However, it is very hard, if not impossible, to know what distribution the application data follows. Worse still, the application data set may not fully follow any "ideal" structure or distribution required by the algorithms. Apart from choosing a suitable clustering algorithm from a large collection of available algorithms, deciding how to standardize the data, to choose a suitable distance function and to select other parameter values (e.g., k in the k-means algorithm) are complex as well. Due to these complexities, the common practice is to run several algorithms using different distance functions and parameter settings, and then to carefully analyze and compare the results.

The interpretation of the results should be based on insight into the meaning of the original data together with knowledge of the algorithms used. That is, it is crucial that the user of a clustering algorithm fully understands the algorithm and its limitations. He/she should also have the domain expertise to examine the clustering results. In many cases, generating cluster descriptions using a supervised learning method (e.g., decision tree induction) can be particularly helpful to the analysis and comparison.

4.9 Cluster Evaluation

After a set of clusters is found, we need to assess the goodness of the clusters. Unlike classification, where it is easy to measure accuracy using labeled test data, for clustering nobody knows what the correct clusters are given a data set. Thus, the quality of a clustering is much harder to evaluate. We introduce a few commonly used evaluation methods below.

User Inspection: A panel of experts is asked to inspect the resulting clusters and to score them. Since this process is subjective, we take the average of the scores from all the experts as the final score of the clustering. This manual inspection is obviously a labor intensive and time consuming task. It is subjective as well. However, in most applications, some level of manual inspection is necessary because no other existing evaluation methods are able to guarantee the quality of the final clusters. It should be noted that direct user inspection may be easy for certain types of data, but not for others. For example, user inspection is not hard for text documents because one can read them easily. However, for a relational table with only numbers, staring at the data instances in each cluster makes no sense. The user can only meaningfully study the centroids of the clusters, or rules that characterize the clusters generated by a decision tree algorithm or some other supervised learning methods (see Sect. 4.3.1).

Ground Truth: In this method, classification data sets are used to evaluate clustering algorithms. Recall that a classification data set has several classes, and each data instance/point is labeled with one class. Using such a data set for cluster evaluation, we make the assumption that each class corresponds to a cluster. For example, if a data set has three classes, we assume that it has three clusters, and we request the clustering algorithm to also produce three clusters. After clustering, we compare the cluster memberships with the class memberships to determine how good the clustering is. A variety of measures can be used to assess the clustering quality, e.g., entropy, purity, precision, recall, and F-score.

To facilitate evaluation, a confusion matrix can be constructed from the resulting clusters. From the matrix, various measurements can be computed. Let the set of classes in the data set D be $C = (c_1, c_2, \ldots, c_k)$. The clustering method also produces k clusters, which partition D into k disjoint subsets, $D_1, D_2, \ldots, D_k$.

Entropy: For each cluster, we can measure its entropy as follows:

$$entropy(D_i) = -\sum_{j=1}^{k} \Pr_i(c_j) \log_2 \Pr_i(c_j), \tag{28}$$

where $\Pr_i(c_j)$ is the proportion of class c_j data points in cluster i or D_i. The total entropy of the whole clustering (which considers all clusters) is

$$entropy_{total}(D) = \sum_{i=1}^{k} \frac{|D_i|}{|D|} \times entropy(D_i). \tag{29}$$

Purity: This measures the extent that a cluster contains only one class of data. The purity of each cluster is computed with

$$purity(D_i) = \max_j(\Pr_i(c_j)). \tag{30}$$

The total purity of the whole clustering (considering all clusters) is

$$purity_{total}(D) = \sum_{i=1}^{k} \frac{|D_i|}{|D|} \times purity(D_i). \tag{31}$$

Precision, recall, and F-score can be computed as well for each cluster based on the class that is the most frequent in the cluster. Note that these measures are based on a single class (see Sect. 3.3.2).

Example 13: Assume we have a text collection D of 900 documents from three topics (or three classes), Science, Sports, and Politics. Each class has 300 documents, and each document is labeled with one of the topics (classes). We use this collection to perform clustering to find three clusters. Class/topic labels are not used in clustering. After clustering, we want to measure the effectiveness of the clustering algorithm.

First, a confusion matrix (Fig. 4.17) is constructed based on the clustering results. From Fig. 4.17, we see that cluster 1 has 250 Science documents, 20 Sports documents, and 10 Politics documents. The entries of the other rows have similar meanings. The last two columns list the entropy and purity values of each cluster and also the total entropy and purity of the whole clustering (last row). We observe that cluster 1, which contains mainly Science documents, is a much better (or purer) cluster than the other two. This fact is also reflected by both their entropy and purity values.

Cluster	Science	Sports	Politics	Entropy	Purity
1	250	20	10	0.589	0.893
2	20	180	80	1.198	0.643
3	30	100	210	1.257	0.617
Total	300	300	300	1.031	0.711

Fig. 4.17. Confusion matrix with entropy and purity values

Obviously, we can use the total entropy or the total purity to compare different clustering results from the same algorithm with different parameter settings or from different algorithms.

Precision and recall may be computed similarly for each cluster. For example, the precision of Science documents in cluster 1 is 0.89. The recall

of Science documents in cluster 1 is 0.83. The F-score for Science documents is thus 0.86. ■

A final remark about this evaluation method is that although an algorithm may perform well on some labeled data sets, there is no guarantee that it will perform well on the actual application data at hand, which has no class labels. However, the fact that it performs well on some labeled data sets does give us some confidence on the quality of the algorithm. This evaluation method is said to be based on **external data** or **information**.

There are also methods that evaluate clusters based on the **internal information** in the clusters (without using external data with class labels). These methods measure **intra-cluster cohesion** (compactness) and **inter-cluster separation** (isolation). Cohesion measures how near the data points in a cluster are to the cluster centroid. Sum of squared error (SSE) is a commonly used measure. Separation measures how far apart different cluster centroids are from one another. Any distance functions can be used for the purpose. We should note, however, that good values for these measurements do not always mean good clusters. In most applications, expert judgments are still the key. Clustering evaluation remains to be a very difficult problem.

Indirect Evaluation: In some applications, clustering is not the primary task. Instead, it is used to help perform another more important task. Then, we can use the performance on the primary task to determine which clustering method is the best for the task. For instance, in a Web usage mining application, the primary task is to recommend books to online shoppers. If the shoppers can be clustered according to their profiles and their past purchasing history, we may be able to provide better recommendations. A few clustering methods can be tried, and their results are then evaluated based on how well they help with the recommendation task. Of course, here we assume that the recommendation results can be reliably evaluated.

4.10 Discovering Holes and Data Regions

In this section, we wander a little to discuss something related but quite different from the preceding algorithms. We show that unsupervised learning tasks may be performed by using supervised learning techniques [350].

In clustering, data points are grouped into clusters according to their distances (or similarities). However, clusters only represent one aspect of the hidden knowledge in data. Another aspect that we have not studied is the **holes**. If we treat data instances as points in an r-dimensional space, a hole

is simply a region in the space that contains no or few data points. The existence of holes is due to the following two reasons:

1. insufficient data in certain areas, and/or
2. certain attribute-value combinations are not possible or seldom occur.

Although clusters are important, holes in the space can be quite useful too. For example, in a disease database we may find that certain symptoms and/or test values do not occur together, or when a certain medicine is used, some test values never go beyond certain ranges. Discovery of such information can be of great importance in medical domains because it could mean the discovery of a cure to a disease or some biological laws.

The technique discussed in this section aims to divide the data space into two types of regions, **data regions** (also called **dense regions**) and **empty regions** (also called **sparse regions**). A data region is an area in the space that contains a concentration of data points and can be regarded as a cluster. An empty region is a **hole**. A supervised learning technique similar to decision tree induction is used to separate the two types of regions. The algorithm (called CLTree for CLuser Tree [350]) works for numeric attributes, but can be extended to discrete or categorical attributes.

Decision tree learning is a popular technique for classifying data of various classes. For a decision tree algorithm to work, we need at least two classes of data. A clustering data set, however, has no class label for each data point. Thus, the technique is not directly applicable. However, the problem can be dealt with by a simple idea.

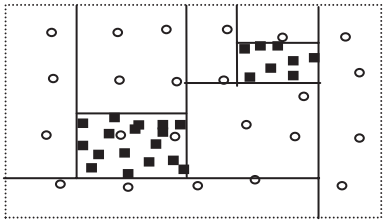
We can regard each data instance/point in the data set as having a class label *Y*. We assume that the data space is uniformly distributed with another type of points, called **non-existing points**, which we will label *N*. With the *N* points added to the original data space, our problem of partitioning the data space into data regions and empty regions becomes a supervised classification problem. The decision tree algorithm can be adapted to solve the problem. Let us use an example to illustrate the idea.

Example 14: Figure 4.18(A) gives a 2-dimensional space with 24 data (*Y*) points. Two data regions (clusters) exist in the space. We then add some uniformly distributed *N* points (represented by "o") to the data space (Fig. 4.18(B)). With the augmented data set, we can run a decision tree algorithm to obtain the partitioning of the space in Fig. 4.18(B). Data regions and empty regions are separated. Each region is a rectangle, which can be expressed as a rule. ■

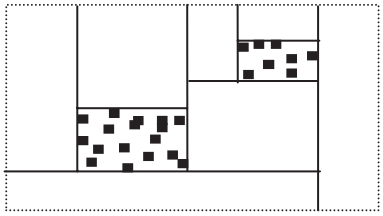
The reason that this technique works is that if there are clusters (or dense data regions) in the data space, the data points cannot be uniformly distributed in the entire space. By adding some uniformly distributed *N*

points, we can isolate data regions because within each data region there are significantly more *Y* points than *N* points. The decision tree technique is well known for this partitioning task.

(A): The original data space



(B). Partitioning with added *N* points



(C). Partitioning without adding *N* points.

Fig. 4.18. Separating data and empty regions using a decision tree

An interesting question is: can the task be performed without physically adding the *N* points to the original data? The answer is yes. Physically adding *N* points increases the size of the data and thus the running time. A more important issue is that it is unlikely that we can have points truly uniformly distributed in a high-dimensional space as we would need an exponential number of them. Fortunately, we do not need to physically add any *N* points. We can compute them when needed. The CLTree method is able to produce the partitioning in Fig. 4.18(C) with no *N* points added. The details are quite involved. Interested readers can refer to [350]. This method has some interesting characteristics:

- It provides descriptions or representations of the resulting data regions and empty regions in terms of hyper-rectangles, which can be expressed as rules as we have seen in Sect. 3.2 of Chap. 3 and in Sect. 4.3.1. Many applications require such descriptions, which can be easily interpreted by users.
- It automatically detects outliers, which are data points in empty regions.
- It may not use all attributes in the data just as in decision tree building

for supervised learning. That is, it can automatically determine what attributes are important and what are not. This means that it can perform subspace clustering, i.e., finding clusters that exist in some subspaces (represented by some subsets of the attributes) of the original space.

This method also has limitations. The main limitation is that data regions of irregular shapes are hard to handle since decision tree learning only generates hyper-rectangles (formed by axis-parallel hyper-planes), which are rules. Hence, an irregularly shaped data or empty region may be split into several hyper-rectangles. Post-processing is needed to join them if desired (see [350] for additional details).

Bibliographic Notes

Clustering or unsupervised learning has a long history and a very large body of work. This chapter described only some widely used core algorithms. Most other algorithms are variations or extensions of these methods. For a comprehensive coverage of clustering, please refer to several books dedicated to clustering, e.g., those by Everitt [167], Hartigan [222], Jain and Dubes [252], Kaufman and Rousseeuw [271], and Mirkin [383]. Most data mining texts also have excellent coverage of clustering techniques, e.g., Han and Kamber [218] and Tan et al. [512], which have influenced the writing of this chapter. Below, we review some more recent developments on clustering and give some further readings.

A density-based clustering algorithm based on local data densities was proposed by Ester et al. [164] and Xu et al. [564] for finding clusters of arbitrary shapes. Hinneburg and Keim [239], Sheikholeslami et al. [485] and Wang et al. [538] proposed several grid-based clustering methods which first partition the space into small grids. A popular neural network clustering algorithm is the Self-Organizing Map (SOM) by Kohonen [287]. Fuzzy clustering (e.g., fuzzy c-means) was studied by Bezdek [50] and Dunn [157]. Cheeseman et al. [94] and Moore [396] studied clustering using mixture models. The method assumes that clusters are a mixture of Gaussians and uses the EM algorithm [127] to learn a mixture density. We will see in Chap. 5 that EM based partially supervised learning algorithms are basically clustering methods with some given initial seeds.

Most clustering algorithms work on numeric data. Categorical data and/or transaction data clustering were investigated by Barbará et al. [36], Ganti et al. [193], Gibson et al. [197], Guha et al. [212], Wang et al. [537], etc. A related area in artificial intelligence is the conceptual clustering, which was studied by Fisher [178], Misha et al. [384] and many others.

UNIT - III
Information Retrieval and Web Search

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.

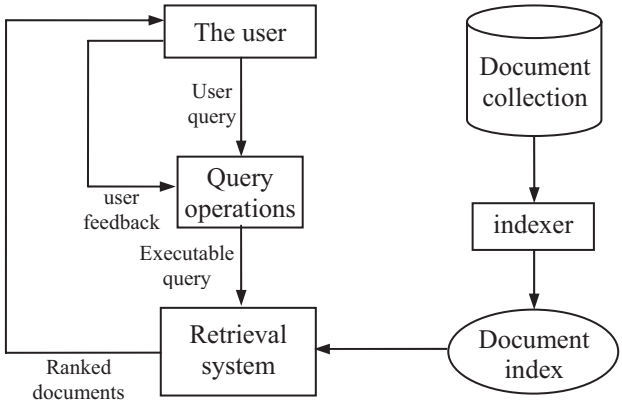


Fig. 6.1. A general IR system architecture

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.,

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

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

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

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, $q = q_1 q_2 \ldots q_m$ and the document collection D be a set of documents, $D = \{d_1, d_2, \ldots, d_N\}$. 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

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

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,

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

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

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

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,

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

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 Roc-

chio 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,

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

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

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

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

$$p(r_i) = \max_{r_i \le r \le r_{10}} p(r). \tag{21}$$

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%

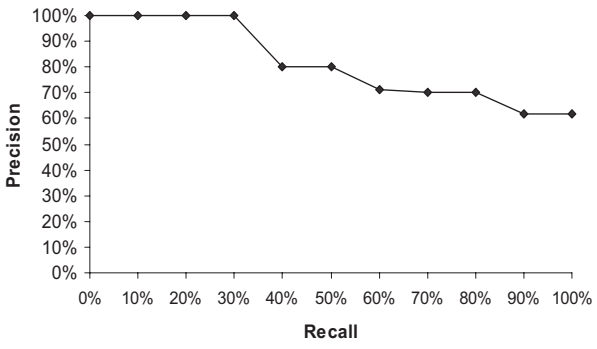


Fig. 6.4. The precision-recall curve ■

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.

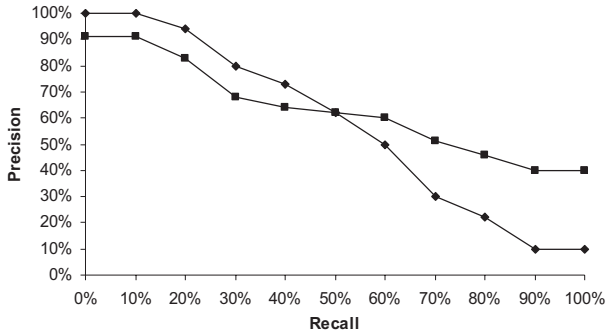


Fig. 6.5. Comparison of two retrieval algorithms based on their precision-recall curves

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:

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

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 prespecified 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-

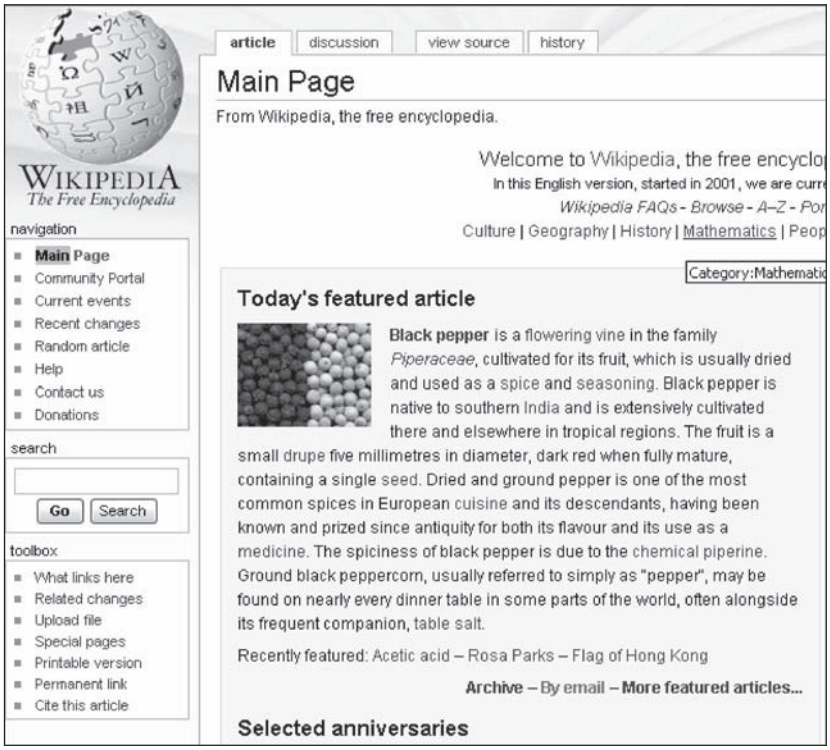


Fig. 6.6. An example of a Web page from Wikipedia

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,

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

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:

id_1: Web mining is useful.
1 2 3 4

id_2: Usage mining applications.
1 2 3

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

id_1: Web mining is useful.
1 2 3 4
id_2: Usage mining applications.
1 2 3
id_3: 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 in-

dices, 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(*klog 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.

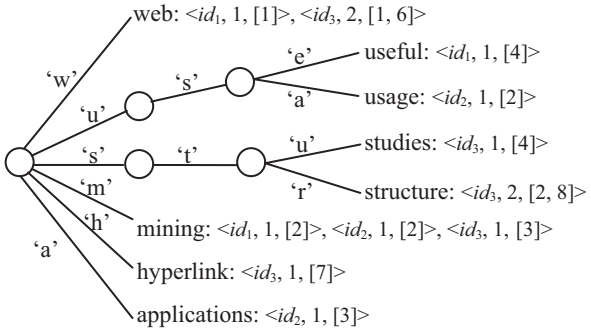


Fig. 6.8. The vocabulary trie and the inverted lists

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,

$$d = 2^{i+1} - b. \tag{25}$$

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.

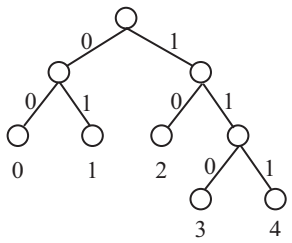


Fig. 6.9. The coding tree for $b = 5$

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.,

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

where

$\boldsymbol{U}$ 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 $\boldsymbol{U}$ are unit orthogonal vectors, i.e., $\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{I}$ (identity matrix).

$\boldsymbol{V}$ 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 $\boldsymbol{V}$ 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 $\boldsymbol{U}$ is in fact an $m \times m$ matrix and $\boldsymbol{V}$ 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 $\boldsymbol{U}$ and the last $n-r$ columns in $\boldsymbol{V}$ can also be dropped.

m is the number of row (terms) in $\boldsymbol{A}$, representing the number of terms.

n is the number of columns in $\boldsymbol{A}$, representing the number of documents.

r is the **rank** of $\boldsymbol{A}$, $r \leq \min(m, n)$.

The singular value decomposition of $\boldsymbol{A}$ always exists and is unique up to

1. allowable permutations of columns of $\boldsymbol{U}$ and $\boldsymbol{V}$ 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 $\boldsymbol{U}$ and $\boldsymbol{V}$ are interchanged.
2. sign (+/–) flip in $\boldsymbol{U}$ and $\boldsymbol{V}$.

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 $\boldsymbol{A}$. 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 $\boldsymbol{A}$ is denoted by $\boldsymbol{A}_k$. We can also reduce the size of the matrices $\boldsymbol{\Sigma}$, $\boldsymbol{U}$ and $\boldsymbol{V}$ by deleting the last $r-k$ rows and columns from $\boldsymbol{\Sigma}$, the last $r-k$ columns in $\boldsymbol{U}$ and the last $r-k$ columns in $\boldsymbol{V}$. We then obtain

$$\boldsymbol{A}_k = \boldsymbol{U}_k \boldsymbol{\Sigma}_k \boldsymbol{V}_k^T, \tag{28}$$

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

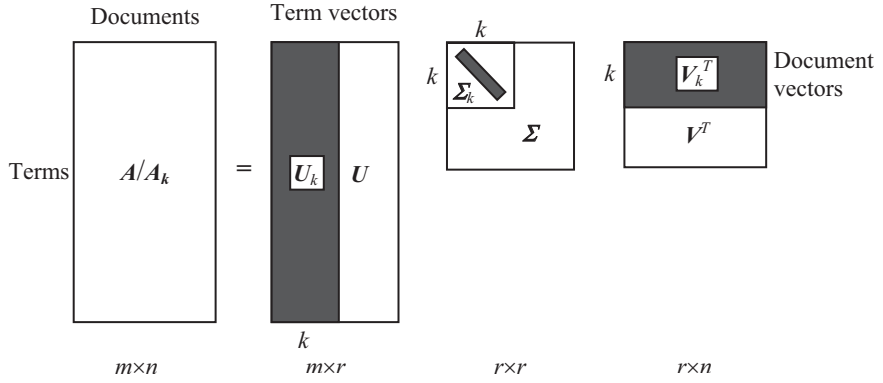


Fig. 6.10. The schematic representation of $\boldsymbol{A}$ and $\boldsymbol{A}_k$

It is critical that the LSI method does not re-construct the original term-document matrix $\boldsymbol{A}$ 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 $\boldsymbol{A}$ 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 $\boldsymbol{U}$, and the direction of y' is represented by the second column vector of $\boldsymbol{U}$. $\boldsymbol{\Sigma V}^T$ represents the documents in the transformed "concept" space. The singular values in $\boldsymbol{\Sigma}$ 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 pro-

jected 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.

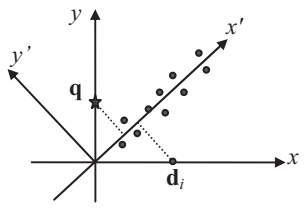


Fig. 6.11. Intuition of the LSI.

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 $1/\sigma_i$ ($1 \le i \le k$), 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 $\boldsymbol{V}_k$ using a similarity measure, e.g., the cosine similarity. Recall that each row of $\boldsymbol{V}_k$ (or each column of $\boldsymbol{V}_k^T$) corresponds to a document (column) in $\boldsymbol{A}$. This method has been used traditionally.

Alternatively, since $\boldsymbol{\Sigma}_k\boldsymbol{V}_k^T$ (not $\boldsymbol{V}_k^T$) represents the documents in the transformed k-concept space, we can compare the similarity of the query document in the transformed space, which is $\boldsymbol{\Sigma}_k\mathbf{q}_k^T$, and each transformed document in $\boldsymbol{\Sigma}_k\boldsymbol{V}_k^T$ for retrieval. The difference between the two methods is obvious. This latter method considers scaling effects of the singular values in $\boldsymbol{\Sigma}_k$, 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 $\boldsymbol{A}$ is given below, which is a 9×12 matrix.

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

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 $\boldsymbol{V}_k^T$:

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

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.

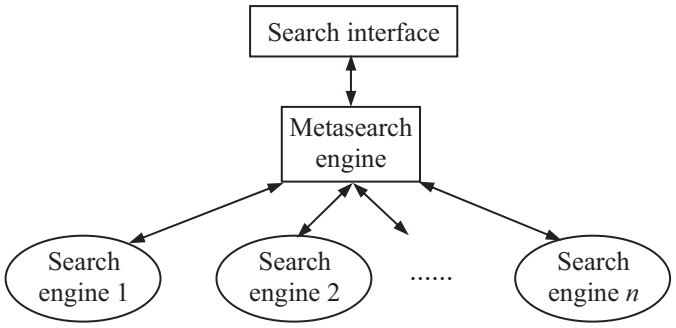


Fig. 6.12. A meta-search architecture

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.

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

CombANZ: It is defined as

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

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

CombMNZ: It is defined as

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

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:

Score(*a*) = 4 + 3 + 2 + 1 + 1.5 = 11.5
Score(*b*) = 3 + 4 + 3 + 3 + 3 = 16
Score(*c*) = 2 + 1 + 4 + 4 + 4 = 15
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, key-

words, 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].

UNIT - IV

Link Analysis and Web Crawling

7 Link Analysis

Early search engines retrieved relevant pages for the user based primarily on the content similarity of the user query and the indexed pages of the search engines. The retrieval and ranking algorithms were simply direct implementation of those from information retrieval. Starting from 1996, it became clear that content similarity alone was no longer sufficient for search due to two reasons. First, the number of Web pages grew rapidly during the middle to late 1990s. Given any query, the number of relevant pages can be huge. For example, given the search query "classification technique", the Google search engine estimates that there are about 10 million relevant pages. This abundance of information causes a major problem for ranking, i.e., how to choose only 30–40 pages and rank them suitably to present to the user. Second, content similarity methods are easily spammed. A page owner can repeat some important words and add many remotely related words in his/her pages to boost the rankings of the pages and/or to make the pages relevant to a large number of possible queries.

Starting from around 1996, researchers in academia and search engine companies began to work on the problem. They resort to hyperlinks. Unlike text documents used in traditional information retrieval, which are often considered independent of one another (i.e., with no explicit relationships or links among them except in citation analysis), Web pages are connected through hyperlinks, which carry important information. Some hyperlinks are used to organize a large amount of information at the same Web site, and thus only point to pages in the same site. Other hyperlinks point to pages in other Web sites. Such out-going hyperlinks often indicate an implicit conveyance of authority to the pages being pointed to. Therefore, those pages that are pointed to by many other pages are likely to contain authoritative or quality information. Such linkages should obviously be used in page evaluation and ranking in search engines.

During the period of 1997-1998, two most influential hyperlink based search algorithms PageRank [68, 422] and HITS [281] were designed. PageRank is the algorithm that powers the successful search engine Google. Both PageRank and HITS were originated from **social network analysis** [540]. They both exploit the hyperlink structure of the Web to rank pages according to their levels of "prestige" or "authority". We will study these

algorithms in this chapter. We should also note that hyperlink-based page evaluation and ranking is not the only method used by search engines. As we discussed in Chap. 6, contents and many other factors are also considered in producing the final ranking presented to the user.

Apart from search ranking, hyperlinks are also useful for finding Web communities. A Web community is a cluster of densely linked pages representing a group of people with a common interest. Beyond explicit hyperlinks on the Web, links in other contexts are useful too, e.g., for discovering communities of named entities (e.g., people and organizations) in free text documents, and for analyzing social phenomena in emails. This chapter will introduce some of the current algorithms.

7.1 Social Network Analysis

Social network is the study of social entities (people in an organization, called **actors**), and their interactions and relationships. The interactions and relationships can be represented with a network or graph, where each vertex (or node) represents an actor and each link represents a relationship. From the network we can study the properties of its structure, and the role, position and prestige of each social actor. We can also find various kinds of sub-graphs, e.g., **communities** formed by groups of actors.

Social network analysis is useful for the Web because the Web is essentially a virtual society, and thus a virtual social network, where each page can be regarded as a social actor and each hyperlink as a relationship. Many of the results from social networks can be adapted and extended for use in the Web context. The ideas from social network analysis are indeed instrumental to the success of Web search engines.

In this section, we introduce two types of social network analysis, **centrality** and **prestige**, which are closely related to hyperlink analysis and search on the Web. Both centrality and prestige are measures of degree of prominence of an actor in a social network. We introduce them below. For a more complete treatment of the topics, please refer to the authoritative text by Wasserman and Faust [540].

7.1.1 Centrality

Important or prominent actors are those that are linked or involved with other actors extensively. In the context of an organization, a person with extensive contacts (links) or communications with many other people in the organization is considered more important than a person with relatively

fewer contacts. The links can also be called **ties**. A **central actor** is one involved in many ties. Fig. 7.1 shows a simple example using an undirected graph. Each node in the social network is an actor and each link indicates that the actors on the two ends of the link communicate with each other. Intuitively, we see that the actor *i* is the most central actor because he/she can communicate with most other actors.

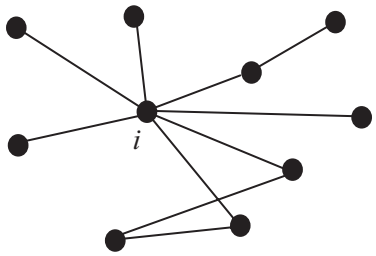


Fig. 7.1. An example of a social network

There are different types of links or involvements between actors. Thus, several types of centrality are defined on undirected and directed graphs. We discuss three popular types below.

Degree Centrality

Central actors are the most active actors that have most links or ties with other actors. Let the total number of actors in the network be n.

Undirected Graph: In an undirected graph, the **degree centrality** of an actor i (denoted by $C_D(i)$) is simply the node degree (the number of edges) of the actor node, denoted by $d(i)$, normalized with the maximum degree, $n-1$.

$$C_D(i) = \frac{d(i)}{n-1}. \tag{1}$$

The value of this measure ranges between 0 and 1 as $n-1$ is the maximum value of $d(i)$.

Directed Graph: In this case, we need to distinguish **in-links** of actor i (links pointing to i), and **out-links** (links pointing out from i). The degree centrality is defined based on only the out-degree (the number of out-links or edges), $d_o(i)$.

$$C_D'(i) = \frac{d_o(i)}{n-1}. \tag{2}$$

Closeness Centrality

This view of centrality is based on the closeness or distance. The basic idea is that an actor x_i is central if it can easily interact with all other actors. That is, its distance to all other actors is short. Thus, we can use the shortest distance to compute this measure. Let the shortest distance from actor i to actor j be $d(i, j)$ (measured as the number of links in a shortest path).

Undirected Graph: The closeness centrality $C_C(i)$ of actor i is defined as

$$C_C(i) = \frac{n-1}{\sum_{j=1}^{n} d(i,j)}. \tag{3}$$

The value of this measure also ranges between 0 and 1 as n–1 is the minimum value of the denominator, which is the sum of the shortest distances from i to all other actors. Note that this equation is only meaningful for a connected graph.

Directed Graph: The same equation can be used for a directed graph. The distance computation needs to consider directions of links or edges.

Betweenness Centrality

If two non-adjacent actors j and k want to interact and actor i is on the path between j and k, then i may have some control over their interactions. Betweenness measures this control of i over other pairs of actors. Thus, if i is on the paths of many such interactions, then i is an important actor.

Undirected Graph: Let p_{jk} be the number of shortest paths between actors j and k. The betweenness of an actor i is defined as the number of shortest paths that pass i (denoted by $p_{jk}(i)$, $j \neq i$ and $k \neq i$) normalized by the total number of shortest paths of all pairs of actors not including i:

$$C_B(i) = \sum_{j<k} \frac{p_{jk}(i)}{p_{jk}}. \tag{4}$$

Note that there may be multiple shortest paths between actor j and actor k. Some pass i and some do not. We assume that all paths are equally likely to be used. $C_B(i)$ has a minimum of 0, attained when i falls on no shortest path. Its maximum is $(n-1)(n-2)/2$, which is the number of pairs of actors not including i.

In the network of Fig. 7.2, actor 1 is the most central actor. It lies on all 15 shortest paths linking the other 6 actors. $C_B(1)$ has the maximum value of 15, and $C_B(2) = C_B(3) = C_B(4) = C_B(5) = C_B(6) = C_B(7) = 0$.

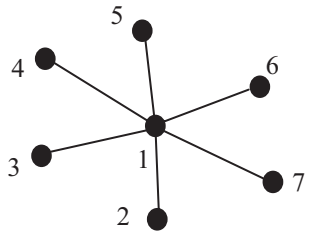


Fig. 7.2. An example of a network illustrating the betweenness centrality

If we are to ensure that the value range is between 0 and 1, we can normalize it with $(n-1)(n-2)/2$, which is the maximum value of $C_B(i)$. The standardized betweenness of actor i is defined as

$$C'_B(i) = \frac{2\sum_{j<k} \frac{p_{jk}(i)}{p_{jk}}}{(n-1)(n-2)}. \tag{5}$$

Unlike the closeness measure, the betweenness can be computed even if the graph is not connected.

Directed Graph: The same equation can be used but must be multiplied by 2 because there are now $(n-1)(n-2)$ pairs considering a path from j to k is different from a path from k to j. Likewise, p_{jk} must consider paths from both directions.

7.1.2 Prestige

Prestige is a more refined measure of prominence of an actor than centrality as we will see below. We need to distinguish between ties sent (out-links) and ties received (in-links). A prestigious actor is defined as one who is object of extensive ties as a recipient. In other words, to compute the prestige of an actor, we only look at the ties (links) directed or pointed to the actor (in-links). Hence, the prestige cannot be computed unless the relation is directional or the graph is directed. The main difference between the concepts of centrality and prestige is that centrality focuses on out-links while prestige focuses on in-links. We define three prestige measures. The third prestige measure (i.e., **rank prestige**) forms the basis of most Web page link analysis algorithms, including PageRank and HITS.

Degree Prestige

Based on the definition of the prestige, it is clear that an actor is prestigious if it receives many in-links or nominations. Thus, the simplest measure of prestige of an actor i (denoted by $P_D(i)$) is its in-degree.

$$P_D(i) = \frac{d_I(i)}{n-1}, \tag{6}$$

where $d_I(i)$ is the in-degree of i (the number of in-links of i) and n is the total number of actors in the network. As in the degree centrality, dividing by $n - 1$ standardizes the prestige value to the range from 0 and 1. The maximum prestige value is 1 when every other actor links to or chooses actor i.

Proximity Prestige

The degree index of prestige of an actor i only considers the actors that are adjacent to i. The proximity prestige generalizes it by considering both the actors directly and indirectly linked to actor i. That is, we consider every actor j that can reach i, i.e., there is a directed path from j to i.

Let I_i be the set of actors that can reach actor i, which is also called the **influence domain** of actor i. The **proximity** is defined as closeness or distance of other actors to i. Let $d(j, i)$ denote the shortest path distance from actor j to actor i. Each link has the unit distance. To compute the proximity prestige, we use the average distance, which is

$$\frac{\sum_{j \in I_i} d(j,i)}{|I_i|}, \tag{7}$$

where $|I_i|$ is the size of the set I_i. If we look at the ratio or proportion of actors who can reach i to the average distance that these actors are from i, we obtain the proximity prestige, which has the value range of [0, 1]:

$$P_P(i) = \frac{|I_i|/(n-1)}{\sum_{j \in I_i} d(j,i) \Big/ |I_i|}, \tag{8}$$

where $|I_i|/(n-1)$ is the proportion of actors that can reach actor i. In one extreme, every actor can reach actor i, which gives $|I_i|/(n-1) = 1$. The denominator is 1 if every actor is adjacent to i. Then, $P_P(i) = 1$. On the other extreme, no actor can reach actor i. Then $|I_i| = 0$, and $P_P(i) = 0$.

Rank Prestige

The above two prestige measures are based on in-degrees and distances. However, an important factor that has not been considered is the **prominence** of individual actors who do the "voting" or "choosing." In the real world, a person i chosen by an important person is more prestigious than chosen by a less important person. For example, a company CEO voting for a person is much more important than a worker voting for the person. If one's circle of influence is full of prestigious actors, then one's own prestige is also high. Thus one's prestige is affected by the ranks or statuses of the involved actors. Based on this intuition, the rank prestige $P_R(i)$ is defined as a linear combination of links that point to i:

$$P_R(i) = A_{1i}P_R(1) + A_{2i}P_R(2) + ... + A_{ni}P_R(n), \tag{9}$$

where $A_{ji} = 1$ if j points to i, and 0 otherwise. This equation says that an actor's rank prestige is a function of the ranks of the actors who vote or choose the actor, which makes perfect sense.

Since we have n equations for n actors, we can write them in the matrix notation. We use $\boldsymbol{P}$ to represent the vector that contains all the rank prestige values, i.e., $\boldsymbol{P} = (P_R(1), P_R(2), \ldots, P_R(n))^T$ (T means **matrix transpose**). $\boldsymbol{P}$ is represented as a column vector. We use matrix $\boldsymbol{A}$ (where $A_{ij} = 1$ if i points to j, and 0 otherwise) to represent the adjacency matrix of the network or graph. As a notational convention, we use bold italic letters to represent matrices. We then have

$$\boldsymbol{P} = \boldsymbol{A}^T\boldsymbol{P}. \tag{10}$$

This equation is precisely the characteristic equation used for finding the **eigensystem** of the matrix $\boldsymbol{A}^T$. $\boldsymbol{P}$ is an **eigenvector** of $\boldsymbol{A}^T$.

This equation and the idea behind it turn out to be very useful in Web search. Indeed, the most well known ranking algorithms for Web search, PageRank and HITS, are directly related to this equation. Sect. 7.3 and 7.4 will focus on these two algorithms and describe how to solve the equation to obtain the prestige value of each actor (or each page on the Web).

7.2 Co-Citation and Bibliographic Coupling

Another area of research concerned with links is the **citation analysis** of scholarly publications. A scholarly publication usually cites related prior work to acknowledge the origins of some ideas in the publication and to compare the new proposal with existing work. Citation analysis is an area

of bibliometric research, which studies citations to establish the relationships between authors and their work.

When a publication (also called a paper) cites another publication, a relationship is established between the publications. Citation analysis uses these relationships (links) to perform various types of analysis. A citation can represent many types of links, such as links between authors, publications, journals and conferences, and fields, or even between countries. We will discuss two specific types of citation analysis, **co-citation** and **bibliographic coupling**. The HITS algorithm of Sect. 7.4 is related to these two types of analysis.

7.2.1 Co-Citation

Co-citation is used to measure the similarity of two documents. If papers *i* and *j* are both cited by paper *k*, then they may be said to be related in some sense to one another, even they do not directly cite each other. Figure 7.3 shows that papers *i* and *j* are co-cited by paper *k*. If papers *i* and *j* are cited together by many papers, it means that *i* and *j* have a strong relationship or similarity. The more papers they are cited by, the stronger their relationship is.

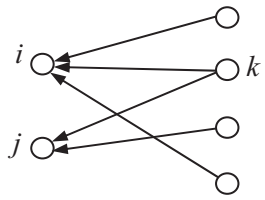


Fig. 7.3. Paper *i* and paper *j* are co-cited by paper *k*

Let $\boldsymbol{L}$ be the citation matrix. Each cell of the matrix is defined as follows: $L_{ij} = 1$ if paper *i* cites paper *j*, and 0 otherwise. **Co-citation** (denoted by C_{ij}) is a similarity measure defined as the number of papers that co-cite *i* and *j*, and is computed with

$$C_{ij} = \sum_{k=1}^{n} L_{ki} L_{kj}, \tag{11}$$

where *n* is the total number of papers. C_{ii} is naturally the number of papers that cite *i*. A square matrix $\boldsymbol{C}$ can be formed with C_{ij}, and it is called the **co-citation matrix**. Co-citation is symmetric, $C_{ij} = C_{ji}$, and is commonly used as a similarity measure of two papers in clustering to group papers of similar topics together.

7.2.2 Bibliographic Coupling

Bibliographic coupling operates on a similar principle, but in a way it is the mirror image of co-citation. Bibliographic coupling links papers that cite the same articles so that if papers *i* and *j* both cite paper *k*, they may be said to be related, even though they do not directly cite each other. The more papers they both cite, the stronger their similarity is. Figure 7.4 shows both papers *i* and *j* citing (referencing) paper *k*.

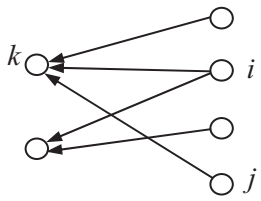


Fig. 7.4. Both paper *i* and paper *j* cite paper *k*

We use B_{ij} to represent the number of papers that are cited by both papers *i* and *j*:

$$B_{ij} = \sum_{k=1}^{n} L_{ik} L_{jk}. \tag{12}$$

B_{ii} is naturally the number of references (in the reference list) of paper *i*. A square matrix $\boldsymbol{B}$ can be formed with B_{ij}, and it is called the **bibliographic coupling matrix**. Bibliographic coupling is also symmetric and is regarded as a similarity measure of two papers in clustering.

We will see later that two important types of pages on the Web, **hubs** and **authorities**, found by the HITS algorithm are directly related to co-citation and bibliographic coupling matrices.

7.3 PageRank

The year 1998 was an important year for Web link analysis and Web search. Both the PageRank and the HITS algorithms were reported in that year. HITS was presented by Jon Kleinberg in January, 1998 at the *Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*. PageRank was presented by Sergey Brin and Larry Page at the *Seventh International World Wide Web Conference* (*WWW7*) in April, 1998. Based on the algorithm, they built the search engine Google. The main ideas of PageRank and HITS are really quite similar. However, it is their dissimilarity that

made a huge difference as we will see later. Since that year, PageRank has emerged as the dominant link analysis model for Web search, partly due to its query-independent evaluation of Web pages and its ability to combat spamming, and partly due to Google's business success. In this section, we focus on PageRank. In the next section, we discuss HITS. A detailed study of these algorithms can also be found in [304].

PageRank relies on the democratic nature of the Web by using its vast link structure as an indicator of an individual page's quality. In essence, PageRank interprets a hyperlink from page x to page y as a vote, by page x, for page y. However, PageRank looks at more than just the sheer number of votes, or links that a page receives. It also analyzes the page that casts the vote. Votes casted by pages that are themselves "important" weigh more heavily and help to make other pages more "important." This is exactly the idea of **rank prestige** in social networks (see Sect. 7.1.2).

7.3.1 PageRank Algorithm

PageRank is a static ranking of Web pages in the sense that a PageRank value is computed for each page off-line and it does not depend on search queries. Since PageRank is based on the measure of prestige in social networks, the PageRank value of each page can be regarded as its prestige. We now derive the PageRank formula. Let us first state some main concepts again in the Web context.

In-links of page i: These are the hyperlinks that point to page i from other pages. Usually, hyperlinks from the same site are not considered.

Out-links of page i: These are the hyperlinks that point out to other pages from page i. Usually, links to pages of the same site are not considered.

From the perspective of prestige, we use the following to derive the PageRank algorithm.

1. A hyperlink from a page pointing to another page is an implicit conveyance of authority to the target page. Thus, the more in-links that a page i receives, the more prestige the page i has.
2. Pages that point to page i also have their own prestige scores. A page with a higher prestige score pointing to i is more important than a page with a lower prestige score pointing to i. In other words, a page is important if it is pointed to by other important pages.

According to rank prestige in social networks, the importance of page i (i's PageRank score) is determined by summing up the PageRank scores of all pages that point to i. Since a page may point to many other pages, its pres-

tige score should be shared among all the pages that it points to. Notice the difference from rank prestige, where the prestige score is not shared.

To formulate the above ideas, we treat the Web as a directed graph $G = (V, E)$, where V is the set of vertices or nodes, i.e., the set of all pages, and E is the set of directed edges in the graph, i.e., hyperlinks. Let the total number of pages on the Web be n (i.e., $n = |V|$). The PageRank score of the page i (denoted by $P(i)$) is defined by:

$$P(i) = \sum_{(j,i)\in E} \frac{P(j)}{O_j}, \tag{13}$$

where O_j is the number of out-links of page j. Mathematically, we have a system of n linear equations (13) with n unknowns. We can use a matrix to represent all the equations. Let $\boldsymbol{P}$ be a n-dimensional column vector of PageRank values, i.e.,

$$\boldsymbol{P} = (P(1), P(2), \ldots, P(n))^T.$$

Let $\boldsymbol{A}$ be the adjacency matrix of our graph with

$$A_{ij} = \begin{cases} \dfrac{1}{O_i} & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

We can write the system of n equations with (similar to Equation 10)

$$\boldsymbol{P} = \boldsymbol{A}^T \boldsymbol{P}. \tag{15}$$

This is the characteristic equation of the **eigensystem**, where the solution to $\boldsymbol{P}$ is an **eigenvector** with the corresponding **eigenvalue** of 1. Since this is a circular definition, an iterative algorithm is used to solve it. It turns out that if *some conditions* are satisfied (which will be described shortly), 1 is the largest **eigenvalue** and the PageRank vector $\boldsymbol{P}$ is the **principal eigenvector**. A well known mathematical technique called **power iteration** can be used to find $\boldsymbol{P}$.

However, the problem is that Equation (15) does not quite suffice because the Web graph does not meet the conditions. To introduce these conditions and the enhanced equation, let us derive the same Equation (15) based on the **Markov chain** [207].

In the Markov chain model, each Web page or node in the Web graph is regarded as a state. A hyperlink is a transition, which leads from one state to another state with a probability. Thus, this framework models Web surfing as a stochastic process. It models a Web surfer randomly surfing the Web as a state transition in the Markov chain. Recall that we used O_i to

denote the number of out-links of a node i. Each transition probability is $1/O_i$ if we assume the Web surfer will click the hyperlinks in the page i uniformly at random, the "back" button on the browser is not used and the surfer does not type in an URL. Let $\boldsymbol{A}$ be the state transition probability matrix, a square matrix of the following format,

$$\boldsymbol{A} = \begin{pmatrix} A_{11} & A_{12} & . & . & . & A_{1n} \\ A_{21} & A_{22} & . & . & . & A_{2n} \\ . & . & & & & . \\ . & . & & & & . \\ . & . & & & & . \\ A_{n1} & A_{n2} & . & . & . & A_{nn} \end{pmatrix}$$

A_{ij} represents the transition probability that the surfer in state i (page i) will move to state j (page j). A_{ij} is defined exactly as in Equation (14).

Given an **initial probability distribution** vector that a surfer is at each state (or page) $\boldsymbol{p}_0 = (p_0(1), p_0(2), \ldots, p_0(n))^T$ (a column vector) and an $n \times n$ **transition probability matrix** $\boldsymbol{A}$, we have

$$\sum_{i=1}^{n} p_0(i) = 1 \tag{16}$$

$$\sum_{j=1}^{n} A_{ij} = 1. \tag{17}$$

Equation (17) is not quite true for some Web pages because they have no out-links. If the matrix $\boldsymbol{A}$ satisfies Equation (17), we say that $\boldsymbol{A}$ is the **stochastic matrix** of a Markov chain. Let us assume $\boldsymbol{A}$ is a stochastic matrix for the time being and deal with it not being that later.

In a Markov chain, a question of common interest is: Given the initial probability distribution $\boldsymbol{p}_0$ at the beginning, what is the probability that m steps/transitions later that the Markov chain will be at each state j? We can determine the probability that the system (or the **random surfer**) is in state j after 1 step (1 state transition) by using the following reasoning:

$$p_1(j) = \sum_{i=1}^{n} A_{ij}(1)\, p_0(i), \tag{18}$$

where $A_{ij}(1)$ is the probability of going from i to j after 1 transition, and $A_{ij}(1) = A_{ij}$. We can write it with a matrix:

$$\boldsymbol{p}_1 = \boldsymbol{A}^T \boldsymbol{p}_0 . \tag{19}$$

In general, the probability distribution after k steps/transitions is:

$$\boldsymbol{p}_k = \boldsymbol{A}^T \boldsymbol{p}_{k-1} . \tag{20}$$

Equation (20) looks very similar to Equation (15). We are getting there.

By the Ergodic Theorem of Markov chains [207], a finite Markov chain defined by the **stochastic transition matrix** $\boldsymbol{A}$ has a unique **stationary probability distribution** if $\boldsymbol{A}$ is **irreducible** and **aperiodic**. These mathematical terms will be defined as we go along.

The stationary probability distribution means that after a series of transitions $\boldsymbol{p}_k$ will converge to a steady-state probability vector $\boldsymbol{\pi}$ regardless of the choice of the initial probability vector $\boldsymbol{p}_0$, i.e.,

$$\lim_{k \to \infty} \boldsymbol{p}_k = \boldsymbol{\pi} . \tag{21}$$

When we reach the steady-state, we have $\boldsymbol{p}_k = \boldsymbol{p}_{k+1} = \boldsymbol{\pi}$, and thus $\boldsymbol{\pi} = \boldsymbol{A}^T \boldsymbol{\pi}$. $\boldsymbol{\pi}$ is the **principal eigenvector** of $\boldsymbol{A}^T$ with **eigenvalue** of 1. In PageRank, $\boldsymbol{\pi}$ is used as the PageRank vector $\boldsymbol{P}$. Thus, we again obtain Equation (15), which is re-produced here as Equation (22):

$$\boldsymbol{P} = \boldsymbol{A}^T \boldsymbol{P} . \tag{22}$$

Using the stationary probability distribution $\boldsymbol{\pi}$ as the PageRank vector is reasonable and quite intuitive because it reflects the long-run probabilities that a random surfer will visit the pages. A page has a high prestige if the probability of visiting it is high.

Now let us come back to the real Web context and see whether the above conditions are satisfied, i.e., whether $\boldsymbol{A}$ is a stochastic matrix and whether it is irreducible and aperiodic. In fact, none of them is satisfied. Hence, we need to extend the ideal-case Equation (22) to produce the "actual PageRank model". Let us look at each condition below.

First of all, $\boldsymbol{A}$ is not a **stochastic (transition) matrix**. A stochastic matrix is the transition matrix for a finite Markov chain whose entries in each row are non-negative real numbers and sum to 1 (i.e., Equation 17). This requires that every Web page must have at least one out-link. This is not true on the Web because many pages have no out-links, which are reflected in transition matrix $\boldsymbol{A}$ by some rows of complete 0's. Such pages are called the **dangling pages** (nodes).

Example 1: Figure 7.5 shows an example of a hyperlink graph.

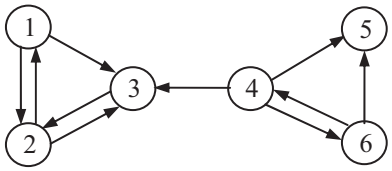


Fig. 7.5. An example of a hyperlink graph

If we assume that the Web surfer will click the hyperlinks in a page uniformly at random, we have the following transition probability matrix:

$$\boldsymbol{A} = \begin{pmatrix} 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 0 & 1/3 & 1/3 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \end{pmatrix}. \tag{23}$$

For example $A_{12} = A_{13} = 1/2$ because node 1 has two out-links. We can see that $\boldsymbol{A}$ is not a stochstic matrix because the fifth row is all 0's, i.e., page 5 is a dangling page. ■

We can fix this problem in several ways in order to convert $\boldsymbol{A}$ to a stochastic transition matrix. We describe only two ways here:

1. Remove those pages with no out-links from the system during the PageRank computation as these pages do not affect the ranking of any other page directly. Out-links from other pages pointing to these pages are also removed. After PageRanks are computed, these pages and hyperlinks pointing to them can be added in. Their PageRanks are easy to calculate based on Equation (22). Note that the transition probabilities of those pages with removed links will be slightly affected but not significantly. This method is suggested in [68].
2. Add a complete set of outgoing links from each such page i to all the pages on the Web. Thus the transition probability of going from i to every page is $1/n$ assuming uniform probability distribution. That is, we replace each row containing all 0's with $\boldsymbol{e}/n$, where $\boldsymbol{e}$ is n-dimensional vector of all 1's.

If we use the second method to make $\boldsymbol{A}$ a stochastic matrix by adding a link from page 5 to every page, we obtain

$$\overline{\boldsymbol{A}} = \begin{pmatrix} 0 & 1/2 & 1/2 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/3 & 0 & 1/3 & 1/3 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 0 & 0 & 0 & 1/2 & 1/2 & 0 \end{pmatrix}. \tag{24}$$

Below, we assume that either one of the above is done to make $\boldsymbol{A}$ a stochastic matrix.

Second, $\boldsymbol{A}$ is not **irreducible**. Irreducible means that the Web graph G is strongly connected.

Definition (strongly connected): A directed graph $G = (V, E)$ is **strongly connected** if and only if, for each pair of nodes $u, v \in V$, there is a path from u to v.

A general Web graph represented by $\boldsymbol{A}$ is not irreducible because for some pair of nodes u and v, there is no path from u to v. For example, in Fig. 7.5, there is no directed path from node 3 to node 4. The adjustment in Equation (24) is not enough to ensure irreducibility. That is, in $\overline{\boldsymbol{A}}$, there is still no directed path from node 3 to node 4. This problem and the next problem can be dealt with using a single strategy (to be described shortly).

Finally, $\boldsymbol{A}$ is not **aperiodic**. A state i in a Markov chain being periodic means that there exists a directed cycle that the chain has to traverse.

Definition (aperiodic): A state i is **periodic** with period $k > 1$ if k is the smallest number such that all paths leading from state i back to state i have a length that is a multiple of k. If a state is not periodic (i.e., $k = 1$), it is **aperiodic**. A Markov chain is **aperiodic** if all states are aperiodic.

Example 2: Figure 7.6 shows a periodic Markov chain with $k = 3$. The transition matrix is given on the left. Each state in this chain has a period of 3. For example, if we start from state 1, to come back to state 1 the only path is 1-2-3-1 for some number of times, say h. Thus any return to state 1 will take $3h$ transitions. In the Web, there could be many such cases. ■

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

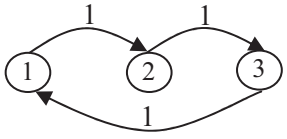


Fig. 7.6. A periodic Markov chain with $k = 3$.

It is easy to deal with the above two problems with a single strategy.

- We add a link from each page to every page and give each link a small transition probability controlled by a parameter *d*.

The augmented transition matrix becomes **irreducible** because it is clearly strongly connected. It is also **aperiodic** because the situation in Fig. 7.6 no longer exists as we now have paths of all possible lengths from state *i* back to state *i*. That is, the random surfer does not have to traverse a fixed cycle for any state. After this augmentation, we obtain an improved PageRank model. In this model, at a page, the random surfer has two options:

1. With probability *d*, he randomly chooses an out-link to follow.
2. With probability 1–*d*, he jumps to a random page without a link.

Equation (25) gives the improved model,

$$\boldsymbol{P} = \left((1-d)\frac{\boldsymbol{E}}{n} + d\boldsymbol{A}^T \right)\boldsymbol{P} \tag{25}$$

where $\boldsymbol{E}$ is $\boldsymbol{e}\boldsymbol{e}^T$ ($\boldsymbol{e}$ is a column vector of all 1's) and thus $\boldsymbol{E}$ is a $n \times n$ square matrix of all 1's. $1/n$ is the probability of jumping to a particular page. n is the total number of nodes in the Web graph. Note that Equation (25) assumes that $\boldsymbol{A}$ has already been made a stochastic matrix.

Example 3: If we follow our example in Fig. 7.5 and Equation (24) (we use $\overline{\boldsymbol{A}}$ for $\boldsymbol{A}$ here), the augmented transition matrix is

$$(1-d)\frac{\boldsymbol{E}}{n} + d\boldsymbol{A}^T = \begin{pmatrix} 1/60 & 7/15 & 1/60 & 1/60 & 1/6 & 1/60 \\ 7/15 & 1/60 & 11/12 & 1/60 & 1/6 & 1/60 \\ 7/15 & 7/15 & 1/60 & 19/60 & 1/6 & 1/60 \\ 1/60 & 1/60 & 1/60 & 1/60 & 1/6 & 7/15 \\ 1/60 & 1/60 & 1/60 & 19/60 & 1/6 & 7/15 \\ 1/60 & 1/60 & 1/60 & 19/60 & 1/6 & 1/60 \end{pmatrix} \tag{26}$$

■

$(1-d)\boldsymbol{E}/n + d\boldsymbol{A}^T$ is a **stochastic matrix** (but transposed). It is also **irreducible** and **aperiodic** as we discussed above. Here we use $d = 0.9$.

If we scale Equation (25) so that $\boldsymbol{e}^T\boldsymbol{P} = n$, we obtain

$$\boldsymbol{P} = (1-d)\boldsymbol{e} + d\boldsymbol{A}^T\boldsymbol{P}\,. \tag{27}$$

Before scaling, we have $\boldsymbol{e}^T\boldsymbol{P} = 1$ (i.e., $P(1) + P(2) + \ldots + P(n) = 1$ if we recall that $\boldsymbol{P}$ is the stationary probability vector $\boldsymbol{\pi}$ of the Markov chain). The scaling is equivalent to multiplying n on both sides of Equation (25).

This gives us the PageRank formula for each page i as follows:

$$P(i) = (1-d) + d\sum_{j=1}^{n} A_{ji}P(j), \tag{28}$$

which is equivalent to the formula given in the PageRank papers [68, 422]:

$$P(i) = (1-d) + d\sum_{(j,i)\in E} \frac{P(j)}{O_j}. \tag{29}$$

The parameter d is called the **damping factor** which can be set to between 0 and 1. $d = 0.85$ is used in [68, 422].

The computation of PageRank values of the Web pages can be done using the well known **power iteration method** [203], which produces the principal eigenvector with the eigenvalue of 1. The algorithm is simple, and is given in Fig. 7.7. One can start with any initial assignments of PageRank values. The iteration ends when the PageRank values do not change much or converge. In Fig. 7.7, the iteration ends after the 1-norm of the residual vector is less than a pre-specified threshold ε. Note that the 1-norm for a vector is simply the sum of all the components.

PageRank-Iterate(G)
  $\boldsymbol{P}_0 \leftarrow \boldsymbol{e}/n$
  $k \leftarrow 1$
  repeat
    $\boldsymbol{P}_k \leftarrow (1-d)\boldsymbol{e} + d\boldsymbol{A}^T \boldsymbol{P}_{k-1}$;
    $k \leftarrow k + 1$;
  until $||\boldsymbol{P}_k - \boldsymbol{P}_{k-1}||_1 < \varepsilon$
  return $\boldsymbol{P}_k$

Fig. 7.7. The power iteration method for PageRank

Since we are only interested in the ranking of the pages, the actual convergence may not be necessary. Thus, fewer iterations are needed. In [68], it is reported that on a database of 322 million links the algorithm converges to an acceptable tolerance in roughly 52 iterations.

7.3.2 Strengths and Weaknesses of PageRank

The main advantage of PageRank is its ability to fight spam. A page is important if the pages pointing to it are important. Since it is not easy for Web page owner to add in-links into his/her page from other important pages, it is thus not easy to influence PageRank. Nevertheless, there are

reported ways to influence PageRank. Recognizing and fighting spam is an important issue in Web search.

Another major advantage of PageRank is that it is a global measure and is query independent. That is, the PageRank values of all the pages on the Web are computed and saved off-line rather than at the query time. At the query time, only a lookup is needed to find the value to be integrated with other strategies to rank the pages. It is thus very efficient at query time. Both these two advantages contributed greatly to Google's success.

The main criticism is also the query-independence nature of PageRank. It could not distinguish between pages that are authoritative in general and pages that are authoritative on the query topic. Google may have other ways to deal with the problem, which we do not know due to the proprietary nature of Google. Another criticism is that PageRank does not consider time. Let us give some explanation to this.

7.3.3 Timed PageRank

The Web is a dynamic environment, and it changes constantly. Quality pages in the past may not be quality pages now or in the future. Thus, search has a temporal dimension. An algorithm called **TimedPageRank** given in [326, 585] adds the temporal dimension to PageRank. The motivations are:

1. Users are often interested in the latest information. Apart from pages that contain well-established facts and classics which do not change significantly over time, most contents on the Web change constantly. New pages or contents are added, and ideally, outdated contents and pages are deleted. However, in practice many outdated pages and links are not deleted. This causes problems for Web search because such outdated pages may still be ranked very high.
2. PageRank favors pages that have many in-links. To some extent, we can say that it favors older pages because they have existed on the Web for a long time and thus have accumulated many in-links. Then the problem is that new pages which are of high quality and also give the up-to-date information will not be assigned high scores and consequently will not be ranked high because they have fewer or no in-links. It is thus difficult for users to find the latest information on the Web based on PageRank.

The idea of TimedPageRank is simple. Instead of using a constant damping factor d as the parameter in PageRank, TimedPageRank uses a function of time $f(t)$ ($0 \leq f(t) \leq 1$), where t is the difference between the current time and the time when the page was last updated. $f(t)$ returns a probability that

the Web surfer will follow an actual link on the page. $1-f(t)$ returns the probability that the surfer will jump to a random page. Thus, at a particular page i, the Web surfer has two options:

1. With probability $f(t_i)$, he randomly chooses an out-going link to follow.
2. With probability $1-f(t_i)$, he jumps to a random page without a link.

The intuition here is that if the page was last updated (or created) a long time ago, the pages that it cites (points to) are even older and are probably out of date. Then the $1-f(t)$ value for such a page should be large, which means that the surfer will have a high probability of jumping to a random page. If a page is new, then its $1-f(t)$ value should be small, which means that the surfer will have a high probability to follow an out-link of the page and a small probability of jumping to a random page.

For a complete new page in a Web site, which does not have any in-links at all, the method given in [326] uses the average TimedPageRank value of the past pages in the Web site.

Finally, we note again that the link-based ranking is not the only strategy used in a search engine. Many other information retrieval methods, heuristics and empirical parameters are also employed. However, their details are not published. We also note that PageRank is not the only link-based static and global ranking algorithm. All major search engines, such as *Yahoo*! and MSN, have their own algorithms but are unpublished.

7.4 HITS

HITS stands for **Hypertext Induced Topic Search** [281]. Unlike PageRank which is a static ranking algorithm, HITS is search query dependent. When the user issues a search query, HITS first expands the list of relevant pages returned by a search engine and then produces two rankings of the expanded set of pages, **authority ranking** and **hub ranking**.

An **authority** is a page with many in-links. The idea is that the page may have good or authoritative content on some topic and thus many people trust it and link to it. A **hub** is a page with many out-links. The page serves as an organizer of the information on a particular topic and points to many good authority pages on the topic. When a user comes to this hub page, he/she will find many useful links which take him/her to good content pages on the topic. Figure 7.8 shows an authority page and a hub page.

The key idea of HITS is that a good hub points to many good authorities and a good authority is pointed to by many good hubs. Thus, authorities and hubs have a **mutual reinforcement** relationship. Figure 7.9 shows a

set of densely linked authorities and hubs (a **bipartite sub-graph**).

Below, we first present the HITS algorithm, and also make a connection between HITS and co-citation and bibliographic coupling in bibliometric research. We then discuss the strengths and weaknesses of HITS, and describe some possible ways to deal with its weaknesses.

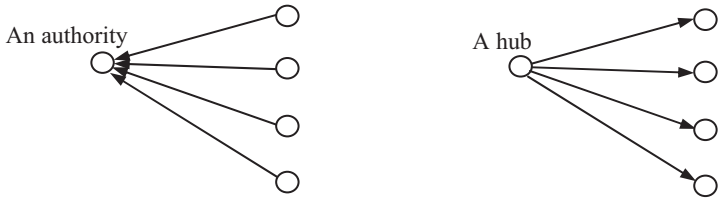


Fig. 7.8. An authority page and a hub page

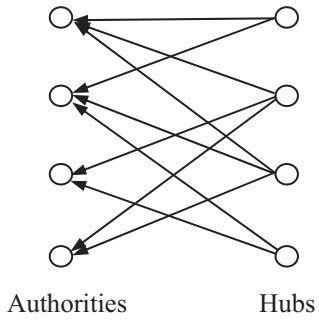


Fig. 7.9. A densely linked set of authorities and hubs

7.4.1 HITS Algorithm

Before describing the HITS algorithm, let us first describe how HITS collects pages to be ranked. Given a broad search query, q, HITS collects a set of pages as follows:

1. It sends the query q to a search engine system. It then collects t ($t = 200$ is used in the HITS paper) highest ranked pages, which assume to be highly relevant to the search query. This set is called the **root** set W.
2. It then grows W by including any page pointed to by a page in W and any page that points to a page in W. This gives a larger set called S. However, this set can be very large. The algorithm restricts its size by allowing each page in W to bring at most k pages ($k = 50$ is used in the HITS paper) pointing to it into S. The set S is called the **base set**.

HITS then works on the pages in S, and assigns every page in S an **authority score** and a **hub score**. Let the number of pages to be studied be n. We again use $G = (V, E)$ to denote the (directed) link graph of S. V is the set of pages (or nodes) and E is the set of directed edges (or links). We use $\boldsymbol{L}$ to denote the adjacency matrix of the graph.

$$L_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases} \tag{30}$$

Let the authority score of the page i be $a(i)$, and the hub score of page i be $h(i)$. The mutual reinforcing relationship of the two scores is represented as follows:

$$a(i) = \sum_{(j,i)\in E} h(j) \tag{31}$$

$$h(i) = \sum_{(i,j)\in E} a(j) \tag{32}$$

Writing them in the matrix form, we use $\boldsymbol{a}$ to denote the column vector with all the authority scores, $\boldsymbol{a} = (a(1), a(2), \ldots, a(n))^T$, and use $\boldsymbol{h}$ to denote the column vector with all the authority scores, $\boldsymbol{h} = (h(1), h(2), \ldots, h(n))^T$,

$$\boldsymbol{a} = \boldsymbol{L}^T\boldsymbol{h} \tag{33}$$

$$\boldsymbol{h} = \boldsymbol{L}\boldsymbol{a} \tag{34}$$

The computation of authority scores and hub scores is basically the same as the computation of the PageRank scores using the power iteration method. If we use $\boldsymbol{a}_k$ and $\boldsymbol{h}_k$ to denote authority and hub scores at the kth iteration, the iterative processes for generating the final solutions are

$$\boldsymbol{a}_k = \boldsymbol{L}^T\boldsymbol{L}\boldsymbol{a}_{k-1} \tag{35}$$

$$\boldsymbol{h}_k = \boldsymbol{L}\boldsymbol{L}^T\boldsymbol{h}_{k-1} \tag{36}$$

starting with

$$\boldsymbol{a}_0 = \boldsymbol{h}_0 = (1, 1, \ldots, 1). \tag{37}$$

Note that Equation (35) (or Equation 36) does not use the hub (or authority) vector due to substitutions of Equation (33) and Equation (34).

After each iteration, the values are also normalized (to keep them small) so that

HITS-Iterate(G)
  $\boldsymbol{a}_0 \leftarrow \boldsymbol{h}_0 \leftarrow (1, 1, \ldots, 1)$;
  $k \leftarrow 1$
  Repeat
    $\boldsymbol{a}_k \leftarrow \boldsymbol{L}^T \boldsymbol{L}\boldsymbol{a}_{k-1}$;
    $\boldsymbol{h}_k \leftarrow \boldsymbol{L}\boldsymbol{L}^T \boldsymbol{h}_{k-1}$;
    $\boldsymbol{a}_k \leftarrow \boldsymbol{a}_k / ||\boldsymbol{a}_k||_1$; // normalization
    $\boldsymbol{h}_k \leftarrow \boldsymbol{h}_k / ||\boldsymbol{h}_k||_1$; // normalization
    $k \leftarrow k + 1$;
  until $||\boldsymbol{a}_k - \boldsymbol{a}_{k-1}||_1 < \varepsilon_a$ and $||\boldsymbol{h}_k - \boldsymbol{h}_{k-1}||_1 < \varepsilon_h$;
  return $\boldsymbol{a}_k$ and $\boldsymbol{h}_k$

Fig. 7.10. The HITS algorithm based on power iteration

$$\sum_{i=1}^{n} a(i) = 1 \tag{38}$$

$$\sum_{i=1}^{n} h(i) = 1 \tag{39}$$

The power iteration algorithm for HITS is given in Fig. 7.10. The iteration ends after the 1-norms of the residual vectors are less than some thresholds ε_a and ε_h. Hence, the algorithm finds the principal eigenvectors at "equilibrium" as in PageRank. The pages with large authority and hub scores are better authorities and hubs respectively. HITS will select a few top ranked pages as authorities and hubs, and return them to the user.

Although HITS will always converge, there is a problem with uniqueness of limiting (converged) authority and hub vectors. It is shown that for certain types of graphs, different initializations to the power method produce different final authority and hub vectors. Some results can be inconsistent or wrong. Farahat et al. [171] gave several examples. The heart of the problem is that there are repeated dominant (principal) eigenvalues (several eigenvalues are the same and are dominant eigenvalues), which are caused by the problem that $\boldsymbol{L}^T\boldsymbol{L}$ (respectively $\boldsymbol{L}\boldsymbol{L}^T$) is reducible [303]. The first PageRank solution (Equation 22) has the same problem. However, the PageRank inventors found a way to get around the problem. A modification similar to PageRank may be applied to HITS.

7.4.2 Finding Other Eigenvectors

The HITS algorithm given in Fig. 7.10 finds the principal eigenvectors, which in a sense represent the most densely connected authorities and hubs in the graph G defined by a query. However, in some cases, we may also be interested in finding several densely linked collections of hubs and authorities among the same base set of pages. Each of such collections could potentially be relevant to the query topic, but they could be well-separated from one another in the graph G for a variety of reasons. For example,

1. The query string may be ambiguous with several very different meanings, e.g., "jaguar", which could be a cat or a car.
2. The query string may represent a topic that may arise as a term in the multiple communities, e.g. "classification".
3. The query string may refer to a highly polarized issue, involving groups that are not likely to link to one another, e.g. "abortion".

In each of these examples, the relevant pages can be naturally grouped into several clusters, also called **communities**. In general, the top ranked authorities and hubs represent the major cluster (or **community**). The smaller clusters (or communities), which are also represented by bipartite subgraphs as that in Fig. 7.9, can be found by computing non-principal eigenvectors. Non-principal eigenvectors are calculated in a similar way to power iteration using methods such as **orthogonal iteration** and **QR iteration**. We will not discuss the details of these methods. Interested readers can refer to the book by Golub and Van Loan [203].

7.4.3 Relationships with Co-Citation and Bibliographic Coupling

Authority pages and hub pages have their matches in the bibliometric citation context. An authority page is like an influential research paper (publication) which is cited by many subsequent papers. A hub page is like a survey paper which cites many other papers (including those influential papers). It is no surprise that there is a connection between authority and hub, and co-citation and bibliographic coupling.

Recall that co-citation of pages i and j, denoted by C_{ij}, is computed as

$$C_{ij} = \sum_{k=1}^{n} L_{ki} L_{kj} = (\boldsymbol{L}^T \boldsymbol{L})_{ij}. \tag{40}$$

This shows that the authority matrix ($\boldsymbol{L}^T\boldsymbol{L}$) of HITS is in fact the co-citation matrix $\boldsymbol{C}$ in the Web context. Likewise, recall that bibliographic

coupling of two pages i and j, denoted by B_{ij}, is computed as

$$B_{ij} = \sum_{k=1}^{n} L_{ik} L_{jk} = (\boldsymbol{L}\boldsymbol{L}^T)_{ij}, \tag{41}$$

which shows that the hub matrix ($\boldsymbol{L}\boldsymbol{L}^T$) of HITS is the bibliographic coupling matrix $\boldsymbol{B}$ in the Web context.

7.4.4 Strengths and Weaknesses of HITS

The main strength of HITS [281] is its ability to rank pages according to the query topic, which may be able to provide more relevant authority and hub pages. The ranking may also be combined with information retrieval based rankings. However, HITS has several disadvantages.

- First of all, it does not have the anti-spam capability of PageRank. It is quite easy to influence HITS by adding out-links from one's own page to point to many good authorities. This boosts the hub score of the page. Because hub and authority scores are interdependent, it in turn also increases the authority score of the page.
- Another problem of HITS is topic drift. In expanding the root set, it can easily collect many pages (including authority pages and hub pages) which have nothing to do the search topic because out-links of a page may not point to pages that are relevant to the topic and in-links to pages in the root set may be irrelevant as well because people put hyperlinks for all kinds of reasons, including spamming.
- The query time evaluation is also a major drawback. Getting the root set, expanding it and then performing eigenvector computation are all time consuming operations.

Over the years, many researchers tried to deal with these problems. We briefly discuss some of them below.

It was reported by several researchers in [52, 310, 405] that small changes to the Web graph topology can significantly change the final authority and hub vectors. Minor perturbations have little effect on PageRank, which is more stable than HITS. This is essentially due to the random jump step of PageRank. Ng et al. [405] proposed a method by introducing the same random jump step to HITS (by jumping to the base set uniformly at random with probability d), and showed that it could improve the stability of HITS significantly. Lempel and Moran [310] proposed SALSA, *a stochastic algorithm for link structure analysis*. SALSA combines some features of both PageRank and HITS to improve the authority and hub computation. It casts the problem as two Markov chains, an authority

Markov chain and a hub Markov chain. SALSA is less susceptible to spam since the coupling between hub and authority scores is much less strict.

Bharat and Henzinger [52] proposed a simple method to fight two site nepotistic links. That means that a set of pages on one host points to a single page on a second host. This drives up the hub scores of the pages on the first host and the authority score of the page on the second host. A similar thing can be done for hubs. These links may be authored by the same person and thus are regarded as "nepotistic" links to drive up the ranking of the target pages. [52] suggests weighting the links to deal with this problem. That is, if there are k edges from documents on a first host to a single document on a second host we give each edge an **authority weight** of $1/k$. If there are l edges from a single page on a first host to a set of pages on a second host, we give each edge a **hub weight** of $1/l$. These weights are used in the authority and hub computation. There are much more sophisticated spam techniques now involving more than two sites.

Regarding the topic drifting of HITS, existing fixes are mainly based on content similarity comparison during the expansion of the root set. In [88], if an expanded page is too different from the pages in the root set in terms of content similarity (based on cosine similarity), it is discarded. The remaining links are also weighted according to similarity. [88] proposes a method that uses the similarity between the anchor text of a link and the search topic to weight the link (instead of giving each link 1 as in HITS). [84] goes further to segment the page based on the DOM (**Document Object Model**) tree structure to identify the blocks or subtrees that are more related to the query topic instead of regarding the whole page as relevant to the search query. This is a good way to deal with multi-topic pages, which are abundant on the Web. A recent work on this is block-based link analysis [78], which segments each Web page into different blocks. Each block is given a different importance value according to its location in the page and other information. The importance value is then used to weight the links in the HITS (and also PageRank) computation. This will reduce the impact of unimportant links, which usually cause topic drifting and may even be a link spam.

7.5 Community Discovery

Intuitively, a community is simply a group of entities (e.g., people or organizations) that shares a common interest or is involved in an activity or event. In Sect. 7.4.2, we showed that the HITS algorithm can be used to find communities. The communities are represented by dense bipartite subgraphs. We now describe several other community finding algorithms.

Apart from the Web, communities also exist in emails and text documents. This section describes two community finding algorithms for the Web, one community finding algorithm for emails, and one community finding algorithm for text documents.

There are many reasons for discovering communities. For example, in the context of the Web, Kumar et al. [293] listed three reasons:

1. Communities provide valuable and possibly the most reliable, timely, and up-to-date information resources for a user interested in them.
2. They represent the sociology of the Web: studying them gives insights into the evolution of the Web.
3. They enable target advertising at a very precise level.

7.5.1 Problem Definition

Definition (community): Given a finite set of **entities** $S = \{s_1, s_2, \ldots, s_n\}$ of the same **type**, a **community** is a pair $C = (T, G)$, where T is the **community theme** and $G \subseteq S$ is the set of all entities in S that shares the theme T. If $s_i \in G$, s_i is said to be a **member** of the community C.

Some remarks about this definition are in order:

- A theme defines a community. That is, given a theme T, the set of members of the community is uniquely determined. Thus, two communities are equal if they have the same theme.
- A theme can be defined arbitrarily. For example, it can be an event (e.g., a sport event or a scandal) or a concept (e.g., Web mining).
- An entity s_i in S can be in any number of communities. That is, communities may overlap, or multiple communities may share members.
- The entities in S are of the same type. For example, this definition does not allow people and organizations to be in the same community.
- By no means does this definition cover every aspect of communities in the real world. For example, it does not consider the temporal dimension of communities. Usually a community exists within a specific period of time. Similarly, an entity may belong to a community during some time periods.
- This is a conceptual definition. In practice, different community mining algorithms have their own operational definitions which usually depend on how communities manifest themselves in the given data (which we will discuss shortly). Furthermore, the algorithms may not be able to discover all the members of a community or its precise theme.

Communities may also have hierarchical structures.

Definition (sub-community, super-community, and sub-theme): A community (T, G) may have a set of **sub-communities** $\{(T_1, G_1), \ldots, (T_m, G_m)\}$, where T_i is a **sub-theme** of T and $G_i \subseteq G$. (T, G) is also called a **super-community** of (T_i, G_i). In the same way, each sub-community (T_i, G_i) can be further decomposed, which gives us a **community hierarchy**.

Community Manifestation in Data: Given a data set, which can be a set of Web pages, a collection of emails, or a set of text documents, we want to find communities of entities in the data. However, the data itself usually does not explicitly give us the themes or the entities (community members) associated with the themes. The system needs to discover the hidden community structures. Thus, the first issue that we need to know is how communities manifest themselves. From such manifested evidences, the system can discover possible communities. Different types of data may have different forms of manifestation. We give three examples.

Web Pages:

1. Hyperlinks: A group of content creators sharing a common interest is usually inter-connected through hyperlinks. That is, members in a community are more likely to be connected among themselves than outside the community.
2. Content words: Web pages of a community usually contain words that are related to the community theme.

Emails:

1. Email exchange between entities: Members of a community are more likely to communicate with one another.
2. Content words: Email contents of a community also contain words related to the theme of the community.

Text documents:

1. Co-occurrence of entities: Members of a community are more likely to appear together in the same sentence and/or the same document.
2. Content words: Words in sentences indicate the community theme.

Clearly, the key form of manifestation of a community is that its members are linked in some way. The associated text often contains words that are indicative of the community theme.

Objective of Community Discovery: Given a data set containing entities, we want to discover hidden communities of the entities. For each community, we want to find the theme and its members. The theme is usually represented with a set of keywords.

7.5.2 Bipartite Core Communities

HITS finds dense bipartite graph communities based on broad topic queries. The question is whether it is possible to find all such communities efficiently from the crawl of the whole Web without using eigenvector computation which is relatively inefficient. Kumar et al. [293] presented a technique for finding bipartite cores, which are defined as follows.

Recall that the node set of a bipartite graph can be partitioned into two subsets, which we denote as set F and set C. A **bipartite core** is a complete bipartite sub-graph with at least i nodes in F and at least j nodes in C. A complete bipartite graph on node sets F and C contains all possible edges between the vertices of F and the vertices of C. Note that edges within F or within C are allowed here to suit the Web context, which deviate from the traditional definition of a complete bipartite graph. Intuitively, the core is a small (i, j)-sized complete bipartite sub-graph of the community, which contains some core members of the community but not all.

The cores that we seek are *directed*, i.e., there is a set of i pages all of which link to a set of j pages, while no assumption is made of links out of the latter set of j pages. Intuitively, the former is the set of pages created by members of the community, pointing to what they believe are the most valuable pages for that community. For this reason we will refer to the i pages that contain the links as **fans**, and the j pages that are referenced as **centers** (as in community centers). Fans are like specialized *hubs*, and centers are like *authorities*. Figure 7.11 shows an example of a bipartite core.

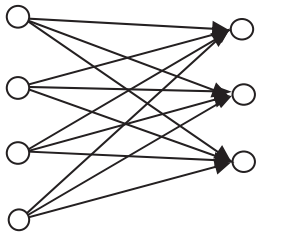


Fig. 7.11. A (4, 3) bipartite core

In Fig. 7.11, each fan page links to every center page. Since there are four fans and three centers, this is called a (4, 3) bipartite core. Such a core almost certainly represents a Web community, but a community may have multiple bipartite cores.

Given a large number of pages crawled from the Web, which is represented as a graph, the procedure for finding bipartite cores consists of two major steps: pruning and core generation.

Step 1: Pruning

We describe two types of pruning to remove those unqualified pages to be fans or centers. There are also other pruning methods given in [293].

1. **Pruning by in-degree:** we can delete all pages that are very highly referenced (linked) on the Web, such as homepages of Web portals (e.g., Yahoo!, AOL, etc). These pages are referenced for a variety of reasons, having little to do with any single emerging community, and they can be safely deleted. That is, we delete pages with the number of in-links great than k, which is determined empirically ($k = 50$ in [293]).
2. **Iterative pruning of fans and centers:** If we are interested in finding (i, j) cores, clearly any potential fan with an out-degree smaller than j can be pruned and the associated edges deleted from the graph. Similarly, any potential center with an in-degree smaller than i can be pruned and the corresponding edges deleted from the graph. This process can be done iteratively: when a fan gets pruned, some of the centers that it points to may have their in-degrees fall below the threshold i and qualify for pruning as a result. Similarly, when a center gets pruned, a fan that points to it could have its out-degree fall below its threshold of j and qualify for pruning.

Step 2: Generating all (i, j) Cores

After pruning, the remaining pages are used to discover cores. The method works as follows: Fixing j, we start with all (1, j) cores. This is simply the set of all vertices with out-degree at least j. We then construct all (2, j) cores by checking every fan which also points to any center in a (1, j) core. All (3, j) cores can be found in the same fashion by checking every fan which points to any center in a (2, j) core, and so on. The idea is similar to the Apriori algorithm for association rule mining (see Chap. 2) as every proper subset of the fans in any (i, j) core forms a core of smaller size.

Based on the algorithm, Kumar et al. found a large number of topic coherent cores from a crawl of the Web [293]. We note that this algorithm only finds the core pages of the communities, not all members (pages). It also does not find the themes of the communities or their hierarchical organizations.

7.5.3 Maximum Flow Communities

Bipartite cores are usually very small and do not represent full communities. In this section, we define and find maximum flow communities based on the work of Flake et al. [180]. The algorithm requires the user to give a

set of seed pages, which are examples of the community that the user wishes to find.

Given a Web link graph $G = (V, E)$, a maximum flow community is defined as a collection $C \subset V$ of Web pages such that each member page $u \in C$ has more hyperlinks (in either direction) within the community C than outside of the community V-C. Identifying such a community is intractable in the general case because it can be mapped into a family of NP-complete graph partition problems. Thus, we need to approximate and recast it into a framework with less stringent conditions based on the network flow model from operations research, specifically the maximum flow model.

The maximum flow model can be stated as follows: We are given a graph $G = (V, E)$, where each edge (u, v) is thought of as having a positive capacity $c(u, v)$ that limits the quantity of a product that may be shipped through the edge. In such a situation, it is often desirable to have the maximum amount of flow from a starting point s (called the **source**) and a terminal point t (called the **sink**). Intuitively, the maximum flow of the graph is determined by the bottleneck edges. For example, given the graph in Fig. 7.12 with the source s and the sink t, if every edge has the unit capacity, the bottleneck edges are W-X and Y-Z.

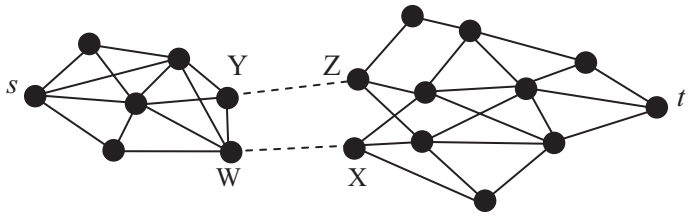


Fig. 7.12. A simple flow network.

The **Max Flow-Min Cut** theorem of Ford and Fulkerson [181] proves that the maximum flow of a network is identical to the minimum cut that separates s and t. Many polynomial time algorithms exist for solving the s-t maximum flow problem. If Fig. 7.12 is a Web link graph, it is natural to cut the edges W-X and Y-Z to produce two Web communities.

The basic idea of the approach in [180] is as follows: It starts with a set S of *seed* pages, which are example pages of the community that the user wishes to find. The system then crawls the Web to find more pages using the seed pages. A maximum flow algorithm is then applied to separate the community C involving the seed pages and the other pages. These steps may need to be repeated in order to find the desired community. Figure 7.13 gives the algorithm.

```
Algorithm Find-Community (S)
  while number of iteration is less than desired do
        build G = (V, E) by doing a fixed depth crawl starting from S;
        k = |S|;
        C = Max-Flow-Community(G, S, k);
        rank all v ∈ C by the number of edges in C;
        add the highest ranked non-seed vertices to S
  end-while
  return all v ∈ V still connected to the source s

Procedure Max-Flow-Community(G, S, k)
  create artificial vertices, s and t and add to V;        // V is the vertex set of G.
  for all v ∈ S do
      add (s, v) to E with c(s, v) = ∞                     // E is the edge set of G.
  endfor
  for all (u, v) ∈ E, u ≠ s do
      c(u, v) = k;
      if (v, u) ∉ E then
         add (v, u) to E with c(v, u) = k
      endif
  endfor
  for all v ∈ V, v ∉ S ∪ {s, t} do
      add (v, t) to E with c(v, t) = 1
  endfor
  Max-Flow(G, s, t);
  return all v ∈ V still connected to s.
```

Fig. 7.13. The algorithm for mining maximum flow communities

The algorithm Find-Community is the control program. It takes a set S of seed Web pages as input, and crawls to a fixed depth including in-links as well as out-links (with in-links found by querying a search engine). It then applies the procedure Max-Flow-Community to the induced graph G from the crawl. After a community C is found, it ranks the pages in the community by the number of edges that each has inside of the community. Some highest ranked non-seed pages are added to the seed set. This is to create a big seed set for the next iteration in order to crawl more pages. The algorithm then iterates the procedure. Note that the first iteration may only identify a very small community. However, when new seeds are added, increasingly larger communities are identified. Heuristics are used to decide when to stop.

The procedure Max-Flow-Community finds the actual community from G. Since a Web graph has no source and sink, it first augments the web

graph by adding an artificial source, s, with infinite capacity edges routed to all seed vertices in S; making each pre-existing edge bidirectional and assigning each edge a constant capacity k. It then adds an artificial sink t and routes all vertices except the source, the sink, and the seed vertices to t with unit capacity. After augmenting the web graph, a residual flow graph is produced by a maximum flow procedure (Max-Flow()). All vertices accessible from s through non-zero positive edges form the desired result. The value k is heuristically chosen to be the size of the set S to ensure that after the artificial source and sink are added to the original graph, the same cuts will be produced as the original graph (see the proof in [179]). Figure 7.14 shows the community finding process.

Finally, we note that this algorithm does not find the theme of the community or the community hierarchy (i.e., sub-communities and so on).

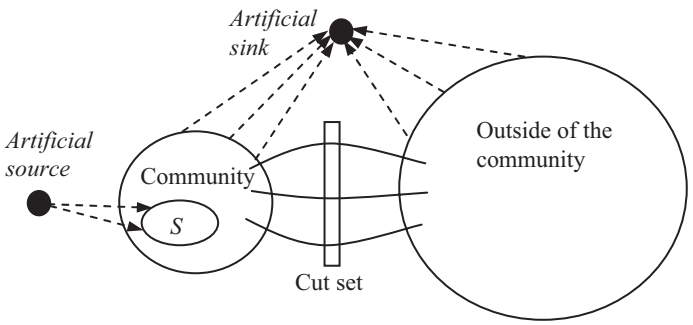


Fig. 7.14. Schematic representation of the community finding process

7.5.4 Email Communities Based on Betweenness

Email has become the predominant means of communication in the information age. It has been established as an indicator of collaboration and knowledge (or information) exchange. Email exchanges provide plenty of data on personal communication for the discovery of shared interests and relationships between people, which were hard to discover previously.

It is fairly straightforward to construct a graph based on email data. People are the vertices and the edges are added between people who corresponded through email. Usually, the edge between two people is added if a minimum number of messages passed between them. The minimum number is controlled by a threshold, which can be tuned.

To analyze an email graph or network, one can make use of all the centrality measures and prestige measures discussed in Sect. 7.1. We now focus on community finding only.

We are interested in people communities, which are subsets of vertices that are related. One way to identify communities is by partitioning the graph into discrete clusters such that there are few edges lying between the clusters. This definition is similar to that of the maximum flow community. **Betweenness** in social networks is a natural measure for identifying those edges in between clusters or communities [523]. The idea is that inter-community links, which are few, have high betweenness values, while the intra-community edges have low betweenness values. However, the betweenness discussed in Sect. 7.1 is evaluated on each person in the network. Here, we need to evaluate the betweenness of each edge. The idea is basically the same and Equation (4) can be used here without normalization because we only find communities in a single graph. The betweenness of an edge is simply the number of shortest paths that pass it.

If the graph is not connected, we identify communities from each connected component. Given a connected graph, the method works iteratively in two steps (Fig. 7.15):

repeat
  Compute the betweenness of each edge in the remaining graph;
  Remove the edge with the highest betweenness
until the graph is suitably partitioned.

Fig. 7.15. Community finding using the betweenness measure.

Since the removal of an edge can strongly affect the betweenness of many other edges, we need to repeatedly re-compute the betweenness of all edges. The idea of the method is very similar to the minimum-cut method discussed in Sect. 7.5.3.

The stopping criteria can be designed according to applications. In general, we consider that the smallest community is a triangle. The algorithm should stop producing more unconnected components if there is no way to generate triangle communities. A component of five or fewer vertices cannot consist of two viable communities. The smallest such component is six, which has two triangles connected by one edge, see Fig. 7.16. If any discovered community does not have a triangle, it may not be considered as a community. Clearly, other stopping criteria can be used.

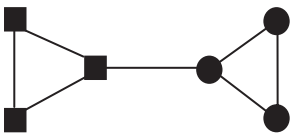


Fig. 7.16. The smallest possible graph of two viable communities.

7.5.5 Overlapping Communities of Named Entities

Most community discovery algorithms are based on graph partitioning, which means that an entity can belong to only a single community. However, in real life, a person can be in multiple communities (see the definition in Sect. 7.5.1). For example, he/she can be in the community of his/her family, the community of his/her colleagues and the community of his/her friends. A heuristic technique is presented in [325] for finding overlapping communities of entities in text documents.

In the Web or email context, there are explicit links connecting entities and forming communities. In free text documents, no explicit links exist. Then the question is: what constitutes a link between two entities in text documents? As we indicated earlier, one simple technique is to regard two entities as being linked if they co-occur in the same sentence. This method is reasonable because if two people are mentioned in a sentence there is usually a relationship between them.

The objective is to find entity communities from a text corpus, which could be a set of given documents or the returned pages from a search engine using a given entity as the search query. An entity here refers to the name of a person or an organization.

The algorithm in [325] consists of four steps:

1. Building a link graph: The algorithm first parses each document. For each sentence, it identifies named entities contained in the sentence. If a sentence has more than one named entities, these entities are pair-wise linked. The keywords in the sentence are attached to the linked pairs to form their textual contents. All the other sentences are discarded.
2. Finding all triangles: The algorithm then finds all triangles, which are the basic building blocks of communities. A triangle consists of three entities bound together. The reason for using triangles is that it has been observed by researchers that a community expands predominantly by triangles sharing a common edge.
3. Finding community cores: It next finds community cores. A community core is a group of tightly bound triangles, which are relaxed complete sub-graphs (or cliques). Intuitively, a core consists of a set of tightly connected members of a community.
4. Clustering around community cores: For those triangles and also entity pairs that are not in any core, they are assigned to cores according to their textual content similarities with the discovered cores.

It is clear that in this algorithm a single entity can appear in multiple communities because an entity can appear in multiple triangles. To finish off, the algorithm also ranks the entities in each community according to de-

gree centrality. Keywords associated with the edges of each community are also ranked. The top keywords are assumed to represent the theme of the community. The technique has been applied to find communities of political figures and celebrities from Web documents with promising results.

Bibliographic Notes

Social network analysis has a relative long history. A large number of interesting problems and algorithms were studied in the past 60 years. The book by Wasserman and Faust [540] is an authoritative text of the field. Co-citation [494] and bibliographic coupling [275] are from bibliometrics, which is a type of research method used in library and information science. The book edited by Borgman [58] is a good source of information on both the research and applications of bibliometrics.

The use of social network analysis in the Web context (also called link analysis) started with the PageRank algorithm proposed by Brin and Page [68] and Page et al. [422], and the HITS algorithm proposed by Kleinberg [281]. PageRank is also the algorithm that powers the Google search engine. Due to several weaknesses of HITS, many researchers have tried to improve it. Various enhancements were reported by Lempel and Moran [310], Bharat and Henzinger [52], Chakrabarti et al. [88], Cai et al. [78], etc. The book by Langville and Meyer [304] contains in-depth analyses of PageRank, HITS and many enhancements to HITS. Other works related to Web link analysis include those in [98, 226, 266, 368] on improving the PageRank computation, in [168] on searching workspace Web, in [103, 182, 183, 416] on the evolution of the Web and the search engine influence on the Web, in [140, 142, 410, 516] on other link based models, in [34, 440, 370, 371] on Web graph and its characteristics, in [37, 51, 235] on sampling of Web pages, and in [32, 425, 585] on the temporal dimension of Web search.

On community discovery, HITS can find some communities by computing non-principal eigenvectors [198, 281]. Kumar et al. [293] proposed the algorithm for finding bipartite cores. Flake et al. [179] introduced the maximum flow community mining. Ino et al. [249] presented a more strict definition of communities. Tyler et al. [523] gave the method for finding email communities based on betweenness. The algorithm for finding overlapping communities of named entities from texts was given by Li et al. [325]. More recent developments on communities and social networks on the Web can be found in [16, 21, 137, 158, 200, 518, 519, 561, 618].

Web Crawling

1

Introduction

A *web crawler* (also known as a *robot* or a *spider*) is a system for the bulk downloading of web pages. Web crawlers are used for a variety of purposes. Most prominently, they are one of the main components of web search engines, systems that assemble a corpus of web pages, index them, and allow users to issue queries against the index and find the web pages that match the queries. A related use is web archiving (a service provided by e.g., the Internet archive [77]), where large sets of web pages are periodically collected and archived for posterity. A third use is web data mining, where web pages are analyzed for statistical properties, or where data analytics is performed on them (an example would be Attributor [7], a company that monitors the web for copyright and trademark infringements). Finally, web monitoring services allow their clients to submit standing queries, or *triggers*, and they continuously crawl the web and notify clients of pages that match those queries (an example would be GigaAlert [64]).

The raison d'être for web crawlers lies in the fact that the web is not a centrally managed repository of information, but rather consists

of hundreds of millions of independent web content providers, each one providing their own services, and many competing with one another. In other words, the web can be viewed as a federated information repository, held together by a set of agreed-upon protocols and data formats, such as the Transmission Control Protocol (TCP), the Domain Name Service (DNS), the Hypertext Transfer Protocol (HTTP), the Hypertext Markup Language (HTML) and the robots exclusion protocol. So, content aggregators (such as search engines or web data miners) have two choices: They can either adopt a pull model where they will proactively scour the web for new or updated information, or they could try to establish a convention and a set of protocols enabling content providers to push content of interest to the aggregators. Indeed, the Harvest system [24], one of the earliest search services, adopted such a push model. However, this approach did not succeed, and virtually all content aggregators adopted the pull approach, with a few provisos to allow content providers to exclude all or part of their content from being crawled (the robots exclusion protocol) and to provide hints about their content, its importance and its rate of change (the Sitemaps protocol [110]).

There are several reasons why the push model did not become the primary means of acquiring content for search engines and other content aggregators: The fact that web servers are highly autonomous means that the barrier of entry to becoming a content provider is quite low, and the fact that the web protocols were at least initially extremely simple lowered the barrier even further — in fact, this simplicity is viewed by many as the reason why the web succeeded where earlier hypertext systems had failed. Adding push protocols would have complicated the set of web protocols and thus raised the barrier of entry for content providers, while the pull model does not require any extra protocols. By the same token, the pull model lowers the barrier of entry for content aggregators as well: Launching a crawler does not require any a priori buy-in from content providers, and indeed there are over 1,500 operating crawlers [47], extending far beyond the systems employed by the big search engines. Finally, the push model requires a trust relationship between content provider and content aggregator, something that is not given on the web at large — indeed, the relationship between

content providers and search engines is characterized by both mutual dependence and adversarial dynamics (see Section 6).

1.1 Challenges

The basic web crawling algorithm is simple: Given a set of seed Uniform Resource Locators (URLs), a crawler downloads all the web pages addressed by the URLs, extracts the hyperlinks contained in the pages, and iteratively downloads the web pages addressed by these hyperlinks. Despite the apparent simplicity of this basic algorithm, web crawling has many inherent challenges:

- **Scale.** The web is very large and continually evolving. Crawlers that seek broad coverage and good freshness must achieve extremely high throughput, which poses many difficult engineering problems. Modern search engine companies employ thousands of computers and dozens of high-speed network links.
- **Content selection tradeoffs.** Even the highest-throughput crawlers do not purport to crawl the whole web, or keep up with all the changes. Instead, crawling is performed selectively and in a carefully controlled order. The goals are to acquire high-value content quickly, ensure eventual coverage of all reasonable content, and bypass low-quality, irrelevant, redundant, and malicious content. The crawler must balance competing objectives such as coverage and freshness, while obeying constraints such as per-site rate limitations. A balance must also be struck between exploration of potentially useful content, and exploitation of content already known to be useful.
- **Social obligations.** Crawlers should be "good citizens" of the web, i.e., not impose too much of a burden on the web sites they crawl. In fact, without the right safety mechanisms a high-throughput crawler can inadvertently carry out a denial-of-service attack.
- **Adversaries.** Some content providers seek to inject useless or misleading content into the corpus assembled by

the crawler. Such behavior is often motivated by financial incentives, for example (mis)directing traffic to commercial web sites.

1.2 Outline

Web crawling is a many-faceted topic, and as with most interesting topics it cannot be split into fully orthogonal subtopics. Bearing that in mind, we structure the survey according to five relatively distinct lines of work that occur in the literature:

- Building an efficient, robust and scalable crawler (Section 2).
- Selecting a traversal order of the web graph, assuming content is well-behaved and is interconnected via HTML hyperlinks (Section 4).
- Scheduling revisitation of previously crawled content (Section 5).
- Avoiding problematic and undesirable content (Section 6).
- Crawling so-called "deep web" content, which must be accessed via HTML forms rather than hyperlinks (Section 7).

Section 3 introduces the theoretical crawl ordering problem studied in Sections 4 and 5, and describes structural and evolutionary properties of the web that influence crawl ordering. Section 8 gives a list of open problems.

2

Crawler Architecture

This section first presents a chronology of web crawler development, and then describes the general architecture and key design points of modern scalable crawlers.

2.1 Chronology

Web crawlers are almost as old as the web itself. In the spring of 1993, shortly after the launch of NCSA Mosaic, Matthew Gray implemented the World Wide Web Wanderer [67]. The Wanderer was written in Perl and ran on a single machine. It was used until 1996 to collect statistics about the evolution of the web. Moreover, the pages crawled by the Wanderer were compiled into an index (the "Wandex"), thus giving rise to the first search engine on the web. In December 1993, three more crawler-based Internet Search engines became available: JumpStation (implemented by Jonathan Fletcher; the design has not been written up), the World Wide Web Worm [90], and the RBSE spider [57]. WebCrawler [108] joined the field in April 1994, and MOMspider [61] was described the same year. This first generation of crawlers identified some of the defining issues in web crawler design. For example, MOM-

spider considered *politeness* policies: It limited the rate of requests to each site, it allowed web sites to exclude themselves from purview through the nascent robots exclusion protocol [83], and it provided a "black-list" mechanism that allowed the crawl operator to exclude sites. WebCrawler supported parallel downloading of web pages by structuring the system into a central crawl manager and 15 separate downloading processes. However, the design of these early crawlers did not focus on scalability, and several of them (RBSE spider and WebCrawler) used general-purpose database management systems to store the state of the crawl. Even the original Lycos crawler [89] ran on a single machine, was written in Perl, and used Perl's associative arrays (spilt onto disk using the DBM database manager) to maintain the set of URLs to crawl.

The following few years saw the arrival of several commercial search engines (Lycos, Infoseek, Excite, AltaVista, and HotBot), all of which used crawlers to index tens of millions of pages; however, the design of these crawlers remains undocumented.

Mike Burner's description of the Internet Archive crawler [29] was the first paper that focused on the challenges caused by the scale of the web. The Internet Archive crawling system was designed to crawl on the order of 100 million URLs. At this scale, it is no longer possible to maintain all the required data in main memory. The solution proposed by the IA paper was to crawl on a site-by-site basis, and to partition the data structures accordingly. The list of URLs to be crawled was implemented as a disk-based queue per web site. To avoid adding multiple instances of the same URL to the queue, the IA crawler maintained an in-memory Bloom filter [20] of all the site's URLs discovered so far. The crawl progressed by dequeuing a URL, downloading the associated page, extracting all links, enqueuing freshly discovered on-site links, writing all off-site links to disk, and iterating. Each crawling process crawled 64 sites in parallel, using non-blocking input/output (I/O) and a single thread of control. Occasionally, a batch process would integrate the off-site link information into the various queues. The IA design made it very easy to throttle requests to a given host, thereby addressing politeness concerns, and DNS and robot exclusion lookups for a given web site were amortized over all the site's URLs crawled in a single round. However, it is not clear whether the batch

process of integrating off-site links into the per-site queues would scale to substantially larger web crawls.

Brin and Page's 1998 paper outlining the architecture of the first-generation Google [25] system contains a short description of their crawler. The original Google crawling system consisted of a single URLserver process that maintained the state of the crawl, and around four crawling processes that downloaded pages. Both URLserver and crawlers were implemented in Python. The crawling process used asynchronous I/O and would typically perform about 300 downloads in parallel. The peak download rate was about 100 pages per second, with an average size of 6 KB per page. Brin and Page identified social aspects of crawling (e.g., dealing with web masters' complaints) as a major challenge in operating a crawling system.

With the Mercator web crawler, Heydon and Najork presented a "blueprint design" for web crawlers [75, 94]. Mercator was written in Java, highly scalable, and easily extensible. The first version [75] was non-distributed; a later distributed version [94] partitioned the URL space over the crawlers according to host name, and avoided the potential bottleneck of a centralized URL server. The second Mercator paper gave statistics of a 17-day, four-machine crawl that covered 891 million pages. Mercator was used in a number of web mining projects [27, 60, 71, 72, 95], and in 2001 replaced the first-generation AltaVista crawler.

Shkapenyuk and Suel's Polybot web crawler [111] represents another "blueprint design." Polybot is a distributed system, consisting of a crawl manager process, multiple downloader processes, and a DNS resolver process. The paper describes scalable data structures for the URL frontier and the "seen-URL" set used to avoid crawling the same URL multiple times; it also discusses techniques for ensuring politeness without slowing down the crawl. Polybot was able to download 120 million pages over 18 days using four machines.

The IBM WebFountain crawler [56] represented another industrial-strength design. The WebFountain crawler was fully distributed. The three major components were multi-threaded crawling processes ("Ants"), duplicate detection processes responsible for identifying downloaded pages with near-duplicate content, and a central controller

process responsible for assigning work to the Ants and for monitoring the overall state of the system. WebFountain featured a very flexible crawl scheduling mechanism that allowed URLs to be prioritized, maintained a politeness policy, and even allowed the policy to be changed on the fly. It was designed from the ground up to support incremental crawling, i.e., the process of recrawling pages regularly based on their historical change rate. The WebFountain crawler was written in C++ and used MPI (the Message Passing Interface) to facilitate communication between the various processes. It was reportedly deployed on a cluster of 48 crawling machines [68].

UbiCrawler [21] is another scalable distributed web crawler. It uses consistent hashing to partition URLs according to their host component across crawling machines, leading to graceful performance degradation in the event of the failure of a crawling machine. UbiCrawler was able to download about 10 million pages per day using five crawling machines. UbiCrawler has been used for studies of properties of the African web [22] and to compile several reference collections of web pages [118].

Recently, Yan et al. described IRLbot [84], a single-process web crawler that is able to scale to extremely large web collections without performance degradation. IRLbot features a "seen-URL" data structure that uses only a fixed amount of main memory, and whose performance does not degrade as it grows. The paper describes a crawl that ran over two months and downloaded about 6.4 billion web pages. In addition, the authors address the issue of crawler traps (web sites with a large, possibly infinite number of low-utility pages, see Section 6.2), and propose ways to ameliorate the impact of such sites on the crawling process.

Finally, there are a number of open-source crawlers, two of which deserve special mention. Heritrix [78, 93] is the crawler used by the Internet Archive. It is written in Java and highly componentized, and its design is quite similar to that of Mercator. Heritrix is multithreaded, but not distributed, and as such suitable for conducting moderately sized crawls. The Nutch crawler [62, 81] is written in Java as well. It supports distributed operation and should therefore be suitable for very large crawls; but as of the writing of [81] it has not been scaled beyond 100 million pages.

2.2 Architecture Overview

Figure 2.1 shows the high-level architecture of a prototypical distributed web crawler. The crawler consists of multiple processes running on different machines connected by a high-speed network. Each crawling process consists of multiple worker threads, and each worker thread performs repeated work cycles.

At the beginning of each work cycle, a worker obtains a URL from the *Frontier* data structure, which dispenses URLs according to their priority and to politeness policies. The worker thread then invokes the *HTTP fetcher*. The fetcher first calls a DNS sub-module to resolve the host component of the URL into the IP address of the corresponding web server (using cached results of prior resolutions if possible), and then connects to the web server, checks for any robots exclusion rules (which typically are cached as well), and attempts to download the web page.

If the download succeeds, the web page may or may not be stored in a repository of harvested web pages (not shown). In either case, the page is passed to the *Link extractor*, which parses the page's HTML content and extracts hyperlinks contained therein. The corresponding URLs are then passed to a *URL distributor*, which assigns each URL to a crawling process. This assignment is typically made by hashing the URLs host component, its domain, or its IP address (the latter requires additional DNS resolutions). Since most hyperlinks refer to pages on the same web site, assignment to the local crawling process is the common case.

Next, the URL passes through the *Custom URL filter* (e.g., to exclude URLs belonging to "black-listed" sites, or URLs with particular file extensions that are not of interest) and into the *Duplicate URL eliminator*, which maintains the set of all URLs discovered so far and passes on only never-before-seen URLs. Finally, the *URL prioritizer* selects a position for the URL in the Frontier, based on factors such as estimated page importance or rate of change.[1]

[1] Change rates play a role in *incremental* crawlers (Section 2.3.5), which route fetched URLs back to the prioritizer and Frontier.

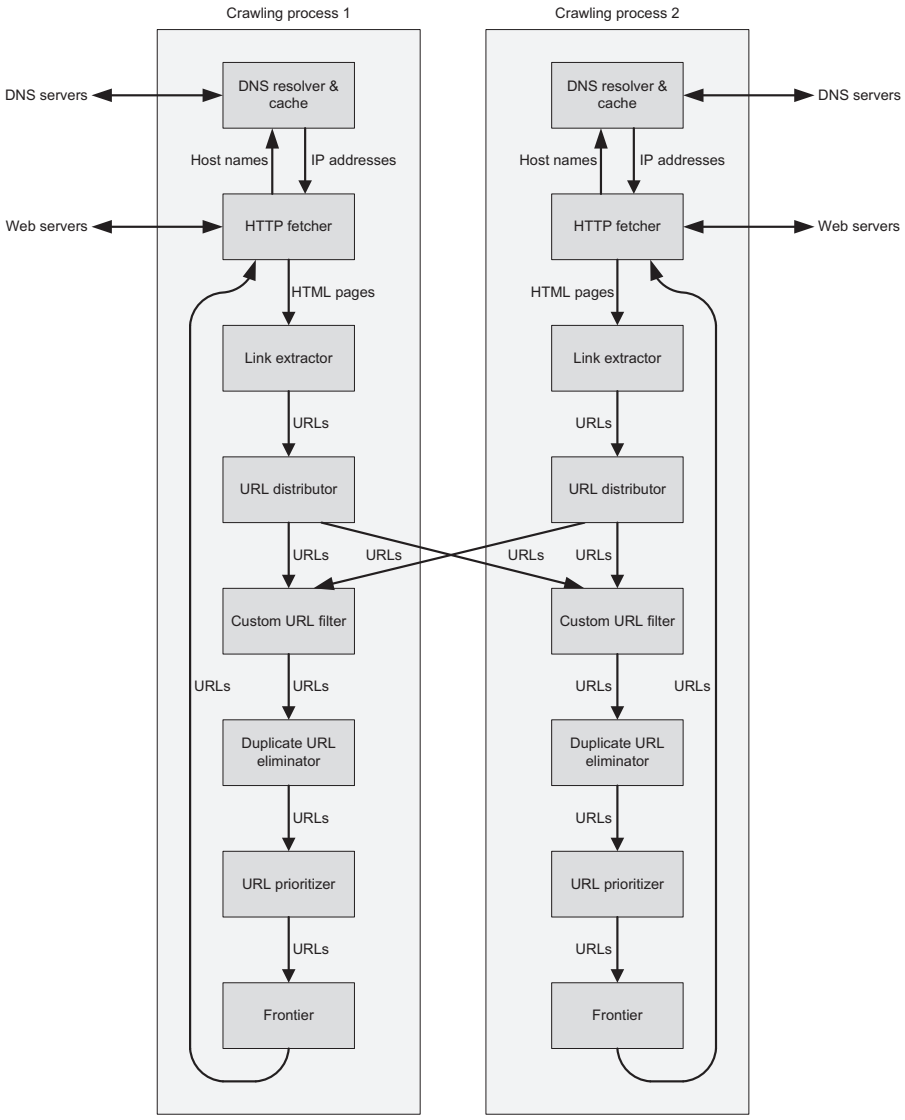


Fig. 2.1 Basic crawler architecture.

2.3 Key Design Points

Web crawlers download web pages by starting from one or more *seed URLs*, downloading each of the associated pages, extracting the

hyperlink URLs contained therein, and recursively downloading those pages. Therefore, any web crawler needs to keep track both of the URLs that are to be downloaded, as well as those that have already been downloaded (to avoid unintentionally downloading the same page repeatedly). The required state is a set of URLs, each associated with a flag indicating whether the page has been downloaded. The operations that must be supported are: Adding a new URL, retrieving a URL, marking a URL as downloaded, and testing whether the set contains a URL. There are many alternative in-memory data structures (e.g., trees or sorted lists) that support these operations. However, such an implementation does not scale to web corpus sizes that exceed the amount of memory available on a single machine.

To scale beyond this limitation, one could either maintain the data structure (e.g., the tree or sorted list) on disk, or use an off-the-shelf database management system. Either solution allows maintaining set sizes that exceed main memory; however, the cost of accessing items in the set (particularly for the purpose of set membership test) typically involves a disk seek, making it a fairly expensive operation. To achieve high performance, a more specialized approach is needed.

Virtually every modern web crawler splits the crawl state into two major data structures: One data structure for maintaining the set of URLs that have been discovered (whether downloaded or not), and a second data structure for maintaining the set of URLs that have yet to be downloaded. The first data structure (sometimes called the "URL-seen test" or the "duplicated URL eliminator") must support set addition and set membership testing, while the second data structure (usually called the *frontier*) must support adding URLs, and selecting a URL to fetch next.

2.3.1 Frontier Data Structure and Politeness

A straightforward implementation of the frontier data structure is a First-in-First-out (FIFO) queue. Such an implementation results in a breadth-first traversal of the web graph. However, this simple approach has drawbacks: Most hyperlinks on the web are "relative" (i.e., refer to another page on the same web server). Therefore, a frontier realized

as a FIFO queue contains long runs of URLs referring to pages on the same web server, resulting in the crawler issuing many consecutive HTTP requests to that server. A barrage of requests in short order is considered "impolite," and may be construed as a denial-of-service attack on the web server. On the other hand, it would be wasteful for the web crawler to space out requests to the same server without doing other useful work in the meantime. This problem is compounded in a multithreaded or distributed crawler that issues many HTTP requests in parallel.

Most web crawlers obey a policy of not issuing multiple overlapping requests to the same server. An easy way to realize this is to maintain a mapping from web servers to crawling threads, e.g., by hashing the host component of each URL.[2] In this design, each crawling thread has a separate FIFO queue, and downloads only URLs obtained from that queue.

A more conservative politeness policy is to space out requests to each web server according to that server's capabilities. For example, a crawler may have a policy to delay subsequent requests to a server by a multiple (say 10×) of the time it took to download the last page from that server. This policy ensures that the crawler consumes a bounded fraction of the web server's resources. It also means that in a given time interval, fewer pages will be downloaded from slow or poorly connected web servers than from fast, responsive web servers. In other words, this crawling policy is biased toward well-provisioned web sites. Such a policy is well-suited to the objectives of search engines, since large and popular web sites tend also to be well-provisioned.

The Mercator web crawler implemented such an adaptive politeness policy. It divided the frontier into two parts, a "front end" and a "back end." The front end consisted of a single queue Q, and URLs were added to the frontier by enqueuing them into that queue. The back

[2] To amortize hardware cost, many web servers use *virtual hosting*, meaning that multiple symbolic host names resolve to the same IP address. Simply hashing the host component of each URL to govern politeness has the potential to overload such web servers. A better scheme is to resolve the URLs symbolic host name to an IP address and use a hash of that address to assign URLs to a queue. The drawback of that approach is that the latency of DNS resolution can be high (see Section 2.3.3), but fortunately there tends to be a high amount of locality in the stream of discovered host names, thereby making caching effective.

end consisted of many separate queues; typically three times as many queues as crawling threads. Each queue contained URLs belonging to a single web server; a table T on the side maintained a mapping from web servers to back-end queues. In addition, associated with each back-end queue q was a time t at which the next URL from q may be processed. These (q, t) pairs were organized into an in-memory priority queue, with the pair with lowest t having the highest priority. Each crawling thread obtained a URL to download by removing the highest-priority entry (q, t) from the priority queue, waiting if necessary until time t had been reached, dequeuing the next URL u from q, downloading it, and finally reinserting the pair $(q, t_{now} + k \cdot x)$ into the priority queue, where t_{now} is the current time, x is the amount of time it took to download u, and k is a "politeness parameter"; typically 10. If dequeuing u from q left q empty, the crawling thread would remove the mapping from $host(u)$ to q from T, repeatedly dequeue a URL u' from Q and enqueue u' into the back-end queue identified by $T(host(u'))$, until it found a u' such that $host(u')$ was not contained in T. At this point, it would enqueue u' in q and update T to map $host(u')$ to q.

In addition to obeying politeness policies that govern the rate at which pages are downloaded from a given web site, web crawlers may also want to prioritize the URLs in the frontier. For example, it may be desirable to prioritize pages according to their estimated usefulness (based for example on their PageRank [101], the traffic they receive, the reputation of the web site, or the rate at which the page has been updated in the past). The page ordering question is discussed in Section 4.

Assuming a mechanism for assigning crawl priorities to web pages, a crawler can structure the frontier (or in the Mercator design described above, the front-end queue) as a disk-based priority queue ordered by usefulness. The standard implementation of a priority queue is a heap, and insertions into a heap of n elements require $\log(n)$ element accesses, each access potentially causing a disk seek, which would limit the data structure to a few hundred insertions per second — far less than the URL ingress required for high-performance crawling.

An alternative solution is to "discretize" priorities into a fixed number of priority levels (say 10 to 100 levels), and maintain a separate URL

FIFO queue for each level. A URL is assigned a discrete priority level, and inserted into the corresponding queue. To dequeue a URL, either the highest-priority nonempty queue is chosen, or a randomized policy biased toward higher-priority queues is employed.

2.3.2 URL Seen Test

As outlined above, the second major data structure in any modern crawler keeps track of the set of URLs that have been previously discovered and added to frontier. The purpose of this data structure is to avoid adding multiple instances of the same URL to the frontier; for this reason, it is sometimes called the *URL-seen test* (UST) or the *duplicate URL eliminator* (DUE). In a simple batch crawling setting in which pages are downloaded only once, the UST needs to support insertion and set membership testing; in a continuous crawling setting in which pages are periodically re-downloaded (see Section 2.3.5), it must also support deletion, in order to cope with URLs that no longer point to a valid page.

There are multiple straightforward in-memory implementations of a UST, e.g., a hash table or Bloom filter [20]. As mentioned above, in-memory implementations do not scale to arbitrarily large web corpora; however, they scale much further than in-memory implementations of the frontier, since each URL can be compressed to a much smaller token (e.g., a 10-byte hash value). Commercial search engines employ distributed crawlers (Section 2.3.4), and a hash table realizing the UST can be partitioned across the machines in the crawling cluster, further increasing the limit of how far such an in-memory implementation can be scaled out.

If memory is at a premium, the state of the UST must reside on disk. In a disk-based hash table, each lookup requires a disk seek, severely limiting the throughput. Caching popular URLs can increase the throughput by about an order of magnitude [27] to a few thousand lookups per second, but given that the average web page contains on the order of a hundred links and that each link needs to be tested for novelty, the crawling rate would still be limited to tens of pages per second under such an implementation.

While the *latency* of disk seeks is poor (a few hundred seeks per second), the *bandwidth* of disk reads and writes is quite high (on the order of 50–100 MB per second in modern disks). So, implementations performing random file accesses perform poorly, but those that perform streaming sequential reads or writes can achieve reasonable throughput. The Mercator crawler leveraged this observation by aggregating many set lookup and insertion operations into a single large batch, and processing this batch by sequentially reading a set of sorted URL hashes from disk and writing them (plus the hashes of previously undiscovered URLs) out to a new file [94].

This approach implies that the set membership is delayed: We only know whether a URL is new after the batch containing the URL has been merged with the disk file. Therefore, we cannot decide whether to add the URL to the frontier until the merge occurs, i.e., we need to retain all the URLs in a batch, not just their hashes. However, it is possible to store these URLs temporarily on disk and read them back at the conclusion of the merge (again using purely sequential I/O), once it is known that they had not previously been encountered and should thus be added to the frontier. Adding URLs to the frontier in a delayed fashion also means that there is a lower bound on how soon they can be crawled; however, given that the frontier is usually far larger than a DUE batch, this delay is imperceptible except for the most high-priority URLs.

The IRLbot crawler [84] uses a refinement of the Mercator scheme, where the batch of URLs arriving at the DUE is also written to disk, distributed over multiple files keyed by the prefix of each hash. Once the size of the largest file exceeds a certain threshold, the files that together hold the batch are read back into memory one by one and merge-sorted into the main URL hash file on disk. At the conclusion of the merge, URLs are forwarded to the frontier as in the Mercator scheme. Because IRLbot stores the batch on disk, the size of a single batch can be much larger than Mercator's in-memory batches, so the cost of the merge-sort with the main URL hash file is amortized over a much larger set of URLs.

In the Mercator scheme and its IRLbot variant, merging a batch of URLs into the disk-based hash file involves reading the entire old hash

file and writing out an updated version. Hence, the time requirement is proportional to the number of discovered URLs. A modification of this design is to store the URL hashes on disk in sorted order as before, but sparsely packed rather than densely packed. The k highest-order bits of a hash determine the disk block where this hash resides (if it is present). Merging a batch into the disk file is done in place, by reading a block for which there are hashes in the batch, checking which hashes are not present in that block, and writing the updated block back to disk. Thus, the time requirement for merging a batch is proportional to the size of the batch, not the number of discovered URLs (albeit with high constant due to disk seeks resulting from skipping disk blocks). Once any block in the file fills up completely, the disk file is rewritten to be twice as large, and each block contains hashes that now share their $k + 1$ highest-order bits.

2.3.3 Auxiliary Data Structures

In addition to the two main data structures discussed in Sections 2.3.1 and 2.3.2 — the frontier and the UST/DUE — web crawlers maintain various auxiliary data structures. We discuss two: The robots exclusion rule cache and the DNS cache.

Web crawlers are supposed to adhere to the *Robots Exclusion Protocol* [83], a convention that allows a web site administrator to bar web crawlers from crawling their site, or some pages within the site. This is done by providing a file at URL `/robots.txt` containing rules that specify which pages the crawler is allowed to download. Before attempting to crawl a site, a crawler should check whether the site supplies a `/robots.txt` file, and if so, adhere to its rules. Of course, downloading this file constitutes crawling activity in itself. To avoid repeatedly requesting `/robots.txt`, crawlers typically cache the results of previous requests of that file. To bound the size of that cache, entries must be discarded through some cache eviction policy (e.g., least-recently used); additionally, web servers can specify an expiration time for their `/robots.txt` file (via the HTTP `Expires` header), and cache entries should be discarded accordingly.

URLs contain a host component (e.g., `www.yahoo.com`), which is "resolved" using the *Domain Name Service* (DNS), a protocol that

exposes a globally distributed mapping from symbolic host names to IP addresses. DNS requests can take quite a long time due to the request-forwarding nature of the protocol. Therefore, crawlers often maintain their own DNS caches. As with the robots exclusion rule cache, entries are expired according to both a standard eviction policy (such as least-recently used), and to expiration directives.

2.3.4 Distributed Crawling

Web crawlers can be distributed over multiple machines to increase their throughput. This is done by partitioning the URL space, such that each crawler machine or *node* is responsible for a subset of the URLs on the web. The URL space is best partitioned across web site boundaries [40] (where a "web site" may refer to all URLs with the same symbolic host name, same domain, or same IP address). Partitioning the URL space across site boundaries makes it easy to obey politeness policies, since each crawling process can schedule downloads without having to communicate with other crawler nodes. Moreover, all the major data structures can easily be partitioned across site boundaries, i.e., the frontier, the DUE, and the DNS and robots exclusion caches of each node contain URL, robots exclusion rules, and name-to-address mappings associated with the sites assigned to that node, and nothing else.

Crawling processes download web pages and extract URLs, and thanks to the prevalence of relative links on the web, they will be themselves responsible for the large majority of extracted URLs. When a process extracts a URL u that falls under the responsibility of another crawler node, it forwards u to that node. Forwarding of URLs can be done through peer-to-peer TCP connections [94], a shared file system [70], or a central coordination process [25, 111]. The amount of communication with other crawler nodes can be reduced by maintaining a cache of popular URLs, used to avoid repeat forwardings [27].

Finally, a variant of distributed web crawling is *peer-to-peer* crawling [10, 87, 100, 112, 121], which spreads crawling over a loosely collaborating set of crawler nodes. Peer-to-peer crawlers typically employ some form of distributed hash table scheme to assign URLs to crawler

nodes, enabling them to cope with sporadic arrival and departure of crawling nodes.

2.3.5 Incremental Crawling

Web crawlers can be used to assemble one or more static snapshots of a web corpus (*batch crawling*), or to perform *incremental* or *continuous crawling*, where the resources of the crawler are divided between downloading newly discovered pages and re-downloading previously crawled pages. Efficient incremental crawling requires a few changes to the major data structures of the crawler. First, as mentioned in Section 2.3.2, the DUE should support the deletion of URLs that are no longer valid (e.g., that result in a 404 HTTP return code). Second, URLs are retrieved from the frontier and downloaded as in batch crawling, but they are subsequently reentered into the frontier. If the frontier allows URLs to be prioritized, the priority of a previously downloaded URL should be dependent on a model of the page's temporal behavior based on past observations (see Section 5). This functionality is best facilitated by augmenting URLs in the frontier with additional information, in particular previous priorities and compact sketches of their previous content. This extra information allows the crawler to compare the sketch of the just-downloaded page to that of the previous version, for example raising the priority if the page has changed and lowering it if it has not. In addition to content evolution, other factors such as page quality are also often taken into account; indeed there are many fast-changing "spam" web pages.

3

Crawl Ordering Problem

Aside from the intra-site politeness considerations discussed in Section 2, a crawler is free to visit URLs in any order. The crawl order is extremely significant, because for the purpose of crawling the web can be considered infinite — due to the growth rate of new content, and especially due to dynamically generated content [8]. Indeed, despite their impressive capacity, modern commercial search engines only index (and likely only crawl) a fraction of discoverable web pages [11]. The crawler ordering question is even more crucial for the countless smaller-scale crawlers that perform *scoped crawling* of targeted subsets of the web.

Sections 3–5 survey work on selecting a good crawler order, with a focus on two basic considerations:

- **Coverage.** The fraction of desired pages that the crawler acquires successfully.
- **Freshness.** The degree to which the acquired page snapshots remain up-to-date, relative to the current "live" web copies.

Issues related to redundant, malicious or misleading content are covered in Section 6. Generally speaking, techniques to avoid unwanted content

can be incorporated into the basic crawl ordering approaches without much difficulty.

3.1 Model

Most work on crawl ordering abstracts away the architectural details of a crawler (Section 2), and assumes that URLs in the frontier data structure can be reordered freely. The resulting simplified crawl ordering model is depicted in Figure 3.1. At a given point in time, some historical crawl order has already been executed (P_1, P_2, P_3, P_4, P_5 in the diagram), and some future crawl order has been planned ($P_6, P_7, P_4, P_8, \ldots$).[1]

In the model, all pages require the same amount of time to download; the (constant) rate of page downloading is called the *crawl rate*, typically measured in pages/second. (Section 2 discussed how to maximize the crawl rate; here it is assumed to be fixed.) The crawl rate is not relevant to batch crawl ordering methods, but it is a key factor when scheduling page revisitations in incremental crawling.

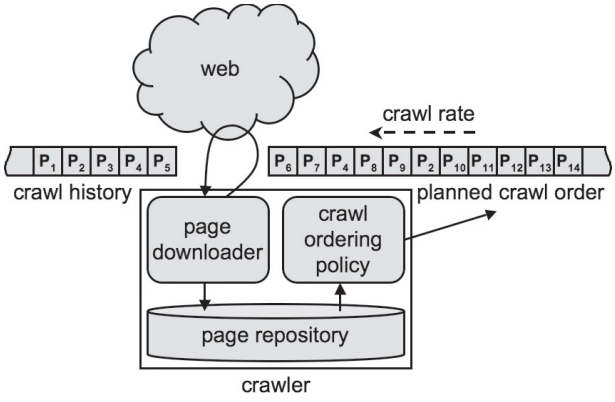


Fig. 3.1 Crawl ordering model.

[1] Some approaches treat the crawl ordering problem hierarchically, e.g., select a visitation order for web sites, and within each site select a page visitation order. This approach helps mitigate the complexity of managing a crawl ordering policy, and is well aligned with policies that rely primarily on site-level metrics such as site-level PageRank to drive crawl ordering decisions. Many of the insights about page-level crawl ordering also apply at the site level.

Pages downloaded by the crawler are stored in a *repository*. The future crawl order is determined, at least in part, by analyzing the repository. For example, one simple policy mentioned earlier, *breadth-first search*, extracts hyperlinks from pages entering the repository, identifies linked-to pages that are not already part of the (historical or planned) crawl order, and adds them to the end of the planned crawl order.

The content of a web page is subject to change over time, and it is sometimes desirable to re-download a page that has already been downloaded, to obtain a more recent snapshot of its content. As mentioned in Section 2.3.5, two approaches exist for managing repeated downloads:

- **Batch crawling.** The crawl order does not contain duplicate occurrences of any page, but the entire crawling process is periodically halted and restarted as a way to obtain more recent snapshots of previously crawled pages. Information gleaned from previous crawl iterations (e.g., page importance score estimates) may be fed to subsequent ones.
- **Incremental crawling.** Pages may appear multiple times in the crawl order, and crawling is a continuous process that conceptually never terminates.

It is believed that most modern commercial crawlers perform incremental crawling, which is more powerful because it allows re-visitation of pages at different rates. (A detailed comparison between incremental and batch crawling is made by Cho and García-Molina [39].)

3.1.1 Limitations

This model has led to a good deal of research with practical implications. However, as with all models, it simplifies reality. For one thing, as discussed in Section 2, a large-scale crawler maintains its frontier data structure on disk, which limits opportunities for reordering. Generally speaking, the approach of maintaining a prioritized ensemble of FIFO queues (see Section 2.3.1) can be used to approximate a desired crawl order. We revisit this issue in Sections 4.3 and 5.3.

Other real-world considerations that fall outside the model include:

- Some pages (or even versions of a page) take longer to download than others, due to differences in size and network latency.
- Crawlers take special care to space out downloads of pages from the same server, to obey politeness constraints, see Section 2.3.1. Crawl ordering policies that assume a single crawl rate constraint can, at least in principle, be applied on a per-server basis, i.e., run n independent copies of the policy for n servers.
- As described in Section 2, modern commercial crawlers utilize many simultaneous page downloader threads, running on many independent machines. Hence rather than a single totally ordered list of pages to download, it is more accurate to think of a set of parallel lists, encoding a partial order.
- Special care must be taken to avoid crawling redundant and malicious content; we treat these issues in Section 6.
- If the page repository runs out of space, and expanding it is not considered worthwhile, is becomes necessary to *retire* some of the pages stored there (although it may make sense to retain some metadata about the page, to avoid recrawling it). We are not aware of any scholarly work on how to select pages for retirement.

3.2 Web Characteristics

Before proceeding, we describe some structural and evolutionary properties of the web that are relevant to the crawl ordering question. The findings presented here are drawn from studies that used data sets of widely varying size and scope, taken at different dates over the span of a decade, and analyzed via a wide array of methods. Hence, caution is warranted in their interpretation.

3.2.1 Static Characteristics

Several studies of the structure of the *web graph*, in which pages are encoded as vertices and hyperlinks as directed edges, have been

conducted. One notable study is by Broder et al. [26], which uncovered a "bowtie" structure consisting of a central strongly connected component (the *core*), a component that can reach the core but cannot be reached from the core, and a component that can be reached from the core but cannot reach the core. (In addition to these three main components there are a number of small, irregular structures such as disconnected components and long "tendrils.")

Hence there exist many ordered pairs of pages (P_1, P_2) such that there is no way to reach P_2 by starting at P_1 and repeatedly following hyperlinks. Even in cases where P_2 is reachable from P_1, the distance can vary greatly, and in many cases hundreds of links must be traversed. The implications for crawling are: (1) one cannot simply crawl to depth N, for a reasonable value of N like $N = 20$, and be assured of covering the entire web graph; (2) crawling "seeds" (the pages at which a crawler commences) should be selected carefully, and multiple seeds may be necessary to ensure good coverage.

In an earlier study, Broder et al. [28] showed that there is an abundance of near-duplicate content of the web. Using a corpus of 30 million web pages collected by the AltaVista crawler, they used the *shingling* algorithm to cluster the corpus into groups of similar pages, and found that 29% of the pages were more than 50% similar to other pages in the corpus, and 11% of the pages were exact duplicates of other pages. Sources of near-duplication include mirroring of sites (or portions of sites) and URL synonymy, see Section 6.1.

Chang et al. [35] studied the "deep web," i.e., web sites whose content is not reachable via hyperlinks and instead can only be retrieved by submitting HTML forms. The findings include: (1) there are over one million deep web sites; (2) more deep web sites have structured (multi-field) query interfaces than unstructured (single-field) ones; and (3) most query interfaces are located within a few links of the root of a web site, and are thus easy to find by shallow crawling from the root page.

3.2.2 Temporal Characteristics

One of the objectives of crawling is to maintain freshness of the crawled corpus. Hence it is important to understand the temporal

characteristics of the web, both in terms of site-level evolution (the appearance and disappearance of pages on a site) and page-level evolution (changing content within a page).

3.2.2.1 Site-Level Evolution

Dasgupta et al. [48] and Ntoulas et al. [96] studied creation and retirement of pages and links inside a number of web sites, and found the following characteristics (these represent averages across many sites):

- New pages are created at a rate of 8% per week.
- Pages are retired at a rapid pace, such that during the course of one year 80% of pages disappear.
- New links are created at the rate of 25% per week, which is significantly faster than the rate of new page creation.
- Links are retired at about the same pace as pages, with 80% disappearing in the span of a year.
- It is possible to discover 90% of new pages by monitoring links spawned from a small, well-chosen set of old pages (for most sites, five or fewer pages suffice, although for some sites hundreds of pages must be monitored for this purpose). However, discovering the remaining 10% requires substantially more effort.

3.2.2.2 Page-Level Evolution

Some key findings about the frequency with which an individual web page undergoes a change are:

- Page change events are governed by a Poisson process, which means that changes occur randomly and independently, at least in the case of pages that change less frequently than once a day [39].[2]
- Page change frequencies span multiple orders of magnitude (sub-hourly, hourly, daily, weekly, monthly, annually), and each order of magnitude includes a substantial fraction of

[2] A Poisson change model was originally postulated by Coffman et al. [46].

pages on the web [2, 39]. This finding motivates the study of non-uniform page revisitation schedules.

- Change frequency is correlated with visitation frequency, URL depth, domain and topic [2], as well as page length [60].
- A page's change frequency tends to remain stationary over time, such that past change frequency is a fairly good predictor of future change frequency [60].

Unfortunately, it appears that there is no simple relationship between the frequency with which a page changes and the cumulative amount of content that changes over time. As one would expect, pages with moderate change frequency tend to exhibit a higher cumulative amount of changed content than pages with a low change frequency. However, pages with high change frequency tend to exhibit *less* cumulative change than pages with moderate change frequency. On the encouraging side, the amount of content that changed on a page in the past is a fairly good predictor of the amount of content that will change in the future (although the degree of predictability varies from web site to web site) [60, 96].

Many changes are confined to a small, contiguous region of a web page [60, 85], and/or only affect transient words that do not characterize the core, time-invariant theme of the page [2]. Much of the "new" content added to web pages is actually taken from other pages [96].

The temporal behavior of (regions of) web pages can be divided into three categories: *Static* (no changes), *churn* (new content supplants old content, e.g., quote of the day), and *scroll* (new content is appended to old content, e.g., blog entries). Simple generative models for the three categories collectively explain nearly all observed temporal web page behavior [99].

Most web pages include at least some static content, resulting in an upper bound on the divergence between an old snapshot of a page and the live copy. The shape of the curve leading to the upper bound depends on the mixture of churn and scroll content, and the rates of churning and scrolling. One simple way to characterize a page is with a pair of numbers: (1) the divergence upper bound (i.e., the amount of non-static content), under some divergence measure such

Technique	Objectives		Factors considered		
	Coverage	*Freshness*	*Importance*	*Relevance*	*Dynamicity*
Breadth-first search [43, 95, 108]	√				
Prioritize by indegree [43]	√		√		
Prioritize by PageRank [43, 45]	√		√		
Prioritize by site size [9]	√		√		
Prioritize by spawning rate [48]	√				√
Prioritize by search impact [104]	√		√	√	
Scoped crawling (Section 4.2)	√			√	
Minimize obsolescence [41, 46]		√	√		√
Minimize age [41]		√	√		√
Minimize incorrect content [99]		√	√		√
Minimize embarrassment [115]		√	√	√	√
Maximize search impact [103]		√	√	√	√
Update capture (Section 5.2)		√		√	√
WebFountain [56]	√	√			√
OPIC [1]	√	√	√		

Fig. 3.2 Taxonomy of crawl ordering techniques.

as shingle [28] or word difference; and (2) the amount of time it takes to reach the upper bound (i.e., the time taken for all non-static content to change) [2].

3.3 Taxonomy of Crawl Ordering Policies

Figure 3.2 presents a high-level taxonomy of published crawl ordering techniques. The first group of techniques focuses exclusively on ordering pages for first-time downloading, which affects *coverage*. These can be applied either in the batch crawling scenario, or in the incremental crawling scenario in conjunction with a separate policy governing re-downloading of pages to maintain *freshness*, which is the focus of the second group of techniques. Techniques in the third group consider the combined problem of interleaving first-time downloads with re-downloads, to balance coverage and freshness.

As reflected in Figure 3.2, crawl ordering decisions tend to be based on some combination of the following factors: (1) *importance* of a page or site, relative to others; (2) *relevance* of a page or site to the purpose served by the crawl; and (3) *dynamicity*, or how the content of a page/site tends to change over time.

Some crawl ordering techniques are broader than others in terms of which factors they consider and which objectives they target. Ones that focus narrowly on a specific aspect of crawling typically aim for a "better" solution with respect to that aspect, compared with broader "all-in-one" techniques. On the other hand, to be usable they may need to be extended or combined with other techniques. In some cases a straightforward extension exists (e.g., add importance weights to an importance-agnostic formula for scheduling revisitations), but often not. There is no published work on the best way to combine multiple specialized techniques into a comprehensive crawl ordering approach that does well across the board.

The next two chapters describe the techniques summarized in Figure 3.2, starting with ones geared toward batch crawling (Section 4), and then moving to incremental crawl ordering techniques (Section 5).

4

Batch Crawl Ordering

A batch crawler traverses links outward from an initial *seed set* of URLs. The seed set may be selected algorithmically, or by hand, based on criteria such as importance, outdegree, or other structural features of the web graph [120]. A common, simple approach is to use the root page of a web directory site such as OpenDirectory, which links to many important sites across a broad range of topics. After the seed set has been visited and links have been extracted from the seed set pages, the crawl ordering policy takes over.

The goal of the crawl ordering policy is to maximize the *weighted coverage* (WC) achieved over time, given a fixed crawl rate. WC is defined as:

$$\mathrm{WC}(t) = \sum_{p \in \mathcal{C}(t)} w(p),$$

where t denotes the time elapsed since the crawl began, $\mathcal{C}(t)$ denotes the set of pages crawled up to time t (under the fixed crawl rate assumption, $|\mathcal{C}(t)| \propto t$), and $w(p)$ denotes a numeric weight associated with page p. The weight function $w(p)$ is chosen to reflect the purpose of the crawl. For example, if the purpose is to crawl pages about helicopters, one sets $w(p) = 1$ for pages about helicopters, and $w(p) = 0$ for all other pages.

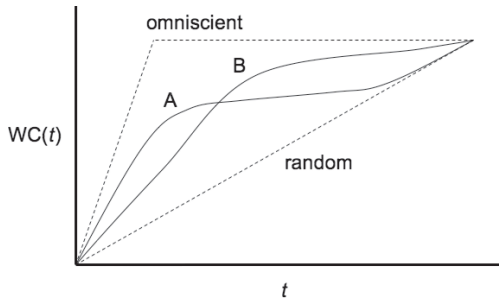


Fig. 4.1 Weighted coverage (WC) as a function of time elapsed (t) since the beginning of a batch crawl.

Figure 4.1 shows some hypothetical WC curves. Typically, $w(p) \geq 0$, and hence $\mathrm{WC}(t)$ is monotonic in t. Under a *random* crawl ordering policy, $\mathrm{WC}(t)$ is roughly linear in t; this line serves as a baseline upon which other policies strive to improve. An *omniscient* policy, which downloads pages in descending order of $w(p)$ yields a theoretical upper-bound curve. (For the helicopter example, the omniscient policy downloads all helicopter pages first, thereby achieving maximal WC before the end of the crawl.) Policies A and B fall in-between the random and omniscient cases, with A performing better in the early stages of the crawl, but B performing better toward the end. The choice between A and B depends on how long the crawl is allowed to run before being stopped (and possibly re-started to bring in a fresh batch of pages), and to what use, if any, pages obtained early in the crawl are put while the crawl is still in flight.

The above framework can be applied to *comprehensive* batch crawling, in which the goal is to achieve broad coverage of all types of content, as well as to *scoped* batch crawling, where the crawler restricts its attention to a relatively narrow slice of the web (e.g., pages about helicopters). This chapter examines the two scenarios in turn, focusing initially on crawl order effectiveness, with implementation and efficiency questions covered at the end.

4.1 Comprehensive Crawling

When the goal is to cover high-quality content of all varieties, a popular choice of weight function is $w(p) = \mathrm{PR}(p)$, where $\mathrm{PR}(p)$ is

p's importance score as measured by PageRank [101].[1] Variations on PageRank used in the crawling context include: Only counting *external* links, i.e., ones that go between two web sites (Najork and Wiener [95] discuss some tradeoffs involved in counting all links versus only external links); biasing the PageRank random jumps to go to a *trusted set* of pages believed not to engage in spamming (see Cho and Schonfeld [45] for discussion); or omitting random jumps entirely, as done by Abiteboul et al. [1].

In view of maximizing coverage weighted by PageRank or some variant, three main types of crawl ordering policies have been examined in the literature. In increasing order of complexity, they are:

- **Breadth-first search [108].** Pages are downloaded in the order in which they are first discovered, where discovery occurs via extracting all links from each page immediately after it is downloaded. Breadth-first crawling is appealing due to its simplicity, and it also affords an interesting coverage guarantee: In the case of PageRank that is biased to a small trusted page set T, a breadth-first crawl of depth d using T as its seed set achieves WC $\geq 1 - \alpha^{d+1}$ [45], where α is the PageRank damping parameter.
- **Prioritize by indegree [43].** The page with the highest number of incoming hyperlinks from previously downloaded pages, is downloaded next. Indegree is sometimes used as a low-complexity stand-in for PageRank, and is hence a natural candidate for crawl ordering under a PageRank-based objective.
- **Prioritize by PageRank (variant/estimate) [1, 43, 45].** Pages are downloaded in descending order of PageRank (or some variant), as estimated based on the pages and links acquired so far by the crawler. Straightforward application

[1] Although this cumulative PageRank measure has been used extensively in the literature, Boldi et al. [23] caution that absolute PageRank scores may not be especially meaningful, and contend that PageRank should be viewed as a way to establish relative page orderings. Moreover, they show that biasing a crawl toward pages with high PageRank in the crawled subgraph leads to subgraphs whose PageRank ordering differs substantially from the PageRank-induced ordering of the same pages in the full graph.

of this method involves recomputing PageRank scores after each download, or updating the PageRank scores incrementally [38]. Another option is to recompute PageRank scores only periodically, and rely on an approximation scheme between recomputations. Lastly, Abiteboul et al. [1] gave an efficient online method of estimating a variant of PageRank that does not include random jumps, designed for use in conjunction with a crawler.

The three published empirical studies that evaluate the above policies over real web data are listed in Figure 4.2 (Najork and Wiener [95] evaluated only breadth-first search). Under the objective of crawling high-PageRank pages early ($w(p) = \mathrm{PR}(p)$), the main findings from these studies are the following:

- Starting from high-PageRank seeds, breadth-first crawling performs well early in the crawl (low t), but not as well as the other policies later in the crawl (medium to high t).
- Perhaps unsurprisingly, prioritization by PageRank performs well throughout the entire crawl. The shortcut of only recomputing PageRank periodically leads to poor performance, but the online approximation scheme by Abiteboul et al. [1] performs well. Furthermore, in the context of repeated batch crawls, it is beneficial to use PageRank values from previous iterations to drive the current iteration.
- There is no consensus on prioritization by indegree: One study (Cho et al. [43]) found that it worked fairly well (almost as well as prioritization by PageRank), whereas another study (Baeza-Yates et al. [9]) found that it performed very

STUDY	DATA SET	DATA SIZE	PUB. YEAR
Cho et al. [43]	Stanford web	10^5	1998
Najork and Wiener [95]	general web	10^8	2001
Baeza-Yates et al. [9]	Chile and Greece	10^6	2005

Fig. 4.2 Empirical studies of batch crawl ordering policies.

> poorly. The reason given by Baeza-Yates et al. [9] for poor performance is that it is overly greedy in going after high-indegree pages, and therefore it takes a long time to find pages that have high indegree and PageRank yet are only discoverable via low-indegree pages. The two studies are over different web collections that differ in size by an order of magnitude, and are seven years apart in time.

In addition to the aforementioned results, Baeza-Yates et al. [9] proposed a crawl policy that gives priority to sites containing a large number of discovered but uncrawled URLs. According to their empirical study, which imposed per-site politeness constraints, toward the end of the crawl (high t) the proposed policy outperforms policies based on breadth-first search, indegree, and PageRank. The reason is that it avoids the problem of being left with a few very large sites at the end, which can cause a politeness bottleneck.

Baeza-Yates and Castillo [8] observed that although the web graph is effectively infinite, most user browsing activity is concentrated within a small distance of the root page of each web site. Arguably, a crawler should concentrate its activities there, and avoid exploring too deeply into any one site.

4.1.1 Search Relevance as the Crawling Objective

Fetterly et al. [58] and Pandey and Olston [104] argued that when the purpose of crawling is to supply content to a search engine, PageRank-weighted coverage may not be the most appropriate objective. According to the argument, it instead makes sense to crawl pages that would be viewed or clicked by search engine users, if present in the search index. For example, one may set out to crawl all pages that, if indexed, would appear in the top ten results of likely queries. Even if PageRank is one of the factors used to rank query results, the top result for query Q_1 may have lower PageRank than the eleventh result for some other query Q_2, especially if Q_2 pertains to a more established topic.

Fetterly et al. [58] evaluated four crawl ordering policies (breadth-first; prioritize by indegree; prioritize by trans-domain indegree;

prioritize by PageRank) under two relevance metrics:

- **MaxNDCG:** The total Normalized Distributed Cumulative Gain (NDCG) [79] score of a set of queries evaluated over the crawled pages, assuming optimal ranking.
- **Click count:** The total number of clicks the crawled pages attracted via a commercial search engine in some time period.

The main findings were that prioritization by PageRank is the most reliable and effective method on these metrics, and that imposing per-domain page limits boosts effectiveness.

Pandey and Olston [104] proposed a technique for explicitly ordering pages by expected relevance impact, under the objective of maximizing coverage weighted by the number of times a page appears among the top N results of a user query. The relatively high computational overhead of the technique is mitigated by concentrating on queries whose results are likely to be improved by crawling additional pages (deemed *needy queries*). Relevance of frontier pages to needy queries is estimated from cues found in URLs and referring anchortext, as first proposed in the context of scoped crawling [43, 74, 108], discussed next.

4.2 Scoped Crawling

A scoped crawler strives to limit crawling activities to pages that fall within a particular category or *scope*, thereby acquiring in-scope content much faster and more cheaply than via a comprehensive crawl. Scope may be defined according to *topic* (e.g., pages about aviation), *geography* (e.g., pages about locations in and around Oldenburg, Germany [6]), *format* (e.g., images and multimedia), *genre* (e.g., course syllabi [51]), *language* (e.g., pages in Portuguese [65]), or other aspects. (Broadly speaking, page importance, which is the primary crawl ordering criterion discussed in Section 4.1, can also be thought of as a form of scope.)

Usage scenarios for scoped crawling include mining tasks that call for crawling a particular type of content (e.g., images of animals), personalized search engines that focus on topics of interest to a particular user (e.g., aviation and gardening), and search engines for the

"deep web" that use a surface-web crawler to locate gateways to deep-web content (e.g., HTML form interfaces). In another scenario, one sets out to crawl the full web by deploying many small-scale crawlers, each of which is responsible for a different slice of the web — this approach permits specialization to different types of content, and also facilitates loosely coupled distributed crawling (Section 2.3.4).

As with comprehensive crawling (Section 4.1), the mathematical objective typically associated with scoped crawling is maximization of weighted coverage $\mathrm{WC}(t) = \sum_{p \in \mathcal{C}(t)} w(p)$. In scoped crawling, the role of the weight function $w(p)$ is to reflect the degree to which page p falls within the intended scope. In the simplest case, $w(p) \in \{0, 1\}$, where 0 denotes that p is outside the scope and 1 denotes that p is in-scope. Hence weighted coverage measures the fraction of crawled pages that are in-scope, analogous to the *precision* metric used in information retrieval.

Typically the in-scope pages form a finite set (whereas the full web is often treated as infinite, as mentioned above). Hence it makes sense to measure *recall* in addition to precision. Two recall-oriented evaluation techniques have been proposed: (1) designate a few representative in-scope pages by hand, and measure what fraction of them are discovered by the crawler [92]; (2) measure the overlap among independent crawls initiated from different seeds, to see whether they converge on the same set of pages [34].

Topical crawling (also known as "focused crawling"), in which in-scope pages are ones that are relevant to a particular topic or set of topics, is by far the most extensively studied form of scoped crawling. Work on other forms of scope — e.g., pages with form interfaces [14], and pages within a geographical scope [6, 63] — tends to use similar methods to the ones used for topical crawling. Hence we primarily discuss topical crawling from this point forward.

4.2.1 Topical Crawling

The basic observation exploited by topical crawlers is that relevant pages tend to link to other relevant pages, either directly or via short chains of links. (This feature of the web has been verified empirically

in many studies, including Chakrabarti et al. [34] and Cho et al. [43].) The first crawl ordering technique to exploit this observation was *fish search* [53]. The fish search crawler categorized each crawled page p as either relevant or irrelevant (a binary categorization), and explored the neighborhood of each relevant page up to depth d looking for additional relevant pages.

A second generation of topical crawlers [43, 74, 108] explored the neighborhoods of relevant pages in a non-uniform fashion, opting to traverse the most promising links first. The link traversal order was governed by individual relevance estimates assigned to each linked-to page (a continuous relevance metric is used, rather than a binary one). If a crawled page p links to an uncrawled page q, the relevance estimate for q is computed via analysis of the text surrounding p's link to q (i.e., the anchortext and text near the anchortext[2]), as well as q's URL. In one variant, relevance estimates are smoothed by associating some portion of p's relevance score (and perhaps also the relevance scores of pages linking to p, and so on in a recursive fashion), with q. The motivation for smoothing the relevance scores is to permit discovery of pages that are relevant yet lack indicative anchortext or URLs.

A third-generation approach based on machine learning and link structure analysis was introduced by Chakrabarti et al. [33, 34]. The approach leverages pre-existing topic taxonomies such as the Open Directory and Yahoo!'s web directory, which supply examples of web pages matching each topic. These example pages are used to train a classifier[3] to map newly encountered pages into the topic taxonomy. The user selects a subset of taxonomy nodes (topics) of interest to crawl, and the crawler preferentially follows links from pages that the classifier deems most relevant to the topics of interest. Links from pages that match "parent topics" are also followed (e.g., if the user indicated an interest in bicycling, the crawler follows links from pages about sports in general). In addition, an attempt is made to identify *hub pages* — pages with a collection of links to topical pages — using the HITS link

[2] This aspect was studied in detail by Pant and Srinivasan [107].
[3] Pant and Srinivasan [106] offer a detailed study of classifier choices for topical crawlers.

analysis algorithm [82]. Links from hub pages are followed with higher priority than other links.

The empirical findings of Chakrabarti et al. [34] established topical crawling as a viable and effective paradigm:

- A general web crawler seeded with topical pages quickly becomes mired in irrelevant regions of the web, yielding very poor weighted coverage. In contrast, a topical crawler successfully stays within scope, and explores a steadily growing population of topical pages over time.
- Two topical crawler instances, started from disparate seeds, converge on substantially overlapping sets of pages.

Beyond the basics of topical crawling discussed above, there are two key considerations [92]: Greediness and adaptivity.

4.2.1.1 Greediness

Paths between pairs of relevant pages sometimes pass through one or more irrelevant pages. A topical crawler that is too *greedy* will stop when it reaches an irrelevant page, and never discovers subsequent relevant page(s). On the other extreme, a crawler that ignores relevance considerations altogether degenerates into a non-topical crawler, and achieves very poor weighted coverage, as we have discussed. The question of how greedily to crawl is an instance of the *explore versus exploit* tradeoff observed in many contexts. In this context, the question is: How should the crawler balance exploitation of direct links to (apparently) relevant pages, with exploration of other links that may, eventually, lead to relevant pages?

In the approach of Hersovici et al. [74], a page p inherits some of the relevance of the pages that link to p, and so on in a recursive fashion. This passing along of relevance forces the crawler to traverse irrelevant pages that have relevant ancestors. A *decay factor* parameter controls how rapidly relevance decays as more links are traversed. Eventually, if no new relevant pages are encountered, relevance approaches zero and the crawler ceases exploration on that path.

Later work by Diligenti et al. [54] proposed to classify pages according to their distance from relevant pages. Each uncrawled page p is assigned a distance estimate $d(p) \in [0, \infty)$ that represents the crawler's best guess as to how many links lie between p and the nearest relevant page.[4] Pages are ordered for crawling according to $d(\cdot)$. As long as one or more uncrawled pages having $d(p) = 0$ are available, the crawler downloads those pages; if not, the crawler resorts to downloading $d(p) = 1$ pages, and so on. The threshold used to separate "relevant" pages from "irrelevant" ones controls greediness: If the threshold is strict (i.e., only pages with strong relevance indications are classified as "relevant"), then the crawler will favor long paths to strongly relevant pages over short paths to weakly relevant pages, and vice versa.

A simple meta-heuristic to control the greediness of a crawler was proposed by Menczer et al. [92]: Rather than continuously adjusting the crawl order as new pages and new links are discovered, commit to crawling N pages from the current crawl order before reordering. This heuristic has the attractive property that it can be applied in conjunction with any existing crawl ordering policy. Menczer et al. [92] demonstrated empirically that this heuristic successfully controls the level of greediness, and that there is benefit in not being too greedy, in terms of improved weighted coverage in the long run. The study done by Diligenti et al. [54] also showed improved long-term weighted coverage by not being overly greedy. We are not aware of any attempts to characterize the optimal level of greediness.

4.2.1.2 Adaptivity

In most topical crawling approaches, once the crawler is unleashed, the page ordering strategy is fixed for the duration of the crawl. Some have studied ways for a crawler to *adapt* its strategy over time, in response to observations made while the crawl is in flight. For example, Aggarwal et al. [5] proposed a method to learn on the fly how best to combine

[4] The $d(\cdot)$ function can be trained in the course of the crawl as relevant and irrelevant pages are encountered at various distances from one another. As an optional enhancement to accelerate the training process, ancestors of page p are located using a full-web search engine that services "links-to" queries, and added to the training data.

relevance signals found in the content of pages linking to p, content of p's "siblings" (pages linked to by the same "parent" page), and p's URL, into a single relevance estimate on which to base the crawl order.

Evolutionary algorithms (e.g., genetic algorithms) have been explored as a means to adapt crawl behavior over time [37, 80, 91]. For example, the *InfoSpiders* approach [91] employs many independent crawling *agents*, each with its own relevance classifier that adapts independently over time. Agents reproduce and die according to an evolutionary process: Agents that succeed in locating relevant content multiply and mutate, whereas unsuccessful ones die off. The idea is to improve relevance estimates over time, and also to achieve specialization whereby different agents become adept at locating different pockets of relevant content.

4.3 Efficient Large-Scale Implementation

As discussed in Section 2.3.1, breadth-first crawl ordering can use simple disk-based FIFO queues. Basic scoped crawling methods also afford a fairly simple and efficient implementation: The process of assessing page relevance and assigning priorities to extracted URLs can occur in the main page processing pipeline,[5] and a disk-based priority queue may be used to maintain the crawling frontier (also discussed in Section 2.3.1).

Most of the comprehensive (non-scoped) approaches also operate according to numerical page priorities, but the priority values of enqueued pages are subject to change over time as new information is uncovered (e.g., new links to a page). The more sophisticated scoped crawling approaches also leverage global information (e.g., Chakrabarti et al. [34] used link analysis to identify topical hub pages), and therefore fall into this category as well. With time-varying priorities, one approach is to recompute priority values periodically — either from

[5] In cases where relevance is assessed via a trained classifier, training (and optional periodic retraining) can occur offline and out of the way of the main crawling pipeline.

scratch or incrementally[6] — using distributed disk-based methods similar to those employed in database and map-reduce environments.

Aside from facilitating scalable implementation, delaying the propagation of new signals to the crawl order has a side-effect of introducing an exploration component into an otherwise exploitation-dominated approach, which is of particular significance in the scoped crawling case (see Section 4.2.1.1). On the other hand, some time-critical crawling opportunities (e.g., a new entry of a popular blog) might be compromised. One way to mitigate this problem is to assign initial priority estimates that are not based on global analysis, e.g., using site-level features (e.g., site-level PageRank), URL features (e.g., number of characters or slashes in the URL string), or features of the page from which a URL has been extracted (e.g., that page's PageRank).

The OPIC approach [1] propagates numerical "cash" values to URLs extracted from crawled pages, including URLs already in the frontier. The intention is to maintain a running approximation of PageRank, without the high overhead of the full PageRank computation. If enough memory is (collectively) available on the crawling machines, cash counters can be held in memory and incremented in near-real-time (cash flows across machine boundaries can utilize MPI or another message-passing protocol). Alternatively, following the deferred-updating rubric mentioned above, cash increments can be logged to a side file, with periodic summing of cash values using a disk-based sort-merge algorithm.

[6] Basic computations like counting links can be maintained incrementally using efficient disk-based view maintenance algorithms; more elaborate computations like PageRank tend to be more difficult but offer some opportunities for incremental maintenance [38].

5

Incremental Crawl Ordering

In contrast to a batch crawler, a *continuous* or *incremental* crawler never "starts over." Instead, it continues running forever (conceptually speaking). To maintain freshness of old crawled content, an incremental crawler interleaves revisitation of previously crawled pages with first-time visitation of new pages. The aim is to achieve good freshness and coverage simultaneously.

Coverage is measured according to the same *weighted coverage* metric applied to batch crawlers (Section 4). An analogous *weighted freshness* metric is as follows:

$$\mathrm{WF}(t) = \sum_{p \in \mathcal{C}(t)} w(p) \cdot f(p,t),$$

where $f(p,t)$ is page p's freshness level at time t, measured in one of several possible ways (see below).[1] One is typically interested in the

[1] Pages that have been removed from the web (i.e., their URL is no longer valid) but whose removal has not yet been detected by the crawler, are assigned a special freshness level, e.g., the minimum value on the freshness scale in use.

steady-state average of WF:

$$\overline{\mathrm{WF}} = \lim_{t\to\infty} \frac{1}{t} \int_0^t \mathrm{WF}(t)\, dt.$$

At each step, an incremental crawler faces a choice between two basic actions:

(1) **Download a new page.** Consequences include:
   (a) May improve coverage.
   (b) May supply new links, which can lead to discovery of new pages.[2] (New links also contribute to the crawler's estimates of page importance, relevance, and other aspects like likelihood of being spam; cf. Section 6.)

(2) **Re-download an old page.** Consequences include:
   (a) May improve freshness.
   (b) May supply new links or reveal the removal of links, with similar ramifications as 1(b) above.

In the presence of dynamic pages and finite crawling resources, there is a tradeoff between coverage and freshness. There is no consensus about the best way to balance the two. Some contend that coverage and freshness are like apples and oranges, and balancing the two objectives should be left as a business decision, i.e., do we prefer broad coverage of content that may be somewhat out-of-date, or narrower coverage with fresher content? Others have proposed specific schemes for combining the two objectives into a single framework: The approach taken in WebFountain [56] focuses on the freshness problem, and folds in coverage by treating uncrawled pages as having a freshness value of zero. The OPIC approach [1] focuses on ensuring coverage of important pages, and in the process periodically revisits old important pages.

Aside from the two approaches just mentioned, most published work on crawling focuses either uniquely on coverage or uniquely on

[2] Pages can also be discovered via "out-of-band" channels, e.g., e-mail messages, RSS feeds, user browsing sessions.

freshness. We have already surveyed coverage-oriented techniques in Section 4, in the context of batch crawling. In incremental crawling, coverage can be expanded not only by following links from newly crawled pages, but also by monitoring old pages to detect any new links that might be added over time. This situation was studied by Dasgupta et al. [48], who used a set-cover formulation to identify small sets of old pages that collectively permit discovery of most new pages.

The remainder of this chapter is dedicated to techniques that revisit old pages to acquire a fresh version of their content (not just links to new pages). Section 5.1 focuses on maximizing the average freshness of crawled content, whereas Section 5.2 studies the subtly different problem of capturing the history of content updates. Following these conceptual discussions, Section 5.3 considers practical implementation strategies.

5.1 Maximizing Freshness

Here the goal is to maximize time-averaged weighted freshness, $\overline{\mathrm{WF}}$, as defined above. To simplify the study of this problem, it is standard practice to assume that the set of crawled pages is fixed (i.e., $\mathcal{C}(t)$ is static, so we drop the dependence on t), and that each page $p \in \mathcal{C}$ exhibits a stationary stochastic pattern of content changes over time. Freshness maximization divides into three relatively distinct sub-problems:

- **Model estimation.** Construct a model for the temporal behavior of each page $p \in \mathcal{C}$.
- **Resource allocation.** Given a maximum crawl rate r, assign to each page $p \in \mathcal{C}$ a *revisitation frequency* $r(p)$ such that $\sum_{p \in \mathcal{C}} r(p) = r$.
- **Scheduling.** Produce a crawl order that adheres to the target revisitation frequencies as closely as possible.

With model estimation, the idea is to estimate the temporal behavior of p, given samples of the content of p or pages related to p. Cho and García-Molina [42] focused on how to deal with samples of p that are not evenly spaced in time, which may be the case if the samples have been gathered by the crawler itself in the past, while operating

under a non-uniform scheduling regime. Barbosa et al. [15] considered how to use content-based features of a *single* past sample of p to infer something about its temporal behavior. Cho and Ntoulas [44] and Tan et al. [113] focused on how to infer the behavior of p from the behavior of related pages — pages on the same web site, or pages with similar content, link structure, or other features.

We now turn to scheduling. Coffman et al. [46] pursued randomized scheduling policies, where at each step page p is selected with probability $r(p)/r$, independent of the past schedule. Wolf et al. [115] formulated the crawl scheduling problem in terms of network flow, for which prior algorithms exist. Cho and García-Molina [41] studied a special case of the scheduling problem in which pages have the same target revisitation frequency $r(p)$. All three works concluded that it is best to space apart downloads of each page p uniformly in time, or as close to uniformly as possible.

Resource allocation is generally viewed as the central aspect of freshness maximization. We divide work on resource allocation into two categories, according to the freshness model adopted:

5.1.1 Binary Freshness Model

In the *binary* freshness model, also known as *obsolescence*, $f(p,t) \in \{0,1\}$. Specifically,

$$f(p,t) = \begin{cases} 1 & \text{if the cached copy of } p \text{ is identical}^3 \text{ to the live copy} \\ 0 & \text{otherwise} \end{cases}.$$

Under the binary freshness model, if $f(p,t) = 1$ then p is said to be "fresh," otherwise it is termed "stale." Although simplistic, a great deal of useful intuition has been derived via this model.

The first to study the freshness maximization problem were Coffman et al. [46], who postulated a Poisson model of web page change. Specifically, a page undergoes discrete change events, which cause the copy

[3] It is common to replace the stipulation "identical" with "near-identical," and ignore minor changes like counters and timestamps. Some of the techniques surveyed in Section 6.1 can be used to classify near-identical web page snapshots.

cached by the crawler to become stale. For each page p, the occurrence of change events is governed by a Poisson process with rate parameter $\lambda(p)$, which means that changes occur randomly and independently, with an average rate of $\lambda(p)$ changes per time unit.

A key observation by Coffman et al. [46] was that in the case of uniform page weights (i.e., all $w(p)$ values are equal), the appealing idea of setting revisitation frequencies in proportion to page change rates, i.e., $r(p) \propto \lambda(p)$ (called *proportional* resource allocation), can be suboptimal. Coffman et al. [46] also provided a closed-form optimal solution for the case in which page weights are proportional to change rates (i.e., $w(p) \propto \lambda(p)$), along with a hint for how one might approach the general case.

Cho and García-Molina [41] continued the work of Coffman et al. [46], and derived a famously counterintuitive result: In the uniform weights case, a *uniform* resource allocation policy, in which $r(p) = r/|\mathcal{C}|$ for all p, achieves higher average binary freshness than proportional allocation. The superiority of the uniform policy to the proportional one holds under any distribution of change rates ($\lambda(p)$ values).

The optimal resource allocation policy for binary freshness, also given by Cho and García-Molina [41], exhibits the following intriguing property: Pages with a very fast rate of change (i.e., $\lambda(p)$ very high relative to $r/|\mathcal{C}|$) ought never to be revised by the crawler, i.e., $r(p) = 0$. The reason is as follows: A page p_1 that changes once per second, and is revisited once per second by the crawler, is on average only half synchronized ($f(p_1) = 0.5$). On the other hand, a page p_2 that changes once per day, and is revisited once per hour by the crawler, has much better average freshness ($f(p_2) = 24/25$ under randomized scheduling, according to the formula given by Cho and García-Molina [41]). The crawling resources required to keep one fast-changing page like p_1 weakly synchronized can be put to better use keeping several slow-changing pages like p_2 tightly synchronized, assuming equal page weights. Hence, in terms of average binary freshness, it is best for the crawler to "give up on" fast-changing pages, and put its energy into synchronizing moderate- and slow-changing ones. This resource allocation tactic is analogous to *advanced triage* in the field of medicine [3].

The discussion so far has focused on a Poisson page change model in which the times at which page changes occur are statistically independent. Under such a model, the crawler cannot time its visits to coincide with page change events. The following approaches relax the independence assumption.

Wolf et al. [115] studied incremental crawling under a *quasi-deterministic* page change model, in which page change events are non-uniform in time, and the distribution of likely change times is known a priori. (This work also introduced a search-centric page weighting scheme, under the terminology *embarrassment level*. The embarrassment-based scheme sets $w(p) \propto c(p)$, where $c(p)$ denotes the probability that a user will click on p after issuing a search query, as estimated from historical search engine usage logs. The aim is to revisit frequently clicked pages preferentially, thereby minimizing "embarrassing" incidents in which a search result contains a stale page.)

The WebFountain technique by Edwards et al. [56] does not assume any particular page evolution model a priori. Instead, it categorizes pages adaptively into one of n *change rate buckets* based on recently observed change rates (this procedure replaces explicit model estimation). Bucket membership yields a working estimate of a page's present change rate, which is in turn used to perform resource allocation and scheduling.

5.1.2 Continuous Freshness Models

In a real crawling scenario, some pages may be "fresher" than others. While there is no consensus about the best way to measure freshness, several non-binary freshness models have been proposed.

Cho and García-Molina [41] introduced a temporal freshness metric, in which $f(p,t) \propto -age(p,t)$, where

$$age(p,t) = \begin{cases} 0 & \text{if the cached copy of } p \text{ is identical to the live copy} \\ a & \text{otherwise} \end{cases},$$

where a denotes the amount of time the copies have differed. The rationale for this metric is that the longer a cached page remains unsynchronized with the live copy, the more their content tends to drift apart.

The optimal resource allocation policy under this age-based freshness metric, assuming a Poisson model of page change, is given by Cho and García-Molina [41]. Unlike in the binary freshness case, there is no "advanced triage" effect — the revisitation frequency $r(p)$ increases monotonically with the page change rate $\lambda(p)$. Since age increases without bound, the crawler cannot afford to "give up on" any page.

Olston and Pandey [99] introduced an approach in which, rather than relying on time as a proxy for degree of change, the idea is to measure changes in page content directly. A content-based freshness framework is proposed, which constitutes a generalization of binary freshness: A page is divided into a set of *content fragments*[4] $f_1, f_2, \ldots, f_n$, each with a corresponding weight $w(f_i)$ that captures the fragment's importance and/or relevance. Freshness is measured as the (weighted) fraction of fragments in common between the cached and live page snapshots, using the well-known Jaccard set similarity measure.

Under a content-based freshness model, the goal is to minimize the amount of incorrect content in the crawler's cache, averaged over time. To succeed in this goal, Olston and Pandey [99] argued that in addition to characterizing the frequency with which pages change, it is necessary to characterize the *longevity* of newly updated page content. Long-lived content (e.g., today's blog entry, which will remain in the blog indefinitely) is more valuable to crawl than ephemeral content (e.g., today's "quote of the day," which will be overwritten tomorrow), because it stays fresh in the cache for a longer period of time. The optimal resource allocation policy for content-based freshness derived by Olston and Pandey [99] differentiates between long-lived and ephemeral content, in addition to differentiating between frequently and infrequently changing pages.

In separate work, Pandey and Olston [103] proposed a search-centric method of assigning weights to individual content changes, based on the degree to which a change is expected to impact search ranking. The rationale is that even if a page undergoes periodic changes, and

[4] Fragments can be determined in a number of ways, e.g., using logical or visual document structure, or using *shingles* [28]).

the new content supplied by the changes is long-lived, if the search engine's treatment of the page is unaffected by these changes, there is no need for the crawler to revisit it.

5.2 Capturing Updates

For some crawling applications, maximizing average freshness of cached pages is not the right objective. Instead, the aim is to capture as many individual content updates as possible. Applications that need to capture updates include historical archival and temporal data mining, e.g., time-series analysis of stock prices.

The two scenarios (maximizing freshness versus capturing updates) lead to very different page revisitation policies. As we saw in Section 5.1, in the freshness maximization scenario, when resources are scarce the crawler should ignore pages that frequently replace their content, and concentrate on maintaining synchronization with pages that supply more persistent content. In contrast, in the update capture scenario, pages that frequently replace their content offer the highest density of events to be captured, and also the highest urgency to capture them before they are lost.

Early update capture systems, e.g., CONQUER [86], focused on user-facing query languages, algorithms to difference page snapshots and extract relevant tidbits (e.g., updated stock price), and query processing techniques. Later work considered the problem of when to revisit each page to check for updates.

In work by Pandey et al. [105], the objective was to maximize the (weighted) number of updates captured. A suitable resource allocation algorithm was given, which was shown empirically to outperform both uniform and proportional resource allocation.

In subsequent work, Pandey et al. [102] added a *timeliness* dimension, to represent the sensitivity of the application to delays in capturing new information from the web. (For example, historical archival has a low timeliness requirement compared with real-time stock market analysis.) The key insight was that revisitation of pages whose updates do not replace old content can be postponed to a degree permitted by the application's timeliness requirement, yielding a higher total

amount of information captured. A timeliness-sensitive resource allocation algorithm was given, along with a formal performance guarantee.

5.3 Efficient Large-Scale Implementation

One key aspect of page revisitation policy is change model estimation. For certain pages (e.g., newly discovered pages on important sites), model estimation is time-critical; but for the vast majority of pages (e.g., unimportant, well-understood, or infrequently crawled ones) it can be performed lazily, in periodic offline batches. Since most change models deal with each page in isolation, this process is trivially parallelizable. For the time-critical pages, the relevant data tend to be small and amenable to caching: Most techniques require a compact signature (e.g., a few shingle hashes) for two to five of the most recent page snapshots. The search-centric approach of Pandey and Olston [103] requires more detailed information from pages, and incorporates model estimation into the process of re-building the search index.

After model estimation, the other two aspects of revisitation policy are resource allocation and scheduling. While there is quite a bit of work on using global optimization algorithms to assign revisitation schedules to individual pages, it is not immediately clear how to incorporate such algorithms into a large-scale crawler. One scalable approach is to group pages into *buckets* according to desired revisitation frequency (as determined by resource allocation, see below), and cycle through each bucket such that the time to complete one cycle matches the bucket's revisitation frequency target. This approach ensures roughly equal spacing of revisitations of a given page, which has been found to be a good rule of thumb for scheduling, as mentioned in Section 5.1. It also yields an obvious disk-based implementation: Each bucket is a FIFO queue, where URLs removed from the head are automatically appended to the tail; crawler machines pull from the queues according to a weighted randomized policy constructed to achieve the target revisitation rates in expectation.

In the bucket-oriented approach, pages that are identical (or near-identical) from the point of view of the adopted model are placed into the same bucket. For example, in the basic freshness maximization

formulation under the binary freshness model (Section 5.1.1), pages can be bucketized by change frequency. Assignment of a revisitation frequency target to each bucket is done by the resource allocation procedure. The WebFountain crawler by Edwards et al. [56] uses change frequency buckets and computes bucket revisitation frequency targets periodically using a nonlinear problem (NLP) solver. Since the NLP is formulated at the bucket granularity (not over individual pages), the computational complexity is kept in check. Most other resource allocation strategies can also be cast into the bucket-oriented NLP framework. To take a simple example, an appealing variant of the *uniform* resource allocation is to direct more resources toward important pages; in this case, one can bucketize pages by importance, and apply a simple NLP over bucket sizes and importance weights to determine the revisitation rate of each bucket.

6

Avoiding Problematic and Undesirable Content

This section discusses detection and avoidance of content that is redundant, wasteful or misleading.

6.1 Redundant Content

As discussed in Section 3.2.1, there is a prevalence of duplicate and near-duplicate content on the web. Shingling [28] is a standard way to identify near-duplicate pages, but shingling is performed on web page content, and thus requires these pages to have been crawled. As such, it does not help to reduce the load on the crawler; however, it can be used to limit and diversify the set of search results presented to a user.

Some duplication stems from the fact that many web sites allow multiple URLs to refer to the same content, or content that is identical modulo ever-changing elements such as rotating banner ads, evolving comments by readers, and timestamps. Schonfeld et al. proposed the "duplicate URL with similar text" (DUST) algorithm [12] to detect this form of aliasing, and to infer rules for normalizing URLs into a canonical form. Dasgupta et al. [49] generalized DUST by introducing a learning algorithm that can generate rules containing regular

expressions, experimentally tripling the number of duplicate URLs that can be detected. Agarwal et al. attempted to bound the computational complexity of learning rules using sampling [4]. Rules inferred using these algorithms can be used by a web crawler to normalize URLs after extracting them from downloaded pages and before passing them through the duplicate URL eliminator (Section 2.3.2) and into the frontier.

Another source of duplication is *mirroring* [18, 19, 52]: Providing all or parts of the same web site on different hosts. Mirrored web sites in turn can be divided into two groups: Sites that are mirrored by the same organization (for example by having one web server serving multiple domains with the same content, or having multiple web servers provide synchronized content), and content that is mirrored by multiple organizations (for example, schools providing Unix `man` pages on the web, or web sites republishing Wikipedia content, often somewhat reformatted). Detecting mirrored content differs from detecting DUST in two ways: On the one hand, with mirroring the duplication occurs across multiple sites, so mirror detection algorithms have to consider the entire corpus. On the other hand, entire trees of URLs are mirrored, so detection algorithms can use URL trees (suitably compacted e.g., through hashing) as a feature to detect mirror candidates, and then compare the content of candidate subtrees (for example via shingling).

6.2 Crawler Traps

Content duplication inflates the web corpus without adding much information. Another phenomenon that inflates the corpus without adding utility is *crawler traps*: Web sites that populate a large, possibly infinite URL space on that site with mechanically generated content. Some crawler traps are non-malicious, for example web-based calendaring tools that generate a page for every month of every year, with a hyperlink from each month to the next (and previous) month, thereby forming an unbounded chain of dynamically generated pages. Other crawler traps are malicious, often set up by "spammers" to inject large amounts of their content into a search engine, in the hope of having their content show up high in search result pages or providing

many hyperlinks to their "landing page," thus biasing link-based ranking algorithms such as PageRank. There are many known heuristics for identifying and avoiding spam pages or sites, see Section 6.3. Not much research has been published on algorithms or heuristics for detecting crawler traps directly. The IRLbot crawler [84] utilizes a heuristic called "Budget Enforcement with Anti-Spam Tactics" (BEAST), which assigns a budget to each web site and prioritizes URLs from each web site based on the site's remaining budget combined with the domain's reputation.

6.3 Web Spam

Web spam may be defined as "web pages that are crafted for the sole purpose of increasing the ranking of these or some affiliated pages, without improving the utility to the viewer" [97]. Web spam is motivated by the monetary value of achieving a prominent position in search-engine result pages. There is a multi-billion dollar industry devoted to *search engine optimization* (SEO), most of it being legitimate but some of it misleading. Web spam can be broadly classified into three categories [69]: *Keyword stuffing*, populating pages with highly searched or highly monetizable terms; *link spam*, creating cliques of tightly interlinked web pages with the goal of biasing link-based ranking algorithms such as PageRank [101]; and *cloaking*, serving substantially different content to web crawlers than to human visitors (to get search referrals for queries on a topic not covered by the page).

Over the past few years, many heuristics have been proposed to identify spam web pages and sites, see for example the series of AIRweb workshops [76]. The problem of identifying web spam can be framed as a classification problem, and there are many well-known classification approaches (e.g., decision trees, Bayesian classifiers, support vector machines). The main challenge is to identify features that are predictive of web spam and can thus be used as inputs to the classifier. Many such features have been proposed, including hyperlink features [16, 17, 50, 116], term and phrase frequency [97], DNS lookup statistics [59], and HTML markup structure [114]. Combined, these features tend to be quite effective, although web spam detection is a

constant arms race, with both spammers and search engines evolving their techniques in response to each other's actions.

Spam detection heuristics are used during the ranking phase of search, but they can also be used during the corpus selection phase (when deciding which pages to index) and crawling phase (when deciding what crawl priority to assign to web pages). Naturally, it is easier to avoid crawling spam content in a continuous or iterative crawling setting, where historical information about domains, sites, and individual pages is available.[1]

6.4 Cloaked Content

Cloaking refers to the practice of serving different content to web crawlers than to human viewers of a site [73]. Not all cloaking is malicious: For example, many web sites with interactive content rely heavily on JavaScript, but most web crawlers do not execute JavaScript, so it is reasonable for such a site to deliver alternative, script-free versions of its pages to a search engine's crawler to enable the engine to index and expose the content.

Web sites distinguish mechanical crawlers from human visitors either based on their `User-Agent` field (an HTTP header that is used to distinguish different web browsers, and by convention is used by crawlers to identify themselves), or by the crawler's IP address (the SEO community maintains lists of the `User-Agent` fields and IP addresses of major crawlers). One way for search engines to detect that a web server employs cloaking is by supplying a different `User-Agent` field [117]. Another approach is to probe the server from IP addresses not known to the SEO community (for example by enlisting the search engine's user base).

A variant of cloaking is called *redirection spam*. A web server utilizing redirection spam serves the same content both to crawlers and to human-facing browser software (and hence, the aforementioned detection techniques will not detect it); however, the content will cause a

[1] There is of course the possibility of spammers acquiring a site with a good history and converting it to spam, but historical reputation-based approaches at least "raise the bar" for spamming.

browser to immediately load a new page presenting different content. Redirection spam is facilitated either through the HTML `META REFRESH` tag (whose presence is easy to detect), or via JavaScript, which most browsers execute but most crawlers do not. Chellapilla and Maykov [36] conducted a study of pages employing JavaScript redirection spam, and found that about half of these pages used JavaScript's `eval` statement to obfuscate the URLs to which they redirect, or even parts of the script itself. This practice makes static detection of the redirection target (or even the fact that redirection is occurring) very difficult. Chellapilla and Maykov argued for the use of lightweight JavaScript parsers and execution engines in the crawling/indexing pipeline to evaluate scripts (in a time-bounded fashion, since scripts may not terminate) to determine whether redirection occurs.

7

Deep Web Crawling

Some content is accessible only by filling in HTML forms, and cannot be reached via conventional crawlers that just follow hyperlinks.[1] Crawlers that automatically fill in forms to reach the content behind them are called *hidden web* or *deep web* crawlers.

The deep web crawling problem is closely related to the problem known as *federated search* or *distributed information retrieval* [30], in which a mediator forwards user queries to multiple searchable collections, and combines the results before presenting them to the user. The crawling approach can be thought of as an *eager* alternative, in which content is collected in advance and organized in a unified index prior to retrieval. Also, deep web crawling considers structured query interfaces in addition to unstructured "search boxes," as we shall see.

7.1 Types of Deep Web Sites

Figure 7.1 presents a simple taxonomy of deep web sites. Content is either unstructured (e.g., free-form text) or structured (e.g., data

[1] Not all content behind form interfaces is unreachable via hyperlinks — some content is reachable in both ways [35].

	Unstructured Content	Structured Content
Unstructured query interface	News archive (simple search)	Product review site
Structured query interface	News archive (advanced search)	Online bookstore

Fig. 7.1 Deep web taxonomy.

records with typed fields). Similarly, the form interface used to query the content is either unstructured (i.e., a single query box that accepts a free-form query string) or structured (i.e., multiple query boxes that pertain to different aspects of the content).[2]

A news archive contains content that is primarily unstructured (of course, some structure is present, e.g., title, date, author). In conjunction with a simple textual search interface, a news archive constitutes an example of an unstructured-content/unstructured-query deep web site. A more advanced query interface might permit structured restrictions on attributes that are extractable from the unstructured content, such as language, geographical references, and media type, yielding an unstructured-content/structured-query instance.

A product review site has relatively structured content (product names, numerical reviews, reviewer reputation, and prices, in addition to free-form textual comments), but for ease of use typically offers an unstructured search interface. Lastly, an online bookstore offers structured content (title, author, genre, publisher, price) coupled with a structured query interface (typically a subset of the content attributes, e.g., title, author and genre).

For simplicity most work focuses on either the upper-left quadrant (which we henceforth call the *unstructured case*), or the lower-right quadrant (*structured case*).

[2] The unstructured versus structured dichotomy is really a continuum, but for simplicity we present it as a binary property.

7.2 Problem Overview

Deep web crawling has three steps:

(1) **Locate deep web content sources.** A human or crawler must identify web sites containing form interfaces that lead to deep web content. Barbosa and Freire [14] discussed the design of a scoped crawler for this purpose.
(2) **Select relevant sources.** For a scoped deep web crawling task (e.g., crawling medical articles), one must select a relevant subset of the available content sources. In the unstructured case this problem is known as *database* or *resource selection* [32, 66]. The first step in resource selection is to model the content available at a particular deep web site, e.g., using *query-based sampling* [31].
(3) **Extract underlying content.** Finally, a crawler must extract the content lying behind the form interfaces of the selected content sources.

For major search engines, Step 1 is almost trivial, since they already possess a comprehensive crawl of the surface web, which is likely to include a plethora of deep web query pages. Steps 2 and 3 pose significant challenges. Step 2 (source selection) has been studied extensively in the distributed information retrieval context [30], and little has been done that specifically pertains to crawling. Step 3 (content extraction) is the core problem in deep web crawling; the rest of this chapter covers the (little) work that has been done on this topic.

7.3 Content Extraction

The main approach to extracting content from a deep web site proceeds in four steps (the first two steps apply only to the structured case):

(1) Select a subset of form elements to populate,[3] or perhaps multiple such subsets. This is largely an open problem, where the goals are to: (a) avoid form elements that merely

[3] The remaining elements can remain blank, or be populated with a wildcard expression when applicable.

affect the presentation of results (e.g., sorting by price versus popularity); and (b) avoid including correlated elements, which artificially increase the dimensionality of the search space [88].

(2) If possible, decipher the role of each of the targeted form elements (e.g., book author versus publication date), or at least understand their domains (proper nouns versus dates). Raghavan and García-Molina [109] and several subsequent papers studied this difficult problem.

(3) Create an initial database of valid data values (e.g., "Ernest Hemingway" and 1940 in the structured case; English words in the unstructured case). Some sources of this information include [109]: (a) a human administrator; (b) non-deep-web online content, e.g., a dictionary (for unstructured keywords) or someone's list of favorite authors; (c) drop-down menus for populating form elements (e.g., a drop-down list of publishers).

(4) Use the database to issue queries to the deep web site (e.g., publisher = "Scribner"), parse the result and extract new data values to insert into the database (e.g., author = "Ernest Hemingway"), and repeat.

We elaborate on Step 4, which has been studied under (variations of) the following model of deep web content and queries [98, 117]:

A deep web site contains one or more *content items*, which are either unstructured documents or structured data records. A content item contains individual *data values*, which are text terms in the unstructured case, or data record elements like author names and dates in the structured case. Data values and content values are related via a bipartite graph, depicted in Figures 7.2 (unstructured case) and 7.3 (structured case).

A query consists of a single data value[4] V submitted to the form interface, which retrieves the set of content items directly connected

[4] It is assumed that any data value can form the basis of a query, even though this is not always the case in practice (e.g., a bookstore may not permit querying by publisher). Also, multi-value queries are not considered.

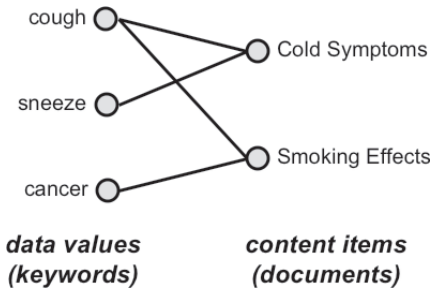


Fig. 7.2 Deep web content model (unstructured content).

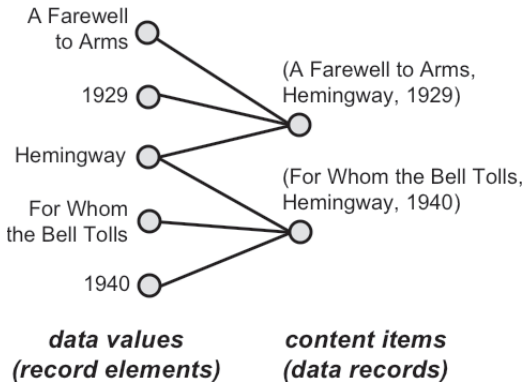


Fig. 7.3 Deep web content model (structured content).

to V via edges in the graph, called V's *result set*. Each query incurs some cost to the crawler, typically dominated by the overhead of downloading and processing each member of the result set, and hence modeled as being linearly proportional to result cardinality.

Under this model, the deep web crawling problem can be cast as a weighted set-cover problem: Select a minimum-cost subset of data values that cover all content items. Unfortunately, unlike in the usual set-cover scenario, in our case the graph is only partially known at the outset, and must be uncovered progressively during the course of the crawl. Hence, adaptive graph traversal strategies are required.

A simple greedy traversal strategy was proposed by Barbosa and Freire [13] for the unstructured case: At each step the crawler issues as a query the highest-frequency keyword that has not yet been issued,

where keyword frequency is estimated by counting occurrences in documents retrieved so far. In the bipartite graph formulation, this strategy is equivalent to selecting the data value vertex of highest degree, according to the set of edges uncovered so far.

A similar strategy was proposed by Wu et al. [117] for the structured case, along with a refinement in which the crawler bypasses data values that are highly correlated with ones that have already been selected, in the sense that they connect to highly overlapping sets of content items.

Ntoulas et al. [98] proposed statistical models for estimating the number of previously unseen content items that a particular data value is likely to cover, focusing on the unstructured case.

Google's deep web crawler [88] uses techniques similar to the ones described above, but adapted to extract a small amount of content from a large number (millions) of sites, rather than aiming for extensive coverage of a handful of sites.

UNIT – V
Opinion Mining and Web Usage Mining

11 Opinion Mining

In Chap. 9, we studied structured data extraction from Web pages. Such data are usually records retrieved from underlying databases and displayed in Web pages following some fixed templates. The Web also contains a huge amount of information in unstructured texts. Analyzing these texts is of great importance and perhaps even more important than extracting structured data because of the sheer volume of valuable information of almost any imaginable types contained in them. In this chapter, we only focus on mining of opinions on the Web. The task is not only technically challenging because of the need for natural language processing, but also very useful in practice. For example, businesses always want to find public or consumer opinions on their products and services. Potential customers also want to know the opinions of existing users before they use a service or purchase a product. Moreover, opinion mining can also provide valuable information for placing advertisements in Web pages. If in a page people express positive opinions or sentiments on a product, it may be a good idea to place an ad of the product. However, if people express negative opinions about the product, it is probably not wise to place an ad of the product. A better idea may be to place an ad of a competitor's product.

The Web has dramatically changed the way that people express their opinions. They can now post reviews of products at merchant sites and express their views on almost anything in Internet forums, discussion groups, blogs, etc., which are commonly called the **user generated content** or **user generated media**. This online word-of-mouth behavior represents new and measurable sources of information with many practical applications. Techniques are now being developed to exploit these sources to help businesses and individuals gain such information effectively and easily.

The first part of this chapter focuses on three mining tasks of **evaluative texts** (which are documents expressing opinions):

1. **Sentiment classification:** This task treats opinion mining as a text classification problem. It classifies an evaluative text as being positive or negative. For example, given a product review, the system determines whether the review expresses a positive or a negative sentiment of the reviewer. The classification is usually at the document-level. No details are discovered about what people liked or didn't like.

1. **Featured-based opinion mining and summarization:** This task goes to the sentence level to discover details, i.e., what aspects of an object that people liked or disliked. The object could be a product, a service, a topic, an individual, an organization, etc. For example, in a product review, this task identifies product features that have been commented on by reviewers and determines whether the comments are positive or negative. In the sentence, "the battery life of this camera is too short," the comment is on the "battery life" and the opinion is negative. A structured summary will also be produced from the mining results.
2. **Comparative sentence and relation mining:** Comparison is another type of evaluation, which directly compares one object against one or more other similar objects. For example, the following sentence compares two cameras: "the battery life of camera A is much shorter than that of camera B." We want to identify such sentences and extract comparative relations expressed in them.

The second part of the chapter discusses **opinion search** and **opinion spam**. Since our focus is on opinions on the Web, opinion search is naturally relevant, and so is opinion spam. An opinion search system enables users to search for opinions on any object. Opinion spam refers to dishonest or malicious opinions aimed at promoting one's own products and services, and/or at damaging the reputations of those of one's competitors. Detecting opinion spam is a challenging problem because for opinions expressed on the Web, the true identities of their authors are often unknown.

The research in opinion mining only began recently. Hence, this chapter should be treated as statements of problems and descriptions of current research rather than a report of mature techniques for solving the problems. We expect major progresses to be made in the coming years.

11.1 Sentiment Classification

Given a set of evaluative texts D, a sentiment classifier classifies each document $d \in D$ into one of the two classes, **positive** and **negative**. Positive means that d expresses a positive opinion. Negative means that d expresses a negative opinion. For example, given some reviews of a movie, the system classifies them into positive reviews and negative reviews.

The main application of sentiment classification is to give a quick determination of the prevailing opinion on an object. The task is similar but also different from classic topic-based text classification, which classifies documents into predefined topic classes, e.g., politics, science, sports, etc. In topic-based classification, topic related words are important. However,

in sentiment classification, topic-related words are unimportant. Instead, sentiment words that indicate positive or negative opinions are important, e.g., great, excellent, amazing, horrible, bad, worst, etc.

The existing research in this area is mainly at the **document-level**, i.e., to classify each whole document as positive or negative (in some cases, the **neutral** class is used as well). One can also extend such classification to the **sentence-level**, i.e., to classify each sentence as expressing a positive, negative or neutral opinion. We discuss several approaches below.

11.1.1 Classification Based on Sentiment Phrases

This method performs classification based on positive and negative sentiment words and phrases contained in each evaluative text. The algorithm described here is based on the work of Turney [521], which is designed to classify customer reviews.

This algorithm makes use of a natural language processing technique called **part-of-speech (POS) tagging**. The part-of-speech of a word is a linguistic category that is defined by its syntactic or morphological behavior. Common POS categories in English grammar are: noun, verb, adjective, adverb, pronoun, preposition, conjunction and interjection. Then, there are many categories which arise from different forms of these categories. For example, a verb can be a verb in its base form, in its past tense, etc. In this book, we use the standard **Penn Treebank POS Tags** as shown in Table 11.1. POS tagging is the task of labeling (or tagging) each word in a sentence with its appropriate part of speech. For details on part-of-speech tagging, please refer to the report by Santorini [472]. The Penn Treebank site is at http://www.cis.upenn.edu/~treebank/home.html.

The algorithm given in [521] consists of three steps:

Step 1: It extracts phrases containing adjectives or adverbs. The reason for doing this is that research has shown that adjectives and adverbs are good indicators of subjectivity and opinions. However, although an isolated adjective may indicate subjectivity, there may be an insufficient context to determine its **semantic** (or **opinion**) **orientation**. For example, the adjective "unpredictable" may have a negative orientation in an automotive review, in such a phrase as "unpredictable steering", but it could have a positive orientation in a movie review, in a phrase such as "unpredictable plot". Therefore, the algorithm extracts two consecutive words, where one member of the pair is an adjective/adverb and the other is a context word.

Two consecutive words are extracted if their POS tags conform to any of the patterns in Table 11.2. For example, the pattern in line 2 means

. **Table 11.1.** Penn Treebank part-of-speech tags (excluding punctuation)

Tag	Description	Tag	Description
CC	Coordinating conjunction	PRP$	Possessive pronoun
CD	Cardinal number	RB	Adverb
DT	Determiner	RBR	Adverb, comparative
EX	Existential *there*	RBS	Adverb, superlative
FW	Foreign word	RP	Particle
IN	Preposition or subordinating conjunction	SYM	Symbol
JJ	Adjective	TO	*to*
JJR	Adjective, comparative	UH	Interjection
JJS	Adjective, superlative	VB	Verb, base form
LS	List item marker	VBD	Verb, past tense
MD	Modal	VBG	Verb, gerund or present participle
NN	Noun, singular or mass	VBN	Verb, past participle
NNS	Noun, plural	VBP	Verb, non-3rd person singular present
NNP	Proper noun, singular	VBZ	Verb, 3rd person singular present
NNPS	Proper noun, plural	WDT	Wh-determiner
PDT	Predeterminer	WP	Wh-pronoun
POS	Possessive ending	WP$	Possessive wh-pronoun
PRP	Personal pronoun	WRB	Wh-adverb

that two consecutive words are extracted if the first word is an adverb and the second word is an adjective, but the third word (which is not extracted) cannot be a noun. NNP and NNPS are avoided so that the names of the objects in the review cannot influence the classification.

Table 11.2. Patterns of tags for extracting two-word phrases from reviews

	First word	Second word	Third word (Not Extracted)
1.	JJ	NN or NNS	anything
2.	RB, RBR, or RBS	JJ	not NN nor NNS
3.	JJ	JJ	not NN nor NNS
4.	NN or NNS	JJ	not NN nor NNS
5.	RB, RBR, or RBS	VB, VBD, VBN, or VBG	anything

Example 1: In the sentence "this camera produces beautiful pictures", "beautiful pictures" will be extracted as it satisfies the first pattern. ■

Step 2: It estimates the semantic orientation of the extracted phrases using the **pointwise mutual information** measure given in Equation 1:

$$PMI(term_1, term_2) = \log_2\left(\frac{\Pr(term_1 \wedge term_2)}{\Pr(term_1)\Pr(term_2)}\right). \quad (1)$$

Here, $\Pr(term_1 \wedge term_2)$ is the co-occurrence probability of $term_1$ and $term_2$, and $\Pr(term_1)\Pr(term_2)$ gives the probability that the two terms co-occur if they are statistically independent. The ratio between $\Pr(term_1 \wedge term_2)$ and $\Pr(term_1)\Pr(term_2)$ is thus a measure of the degree of statistical dependence between them. The log of this ratio is the amount of information that we acquire about the presence of one of the words when we observe the other.

The semantic/opinion orientation (SO) of a phrase is computed based on its association with the positive reference word "excellent" and its association with the negative reference word "poor":

$$SO(phrase) = PMI(phrase, \text{“}excellent\text{”}) - PMI(phrase, \text{“}poor\text{”}). \quad (2)$$

The probabilities are calculated by issuing queries to a search engine and collecting the number of **hits**. For each search query, a search engine usually gives the number of relevant documents to the query, which is the number of hits. Thus, by searching the two terms together and separately, we can estimate the probabilities in Equation 1. Turney [521] used the AltaVista search engine because it has a NEAR operator, which constrains the search to documents that contain the words within ten words of one another, in either order. Let *hits*(*query*) be the number of hits returned. Equation 2 can be rewritten as:

$$SO(phrase) = \log_2\left(\frac{hits(\text{phrase } NEAR \text{ "excellent"})hits(\text{"poor"})}{hits(\text{phrase } NEAR \text{ "poor"})hits(\text{"excellent"})}\right). \quad (3)$$

To avoid division by zero, 0.01 is added to the hits.

Step 3: Given a review, the algorithm computes the average *SO* of all phrases in the review, and classifies the review as recommended if the average *SO* is positive, not recommended otherwise.

Final classification accuracies on reviews from various domains range from 84% for automobile reviews to 66% for movie reviews.

11.1.2 Classification Using Text Classification Methods

The simplest approach to sentiment classification is to treat the problem as a topic-based text classification problem. Then, any text classification algorithm can be employed, e.g., naïve Bayesian, SVM, *k*NN, etc.

The approach was experimented by Pang et al. [428] using movie reviews of two classes, positive and negative. It was shown that using a unigram (a bag of individual words) in classification performed well using either naïve Bayesian or SVM. Test results using 700 positive reviews and 700 negative reviews showed that these two classification algorithms achieved 81% and 82.9% accuracy respectively with 3-fold cross validation. However, neutral reviews were not used in this work, which made the problem easier. No stemming or stopword removal was applied.

11.1.3 Classification Using a Score Function

A custom score function for review sentiment classification was given by Dave et al. [122]. The algorithm consists of two steps:

Step 1: It scores each term in the training set using the following equation,

$$score(t_i) = \frac{\Pr(t_i \mid C) - \Pr(t_i \mid C')}{\Pr(t_i \mid C) + \Pr(t_i \mid C')}, \tag{4}$$

where t_i is a term and C is a class and C' is its complement, i.e., not C, and $\Pr(t_i|C)$ is the conditional probability of term t_i in class C. It is computed by taking the number of times that a term t_i occurs in class C reviews and dividing it by the total number of terms in the reviews of class C. A term's score is thus a measure of bias towards either class ranging from –1 and 1.

Step 2: To classify a new document $d_i = t_1 \ldots t_n$, the algorithm sums up the scores of all terms and uses the sign of the total to determine the class. That is, it uses the following equation for classification,

$$class(d_i) = \begin{cases} C & eval(d_i) > 0 \\ C' & \text{otherwise,} \end{cases} \tag{5}$$

where

$$eval(d_i) = \sum_j score(t_j) \cdot \tag{6}$$

Experiments were conducted based on a large number of reviews (more than 13000) of seven types of products. The results showed that the bigrams (consecutive two words) and trigrams (consecutive three words) as terms gave (similar) best accuracies (84.6%–88.3%), on two different review data sets. No stemming or stopword removal was applied.

In this paper, the authors experimented with many alternative classification techniques, e.g., naïve Bayesian, SVM, and several algorithms based

on other score functions. They also tried some word substitution strategies to improve generalization, e.g.,

- replace product names with a token ("_productname");
- replace rare words with a token ("_unique");
- replace category-specific words with a token ("_producttypeword");
- replace numeric tokens with NUMBER.

Some linguistic modifications using WordNet, stemming, negation, and collocation were tested too. However, they were not helpful, and usually degraded the classification accuracy.

In summary, the main advantage of document level sentiment classification is that it provides a prevailing opinion on an object, topic or event. The main shortcomings of the document-level classification are:

- It does not give details on what people liked or disliked. In a typical evaluative text such as a review, the author usually writes specific aspects of an object that he/she likes or dislikes. The ability to extract such details is useful in practice.
- It is not easily applicable to non-reviews, e.g., forum and blog postings, because although their main focus may not be evaluation or reviewing of a product, they may still contain a few opinion sentences. In such cases, we need to identify and extract opinion sentences.

There are several variations of the algorithms discussed in this section (see Bibliographic Notes). Apart from these learning based methods, there are also manual approaches for specific applications. For example, Tong [517] reported a system that generates sentiment timelines. The system tracks online discussions about movies and displays a plot of the number of positive sentiment and negative sentiment messages (*Y*-axis) over time (*X*-axis). Messages are classified by matching specific phrases that indicate sentiments of the author towards the movie (e.g., "great acting", "wonderful visuals", "uneven editing", "highly recommend it", and "it sucks"). The phrases were manually compiled and tagged as indicating positive or negative sentiments to form a lexicon. The lexicon is specific to the domain (e.g., movies) and must be built anew for each new domain.

11.2 Feature-Based Opinion Mining and Summarization

Although studying evaluative texts at the document level is useful in many cases, it leaves much to be desired. A positive evaluative text on a particular object does not mean that the author has positive opinions on every aspect of the object. Likewise, a negative evaluative text does not mean that

the author dislikes everything about the object. For example, in a product review, the reviewer usually writes both positive and negative aspects of the product, although the general sentiment on the product could be positive or negative. To obtain such detailed aspects, we need to go to the sentence level. Two tasks are apparent [245]:

1. Identifying and extracting features of the product that the reviewers have expressed their opinions on, called **product features**. For instance, in the sentence "the picture quality of this camera is amazing," the product feature is "picture quality".
2. Determining whether the opinions on the features are positive, negative or neutral. In the above sentence, the opinion on the feature "picture quality" is positive.

11.2.1 Problem Definition

In general, the opinions can be expressed on anything, e.g., a product, an individual, an organization, an event, a topic, etc. We use the general term "**object**" to denote the entity that has been commented on. The object has a set of **components** (or parts) and also a set of attributes (or properties). Thus the object can be hierarchically decomposed according to the **part-of** relationship, i.e., each component may also have its sub-components and so on. For example, a product (e.g., a car, a digital camera) can have different components, an event can have sub-events, a topic can have subtopics, etc. Formally, we have the following definition:

Definition (object): An **object** O is an entity which can be a product, person, event, organization, or topic. It is associated with a pair, O: (T, A), where T is a hierarchy or taxonomy of **components** (or **parts**), **sub-components**, and so on, and A is a set of **attributes** of O. Each component has its own set of sub-components and attributes.

Example 2: A particular brand of digital camera is an object. It has a set of components, e.g., lens, battery, view-finder, etc., and also a set of attributes, e.g., picture quality, size, weight, etc. The battery component also has its set of attributes, e.g., battery life, battery size, battery weight, etc. ■

Essentially, an object is represented as a tree. The root is the object itself. Each non-root node is a component or sub-component of the object. Each link represents a *part-of* relationship. Each node is also associated with a set of attributes. An opinion can be expressed on any node and any attribute of the node.

Example 3: Following Example 2, one can express an opinion on the camera (the root node), e.g., "I do not like this camera", or on one of its attributes, e.g., "the picture quality of this camera is poor". Likewise, one can also express an opinion on one of the camera's components, e.g., "the battery of this camera is bad", or an opinion on the attribute of the component, "the battery life of this camera is too short." ■

To simplify our discussion, we use the word "**features**" to represent both components and attributes, which allows us to omit the hierarchy. Using features for products is also quite common in practice. For an ordinary user, it is probably too complex to use a hierarchical representation of product features and opinions. We note that in this framework the object itself is also treated as a feature.

Let the evaluative text (e.g., a product review) be r. In the most general case, r consists of a sequence of sentences $r = \langle s_1, s_2, \ldots, s_m \rangle$.

Definition (explicit and implicit feature): If a feature f appears in evaluative text r, it is called an **explicit feature** in r. If f does not appear in r but is implied, it is called an **implicit feature** in r.

Example 4: "battery life" in the following sentence is an explicit feature:

"The battery life of this camera is too short".

"Size" is an implicit feature in the following sentence as it does not appear in the sentence but it is implied:

"This camera is too large".

Definition (opinion passage on a feature): The **opinion passage** on feature f of an object evaluated in r is a group of consecutive sentences in r that expresses a positive or negative opinion on f.

It is common that a sequence of sentences (at least one) in an evaluative text together expresses an opinion on an object or a feature of the object. Also, it is possible that a single sentence expresses opinions on more than one feature:

"The picture quality is good, but the battery life is short".

Most current research focuses on sentences, i.e., each passage consisting of a single sentence. Thus, in our subsequent discussion, we use **sentences** and **passages** interchangeably.

Definition (explicit and implicit opinion): An **explicit opinion** on feature f is a subjective sentence that directly expresses a positive or negative opinion. An **implicit opinion** on feature f is an objective sentence that implies a positive or negative opinion.

Example 5: The following sentence expresses an explicit positive opinion:

"The picture quality of this camera is amazing."

The following sentence expresses an implicit negative opinion:

"The earphone broke in two days."

Although this sentence states an objective fact (assume it is true), it implicitly expresses a negative opinion on the earphone. ■

Definition (opinion holder): The **holder** of a particular opinion is a person or an organization that holds the opinion.

In the case of product reviews, forum postings and blogs, opinion holders are usually the authors of the postings, although occasionally some authors cite or repeat the opinions of others. Opinion holders are more important in news articles because they often explicitly state the person or organization that holds a particular view. For example, the opinion holder in the sentence "John expressed his disagreement on the treaty" is "John".

We now put things together to define a **model** of an object and a set of opinions on the object. An object is represented with a finite set of features, $F = \{f_1, f_2, \ldots, f_n\}$. Each feature f_i in F can be expressed with a finite set of words or phrases W_i, which are **synonyms**. That is, we have a set of corresponding synonym sets $W = \{W_1, W_2, \ldots, W_n\}$ for the n features. Since each feature f_i in F has a name (denoted by f_i), then $f_i \in W_i$. Each author or **opinion holder** j comments on a subset of the features $S_j \subseteq F$. For each feature $f_k \in S_j$ that opinion holder j comments on, he/she chooses a word or phrase from W_k to describe the feature, and then expresses a positive or negative opinion on it.

This simple model covers most but not all cases. For example, it does not cover the situation described in the following sentence: "the viewfinder and the lens of this camera are too close", which expresses a negative opinion on the distance of the two components. We will follow this simplified model in the rest of this chapter.

This model introduces three main practical problems. Given a collection of evaluative texts D as input, we have:

Problem 1: Both F and W are unknown. Then, in opinion mining, we need to perform three tasks:

Task 1: Identifying and extracting object features that have been commented on in each evaluative text $d \in D$.

Task 2: Determining whether the opinions on the features are positive, negative or neutral.

Task 3: Grouping synonyms of features, as different people may use different words or phrases to express the same feature.

Problem 2: *F* is known but *W* is unknown. This is similar to Problem 1, but slightly easier. All the three tasks for Problem 1 still need to be performed, but Task 3 becomes the problem of matching discovered features with the set of given features *F*.

Problem 3: *W* is known (then *F* is also known). We only need to perform Task 2 above, namely, determining whether the opinions on the known features are positive, negative or neutral after all the sentences that contain them are extracted (which is simple).

Clearly, the first problem is the most difficult to solve. Problem 2 is slightly easier. Problem 3 is the easiest, but still realistic.

Example 6: A cellular phone company wants to mine customer reviews on a few models of its phones. It is quite realistic to produce the feature set *F* that the company is interested in and also the set of synonyms of each feature (although the set might not be complete). Then there is no need to perform Tasks 1 and 3 (which are very challenging problems). ■

Output: The final output for each evaluative text *d* is a set of pairs. Each pair is denoted by (*f*, *SO*), where *f* is a feature and *SO* is the semantic or opinion orientation (positive or negative) expressed in *d* on feature *f*. We ignore neutral opinions in the output as they are not usually useful.

Note that this model does not consider the strength of each opinion, i.e., whether the opinion is strongly negative (or positive) or weakly negative (or positive), but it can be added easily (see [548] for a related work).

There are many ways to use the results. A simple way is to produce a **feature-based summary** of opinions on the object. We use an example to illustrate what that means.

Example 7: Assume we summarize the reviews of a particular digital camera, *digital_camera_*1. The summary looks like that in Fig. 11.1.

In Fig. 11.1, "picture quality" and (camera) "size" are the product features. There are 123 reviews that express positive opinions about the picture quality, and only 6 that express negative opinions. The <individual review sentences> link points to the specific sentences and/or the whole reviews that give positive or negative comments about the feature.

With such a summary, the user can easily see how the existing customers feel about the digital camera. If he/she is very interested in a particular feature, he/she can drill down by following the <individual review sentences> link to see why existing customers like it and/or what they are not

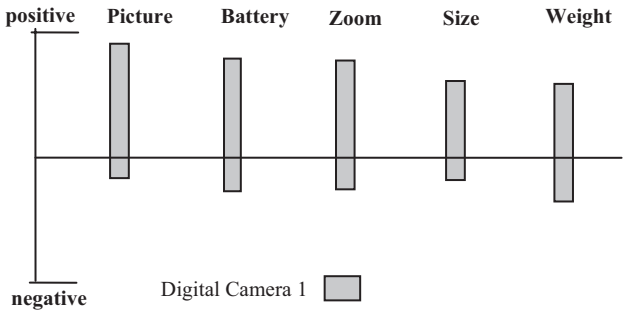
*Digital_camera_*1:

Feature: **picture quality**
Positive: 123 <individual review sentences>
Negative: 6 <individual review sentences>
Feature: **size**
Positive: 82 <individual review sentences>
Negative: 10 <individual review sentences>
…

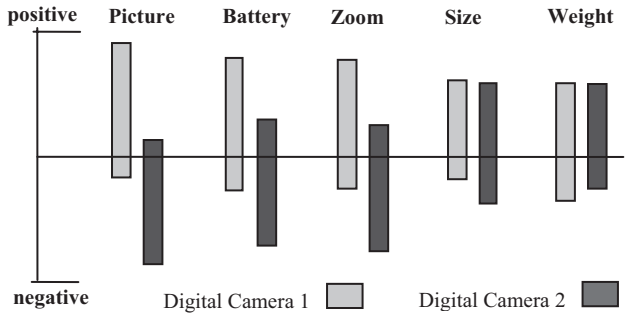
Fig. 11.1. An example of a feature-based summary of opinions

satisfied with. The summary can also be visualized using a bar chart. Figure 11.2(A) shows the feature-based opinion summary of a digital camera.

In the figure, the bars above the *X*-axis in the middle show the percentages of positive opinions on various features (given at the top), and the bars below the *X*-axis show the percentages of negative opinions on the same features.



(A) Feature-based summary of opinions on a digital camera



(B) Opinion comparison of two digital cameras

Fig. 11.2. Visualization of feature-based opinion summary and comparison

Comparing the opinion summaries of a few competing products is even more interesting. Figure 11.2(B) shows a visual comparison of consumer opinions on two competing digital cameras. We can clearly see how consumers view different features of each product. Digital camera 1 is clearly superior to digital camera 2. Specifically, most customers have negative opinions about the picture quality, battery and zoom of digital camera 2. However, on the same three features, customers are mostly positive about digital camera 1. Regarding size and weight, customers have similar opinions on both cameras. Thus, the visualization enables users to clearly see how the cameras compare with each other along each feature dimension. ■

Below, we discuss four other important issues.

Separation of Opinions on the Object itself and its Features: It is often useful to separate opinions on the object itself and opinions on the features of the object. The opinion on the object itself reflects the general sentiment of the author (or the opinion holder) on the object, which is what sentiment classification tries to discover at the document level.

Granularity of Analysis: Let us go back to the general representation of an object with a component tree and each component with a set of attributes. We can study opinions at any level.

At level 1: We identify opinions on the object itself and its attributes.
At level 2: We identify opinions on the major components of the object, and also opinions on the attributes of the components.

At other levels, similar tasks can be performed. However, in practice, analysis at level 1 and level 2 are usually sufficient.

Example 8: Given the following review of a camera (the object),

> "I like this camera. Its picture quality is amazing. However, the battery life is a little short",

in the first sentence, the positive opinion is at level 1, i.e., a positive opinion on the camera itself. The positive opinion on the picture quality in the second sentence is also at level 1 as "picture quality" is an attribute of the camera. The third sentence expresses a negative opinion on an attribute of the battery (at level 2), which is a component of the camera. ■

Opinion Holder Identification: In some applications, it is useful to identify and extract opinion holders, i.e., persons or organizations that have expressed certain opinions. As we mentioned earlier, opinion holders are more useful for news articles and other types of formal documents, in which the person or organization that expressed an opinion is usually stated in the text explicitly. However, such holders need to be identified by

the system. In the case of the user-generated content on the Web, the opinion holders are often the authors of discussion posts, bloggers, or reviewers, whose login ids are often known although their true identities in the real-world may be unknown. We will not discuss opinion holders in the chapter further due to our focus on the user-generated content on the Web. Interested readers, please refer to [276].

Opinioned Object Identification and Pronoun Resolution: In product reviews, the reviewed objects are usually known. However, this is not the case for opinions expressed in blogs and discussions. For example, in the following post "I have a Canon S50 camera purchased from Amazon. It takes great photos.", two interesting questions can be asked: (1) what object does the post praise? and (2) what "it" means in the second sentence? Clearly, we know that the post praises "Canon S50 camera", which is the problem of **opinioned object identification**, and we also know that "it" here means "Canon S50 camera" too, which is the problem of **pronoun resolution**. However, to automatically discover answers to the questions is a very challenging problem. So far, little work has been done.

11.2.2 Object Feature Extraction

Current research on feature extraction is mainly carried out from online **product reviews**. We focus on such reviews in this subsection as well.

It is a common practice for online merchants (e.g., amazon.com) to ask their customers to review the products that they have purchased. There are also dedicated review sites like epinions.com. There are three main review formats on the Web. Different review formats may need different techniques to perform the feature extraction task.

Format 1 – Pros, cons and the detailed review: The reviewer is asked to describe pros and cons separately and also write a detailed review. An example of such a review is given in Fig. 11.3.

Format 2 – Pros and cons: The reviewer is asked to describe pros and cons separately, but there is not a separate detailed review as in format 1. That is, the details are in pros and cons. An example of such a review is given in Fig. 11.4.

Format 3 – Free format: The reviewer can write freely, i.e., no separation of pros and cons. An example of such a review is given in Fig. 11.5.

For formats 1 and 2, opinion (or semantic) orientations (positive or negative) of the features are known because pros and cons are separated. Only product features need to be identified. For format 3, we need to identify both product features and opinion orientations.

My SLR is on the shelf

by camerafun4. Aug 09 '04

Pros: Great photos, easy to use, very small
Cons: Battery usage; included memory is stingy.

I had never used a digital camera prior to purchasing this Canon A70. I have always used a SLR … **Read the full review**

Fig. 11.3. An example of a review of format 1.

"It is a great digital still camera for this century"

September 1 2004.

Pros:
It's small in size, and the rotatable lens is great. It's very easy to use, and has fast response from the shutter. The LCD …

Cons:
It almost has no cons. It could be better if the LCD is bigger and it's going to be best if the model is designed to a smaller size.

Fig. 11.4. An example of a review of format 2.

GREAT Camera., Jun 3, 2004

Reviewer: **jprice174** from Atlanta, Ga.

I did a lot of research last year before I bought this camera... It kinda hurt to leave behind my beloved nikon 35mm SLR, but I was going to Italy, and I needed something smaller, and digital.

The pictures coming out of this camera are amazing. The 'auto' feature takes great pictures most of the time. And with digital, you're not wasting film if the picture doesn't come out. …

Fig. 11.5. An example of a review of format 3.

In both formats 2 and 3, reviewers typically use full sentences. However, for format 1, pros and cons tend to be very brief. For example, in Fig. 11.3, under pros, we have "Great photos, easy to use, take videos", which are elaborated in the detailed review.

Let us deal with pros and cons of format 1 first. The detailed reviews of format 1 are not used as they are elaborations of pros and cons. Analyzing short sentence segments in pros and cons produces more accurate results. Detailed reviews of format 1 are the same as reviews of format 3.

11.2.3 Feature Extraction from Pros and Cons of Format 1

We now describe a supervised pattern learning approach to extract product features from pros and cons in the reviews of format 1. These patterns are

generated from **label sequential rules** (LSR) (see Sect. 2.9.2). This method is based on the algorithm in [247, 347].

A product feature can be expressed with a noun, adjective, verb or adverb. The labels and their POS tags used in mining LSRs are: {$feature, NN}, {$feature, JJ}, {$feature, VB} and {$feature, RB}, where $feature denotes a feature to be extracted, and NN stands for noun, VB for verb, JJ for adjective, and RB for adverb. They represent both explicit features and implicit feature indicators. We call a word that indicates an implicit feature an **implicit feature indicator**. For example, in the sentence "this camera is too heavy", "heavy" is an adjective and is an implicit feature indicator for feature "weight".

The feature extraction technique is based on the following observation:

- Each sentence segment in pros and cons contains only one feature. Sentence segments are separated by commas, periods, semi-colons, hyphens, '&''s, 'and''s, 'but''s, etc.

Example 9: Pros in Fig. 11.3 can be separated into three segments:

great photos	⟨photo⟩
easy to use	⟨use⟩
very small	⟨small⟩ ⇒ ⟨size⟩.

Cons in Fig. 11.3 can be separated into two segments:

battery usage	⟨battery⟩
included memory is stingy	⟨memory⟩

■

We can see that each segment describes a product feature, which is listed within ⟨ ⟩. Notice that ⟨small⟩ is an implicit feature indicator and ⟨size⟩ is the implicit feature.

One point to note is that an explicit feature may not be a noun or noun phrase. Verbs can be explicit features as well, e.g., "use" in "easy to use". In general, 60–70% of the features are explicit noun features. A small proportion of explicit features are verbs. 20–30% of the features are implicit features represented by their indicators. Let us now describe the method.

Given a set of reviews, this method consists of the following two steps:

1. **Training data preparation for LSR mining:** It consists of 4 sub-steps:
   - Part-Of-Speech (POS) tagging and sequence generation: For each sentence segment, the algorithm first performs POS tagging, and then produces a sequence. For example, the sentence segment,

     "Included memory is stingy".

     is turned into a sequence with POS tags:

     ⟨{included, VB}{memory, NN}{is, VB}{stingy, JJ}⟩.

- Replace the actual feature words with {$feature, <tag>}, where $feature represents a feature. This replacement is necessary because different products have different features, and the replacement ensures that we can find general language patterns to extract any product feature. After replacement, the above example becomes:

  ⟨{included, VB}{$feature, NN}{is, VB}{stingy, JJ}⟩.

- Use an *n*-gram to produce shorter segments from long ones: For example, the above sequence will generate two trigram sequences:

  ⟨{included, VB}{$feature, NN}{is, VB}⟩
  ⟨{$feature, NN}{is, VB}{stingy, JJ}⟩.

  Trigrams are usually sufficient. The reason for using *n*-grams rather than full sentences is because most product features can be found based on local information and POS tags. Using long sentences tend to generate a large number of spurious rules.
- Perform word stemming: This reduces a word to its stem (see Sect. 6.5.2).

After the four-step pre-processing, the resulting sentence (trigram) segments are saved in a sequence database for label sequential rule mining. In this file, each line contains one processed sequence.

2. **Label sequential rule mining:** A LSR mining system is applied to find all rules that involve a feature, i.e., $feature. An example rule is:

   ⟨{easy, JJ }{to}{*, VB}⟩ → ⟨{easy, JJ}{to}{$feature, VB}⟩.

   Note that both POS tags and words may appear in a rule. A suitable minimum confidence and minimum support should be used, which can be chosen based on experiments. The right-hand-side of the rule is also called a **language pattern**.

3. **Feature extraction:** The resulting language patterns are used to match each sentence segment in a new review to extract product features. That is, the word in the sentence segment that matches $feature in a language pattern is extracted. Three situations are considered in extraction:

   - If a sentence segment satisfies multiple rules, we search for a matching rule in the following order: {$feature, NN}, {$feature, JJ}, {$feature, VB} and {$feature, RB}. The reason for this ordering is that noun features appear more frequently than other types. For rules of the same tag, the rule with the highest confidence is used since higher confidence indicates higher predictive accuracy.

- For sentence segments that no rules apply, nouns or noun phrases produced by a POS tagger are extracted as features if such nouns or noun phrases exist.
- For a sentence segment with only a single word (e.g., "heavy" and "big"), this pattern-based method does not apply. In such cases, the single words are treated as (implicit or explicit) features.

After extraction, we need to deal with several other important problems:

Mapping to Implicit Features: There are many types of implicit feature indicators. Adjectives are perhaps the most common type. Many adjectives modify or describe some specific attributes or properties of objects. For example, the adjective "heavy" usually describes the attribute "weight" of an object. "Beautiful" is normally used to describe (positively) the attribute "look" or "appearance" of an object. By no means, however, does this say that these adjectives only describe such attributes. Their exact meaning can be domain dependent. For example, "heavy" in the sentence "the traffic is heavy" does not describe the "weight" of the traffic.

One way to map indicator words to implicit features is to manually compile a list of such mappings during training data annotation, which can then be used in the same domain in the future. However, it is not clear whether this is an effective approach as little research has been done.

Grouping Synonyms: It is common that people use different words or phrases to describe the same feature. For example, "photo" and "picture" refer to the same feature in digital camera reviews. Identifying and grouping synonyms is essential for practical applications. Although WordNet [175] and other thesaurus dictionaries help to some extent, they are far from sufficient due to the fact that many synonyms are domain dependent. For example, "picture" and "movie" are synonyms in movie reviews. However, they are not synonyms in digital camera reviews as "picture" is more related to "photo" while "movie" refers to "video".

Liu et al. [347] made an attempt using synonyms in WordNet. Carenini et al. [80] proposes a more sophisticated method based on several similarity metrics that require the taxonomy of features to be given. The system merges each discovered feature to a feature node in the taxonomy. The similarity metrics are defined based on string similarity, synonyms and other distances measured using WordNet. Experimental results based on digital camera and DVD reviews show promising results. Clearly, many ideas and techniques described in Chap. 10 for information integration are applicable here.

Granularity of Features: In the sentence segment "great photos", it is easy to decide that "photo" is the feature. However, in "battery usage", we

can use either "battery usage" or "battery" as the feature. As we discussed in Sect. 11.2.1, each object has a component/part tree and each component node has a set of attributes. In a practical application, we need to determine the right level of analysis. If it is too general, it may not be useful. If it is too specific, it may result in a large number of features and also make the extraction very difficult and inaccurate.

11.2.4 Feature Extraction from Reviews of Formats 2 and 3

Pros and cons of format 1 mainly consist of short phrases and incomplete sentences. The reviews of formats 2 and 3 usually consist of complete sentences. To extract features from such reviews, the above algorithm can also be applied. However, some preliminary experiments show that it is not effective because complete sentences are more complex and contain a large amount of irrelevant information. Below, we describe an unsupervised method for finding explicit features that are nouns and noun phrases. This method requires a large number of reviews, and consists of two steps:

1. Finding frequent nouns and noun phrases. Nouns and noun phrases (or groups) are identified by using a POS tagger. We then count their frequency and only keep the frequent ones. A frequency threshold can be decided experimentally. The reason for using this approach is that most product features are nouns, and those nouns that are frequently talked about are usually genuine and important features. Irrelevant contents (see Fig. 11.5) in reviews are often diverse, i.e., they are quite different in different reviews. When people comment on product features, the vocabulary that they use converges. Those nouns that are infrequent are likely to be non-features or less important features.
2. Finding infrequent features by making use of sentiment words. **Sentiment words** (also called **opinion words**) are usually adjectives and adverbs that express positive or negative opinions, e.g., great, amazing, bad, and expensive. The idea is as follows: The same opinion word can be used to describe different objects. Opinion words that modify frequent features can be used to find infrequent features. For example, "picture" is found to be a frequent feature, and we have the sentence,

   "The pictures are absolutely amazing."

   We also know that "amazing" is a positive opinion word (to be discussed in Sect. 11.2.5). Then "software" may also be extracted as a feature from the following sentence,

   "The software is amazing."

because the two sentences follow the same language pattern and "software" in the sentence is also a noun.

This two-step approach is based on the work of Hu and Liu [245]. At the time this book was written, the shopping site Froogle of the search engine Google implemented a method similar to step 1 of the algorithm. However, it does not restrict frequent terms to be nouns or noun phrases.

The precision of step 1 of the above algorithm was improved by Popescu and Etzioni in [447]. Their algorithm tries to remove those noun phrases that may not be product features. It evaluates each noun phrase by computing a PMI score between the phrase and **meronymy discriminators** associated with the product class, e.g., a scanner class. The meronymy discriminators for the scanner class are, "of scanner", "scanner has", "scanner comes with", etc., which are used to find components or parts of scanners by searching on the Web (see [166] also). The PMI measure is a simplified version of the measure given in Sect. 11.1.1:

$$PMI(f,d) = \frac{hits(f \wedge d)}{hits(f)hits(d)}, \tag{7}$$

where f is a candidate feature identified in step 1 and d is a discriminator. Web search is used to find the number of hits. The idea of this approach is clear. If the PMI value of a candidate feature is too low, it may not be a component of the product because f and d do not co-occur frequently. The algorithm also distinguishes components/parts from attributes/properties using WordNet's *is-a* hierarchy (which enumerates different kinds of properties) and morphological cues (e.g., "-iness", "-ity" suffixes).

Finally, we note that many information extraction techniques are also applicable, e.g., conditional random fields (CRF) [298], hidden Markov models (HMM) [185], and many others. However, no comparative evaluation of these methods on this problem has been reported so far.

11.2.5 Opinion Orientation Classification

For reviews of format 3, we need to classify each sentence that contains a product feature as positive, negative or neutral. This classification may also be needed for reviews of format 2 because although pros and cons are separated in format 2, some sentences containing features are neutral.

We describe two main techniques below. The accuracy is usually reasonable (greater than 80%) if the sentences are either positive or negative, but if neutral sentences are included, the accuracy often drops significantly. Sentences containing negations also pose difficulties.

1. Using **sentiment words** and **phrases**: As explained above, sentiment words and phrases are words and phrases that express positive or negative sentiments (or opinions). They are mostly adjectives and adverbs, but can be verbs and nouns too. Researchers have compiled sets of such words and phrases for adjectives, adverbs, verbs, and nouns respectively. Each set is usually obtained through a bootstrapping process:
   - Manually find a set of seed positive and negative words. Separate seed sets are prepared for adjectives, adverbs, verbs and nouns.
   - Grow each of the seed set by iteratively searching for their synonyms and antonyms in WordNet until convergence, i.e., until no new words can be added to the set. Antonyms of positive (or negative) words will be added to the negative (or positive) set.
   - Manually inspect the results to remove those incorrect words. Although this step is time consuming, it is only an one-time effort.

   Apart from a set of opinion words, there are also **idioms**, which can be classified as positive, negative and neutral as well. Many language patterns also indicate positive or negative sentiments. They can be manually compiled and/or discovered using pattern discovery methods.

   Using the final lists of positive and negative words, phrases, idioms and patterns, each sentence that contains product features can be classified as follows: Sentiment words and phrases in the sentence are identified first. A positive word or phrase is assigned a score of +1 and a negative word or phrase is assigned a score of −1. All the scores are then summed up. If the final total is positive, then the sentence is positive, otherwise it is negative. If a negation word is near a sentiment word, the opinion is reversed. A sentence that contains a "but" clause (subsentence that starts with "but", "however", etc.) indicates a sentiment change for the feature in the clause.

   This method is based on the techniques given by Hu and Liu [245], and Kim and Hovy [276]. In [447], Popescu and Etzioni proposed a more complex method, which makes use of syntactical dependencies produced by a parser. Yu and Hatzivassiloglou [584] presented a method similar to that in Sect. 11.1.1 but used a large number of seeds. Recall that Turney [521] used only two seeds (see Sect. 11.1.1), "excellent" for positive and "poor" for negative. The sentence orientation is determined by a threshold of the average score of the words in the sentence. It is not clear which method performs better because there is little comparative evaluation.

   Note that the opinion orientations of many words are **domain** and/or **sentence context dependent**. Such situations are usually hard to deal with. It can be easy in some cases. For example, "small" can be positive

or negative. However, if there is a "too" before it, it normally indicates a negative sentiment, e.g., "this camera is too small for me".

2. The methods described in Sect. 11.1 for sentiment classification are applicable here. Using supervised learning, we need to prepare a set of manually labeled positive, negative and neutral sentences as the training data. If sentiment words and phrases, idioms and patterns are used also as attributes, the classification results can be further improved. Sentences containing negations and clauses starting with "but", "however", etc., need special handling since one part of the sentence may be positive and another part may be negative, e.g., "The pictures of this camera are great, but the camera itself is a bit too heavy."

In summary, although many classification techniques have been proposed, little comparative study of these techniques has been reported. A promising approach is to combine these techniques to produce a better classifier.

11.3 Comparative Sentence and Relation Mining

Directly expressing positive or negative opinions on an object is only one form of evaluation. Comparing the object with some other similar objects is another. Comparison is perhaps a more convincing way of evaluation. For example, when a person says that something is good or bad, one often asks "compared to what?" Thus, one of the most important ways of evaluating an object is to directly compare it with some other similar objects.

Comparisons are related to but also different from typical opinions. They have different semantic meanings and different syntactic forms. Comparisons may be subjective or objective. For example, a typical opinion sentence is "the picture quality of camera x is great." A subjective comparison is "the picture quality of camera x is better than that of camera y." An objective comparison is "camera x is 20 grams heavier than camera y", which may be a statement of a fact and may not have an implied opinion on which camera is better.

In this section, we study the problem of identifying comparative sentences and comparative relations (defined shortly) in text documents, e.g., consumer reviews, forum discussions and news articles. This problem is also challenging because although we can see that the above example sentences all contain some indicators, i.e., "better" and "longer", many sentences that contain such words are not comparisons, e.g., "in the context of speed, faster means better". Similarly, many sentences that do not contain such indicators are comparative sentences, e.g., "cellphone X has bluetooth, but cellphone Y does not," and "Intel is way ahead of AMD."

11.3.1 Problem Definition

A **comparative sentence** is a sentence that expresses a relation based on similarities or differences of more than one object. The comparison in a comparative sentence is usually expressed using the **comparative** or the **superlative** form of an adjective or adverb. The comparative is used to state that one thing has more (bigger, smaller) "value" than the other. The superlative is used to say that one thing has the most (the biggest, the smallest) "value". The structure of a comparative consists normally of the stem of an adjective or adverb, plus the suffix *-er*, or the modifier "more" or "less" before the adjective or adverb. For example, in "John is taller than James", "taller" is the comparative form of the adjective "tall". The structure of a superlative consists normally of the stem of an adjective or adverb, plus the suffix *-est*, or the modifier "most" or "least" before the adjective or adverb. In "John is the tallest in the class", "tallest" is the superlative form of the adjective "tall".

A comparison can be between two or more objects, groups of objects, one object and the rest of the objects. It can also be between an object and its previous or future versions.

Types of Important Comparisons: We can classify comparisons into four main types. The first three types are **gradable comparisons** and the last one is the **non-gradable comparison**. The gradable types are defined based on the relationships of *greater* or *less than*, *equal to*, and *greater* or *less than all others*.

1. *Non-equal gradable comparisons*: Relations of the type *greater* or *less than* that express an ordering of some objects with regard to some of their features, e.g., "the Intel chip is faster than that of AMD". This type also includes user preferences, e.g., "I prefer Intel to AMD".
2. *Equative comparisons*: Relations of the type *equal to* that state two objects are equal with respect to some of their features, e.g., "the picture quality of camera A is as good as that of camera B"
3. *Superlative comparisons*: Relations of the type *greater* or *less than all others* that rank one object over *all* others, e.g., "the Intel chip is the fastest".
4. *Non-gradable comparisons*: Sentences that compare features of two or more objects, but do not grade them. There are three main types:
   - Object A is similar to or different from object B with regard to some features, e.g., "Coke tastes differently from Pepsi".
   - Object A has feature f_1, and object B has feature f_2 (f_1 and f_2 are usually substitutable), e.g., "desktop PCs use external speakers but laptops use internal speakers".

- Object *A* has feature *f*, but object *B* does not have, e.g., "cell phone A has an earphone, but cell phone B does not have".

Gradable comparisons can be classified further into two types: **adjectival comparisons** and **adverbial comparisons**. Adjectival comparisons involve comparisons of degrees associated with adjectives, e.g., in "John is taller than Mary," and "John is the tallest in the class"). Adverbial comparisons are similar but usually occur after verb phrases, e.g., "John runs faster than James," and "John runs the fastest in the class".

Given an evaluative text *d*, **comparison mining** consists of two tasks:

1. Identify comparative passages or sentences from *d*, and classify the identified comparative sentences into different types or classes.
2. Extract comparative relations from the identified sentences. This involves the extraction of entities and their features that are being compared, and the comparative keywords. Relations in gradable adjectival comparisons can be expressed with

   (<*relationWord*>, <*features*>, <*entityS1*>, <*entityS2*>, <*type*>)

   where:

   relationWord: The comparative keyword used to express a comparative relation in a sentence.

   features: a set of features being compared.

   *entityS*1 and *entityS*2: Sets of entities being compared. Entities in entityS1 appear to the left of the relation word and entities in entityS2 appear to the right of the relation word.

   type: *non-equal gradable*, *equative* or *superlative*.

   Example 10: Consider the comparative sentence "Canon's optics is better than those of Sony and Nikon." The extracted relation is:

   (better, {optics}, {Canon}, {Sony, Nikon}, *non-equal gradable*). ■

We can also design relation representations for adverbial comparisons and non-gradable comparisons. In this section, however, we only focus on adjectival gradable comparisons as there is little study on relation extraction of the other types. For simplicity, we will use *comparative sentences* and *gradable comparative sentences* interchangeably from now on.

Finally, we note that there are other types of comparatives in linguistics that are used to express different types of relations. However, they are relatively rare in evaluative texts and/or less useful in practice. For example, a meta-linguistic comparative compares the extent to which a single object has one property to a greater or lesser extent than another property, e.g., "Ronaldo is angrier than upset" (see [150, 274, 313, 393]).

11.3.2 Identification of Gradable Comparative Sentences

This is a classification problem. A machine learning algorithm is applicable to solve this problem. The main issue is what attributes to use.

An interesting phenomenon about comparative sentences is that such a sentence usually has a comparative keyword. It is shown in [256] that using a set of 83 keywords, 98% of the comparative sentences (recall = 98%) can be identified with a precision of 32% using the authors' data set. Let us see what the keywords are:

1. **Comparative adjectives** (with the POS tag of JJR) and **comparative adverbs** (with the POS tag of RBR), e.g., *more*, *less*, *better*, *longer* and words ending with *-er*.
2. **Superlative adjectives** (with the POS tag of JJS) and **superlative adverbs** (with the POS tag of RBS), e.g., *most*, *least*, *best*, *tallest* and words ending with *-est*.
3. Words like *same*, *similar*, *differ* and those used with equative *as*, e.g., *same as*, *as well as*, etc.
4. Others, such as *favor*, *beat*, *win, exceed*, *outperform, prefer*, *ahead, than*, *superior*, *inferior*, *number one, up against*, etc.

Note that those words with POS tags of JJR, RBR, JJS and RBS are not used as keywords themselves. Instead, their POS tags, JJR, RBR, JJS and RBS, are treated as four keywords only. There are four exceptions: *more*, *less*, *most*, and *least* are treated as individual keywords because their usages are diverse, and using them as individual keywords enables the system to catch their individual usage patterns for classification.

Since keywords alone are able to achieve a very high recall, the following learning approach is used in [255] to improve the precision:

- Use the set of keywords to filter out those sentences that are unlikely to be comparative sentences (do not contain any keywords). The remaining set of sentences R forms the candidate set of comparative sentences.
- Work on R to improve the precision, i.e., to classify the sentences in R into *comparative* and *non-comparative sentences*, and then into different types of comparative sentences.

It is also observed in [255] that comparative sentences have strong patterns involving comparative keywords, which is not surprising. These patterns can be used as attributes in learning. To discover these patterns, **class sequential rule (CSR) mining** (see Sect. 2.9.3) was used. Each training example used for mining CSRs is a pair (s_i, y_i), where s_i is a sequence and y_i is a class, $y_i \in \{comparative, non\text{-}comparative\}$.

Training Data Preparation: The sequences in the training data are generated from sentences. Since we want to find patterns surrounding specific keywords, we use keywords as **pivots** to produce sequences.

Let the set of pivots be P. We generate the training sequence database as follows:

1. For each sentence, only words within the radius of r of the keyword pivot $p_i \in P$ are used to form a sequence. In [256], r is set to 3. Each pivot in a sentence forms a separate sequence.
2. Each word is then replaced with its POS tag. The actual words are not used because the contents of sentences may be very different, but their underlying language patterns can be the same. Using POS tags allow us to capture content independent patterns. There is an exception. For each keyword (except those represented by JJR, RBR, JJS and RBS), the actual word and its POS tag are combined together to form a single item. The reason for this is that some keywords have multiple POS tags depending on their use. Their specific usages can be important in determining whether a sentence is a comparative sentence or not. For example, the keyword "more" can be a comparative adjective (more/JJR) or a comparative adverb (more/RBR) in a sentence.
3. A class is attached to each sequence according to whether the sentence is a comparative or non-comparative sentence.

Example 11: Consider the comparative sentence "this/DT camera/NN has/VBZ significantly/RB more/JJR noise/NN at/IN iso/NN 100/CD than/IN the/DT nikon/NN 4500/CD." It has the keywords "more" and "than". The sequence involving "more" put in the training set is:

⟨{NN}{VBZ}{RB}{more/JJR}{NN}{IN}{NN}⟩, comparative)

CSR Generation: Using the training data, CSRs can be generated. Recall that a CSR is an implication of the form, $X \rightarrow y$, where X is a sequence and y is a class. Due to the fact that some keywords appear very frequently in the data and some appear rarely, multiple minimum supports are used in mining. The minimum item support for each keyword is computed with $freq * \tau$, where τ is set to 0.1 ($freq$ is the actual frequency of its occurrence). See Sect. 2.7.2 or Sect. 2.8.2 in Chap. 2 for details on mining with multiple minimum supports.

In addition to the automatically generated rules, some manually compiled rules are also used in [255, 256], which are more complex and difficult to generate by current rule mining techniques.

Classifier Building: There are many ways to build classifiers using the discovered CSRs, we describe two methods:

1. Treat all CSRs as a classifier. A CSR simply expresses the conditional probability that a sentence is a comparison if it contains the sequence pattern X. These rules can thus be used for classification. That is, for each test sentence, the algorithm finds all the rules satisfied by the sentence, and then chooses the rule with the highest confidence to classify the sentence. This is basically the "use the strongest rule" method discussed in Sect. 3.5.1 in Chap. 3.
1. Use CSRs as attributes to create a data set and then learn a naïve Bayesian (NB) classifier (or any other types of classifiers) (see Sect. 3.5.2 in Chap. 3). The data set uses the following attribute set:

$$\begin{aligned} \textit{Attribute Set} = \; & \{X \mid X \text{ is the sequential pattern in CSR } X \rightarrow y\} \cup \\ & \{Z \mid Z \text{ is the pattern in a manual rule } Z \rightarrow y\}. \end{aligned}$$

The class is not used but only the sequence pattern X (or Z) of each rule. The idea is that these patterns are predictive of the classes. A rule's predictability is indicated by its confidence. The minimum confidence of 60% is used in [255].

Each sentence forms an example in the training data. If the sentence has a particular pattern in the attribute set, the corresponding attribute value is 1, and is 0 otherwise. Using the resulting data, it is straightforward to perform NB learning. Other learning methods can be used as well, but according to [255], NB seems to perform better.

Classify Comparative Sentences into Three Types: This step classifies comparative sentences obtained from the last step into one of the three types or classes, *non-equal gradable*, *equative*, and *superlative*. For this task, the keywords alone are already sufficient. That is, we use the set of keywords as the attribute set for machine learning. If the sentence has a particular keyword in the attribute set, the corresponding attribute value is 1, and otherwise it is 0. SVM gives the best results in this case.

11.3.3 Extraction of Comparative Relations

We now discuss how to extract relation entries/items. Label sequential rules are again used for this task. The algorithm presented below is based on the work in [256], which makes the following assumptions:

1. There is only one relation in a sentence. In practice, this is violated only in a very small number of cases.
2. Entities or features are nouns (includes nouns, plural nouns and proper nouns) and pronouns. These cover most cases. However, a feature can sometimes be a noun used in its verb form or some action described as a

verb (e.g., "Intel costs more"; "costs" is a verb and a feature). Such comparisons are adverbial comparisons and are not considered in [256].

Sequence Data Generation: A sequence database for mining is created as follows: Since we are interested in predicting and extracting items representing *entityS1* (denoted by $entityS1), *entityS2* (denoted by $entityS2), and features (denoted by $feature), which are all called **labels**, we first manually mark/label such words in each sentence in the training data. For example, in the sentence "Intel/NNP is/VBZ better/JJR than/IN amd/NN", the proper noun "Intel" is labeled with $entityS1, and the noun "amd" is labeled with $entityS2. The two labels are then used as pivots to generate sequence data. For every occurrence of a label in a sentence, a separate sequence is created and put in the sequence database. A radius of 4 is used in [256]. The following position words are also added to keep track of the distance between two items in a generated pattern:

1. Distance words = {*l*1, *l*2, *l*3, *l*4, *r*1, *r*2, *r*3, *r*4}, where *li* means distance of *i* to the left of the pivot, and *ri* means the distance of *i* to the right of pivot.
2. Special words *#start* and *#end* are used to mark the start and the end of a sentence.

Example 12: The comparative sentence "Canon/NNP has/VBZ better/JJR optics/NNS than/IN Nikon/NNP" has $entityS1 "Canon", $feature "optics" and $entityS2 "Nikon". The three sequences corresponding to the two entities and one feature put in the database are:

⟨{#start}{*l*1}{**$entityS1,** NNP}{*r*1}{has, VBZ}{r2}{better, JJR}
{*r*3}{$feature, NNS}{*r*4}{thanIN}⟩

⟨{#start}{*l*4}{$entityS1, NNP}{*l*3}{has, VBZ}{*l*2}{better, JJR} {*l*1}
{**$feature**, NNS}{*r*1}{thanIN}{*r*2}{entityS2, NNP}{*r*3} {#end}⟩

⟨{has, VBZ}{*l*4}{better, JJR}{*l*3}{$feature, NNS}{*l*2}{thanIN}
{*l*1}{**$entityS2,** NNP}{*r*1}{#end}⟩.

The keyword "than" is merged with its POS tag to form a single item.

LSR Generation: After the sequence database is built, a rule mining system is applied to generate label sequential rules. Note that only those rules that contain one or more labels (i.e., $entityS1, $entityS2, and $feature) will be generated. An example of a LSR rule is as follows

Rule 1: ⟨{*, NN}{VBZ}{JJR}{thanIN}{*, NN}⟩ →
⟨{$entityS1, NN}{VBZ}{JJR}{thanIN}{$entityS2, NN}⟩.

Relation Item Extraction: The generated LSRs are used to extract relation items from each input (or test) sentence. One strategy is to use all the

rules to match the sentence and to extract the relation items using the rule with the highest confidence. For example, the above rule will label and extract "coke" as entityS1, and "pepsi" as entityS2 from the following sentence:

⟨{coke, NN}{is, VBZ}{definitely, RB}{better, JJR}{thanIN}{pepsi, NN}⟩.

There is no feature in this sentence. The *relationWord* is simply the keyword that identifies the sentence as a comparative sentence. In this case, it is "better." A similar but more complex method is used in [256].

Again, many other methods can also be applied to the extraction, e.g., conditional random fields, hidden Markov models, and others. Results in [256] show that the LSR-based method outperforms conditional random fields. Further research and more comprehensive evaluations are needed to assess the strengths and weaknesses of these methods.

11.4 Opinion Search

Like the general Web search, one can also crawl the user-generated content on the Web and provide an opinion search service. The objective is to enable users to search for opinions on any object. Let us look at some typical opinion search queries:

1. Search for opinions on a particular object or feature of an object, e.g., customer opinions on a digital camera or the picture quality of a digital camera, or public opinions on a political topic. Recall that the object can be a product, organization, topic, etc.
2. Search for opinions of a person or organization (i.e., opinion holder) on a particular object or feature of the object. For example, one may search for Bill Clinton's opinion on abortion or a particular aspect of it. This type of search is particularly relevant to news documents, where individuals or organizations who express opinions are explicitly stated. In the user-generated content on the Web, the opinions are mostly expressed by authors of the postings.

For the first type of queries, the user may simply give the name of the object and/or some features of the object. For the second type of queries, the user may give the name of the opinion holder and also the name of the object. Clearly, it is not appropriate to simply apply keyword matching for either type of queries because a document containing the query words may not have opinions. For example, many discussion and blog posts do not contain opinions, but only questions and answers on some objects. Opinionated documents or sentences need to be identified before search is per-

formed. Thus, the simplest form of opinion search can be keyword-based search applied to the identified opinionated documents/sentences.

As for ranking, traditional Web search engines rank Web pages based on authority and relevance scores. The basic premise is that the top ranked pages (ideally the first page) contain sufficient information to satisfy the user's information need. This may be fine for the second type of queries because the opinion holder usually has only one opinion on the search object, and the opinion is usually contained in a single document or page (in some cases, using a general search engine with an appropriate set of keywords may be sufficient to find answers for such queries). However, for the first type of opinion queries, the top ranked documents only represent the opinions of a few persons. Therefore, they need to reflect the natural distribution of positive and negative sentiments of the whole population. Moreover, in many cases, opinionated documents are very long (e.g., reviews). It is hard for the user to read many of them in order to obtain a complete picture of the prevailing sentiments. Some form of summary of opinions is desirable, which can be either a simple rating average of reviews and proportions of positive and negative opinions, or a sophisticated feature-based summary as we discussed earlier. To make it even easier for the user, two rankings may be produced, one for positive opinions and one for negative opinions.

Providing a feature-based summary for each search query is an ideal solution. An analogy can be drawn from traditional surveys or opinion polls. An opinionated document is analogous to a filled survey form. Once all or a sufficient number of survey forms are collected, some analysts will analyze them to produce a survey summary, which is usually in the form of a bar or pie chart. One seldom shows all the filled survey forms to users (e.g., the management team of an organization or the general public) and asks them to read everything in order to draw their own conclusions. However, automatically generating a feature-based summary for each search object (or query) is a very challenging problem. To build a practical search system, some intermediate solution based on Problem 2 and 3 in Sect. 11.2.1 may be more appropriate.

Opinions also have a temporal dimension. For example, the opinions of people on a particular object, e.g., a product or a topic, may change over time. Displaying the changing trend of sentiments along the time axis can be very useful in many applications.

Finally, like opinion search, comparison search will be useful as well. For example, when you want to register for a free email account, you most probably want to know which email system is best for you, e.g., hotmail, gmail or *Yahoo*! mail. Wouldn't it be nice if you can find comparisons of

features of these email systems from existing users by issuing a search query "hotmail vs. gmail vs. yahoo mail."?

11.5 Opinion Spam

In Sect. 6.10, we discussed Web spam, which refers to the use of "illegitimate means" to boost the search rank position of some target Web pages. The reason for spamming is because of the economic and/or publicity value of the rank position of a page returned by a search engine. In the context of opinions on the Web, the problem is similar. It has become a common practice for people to find and to read opinions on the Web for many purposes. For example, if one wants to buy a product, one typically goes to a merchant or review site (e.g., amazon.com) to read some reviews of existing users of the product. If one sees many positive reviews of the product, one is very likely to buy the product. On the contrary, if one sees many negative reviews, he/she will most likely choose another product. Positive opinions can result in significant financial gains and/or fames for organizations and individuals. This, unfortunately, gives good incentives for **opinion spam**, which refers to human activities (e.g., write spam reviews) that try to deliberately mislead readers or automated opinion mining systems by giving undeserving positive opinions to some target objects in order to promote the objects and/or by giving unjust or false negative opinions on some other objects in order to damage their reputation. In this section, we use customer reviews of products as an example to study opinion spam on the Web. Most of the analyses are also applicable to opinions expressed in other forms of user-generated contents, e.g., forum postings, group discussions, and blogs.

11.5.1 Objectives and Actions of Opinion Spamming

As we indicated above, there are two main objectives for writing spam reviews:

1. To promote some target objects, e.g., one's own products.
2. To damage the reputation of some other target objects, e.g., products of one's competitors.

In certain cases, the spammer may want to achieve both objectives, while in others, he/she only aims to achieve one of them because either he/she does not have an object to promote or there is no competition. Another objective is also possible but may be rare. That is, the spammer writes some

irrelevant information or false information in order to annoy readers and to fool automated opinion mining systems.

To achieve the above objectives, the spammer usually takes both or one of the actions below:

- Write undeserving positive reviews for the target objects in order to promote them. We call such spam reviews **hype spam**.
- Write unfair or malicious negative reviews for the target objects to damage their reputation. We call such spam review **defaming spam**.

11.5.2 Types of Spam and Spammers

Table 11.3 below gives a simplified view of spam reviews. Spam reviews in regions 1, 3 and 5 are typically written by owners or manufacturers of the product or persons who have direct economic or other interests in the product. Their main goal is to promote the product. Although opinions expressed in reviews of region 1 may be true, reviewers do not announce their conflict of interests.

Spam reviews in regions 2, 4, and 6 are likely to be written by competitors, who give false information in order to damage the reputation of the product. Although opinions in reviews of region 4 may be true, reviewers do not announce their conflict of interests and may have malicious intensions.

Table 11.3. Spam reviews vs. product quality

	Hype spam review	Defaming spam review
Good quality product	1	**2**
Poor quality product	**3**	4
In-between good and poor quality product	**5**	**6**

Clearly, spam reviews in region 1 and 4 are not so damaging, while spam reviews in regions 2, 3, 5 and 6 are very harmful. Thus, spam detection techniques should focus on identifying reviews in these regions.

Manual and Automated Spam: Spam reviews may be manually written or automatically generated. Writing spam reviews manually is not a simple task if one wants to spam on a product at many review sites and write them differently to avoid being detected by methods that catch near duplicate reviews. Using some language templates, it is also possible to automatically generate many different variations of the same review.

Individual Spammers and Group Spammers: A spammer may act individually (e.g., the author of a book) or as a member of a group (e.g., a group of employees of a company).

Individual spammers: In this case, a spammer, who does not work with anyone else, writes spam reviews. The spammer may register at a review site as a single user, or as "many users" using different user-ids. He/she can also register at multiple review sites and write spam reviews.

Group spammers: A group of spammers works collaboratively to promote a target object and/or to damage the reputation of another object. They may also register at multiple sites and spam at these sites. Group spam can be very damaging because they may take control of the sentiments on the product and completely mislead potential customers.

11.5.3 Hiding Techniques

In order to avoid being detected, spammers may take a variety of precautions. We study individual and group of spammers separately. The lists are by no means exhaustive and should be considered as just examples.

An Individual Spammer

1. The spammer builds up reputation by reviewing other products in the same or different categories/brands that he/she does not care about and give them agreeable ratings and reasonable reviews. Then, he/she becomes a trustworthy reviewer. However, he/she may write spam reviews on the products that he/she really cares about. This hiding method is useful because some sites rank reviewers based on their reviews that are found helpful by readers, e.g., amazon.com. Some sites also have trust systems that allow readers to assign trust scores to reviewers.
2. The spammer registers multiple times at a site using different user-ids and write multiple spam reviews under these user-ids so that their reviews or ratings will not appear as outliers. The spammer may even use different machines to avoid being detected by server log based detection methods that can compare IP addresses of reviewers (discussed below).
3. The spammer gives a reasonably high rating but write a critical (negative) review. This may fool detection methods that find outliers based on ratings alone. Yet, automated review mining systems will pick up all the negative sentiments in the actual review content.
4. Spammers write either only positive reviews on his/her own products or only negative reviews on the products of his/her competitors, but not both. This is to hide from spam detection methods that compare one's reviews on competing products from different brands.

A Group of Spammers

1. Every member of the group reviews the same product to lower the rating deviation.
2. Every member of the group writes a review roughly at the time when the product is launched in order to take control of the product. It is generally not a good idea to write many spam reviews at the same time after many reviews have been written by others because a spike will appear, which can be easily detected.
3. Members of the group write reviews at random or irregular intervals to hide spikes.
4. If the group is sufficiently large, it may be divided into sub-groups so that each sub-group can spam at different web sites (instead of only spam at the same site) to avoid being detected by methods that compare average ratings and content similarities of reviews from different sites.

11.5.4 Spam Detection

So far, little study has been done on opinion spam detection. This subsection outlines some possible approaches. We note that each individual technique below may not be able to reliably detect spam reviews, but it can be treated as a spam indicator. A holistic approach that combines all evidences is likely to be effective. One possible combination method is to treat spam detection as a classification problem. All the individual methods simply compute spam evidences which can be put in a data set from which a spam classifier can be learned. For this approach to work, a set of reviews needs to be manually labeled for learning. The resulting classifier can be used to classify each new review as a spam review or not one.

Review Centric Spam Detection: In this approach, spam detection is based only on reviews. A review has two main parts: rating and content.

Compare content similarity: In order to have the maximum impact, a spammer may write multiple reviews on the same product (using different user-ids) or multiple products of the same brand. He/she may also write reviews at multiple review sites. However, for a single spammer to write multiple reviews that look very different is not an easy task. Thus, some spammers simply use the same review or slight variations of the same review. In a recent study of reviews from amazon.com, it was found that some spammers were so lazy that they simply copied the same review and pasted it for many different products of the same brand. Techniques that can detect near duplicate documents are useful here (see Sect. 6.5.5). For automatically generated spam reviews based on lan-

guage templates, sophisticated pattern mining methods may be needed to detect them.

Detect rating and content outliers: If we assume that reviews of a product contain only a very small proportion of spam, we can detect possible spam activities based on rating deviations, especially for reviews in region 2 and 3, because they tend to be outliers. For reviews in regions 5 and 6, this method may not be effective.

If a product has a large proportion of spam reviews, it is hard to detect them based on review ratings, even though each spammer may act independently, because they are no longer outliers. In this case, we may need to employ reviewer centric and server centric spam detection methods below. This case is similar to group spam, which is also hard to detect based on content alone because the spam reviews are written by different members of the group and there are a large number of them. Hence, their reviews are not expected to be outliers. However, members of the group may be detected based on reviewer centric detection methods and server centric detection methods. The following methods are also helpful.

Compare average ratings from multiple sites: This method is useful to access the level of spam activities from a site if only a small number of review sites are spammed. For example, if the averages rating at many review sites for a product are quite high but at one site it is quite low, this is an indication that there may be some group spam activities going on.

Detect rating spikes: This method looks at the review ratings (or contents) from the time series point of view. If a number of reviews with similar ratings come roughly at the same time, a spike will appear which indicates a possible group spam.

Reviewer Centric Spam Detection: In this approach, "unusual" behaviors of reviewers are exploited for spam detection. It is assumed that all the reviews of each reviewer at a particular site are known. Most review sites provide such information, e.g., amazon.com, or such information can be found by matching user-ids.

Watch early reviewers: Spammers are often the first few reviewers to review a product because earlier reviews tend to have a bigger impact. Their ratings for each product are in one of the two extremes, either very high or very low. They may do this consistently for a number of products of the same brand.

Detect early remedial actions: For a given product, as soon as someone writes a (or the first) negative review to a product, the spammer gives a positive review just after it, or vice versa.

Compare review ratings of the same reviewer on products from different brands: A spammer often writes very positive reviews for products of

one brand (to promote the product) and very negative reviews for similar products of another brand. A rating (or content) comparison will show discrepancies. If some of the ratings also deviate a great deal from the average ratings of the products, this is a good indicator of possible spam.

Compare review times: A spammer may review a few products from different brands at roughly the same time. Such behaviors are unusual for a normal reviewer.

As we mentioned above, detecting a group of spammers is difficult. However, we can reduce their negative impact by detecting each individual member in the group using the above and below methods.

Server centric spam detection: The server log at the review site can be helpful in spam detection as well. If a single person registers multiple times at a Web site having the same IP address, and the person also writes multiple reviews for the same product or even different products using different user-ids, it is fairly certain that the person is a spammer. Using the server log may also detect some group spam activities. For example, if most good reviews of a product are from a particular region where the company that produces the product is located, it is a good indication that these are likely spam.

As more and more people and organizations are using opinions on the Web for decision making, spammers have more and more incentives to express false sentiments in product reviews, discussions and blogs. To ensure the quality of information provided by an opinion mining and/or search system, spam detection is a critical task. Without effective detection, opinions on the Web may become useless. This section analyzed various aspects of opinion spam and outlined some possible detection methods. This may just be the beginning of a long journey of the "arms race" between spam and detection of spam.

Bibliographic Notes

Opinion mining received a great deal of attention recently due to the availability of a huge volume of online documents and user-generated content on the Web, e.g., reviews, forum discussions, and blogs. The problem is intellectually challenging, and also practically useful. The most widely studied sub-problem is sentiment classification, which classifies evaluative texts or sentences as positive, negative, or neutral. Representative works on classification at the document level include those by Turney [521], Pang et al. [428], and Dave et al. [122]. They have been discussed in this

12 Web Usage Mining

By *Bamshad Mobasher*

With the continued growth and proliferation of e-commerce, Web services, and Web-based information systems, the volumes of **clickstream** and **user data** collected by Web-based organizations in their daily operations has reached astronomical proportions. Analyzing such data can help these organizations determine the life-time value of clients, design cross-marketing strategies across products and services, evaluate the effectiveness of promotional campaigns, optimize the functionality of Web-based applications, provide more personalized content to visitors, and find the most effective logical structure for their Web space. This type of analysis involves the automatic discovery of meaningful patterns and relationships from a large collection of primarily semi-structured data, often stored in Web and applications server access logs, as well as in related operational data sources.

Web usage mining refers to the automatic discovery and analysis of patterns in clickstream and associated data collected or generated as a result of user interactions with Web resources on one or more Web sites [114, 387, 505]. The goal is to capture, model, and analyze the behavioral patterns and profiles of users interacting with a Web site. The discovered patterns are usually represented as collections of pages, objects, or resources that are frequently accessed by groups of users with common needs or interests.

Following the standard data mining process [173], the overall Web usage mining process can be divided into three inter-dependent stages: data collection and pre-processing, pattern discovery, and pattern analysis. In the pre-processing stage, the clickstream data is cleaned and partitioned into a set of user transactions representing the activities of each user during different visits to the site. Other sources of knowledge such as the site content or structure, as well as semantic domain knowledge from site **ontologies** (such as **product catalogs** or **concept hierarchies**), may also be used in pre-processing or to enhance user transaction data. In the pattern discovery stage, statistical, database, and machine learning operations are performed to obtain hidden patterns reflecting the typical behavior of users, as well as summary statistics on Web resources, sessions, and users. In the final stage of the process, the discovered patterns and statistics are further processed, filtered, possibly resulting in aggregate user models that can be

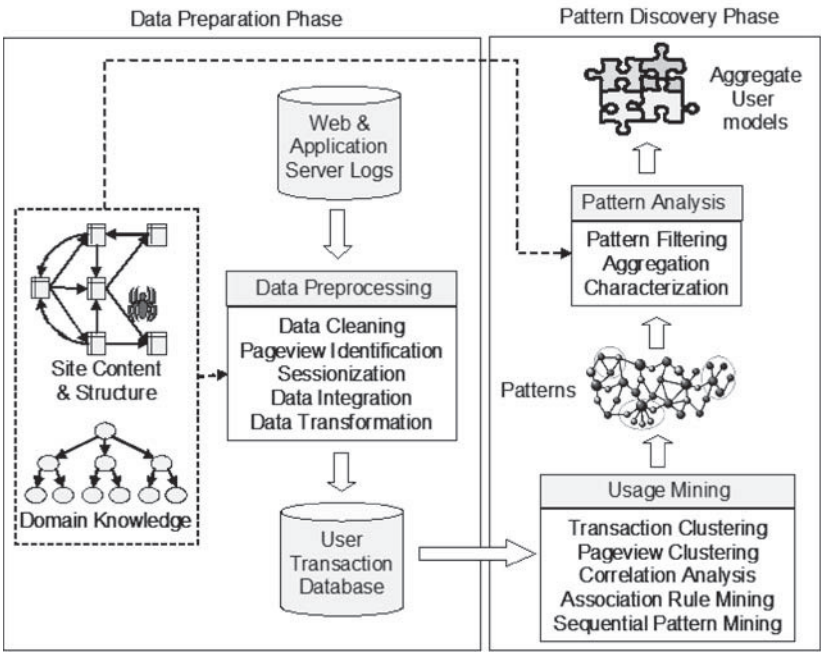


Fig. 12.1. The Web usage mining process

used as input to applications such as **recommendation engines**, visualization tools, and Web analytics and report generation tools. The overall process is depicted in Fig. 12.1.

In the remainder of this chapter, we provide a detailed examination of Web usage mining as a process, and discuss the relevant concepts and techniques commonly used in all the various stages mentioned above.

12.1 Data Collection and Pre-Processing

An important task in any data mining application is the creation of a suitable target data set to which data mining and statistical algorithms can be applied. This is particularly important in Web usage mining due to the characteristics of clickstream data and its relationship to other related data collected from multiple sources and across multiple channels. The data preparation process is often the most time consuming and computationally intensive step in the Web usage mining process, and often requires the use of special algorithms and heuristics not commonly employed in other domains. This process is critical to the successful extraction of useful patterns

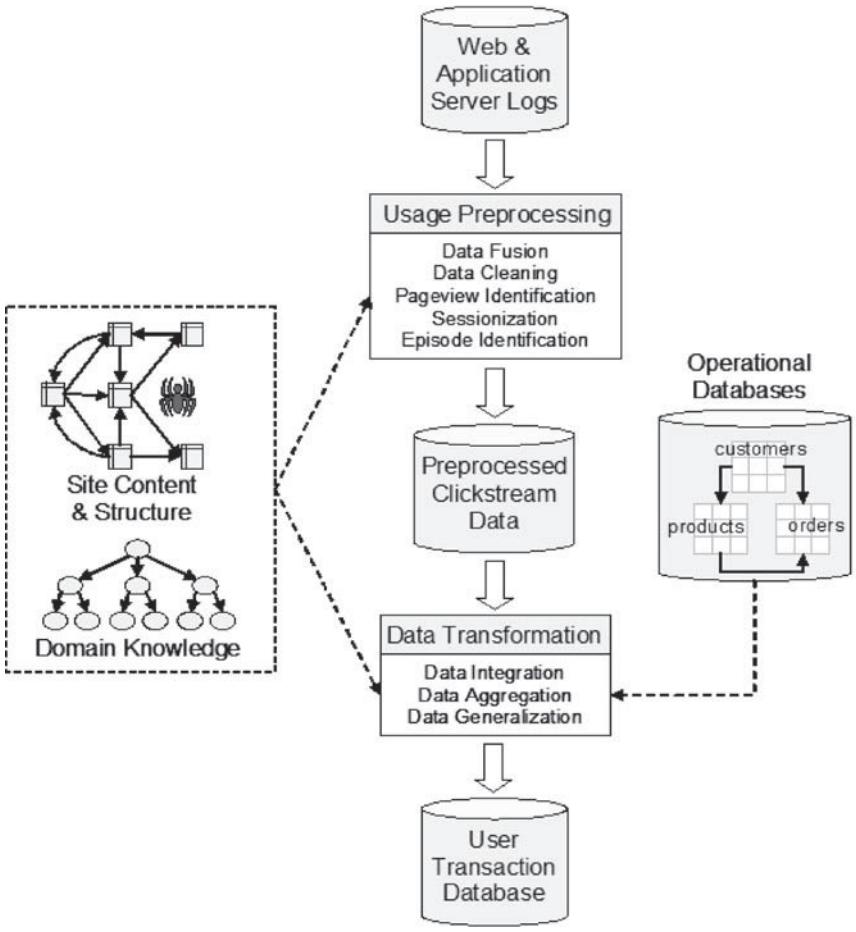


Fig. 12.2. Steps in data preparation for Web usage mining.

from the data. The process may involve pre-processing the original data, integrating data from multiple sources, and transforming the integrated data into a form suitable for input into specific data mining operations. Collectively, we refer to this process as *data preparation*.

Much of the research and practice in usage data preparation has been focused on pre-processing and integrating these data sources for different analysis. Usage data preparation presents a number of unique challenges which have led to a variety of algorithms and heuristic techniques for preprocessing tasks such as data fusion and cleaning, user and session identification, pageview identification [115]. The successful application of data mining techniques to Web usage data is highly dependent on the correct application of the pre-processing tasks. Furthermore, in the context of e-

commerce data analysis, these techniques have been extended to allow for the discovery of important and insightful user and site metrics [286].

Figure 12.2 provides a summary of the primary tasks and elements in usage data pre-processing. We begin by providing a summary of data types commonly used in Web usage mining and then provide a brief discussion of some of the primary data preparation tasks.

12.1.1 Sources and Types of Data

The primary data sources used in Web usage mining are the **server log** files, which include **Web server access logs** and **application server logs**. Additional data sources that are also essential for both data preparation and pattern discovery include the site files and meta-data, operational databases, application templates, and domain knowledge. In some cases and for some users, additional data may be available due to client-side or proxy-level (Internet Service Provider) data collection, as well as from external clickstream or **demographic data** sources such as those provided by data aggregation services from ComScore (www.comscore.com), NetRatings (www.nielsen-netratings.com), and Acxiom (www.acxiom.com).

The data obtained through various sources can be categorized into four primary groups [115, 505].

Usage Data: The log data collected automatically by the Web and application servers represents the fine-grained navigational behavior of visitors. It is the primary source of data in Web usage mining. Each hit against the server, corresponding to an HTTP request, generates a single entry in the server access logs. Each log entry (depending on the log format) may contain fields identifying the time and date of the request, the IP address of the client, the resource requested, possible parameters used in invoking a Web application, status of the request, HTTP method used, the user agent (browser and operating system type and version), the referring Web resource, and, if available, client-side **cookies** which uniquely identify a repeat visitor. A typical example of a server access log is depicted in Fig. 12.3, in which six partial log entries are shown. The user IP addresses in the log entries have been changed to protect privacy.

For example, log entry 1 shows a user with IP address "1.2.3.4" accessing a resource: "/classes/cs589/papers.html" on the server (maya.cs.depaul.edu). The browser type and version, as well as operating system information on the client machine are captured in the agent field of the entry. Finally, the referrer field indicates that the user came to this location from an outside source: "http://dataminingresources.blogspot.com/". The next log entry shows that this user has navigated from "papers.html" (as re-

1	2006-02-01 00:08:43 1.2.3.4 - GET /classes/cs589/papers.html - 200 9221 HTTP/1.1 maya.cs.depaul.edu Mozilla/4.0+(compatible;+MSIE+6.0;+Windows+NT+5.1;+SV1;+.NET+CLR+2.0.50727) http://dataminingresources.blogspot.com/
2	2006-02-01 00:08:46 1.2.3.4 - GET /classes/cs589/papers/cms-tai.pdf - 200 4096 HTTP/1.1 maya.cs.depaul.edu Mozilla/4.0+(compatible;+MSIE+6.0;+Windows+NT+5.1;+SV1;+.NET+CLR+2.0.50727) http://maya.cs.depaul.edu/~classes/cs589/papers.html
3	2006-02-01 08:01:28 2.3.4.5 - GET /classes/ds575/papers/hyperlink.pdf - 200 318814 HTTP/1.1 maya.cs.depaul.edu Mozilla/4.0+(compatible;+MSIE+6.0;+Windows+NT+5.1) http://www.google.com/search?hl=en&lr=&q=hyperlink+analysis+for+the+web+survey
4	2006-02-02 19:34:45 3.4.5.6 - GET /classes/cs480/announce.html - 200 3794 HTTP/1.1 maya.cs.depaul.edu Mozilla/4.0+(compatible;+MSIE+6.0;+Windows+NT+5.1;+SV1) http://maya.cs.depaul.edu/~classes/cs480/
5	2006-02-02 19:34:45 3.4.5.6 - GET /classes/cs480/styles2.css - 200 1636 HTTP/1.1 maya.cs.depaul.edu Mozilla/4.0+(compatible;+MSIE+6.0;+Windows+NT+5.1;+SV1) http://maya.cs.depaul.edu/~classes/cs480/announce.html
6	2006-02-02 19:34:45 3.4.5.6 - GET /classes/cs480/header.gif - 200 6027 HTTP/1.1 maya.cs.depaul.edu Mozilla/4.0+(compatible;+MSIE+6.0;+Windows+NT+5.1;+SV1) http://maya.cs.depaul.edu/~classes/cs480/announce.html

Fig. 12.3. Portion of a typical server log

flected in the referrer field of entry 2) to access another resource: "/classes/cs589/papers/cms-tai.pdf". Log entry 3 shows a user who has arrived at the resource "/classes/ds575/papers/hyperlink.pdf" by doing a search on Google using keyword query: "hyperlink analysis for the web survey". Finally, entries 4–6 all correspond to a single click-through by a user who has accessed the resource "/classes/cs480/announce.html". Entries 5 and 6 are images embedded in the file "announce.html" and thus two additional HTTP request are registered as hits in the server log corresponding to these images.

Depending on the goals of the analysis, this data needs to be transformed and aggregated at different levels of abstraction. In Web usage mining, the most basic level of data abstraction is that of a **pageview**. A pageview is an aggregate representation of a collection of Web objects contributing to the display on a user's browser resulting from a single user action (such as a click-through). Conceptually, each pageview can be viewed as a collection of Web objects or resources representing a specific "user event," e.g., reading an article, viewing a product page, or adding a product to the shopping cart. At the user level, the most basic level of behavioral abstraction is that of a **session**. A session is a sequence of pageviews by a single user during a single visit. The notion of a session can be

further abstracted by selecting a subset of pageviews in the session that are significant or relevant for the analysis tasks at hand.

Content Data: The content data in a site is the collection of objects and relationships that is conveyed to the user. For the most part, this data is comprised of combinations of textual materials and images. The data sources used to deliver or generate this data include static HTML/XML pages, multimedia files, dynamically generated page segments from scripts, and collections of records from the operational databases. The site content data also includes semantic or structural meta-data embedded within the site or individual pages, such as descriptive keywords, document attributes, semantic tags, or HTTP variables. The underlying domain ontology for the site is also considered part of the content data. Domain ontologies may include conceptual hierarchies over page contents, such as product categories, explicit representations of semantic content and relationships via an ontology language such as RDF, or a database schema over the data contained in the operational databases.

Structure Data: The structure data represents the designer's view of the content organization within the site. This organization is captured via the inter-page linkage structure among pages, as reflected through hyperlinks. The structure data also includes the intra-page structure of the content within a page. For example, both HTML and XML documents can be represented as tree structures over the space of tags in the page. The hyperlink structure for a site is normally captured by an automatically generated "site map." A site mapping tool must have the capability to capture and represent the inter- and intra-pageview relationships. For dynamically generated pages, the site mapping tools must either incorporate intrinsic knowledge of the underlying applications and scripts that generate HTML content, or must have the ability to generate content segments using a sampling of parameters passed to such applications or scripts.

User Data: The operational database(s) for the site may include additional user profile information. Such data may include demographic information about registered users, user ratings on various objects such as products or movies, past purchases or visit histories of users, as well as other explicit or implicit representations of users' interests. Some of this data can be captured anonymously as long as it is possible to distinguish among different users. For example, anonymous information contained in client-side cookies can be considered a part of the users' profile information, and used to identify repeat visitors to a site. Many personalization applications require the storage of prior user profile information.

12.1.2 Key Elements of Web Usage Data Pre-Processing

As noted in Fig. 12.2, the required high-level tasks in usage data pre-processing include the fusion and synchronization of data from multiple log files, data cleaning, pageview identification, user identification, session identification (or sessionization), episode identification, and the integration of clickstream data with other data sources such as content or semantic information, as well as user and product information from operational databases. We now examine some of the essential tasks in pre-processing.

Data Fusion and Cleaning

In large-scale Web sites, it is typical that the content served to users comes from multiple Web or application servers. In some cases, multiple servers with redundant content are used to reduce the load on any particular server. Data fusion refers to the merging of log files from several Web and application servers. This may require global synchronization across these servers. In the absence of shared embedded session ids, heuristic methods based on the "referrer" field in server logs along with various sessionization and user identification methods (see below) can be used to perform the merging. This step is essential in "inter-site" Web usage mining where the analysis of user behavior is performed over the log files of multiple related Web sites [513].

Data cleaning is usually site-specific, and involves tasks such as, removing extraneous references to embedded objects that may not be important for the purpose of analysis, including references to style files, graphics, or sound files. The cleaning process also may involve the removal of at least some of the data fields (e.g. number of bytes transferred or version of HTTP protocol used, etc.) that may not provide useful information in analysis or data mining tasks.

Data cleaning also entails the removal of references due to crawler navigations. It is not uncommon for a typical log file to contain a significant (sometimes as high as 50%) percentage of references resulting from search engine or other crawlers (or spiders). Well-known search engine crawlers can usually be identified and removed by maintaining a list of known crawlers. Other "well-behaved" crawlers which abide by standard robot exclusion protocols, begin their site crawl by first attempting to access to exclusion file "robot.txt" in the server root directory. Such crawlers, can therefore, be identified by locating all sessions that begin with an (attempted) access to this file. However, a significant portion of crawlers references are from those that either do not identify themselves explicitly (e.g., in the "agent" field) or implicitly; or from those crawlers that delib-

erately masquerade as legitimate users. In this case, identification and removal of crawler references may require the use of heuristic methods that distinguish typical behavior of Web crawlers from those of actual users. Some work has been done on using classification algorithms to build models of crawlers and Web robot navigations [510], but such approaches have so far been met with only limited success and more work in this area is required.

Pageview Identification

Identification of pageviews is heavily dependent on the intra-page structure of the site, as well as on the page contents and the underlying site domain knowledge. Recall that, conceptually, each pageview can be viewed as a collection of Web objects or resources representing a specific "user event," e.g., clicking on a link, viewing a product page, adding a product to the shopping cart. For a static single frame site, each HTML file may have a one-to-one correspondence with a pageview. However, for multi-framed sites, several files make up a given pageview. For dynamic sites, a pageview may represent a combination of static templates and content generated by application servers based on a set of parameters.

In addition, it may be desirable to consider pageviews at a higher level of aggregation, where each pageview represents a collection of pages or objects, for examples, pages related to the same concept category. In e-commerce Web sites, pageviews may correspond to various product-oriented events, such as product views, registration, shopping cart changes, purchases, etc. In this case, identification of pageviews may require *a priori* specification of an "event model" based on which various user actions can be categorized.

In order to provide a flexible framework for a variety of data mining activities a number of attributes must be recorded with each pageview. These attributes include the pageview id (normally a URL uniquely representing the pageview), static pageview type (e.g., information page, product view, category view, or index page), and other metadata, such as content attributes (e.g., keywords or product attributes).

User Identification

The analysis of Web usage does not require knowledge about a user's identity. However, it is necessary to distinguish among different users. Since a user may visit a site more than once, the server logs record multiple sessions for each user. We use the phrase **user activity record** to refer to the sequence of logged activities belonging to the same user.

In the absence of authentication mechanisms, the most widespread approach to distinguishing among unique visitors is the use of client-side cookies. Not all sites, however, employ cookies, and due to privacy concerns, client-side cookies are sometimes disabled by users. IP addresses, alone, are not generally sufficient for mapping log entries onto the set of unique visitors. This is mainly due to the proliferation of ISP proxy servers which assign rotating IP addresses to clients as they browse the Web. It is not uncommon to find many log entries corresponding to a limited number of proxy server IP addresses from large Internet Service Providers such as America Online. Therefore, two occurrences of the same IP address (separated by a sufficient amount of time), in fact, might correspond to two different users. Without user authentication or client-side cookies, it is still possible to accurately identify unique users through a combination of IP addresses and other information such as user agents and referrers [115].

Consider, for instance, the example of Fig. 12.4. On the left, the figure depicts a portion of a partly preprocessed log file (the time stamps are given as hours and minutes only). Using a combination of IP and Agent fields in the log file, we are able to partition the log into activity records for three separate users (depicted on the right).

Time	IP	URL	Ref	Agent
0:01	1.2.3.4	A	-	IE5;Win2k
0:09	1.2.3.4	B	A	IE5;Win2k
0:10	2.3.4.5	C	-	IE6;WinXP;SP1
0:12	2.3.4.5	B	C	IE6;WinXP;SP1
0:15	2.3.4.5	E	C	IE6;WinXP;SP1
0:19	1.2.3.4	C	A	IE5;Win2k
0:22	2.3.4.5	D	B	IE6;WinXP;SP1
0:22	1.2.3.4	A	-	IE6;WinXP;SP2
0:25	1.2.3.4	E	C	IE5;Win2k
0:25	1.2.3.4	C	A	IE6;WinXP;SP2
0:33	1.2.3.4	B	C	IE6;WinXP;SP2
0:58	1.2.3.4	D	B	IE6;WinXP;SP2
1:10	1.2.3.4	E	D	IE6;WinXP;SP2
1:15	1.2.3.4	A	-	IE5;Win2k
1:16	1.2.3.4	C	A	IE5;Win2k
1:17	1.2.3.4	F	C	IE6;WinXP;SP2
1:26	1.2.3.4	F	C	IE5;Win2k
1:30	1.2.3.4	B	A	IE5;Win2k
1:36	1.2.3.4	D	B	IE5;Win2k

User 1

0:01	1.2.3.4	A	-
0:09	1.2.3.4	B	A
0:19	1.2.3.4	C	A
0:25	1.2.3.4	E	C
1:15	1.2.3.4	A	-
1:26	1.2.3.4	F	C
1:30	1.2.3.4	B	A
1:36	1.2.3.4	D	B

User 2

0:10	2.3.4.5	C	-
0:12	2.3.4.5	B	C
0:15	2.3.4.5	E	C
0:22	2.3.4.5	D	B

User 3

0:22	1.2.3.4	A	-
0:25	1.2.3.4	C	A
0:33	1.2.3.4	B	C
0:58	1.2.3.4	D	B
1:10	1.2.3.4	E	D
1:17	1.2.3.4	F	C

Fig. 12.4. Example of user identification using IP + Agent

Sessionization

Sessionization is the process of segmenting the user activity record of each user into sessions, each representing a single visit to the site. Web sites without the benefit of additional authentication information from users and without mechanisms such as embedded session ids must rely on heuristics methods for sessionization. The goal of a sessionization heuristic is to re-construct, from the clickstream data, the actual sequence of actions performed by one user during one visit to the site.

We denote the "conceptual" set of real sessions by R, representing the real activity of the user on the Web site. A sessionization heuristic h attempts to map R into a set of constructed sessions, denoted by C_h. For the ideal heuristic, h^*, we have $C_{h^*} = R$. In other words, the ideal heuristic can re-construct the exact sequence of user navigation during a session. Generally, sessionization heuristics fall into two basic categories: time-oriented or structure-oriented. Time-oriented heuristics apply either global or local time-out estimates to distinguish between consecutive sessions, while structure-oriented heuristics use either the static site structure or the implicit linkage structure captured in the referrer fields of the server logs. Various heuristics for sessionization have been identified and studied [115]. More recently, a formal framework for measuring the effectiveness of such heuristics has been proposed [498], and the impact of different heuristics on various Web usage mining tasks has been analyzed [46].

As an example, two variations of time-oriented heuristics and a basic navigation-oriented heuristic are given below. Each heuristic h scans the user activity logs to which the Web server log is partitioned after user identification, and outputs a set of constructed sessions:

- ***h*1:** Total session duration may not exceed a threshold θ. Given t_0, the timestamp for the first request in a constructed session S, the request with a timestamp t is assigned to S, iff $t - t_0 \leq \theta$.
- ***h*2:** Total time spent on a page may not exceed a threshold δ. Given t_1, the timestamp for request assigned to constructed session S, the next request with timestamp t_2 is assigned to S, iff $t_2 - t_1 \leq \delta$.
- ***h*-ref:** A request q is added to constructed session S if the referrer for q was previously invoked in S. Otherwise, q is used as the start of a new constructed session. Note that with this heuristic it is possible that a request q may potentially belong to more than one "open" constructed session, since q may have been accessed previously in multiple sessions. In this case, additional information can be used for disambiguation. For example, q could be added to the most recently opened session satisfying the above condition.

User 1

Time	IP	URL	Ref
0:01	1.2.3.4	A	-
0:09	1.2.3.4	B	A
0:19	1.2.3.4	C	A
0:25	1.2.3.4	E	C
1:15	1.2.3.4	A	-
1:26	1.2.3.4	F	C
1:30	1.2.3.4	B	A
1:36	1.2.3.4	D	B

Session 1

0:01	1.2.3.4	A	-
0:09	1.2.3.4	B	A
0:19	1.2.3.4	C	A
0:25	1.2.3.4	E	C

Session 2

1:15	1.2.3.4	A	-
1:26	1.2.3.4	F	C
1:30	1.2.3.4	B	A
1:36	1.2.3.4	D	B

Fig. 12.5. Example of sessionization with a time-oriented heuristic

User 1

Time	IP	URL	Ref
0:01	1.2.3.4	A	-
0:09	1.2.3.4	B	A
0:19	1.2.3.4	C	A
0:25	1.2.3.4	E	C
1:15	1.2.3.4	A	-
1:26	1.2.3.4	F	C
1:30	1.2.3.4	B	A
1:36	1.2.3.4	D	B

Session 1

0:01	1.2.3.4	A	-
0:09	1.2.3.4	B	A
0:19	1.2.3.4	C	A
0:25	1.2.3.4	E	C
1:26	1.2.3.4	F	C

Session 2

1:15	1.2.3.4	A	-
1:30	1.2.3.4	B	A
1:36	1.2.3.4	D	B

Fig. 12.6. Example of sessionization with the h-ref heuristic

An example of the application of sessionization heuristics is given in Fig. 12.5 and Fig. 12.6. In Fig. 12.5, the heuristic h_1, described above, with $\theta = 30$ minutes has been used to partition a user activity record (from the example of Fig. 12.4) into two separate sessions.

If we were to apply h_2 with a threshold of 10 minutes, the user record would be seen as three sessions, namely, A→B→C→E, A, and F→B→D. On the other hand, Fig. 12.6 depicts an example of using *h*-ref heuristic on the same user activity record. In this case, once the request for F (with time stamp 1:26) is reached, there are two open sessions, namely, A→B→C→E and A. But F is added to the first because its referrer, C, was invoked in session 1. The request for B (with time stamp 1:30) may potentially belong to both open sessions, since its referrer, A, is invoked both in session 1 and in session 2. In this case, it is added to the second session, since it is the most recently opened session.

Episode identification can be performed as a final step in pre-processing of the clickstream data in order to focus on the relevant subsets of pageviews in each user session. An **episode** is a subset or subsequence of a session comprised of semantically or functionally related pageviews. This task may require the automatic or semi-automatic classification of page-

views into different functional types or into concept classes according to a domain ontology or concept hierarchy. In highly dynamic sites, it may also be necessary to map pageviews within each session into "service-based" classes according to a concept hierarchy over the space of possible parameters passed to script or database queries [47]. For example, the analysis may ignore the quantity and attributes of an item added to the shopping cart, and focus only on the action of adding the item to the cart.

Path Completion

Another potentially important pre-processing task which is usually performed after sessionization is **path completion**. Client- or proxy-side caching can often result in missing access references to those pages or objects that have been cached. For instance, if a user returns to a page A during the same session, the second access to A will likely result in viewing the previously downloaded version of A that was cached on the client-side, and therefore, no request is made to the server. This results in the second reference to A not being recorded on the server logs. **Missing references** due to caching can be heuristically inferred through path completion which relies on the knowledge of site structure and referrer information from server logs [115]. In the case of dynamically generated pages, form-based applications using the HTTP POST method result in all or part of the user input parameter not being appended to the URL accessed by the user (though, in the latter case, it is possible to recapture the user input through **packet sniffers** which listen to all incoming and outgoing TCP/IP network traffic on the server side).

A simple example of missing references is given in Fig. 12.7. On the left, a graph representing the linkage structure of the site is given. The dotted arrows represent the navigational path followed by a hypothetical user. After reaching page E, the user has backtracked (e.g., using the browser's "back" button) to page D and then B from which she has navigated to page C. The back references to D and B do not appear in the log file because these pages where cached on the client-side (thus no explicit server request was made for these pages). The log file shows that after a request for E, the next request by the user is for page C with a referrer B. In other words, there is a gap in the activity record corresponding to user's navigation from page E to page B. Given the site graph, it is possible to infer the two missing references (i.e., E → D and D → B) from the site structure and the referrer information given above. It should be noted that there are, in general, many (possibly infinite), candidate completions (for example, consider the sequence E → D, D → B, B → A, A → B). A simple heuristic

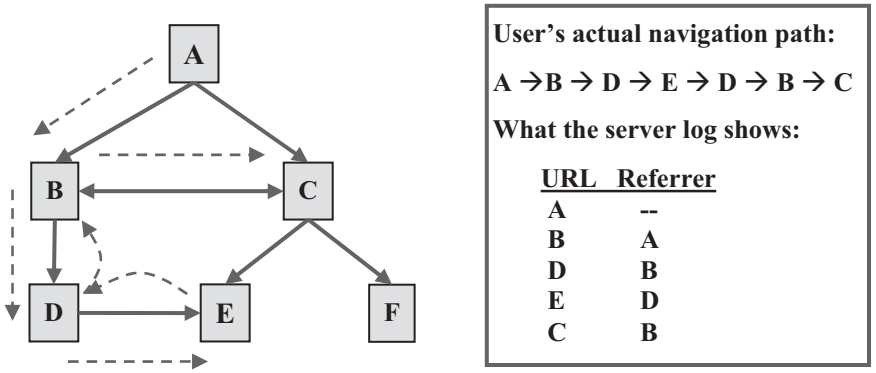


Fig. 12.7. Missing references due to caching.

that can be used for disambiguating among candidate paths is to select the one requiring the fewest number of "back" references.

Data Integration

The above pre-processing tasks ultimately result in a set of user sessions (or episodes), each corresponding to a delimited sequence of pageviews. However, in order to provide the most effective framework for pattern discovery, data from a variety of other sources must be integrated with the preprocessed clickstream data. This is particularly the case in e-commerce applications where the integration of both user data (e.g., demographics, ratings, and purchase histories) and product attributes and categories from operational databases is critical. Such data, used in conjunction with usage data, in the mining process can allow for the discovery of important business intelligence metrics such as **customer conversion ratios** and **lifetime values** [286].

In addition to user and product data, e-commerce data includes various product-oriented events such as shopping cart changes, order and shipping information, **impressions** (when the user visits a page containing an item of interest), **click-throughs** (when the user actually clicks on an item of interest in the current page), and other basic metrics primarily used for data analysis. The successful integration of these types of data requires the creation of a site-specific "event model" based on which subsets of a user's clickstream are aggregated and mapped to specific events such as the addition of a product to the shopping cart. Generally, the integrated e-commerce data is stored in the final transaction database. To enable full-featured Web analytics applications, this data is usually stored in a data warehouse called an **e-commerce data mart**. The e-commerce data mart

is a multi-dimensional database integrating data from various sources, and at different levels of aggregation. It can provide pre-computed e-metrics along multiple dimensions, and is used as the primary data source for OLAP (Online Analytical Processing), for data visualization, and in data selection for a variety of data mining tasks [71, 279]. Some examples of such metrics include frequency or monetary value of purchases, average size of market baskets, the number of different items purchased, the number of different item categories purchased, the amount of time spent on pages or sections of the site, day of week and time of day when a certain activity occurred, response to recommendations and online specials, etc.

12.2 Data Modeling for Web Usage Mining

Usage data pre-processing results in a set of n pageviews, $P = \{p_1, p_2, \cdots, p_n\}$, and a set of m **user transactions**, $T = \{t_1, t_2, \cdots, t_m\}$, where each t_i in T is a subset of P. **Pageviews** are semantically meaningful entities to which mining tasks are applied (such as pages or products). Conceptually, we view each transaction t as an l-length sequence of ordered pairs:

$$t = \left\langle (p_1^t, w(p_1^t)), (p_2^t, w(p_2^t)), \cdots, (p_l^t, w(p_l^t)) \right\rangle,$$

where each $p_i^t = p_j$ for some j in $\{1, 2, \cdots, n\}$, and $w(p_i^t)$ is the weight associated with pageview p_i^t in transaction t, representing its significance. The weights can be determined in a number of ways, in part based on the type of analysis or the intended personalization framework. For example, in **collaborative filtering** applications which rely on the profiles of similar users to make recommendations to the current user, weights may be based on user ratings of items. In most Web usage mining tasks the weights are either binary, representing the existence or non-existence of a pageview in the transaction; or they can be a function of the duration of the pageview in the user's session. In the case of time durations, it should be noted that usually the time spent by a user on the last pageview in the session is not available. One commonly used option is to set the weight for the last pageview to be the mean time duration for the page taken across all sessions in which the pageview does not occur as the last one. In practice, it is common to use a normalized value of page duration instead of raw time duration in order to account for user variances. In some applications, the log of pageview duration is used as the weight to reduce the noise in the data.

For many data mining tasks, such as clustering and association rule mining, where the ordering of pageviews in a transaction is not relevant, we can represent each user transaction as a vector over the n-dimensional space

Pageviews

Sessions / users	A	B	C	D	E	F
user0	15	5	0	0	0	185
user1	0	0	32	4	0	0
user2	12	0	0	56	236	0
user3	9	47	0	0	0	134
user4	0	0	23	15	0	0
user5	17	0	0	157	69	0
user6	24	89	0	0	0	354
user7	0	0	78	27	0	0
user8	7	0	45	20	127	0
user9	0	38	57	0	0	15

Fig. 12.8. An example of a user-pageview matrix (or transaction matrix)

of pageviews. Given the transaction t above, the transaction vector **t** (we use a bold face lower case letter to represent a vector) is given by:

$$\mathbf{t} = \left(w^t_{p_1}, w^t_{p_2}, \cdots, w^t_{p_n}\right),$$

where each $w^t_{p_i} = w(p^t_j)$, for some j in $\{1, 2, \cdots, n\}$, if p_j appears in the transaction t, and $w^t_{p_i} = 0$ otherwise. Thus, conceptually, the set of all user transactions can be viewed as an $m \times n$ **user-pageview matrix** (also called the **transaction matrix**), denoted by *UPM*.

An example of a hypothetical user-pageview matrix is depicted in Fig. 12.8. In this example, the weights for each pageview is the amount of time (e.g., in seconds) that a particular user spent on the pageview. In practice, these weights must be normalized to account for variances in viewing times by different users. It should also be noted that the weights may be composite or aggregate values in cases where the pageview represents a collection or sequence of pages and not a single page.

Given a set of transactions in the user-pageview matrix as described above, a variety of unsupervised learning techniques can be applied to obtain patterns. These techniques such as clustering of transactions (or sessions) can lead to the discovery of important user or visitor segments. Other techniques such as item (e.g., pageview) clustering and association or sequential pattern mining can find important relationships among items based on the navigational patterns of users in the site.

As noted earlier, it is also possible to integrate other sources of knowledge, such as semantic information from the content of Web pages with the Web usage mining process. Generally, the textual features from the content of Web pages represent the underlying semantics of the site. Each

pageview p can be represented as a r-dimensional feature vector, where r is the total number of extracted features (words or concepts) from the site in a global dictionary. This vector, denoted by **p**, can be given by:

$$\mathbf{p} = \left(fw^p(f_1), fw^p(f_2), ..., fw^p(f_r) \right)$$

where $fw^p(f_j)$ is the weight of the jth feature (i.e., f_j) in pageview p, for $1 \leq j \leq r$. For the whole collection of pageviews in the site, we then have an $n \times r$ **pageview-feature matrix** $PFM = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n\}$. The integration process may, for example, involve the transformation of user transactions (in user-pageview matrix) into "content-enhanced" transactions containing the semantic features of the pageviews. The goal of such a transformation is to represent each user session (or more generally, each user profile) as a vector of semantic features (i.e., textual features or concept labels) rather than as a vector over pageviews. In this way, a user's session reflects not only the pages visited, but also the significance of various concepts or context features that are relevant to the user's interaction.

While, in practice, there are several ways to accomplish this transformation, the most direct approach involves mapping each pageview in a transaction to one or more content features. The range of this mapping can be the full feature space, or feature sets (composite features) which in turn may represent concepts and concept categories. Conceptually, the transformation can be viewed as the multiplication of the user-pageview matrix *UPM*, defined earlier, with the pageview-feature matrix *PFM*. The result is a new matrix, $TFM = \{\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_m\}$, where each $\mathbf{t}_i$ is a r-dimensional vector over the feature space. Thus, a user transaction can be represented as a content feature vector, reflecting that user's interests in particular concepts or topics.

As an example of content-enhanced transactions, consider Fig. 12.9 which shows a hypothetical matrix of user sessions (user-pageview matrix) as well as a document index for the corresponding Web site conceptually represented as a **term-pageview matrix**. Note that the transpose of this term-pageview matrix is the pageview-feature matrix. The user-pageview matrix simply reflects the pages visited by users in various sessions. On the other hand, the term-pageview matrix represents the concepts that appear in each page. For simplicity we have assumed that all the weights are binary (however, note that in practice weights in the user transaction data are usually not binary and represent some measure of significance of the page in that transaction; and the weights in the term-pageview matrix are usually a function of term frequencies).

In this case, the corresponding **content-enhanced transaction matrix** (derived by multiplying the user-pageview matrix and the transpose of the term-pageview matrix) is depicted in Fig. 12.10. The resulting matrix

	A.html	B.html	C.html	D.html	E.html
user1	1	0	1	0	1
user2	1	1	0	0	1
user3	0	1	1	1	0
user4	1	0	1	1	1
user5	1	1	0	0	1
user6	1	0	1	1	1

	A.html	B.html	C.html	D.html	E.html
web	0	0	1	1	1
data	0	1	1	1	0
mining	0	1	1	1	0
business	1	1	0	0	0
intelligence	1	1	0	0	1
marketing	1	1	0	0	1
ecommerce	0	1	1	0	0
search	1	0	1	0	0
information	1	0	1	1	1
retrieval	1	0	1	1	1

Fig. 12.9. Examples of a user-pageview matrix (top) and a term-pageview matrix (bottom)

	web	data	mining	business	intelligence	marketing	ecommerce	search	information	retrieval
user1	2	1	1	1	2	2	1	2	3	3
user2	1	1	1	2	3	3	1	1	2	2
user3	2	3	3	1	1	1	2	1	2	2
user4	3	2	2	1	2	2	1	2	4	4
user5	1	1	1	2	3	3	1	1	2	2
user6	3	2	2	1	2	2	1	2	4	4

Fig. 12.10. The content-enhanced transaction matrix from matrices of Fig. 12.9

shows, for example, that users 4 and 6 are more interested in Web information retrieval, while user 3 is more interested in data mining.

Various data mining tasks can now be performed on the content-enhanced transaction data. For example, clustering the enhanced transaction matrix of Fig. 12.10 may reveal segments of users that have common interests in different concepts as indicated from their navigational behaviors.

If the content features include relational attributes associated with items on the Web site, then the discovered patterns may reveal user interests at the deeper semantic level reflected in the underlying properties of the items that are accessed by the users on the Web site. As an example, consider a site containing information about movies. The site may contain pages related to the movies themselves, as well as attributes describing the properties of each movie, such as actors, directors, and genres. The mining

process may, for instance, generate an association rule such as: {"British", "Romance", "Comedy" ⇒ "Hugh Grant"}, suggesting that users who are interested in British romantic comedies may also like the actor Hugh Grant (with a certain degree of confidence). Therefore, the integration of semantic content with Web usage mining can potentially provide a better understanding of the underlying relationships among objects.

12.3 Discovery and Analysis of Web Usage Patterns

The types and levels of analysis, performed on the integrated usage data, depend on the ultimate goals of the analyst and the desired outcomes. In this section we describe some of the most common types of pattern discovery and analysis techniques employed in the Web usage mining domain and discuss some of their applications.

12.3.1 Session and Visitor Analysis

The statistical analysis of pre-processed session data constitutes the most common form of analysis. In this case, data is aggregated by predetermined units such as days, sessions, visitors, or domains. Standard statistical techniques can be used on this data to gain knowledge about visitor behavior. This is the approach taken by most commercial tools available for Web log analysis. Reports based on this type of analysis may include information about most frequently accessed pages, average view time of a page, average length of a path through a site, common entry and exit points, and other aggregate measures. Despite a lack of depth in this type of analysis, the resulting knowledge can be potentially useful for improving the system performance, and providing support for marketing decisions. Furthermore, commercial Web analytics tools are increasingly incorporating a variety of data mining algorithms resulting in more sophisticated site and customer metrics.

Another form of analysis on integrated usage data is Online Analytical Processing (OLAP). OLAP provides a more integrated framework for analysis with a higher degree of flexibility. The data source for OLAP analysis is usually a multidimensional data warehouse which integrates usage, content, and e-commerce data at different levels of aggregation for each dimension. OLAP tools allow changes in aggregation levels along each dimension during the analysis. Analysis dimensions in such a structure can be based on various fields available in the log files, and may include time duration, domain, requested resource, user agent, and referrers.

This allows the analysis to be performed on portions of the log related to a specific time interval, or at a higher level of abstraction with respect to the URL path structure. The integration of e-commerce data in the data warehouse can further enhance the ability of OLAP tools to derive important business intelligence metrics [71]. The output from OLAP queries can also be used as the input for a variety of data mining or data visualization tools.

12.3.2 Cluster Analysis and Visitor Segmentation

Clustering is a data mining technique that groups together a set of items having similar characteristics. In the usage domain, there are two kinds of interesting clusters that can be discovered: user clusters and page clusters.

Clustering of user records (sessions or transactions) is one of the most commonly used analysis tasks in Web usage mining and Web analytics. Clustering of users tends to establish groups of users exhibiting similar browsing patterns. Such knowledge is especially useful for inferring user demographics in order to perform market segmentation in e-commerce applications or provide **personalized Web content** to the users with similar interests. Further analysis of user groups based on their demographic attributes (e.g., age, gender, income level, etc.) may lead to the discovery of valuable business intelligence. **Usage-based clustering** has also been used to create Web-based "user communities" reflecting similar interests of groups of users [423], and to learn user models that can be used to provide dynamic recommendations in Web personalization applications [390].

Given the mapping of user transactions into a multi-dimensional space as vectors of pageviews (see Fig. 12.8), standard clustering algorithms, such as *k*-means, can partition this space into groups of transactions that are close to each other based on a measure of distance or similarity among the vectors (see Chap. 4). Transaction clusters obtained in this way can represent user or visitor segments based on their navigational behavior or other attributes that have been captured in the transaction file. However, transaction clusters by themselves are not an effective means of capturing the aggregated view of common user patterns. Each transaction cluster may potentially contain thousands of user transactions involving hundreds of pageview references. The ultimate goal in clustering user transactions is to provide the ability to analyze each segment for deriving business intelligence, or to use them for tasks such as personalization.

One straightforward approach in creating an aggregate view of each cluster is to compute the **centroid** (or the mean vector) of each cluster. The dimension value for each pageview in the mean vector is computed by finding the ratio of the sum of the pageview weights across transactions to

the total number of transactions in the cluster. If pageview weights in the original transactions are binary, then the dimension value of a pageview p in a cluster centroid represents the percentage of transactions in the cluster in which p occurs. Thus, the centroid dimension value of p provides a measure of its significance in the cluster. Pageviews in the centroid can be sorted according to these weights and lower weight pageviews can be filtered out. The resulting set of pageview-weight pairs can be viewed as an "aggregate usage profile" representing the interests or behavior of a significant group of users.

More formally, given a transaction cluster *cl,* we can construct the aggregate profile pr_{cl} as a set of **pageview-weight** pairs by computing the centroid of *cl:*

$$pr_{cl} = \{(p, weight(p, pr_{cl})) \mid weight(p, pr_{cl}) \geq \mu\}, \tag{1}$$

where:

- the significance weight, $weight(p, pr_{cl})$, of the page p within the aggregate profile pr_{cl} is given by

$$weight(p, pr_{cl}) = \frac{1}{|cl|} \sum_{\mathbf{s} \in cl} w(p, \mathbf{s}); \tag{2}$$

- $|cl|$ is the number of transactions in cluster *cl*;
- $w(p, \mathbf{s})$ is the weight of page p in transaction vector $\mathbf{s}$ of cluster *cl*; and
- the threshold μ is used to focus only on those pages in the cluster that appear in a sufficient number of vectors in that cluster.

Each such profile, in turn, can be represented as a vector in the original n-dimensional space of pageviews. This aggregate representation can be used directly for predictive modeling and in applications such as recommender systems: given a new user, u, who has accessed a set of pages, P_u, so far, we can measure the similarity of P_u to the discovered profiles, and recommend to the user those pages in matching profiles which have not yet been accessed by the user.

As an example, consider the transaction data depicted in Fig. 12.11 (left). For simplicity we assume that feature (pageview) weights in each transaction vector are binary (in contrast to weights based on a function of pageview duration). We assume that the data has already been clustered using a standard clustering algorithm such as *k*-means, resulting in three clusters of user transactions. The table on the right of Fig. 12.11 shows the aggregate profile corresponding to cluster 1. As indicated by the pageview weights, pageviews B and F are the most significant pages characterizing the common interests of users in this segment. Pageview C, however, only appears in one transaction and might be removed given a filtering thresh-

		A	B	C	D	E	F
Cluster 0	user 1	0	0	1	1	0	0
	user 4	0	0	1	1	0	0
	user 7	0	0	1	1	0	0
Cluster 1	user 0	1	1	0	0	0	1
	user 3	1	1	0	0	0	1
	user 6	1	1	0	0	0	1
	user 9	0	1	1	0	0	1
Cluster 2	user 2	1	0	0	1	1	0
	user 5	1	0	0	1	1	0
	user 8	1	0	1	1	1	0

Aggregated Profile for Cluster 1	
Weight	Pageview
1.00	B
1.00	F
0.75	A
0.25	C

Fig. 12.11. Derivation of aggregate profiles from Web transaction clusters

old greater than 0.25. Such patterns are useful for characterizing user or customer segments. This example, for instance, indicates that the resulting user segment is clearly interested in items B and F and to a lesser degree in item A. Given a new user who shows interest in items A and B, this pattern may be used to infer that the user might belong to this segment and, therefore, we might recommend item F to that user.

Clustering of pages (or items) can be performed based on the usage data (i.e., starting from the user sessions or transaction data), or based on the content features associated with pages or items (keywords or product attributes). In the case of content-based clustering, the result may be collections of pages or products related to the same topic or category. In usage-based clustering, items that are commonly accessed or purchased together can be automatically organized into groups. It can also be used to provide permanent or dynamic HTML pages that suggest related hyperlinks to the users according to their past history of navigational or purchase activities.

A variety of stochastic methods have also been proposed recently for clustering of user transactions, and more generally for user modeling. For example, recent work in this area has shown that **mixture models** are able to capture more complex, dynamic user behavior. This is, in part, because the observation data (i.e., the user-item space) in some applications (such as large and very dynamic Web sites) may be too complex to be modeled by basic probability distributions such as a normal or a multinomial distribution. In particular, each user may exhibit different "types" of behavior corresponding to different tasks, and common behaviors may each be reflected in a different distribution within the data.

The general idea behind mixture models (such as a **mixture of Markov models**) is as follow. We assume that there exist k types of user behavior (or k user clusters) within the data, and each user session is assumed to be generated via a generative process which models the probability distributions of the observed variables and hidden variables. First, a user cluster is chosen with some probability. Then, the user session is generated from a Markov model with parameters specific to that user cluster. The probabilities of each user cluster is estimated, usually via the **EM** [127] algorithm, as well as the parameters of each mixture component. Mixture-based user models can provide a great deal of flexibility. For example, a mixture of first-order Markov models [76] not only can probabilistically cluster user sessions based on similarities in navigation behavior, but also characterize each type of user behavior using a first-order Markov model, thus capturing popular navigation paths or characteristics of each user cluster. A mixture of hidden Markov models was proposed in [580] for modeling clickstream of Web surfers. In addition to user-based clustering, this approach can also be used for automatic page classification. Incidentally, mixture models have been discussed in Sect. 3.7 in the context of naïve Bayesian classification. The EM algorithm is used in the same context in Sect. 5.1.

Mixture models tend to have their own shortcomings. From the data generation perspective, each individual observation (such as a user session) is generated from one and only one component model. The probability assignment to each component only measures the uncertainty about this assignment. This assumption limits this model's ability of capturing complex user behavior, and more seriously, may result in overfitting.

Probabilistic Latent Semantic Analysis (PLSA) provides a reasonable solution to the above problem [240]. In the context of Web user navigation, each observation (a user visiting a page) is assumed to be generated based on a set of unobserved (hidden) variables which "explain" the user-page observations. The data generation process is as follows: a user is selected with a certain probability, next conditioned on the user, a hidden variable is selected, and then the page to visit is selected conditioned on the chosen hidden variable. Since each user usually visits multiple pages, this data generation process ensures that each user is explicitly associated with multiple hidden variables, thus reducing the overfitting problems associated with the above mixture models. The PLSA model also uses the EM algorithm to estimate the parameters which probabilistically characterize the hidden variables underlying the co-occurrence observation data, and measure the relationship among hidden and observed variables.

This approach provides a great deal of flexibility since it provides a single framework for quantifying the relationships between users, between items, between users and items, and between users or items and hidden

variables that "explain" the observed relationships [254]. Given a set of n user profiles (or transaction vectors), $UP = \{u_1, u_2, \ldots, u_n\}$, and a set of m items (e.g., pages or products), $I = \{i_1, i_2, \ldots, i_m\}$, the PLSA model associates a set of unobserved factor variables $Z = \{z_1, z_2, \ldots, z_q\}$ with observations in the data (q is specified by the user). Each observation corresponds to a weight $w_{u_k}(i_j)$ for an item i_j in the user profile for a user u_k. This weight may, for example, correspond to the significance of the page in the user transaction or the user rating associated with the item. For a given user u and a given item i, the following joint probability can be derived (see [254] for details of the derivation):

$$\Pr(u,i) = \sum_{k=1}^{q} \Pr(z_k) \Pr(u \mid z_k) \Pr(i \mid z_k). \tag{3}$$

In order to explain the observations in (UP, I), we need to estimate the parameters $\Pr(z_k)$, $\Pr(u|z_k)$, and $\Pr(i|z_k)$, while maximizing the following likelihood $L(UP, I)$ of the observation data:

$$L(UP,I) = \sum_{u \in UP} \sum_{i \in I} w_u(i) \log \Pr(u,i). \tag{4}$$

The Expectation–Maximization (EM) algorithm is used to perform maximum likelihood parameter estimation. Based on initial values of $\Pr(z_k)$, $\Pr(u|z_k)$, and $\Pr(i|z_k)$, the algorithm alternates between an expectation step and maximization step. In the expectation step, posterior probabilities are computed for latent variables based on current estimates, and in the maximization step the re-estimated parameters are obtained. Iterating the expectation and maximization steps monotonically increases the total likelihood of the observed data $L(UP, I)$, until a local optimal solution is reached. Details of this approach can be found in [254].

Again, one of the main advantages of PLSA model in Web usage mining is that using probabilistic inference with the above estimated parameters, we can derive relationships among users, among pages, and between users and pages. Thus this framework provides a flexible approach to model a variety of types of usage patterns.

12.3.3 Association and Correlation Analysis

Association rule discovery and statistical correlation analysis can find groups of items or pages that are commonly accessed or purchased together. This, in turn, enables Web sites to organize the site content more efficiently, or to provide effective cross-sale product recommendations.

Most common approaches to association discovery are based on the Apriori algorithm (see Sect. 2.2). This algorithm finds groups of items (pageviews appearing in the preprocessed log) occurring frequently together in many transactions (i.e., satisfying a user specified minimum support threshold). Such groups of items are referred to as **frequent itemsets**. Association rules which satisfy a minimum confidence threshold are then generated from the frequent itemsets.

Recall an association rule is an expression of the form $X \rightarrow Y$ [*sup, conf*], where X and Y are itemsets, *sup* is the support of the itemset $X \cup Y$ representing the probability that X and Y occur together in a transaction, and *conf* is the confidence of the rule, defined by $sup(X \cup Y) / sup(X)$, representing the conditional probability that Y occurs in a transaction given that X has occurred in that transaction. More details on association rule discovery can be found in Chap. 2.

The mining of association rules in Web transaction data has many advantages. For example, a high-confidence rule such as

$$\text{special-offers/, /products/software/} \rightarrow \text{shopping-cart/}$$

might provide some indication that a promotional campaign on software products is positively affecting online sales. Such rules can also be used to optimize the structure of the site. For example, if a site does not provide direct linkage between two pages A and B, the discovery of a rule, $\mathsf{A} \rightarrow \mathsf{B}$, would indicates that providing a direct hyperlink from A to B might aid users in finding the intended information. Both association analysis (among products or pageviews) and statistical correlation analysis (generally among customers or visitors) have been used successfully in Web personalization and recommender systems [236, 389].

Indeed, one of the primary applications of association rule mining in Web usage or e-commerce data is in recommendation. For example, in the collaborative filtering context, Sarwar et al. [474] used association rules in the context of a *top-N* recommender system for e-commerce. The preferences of the target user are matched against the items in the antecedent X of each rule, and the items on the right hand side of the matching rules are sorted according to the confidence values. Then the top N ranked items from this list are recommended to the target user (see Sect. 3.5.3).

One problem for association rule recommendation systems is that a system cannot give any recommendations when the dataset is sparse (which is often the case in Web usage mining and collaborative filtering applications). The reason for this sparsity is that any given user visits (or rates) only a very small fraction of the available items, and thus it is often difficult to find a sufficient number of common items in multiple user profiles. Sarwar et al. [474] relied on some standard dimensionality reduction tech-

niques to alleviate this problem. One deficiency of this and other dimensionality reduction approaches is that some of the useful or interesting items may be removed, and therefore, may not appear in the final patterns. Fu et al. [187] proposed two potential solutions to this problem. The first solution is to rank all the discovered rules based on the degree of intersection between the left-hand side of each rule and the user's active session and then to generate the top k recommendations. This approach will relax the constraint of having to obtain a complete match with the left-hand-side of the rules. The second solution is to utilize **collaborative filtering**: the system finds "close neighbors" who have similar interest to a target user and makes recommendations based on the close neighbors' histories.

Lin et al. [337] proposed a **collaborative recommendation** system using association rules. The proposed mining algorithm finds an appropriate number of rules for each target user by automatically selecting the minimum support. The system generates association rules among users (user associations), as well as among items (item associations). If a user minimum support is greater than a threshold, the system generates recommendations based on user associations, else it uses item associations.

Because it is difficult to find matching rule antecedent with a full user profile (e.g., a full user session or transaction), association-based recommendation algorithms typically use a sliding window w over the target user's active profile or session. The window represents the portion of user's history that will be used to predict future user actions (based on matches with the left-hand sides of the discovered rules). The size of this window is iteratively decreased until an exact match with the antecedent of a rule is found. A problem with the naive approach to this algorithm is that it requires repeated search through the rule-base. However, efficient trie-based data structure can be used to store the discovered itemsets and allow for efficient generation of recommendations without the need to generate all association rules from frequent itemsets [389]. Such data structures are commonly used for string or sequence searching applications. In the context of association rule mining, the frequent itemsets are stored in a directed acyclic graph. This **frequent itemset graph** is an extension of the lexicographic tree used in the tree projection mining algorithm of Agarwal, et al. [2]. The graph is organized into levels from 0 to k, where k is the maximum size among all frequent itemsets. Each node at depth d in the graph corresponds to an itemset, X, of size d and is linked to itemsets of size d+1 that contain X at level d+1. The single root node at level 0 corresponds to the empty itemset. To be able to search for different orderings of an itemset, all itemsets are sorted in lexicographic order before being inserted into the graph. If the graph is used to recommend items to a new

target user, that user's active session is also sorted in the same manner before matching with itemsets.

As an example, suppose that in a hypothetical Web site with user transaction data depicted in the left table of Fig. 12.12. Using a minimum support (minsup) threshold of 4 (i.e., 80%), the Apriori algorithm discovers the frequent itemsets given in the right table. For each itemset, the support is also given. The corresponding frequent itemset graph is depicted in Fig. 12.13.

A recommendation engine based on this framework matches the current user session window with the previously discovered frequent itemsets to find candidate items (pages) for recommendation. Given an active session window w and a group of frequent itemsets, the algorithm considers all the frequent itemsets of size $|w| + 1$ containing the current session window by performing a depth-first search of the Frequent Itemset Graph to level $|w|$. The recommendation value of each candidate is based on the confidence of the corresponding association rule whose consequent is the singleton containing the page to be recommended. If a match is found, then the children of the matching node n containing w are used to generate candidate recommendations. In practice, the window w can be incrementally decreased until a match is found with and itemset. For example, given user active session window $<B, E>$, the recommendation generation algorithm, using the graph of Fig. 12.13, finds items A and C as candidate recommendations. The recommendation scores of item A and C are 1 and 4/5, corresponding to the confidences of the rules, $B, E \rightarrow A$ and $B, E \rightarrow C$, respectively.

A problem with using a single global minimum support threshold in association rule mining is that the discovered patterns will not include "rare" but important items which may not occur frequently in the transaction data. This is particularly important when dealing with Web usage data, it is often the case that references to deeper content or product-oriented pages oc-

Transactions
A, B, D, E
A, B, E, C, D
A, B, E, C
B, E, B, A, C
D, A, B, E, C

Size 1		Size 2		Size 3		Size 4	
Itemset	Supp.	Itemset	Supp.	Itemset	Supp.	Itemset	Supp.
A	5	A,B	5	A,B,C	4	A,B,C,E	4
B	5	A,C	4	A,B,E	5		
C	4	A,E	5	A,C,E	4		
E	5	B,C	4	B,C,E	4		
		B,E	5				
		C,E	4				

Fig. 12.12. Web transactions and resulting frequent itemsets (minsup = 4)

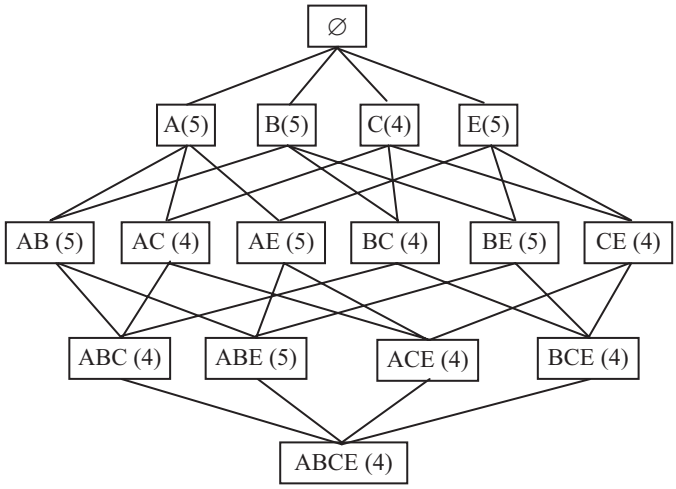


Fig. 12.13. A frequent itemset graph.

cur far less frequently than those of top level navigation-oriented pages. Yet, for effective Web personalization, it is important to capture patterns and generate recommendations that contain these items. A mining method based on **multiple minimum supports** is proposed in [344] that allows users to specify different support values for different items. In this method, the support of an itemset is defined as the minimum support of all items contained in the itemset. For more details on mining using multiple minimum supports, see Sect. 2.4. The specification of multiple minimum supports thus allows frequent itemsets to potentially contain rare items which are deemed important. It has been shown that the use of multiple support association rules in the context of Web personalization can dramatically increase the coverage (or recall) of recommendations while maintaining a reasonable precision [389].

12.3.4 Analysis of Sequential and Navigational Patterns

The technique of sequential pattern mining attempts to find inter-session patterns such that the presence of a set of items is followed by another item in a time-ordered set of sessions or episodes. By using this approach, Web marketers can predict future visit patterns which will be helpful in placing advertisements aimed at certain user groups. Other types of temporal analysis that can be performed on sequential patterns include trend analysis, change point detection, or similarity analysis. In the context of Web

usage data, **sequential pattern mining** can be used to capture frequent navigational paths among user trails.

Sequential patterns (SPs) in Web usage data capture the Web page trails that are often visited by users, in the order that they were visited. Sequential patterns are those sequences of items that frequently occur in a sufficiently large proportion of (sequence) transactions. A sequence $\langle s_1 s_2 ... s_n \rangle$ occurs in a transaction $t = \langle p_1, p_2, \ldots, p_m \rangle$ (where $n \leq m$) if there exist n positive integers $1 \leq a_1 < a_2 < \ldots < a_n \leq m$, and $s_i = p_{a_i}$ for all i. We say that $\langle cs_1\ cs_2 ... cs_n \rangle$ is a **contiguous sequence** in t if there exists an integer $0 \leq b \leq m - n$, and $cs_i = p_{b+i}$ for all $i = 1$ to n. In a **contiguous sequential pattern** (CSP), each pair of adjacent items, s_i and s_{i+1}, must appear consecutively in a transaction t which supports the pattern. A normal sequential pattern can represent non-contiguous frequent sequences in the underlying set of sequence transactions.

Given a sequence transaction set T, the support (denoted by $sup(S)$) of a sequential (respectively, contiguous sequential) pattern S in T is the fraction of transactions in T that contain S. The confidence of the rule $X \rightarrow Y$, where X and Y are (contiguous) sequential patterns, is defined as:

$$conf(X \rightarrow Y) = sup(X \circ Y) / sup(X) ,$$

where $\circ$ denotes the concatenation operator.

In the context of Web usage data, CSPs can be used to capture frequent navigational paths among user trails [497]. In contrast, items appearing in SPs, while preserving the underlying ordering, need not be adjacent, and thus they represent more general navigational patterns within the site. Note that sequences and sequential patterns or rules discussed here are special cases of those defined in Sect. 2.9.

The view of Web transactions as sequences of pageviews allows for a number of useful and well-studied models to be used in discovering or analyzing user navigation patterns. One such approach is to model the navigational activities in the Web site as a Markov model: each pageview (or a category) can be represented as a state and the transition probability between two states can represent the likelihood that a user will navigate from one state to the other. This representation allows for the computation of a number of useful user or site metrics. For example, one might compute the probability that a user will make a purchase, given that she has performed a search in an online catalog. **Markov models** have been proposed as the underlying modeling machinery for link prediction as well as for Web prefetching to minimize system latencies [132, 473]. The goal of such approaches is to predict the *next* user action based on a user's previous surfing behavior. They have also been used to discover high probability user navigational trails in a Web site [57]. More sophisticated statistical learn-

ing techniques, such as mixtures of Markov models, have also been used to cluster navigational sequences and perform exploratory analysis of users' navigational behavior in a site [76].

More formally, a Markov model is characterized by a set of states $\{s_1, s_2, \ldots, s_n\}$ and a transition probability matrix, $[Pr_{i,j}]_{n \times n}$, where $Pr_{i,j}$ represents the probability of a transition from state s_i to state s_j. Markov models are especially suited for predictive modeling based on contiguous sequences of events. Each state represents a contiguous subsequence of prior events. The order of the Markov model corresponds to the number of prior events used in predicting a future event. So, a kth-order Markov model predicts the probability of next event by looking the past k events. Given a set of all paths R, the probability of reaching a state s_j from a state s_i via a (non-cyclic) path $r \in R$ is the product of all the transition probabilities along the path and is given by $Pr(r) = \prod Pr_{m,m+1}$, where m ranges from i to $j - 1$. The probability of reaching s_j from s_i is the sum of these path probabilities over all paths: $Pr(j|i) = \sum_{r \in R} Pr(r)$.

As an example of how Web transactions can be modeled as a Markov model, consider the set of Web transaction given in Fig. 12.14 (left). The Web transactions involve pageviews A, B, C, D, and E. For each transaction the frequency of occurrences of that transaction in the data is given in the table's second column (thus there are a total of 50 transactions in the data set). The (absorbing) Markov model for this data is also given in Fig. 12.14 (right). The transitions from the "start" state represent the prior probabilities for transactions starting with pageviews A and B. The transitions into the "final" state represent the probabilities that the paths end with the specified originating pageviews. For example, the transition probability from the state A to B is 16/28 = 0.57 since out of the 28 occurrences of A in transactions, in 16 cases, B occurs immediately after A.

Higher-order Markov models generally provide a higher prediction accuracy. However, this is usually at the cost of lower coverage (or recall) and much higher model complexity due to the larger number of states. In order to remedy the coverage and space complexity problems, Pitkow and Pirolli [446] proposed all-kth-order Markov models (for coverage improvement) and a new state reduction technique, called **longest repeating subsequences** (LRS) (for reducing model size). The use of all-kth-order Markov models generally requires the generation of separate models for each of the k orders: if the model cannot make a prediction using the kth order, it will attempt to make a prediction by incrementally decreasing the model order. This scheme can easily lead to even higher space complexity since it requires the representation of all possible states for each k. Deshpande and Karypis [132] proposed selective Markov models, introducing several schemes in order to tackle the model complexity problems

Transaction	Frequency
A, B, E	10
B, D, B, C	4
B, C, E	10
A, B, E, F	6
A, D, B	12
B, D, B, E	8

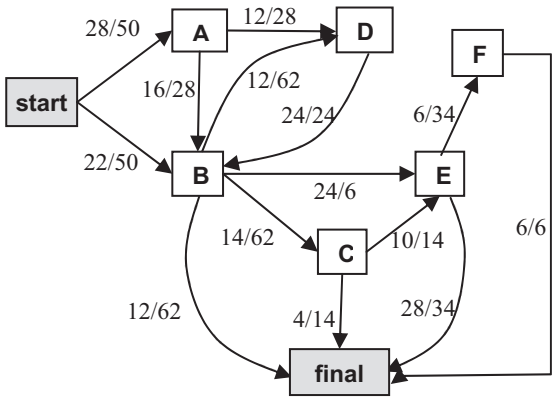


Fig. 12.14. An example of modeling navigational trails as a Markov

with all-*k*th-order Markov models. The proposed schemes involve pruning the model based on criteria such as support, confidence, and error rate. In particular, the support-pruned Markov models eliminate all states with low support determined by a minimum frequency threshold.

Another way of efficiently representing contiguous navigational trails is by inserting each trail into a *trie* structure. A good example of this approach is the notion of aggregate tree introduced as part of the WUM (Web Utilization Miner) system [497]. The aggregation service of WUM extracts the transactions from a collection of Web logs, transforms them into sequences, and merges those sequences with the same prefix into the aggregate tree (a trie structure). Each node in the tree represents a navigational subsequence from the root (an empty node) to a page and is annotated by the frequency of occurrences of that subsequence in the transaction data (and possibly other information such as markers to distinguish among repeat occurrences of the corresponding page in the subsequence). WUM uses a mining query language, called MINT, to discover generalized navigational patterns from this trie structure. MINT includes mechanisms to specify sophisticated constraints on pattern templates, such as wildcards with user-specified boundaries, as well as other statistical thresholds such as support and confidence. This approach and its extensions have proved useful in evaluating the navigational design of a Web site [496].

As an example, again consider the set of Web transactions given in the previous example. Figure 12.15 shows a simplified version of WUM's aggregate tree structure derived from these transactions. Each node in the tree represents a navigational subsequence from the root (an empty node) to a page and is annotated by the frequency of occurrences of that subsequence in the session data. The advantage of this approach is that the search for

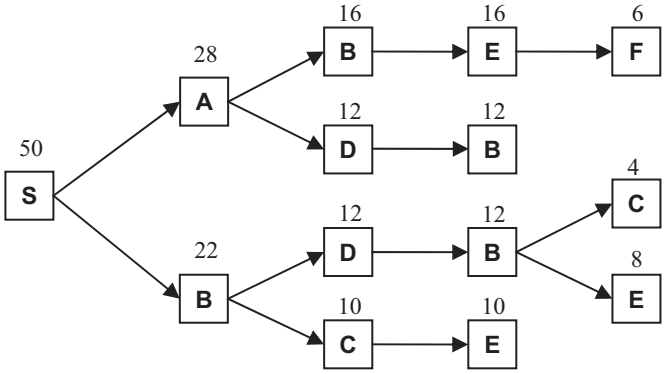


Fig. 12.15. An example of modeling navigational trails in an aggregate tree

navigational patterns can be performed very efficiently and the confidence and support for the navigational patterns can be readily obtained from the node annotations in the tree. For example, consider the contiguous navigational sequence <A, B, E, F>. The support for this sequence can be computed as the support of the last page in the sequence, *F*, divided by the support of the root node: 6/50 = 0.12, and the confidence of the sequence is the support of *F* divided by the support of its predecessor, E, or 6/16 = 0.375. If there are multiple branches in the tree containing the same navigational sequence, then the support for the sequence is the sum of the supports for all occurrences of the sequence in the tree and the confidence is updated accordingly. For example, the support of the sequence <D, B> is (12+12)/50 = 0.48, while the confidence is the aggregate support for B divided by the aggregate support for D, i.e., 24/24 = 1.0. The disadvantage of this approach is the possibly high space complexity, especially in a site with many dynamically generated pages.

12.3.5 Classification and Prediction based on Web User Transactions

Classification is the task of mapping a data item into one of several predefined classes. In the Web domain, one is interested in developing a profile of users belonging to a particular class or category. This requires extraction and selection of features that best describe the properties of given the class or category. Classification can be done by using supervised learning algorithms such as decision trees, naive Bayesian classifiers, *k*-nearest neighbor classifiers, and Support Vector Machines (Chap. 3). It is also

possible to use previously discovered clusters and association rules for classification of new users (Sect. 3.5).

Classification techniques play an important role in Web analytics applications for modeling the users according to various predefined metrics. For example, given a set of user transactions, the sum of purchases made by each user within a specified period of time can be computed. A classification model can then be built based on this enriched data in order to classify users into those who have a high propensity to buy and those who do not, taking into account features such as users' demographic attributes, as well their navigational activities.

Another important application of classification and prediction in the Web domain is that of **collaborative filtering**. Most collaborative filtering applications in existing recommender systems use *k*-nearest neighbor classifiers to predict user ratings or purchase propensity by measuring the correlations between a current (target) user's profile (which may be a set of item ratings or a set of items visited or purchased) and past user profiles in order to find users in the database with similar characteristics or preferences [236]. Many of the Web usage mining approaches discussed earlier can also be used to automatically discover user models and then apply these models to provide personalized content to an active user [386, 445].

Basically, collaborative filtering based on the *k*-nearest neighbor (*k*NN) approach involves comparing the activity record for a target user with the historical records T of other users in order to find the top k users who have similar tastes or interests. The mapping of a visitor record to its neighborhood could be based on similarity in ratings of items, access to similar content or pages, or purchase of similar items. In most typical collaborative filtering applications, the user records or profiles are a set of ratings for a subset of items. The identified neighborhood is then used to recommend items not already accessed or purchased by the active user. Thus, there are two primary phases in collaborative filtering: the neighborhood formation phase and the recommendation phase. In the context of Web usage mining, *k*NN involves measuring the similarity or correlation between the target user's active session $\mathbf{u}$ (represented as a vector) and each past transaction vector $\mathbf{v}$ (where $\mathbf{v} \in T$). The top k most similar transactions to $\mathbf{u}$ are considered to be the neighborhood for the session $\mathbf{u}$. More specifically, the similarity between the target user, $\mathbf{u}$, and a neighbor, $\mathbf{v}$, can be calculated by the **Pearson's correlation coefficient** defined below:

$$sim(\mathbf{u},\mathbf{v}) = \frac{\sum_{i\in C}(r_{\mathbf{u},i} - \bar{r}_{\mathbf{u}})(r_{\mathbf{v},i} - \bar{r}_{\mathbf{v}})}{\sqrt{\sum_{i\in C}(r_{\mathbf{u},i} - \bar{r}_{\mathbf{u}})^2}\sqrt{\sum_{i\in C}(r_{\mathbf{v},i} - \bar{r}_{\mathbf{v}})^2}}, \tag{5}$$

where C is the set of items that are co-rated by $\mathbf{u}$ and $\mathbf{v}$ (i.e., items that

have been rated by both of them), $r_{\mathbf{u},i}$ and $r_{\mathbf{v},i}$ are the ratings (or weights) of some item i for the target user **u** and a possible neighbor **v** respectively, and $\bar{r}_{\mathbf{u}}$ and $\bar{r}_{\mathbf{v}}$ are the average ratings (or weights) of **u** and **v** respectively. Once similarities are calculated, the most similar users are selected.

It is also common to filter out neighbors with a similarity of less than a specific threshold to prevent predictions being based on very distant or negative correlations. Once the most similar user transactions are identified, the following formula can be used to compute the rating prediction of an item i for target user **u**.

$$p(\mathbf{u},i) = \bar{r}_{\mathbf{u}} + \frac{\sum_{\mathbf{v}\in V} sim(\mathbf{u},\mathbf{v}) \times (r_{\mathbf{v},i} - \bar{r}_{\mathbf{v}})}{\sum_{\mathbf{v}\in V} |sim(\mathbf{u},\mathbf{v})|}, \tag{6}$$

where V is the set of k similar users, $r_{\mathbf{v},i}$ are the ratings of those users on item i, and $sim(\mathbf{u}, \mathbf{v})$ is the Pearson correlation described above. The formula in essence computes the degree of preference of all the neighbors weighted by their similarity and then adds this to the target user's average rating, the idea being that different users may have different "baselines" around which their ratings are distributed.

The problem with the user-based formulation of the collaborative filtering problem is the lack of scalability: it requires the real-time comparison of the target user to all user records in order to generate predictions. A variation of this approach that remedies this problem is called **item-based collaborative filtering** [475]. Item-based collaborative filtering works by comparing items based on their pattern of ratings across users. Again, a nearest-neighbor approach can be used. The *k*NN algorithm attempts to find k similar items that are co-rated by different users similarly. The similarity measure typically used is the **adjusted cosine similarity** given below:

$$sim(i,j) = \frac{\sum_{\mathbf{u}\in U} (r_{\mathbf{u},i} - \bar{r}_{\mathbf{u}})(r_{\mathbf{u},j} - \bar{r}_{\mathbf{u}})}{\sqrt{\sum_{\mathbf{u}\in U} (r_{\mathbf{u},i} - \bar{r}_{\mathbf{u}})^2} \sqrt{\sum_{\mathbf{u}\in U} (r_{\mathbf{u},j} - \bar{r}_{\mathbf{u}})^2}}, \tag{7}$$

where U is the set of all users, i and j are items, $r_{\mathbf{u},i}$ represents the rating of user $\mathbf{u} \in U$ on item i, and $\bar{r}_{\mathbf{u}}$ is the average of the user **u**'s ratings as before. Note that in this case, we are computing the pair-wise similarities among items (not users) based on the ratings for these items across all users. After computing the similarity between items we select a set of k most similar items to the target item (i.e., the item for which we are interested in predicting a rating value) and generate a predicted value of user **u**'s rating by using the following formula

$$p(\mathbf{u}, i) = \frac{\sum_{j \in J} r_{\mathbf{u},j} \times sim(i, j)}{\sum_{j \in J} sim(i, j)}, \tag{8}$$

where J is the set of k similar items, $r_{\mathbf{u},j}$ is the rating of user $\mathbf{u}$ on item j, and $sim(i, j)$ is the similarity between items i and j as defined above. It is also common to ignore items with negative similarity to the target item. The idea here is to use the user's own ratings for the similar items to extrapolate the prediction for the target item.

12.4 Discussion and Outlook

Web usage mining has emerged as the essential tool for realizing more personalized, user-friendly and business-optimal Web services. Advances in data pre-processing, modeling, and mining techniques, applied to the Web data, have already resulted in many successful applications in adaptive information systems, personalization services, Web analytics tools, and content management systems. As the complexity of Web applications and user's interaction with these applications increases, the need for intelligent analysis of the Web usage data will also continue to grow.

Usage patterns discovered through Web usage mining are effective in capturing item-to-item and user-to-user relationships and similarities at the level of user sessions. However, without the benefit of deeper domain knowledge, such patterns provide little insight into the underlying reasons for which such items or users are grouped together. Furthermore, the inherent and increasing heterogeneity of the Web has required Web-based applications to more effectively integrate a variety of types of data across multiple channels and from different sources.

Thus, a focus on techniques and architectures for more effective integration and mining of content, usage, and structure data from different sources is likely to lead to the next generation of more useful and more intelligent applications, and more sophisticated tools for Web usage mining that can derive intelligence from user transactions on the Web.

Bibliographic Notes

Web usage mining as a complete process, integrating various stages of data mining cycle, including data preparation, pattern discovery, and interpreta-