perform superset checking. This is because if a frequent itemset $X \cup Y$ is found later than itemset X, and carries the same support as X, it must be in X's projected database and must have been generated during itemset merging.

To assist in subset checking, a compressed **pattern-tree** can be constructed to maintain the set of closed itemsets mined so far. The pattern-tree is similar in structure to the FP-tree except that all the closed itemsets found are stored explicitly in the corresponding tree branches. For efficient subset checking, we can use the following property: *If the current itemset* S_c *can be subsumed by another already found closed itemset* S_a*, then (1)* S_c *and* S_a *have the same support, (2) the length of* S_c *is smaller than that of* S_a*, and (3) all of the items in* S_c *are contained in* S_a.

Based on this property, a **two-level hash index structure** can be built for fast accessing of the pattern-tree: The first level uses the identifier of the last item in S_c as a hash key (since this identifier must be within the branch of S_c), and the second level uses the support of S_c as a hash key (since S_c and S_a have the same support). This will substantially speed up the subset checking process.

This discussion illustrates methods for efficient mining of closed frequent itemsets. *"Can we extend these methods for efficient mining of maximal frequent itemsets?"* Because maximal frequent itemsets share many similarities with closed frequent itemsets, many of the optimization techniques developed here can be extended to mining maximal frequent itemsets. However, we leave this method as an exercise for interested readers.

6.3 Which Patterns Are Interesting?—Pattern Evaluation Methods

Most association rule mining algorithms employ a support–confidence framework. Although minimum support and confidence thresholds *help* weed out or exclude the exploration of a good number of uninteresting rules, many of the rules generated are still not interesting to the users. Unfortunately, this is especially true *when mining at low support thresholds or mining for long patterns.* This has been a major bottleneck for successful application of association rule mining.

In this section, we first look at how even strong association rules can be uninteresting and misleading (Section 6.3.1). We then discuss how the support–confidence framework can be supplemented with additional interestingness measures based on *correlation analysis* (Section 6.3.2). Section 6.3.3 presents additional pattern evaluation measures. It then provides an overall comparison of all the measures discussed here. By the end, you will learn which pattern evaluation measures are most effective for the discovery of only interesting rules.

6.3.1 Strong Rules Are Not Necessarily Interesting

Whether or not a rule is interesting can be assessed either subjectively or objectively. Ultimately, only the user can judge if a given rule is interesting, and this judgment, being

subjective, may differ from one user to another. However, objective interestingness measures, based on the statistics "behind" the data, can be used as one step toward the goal of weeding out uninteresting rules that would otherwise be presented to the user.

"*How can we tell which strong association rules are really interesting?*" Let's examine the following example.

Example 6.7 A misleading "strong" association rule. Suppose we are interested in analyzing transactions at *AllElectronics* with respect to the purchase of computer games and videos. Let *game* refer to the transactions containing computer games, and *video* refer to those containing videos. Of the 10,000 transactions analyzed, the data show that 6000 of the customer transactions included computer games, while 7500 included videos, and 4000 included both computer games and videos. Suppose that a data mining program for discovering association rules is run on the data, using a minimum support of, say, 30% and a minimum confidence of 60%. The following association rule is discovered:

$$buys(X, \text{"}computer\ games\text{"}) \Rightarrow buys(X, \text{"}videos\text{"})$$
$$[support = 40\%, confidence = 66\%]. \tag{6.6}$$

Rule (6.6) is a strong association rule and would therefore be reported, since its support value of $\frac{4000}{10{,}000} = 40\%$ and confidence value of $\frac{4000}{6000} = 66\%$ satisfy the minimum support and minimum confidence thresholds, respectively. However, Rule (6.6) is misleading because the probability of purchasing videos is 75%, which is even larger than 66%. In fact, computer games and videos are negatively associated because the purchase of one of these items actually decreases the likelihood of purchasing the other. Without fully understanding this phenomenon, we could easily make unwise business decisions based on Rule (6.6). ■

Example 6.7 also illustrates that the confidence of a rule $A \Rightarrow B$ can be deceiving. It does not measure the *real strength* (or lack of strength) of the *correlation* and *implication* between A and B. Hence, alternatives to the support–confidence framework can be useful in mining interesting data relationships.

6.3.2 From Association Analysis to Correlation Analysis

As we have seen so far, the support and confidence measures are insufficient at filtering out uninteresting association rules. To tackle this weakness, a correlation measure can be used to augment the support–confidence framework for association rules. This leads to *correlation rules* of the form

$$A \Rightarrow B\ [support, confidence, correlation]. \tag{6.7}$$

That is, a correlation rule is measured not only by its support and confidence but also by the correlation between itemsets A and B. There are many different correlation measures from which to choose. In this subsection, we study several correlation measures to determine which would be good for mining large data sets.

Lift is a simple correlation measure that is given as follows. The occurrence of itemset A is **independent** of the occurrence of itemset B if $P(A \cup B) = P(A)P(B)$; otherwise, itemsets A and B are **dependent** and **correlated** as events. This definition can easily be extended to more than two itemsets. The **lift** between the occurrence of A and B can be measured by computing

$$lift(A, B) = \frac{P(A \cup B)}{P(A)P(B)}. \tag{6.8}$$

If the resulting value of Eq. (6.8) is less than 1, then the occurrence of A is *negatively correlated with* the occurrence of B, meaning that the occurrence of one likely leads to the absence of the other one. If the resulting value is greater than 1, then A and B are *positively correlated*, meaning that the occurrence of one implies the occurrence of the other. If the resulting value is equal to 1, then A and B are *independent* and there is no correlation between them.

Equation (6.8) is equivalent to $P(B|A)/P(B)$, or $conf(A \Rightarrow B)/sup(B)$, which is also referred to as the *lift* of the association (or correlation) rule $A \Rightarrow B$. In other words, it assesses the degree to which the occurrence of one "lifts" the occurrence of the other. For example, if A corresponds to the sale of computer games and B corresponds to the sale of videos, then given the current market conditions, the sale of games is said to increase or "lift" the likelihood of the sale of videos by a factor of the value returned by Eq. (6.8).

Let's go back to the computer game and video data of Example 6.7.

Example 6.8 Correlation analysis using lift. To help filter out misleading "strong" associations of the form $A \Rightarrow B$ from the data of Example 6.7, we need to study how the two itemsets, A and B, are correlated. Let $\overline{game}$ refer to the transactions of Example 6.7 that do not contain computer games, and $\overline{video}$ refer to those that do not contain videos. The transactions can be summarized in a *contingency table*, as shown in Table 6.6.

From the table, we can see that the probability of purchasing a computer game is $P(\{game\}) = 0.60$, the probability of purchasing a video is $P(\{video\}) = 0.75$, and the probability of purchasing both is $P(\{game, video\}) = 0.40$. By Eq. (6.8), the lift of Rule (6.6) is $P(\{game, video\})/(P(\{game\}) \times P(\{video\})) = 0.40/(0.60 \times 0.75) = 0.89$. Because this value is less than 1, there is a negative correlation between the occurrence of $\{game\}$ and $\{video\}$. The numerator is the likelihood of a customer purchasing both, while the denominator is what the likelihood would have been if the two purchases were completely independent. Such a negative correlation cannot be identified by a support–confidence framework. ■

The second correlation measure that we study is the χ^2 measure, which was introduced in Chapter 3 (Eq. 3.1). To compute the χ^2 value, we take the squared difference between the observed and expected value for a slot (A and B pair) in the contingency table, divided by the expected value. This amount is summed for all slots of the contingency table. Let's perform a χ^2 analysis of Example 6.8.

Table 6.6 2×2 Contingency Table Summarizing the Transactions with Respect to Game and Video Purchases

	game	$\overline{game}$	Σ_{row}
video	4000	3500	7500
$\overline{video}$	2000	500	2500
Σ_{col}	6000	4000	10,000

Table 6.7 Table 6.6 Contingency Table, Now with the Expected Values

	game	$\overline{game}$	Σ_{row}
video	4000 (4500)	3500 (3000)	7500
$\overline{video}$	2000 (1500)	500 (1000)	2500
Σ_{col}	6000	4000	10,000

Example 6.9 **Correlation analysis using χ^2.** To compute the correlation using χ^2 analysis for nominal data, we need the observed value and expected value (displayed in parenthesis) for each slot of the contingency table, as shown in Table 6.7. From the table, we can compute the χ^2 value as follows:

$$\chi^2 = \Sigma \frac{(observed - expected)^2}{expected} = \frac{(4000 - 4500)^2}{4500} + \frac{(3500 - 3000)^2}{3000}$$
$$+ \frac{(2000 - 1500)^2}{1500} + \frac{(500 - 1000)^2}{1000} = 555.6.$$

Because the χ^2 value is greater than 1, and the observed value of the slot (*game*, *video*) = 4000, which is less than the expected value of 4500, *buying game* and *buying video* are *negatively correlated*. This is consistent with the conclusion derived from the analysis of the *lift* measure in Example 6.8. ■

6.3.3 A Comparison of Pattern Evaluation Measures

The above discussion shows that instead of using the simple support–confidence framework to evaluate frequent patterns, other measures, such as *lift* and χ^2, often disclose more intrinsic pattern relationships. How effective are these measures? Should we also consider other alternatives?

Researchers have studied many pattern evaluation measures even before the start of in-depth research on scalable methods for mining frequent patterns. Recently, several other pattern evaluation measures have attracted interest. In this subsection, we present

four such measures: *all_confidence*, *max_confidence*, *Kulczynski*, and *cosine*. We'll then compare their effectiveness with respect to one another and with respect to the *lift* and χ^2 measures.

Given two itemsets, A and B, the **all_confidence** measure of A and B is defined as

$$all\_conf(A,B) = \frac{sup(A \cup B)}{max\{sup(A), sup(B)\}} = min\{P(A|B), P(B|A)\}, \tag{6.9}$$

where $max\{sup(A), sup(B)\}$ is the maximum support of the itemsets A and B. Thus, $all\_conf(A,B)$ is also the minimum confidence of the two association rules related to A and B, namely, "$A \Rightarrow B$" and "$B \Rightarrow A$."

Given two itemsets, A and B, the **max_confidence** measure of A and B is defined as

$$max\_conf(A, B) = max\{P(A \mid B), P(B \mid A)\}. \tag{6.10}$$

The *max_conf* measure is the maximum confidence of the two association rules, "$A \Rightarrow B$" and "$B \Rightarrow A$."

Given two itemsets, A and B, the **Kulczynski** measure of A and B (abbreviated as **Kulc**) is defined as

$$Kulc(A, B) = \frac{1}{2}(P(A|B) + P(B|A)). \tag{6.11}$$

It was proposed in 1927 by Polish mathematician S. Kulczynski. It can be viewed as an average of two confidence measures. That is, it is the average of two conditional probabilities: the probability of itemset B given itemset A, and the probability of itemset A given itemset B.

Finally, given two itemsets, A and B, the **cosine** measure of A and B is defined as

$$\begin{aligned} cosine(A, B) &= \frac{P(A \cup B)}{\sqrt{P(A) \times P(B)}} = \frac{sup(A \cup B)}{\sqrt{sup(A) \times sup(B)}} \\ &= \sqrt{P(A|B) \times P(B|A)}. \end{aligned} \tag{6.12}$$

The *cosine* measure can be viewed as a *harmonized lift* measure: The two formulae are similar except that for cosine, the *square root* is taken on the product of the probabilities of A and B. This is an important difference, however, because by taking the square root, the cosine value is only influenced by the supports of A, B, and $A \cup B$, and not by the total number of transactions.

Each of these four measures defined has the following property: Its value is only influenced by the supports of A, B, and $A \cup B$, or more exactly, by the conditional probabilities of $P(A|B)$ and $P(B|A)$, but not by the total number of transactions. Another common property is that each measure ranges from 0 to 1, and the higher the value, the closer the relationship between A and B.

Now, together with *lift* and χ^2, we have introduced in total six pattern evaluation measures. You may wonder, "*Which is the best in assessing the discovered pattern relationships?*" To answer this question, we examine their performance on some typical data sets.

Table 6.8 2×2 Contingency Table for Two Items

	milk	$\overline{milk}$	Σ_{row}
coffee	mc	$\overline{m}c$	c
$\overline{coffee}$	$m\bar{c}$	$\overline{mc}$	$\bar{c}$
Σ_{col}	m	$\overline{m}$	Σ

Table 6.9 Comparison of Six Pattern Evaluation Measures Using Contingency Tables for a Variety of Data Sets

Data Set	*mc*	$\overline{m}c$	$m\bar{c}$	$\overline{mc}$	χ^2	*lift*	*all_conf.*	*max_conf.*	*Kulc.*	*cosine*
D_1	10,000	1000	1000	100,000	90557	9.26	0.91	0.91	0.91	0.91
D_2	10,000	1000	1000	100	0	1	0.91	0.91	0.91	0.91
D_3	100	1000	1000	100,000	670	8.44	0.09	0.09	0.09	0.09
D_4	1000	1000	1000	100,000	24740	25.75	0.5	0.5	0.5	0.5
D_5	1000	100	10,000	100,000	8173	9.18	0.09	0.91	0.5	0.29
D_6	1000	10	100,000	100,000	965	1.97	0.01	0.99	0.5	0.10

Example 6.10 **Comparison of six pattern evaluation measures on typical data sets.** The relationships between the purchases of two items, *milk* and *coffee*, can be examined by summarizing their purchase history in Table 6.8, a 2×2 contingency table, where an entry such as mc represents the number of transactions containing both milk and coffee.

Table 6.9 shows a set of transactional data sets with their corresponding contingency tables and the associated values for each of the six evaluation measures. Let's first examine the first four data sets, D_1 through D_4. From the table, we see that m and c are positively associated in D_1 and D_2, negatively associated in D_3, and neutral in D_4. For D_1 and D_2, m and c are positively associated because mc (10,000) is considerably greater than $\overline{m}c$ (1000) and $m\bar{c}$ (1000). Intuitively, for people who bought milk ($m = 10,000 + 1000 = 11,000$), it is very likely that they also bought coffee ($mc/m = 10/11 = 91\%$), and vice versa.

The results of the four newly introduced measures show that m and c are strongly positively associated in both data sets by producing a measure value of 0.91. However, *lift* and χ^2 generate dramatically different measure values for D_1 and D_2 due to their sensitivity to $\overline{mc}$. In fact, in many real-world scenarios, $\overline{mc}$ is usually huge and unstable. For example, in a market basket database, the total number of transactions could fluctuate on a daily basis and overwhelmingly exceed the number of transactions containing any particular itemset. Therefore, a good interestingness measure should not be affected by transactions that do not contain the itemsets of interest; otherwise, it would generate unstable results, as illustrated in D_1 and D_2.

Similarly, in D_3, the four new measures correctly show that m and c are strongly negatively associated because the m to c ratio equals the mc to m ratio, that is, $100/1100 = 9.1\%$. However, *lift* and χ^2 both contradict this in an incorrect way: Their values for D_2 are between those for D_1 and D_3.

For data set D_4, both *lift* and χ^2 indicate a highly positive association between m and c, whereas the others indicate a "neutral" association because the ratio of mc to $\overline{m}c$ equals the ratio of mc to $m\overline{c}$, which is 1. This means that if a customer buys coffee (or milk), the probability that he or she will also purchase milk (or coffee) is exactly 50%. ■

"*Why are* lift *and* χ^2 *so poor at distinguishing pattern association relationships in the previous transactional data sets?*" To answer this, we have to consider the *null-transactions*. A **null-transaction** is a transaction that does not contain any of the itemsets being examined. In our example, $\overline{mc}$ represents the number of null-transactions. *Lift* and χ^2 have difficulty distinguishing interesting pattern association relationships because they are both strongly influenced by $\overline{mc}$. Typically, the number of null-transactions can outweigh the number of individual purchases because, for example, many people may buy neither milk nor coffee. On the other hand, the other four measures are good indicators of interesting pattern associations because their definitions remove the influence of $\overline{mc}$ (i.e., they are not influenced by the number of null-transactions).

This discussion shows that it is highly desirable to have a measure that has a value that is independent of the number of null-transactions. A measure is **null-invariant** if its value is free from the influence of null-transactions. Null-invariance is an important property for measuring association patterns in large transaction databases. Among the six discussed measures in this subsection, only *lift* and χ^2 are not null-invariant measures.

"*Among the* all_confidence, max_confidence, Kulczynski, *and* cosine *measures, which is best at indicating interesting pattern relationships*?"

To answer this question, we introduce the **imbalance ratio** (**IR**), which assesses the imbalance of two itemsets, A and B, in rule implications. It is defined as

$$IR(A, B) = \frac{|sup(A) - sup(B)|}{sup(A) + sup(B) - sup(A \cup B)}, \tag{6.13}$$

where the numerator is the absolute value of the difference between the support of the itemsets A and B, and the denominator is the number of transactions containing A or B. If the two directional implications between A and B are the same, then $IR(A, B)$ will be zero. Otherwise, the larger the difference between the two, the larger the imbalance ratio. This ratio is independent of the number of null-transactions and independent of the total number of transactions.

Let's continue examining the remaining data sets in Example 6.10.

Example 6.11 Comparing null-invariant measures in pattern evaluation. Although the four measures introduced in this section are null-invariant, they may present dramatically

different values on some subtly different data sets. Let's examine data sets D_5 and D_6, shown earlier in Table 6.9, where the two events m and c have unbalanced conditional probabilities. That is, the ratio of mc to c is greater than 0.9. This means that knowing that c occurs should strongly suggest that m occurs also. The ratio of mc to m is less than 0.1, indicating that m implies that c is quite unlikely to occur. The *all_confidence* and *cosine* measures view both cases as negatively associated and the *Kulc* measure views both as neutral. The *max_confidence* measure claims strong positive associations for these cases. The measures give very diverse results!

"*Which measure intuitively reflects the true relationship between the purchase of milk and coffee?*" Due to the "*balanced*" skewness of the data, it is difficult to argue whether the two data sets have positive or negative association. From one point of view, only $mc/(mc + m\bar{c}) = 1000/(1000 + 10,000) = 9.09\%$ of milk-related transactions contain coffee in D_5 and this percentage is $1000/(1000 + 100,000) = 0.99\%$ in D_6, both indicating a negative association. On the other hand, 90.9% of transactions in D_5 (i.e., $mc/(mc + \overline{m}c) = 1000/(1000 + 100)$) and 9% in D_6 (i.e., $1000/(1000 + 10)$) containing coffee contain milk as well, which indicates a positive association between milk and coffee. These draw very different conclusions.

For such "balanced" skewness, it could be fair to treat it as neutral, as *Kulc* does, and in the meantime indicate its skewness using the *imbalance ratio (IR)*. According to Eq. (6.13), for D_4 we have $IR(m, c) = 0$, a perfectly balanced case; for D_5, $IR(m, c) = 0.89$, a rather imbalanced case; whereas for D_6, $IR(m, c) = 0.99$, a very skewed case. Therefore, the two measures, *Kulc* and *IR*, work together, presenting a clear picture for all three data sets, D_4 through D_6. ■

In summary, the use of only support and confidence measures to mine associations may generate a large number of rules, many of which can be uninteresting to users. Instead, we can augment the support–confidence framework with a pattern interestingness measure, which helps focus the mining toward rules with strong pattern relationships. The added measure substantially reduces the number of rules generated and leads to the discovery of more meaningful rules. Besides those introduced in this section, many other interestingness measures have been studied in the literature. Unfortunately, most of them do not have the null-invariance property. Because large data sets typically have many null-transactions, it is important to consider the null-invariance property when selecting appropriate interestingness measures for pattern evaluation. Among the four null-invariant measures studied here, namely *all_confidence*, *max_confidence*, *Kulc*, and *cosine*, we recommend using *Kulc* in conjunction with the imbalance ratio.

6.4 Summary

- The discovery of frequent patterns, associations, and correlation relationships among huge amounts of data is useful in selective marketing, decision analysis, and business management. A popular area of application is **market basket analysis**, which studies

customers' buying habits by searching for itemsets that are frequently purchased together (or in sequence).

- **Association rule mining** consists of first finding **frequent itemsets** (sets of items, such as A and B, satisfying a *minimum support threshold*, or percentage of the task-relevant tuples), from which **strong** association rules in the form of $A \Rightarrow B$ are generated. These rules also satisfy a *minimum confidence threshold* (a prespecified probability of satisfying B under the condition that A is satisfied). Associations can be further analyzed to uncover **correlation rules**, which convey statistical correlations between itemsets A and B.
- Many efficient and scalable algorithms have been developed for **frequent itemset mining**, from which association and correlation rules can be derived. These algorithms can be classified into three categories: (1) *Apriori-like algorithms*, (2) *frequent pattern growth–based algorithms* such as FP-growth, and (3) *algorithms that use the vertical data format.*
- The **Apriori algorithm** is a seminal algorithm for mining frequent itemsets for Boolean association rules. It explores the level-wise mining Apriori property that *all nonempty subsets of a frequent itemset must also be frequent.* At the kth iteration (for $k \geq 2$), it forms frequent k-itemset candidates based on the frequent $(k-1)$-itemsets, and scans the database once to find the *complete* set of frequent k-itemsets, L_k.

  Variations involving hashing and transaction reduction can be used to make the procedure more efficient. Other variations include partitioning the data (mining on each partition and then combining the results) and sampling the data (mining on a data subset). These variations can reduce the number of data scans required to as little as two or even one.
- **Frequent pattern growth** is a method of mining frequent itemsets without candidate generation. It constructs a highly compact data structure (an *FP-tree*) to compress the original transaction database. Rather than employing the generate-and-test strategy of Apriori-like methods, it focuses on frequent pattern (fragment) growth, which avoids costly candidate generation, resulting in greater efficiency.
- **Mining frequent itemsets using the vertical data format (Eclat)** is a method that transforms a given data set of transactions in the horizontal data format of *TID-itemset* into the vertical data format of *item-TID_set*. It mines the transformed data set by *TID_set* intersections based on the Apriori property and additional optimization techniques such as *diffset.*
- Not all strong association rules are interesting. Therefore, the support–confidence framework should be augmented with a pattern evaluation measure, which promotes the mining of *interesting* rules. A measure is **null-invariant** if its value is free from the influence of **null-transactions** (i.e., the *transactions that do not contain any of the itemsets being examined*). Among many pattern evaluation measures, we examined ***lift***, χ^2***, all_confidence, max_confidence, Kulczynski***, and ***cosine***, and showed

that only the latter four are null-invariant. We suggest using the Kulczynski measure, together with the imbalance ratio, to present pattern relationships among itemsets.

6.5 Exercises

6.1 Suppose you have the set $\mathcal{C}$ of all frequent closed itemsets on a data set D, as well as the support count for each frequent closed itemset. Describe an algorithm to determine whether a given itemset X is frequent or not, and the support of X if it is frequent.

6.2 An itemset X is called a *generator* on a data set D if there does not exist a proper sub-itemset $Y \subset X$ such that $support(X) = support(Y)$. A generator X is a *frequent generator* if $support(X)$ passes the minimum support threshold. Let $\mathcal{G}$ be the set of all frequent generators on a data set D.

(a) Can you determine whether an itemset A is frequent and the support of A, if it is frequent, using only $\mathcal{G}$ and the support counts of all frequent generators? If yes, present your algorithm. Otherwise, what other information is needed? Can you give an algorithm assuming the information needed is available?

(b) What is the relationship between closed itemsets and generators?

6.3 The Apriori algorithm makes use of *prior knowledge* of subset support properties.

(a) Prove that all nonempty subsets of a frequent itemset must also be frequent.

(b) Prove that the support of any nonempty subset s' of itemset s must be at least as great as the support of s.

(c) Given frequent itemset l and subset s of l, prove that the confidence of the rule "$s' \Rightarrow (l - s')$" cannot be more than the confidence of "$s \Rightarrow (l - s)$," where s' is a subset of s.

(d) A *partitioning* variation of Apriori subdivides the transactions of a database D into n nonoverlapping partitions. Prove that any itemset that is frequent in D must be frequent in at least one partition of D.

6.4 Let c be a candidate itemset in C_k generated by the Apriori algorithm. How many length-$(k-1)$ subsets do we need to check in the prune step? Per your previous answer, can you give an improved version of procedure has_infrequent_subset in Figure 6.4?

6.5 Section 6.2.2 describes a method for *generating association rules* from frequent itemsets. Propose a more efficient method. Explain why it is more efficient than the one proposed there. (*Hint:* Consider incorporating the properties of Exercises 6.3(b), (c) into your design.)

6.6 A database has five transactions. Let $min\_sup = 60\%$ and $min\_conf = 80\%$.

TID	*items_bought*
T100	{M, O, N, K, E, Y}
T200	{D, O, N, K, E, Y }
T300	{M, A, K, E}
T400	{M, U, C, K, Y}
T500	{C, O, O, K, I, E}

(a) Find all frequent itemsets using Apriori and FP-growth, respectively. Compare the efficiency of the two mining processes.

(b) List all the *strong* association rules (with support s and confidence c) matching the following metarule, where X is a variable representing customers, and $item_i$ denotes variables representing items (e.g., "*A*," "*B*,"):

$$\forall x \in transaction,\ buys(X, item_1) \wedge buys(X, item_2) \Rightarrow buys(X, item_3) \quad [s, c]$$

6.7 (**Implementation project**) Using a programming language that you are familiar with, such as C++ or Java, implement three *frequent itemset mining* algorithms introduced in this chapter: (1) Apriori [AS94b], (2) FP-growth [HPY00], and (3) Eclat [Zak00] (mining using the vertical data format). Compare the performance of each algorithm with various kinds of large data sets. Write a report to analyze the situations (e.g., data size, data distribution, minimal support threshold setting, and pattern density) where one algorithm may perform better than the others, and state why.

6.8 A database has four transactions. Let $min\_sup = 60\%$ and $min\_conf = 80\%$.

cust_ID	*TID*	*items_bought* (in the form of *brand-item_category*)
01	T100	{King's-Crab, Sunset-Milk, Dairyland-Cheese, Best-Bread}
02	T200	{Best-Cheese, Dairyland-Milk, Goldenfarm-Apple, Tasty-Pie, Wonder-Bread}
01	T300	{Westcoast-Apple, Dairyland-Milk, Wonder-Bread, Tasty-Pie}
03	T400	{Wonder-Bread, Sunset-Milk, Dairyland-Cheese}

(a) At the granularity of *item_category* (e.g., $item_i$ could be "*Milk*"), for the rule template,

$$\forall X \in transaction,\ buys(X, item_1) \wedge buys(X, item_2) \Rightarrow buys(X, item_3) \quad [s, c],$$

list the frequent k-itemset for the largest k, and *all* the *strong* association rules (with their support s and confidence c) containing the frequent k-itemset for the largest k.

(b) At the granularity of *brand-item_category* (e.g., $item_i$ could be "*Sunset-Milk*"), for the rule template,

$$\forall X \in customer,\ buys(X, item_1) \wedge buys(X, item_2) \Rightarrow buys(X, item_3),$$

list the frequent k-itemset for the largest k (but do not print any rules).

6.9 Suppose that a large store has a transactional database that is *distributed* among four locations. Transactions in each component database have the same format, namely $T_j : \{i_1, \ldots, i_m\}$, where T_j is a transaction identifier, and i_k $(1 \leq k \leq m)$ is the identifier of an item purchased in the transaction. Propose an efficient algorithm to mine global association rules. You may present your algorithm in the form of an outline. Your algorithm should not require shipping all the data to one site and should not cause excessive network communication overhead.

6.10 Suppose that frequent itemsets are saved for a large transactional database, *DB*. Discuss how to efficiently mine the (global) association rules under the same minimum support threshold, if a set of new transactions, denoted as ΔDB, is *(incrementally) added in*?

6.11 Most frequent pattern mining algorithms consider only distinct items in a transaction. However, multiple occurrences of an item in the same shopping basket, such as four cakes and three jugs of milk, can be important in transactional data analysis. How can one mine frequent itemsets efficiently considering multiple occurrences of items? Propose modifications to the well-known algorithms, such as Apriori and FP-growth, to adapt to such a situation.

6.12 (**Implementation project**) Many techniques have been proposed to further improve the performance of frequent itemset mining algorithms. Taking FP-tree–based frequent pattern growth algorithms (e.g., FP-growth) as an example, implement one of the following optimization techniques. Compare the performance of your new implementation with the unoptimized version.

(a) The frequent pattern mining method of Section 6.2.4 uses an FP-tree to generate conditional pattern bases using a bottom-up projection technique (i.e., project onto the prefix path of an item p). However, one can develop a *top-down projection* technique, that is, project onto the suffix path of an item p in the generation of a conditional pattern base. Design and implement such a top-down FP-tree mining method. Compare its performance with the bottom-up projection method.

(b) Nodes and pointers are used uniformly in an FP-tree in the FP-growth algorithm design. However, such a structure may consume a lot of space when the data are sparse. One possible alternative design is to explore *array- and pointer-based hybrid implementation*, where a node may store multiple items when it contains no splitting point to multiple sub-branches. Develop such an implementation and compare it with the original one.

(c) It is time and space consuming to generate numerous conditional pattern bases during pattern-growth mining. An interesting alternative is to *push right* the branches that have been mined for a particular item p, that is, to push them to the remaining branch(es) of the FP-tree. This is done so that fewer conditional pattern bases have to be generated and additional sharing can be explored when mining the remaining FP-tree branches. Design and implement such a method and conduct a performance study on it.

6.13 Give a short example to show that items in a strong association rule actually may be *negatively correlated*.

6.14 The following contingency table summarizes supermarket transaction data, where *hot dogs* refers to the transactions containing hot dogs, $\overline{hot\ dogs}$ refers to the transactions that do not contain hot dogs, *hamburgers* refers to the transactions containing hamburgers, and $\overline{hamburgers}$ refers to the transactions that do not contain hamburgers.

	hot dogs	$\overline{hot\ dogs}$	Σ_{row}
hamburgers	2000	500	2500
$\overline{hamburgers}$	1000	1500	2500
Σ_{col}	3000	2000	5000

(a) Suppose that the association rule "*hot dogs* $\Rightarrow$ *hamburgers*" is mined. Given a minimum support threshold of 25% and a minimum confidence threshold of 50%, is this association rule strong?

(b) Based on the given data, is the purchase of *hot dogs* independent of the purchase of *hamburgers*? If not, what kind of *correlation* relationship exists between the two?

(c) Compare the use of the *all_confidence*, *max_confidence*, *Kulczynski*, and *cosine* measures with *lift* and *correlation* on the given data.

6.15 (**Implementation project**) The DBLP data set *(www.informatik.uni-trier .de/~ley/db/)* consists of over one million entries of research papers published in computer science conferences and journals. Among these entries, there are a good number of authors that have coauthor relationships.

(a) Propose a method to efficiently mine a set of coauthor relationships that are closely correlated (e.g., often coauthoring papers together).

(b) Based on the mining results and the pattern evaluation measures discussed in this chapter, discuss which measure may convincingly uncover close collaboration patterns better than others.

(c) Based on the study in (a), develop a method that can roughly predict advisor and advisee relationships and the approximate period for such advisory supervision.

6.6 Bibliographic Notes

Association rule mining was first proposed by Agrawal, Imielinski, and Swami [AIS93]. The Apriori algorithm discussed in Section 6.2.1 for frequent itemset mining was presented in Agrawal and Srikant [AS94b]. A variation of the algorithm using a similar pruning heuristic was developed independently by Mannila, Tiovonen, and Verkamo

[MTV94]. A joint publication combining these works later appeared in Agrawal, Mannila, Srikant et al. [AMS+96]. A method for generating association rules from frequent itemsets is described in Agrawal and Srikant [AS94a].

References for the variations of Apriori described in Section 6.2.3 include the following. The use of hash tables to improve association mining efficiency was studied by Park, Chen, and Yu [PCY95a]. The partitioning technique was proposed by Savasere, Omiecinski, and Navathe [SON95]. The sampling approach is discussed in Toivonen [Toi96]. A dynamic itemset counting approach is given in Brin, Motwani, Ullman, and Tsur [BMUT97]. An efficient incremental updating of mined association rules was proposed by Cheung, Han, Ng, and Wong [CHNW96]. Parallel and distributed association data mining under the Apriori framework was studied by Park, Chen, and Yu [PCY95b]; Agrawal and Shafer [AS96]; and Cheung, Han, Ng, et al. [CHN+96]. Another parallel association mining method, which explores itemset clustering using a vertical database layout, was proposed in Zaki, Parthasarathy, Ogihara, and Li [ZPOL97].

Other scalable frequent itemset mining methods have been proposed as alternatives to the Apriori-based approach. FP-growth, a pattern-growth approach for mining frequent itemsets without candidate generation, was proposed by Han, Pei, and Yin [HPY00] (Section 6.2.4). An exploration of hyper structure mining of frequent patterns, called H-Mine, was proposed by Pei, Han, Lu, et al. [PHL+01]. A method that integrates top-down and bottom-up traversal of FP-trees in pattern-growth mining was proposed by Liu, Pan, Wang, and Han [LPWH02]. An array-based implementation of prefix-tree structure for efficient pattern growth mining was proposed by Grahne and Zhu [GZ03b]. Eclat, an approach for mining frequent itemsets by exploring the vertical data format, was proposed by Zaki [Zak00]. A depth-first generation of frequent itemsets by a tree projection technique was proposed by Agarwal, Aggarwal, and Prasad [AAP01]. An integration of association mining with relational database systems was studied by Sarawagi, Thomas, and Agrawal [STA98].

The mining of frequent closed itemsets was proposed in Pasquier, Bastide, Taouil, and Lakhal [PBTL99], where an Apriori-based algorithm called A-Close for such mining was presented. CLOSET, an efficient closed itemset mining algorithm based on the frequent pattern growth method, was proposed by Pei, Han, and Mao [PHM00]. CHARM by Zaki and Hsiao [ZH02] developed a compact vertical TID list structure called *diffset*, which records only the difference in the TID list of a candidate pattern from its prefix pattern. A fast hash-based approach is also used in CHARM to prune nonclosed patterns. CLOSET+ by Wang, Han, and Pei [WHP03] integrates previously proposed effective strategies as well as newly developed techniques such as hybrid tree-projection and item skipping. AFOPT, a method that explores a *right push* operation on FP-trees during the mining process, was proposed by Liu, Lu, Lou, and Yu [LLLY03]. Grahne and Zhu [GZ03b] proposed a prefix-tree–based algorithm integrated with array representation, called FPClose, for mining closed itemsets using a pattern-growth approach.

Pan, Cong, Tung, et al. [PCT+03] proposed CARPENTER, a method for finding closed patterns in long biological data sets, which integrates the advantages of vertical

data formats and pattern growth methods. Mining max-patterns was first studied by Bayardo [Bay98], where MaxMiner, an Apriori-based, level-wise, breadth-first search method, was proposed to find *max-itemset* by performing *superset frequency pruning* and *subset infrequency pruning* for search space reduction. Another efficient method, MAFIA, developed by Burdick, Calimlim, and Gehrke [BCG01], uses vertical bitmaps to compress TID lists, thus improving the counting efficiency. A FIMI (Frequent Itemset Mining Implementation) workshop dedicated to implementation methods for frequent itemset mining was reported by Goethals and Zaki [GZ03a].

The problem of mining interesting rules has been studied by many researchers. The statistical independence of rules in data mining was studied by Piatetski-Shapiro [P-S91]. The interestingness problem of strong association rules is discussed in Chen, Han, and Yu [CHY96]; Brin, Motwani, and Silverstein [BMS97]; and Aggarwal and Yu [AY99], which cover several interestingness measures, including *lift*. An efficient method for generalizing associations to correlations is given in Brin, Motwani, and Silverstein [BMS97]. Other alternatives to the support–confidence framework for assessing the interestingness of association rules are proposed in Brin, Motwani, Ullman, and Tsur [BMUT97] and Ahmed, El-Makky, and Taha [AEMT00].

A method for mining strong gradient relationships among itemsets was proposed by Imielinski, Khachiyan, and Abdulghani [IKA02]. Silverstein, Brin, Motwani, and Ullman [SBMU98] studied the problem of mining causal structures over transaction databases. Some comparative studies of different interestingness measures were done by Hilderman and Hamilton [HH01]. The notion of null transaction invariance was introduced, together with a comparative analysis of interestingness measures, by Tan, Kumar, and Srivastava [TKS02]. The use of *all_confidence* as a correlation measure for generating interesting association rules was studied by Omiecinski [Omi03] and by Lee, Kim, Cai, and Han [LKCH03]. Wu, Chen, and Han [WCH10] introduced the Kulczynski measure for associative patterns and performed a comparative analysis of a set of measures for pattern evaluation.

7 Advanced Pattern Mining

Frequent pattern mining has reached far beyond the basics due to substantial research, numerous extensions of the problem scope, and broad application studies. In this chapter, you will learn methods for advanced pattern mining. We begin by laying out a general road map for pattern mining. We introduce methods for mining various kinds of patterns, and discuss extended applications of pattern mining. We include in-depth coverage of methods for mining many kinds of patterns: multilevel patterns, multidimensional patterns, patterns in continuous data, rare patterns, negative patterns, constrained frequent patterns, frequent patterns in high-dimensional data, colossal patterns, and compressed and approximate patterns. Other pattern mining themes, including mining sequential and structured patterns and mining patterns from spatiotemporal, multimedia, and stream data, are considered more advanced topics and are not covered in this book. Notice that *pattern mining* is a more general term than *frequent pattern mining* since the former covers rare and negative patterns as well. However, when there is no ambiguity, the two terms are used interchangeably.

7.1 Pattern Mining: A Road Map

Chapter 6 introduced the basic concepts, techniques, and applications of frequent pattern mining using market basket analysis as an example. Many other kinds of data, user requests, and applications have led to the development of numerous, diverse methods for mining patterns, associations, and correlation relationships. Given the rich literature in this area, it is important to lay out a clear road map to help us get an organized picture of the field and to select the best methods for pattern mining applications.

Figure 7.1 outlines a general road map on pattern mining research. Most studies mainly address three pattern mining aspects: the kinds of patterns mined, mining methodologies, and applications. Some studies, however, integrate multiple aspects; for example, different applications may need to mine different patterns, which naturally leads to the development of new mining methodologies.

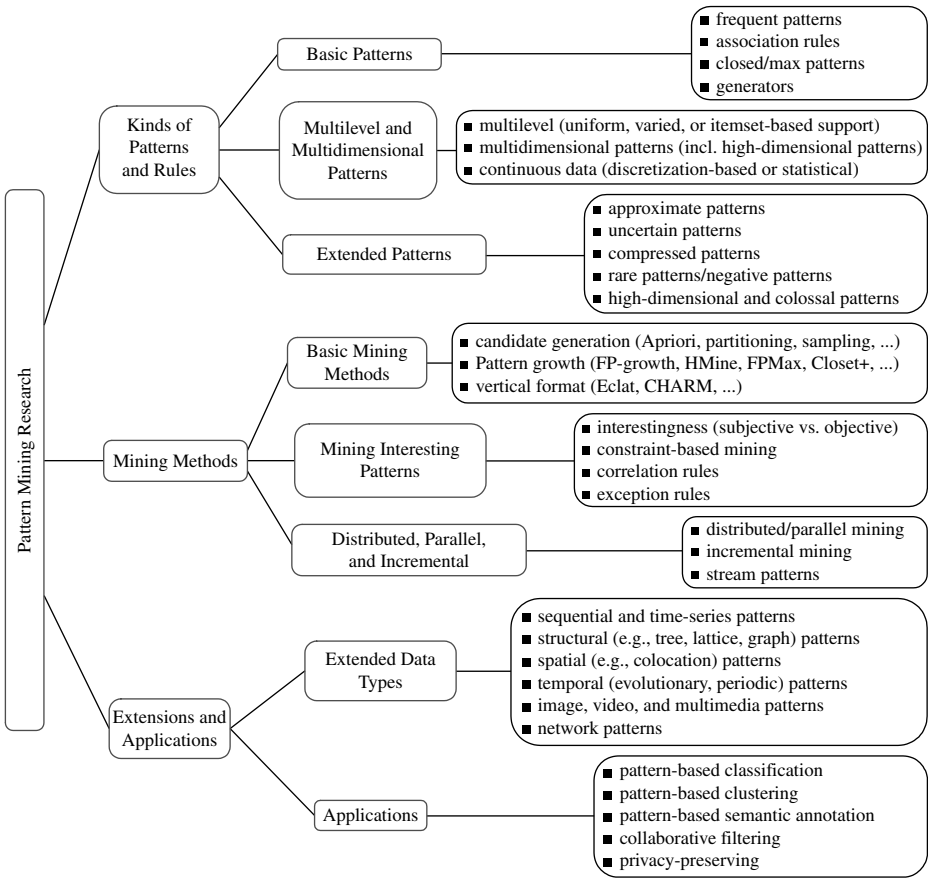

Figure 7.1 A general road map on pattern mining research.

Based on pattern diversity, pattern mining can be classified using the following criteria:

- **Basic patterns:** As discussed in Chapter 6, a frequent pattern may have several alternative forms, including a simple frequent pattern, a closed pattern, or a max-pattern. To review, a **frequent pattern** is a pattern (or itemset) that satisfies a minimum support threshold. A pattern p is a **closed pattern** if there is no superpattern p' with the same support as p. Pattern p is a **max-pattern** if there exists no frequent superpattern of p. Frequent patterns can also be mapped into **association rules**, or other kinds of rules based on interestingness measures. Sometimes we may also be interested in **infrequent** or **rare patterns** (i.e., patterns that occur rarely but are of critical importance, or **negative patterns** (i.e., patterns that reveal a negative correlation between items).

- **Based on the *abstraction* levels involved in a pattern:** Patterns or association rules may have items or concepts residing at high, low, or multiple abstraction levels. For example, suppose that a set of association rules mined includes the following rules where X is a variable representing a customer:

$$buys(X, \text{“}computer\text{”}) \Rightarrow buys(X, \text{“}printer\text{”}) \tag{7.1}$$

$$buys(X, \text{“}laptop\ computer\text{”}) \Rightarrow buys(X, \text{“}color\ laser\ printer\text{”}) \tag{7.2}$$

  In Rules (7.1) and (7.2), the items bought are referenced at different abstraction levels (e.g., “*computer*” is a higher-level abstraction of “*laptop computer*,” and “*color laser printer*” is a lower-level abstraction of “*printer*”). We refer to the rule set mined as consisting of **multilevel association rules**. If, instead, the rules within a given set do not reference items or attributes at different abstraction levels, then the set contains **single-level association rules**.

- **Based on the *number of dimensions* involved in the rule or pattern:** If the items or attributes in an association rule or pattern reference only one dimension, it is a **single-dimensional association rule/pattern**. For example, Rules (7.1) and (7.2) are single-dimensional association rules because they each refer to only one dimension, *buys*.[1]

  If a rule/pattern references two or more dimensions, such as *age, income*, and *buys*, then it is a **multidimensional association rule/pattern**. The following is an example of a multidimensional rule:

$$age(X, \text{“}20 \ldots 29\text{”}) \wedge income(X, \text{“}52K \ldots 58K\text{”}) \Rightarrow buys(X, \text{“}iPad\text{”}). \tag{7.3}$$

- **Based on the *types of values* handled in the rule or pattern:** If a rule involves associations between the presence or absence of items, it is a **Boolean association rule**. For example, Rules (7.1) and (7.2) are Boolean association rules obtained from market basket analysis.

  If a rule describes associations between quantitative items or attributes, then it is a **quantitative association rule**. In these rules, quantitative values for items or attributes are partitioned into intervals. Rule (7.3) can also be considered a quantitative association rule where the quantitative attributes *age* and *income* have been discretized.

- **Based on the *constraints* or *criteria* used to mine *selective patterns*:** The patterns or rules to be discovered can be **constraint-based** (i.e., satisfying a set of user-defined constraints), **approximate**, **compressed**, **near-match** (i.e., those that tally the support count of the near or almost matching itemsets), **top-*k*** (i.e., the k most frequent itemsets for a user-specified value, k), **redundancy-aware top-*k*** (i.e., the top-k patterns with similar or redundant patterns excluded), and so on.

[1] Following the terminology used in multidimensional databases, we refer to each distinct predicate in a rule as a *dimension*.

Alternatively, pattern mining can be classified with respect to the kinds of data and applications involved, using the following criteria:

- **Based on *kinds of data and features* to be mined:** Given relational and data warehouse data, most people are interested in itemsets. Thus, frequent pattern mining in this context is essentially **frequent itemset mining**, that is, to mine frequent *sets of items*. However, in many other applications, patterns may involve sequences and structures. For example, by studying the order in which items are frequently purchased, we may find that customers tend to first buy a PC, followed by a digital camera, and then a memory card. This leads to **sequential patterns**, that is, frequent *subsequences* (which are often separated by some other events) in a *sequence of ordered events*.

  We may also mine **structural patterns**, that is, frequent *substructures*, in a *structured data set*. Note that *structure* is a general concept that covers many different kinds of structural forms such as directed graphs, undirected graphs, lattices, trees, sequences, sets, single items, or combinations of such structures. Single items are the simplest form of structure. Each element of a general pattern may contain a subsequence, a subtree, a subgraph, and so on, and such containment relationships can be defined recursively. Therefore, structural pattern mining can be considered as the most general form of frequent pattern mining.

- **Based on *application domain-specific semantics*:** Both data and applications can be very diverse, and therefore the patterns to be mined can differ largely based on their domain-specific semantics. Various kinds of application data include spatial data, temporal data, spatiotemporal data, multimedia data (e.g., image, audio, and video data), text data, time-series data, DNA and biological sequences, software programs, chemical compound structures, web structures, sensor networks, social and information networks, biological networks, data streams, and so on. This diversity can lead to dramatically different pattern mining methodologies.

- **Based on *data analysis usages*:** Frequent pattern mining often serves as an intermediate step for improved data understanding and more powerful data analysis. For example, it can be used as a feature extraction step for classification, which is often referred to as **pattern-based classification**. Similarly, **pattern-based clustering** has shown its strength at clustering high-dimensional data. For improved data understanding, patterns can be used for semantic annotation or contextual analysis. Pattern analysis can also be used in **recommender systems**, which recommend information items (e.g., books, movies, web pages) that are likely to be of interest to the user based on similar users' patterns. Different analysis tasks may require mining rather different kinds of patterns as well.

The next several sections present advanced methods and extensions of pattern mining, as well as their application. Section 7.2 discusses methods for mining multilevel patterns, multidimensional patterns, patterns and rules with continuous attributes, rare patterns, and negative patterns. Constraint-based pattern mining is studied in

Section 7.3. Section 7.4 explains how to mine high-dimensional and colossal patterns. The mining of compressed and approximate patterns is detailed in Section 7.5. Section 7.6 discusses the exploration and applications of pattern mining. More advanced topics regarding mining sequential and structural patterns, and pattern mining in complex and diverse kinds of data are briefly introduced in Chapter 13.

7.2 Pattern Mining in Multilevel, Multidimensional Space

This section focuses on methods for mining in multilevel, multidimensional space. In particular, you will learn about mining multilevel associations (Section 7.2.1), multidimensional associations (Section 7.2.2), quantitative association rules (Section 7.2.3), and rare patterns and negative patterns (Section 7.2.4). *Multilevel associations* involve concepts at different abstraction levels. *Multidimensional associations* involve more than one dimension or predicate (e.g., rules that relate what a customer *buys* to his or her *age*). *Quantitative association rules* involve numeric attributes that have an implicit ordering among values (e.g., *age*). *Rare patterns* are patterns that suggest interesting although rare item combinations. *Negative patterns* show negative correlations between items.

7.2.1 Mining Multilevel Associations

For many applications, strong associations discovered at high abstraction levels, though with high support, could be commonsense knowledge. We may want to drill down to find novel patterns at more detailed levels. On the other hand, there could be too many scattered patterns at low or primitive abstraction levels, some of which are just trivial specializations of patterns at higher levels. Therefore, it is interesting to examine how to develop effective methods for mining patterns at multiple abstraction levels, with sufficient flexibility for easy traversal among different abstraction spaces.

Example 7.1 **Mining multilevel association rules.** Suppose we are given the task-relevant set of transactional data in Table 7.1 for sales in an *AllElectronics* store, showing the items purchased for each transaction. The concept hierarchy for the items is shown in Figure 7.2. A concept hierarchy defines a sequence of mappings from a set of low-level concepts to a higher-level, more general concept set. Data can be generalized by replacing low-level concepts within the data by their corresponding higher-level concepts, or *ancestors*, from a concept hierarchy.

Figure 7.2's concept hierarchy has five levels, respectively referred to as levels 0 through 4, starting with level 0 at the root node for **all** (the most general abstraction level). Here, level 1 includes *computer, software, printer and camera,* and *computer accessory*; level 2 includes *laptop computer, desktop computer*, *office software, antivirus software*, etc.; and level 3 includes *Dell desktop computer, . . . , Microsoft office software*, etc. Level 4 is the most specific abstraction level of this hierarchy. It consists of the raw data values.

Table 7.1 Task-Relevant Data, *D*

TID	*Items Purchased*
T100	Apple 17″ MacBook Pro Notebook, HP Photosmart Pro b9180
T200	Microsoft Office Professional 2010, Microsoft Wireless Optical Mouse 5000
T300	Logitech VX Nano Cordless Laser Mouse, Fellowes GEL Wrist Rest
T400	Dell Studio XPS 16 Notebook, Canon PowerShot SD1400
T500	Lenovo ThinkPad X200 Tablet PC, Symantec Norton Antivirus 2010
…	…

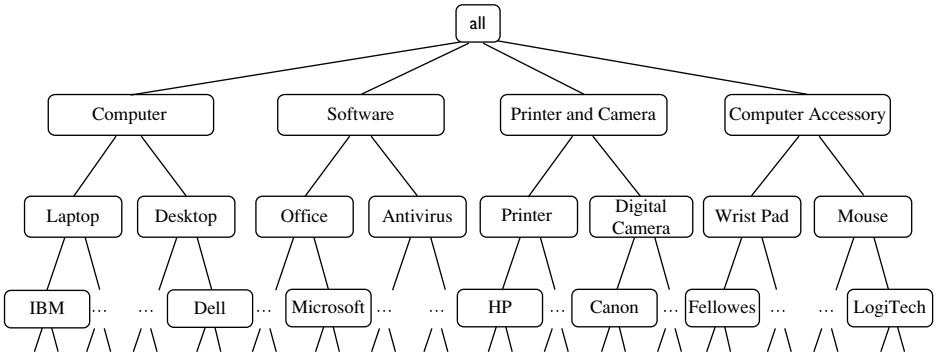

Figure 7.2 Concept hierarchy for *AllElectronics* computer items.

Concept hierarchies for nominal attributes are often implicit within the database schema, in which case they may be automatically generated using methods such as those described in Chapter 3. For our example, the concept hierarchy of Figure 7.2 was generated from data on product specifications. Concept hierarchies for numeric attributes can be generated using discretization techniques, many of which were introduced in Chapter 3. Alternatively, concept hierarchies may be specified by users familiar with the data such as store managers in the case of our example.

The items in Table 7.1 are at the lowest level of Figure 7.2's concept hierarchy. It is difficult to find interesting purchase patterns in such raw or primitive-level data. For instance, if "*Dell Studio XPS 16 Notebook*" or "*Logitech VX Nano Cordless Laser Mouse*" occurs in a very small fraction of the transactions, then it can be difficult to find strong associations involving these specific items. Few people may buy these items together, making it unlikely that the itemset will satisfy minimum support. However, we would expect that it is easier to find strong associations between generalized abstractions of these items, such as between "*Dell Notebook*" and "*Cordless Mouse.*" ■

Association rules generated from mining data at multiple abstraction levels are called **multiple-level** or **multilevel association rules**. Multilevel association rules can be

mined efficiently using concept hierarchies under a support-confidence framework. In general, a top-down strategy is employed, where counts are accumulated for the calculation of frequent itemsets at each concept level, starting at concept level 1 and working downward in the hierarchy toward the more specific concept levels, until no more frequent itemsets can be found. For each level, any algorithm for discovering frequent itemsets may be used, such as Apriori or its variations.

A number of variations to this approach are described next, where each variation involves "playing" with the support threshold in a slightly different way. The variations are illustrated in Figures 7.3 and 7.4, where nodes indicate an item or itemset that has been examined, and nodes with thick borders indicate that an examined item or itemset is frequent.

- **Using uniform minimum support for all levels** (referred to as **uniform support**): The same minimum support threshold is used when mining at each abstraction level. For example, in Figure 7.3, a minimum support threshold of 5% is used throughout (e.g., for mining from "*computer*" downward to "*laptop computer*"). Both "*computer*" and "*laptop computer*" are found to be frequent, whereas "*desktop computer*" is not.

  When a uniform minimum support threshold is used, the search procedure is simplified. The method is also simple in that users are required to specify only

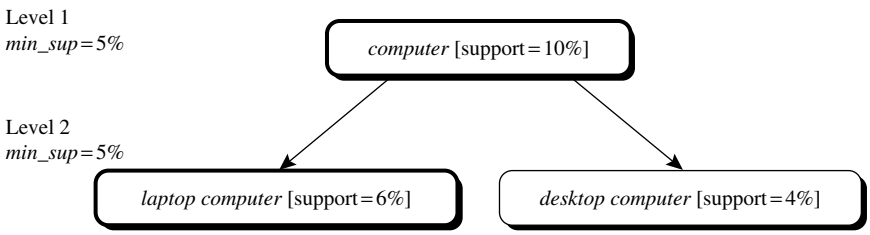

Figure 7.3 Multilevel mining with uniform support.

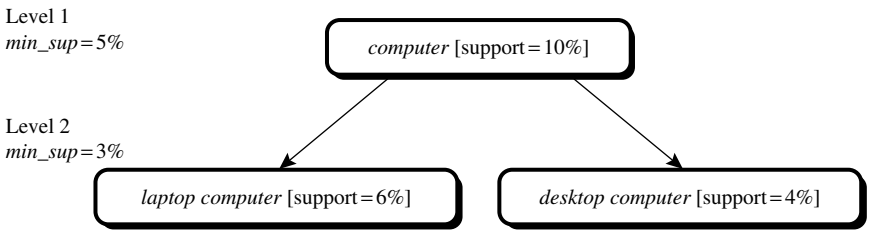

Figure 7.4 Multilevel mining with reduced support.

one minimum support threshold. An Apriori-like optimization technique can be adopted, based on the knowledge that an ancestor is a superset of its descendants: The search avoids examining itemsets containing any item of which the ancestors do not have minimum support.

The uniform support approach, however, has some drawbacks. It is unlikely that items at lower abstraction levels will occur as frequently as those at higher abstraction levels. If the minimum support threshold is set too high, it could miss some meaningful associations occurring at low abstraction levels. If the threshold is set too low, it may generate many uninteresting associations occurring at high abstraction levels. This provides the motivation for the next approach.

- **Using reduced minimum support at lower levels** (referred to as **reduced support**): Each abstraction level has its own minimum support threshold. The deeper the abstraction level, the smaller the corresponding threshold. For example, in Figure 7.4, the minimum support thresholds for levels 1 and 2 are 5% and 3%, respectively. In this way, "*computer,*" "*laptop computer,*" and "*desktop computer*" are all considered frequent.
- **Using item or group-based minimum support (referred to as group-based support)**: Because users or experts often have insight as to which groups are more important than others, it is sometimes more desirable to set up user-specific, item, or group-based minimal support thresholds when mining multilevel rules. For example, a user could set up the minimum support thresholds based on product price or on items of interest, such as by setting particularly low support thresholds for "*camera with price over $1000*" or "*Tablet PC,*" to pay particular attention to the association patterns containing items in these categories.

  For mining patterns with mixed items from groups with different support thresholds, usually the lowest support threshold among all the participating groups is taken as the support threshold in mining. This will avoid filtering out valuable patterns containing items from the group with the lowest support threshold. In the meantime, the minimal support threshold for each individual group should be kept to avoid generating uninteresting itemsets from each group. Other interestingness measures can be used after the itemset mining to extract truly interesting rules.

Notice that the Apriori property may not always hold uniformly across all of the items when mining under reduced support and group-based support. However, efficient methods can be developed based on the extension of the property. The details are left as an exercise for interested readers.

A serious side effect of mining multilevel association rules is its generation of many redundant rules across multiple abstraction levels due to the "ancestor" relationships among items. For example, consider the following rules where "*laptop computer*" is an ancestor of "*Dell laptop computer*" based on the concept hierarchy of Figure 7.2, and

where X is a variable representing customers who purchased items in *AllElectronics* transactions.

$$buys(X, \text{“}laptop\ computer\text{”}) \Rightarrow buys(X, \text{“}HP\ printer\text{”})$$
$$[support = 8\%, confidence = 70\%] \quad (7.4)$$

$$buys(X, \text{“}Dell\ laptop\ computer\text{”}) \Rightarrow buys(X, \text{“}HP\ printer\text{”})$$
$$[support = 2\%, confidence = 72\%] \quad (7.5)$$

“*If Rules (7.4) and (7.5) are both mined, then how useful is Rule (7.5)? Does it really provide any novel information?*” If the latter, less general rule does not provide new information, then it should be removed. Let's look at how this may be determined. A rule $R1$ is an **ancestor** of a rule $R2$, if $R1$ can be obtained by replacing the items in $R2$ by their ancestors in a concept hierarchy. For example, Rule (7.4) is an ancestor of Rule (7.5) because “*laptop computer*” is an ancestor of “*Dell laptop computer.*” Based on this definition, a rule can be considered redundant if its support and confidence are close to their “expected” values, based on an ancestor of the rule.

Example 7.2 **Checking redundancy among multilevel association rules.** Suppose that Rule (7.4) has a 70% confidence and 8% support, and that about one-quarter of all “*laptop computer*” sales are for “*Dell laptop computers.*” We may expect Rule (7.5) to have a confidence of around 70% (since all data samples of “*Dell laptop computer*” are also samples of “*laptop computer*”) and a support of around 2% (i.e., $8\% \times \frac{1}{4}$). If this is indeed the case, then Rule (7.5) is not interesting because it does not offer any additional information and is less general than Rule (7.4). ■

7.2.2 Mining Multidimensional Associations

So far, we have studied association rules that imply a single predicate, that is, the predicate *buys*. For instance, in mining our *AllElectronics* database, we may discover the Boolean association rule

$$buys(X, \text{“}digital\ camera\text{”}) \Rightarrow buys(X, \text{“}HP\ printer\text{”}). \quad (7.6)$$

Following the terminology used in multidimensional databases, we refer to each distinct predicate in a rule as a dimension. Hence, we can refer to Rule (7.6) as a **single-dimensional** or **intradimensional association rule** because it contains a single distinct predicate (e.g., *buys*) with multiple occurrences (i.e., the predicate occurs more than once within the rule). Such rules are commonly mined from transactional data.

Instead of considering transactional data only, sales and related information are often linked with relational data or integrated into a data warehouse. Such data stores are multidimensional in nature. For instance, in addition to keeping track of the items purchased in sales transactions, a relational database may record other attributes associated

with the items and/or transactions such as the item description or the branch location of the sale. Additional relational information regarding the customers who purchased the items (e.g., customer age, occupation, credit rating, income, and address) may also be stored. Considering each database attribute or warehouse dimension as a predicate, we can therefore mine association rules containing *multiple* predicates such as

$$age(X, \text{“20}\ldots\text{29”}) \wedge occupation(X, \text{“}student\text{”}) \Rightarrow buys(X, \text{“}laptop\text{”}). \quad (7.7)$$

Association rules that involve two or more dimensions or predicates can be referred to as **multidimensional association rules**. Rule (7.7) contains three predicates (*age*, *occupation*, and *buys*), each of which occurs *only once* in the rule. Hence, we say that it has **no repeated predicates**. Multidimensional association rules with no repeated predicates are called **interdimensional association rules**. We can also mine multidimensional association rules with repeated predicates, which contain multiple occurrences of some predicates. These rules are called **hybrid-dimensional association rules**. An example of such a rule is the following, where the predicate *buys* is repeated:

$$age(X, \text{“20}\ldots\text{29”}) \wedge buys(X, \text{“}laptop\text{”}) \Rightarrow buys(X, \text{“}HP\ printer\text{”}). \quad (7.8)$$

Database attributes can be nominal or quantitative. The values of **nominal** (or categorical) attributes are “names of things.” Nominal attributes have a finite number of possible values, with no ordering among the values (e.g., *occupation, brand*, *color*). **Quantitative** attributes are numeric and have an implicit ordering among values (e.g., *age, income*, *price*). Techniques for mining multidimensional association rules can be categorized into two basic approaches regarding the treatment of quantitative attributes.

In the first approach, *quantitative attributes are discretized using predefined concept hierarchies*. This discretization occurs before mining. For instance, a concept hierarchy for *income* may be used to replace the original numeric values of this attribute by interval labels such as “0..20K,” “21K..30K,” “31K..40K,” and so on. Here, discretization is *static* and predetermined. Chapter 3 on data preprocessing gave several techniques for discretizing numeric attributes. The discretized numeric attributes, with their interval labels, can then be treated as nominal attributes (where each interval is considered a category). We refer to this as **mining multidimensional association rules using static discretization of quantitative attributes**.

In the second approach, *quantitative attributes are discretized or clustered into “bins” based on the data distribution*. These bins may be further combined during the mining process. The discretization process is *dynamic* and established so as to satisfy some mining criteria such as maximizing the confidence of the rules mined. Because this strategy treats the numeric attribute values as quantities rather than as predefined ranges or categories, association rules mined from this approach are also referred to as **(dynamic) quantitative association rules**.

Let’s study each of these approaches for mining multidimensional association rules. For simplicity, we confine our discussion to interdimensional association rules. Note that rather than searching for frequent itemsets (as is done for single-dimensional association rule mining), in multidimensional association rule mining we search for

frequent *predicate sets*. A ***k*-predicate set** is a set containing k conjunctive predicates. For instance, the set of predicates {*age, occupation, buys*} from Rule (7.7) is a 3-predicate set. Similar to the notation used for itemsets in Chapter 6, we use the notation L_k to refer to the set of frequent k-predicate sets.

7.2.3 Mining Quantitative Association Rules

As discussed earlier, relational and data warehouse data often involve quantitative attributes or measures. We can discretize quantitative attributes into multiple intervals and then treat them as nominal data in association mining. However, such simple discretization may lead to the generation of an enormous number of rules, many of which may not be useful. Here we introduce three methods that can help overcome this difficulty to discover novel association relationships: (1) a data cube method, (2) a clustering-based method, and (3) a statistical analysis method to uncover exceptional behaviors.

Data Cube–Based Mining of Quantitative Associations

In many cases quantitative attributes can be discretized before mining using predefined concept hierarchies or data discretization techniques, where numeric values are replaced by interval labels. Nominal attributes may also be generalized to higher conceptual levels if desired. If the resulting task-relevant data are stored in a relational table, then any of the frequent itemset mining algorithms we have discussed can easily be modified so as to find all frequent predicate sets. In particular, instead of searching on only one attribute like *buys*, we need to search through all of the relevant attributes, treating each attribute–value pair as an itemset.

Alternatively, the transformed multidimensional data may be used to construct a *data cube*. Data cubes are well suited for the mining of multidimensional association rules: They store aggregates (e.g., counts) in multidimensional space, which is essential for computing the support and confidence of multidimensional association rules. An overview of data cube technology was presented in Chapter 4. Detailed algorithms for data cube computation were given in Chapter 5. Figure 7.5 shows the lattice of cuboids defining a data cube for the dimensions *age, income*, and *buys*. The cells of an n-dimensional cuboid can be used to store the support counts of the corresponding n-predicate sets. The base cuboid aggregates the task-relevant data by *age, income*, and *buys*; the 2-D cuboid, (*age, income*), aggregates by *age* and *income*, and so on; the 0-D (apex) cuboid contains the total number of transactions in the task-relevant data.

Due to the ever-increasing use of data warehouse and OLAP technology, it is possible that a data cube containing the dimensions that are of interest to the user may already exist, fully or partially materialized. If this is the case, we can simply fetch the corresponding aggregate values or compute them using lower-level materialized aggregates, and return the rules needed using a rule generation algorithm. Notice that even in this case, the Apriori property can still be used to prune the search space. If a given k-predicate set has support *sup*, which does not satisfy minimum support, then further

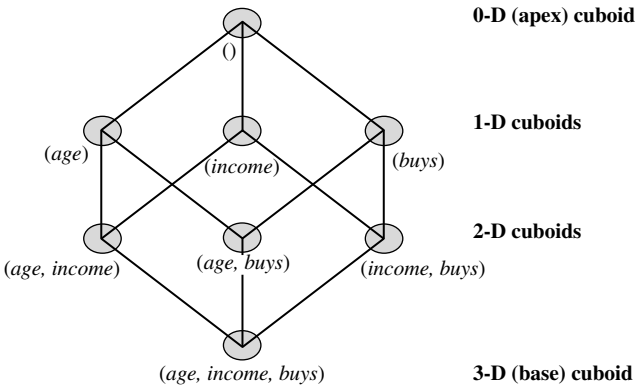

Figure 7.5 Lattice of cuboids, making up a 3-D data cube. Each cuboid represents a different group-by. The base cuboid contains the three predicates *age, income*, and *buys*.

exploration of this set should be terminated. This is because any more-specialized version of the *k*-itemset will have support no greater than *sup* and, therefore, will not satisfy minimum support either. In cases where no relevant data cube exists for the mining task, we must create one on-the-fly. This becomes an iceberg cube computation problem, where the minimum support threshold is taken as the iceberg condition (Chapter 5).

Mining Clustering-Based Quantitative Associations

Besides using discretization-based or data cube–based data sets to generate quantitative association rules, we can also generate *quantitative association rules* by clustering data in the quantitative dimensions. (Recall that objects within a cluster are similar to one another and dissimilar to those in other clusters.) The general assumption is that interesting frequent patterns or association rules are in general found at relatively dense clusters of quantitative attributes. Here, we describe a top-down approach and a bottom-up approach to clustering that finds quantitative associations.

A typical top-down approach for finding clustering-based quantitative frequent patterns is as follows. For each quantitative dimension, a standard clustering algorithm (e.g., *k*-means or a density-based clustering algorithm, as described in Chapter 10) can be applied to find clusters in this dimension that satisfy the minimum support threshold. For each cluster, we then examine the 2-D spaces generated by combining the cluster with a cluster or nominal value of another dimension to see if such a combination passes the minimum support threshold. If it does, we continue to search for clusters in this 2-D region and progress to even higher-dimensional combinations. The Apriori pruning still applies in this process: If, at any point, the support of a combination does not have minimum support, its further partitioning or combination with other dimensions cannot have minimum support either.

A bottom-up approach for finding clustering-based frequent patterns works by first clustering in high-dimensional space to form clusters with support that satisfies the minimum support threshold, and then projecting and merging those clusters in the space containing fewer dimensional combinations. However, for high-dimensional data sets, finding high-dimensional clustering itself is a tough problem. Thus, this approach is less realistic.

Using Statistical Theory to Disclose Exceptional Behavior

It is possible to discover quantitative association rules that disclose exceptional behavior, where "exceptional" is defined based on a statistical theory. For example, the following association rule may indicate exceptional behavior:

$$sex = female \Rightarrow meanwage = \$7.90/hr \ (overall\_mean\_wage = \$9.02/hr). \tag{7.9}$$

This rule states that the average wage for females is only \$7.90/hr. This rule is (subjectively) interesting because it reveals a group of people earning a significantly lower wage than the average wage of \$9.02/hr. (If the average wage was close to \$7.90/hr, then the fact that females also earn \$7.90/hr would be "uninteresting.")

An integral aspect of our definition involves applying statistical tests to confirm the validity of our rules. That is, Rule (7.9) is only accepted if a statistical test (in this case, a Z-test) confirms that with high confidence it can be inferred that the mean wage of the female population is indeed lower than the mean wage of the rest of the population. (The above rule was mined from a real database based on a 1985 U.S. census.)

An association rule under the new definition is a rule of the form:

$$population\_subset \Rightarrow \textbf{mean}\_of\_values\_for\_the\_subset, \tag{7.10}$$

where the mean of the subset is significantly different from the mean of its complement in the database (and this is validated by an appropriate statistical test).

7.2.4 Mining Rare Patterns and Negative Patterns

All the methods presented so far in this chapter have been for mining frequent patterns. Sometimes, however, it is interesting to find patterns that are rare instead of frequent, or patterns that reflect a negative correlation between items. These patterns are respectively referred to as rare patterns and negative patterns. In this subsection, we consider various ways of defining rare patterns and negative patterns, which are also useful to mine.

Example 7.3 **Rare patterns and negative patterns.** In jewelry sales data, sales of diamond watches are rare; however, patterns involving the selling of diamond watches could be interesting. In supermarket data, if we find that customers frequently buy Coca-Cola Classic or Diet Coke but not both, then buying Coca-Cola Classic and buying Diet Coke together

is considered a negative (correlated) pattern. In car sales data, a dealer sells a few fuel-thirsty vehicles (e.g., SUVs) to a given customer, and then later sells hybrid mini-cars to the same customer. Even though buying SUVs and buying hybrid mini-cars may be negatively correlated events, it can be interesting to discover and examine such exceptional cases. ■

An **infrequent** (or **rare**) **pattern** is a pattern with a frequency support that is *below* (or *far below*) a user-specified minimum support threshold. However, since the occurrence frequencies of the majority of itemsets are usually below or even far below the minimum support threshold, it is desirable in practice for users to specify other conditions for rare patterns. For example, if we want to find patterns containing at least one item with a value that is over $500, we should specify such a constraint explicitly. Efficient mining of such itemsets is discussed under mining multidimensional associations (Section 7.2.1), where the strategy is to adopt multiple (e.g., item- or group-based) minimum support thresholds. Other applicable methods are discussed under constraint-based pattern mining (Section 7.3), where user-specified constraints are pushed deep into the iterative mining process.

There are various ways we could define a negative pattern. We will consider three such definitions.

Definition 7.1: If itemsets X and Y are both frequent but rarely occur together (i.e., $sup(X \cup Y) < sup(X) \times sup(Y)$), then itemsets X and Y are **negatively correlated**, and the pattern $X \cup Y$ is a **negatively correlated pattern**. If $sup(X \cup Y) \ll sup(X) \times sup(Y)$, then X and Y are **strongly negatively correlated**, and the pattern $X \cup Y$ is a **strongly negatively correlated pattern**. □

This definition can easily be extended for patterns containing k-itemsets for $k > 2$.

A problem with the definition, however, is that it is not *null-invariant*. That is, its value can be misleadingly influenced by null transactions, where a *null-transaction* is a transaction that does not contain any of the itemsets being examined (Section 6.3.3). This is illustrated in Example 7.4.

Example 7.4 **Null-transaction problem with Definition 7.1.** If there are a lot of null-transactions in the data set, then the number of null-transactions rather than the patterns observed may strongly influence a measure's assessment as to whether a pattern is negatively correlated. For example, suppose a sewing store sells needle packages A and B. The store sold 100 packages each of A and B, but only one transaction contains both A and B. Intuitively, A is negatively correlated with B since the purchase of one does not seem to encourage the purchase of the other.

Let's see how the above Definition 7.1 handles this scenario. If there are 200 transactions, we have $sup(A \cup B) = 1/200 = 0.005$ and $sup(A) \times sup(B) = 100/200 \times 100/200 = 0.25$. Thus, $sup(A \cup B) \ll sup(A) \times sup(B)$, and so Definition 7.1 indicates that A and B are strongly negatively correlated. What if, instead of only 200 transactions in the database, there are 10^6? In this case, there are many null-transactions, that is, many contain neither A nor B. How does the definition hold up? It computes $sup(A \cup B) = 1/10^6$ and $sup(X) \times sup(Y) = 100/10^6 \times 100/10^6 = 1/10^8$.

Thus, $sup(A \cup B) \gg sup(X) \times sup(Y)$, which contradicts the earlier finding even though the number of occurrences of A and B has not changed. The measure in Definition 7.1 is not null-invariant, where *null-invariance* is essential for quality interestingness measures as discussed in Section 6.3.3. ■

Definition 7.2: If X and Y are strongly negatively correlated, then

$$sup(X \cup \overline{Y}) \times sup(\overline{X} \cup Y) \gg sup(X \cup Y) \times sup(\overline{X} \cup \overline{Y}).$$

Is this measure null-invariant? □

Example 7.5 **Null-transaction problem with Definition 7.2.** Given our needle package example, when there are in total 200 transactions in the database, we have

$$sup(A \cup \overline{B}) \times sup(\overline{A} \cup B) = 99/200 \times 99/200 = 0.245$$
$$\gg sup(A \cup B) \times sup(\overline{A} \cup \overline{B}) = 199/200 \times 1/200 \approx 0.005,$$

which, according to Definition 7.2, indicates that A and B are strongly negatively correlated. What if there are 10^6 transactions in the database? The measure would compute

$$sup(A \cup \overline{B}) \times sup(\overline{A} \cup B) = 99/10^6 \times 99/10^6 = 9.8 \times 10^{-9}$$
$$\ll sup(A \cup B) \times sup(\overline{A} \cup \overline{B}) = 199/10^6 \times (10^6 - 199)/10^6 \approx 1.99 \times 10^{-4}.$$

This time, the measure indicates that A and B are positively correlated, hence, a contradiction. The measure is not null-invariant. ■

As a third alternative, consider Definition 7.3, which is based on the Kulczynski measure (i.e., the average of conditional probabilities). It follows the spirit of interestingness measures introduced in Section 6.3.3.

Definition 7.3: Suppose that itemsets X and Y are both frequent, that is, $sup(X) \geq min\_sup$ and $sup(Y) \geq min\_sup$, where min_sup is the minimum support threshold. If $(P(X|Y) + P(Y|X))/2 < \epsilon$, where ϵ is a negative pattern threshold, then pattern $X \cup Y$ is a **negatively correlated pattern**. □

Example 7.6 **Negatively correlated patterns using Definition 7.3, based on the Kulczynski measure.** Let's reexamine our needle package example. Let min_sup be 0.01% and $\epsilon = 0.02$. When there are 200 transactions in the database, we have $sup(A) = sup(B) = 100/200 = 0.5 > 0.01\%$ and $(P(B|A) + P(A|B))/2 = (0.01 + 0.01)/2 < 0.02$; thus A and B are negatively correlated. Does this still hold true if we have many more transactions? When there are 10^6 transactions in the database, the measure computes $sup(A) = sup(B) = 100/10^6 = 0.01\% \geq 0.01\%$ and $(P(B|A) + P(A|B))/2 = (0.01 + 0.01)/2 < 0.02$, again indicating that A and B are negatively correlated. This matches our intuition. The measure does not have the null-invariance problem of the first two definitions considered.

Let's examine another case: Suppose that among 100,000 transactions, the store sold 1000 needle packages of A but only 10 packages of B; however, every time package B is

sold, package A is also sold (i.e., they appear in the same transaction). In this case, the measure computes $(P(B|A) + P(A|B))/2 = (0.01 + 1)/2 = 0.505 \gg 0.02$, which indicates that A and B are positively correlated instead of negatively correlated. This also matches our intuition. ■

With this new definition of negative correlation, efficient methods can easily be derived for mining negative patterns in large databases. This is left as an exercise for interested readers.

7.3 Constraint-Based Frequent Pattern Mining

A data mining process may uncover thousands of rules from a given data set, most of which end up being unrelated or uninteresting to users. Often, users have a good sense of which "direction" of mining may lead to interesting patterns and the "form" of the patterns or rules they want to find. They may also have a sense of "conditions" for the rules, which would eliminate the discovery of certain rules that they know would not be of interest. Thus, a good heuristic is to have the users specify such intuition or expectations as *constraints* to confine the search space. This strategy is known as **constraint-based mining**. The constraints can include the following:

- **Knowledge type constraints:** These specify the type of knowledge to be mined, such as association, correlation, classification, or clustering.
- **Data constraints:** These specify the set of task-relevant data.
- **Dimension/level constraints:** These specify the desired dimensions (or attributes) of the data, the abstraction levels, or the level of the concept hierarchies to be used in mining.
- **Interestingness constraints:** These specify thresholds on statistical measures of rule interestingness such as support, confidence, and correlation.
- **Rule constraints:** These specify the form of, or conditions on, the rules to be mined. Such constraints may be expressed as metarules (rule templates), as the maximum or minimum number of predicates that can occur in the rule antecedent or consequent, or as relationships among attributes, attribute values, and/or aggregates.

These constraints can be specified using a high-level declarative data mining query language and user interface.

The first four constraint types have already been addressed in earlier sections of this book and this chapter. In this section, we discuss the use of *rule constraints* to focus the mining task. This form of constraint-based mining allows users to describe the rules that they would like to uncover, thereby making the data mining process more *effective*. In addition, a sophisticated mining query optimizer can be used to exploit the constraints specified by the user, thereby making the mining process more *efficient*.

Constraint-based mining encourages interactive exploratory mining and analysis. In Section 7.3.1, you will study metarule-guided mining, where syntactic rule constraints are specified in the form of rule templates. Section 7.3.2 discusses the use of *pattern space pruning* (which prunes patterns being mined) and *data space pruning* (which prunes pieces of the data space for which further exploration cannot contribute to the discovery of patterns satisfying the constraints).

For pattern space pruning, we introduce three classes of properties that facilitate constraint-based search space pruning: *antimonotonicity, monotonicity*, and *succinctness*. We also discuss a special class of constraints, called *convertible constraints*, where by proper data ordering, the constraints can be pushed deep into the iterative mining process and have the same pruning power as monotonic or antimonotonic constraints. For data space pruning, we introduce two classes of properties—*data succinctness* and *data antimonotonicty*—and study how they can be integrated within a data mining process.

For ease of discussion, we assume that the user is searching for association rules. The procedures presented can be easily extended to the mining of correlation rules by adding a correlation measure of interestingness to the support-confidence framework.

7.3.1 Metarule-Guided Mining of Association Rules

"*How are metarules useful?*" Metarules allow users to specify the syntactic form of rules that they are interested in mining. The rule forms can be used as constraints to help improve the efficiency of the mining process. Metarules may be based on the analyst's experience, expectations, or intuition regarding the data or may be automatically generated based on the database schema.

Example 7.7 **Metarule-guided mining.** Suppose that as a market analyst for *AllElectronics* you have access to the data describing customers (e.g., customer age, address, and credit rating) as well as the list of customer transactions. You are interested in finding associations between customer traits and the items that customers buy. However, rather than finding *all* of the association rules reflecting these relationships, you are interested only in determining which pairs of customer traits promote the sale of office software. A metarule can be used to specify this information describing the form of rules you are interested in finding. An example of such a metarule is

$$P_1(X, Y) \wedge P_2(X, W) \Rightarrow buys(X, \text{"}office\ software\text{"}), \tag{7.11}$$

where P_1 and P_2 are **predicate variables** that are instantiated to attributes from the given database during the mining process, X is a variable representing a customer, and Y and W take on values of the attributes assigned to P_1 and P_2, respectively. Typically, a user will specify a list of attributes to be considered for instantiation with P_1 and P_2. Otherwise, a default set may be used.

In general, a metarule forms a hypothesis regarding the relationships that the user is interested in probing or confirming. The data mining system can then search for

rules that match the given metarule. For instance, Rule (7.12) matches or **complies with** Metarule (7.11):

$$age(X, \text{“30..39”}) \wedge income(X, \text{“41}K\text{..60}K\text{”}) \Rightarrow buys(X, \text{“}office\ software\text{”}). \quad (7.12)$$

■

“*How can metarules be used to guide the mining process?*” Let's examine this problem closely. Suppose that we wish to mine interdimensional association rules such as in Example 7.7. A metarule is a rule template of the form

$$P_1 \wedge P_2 \wedge \cdots \wedge P_l \Rightarrow Q_1 \wedge Q_2 \wedge \cdots \wedge Q_r, \quad (7.13)$$

where P_i $(i = 1, \ldots, l)$ and Q_j $(j = 1, \ldots, r)$ are either instantiated predicates or predicate variables. Let the number of predicates in the metarule be $p = l + r$. To find interdimensional association rules satisfying the template,

- We need to find all frequent p-predicate sets, L_p.
- We must also have the support or count of the l-predicate subsets of L_p to compute the confidence of rules derived from L_p.

This is a typical case of mining multidimensional association rules. By extending such methods using the constraint-pushing techniques described in the following section, we can derive efficient methods for metarule-guided mining.

7.3.2 Constraint-Based Pattern Generation: Pruning Pattern Space and Pruning Data Space

Rule constraints specify expected set/subset relationships of the variables in the mined rules, constant initiation of variables, and constraints on aggregate functions and other forms of constraints. Users typically employ their knowledge of the application or data to specify rule constraints for the mining task. These rule constraints may be used together with, or as an alternative to, metarule-guided mining. In this section, we examine rule constraints as to how they can be used to make the mining process more efficient. Let's study an example where rule constraints are used to mine hybrid-dimensional association rules.

Example 7.8 **Constraints for mining association rules.** Suppose that *AllElectronics* has a sales multidimensional database with the following interrelated relations:

- *item(item_ID, item_name, description, category, price)*
- *sales(transaction_ID, day, month, year, store_ID, city)*
- *trans_item(item_ID, transaction_ID)*

Here, the *item* table contains attributes *item_ID, item_name, description, category*, and *price*; the *sales* table contains attributes *transaction_ID day, month, year, store_ID*, and *city*; and the two tables are linked via the foreign key attributes, *item_ID* and *transaction_ID*, in the table *trans_item*.

Suppose our association mining query is "*Find the patterns or rules about the sales of which cheap items (where the sum of the prices is less than $10) may promote (i.e., appear in the same transaction) the sales of which expensive items (where the minimum price is $50), shown in the sales in Chicago in 2010.*"

This query contains the following four constraints: (1) $sum(I.price) < \$10$, where I represents the *item_ID* of a cheap item; (2) $min(J.price) \geq \$50$), where J represents the *item_ID* of an expensive item; (3) $T.city = Chicago$; and (4) $T.year = 2010$, where T represents a *transaction_ID*. For conciseness, we do not show the mining query explicitly here; however, the constraints' context is clear from the mining query semantics. ■

Dimension/level constraints and interestingness constraints can be applied after mining to filter out discovered rules, although it is generally more efficient and less expensive to use them *during* mining to help prune the search space. Dimension/level constraints were discussed in Section 7.2, and interestingness constraints, such as support, confidence, and correlation measures, were discussed in Chapter 6. Let's focus now on rule constraints.

"*How can we use rule constraints to prune the search space? More specifically, what kind of rule constraints can be 'pushed' deep into the mining process and still ensure the completeness of the answer returned for a mining query?*"

In general, an efficient frequent pattern mining processor can prune its search space during mining in two major ways: *pruning pattern search space* and *pruning data search space*. The former checks candidate patterns and decides whether a pattern can be pruned. Applying the Apriori property, it prunes a pattern if no superpattern of it can be generated in the remaining mining process. The latter checks the data set to determine whether the particular data piece will be able to contribute to the subsequent generation of satisfiable patterns (for a particular pattern) in the remaining mining process. If not, the data piece is pruned from further exploration. A constraint that may facilitate pattern space pruning is called a *pattern pruning constraint*, whereas one that can be used for data space pruning is called a *data pruning constraint*.

Pruning Pattern Space with Pattern Pruning Constraints

Based on how a constraint may interact with the pattern mining process, there are five categories of pattern mining constraints: (1) *antimonotonic,* (2) *monotonic,* (3) *succinct,* (4) *convertible,* and (5) *inconvertible*. For each category, we use an example to show its characteristics and explain how such kinds of constraints can be used in the mining process.

The first category of constraints is **antimonotonic**. Consider the rule constraint "$sum(I.price) \leq \$100$" of Example 7.8. Suppose we are using the Apriori framework, which explores itemsets of size k at the kth iteration. If the price summation of the items in a candidate itemset is no less than \$100, this itemset can be pruned from the search space, since adding more items into the set (assuming price is no less than zero) will only make it more expensive and thus will never satisfy the constraint. In other words, if an itemset does not satisfy this rule constraint, none of its supersets can satisfy the constraint. If a rule constraint obeys this property, it is **antimonotonic**. Pruning by antimonotonic constraints can be applied at each iteration of Apriori-style algorithms to help improve the efficiency of the overall mining process while guaranteeing completeness of the data mining task.

The Apriori property, which states that all nonempty subsets of a frequent itemset must also be frequent, is antimonotonic. If a given itemset does not satisfy minimum support, none of its supersets can. This property is used at each iteration of the Apriori algorithm to reduce the number of candidate itemsets examined, thereby reducing the search space for association rules.

Other examples of antimonotonic constraints include "$min(J.price) \geq \$50$," "$count(I) \leq 10$," and so on. Any itemset that violates either of these constraints can be discarded since adding more items to such itemsets can never satisfy the constraints. Note that a constraint such as "$avg(I.price) \leq \$10$" is not antimonotonic. For a given itemset that does not satisfy this constraint, a superset created by adding some (cheap) items may result in satisfying the constraint. Hence, pushing this constraint inside the mining process will not guarantee completeness of the data mining task. A list of SQL primitives–based constraints is given in the first column of Table 7.2. The antimonotonicity of the constraints is indicated in the second column. To simplify our discussion, only existence operators (e.g., $=$, $\in$, but not $\neq$, $\notin$) and comparison (or containment) operators with equality (e.g., $\leq$, $\subseteq$) are given.

The second category of constraints is **monotonic**. If the rule constraint in Example 7.8 were "$sum(I.price) \geq \$100$," the constraint-based processing method would be quite different. If an itemset I satisfies the constraint, that is, the sum of the prices in the set is no less than \$100, further addition of more items to I will increase cost and will always satisfy the constraint. Therefore, further testing of this constraint on itemset I becomes redundant. In other words, if an itemset satisfies this rule constraint, so do all of its supersets. If a rule constraint obeys this property, it is **monotonic**. Similar rule monotonic constraints include "$min(I.price) \leq \$10$," "$count(I) \geq 10$," and so on. The monotonicity of the list of SQL primitives–based constraints is indicated in the third column of Table 7.2.

The third category is **succinct constraints**. For this constraints category, we can *enumerate all and only those sets that are guaranteed to satisfy the constraint.* That is, if a rule constraint is **succinct**, we can directly generate precisely the sets that satisfy it, even before support counting begins. This avoids the substantial overhead of the generate-and-test paradigm. In other words, such constraints are *precounting prunable.* For example, the constraint "$min(J.price) \geq \$50$" in Example 7.8 is succinct because we can explicitly and precisely generate all the itemsets that satisfy the constraint.

Table 7.2 Characterization of Commonly Used SQL-Based Pattern Pruning Constraints

Constraint	*Antimonotonic*	*Monotonic*	*Succinct*
$v \in S$	no	yes	yes
$S \supseteq V$	no	yes	yes
$S \subseteq V$	yes	no	yes
$min(S) \leq v$	no	yes	yes
$min(S) \geq v$	yes	no	yes
$max(S) \leq v$	yes	no	yes
$max(S) \geq v$	no	yes	yes
$count(S) \leq v$	yes	no	weakly
$count(S) \geq v$	no	yes	weakly
$sum(S) \leq v$ ($\forall a \in S, a \geq 0$)	yes	no	no
$sum(S) \geq v$ ($\forall a \in S, a \geq 0$)	no	yes	no
$range(S) \leq v$	yes	no	no
$range(S) \geq v$	no	yes	no
$avg(S)\ \theta\ v, \theta \in \{\leq, \geq\}$	convertible	convertible	no
$support(S) \geq \xi$	yes	no	no
$support(S) \leq \xi$	no	yes	no
$all\_confidence(S) \geq \xi$	yes	no	no
$all\_confidence(S) \leq \xi$	no	yes	no

Specifically, such a set must consist of a nonempty set of items that have a price no less than \$50. It is of the form S, where $S \neq \emptyset$ is a subset of the set of all items with prices no less than \$50. Because there is a precise "formula" for generating all the sets satisfying a succinct constraint, there is no need to iteratively check the rule constraint during the mining process. The succinctness of the list of SQL primitives–based constraints is indicated in the fourth column of Table 7.2.[2]

The fourth category is **convertible constraints**. Some constraints belong to none of the previous three categories. However, if the items in the itemset are arranged in a particular order, the constraint may become monotonic or antimonotonic with regard to the frequent itemset mining process. For example, the constraint "$avg(I.price) \leq \$10$" is neither antimonotonic nor monotonic. However, if items in a transaction are added to an itemset in price-ascending order, the constraint becomes *antimonotonic*, because if an itemset I violates the constraint (i.e., with an average price greater than \$10), then further addition of more expensive items into the itemset will never make it

[2]For constraint $count(S) \leq v$ (and similarly for $count(S) \geq v$), we can have a member generation function based on a cardinality constraint (i.e., $\{X \mid X \subseteq Itemset \wedge |X| \leq v\}$). Member generation in this manner is of a different flavor and thus is called *weakly succinct*.

satisfy the constraint. Similarly, if items in a transaction are added to an itemset in price-descending order, it becomes *monotonic*, because if the itemset satisfies the constraint (i.e., with an average price no greater than \$10), then adding cheaper items into the current itemset will still make the average price no greater than \$10. Aside from "$avg(S) \leq v$" and "$avg(S) \geq v$," given in Table 7.2, there are many other convertible constraints such as "$variance(S) \geq v$" "$standard\_deviation(S) \geq v$," and so on.

Note that the previous discussion does not imply that every constraint is convertible. For example, "$sum(S)\,\theta\, v$," where $\theta \in \{\leq, \geq\}$ and each element in S could be of any real value, is not convertible. Therefore, there is yet a fifth category of constraints, called **inconvertible constraints**. The good news is that although there still exist some tough constraints that are not convertible, most simple SQL expressions with built-in SQL aggregates belong to one of the first four categories to which efficient constraint mining methods can be applied.

Pruning Data Space with Data Pruning Constraints

The second way of search space pruning in constraint-based frequent pattern mining is *pruning data space*. This strategy prunes pieces of data if they will not contribute to the subsequent generation of satisfiable patterns in the mining process. We consider two properties: *data succinctness* and *data antimonotonicity*.

Constraints are **data-succinct** if they can be used *at the beginning of a pattern mining process* to prune the data subsets that cannot satisfy the constraints. For example, if a mining query requires that the mined pattern must contain *digital camera*, then any transaction that does not contain *digital camera* can be pruned at the beginning of the mining process, which effectively reduces the data set to be examined.

Interestingly, many constraints are **data-antimonotonic** in the sense that *during the mining process*, if a data entry cannot satisfy a data-antimonotonic constraint based on the current pattern, then it can be pruned. We prune it because it will not be able to contribute to the generation of any superpattern of the current pattern in the remaining mining process.

Example 7.9 **Data antimonotonicity.** A mining query requires that $C_1 : sum(I.price) \geq \$100$, that is, the sum of the prices of the items in the mined pattern must be no less than \$100. Suppose that the current frequent itemset, S, does not satisfy constraint C_1 (say, because the sum of the prices of the items in S is \$50). If the remaining frequent items in a transaction T_i are such that, say, $\{i_2.price = \$5, i_5.price = \$10, i_8.price = \$20\}$, then T_i will not be able to make S satisfy the constraint. Thus, T_i cannot contribute to the patterns to be mined from S, and thus can be pruned.

Note that such pruning cannot be done at the beginning of the mining because at that time, we do not know yet if the total sum of the prices of all the items in T_i will be over \$100 (e.g., we may have $i_3.price = \$80$). However, during the iterative mining process, we may find some items (e.g., i_3) that are not frequent with S in the transaction data set, and thus they would be pruned. Therefore, such checking and pruning should be enforced at each iteration to reduce the data search space. ■

Notice that constraint C_1 is a monotonic constraint with respect to pattern space pruning. As we have seen, this constraint has very limited power for reducing the search space in pattern pruning. However, the same constraint can be used for effective reduction of the data search space.

For an antimonotonic constraint, such as $C_2 : sum(I.price) \leq \$100$, we can prune both pattern and data search spaces at the same time. Based on our study of pattern pruning, we already know that the current itemset can be pruned if the sum of the prices in it is over \$100 (since its further expansion can never satisfy C_2). At the same time, we can also prune any remaining items in a transaction T_i that cannot make the constraint C_2 valid. For example, if the sum of the prices of items in the current itemset S is \$90, any patterns over \$10 in the remaining frequent items in T_i can be pruned. If none of the remaining items in T_i can make the constraint valid, the entire transaction T_i should be pruned.

Consider pattern constraints that are neither antimonotonic nor monotonic such as "$C_3 : avg(I.price) \leq 10$." These can be data-antimonotonic because if the remaining items in a transaction T_i cannot make the constraint valid, then T_i can be pruned as well. Therefore, data-antimonotonic constraints can be quite useful for constraint-based data space pruning.

Notice that search space pruning by data antimonotonicity is confined only to a pattern growth–based mining algorithm because the pruning of a data entry is determined based on whether it can contribute to a specific pattern. Data antimonotonicity cannot be used for pruning the data space if the Apriori algorithm is used because the data are associated with all of the currently active patterns. At any iteration, there are usually many active patterns. A data entry that cannot contribute to the formation of the superpatterns of a given pattern may still be able to contribute to the superpattern of other active patterns. Thus, the power of data space pruning can be very limited for nonpattern growth–based algorithms.

7.4 Mining High-Dimensional Data and Colossal Patterns

The frequent pattern mining methods presented so far handle large data sets having a small number of dimensions. However, some applications may need to mine *high-dimensional data* (i.e., data with hundreds or thousands of dimensions). Can we use the methods studied so far to mine high-dimensional data? The answer is unfortunately negative because the search spaces of such typical methods grow exponentially with the number of dimensions.

Researchers have overcome this difficulty in two directions. One direction extends a pattern growth approach by further exploring the vertical data format to handle data sets with a large number of *dimensions* (also called *features* or *items*, e.g., genes) but a *small* number of *rows* (also called *transactions* or *tuples*, e.g., samples). This is useful in applications like the analysis of gene expressions in bioinformatics, for example, where we often need to analyze microarray data that contain a *large* number of genes

(e.g., 10,000 to 100,000) but only a *small* number of samples (e.g., 100 to 1000). The other direction develops a new mining methodology, called *Pattern-Fusion*, which mines *colossal patterns*, that is, patterns of very long length.

Let's first briefly examine the first direction, in particular, a pattern growth–based row enumeration approach. Its general philosophy is to explore the *vertical data format*, as described in Section 6.2.5, which is also known as **row enumeration**. Row enumeration differs from traditional column (i.e., item) enumeration (also known as the *horizontal data format*). In traditional column enumeration, the data set, D, is viewed as a set of rows, where each row consists of an itemset. In row enumeration, the data set is instead viewed as an itemset, each consisting of a set of *row_ID*s indicating where the item appears in the traditional view of D. The original data set, D, can easily be transformed into a transposed data set, T. A data set with a small number of rows but a large number of dimensions is then transformed into a transposed data set with a large number of rows but a small number of dimensions. Efficient pattern growth methods can then be developed on such relatively low-dimensional data sets. The details of such an approach are left as an exercise for interested readers.

The remainder of this section focuses on the second direction. We introduce Pattern-Fusion, a new mining methodology that mines *colossal patterns* (i.e., patterns of very long length). This method takes leaps in the pattern search space, leading to a good approximation of the complete set of colossal frequent patterns.

7.4.1 Mining Colossal Patterns by Pattern-Fusion

Although we have studied methods for mining frequent patterns in various situations, many applications have hidden patterns that are tough to mine, due mainly to their immense length or size. Consider bioinformatics, for example, where a common activity is DNA or microarray data analysis. This involves mapping and analyzing very long DNA and protein sequences. Researchers are more interested in finding large patterns (e.g., long sequences) than finding small ones since larger patterns usually carry more significant meaning. We call these large patterns *colossal* patterns, as distinguished from patterns with large support sets. Finding colossal patterns is challenging because incremental mining tends to get "trapped" by an explosive number of midsize patterns before it can even reach candidate patterns of large size. This is illustrated in Example 7.10.

Example 7.10 The challenge of mining colossal patterns. Consider a 40×40 square table where each row contains the integers 1 through 40 in increasing order. Remove the integers on the diagonal, and this gives a 40×39 table. Add 20 identical rows to the bottom of the table, where each row contains the integers 41 through 79 in increasing order, resulting in a 60×39 table (Figure 7.6). We consider each row as a transaction and set the minimum support threshold at 20. The table has an exponential number (i.e., $\binom{40}{20}$) of midsize closed/maximal frequent patterns of size 20, but only one that is colossal: $\alpha = (41, 42, \ldots, 79)$ of size 39. None of the frequent pattern mining algorithms that we have introduced so far can complete execution in a reasonable amount of time.

row/col	1	2	3	4	…	38	39
1	2	3	4	5	…	39	40
2	1	3	4	5	…	39	40
3	1	2	4	5	…	39	40
4	1	2	3	5	…	39	40
5	1	2	3	4	…	39	40
…	…	…	…	…	…	…	…
39	1	2	3	4	…	38	40
40	1	2	3	4	…	38	39
41	41	42	43	44	…	78	79
42	41	42	43	44	…	78	79
…	…	…	…	…	…	…	…
60	41	42	43	44	…	78	79

Figure 7.6 A simple colossal patterns example: The data set contains an exponential number of midsize patterns of size 20 but only one that is colossal, namely $(41, 42, \ldots, 79)$.

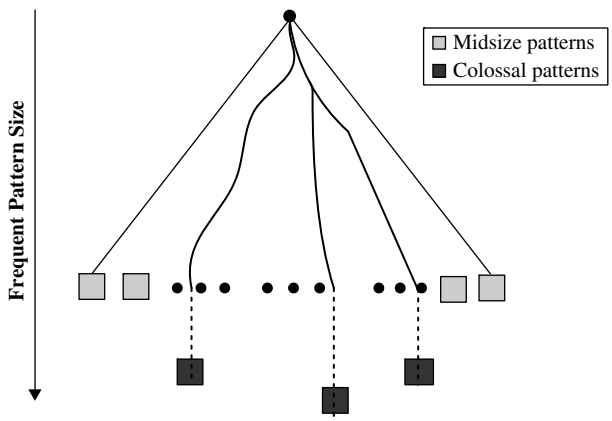

Figure 7.7 Synthetic data that contain some colossal patterns but exponentially many midsize patterns.

The pattern search space is similar to that in Figure 7.7, where midsize patterns largely outnumber colossal patterns. ■

All of the pattern mining strategies we have studied so far, such as Apriori and FP-growth, use an incremental growth strategy by nature, that is, they increase the length of candidate patterns by one at a time. Breadth-first search methods like Apriori cannot bypass the generation of an explosive number of midsize patterns generated,

making it impossible to reach colossal patterns. Even depth-first search methods like FP-growth can be easily trapped in a huge amount of subtrees before reaching colossal patterns. Clearly, a completely new mining methodology is needed to overcome such a hurdle.

A new mining strategy called *Pattern-Fusion* was developed, which fuses a small number of shorter frequent patterns into colossal pattern candidates. It thereby takes leaps in the pattern search space and avoids the pitfalls of both breadth-first and depth-first searches. This method finds a good approximation to the complete set of *colossal* frequent patterns.

The Pattern-Fusion method has the following major characteristics. First, it traverses the tree in a bounded-breadth way. Only a fixed number of patterns in a bounded-size candidate pool are used as starting nodes to search downward in the pattern tree. As such, it avoids the problem of exponential search space.

Second, Pattern-Fusion has the capability to identify "shortcuts" whenever possible. Each pattern's growth is not performed with one-item addition, but with an agglomeration of multiple patterns in the pool. These shortcuts direct Pattern-Fusion much more rapidly down the search tree toward the colossal patterns. Figure 7.8 conceptualizes this mining model.

As Pattern-Fusion is designed to give an approximation to the colossal patterns, a quality evaluation model is introduced to assess the patterns returned by the algorithm. An empirical study verifies that Pattern-Fusion is able to efficiently return high-quality results.

Let's examine the Pattern-Fusion method in more detail. First, we introduce the concept of **core pattern**. For a pattern α, an itemset $\beta \subseteq \alpha$ is said to be a τ*-core pattern* of α if $\frac{|D_\alpha|}{|D_\beta|} \geq \tau$, $0 < \tau \leq 1$, where $|D_\alpha|$ is the number of patterns containing α in database

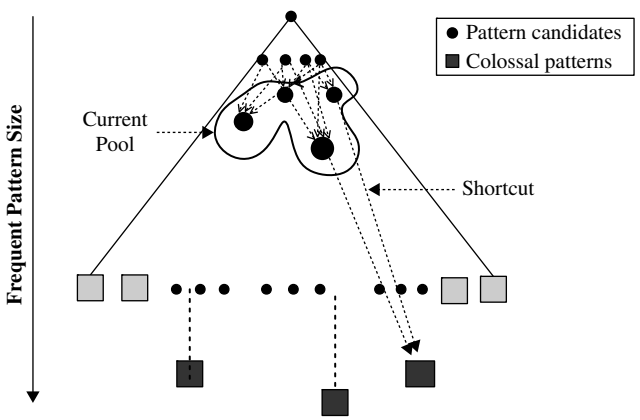

Figure 7.8 Pattern tree traversal: Candidates are taken from a pool of patterns, which results in shortcuts through pattern space to the colossal patterns.

D. τ is called the *core ratio*. A pattern α is (d, τ)-*robust* if d is the maximum number of items that can be removed from α for the resulting pattern to remain a τ-core pattern of α, that is,

$$d = \max_{\beta}\{|\alpha| - |\beta| | \beta \subseteq \alpha, \text{ and } \beta \text{ is a } \tau\text{-core pattern of } \alpha\}.$$

Example 7.11 **Core patterns.** Figure 7.9 shows a simple transaction database of four distinct transactions, each with 100 duplicates: $\{\alpha_1 = (abe),\ \alpha_2 = (bcf),\ \alpha_3 = (acf),\ \alpha_4 = (abcfe)\}$. If we set $\tau = 0.5$, then (ab) is a core pattern of α_1 because (ab) is contained only by α_1 and α_4. Therefore, $\frac{|D_{\alpha_1}|}{|D_{(ab)}|} = \frac{100}{200} \geq \tau$. α_1 is $(2, 0.5)$-robust while α_4 is $(4, 0.5)$-robust. The table also shows that larger patterns (e.g., $(abcfe)$) have far more core patterns than smaller ones (e.g., (bcf)). ■

From Example 7.11, we can deduce that large or colossal patterns have far more core patterns than smaller patterns do. Thus, a colossal pattern is more robust in the sense that *if a small number of items are removed from the pattern, the resulting pattern would have a similar support set.* The larger the pattern size, the more prominent this robustness. Such a robustness relationship between a colossal pattern and its corresponding core patterns can be extended to multiple levels. The lower-level core patterns of a colossal pattern are called **core descendants**.

Given a small c, a colossal pattern usually has far more core descendants of size c than a smaller pattern. This means that if we were to draw randomly from the complete set of patterns of size c, we would be more likely to pick a core descendant of a colossal pattern than that of a smaller pattern. In Figure 7.9, consider the complete set of patterns of size $c = 2$, which contains $\binom{5}{2} = 10$ patterns in total. For illustrative purposes, let's assume that the larger pattern, $abcef$, is colossal. The probability of being able to randomly draw a core descendant of $abcef$ is 0.9. Contrast this to the probability of randomly drawing a core descendent of smaller (noncolossal) patterns, which is at most 0.3. Therefore, a colossal pattern can be generated by merging a proper set of

Transactions (# of Transactions)	*Core Patterns* ($\tau = 0.5$)
(abe) (100)	(abe), (ab), (be), (ae), (e)
(bcf) (100)	(bcf), (bc), (bf)
(acf) (100)	(acf), (ac), (af)
$(abcef)$ (100)	(ab), (ac), (af), (ae), (bc), (bf), (be), (ce), (fe), (e), (abc), (abf), (abe), (ace), (acf), (afe), (bcf), (bce), (bfe), (cfe), $(abcf)$, $(abce)$, $(bcfe)$, $(acfe)$, $(abfe)$, $(abcef)$

Figure 7.9 A transaction database, which contains duplicates, and core patterns for each distinct transaction.

its core patterns. For instance, *abcef* can be generated by merging just two of its core patterns, *ab* and *cef*, instead of having to merge all of its 26 core patterns.

Now, let's see how these observations can help us leap through pattern space more directly toward colossal patterns. Consider the following scheme. First, generate a complete set of frequent patterns up to a user-specified small size, and then randomly pick a pattern, β. β will have a high probability of being a core-descendant of some colossal pattern, α. Identify all of α's core-descendants in this complete set, and merge them. This generates a much larger core-descendant of α, giving us the ability to leap along a path toward α in the core-pattern tree, T_α. In the same fashion we select K patterns. The set of larger core-descendants generated is the candidate pool for the next iteration.

A question arises: Given β, a core-descendant of a colossal pattern α, how can we find the other core-descendants of α? Given two patterns, α and β, the pattern distance between them is defined as $Dist(\alpha, \beta) = 1 - \frac{|D_\alpha \cap D_\beta|}{|D_\alpha \cup D_\beta|}$. Pattern distance satisfies the triangle inequality.

For a pattern, α, let C_α be the set of all its core patterns. It can be shown that C_α is bounded in metric space by a "ball" of diameter $r(\tau)$, where $r(\tau) = 1 - \frac{1}{2/\tau - 1}$. This means that given a core pattern $\beta \in C_\alpha$, we can identify all of α's core patterns in the current pool by posing a range query. Note that in the mining algorithm, each randomly drawn pattern could be a core-descendant of more than one colossal pattern, and as such, when merging the patterns found by the "ball," more than one larger core-descendant could be generated.

From this discussion, the Pattern-Fusion method is outlined in the following two phases:

1. **Initial Pool:** Pattern-Fusion assumes an initial pool of small frequent patterns is available. This is the complete set of frequent patterns up to a small size (e.g., 3). This initial pool can be mined with any existing efficient mining algorithm.
2. **Iterative Pattern-Fusion:** Pattern-Fusion takes as input a user-specified parameter, K, which is the maximum number of patterns to be mined. The mining process is iterative. At each iteration, K seed patterns are randomly picked from the current pool. For each of these K seeds, we find all the patterns within a ball of a size specified by τ. All the patterns in each "ball" are then fused together to generate a set of superpatterns. These superpatterns form a new pool. If the pool contains more than K patterns, the next iteration begins with this pool for the new round of random drawing. As the support set of every superpattern shrinks with each new iteration, the iteration process terminates.

Note that *Pattern-Fusion merges small subpatterns of a large pattern instead of incrementally-expanding patterns with single items.* This gives the method an advantage to circumvent midsize patterns and progress on a path leading to a potential colossal pattern. The idea is illustrated in Figure 7.10. Each point shown in the metric space

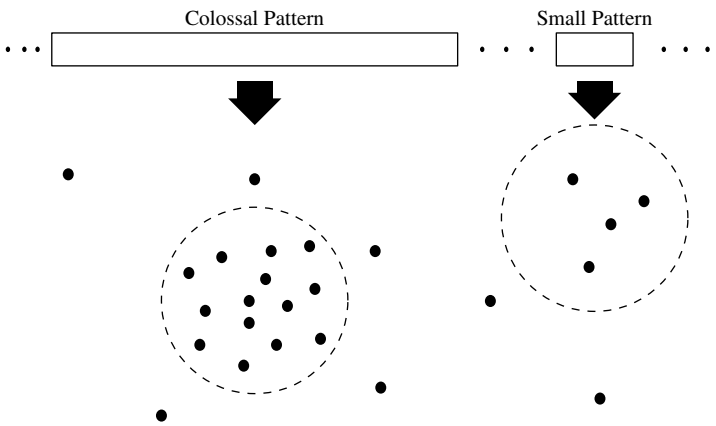

Figure 7.10 Pattern metric space: Each point represents a core pattern. The core patterns of a colossal pattern are denser than those of a small pattern, as shown within the dotted lines.

represents a core pattern. In comparison to a smaller pattern, a larger pattern has far more core patterns that are close to one another, all of which are bounded by a ball, as shown by the dotted lines. When drawing randomly from the initial pattern pool, we have a much higher probability of getting a core pattern of a large pattern, because the ball of a larger pattern is much denser.

It has been theoretically shown that Pattern-Fusion leads to a good approximation of colossal patterns. The method was tested on synthetic and real data sets constructed from program tracing data and microarray data. Experiments show that the method can find most of the colossal patterns with high efficiency.

7.5 Mining Compressed or Approximate Patterns

A major challenge in frequent pattern mining is the huge number of discovered patterns. Using a minimum support threshold to control the number of patterns found has limited effect. Too low a value can lead to the generation of an explosive number of output patterns, while too high a value can lead to the discovery of only commonsense patterns.

To reduce the huge set of frequent patterns generated in mining while maintaining high-quality patterns, we can instead mine a compressed or approximate set of frequent patterns. *Top-k most frequent closed patterns* were proposed to make the mining process concentrate on only the set of k most frequent patterns. Although interesting, they usually do not epitomize the k most representative patterns because of the uneven frequency distribution among itemsets. *Constraint-based mining* of frequent patterns (Section 7.3) incorporates user-specified constraints to filter out uninteresting patterns. Measures of

pattern/rule *interestingness* and *correlation* (Section 6.3) can also be used to help confine the search to patterns/rules of interest.

In this section, we look at two forms of "compression" of frequent patterns that build on the concepts of closed patterns and max-patterns. Recall from Section 6.2.6 that a *closed pattern* is a lossless compression of the set of frequent patterns, whereas a *max-pattern* is a lossy compression. In particular, Section 7.5.1 explores *clustering-based compression of frequent patterns*, which groups patterns together based on their similarity and frequency support. Section 7.5.2 takes a "*summarization*" *approach*, where the aim is to derive redundancy-aware top-k representative patterns that cover the whole set of (closed) frequent itemsets. The approach considers not only the representativeness of patterns but also their mutual independence to avoid redundancy in the set of generated patterns. The k representatives provide compact compression over the collection of frequent patterns, making them easier to interpret and use.

7.5.1 Mining Compressed Patterns by Pattern Clustering

Pattern compression can be achieved by pattern clustering. Clustering techniques are described in detail in Chapters 10 and 11. In this section, it is not necessary to know the fine details of clustering. Rather, you will learn how the concept of clustering can be applied to compress frequent patterns. Clustering is the automatic process of grouping like objects together, so that objects within a cluster are similar to one another and dissimilar to objects in other clusters. In this case, the objects are frequent patterns. The frequent patterns are clustered using a tightness measure called δ-cluster. A representative pattern is selected for each cluster, thereby offering a compressed version of the set of frequent patterns.

Before we begin, let's review some definitions. An itemset X is a **closed frequent itemset** in a data set D if X is frequent and there exists no proper super-itemset Y of X such that Y has the same support count as X in D. An itemset X is a **maximal frequent itemset** in data set D if X is frequent and there exists no super-itemset Y such that $X \subset Y$ and Y is frequent in D. Using these concepts alone is not enough to obtain a good representative compression of a data set, as we see in Example 7.12.

Example 7.12 Shortcomings of closed itemsets and maximal itemsets for compression. Table 7.3 shows a subset of frequent itemsets on a large data set, where *a, b, c, d, e, f* represent individual items. There are no closed itemsets here; therefore, we cannot use closed frequent itemsets to compress the data. The only maximal frequent itemset is P_3. However, we observe that itemsets P_2, P_3, and P_4 are significantly different with respect to their support counts. If we were to use P_3 to represent a compressed version of the data, we would lose this support count information entirely. From visual inspection, consider the two pairs (P_1, P_2) and (P_4, P_5). The patterns within each pair are very similar with respect to their support and expression. Therefore, intuitively, P_2, P_3, and P_4, collectively, should serve as a better compressed version of the data. ■

Table 7.3 Subset of Frequent Itemsets

ID	*Itemsets*	*Support*
P_1	$\{b, c, d, e\}$	205,227
P_2	$\{b, c, d, e, f\}$	205,211
P_3	$\{a, b, c, d, e, f\}$	101,758
P_4	$\{a, c, d, e, f\}$	161,563
P_5	$\{a, c, d, e\}$	161,576

So, let's see if we can find a way of clustering frequent patterns as a means of obtaining a compressed representation of them. We will need to define a good similarity measure, cluster patterns according to this measure, and then select and output only a *representative pattern* for each cluster. Since the set of closed frequent patterns is a lossless compression over the original frequent patterns set, it is a good idea to discover representative patterns over the collection of *closed* patterns.

We can use the following distance measure between closed patterns. Let P_1 and P_2 be two closed patterns. Their supporting transaction sets are $T(P_1)$ and $T(P_2)$, respectively. The **pattern distance** of P_1 and P_2, $Pat\_Dist(P_1, P_2)$, is defined as

$$Pat\_Dist(P_1, P_2) = 1 - \frac{|T(P_1) \cap T(P_2)|}{|T(P_1) \cup T(P_2)|}. \tag{7.14}$$

Pattern distance is a valid distance metric defined on the set of transactions. Note that it incorporates the *support* information of patterns, as desired previously.

Example 7.13 **Pattern distance.** Suppose P_1 and P_2 are two patterns such that $T(P_1) = \{t_1, t_2, t_3, t_4, t_5\}$ and $T(P_2) = \{t_1, t_2, t_3, t_4, t_6\}$, where t_i is a transaction in the database. The distance between P_1 and P_2 is $Pat\_Dist(P_1, P_2) = 1 - \frac{4}{6} = \frac{1}{3}$. ■

Now, let's consider the *expression* of patterns. Given two patterns A and B, we say B can be **expressed** by A if $O(B) \subset O(A)$, where $O(A)$ is the corresponding itemset of pattern A. Following this definition, assume patterns $P_1, P_2, \ldots, P_k$ are in the same cluster. The representative pattern P_r of the cluster should be able to *express* all the other patterns in the cluster. Clearly, we have $\cup_{i=1}^{k} O(P_i) \subseteq O(P_r)$.

Using the distance measure, we can simply apply a clustering method, such as k-means (Section 10.2), on the collection of frequent patterns. However, this introduces two problems. First, the quality of the clusters cannot be guaranteed; second, it may not be able to find a representative pattern for each cluster (i.e., the pattern P_r may not belong to the same cluster). To overcome these problems, this is where the concept of δ-cluster comes in, where δ $(0 \leq \delta \leq 1)$ measures the tightness of a cluster.

A pattern P is **δ-covered** by another pattern P' if $O(P) \subseteq O(P')$ and $Pat\_Dist(P, P') \leq \delta$. A set of patterns form a **δ-cluster** if there exists a representative pattern P_r such that for each pattern P in the set, P is δ-covered by P_r.

Note that according to the concept of δ-cluster, a pattern can belong to multiple clusters. Also, using δ-cluster, we only need to compute the distance between each pattern and the representative pattern of the cluster. Because a pattern P is δ-covered by a representative pattern P_r only if $O(P) \subseteq O(P_r)$, we can simplify the distance calculation by considering only the supports of the patterns:

$$Pat\_Dist(P, P_r) = 1 - \frac{|T(P) \cap T(P_r)|}{|T(P) \cup T(P_r)|} = 1 - \frac{|T(P_r)|}{|T(P)|}. \tag{7.15}$$

If we restrict the representative pattern to be frequent, then the number of representative patterns (i.e., clusters) is no less than the number of maximal frequent patterns. This is because a maximal frequent pattern can only be covered by itself. To achieve more succinct compression, we relax the constraints on representative patterns, that is, we allow the support of representative patterns to be *somewhat* less than *min_sup*.

For any representative pattern P_r, assume its support is k. Since it has to *cover* at least one frequent pattern (i.e., P) with support that is at least *min_sup*, we have

$$\delta \geq Pat\_Dist(P, P_r) = 1 - \frac{|T(P_r)|}{|T(P)|} \geq 1 - \frac{k}{min\_sup}. \tag{7.16}$$

That is, $k \geq (1 - \delta) \times min\_sup$. This is the minimum support for a representative pattern, denoted as min_sup_r.

Based on the preceding discussion, the pattern compression problem can be defined as follows: *Given a transaction database, a minimum support* min_sup, *and the cluster quality measure δ, the pattern compression problem is to find a set of representative patterns R such that for each frequent pattern P (with respect to* min_sup), *there is a representative pattern $P_r \in R$ (with respect to* min_sup_r), *which covers P, and the value of $|R|$ is minimized.*

Finding a minimum set of representative patterns is an NP-Hard problem. However, efficient methods have been developed that reduce the number of closed frequent patterns generated by orders of magnitude with respect to the original collection of closed patterns. The methods succeed in finding a high-quality compression of the pattern set.

7.5.2 Extracting Redundancy-Aware Top-*k* Patterns

Mining the top-k most frequent patterns is a strategy for reducing the number of patterns returned during mining. However, in many cases, frequent patterns are not mutually independent but often clustered in small regions. This is somewhat like finding 20 population centers in the world, which may result in cities clustered in a small number of countries rather than evenly distributed across the globe. Instead, most users would prefer to derive the k most interesting patterns, which are not only significant, but also mutually independent and containing little redundancy. A small set of

k representative patterns that have not only high significance but also low redundancy are called **redundancy-aware top-k patterns**.

Example 7.14 **Redundancy-aware top-k strategy versus other top-k strategies.** Figure 7.11 illustrates the intuition behind *redundancy-aware top-k patterns* versus *traditional top-k patterns* and *k-summarized patterns*. Suppose we have the frequent patterns set shown in Figure 7.11(a), where each circle represents a pattern of which the significance is colored in grayscale. The distance between two circles reflects the redundancy of the two corresponding patterns: The closer the circles are, the more redundant the respective patterns are to one another. Let's say we want to find three patterns that will best represent the given set, that is, $k = 3$. Which three should we choose?

Arrows are used to show the patterns chosen if using redundancy-aware top-k patterns (Figure 7.11b), traditional top-k patterns (Figure 7.11c), or k-summarized patterns (Figure 7.11d). In Figure 7.11(c), the **traditional top-k strategy** relies solely on significance: It selects the three most significant patterns to represent the set.

In Figure 7.11(d), the ***k*-summarized pattern strategy** selects patterns based solely on nonredundancy. It detects three clusters, and finds the most representative patterns to

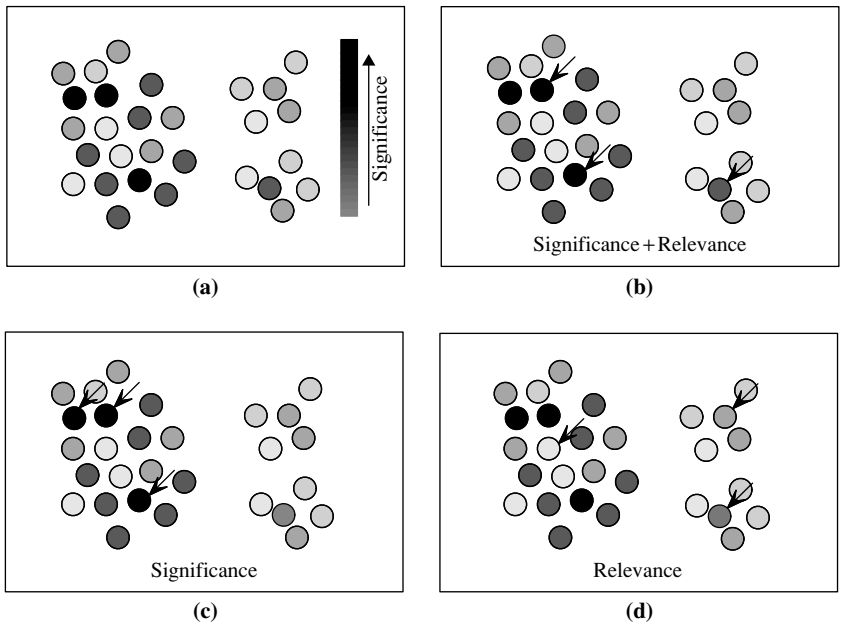

Figure 7.11 Conceptual view comparing top-k methodologies (where gray levels represent pattern significance, and the closer that two patterns are displayed, the more redundant they are to one another): (a) original patterns, (b) redundancy-aware top-k patterns, (c) traditional top-k patterns, and (d) k-summarized patterns.

be the "centermost'" pattern from each cluster. These patterns are chosen to represent the data. The selected patterns are considered "summarized patterns" in the sense that they represent or "provide a summary" of the clusters they stand for.

By contrast, in Figure 7.11(d) the **redundancy-aware top-*k* patterns** make a trade-off between significance and redundancy. The three patterns chosen here have high significance and low redundancy. Observe, for example, the two highly significant patterns that, based on their redundancy, are displayed next to each other. The redundancy-aware top-k strategy selects only one of them, taking into consideration that two would be redundant. To formalize the definition of redundancy-aware top-k patterns, we'll need to define the concepts of significance and redundancy. ■

A **significance measure** S is a function mapping a pattern $p \in \mathcal{P}$ to a real value such that $S(p)$ is the degree of interestingness (or usefulness) of the pattern p. In general, significance measures can be either objective or subjective. *Objective measures* depend only on the structure of the given pattern and the underlying data used in the discovery process. Commonly used objective measures include support, confidence, correlation, and *tf-idf* (or *term frequency* versus *inverse document frequency*), where the latter is often used in information retrieval. *Subjective* measures are based on user beliefs in the data. They therefore depend on the users who examine the patterns. A subjective measure is usually a relative score based on user prior knowledge or a background model. It often measures the unexpectedness of a pattern by computing its divergence from the background model. Let $S(p, q)$ be the **combined significance** of patterns p and q, and $S(p|q) = S(p, q) - S(q)$ be the **relative significance** of p given q. Note that the combined significance, $S(p, q)$, means the collective significance of two individual patterns p and q, not the significance of a single super pattern $p \cup q$.

Given the significance measure S, the **redundancy** R **between two patterns** p and q is defined as $R(p, q) = S(p) + S(q) - S(p, q)$. Subsequently, we have $S(p|q) = S(p) - R(p, q)$.

We assume that the combined significance of two patterns is no less than the significance of any individual pattern (since it is a collective significance of two patterns) and does not exceed the sum of two individual significance patterns (since there exists redundancy). That is, the redundancy between two patterns should satisfy

$$0 \leq R(p, q) \leq \min(S(p), S(q)). \tag{7.17}$$

The ideal redundancy measure $R(p, q)$ is usually hard to obtain. However, we can approximate redundancy using distance between patterns such as with the distance measure defined in Section 7.5.1.

The problem of finding redundancy-aware top-k patterns can thus be transformed into finding a k-pattern set that maximizes the marginal significance, which is a well-studied problem in information retrieval. In this field, a document has high marginal relevance if it is both relevant to the query and contains minimal marginal similarity to previously selected documents, where the marginal similarity is computed by choosing the most relevant selected document. Experimental studies have shown this method to be efficient and able to find high-significance and low-redundancy top-k patterns.

7.6 Pattern Exploration and Application

For discovered frequent patterns, is there any way the mining process can return additional information that will help us to better understand the patterns? What kinds of applications exist for frequent pattern mining? These topics are discussed in this section. Section 7.6.1 looks at the automated generation of **semantic annotations** for frequent patterns. These are dictionary-like annotations. They provide semantic information relating to patterns, based on the context and usage of the patterns, which aids in their understanding. Semantically similar patterns also form part of the annotation, providing a more direct connection between discovered patterns and any other patterns already known to the users.

Section 7.6.2 presents an overview of applications of frequent pattern mining. While the applications discussed in Chapter 6 and this chapter mainly involve market basket analysis and correlation analysis, there are many other areas in which frequent pattern mining is useful. These range from data preprocessing and classification to clustering and the analysis of complex data.

7.6.1 Semantic Annotation of Frequent Patterns

Pattern mining typically generates a huge set of frequent patterns without providing enough information to interpret the meaning of the patterns. In the previous section, we introduced pattern processing techniques to shrink the size of the output set of frequent patterns such as by extracting redundancy-aware top-k patterns or compressing the pattern set. These, however, do not provide any semantic interpretation of the patterns. It would be helpful if we could also generate semantic annotations for the frequent patterns found, which would help us to better understand the patterns.

"*What is an appropriate semantic annotation for a frequent pattern?*" Think about what we find when we look up the meaning of terms in a dictionary. Suppose we are looking up the term *pattern*. A dictionary typically contains the following components to explain the term:

1. *A set of definitions*, such as "a decorative design, as for wallpaper, china, or textile fabrics, etc.; a natural or chance configuration"
2. *Example sentences*, such as "*patterns* of frost on the window; the behavior *patterns* of teenagers, . . . "
3. *Synonyms from a thesaurus*, such as "model, archetype, design, exemplar, motif, ..."

Analogically, what if we could extract similar types of semantic information and provide such structured annotations for frequent patterns? This would greatly help users in interpreting the meaning of patterns and in deciding on how or whether to further explore them. Unfortunately, it is infeasible to provide such precise semantic definitions for patterns without expertise in the domain. Nevertheless, we can explore how to *approximate* such a process for frequent pattern mining.

Pattern: "{*frequent, pattern*}"
context indicators:
"mining," "constraint," "Apriori," "FP-growth,"
"rakesh agrawal," "jiawei han," ...
representative transactions:
1) mining *frequent patterns* without candidate ...
2) ... mining closed *frequent* graph *patterns*
semantically similar patterns:
"{*frequent, sequential, pattern*}," "{*graph, pattern*}"
"{*maximal, pattern*}," "{*frequent, closed, pattern*}," ...

Figure 7.12 Semantic annotation of the pattern "{*frequent, pattern*}."

In general, the hidden meaning of a pattern can be inferred from patterns with similar meanings, data objects co-occurring with it, and transactions in which the pattern appears. Annotations with such information are analogous to dictionary entries, which can be regarded as annotating each term with structured semantic information. Let's examine an example.

Example 7.15 Semantic annotation of a frequent pattern. Figure 7.12 shows an example of a semantic annotation for the pattern "{*frequent, pattern*}." This dictionary-like annotation provides semantic information related to "{*frequent, pattern*}," consisting of its strongest *context indicators*, the most *representative data transactions*, and the most *semantically similar patterns*. This kind of semantic annotation is similar to natural language processing. The semantics of a word can be inferred from its context, and words sharing similar contexts tend to be semantically similar. The context indicators and the representative transactions provide a view of the context of the pattern from different angles to help users understand the pattern. The semantically similar patterns provide a more direct connection between the pattern and any other patterns already known to the users. ■

"*How can we perform automated semantic annotation for a frequent pattern?*" The key to high-quality semantic annotation of a frequent pattern is the successful context modeling of the pattern. For context modeling of a pattern, p, consider the following.

- A **context unit** is a basic object in a database, D, that carries semantic information and co-occurs with at least one frequent pattern, p, in at least one transaction in D. A context unit can be an item, a pattern, or even a transaction, depending on the specific task and data.
- The **context of a pattern**, p, is a selected set of weighted context units (referred to as **context indicators**) in the database. It carries semantic information, and co-occurs with a frequent pattern, p. The context of p can be modeled using a vector space model, that is, the context of p can be represented as $C(p) = \langle w(u_1),$

$w(u_2), \ldots, w(u_n)\rangle$, where $w(u_i)$ is a weight function of term u_i. A transaction t is represented as a vector $\langle v_1, v_2, \ldots, v_m\rangle$, where $v_i = 1$ if and only if $v_i \in t$, otherwise $v_i = 0$.

Based on these concepts, we can define the basic task of **semantic pattern annotation** as follows:

1. Select context units and design a strength weight for each unit to model the contexts of frequent patterns.
2. Design similarity measures for the contexts of two patterns, and for a transaction and a pattern context.
3. For a given frequent pattern, extract the most significant context indicators, representative transactions, and semantically similar patterns to construct a structured annotation.

"*Which context units should we select as context indicators?*" Although a context unit can be an item, a transaction, or a pattern, typically, frequent patterns provide the most semantic information of the three. There are usually a large number of frequent patterns associated with a pattern, p. Therefore, we need a systematic way to select only the important and nonredundant frequent patterns from a large pattern set.

Considering that the closed patterns set is a lossless compression of frequent pattern sets, we can first derive the closed patterns set by applying efficient closed pattern mining methods. However, as discussed in Section 7.5, a closed pattern set is not compact enough, and pattern compression needs to be performed. We could use the pattern compression methods introduced in Section 7.5.1 or explore alternative compression methods such as microclustering using the Jaccard coefficient (Chapter 2) and then selecting the most representative patterns from each cluster.

"*How, then, can we assign weights for each context indicator?*" A good weighting function should obey the following properties: (1) the best semantic indicator of a pattern, p, is itself, (2) assign the same score to two patterns if they are equally strong, and (3) if two patterns are independent, neither can indicate the meaning of the other. The meaning of a pattern, p, can be inferred from either the appearance or absence of indicators.

Mutual information is one of several possible weighting functions. It is widely used in information theory to measure the mutual independency of two random variables. Intuitively, it measures how much information a random variable tells about the other. Given two frequent patterns, p_α and p_β, let $X = \{0, 1\}$ and $Y = \{0, 1\}$ be two random variables representing the appearance of p_α and p_β, respectively. **Mutual information** $I(X; Y)$ is computed as

$$I(X; Y) = \sum_{x \in X} \sum_{y \in Y} P(x, y) log \frac{P(x, y)}{P(x)P(y)}, \tag{7.18}$$

where $P(x=1,y=1)=\frac{|D_\alpha \cap D_\beta|}{|D|}$, $P(x=0,y=1)=\frac{|D_\beta|-|D_\alpha \cap D_\beta|}{|D|}$, $P(x=1,y=0)=\frac{|D_\alpha|-|D_\alpha \cap D_\beta|}{|D|}$, and $P(x=0,y=0)=\frac{|D|-|D_\alpha \cup D_\beta|}{|D|}$. Standard Laplace smoothing can be used to avoid zero probability.

Mutual information favors strongly correlated units and thus can be used to model the indicative strength of the context units selected. With context modeling, pattern annotation can be accomplished as follows:

1. To extract the most significant context indicators, we can use cosine similarity (Chapter 2) to measure the semantic similarity between pairs of context vectors, rank the context indicators by the weight strength, and extract the strongest ones.
2. To extract representative transactions, represent each transaction as a context vector. Rank the transactions with semantic similarity to the pattern p.
3. To extract semantically similar patterns, rank each frequent pattern, p, by the semantic similarity between their context models and the context of p.

Based on these principles, experiments have been conducted on large data sets to generate semantic annotations. Example 7.16 illustrates one such experiment.

Example 7.16 **Semantic annotations generated for frequent patterns from the DBLP Computer Science Bibliography.** Table 7.4 shows annotations generated for frequent patterns from a portion of the DBLP data set.[3] The DBLP data set contains papers from the proceedings of 12 major conferences in the fields of database systems, information retrieval, and data mining. Each transaction consists of two parts: the authors and the title of the corresponding paper.

Consider two types of patterns: (1) *frequent author* or *coauthorship*, each of which is a frequent itemset of authors, and (2) *frequent title terms*, each of which is a frequent sequential pattern of the title words. The method can automatically generate dictionary-like annotations for different kinds of frequent patterns. For frequent itemsets like coauthorship or single authors, the strongest context indicators are usually the other coauthors and discriminative title terms that appear in their work. The semantically similar patterns extracted also reflect the authors and terms related to their work. However, these similar patterns may not even co-occur with the given pattern in a paper. For example, the patterns "*timos_k_selli*," "*ramakrishnan_srikant*," and so on, do not co-occur with the pattern "*christos_faloutsos*," but are extracted because their contexts are similar since they all are database and/or data mining researchers; thus the annotation is meaningful.

For the title term "*information retrieval*," which is a sequential pattern, its strongest context indicators are usually the authors who tend to use the term in the titles of their papers, or the terms that tend to coappear with it. Its semantically similar patterns usually provide interesting concepts or descriptive terms, which are close in meaning (e.g., "*information retrieval* → *information filter*)."

[3] *www.informatik.uni-trier.de/~ley/db/*.

Table 7.4 Annotations Generated for Frequent Patterns in the DBLP Data Set

Pattern	*Type*	*Annotations*
christos_faloutsos	Context indicator	spiros_papadimitriou; fast; use fractal; graph; use correlate
	Representative transactions	multi-attribute hash use gray code
	Representative transactions	recovery latent time-series observe sum network tomography particle filter
	Representative transactions	index multimedia database tutorial
	Semantic similar patterns	spiros_papadimitriou&christos_faloutsos; spiros_papadimitriou; flip_korn; timos_k_selli;
		ramakrishnan_srikant; ramakrishnan_srikant&rakesh_agrawal
information retrieval	Context indicator	w_bruce_croft; web information; monika_ rauch_henzinger;
		james_p_callan; full-text
	Representative transactions	web information retrieval
	Representative transactions	language model information retrieval
	Semantic similar patterns	information use; web information; probabilistic information; information filter;
		text information

In both scenarios, the representative transactions extracted give us the titles of papers that effectively capture the meaning of the given patterns. The experiment demonstrates the effectiveness of semantic pattern annotation to generate a dictionary-like annotation for frequent patterns, which can help a user understand the meaning of annotated patterns. ■

The context modeling and semantic analysis method presented here is general and can deal with any type of frequent patterns with context information. Such semantic annotations can have many other applications such as ranking patterns, categorizing and clustering patterns with semantics, and summarizing databases. Applications of the pattern context model and semantical analysis method are also not limited to pattern annotation; other example applications include pattern compression, transaction clustering, pattern relations discovery, and pattern synonym discovery.

7.6.2 Applications of Pattern Mining

We have studied many aspects of frequent pattern mining, with topics ranging from efficient mining algorithms and the diversity of patterns to pattern interestingness, pattern

compression/approximation, and semantic pattern annotation. Let's take a moment to consider why this field has generated so much attention. What are some of the application areas in which frequent pattern mining is useful? This section presents an overview of applications for frequent pattern mining. We have touched on several application areas already, such as market basket analysis and correlation analysis, yet frequent pattern mining can be applied to many other areas as well. These range from data preprocessing and classification to clustering and the analysis of complex data.

To summarize, frequent pattern mining is a data mining task that discovers patterns that occur frequently together and/or have some distinctive properties that distinguish them from others, often disclosing something inherent and valuable. The patterns may be itemsets, subsequences, substructures, or values. The task also includes the discovery of rare patterns, revealing items that occur very rarely together yet are of interest. Uncovering frequent patterns and rare patterns leads to many broad and interesting applications, described as follows.

Pattern mining is widely used for **noise filtering and data cleaning** as **preprocessing** in many data-intensive applications. We can use it to analyze microarray data, for instance, which typically consists of tens of thousands of dimensions (e.g., representing genes). Such data can be rather noisy. Frequent pattern data mining can help us distinguish between what is noise and what isn't. We may assume that items that occur frequently together are less likely to be random noise and should not be filtered out. On the other hand, those that occur very frequently (similar to stopwords in text documents) are likely indistinctive and may be filtered out. Frequent pattern mining can help in background information identification and noise reduction.

Pattern mining often helps in the **discovery of inherent structures and clusters hidden in the data**. Given the DBLP data set, for instance, frequent pattern mining can easily find interesting clusters like coauthor clusters (by examining authors who frequently collaborate) and conference clusters (by examining the sharing of many common authors and terms). Such structure or cluster discovery can be used as preprocessing for more sophisticated data mining.

Although there are numerous classification methods (Chapters 8 and 9), research has found that frequent patterns can be used as building blocks in the construction of high-quality classification models, hence called **pattern-based classification**. The approach is successful because (1) the appearance of very infrequent item(s) or itemset(s) can be caused by random noise and may not be reliable for model construction, yet a relatively frequent pattern often carries more information gain for constructing more reliable models; (2) patterns in general (i.e., itemsets consisting of multiple attributes) usually carry more information gain than a single attribute (feature); and (3) the patterns so generated are often intuitively understandable and easy to explain. Recent research has reported several methods that mine interesting, frequent, and discriminative patterns and use them for effective classification. Pattern-based classification methods are introduced in Chapter 9.

Frequent patterns can also be used effectively for **subspace clustering in high-dimensional space**. Clustering is challenging in high-dimensional space, where the distance between two objects is often difficult to measure. This is because such a distance is dominated by the different sets of dimensions in which the objects are residing.

Thus, instead of clustering objects in their full high-dimensional spaces, it can be more meaningful to find clusters in certain subspaces. Recently, researchers have developed subspace-based pattern growth methods that cluster objects based on their common frequent patterns. They have shown that such methods are effective for clustering microarray-based gene expression data. Subspace clustering methods are discussed in Chapter 11.

Pattern analysis is useful in the **analysis of spatiotemporal data, time-series data, image data, video data, and multimedia data**. An area of *spatiotemporal data analysis* is the discovery of **colocation patterns**. These, for example, can help determine if a certain disease is geographically colocated with certain objects like a well, a hospital, or a river. In *time-series data analysis*, researchers have discretized time-series values into multiple intervals (or levels) so that tiny fluctuations and value differences can be ignored. The data can then be summarized into sequential patterns, which can be indexed to facilitate similarity search or comparative analysis. In *image analysis and pattern recognition*, researchers have also identified frequently occurring visual fragments as "visual words," which can be used for effective clustering, classification, and comparative analysis.

Pattern mining has also been used for the **analysis of sequence or structural data** such as trees, graphs, subsequences, and networks. In software engineering, researchers have identified consecutive or gapped subsequences in program execution as sequential patterns that help identify software bugs. Copy-and-paste bugs in large software programs can be identified by extended sequential pattern analysis of source programs. Plagiarized software programs can be identified based on their essentially identical program flow/loop structures. Authors' commonly used sentence substructures can be identified and used to distinguish articles written by different authors.

Frequent and discriminative patterns can be used as primitive **indexing structures** (known as graph indices) to help search large, complex, structured data sets and networks. These support a similarity search in graph-structured data such as chemical compound databases or XML-structured databases. Such patterns can also be used for data compression and summarization.

Furthermore, frequent patterns have been used in **recommender systems**, where people can find correlations, clusters of customer behaviors, and classification models based on commonly occurring or discriminative patterns (Chapter 13).

Finally, studies on efficient computation methods in pattern mining mutually enhance many other studies on **scalable computation**. For example, the computation and materialization of **iceberg cubes** using the BUC and Star-Cubing algorithms (Chapter 5) respectively share many similarities to computing frequent patterns by the Apriori and FP-growth algorithms (Chapter 6).

7.7 Summary

- The **scope** of frequent pattern mining research reaches far beyond the basic concepts and methods introduced in Chapter 6 for mining frequent itemsets and associations. This chapter presented a road map of the field, where topics are organized

with respect to the kinds of patterns and rules that can be mined, mining methods, and applications.

- In addition to mining for basic frequent itemsets and associations, **advanced forms of patterns** can be mined such as multilevel associations and multidimensional associations, quantitative association rules, rare patterns, and negative patterns. We can also mine high-dimensional patterns and compressed or approximate patterns.
- **Multilevel associations** involve data at more than one abstraction level (e.g., "*buys computer*" and "*buys laptop*"). These may be mined using multiple minimum support thresholds. **Multidimensional associations** contain more than one dimension. Techniques for mining such associations differ in how they handle repetitive predicates. **Quantitative association rules** involve quantitative attributes. Discretization, clustering, and statistical analysis that discloses exceptional behavior can be integrated with the pattern mining process.
- **Rare patterns** occur rarely but are of special interest. **Negative patterns** are patterns with components that exhibit negatively correlated behavior. Care should be taken in the definition of negative patterns, with consideration of the null-invariance property. Rare and negative patterns may highlight exceptional behavior in the data, which is likely of interest.
- **Constraint-based mining** strategies can be used to help direct the mining process toward patterns that match users' intuition or satisfy certain constraints. Many user-specified constraints can be pushed deep into the mining process. Constraints can be categorized into **pattern-pruning** and **data-pruning** constraints. Properties of such constraints include *monotonicity, antimonotonicity*, *data-antimonotonicity*, and *succinctness*. Constraints with such properties can be properly incorporated into efficient pattern mining processes.
- Methods have been developed for mining patterns in **high-dimensional space**. This includes a pattern growth approach based on *row enumeration* for mining data sets where the number of dimensions is large and the number of data tuples is small (e.g., for microarray data), as well as mining **colossal patterns** (i.e., patterns of very long length) by a *Pattern-Fusion* method.
- To reduce the number of patterns returned in mining, we can instead mine compressed patterns or approximate patterns. *Compressed patterns* can be mined with representative patterns defined based on the concept of clustering, and *approximate patterns* can be mined by extracting **redundancy-aware top-*k* patterns** (i.e., a small set of *k*-representative patterns that have not only high significance but also low redundancy with respect to one another).
- **Semantic annotations** can be generated to help users understand the meaning of the frequent patterns found, such as for textual terms like "{*frequent, pattern*}." These are dictionary-like annotations, providing semantic information relating to the term. This information consists of *context indicators* (e.g., terms indicating the context of that pattern), the most *representative data transactions* (e.g., fragments or sentences

containing the term), and the most *semantically similar patterns* (e.g., "{*maximal*, *pattern*}" is semantically similar to "{*frequent*, *pattern*}"). The annotations provide a view of the pattern's context from different angles, which aids in their understanding.

- Frequent pattern mining has many diverse applications, ranging from pattern-based data cleaning to pattern-based classification, clustering, and outlier or exception analysis. These methods are discussed in the subsequent chapters in this book.

7.8 Exercises

7.1 Propose and outline a **level-shared mining** approach to mining multilevel association rules in which each item is encoded by its level position. Design it so that an initial scan of the database collects the count for each item *at each concept level*, identifying frequent and subfrequent items. Comment on the processing cost of mining multilevel associations with this method in comparison to mining single-level associations.

7.2 Suppose, as manager of a chain of stores, you would like to use sales transactional data to analyze the effectiveness of your store's advertisements. In particular, you would like to study how specific factors influence the effectiveness of advertisements that announce a particular category of items on sale. The factors to study are the *region* in which customers live and the *day-of-the-week* and *time-of-the-day* of the ads. Discuss how to design an efficient method to mine the transaction data sets and explain how **multidimensional** and **multilevel mining** methods can help you derive a good solution.

7.3 **Quantitative association rules** may disclose exceptional behaviors within a data set, where "exceptional" can be defined based on statistical theory. For example, Section 7.2.3 shows the association rule

$$sex = female \Rightarrow mean\_wage = \$7.90/hr \ (overall\_mean\_wage = \$9.02/hr),$$

which suggests an exceptional pattern. The rule states that the average wage for females is only \$7.90 per hour, which is a significantly lower wage than the overall average of \$9.02 per hour. Discuss how such quantitative rules can be discovered systematically and efficiently in large data sets with quantitative attributes.

7.4 In multidimensional data analysis, it is interesting to extract pairs of *similar* cell characteristics associated with substantial changes in measure in a data cube, where cells are considered *similar* if they are related by roll-up (i.e, *ancestors*), drill-down (i.e, *descendants*), or 1-D mutation (i.e, *siblings*) operations. Such an analysis is called **cube gradient analysis**.

Suppose the measure of the cube is *average*. A user poses a set of *probe cells* and would like to find their corresponding sets of *gradient cells*, each of which satisfies a certain gradient threshold. For example, find the set of corresponding gradient cells that have an average sale price greater than 20% of that of the given probe cells. Develop an algorithm than mines the set of constrained gradient cells efficiently in a large data cube.

7.5 Section 7.2.4 presented various ways of defining negatively correlated patterns. Consider Definition 7.3: "Suppose that itemsets X and Y are both frequent, that is, $sup(X) \geq min\_sup$ and $sup(Y) \geq min\_sup$, where min_sup is the minimum support threshold. If $(P(X|Y) + P(Y|X))/2 < \epsilon$, where ϵ is a negative pattern threshold, then pattern $X \cup Y$ is a **negatively correlated pattern**." Design an efficient pattern growth algorithm for mining the set of negatively correlated patterns.

7.6 Prove that each entry in the following table correctly characterizes its corresponding **rule constraint** for frequent itemset mining.

	Rule Constraint	*Antimonotonic*	*Monotonic*	*Succinct*
(a)	$v \in S$	no	yes	yes
(b)	$S \subseteq V$	yes	no	yes
(c)	$min(S) \leq v$	no	yes	yes
(d)	$range(S) \leq v$	yes	no	no
(e)	$variance(S) \leq v$	convertible	convertible	no

7.7 The price of each item in a store is non-negative. The store manager is only interested in rules of certain forms, using the constraints given in (a)–(b). For each of the following cases, identify the kinds of **constraints** they represent and briefly discuss how to mine such association rules using **constraint-based pattern mining**.

(a) Containing at least one Blu-ray DVD movie.

(b) Containing items with a sum of the prices that is less than \$150.

(c) Containing one free item and other items with a sum of the prices that is at least \$200.

(d) Where the average price of all the items is between \$100 and \$500.

7.8 Section 7.4.1 introduced a core Pattern-Fusion method for **mining high-dimensional data**. Explain why a long pattern, if one exists in the data set, is likely to be discovered by this method.

7.9 Section 7.5.1 defined a **pattern distance measure** between closed patterns P_1 and P_2 as

$$Pat\_Dist(P_1, P_2) = 1 - \frac{|T(P_1) \cap T(P_2)|}{|T(P_1) \cup T(P_2)|},$$

where $T(P_1)$ and $T(P_2)$ are the supporting transaction sets of P_1 and P_2, respectively. Is this a valid distance metric? Show the derivation to support your answer.

7.10 Association rule mining often generates a large number of rules, many of which may be similar, thus not containing much novel information. Design an efficient algorithm that **compresses** a large set of patterns into a small compact set. Discuss whether your mining method is robust under different pattern similarity definitions.

7.11 Frequent pattern mining may generate many superfluous patterns. Therefore, it is important to develop methods that mine compressed patterns. Suppose a user would like to obtain only k patterns (where k is a small integer). Outline an efficient method that generates the ***k* most representative patterns**, where more distinct patterns are preferred over very similar patterns. Illustrate the effectiveness of your method using a small data set.

7.12 It is interesting to generate **semantic annotations** for mined patterns. Section 7.6.1 presented a pattern annotation method. Alternative methods are possible, such as by utilizing type information. In the DBLP data set, for example, authors, conferences, terms, and papers form multi-typed data. Develop a method for automated semantic pattern annotation that makes good use of typed information.

7.9 Bibliographic Notes

This chapter described various ways in which the basic techniques of frequent itemset mining (presented in Chapter 6) have been extended. One line of extension is mining multilevel and multidimensional association rules. Multilevel association mining was studied in Srikant and Agrawal [SA95] and Han and Fu [HF95]. In Srikant and Agrawal [SA95], such mining was studied in the context of *generalized association rules*, and an R-interest measure was proposed for removing redundant rules. Mining multidimensional association rules using static discretization of quantitative attributes and data cubes was studied by Kamber, Han, and Chiang [KHC97].

Another line of extension is to mine patterns on numeric attributes. Srikant and Agrawal [SA96] proposed a nongrid-based technique for mining quantitative association rules, which uses a measure of partial completeness. Mining quantitative association rules based on rule clustering was proposed by Lent, Swami, and Widom [LSW97]. Techniques for mining quantitative rules based on x-monotone and rectilinear regions were presented by Fukuda, Morimoto, Morishita, and Tokuyama [FMMT96] and Yoda, Fukuda, Morimoto, et al. [YFM+97]. Mining (distance-based) association rules over interval data was proposed by Miller and Yang [MY97]. Aumann and Lindell [AL99] studied the mining of quantitative association rules based on a statistical theory to present only those rules that deviate substantially from normal data.

Mining rare patterns by pushing group-based constraints was proposed by Wang, He, and Han [WHH00]. Mining negative association rules was discussed by Savasere, Omiecinski, and Navathe [SON98] and by Tan, Steinbach, and Kumar [TSK05].

Constraint-based mining directs the mining process toward patterns that are likely of interest to the user. The use of metarules as syntactic or semantic filters defining the form of interesting single-dimensional association rules was proposed in Klemettinen, Mannila, Ronkainen, et al. [KMR+94]. Metarule-guided mining, where the metarule consequent specifies an action (e.g., Bayesian clustering or plotting) to be applied to the data satisfying the metarule antecedent, was proposed in Shen, Ong, Mitbander,

and Zaniolo [SOMZ96]. A relation-based approach to metarule-guided mining of association rules was studied in Fu and Han [FH95].

Methods for constraint-based mining using pattern pruning constraints were studied by Ng, Lakshmanan, Han, and Pang [NLHP98]; Lakshmanan, Ng, Han, and Pang [LNHP99]; and Pei, Han, and Lakshmanan [PHL01]. Constraint-based pattern mining by data reduction using data pruning constraints was studied by Bonchi, Giannotti, Mazzanti, and Pedreschi [BGMP03] and Zhu, Yan, Han, and Yu [ZYHY07]. An efficient method for mining constrained correlated sets was given in Grahne, Lakshmanan, and Wang [GLW00]. A dual mining approach was proposed by Bucila, Gehrke, Kifer, and White [BGKW03]. Other ideas involving the use of templates or predicate constraints in mining have been discussed in Anand and Kahn [AK93]; Dhar and Tuzhilin [DT93]; Hoschka and Klösgen [HK91]; Liu, Hsu, and Chen [LHC97]; Silberschatz and Tuzhilin [ST96]; and Srikant, Vu, and Agrawal [SVA97].

Traditional pattern mining methods encounter challenges when mining high-dimensional patterns, with applications like bioinformatics. Pan, Cong, Tung, et al. [PCT+03] proposed CARPENTER, a method for finding closed patterns in high-dimensional biological data sets, which integrates the advantages of vertical data formats and pattern growth methods. Pan, Tung, Cong, and Xu [PTCX04] proposed COBBLER, which finds frequent closed itemsets by integrating row enumeration with column enumeration. Liu, Han, Xin, and Shao [LHXS06] proposed TDClose to mine frequent closed patterns in high-dimensional data by starting from the maximal rowset, integrated with a row-enumeration tree. It uses the pruning power of the minimum support threshold to reduce the search space. For mining rather long patterns, called *colossal patterns*, Zhu, Yan, Han, et al. [ZYH+07] developed a core Pattern-Fusion method that leaps over an exponential number of intermediate patterns to reach colossal patterns.

To generate a reduced set of patterns, recent studies have focused on mining compressed sets of frequent patterns. Closed patterns can be viewed as a lossless compression of frequent patterns, whereas maximal patterns can be viewed as a simple lossy compression of frequent patterns. Top-k patterns, such as by Wang, Han, Lu, and Tsvetkov [WHLT05], and error-tolerant patterns, such as by Yang, Fayyad, and Bradley [YFB01], are alternative forms of interesting patterns. Afrati, Gionis, and Mannila [AGM04] proposed to use k-itemsets to cover a collection of frequent itemsets. For frequent itemset compression, Yan, Cheng, Han, and Xin [YCHX05] proposed a profile-based approach, and Xin, Han, Yan, and Cheng [XHYC05] proposed a clustering-based approach. By taking into consideration both pattern significance and pattern redundancy, Xin, Cheng, Yan, and Han [XCYH06] proposed a method for extracting redundancy-aware top-k patterns.

Automated semantic annotation of frequent patterns is useful for explaining the meaning of patterns. Mei, Xin, Cheng, et al. [MXC+07] studied methods for semantic annotation of frequent patterns.

An important extension to frequent itemset mining is mining sequence and structural data. This includes mining sequential patterns (Agrawal and Srikant [AS95]; Pei, Han, Mortazavi-Asl, et al. [PHM-A+01, PHM-A+04]; and Zaki [Zak01]); mining frequent espisodes (Mannila, Toivonen, and Verkamo [MTV97]); mining structural

patterns (Inokuchi, Washio, and Motoda [IWM98]; Kuramochi and Karypis [KK01]; and Yan and Han [YH02]); mining cyclic association rules (Özden, Ramaswamy, and Silberschatz [ORS98]); intertransaction association rule mining (Lu, Han, and Feng [LHF98]); and calendric market basket analysis (Ramaswamy, Mahajan, and Silberschatz [RMS98]). Mining such patterns is considered an advanced topic and readers are referred to these sources.

Pattern mining has been extended to help effective data classification and clustering. Pattern-based classification (Liu, Hsu, and Ma [LHM98] and Cheng, Yan, Han, and Hsu [CYHH07]) is discussed in Chapter 9. Pattern-based cluster analysis (Agrawal, Gehrke, Gunopulos, and Raghavan [AGGR98] and H. Wang, W. Wang, Yang, and Yu [WWYY02]) is discussed in Chapter 11.

Pattern mining also helps many other data analysis and processing tasks such as cube gradient mining and discriminative analysis (Imielinski, Khachiyan, and Abdulghani [IKA02]; Dong, Han, Lam, et al. [DHL+04]; Ji, Bailey, and Dong [JBD05]), discriminative pattern-based indexing (Yan, Yu, and Han [YYH05]), and discriminative pattern-based similarity search (Yan, Zhu, Yu, and Han [YZYH06]).

Pattern mining has been extended to mining spatial, temporal, time-series, and multimedia data, and data streams. Mining spatial association rules or spatial collocation rules was studied by Koperski and Han [KH95]; Xiong, Shekhar, Huang, et al. [XSH+04]; and Cao, Mamoulis, and Cheung [CMC05]. Pattern-based mining of time-series data is discussed in Shieh and Keogh [SK08] and Ye and Keogh [YK09]. There are many studies on pattern-based mining of multimedia data such as Zaïane, Han, and Zhu [ZHZ00] and Yuan, Wu, and Yang [YWY07]. Methods for mining frequent patterns on stream data have been proposed by many researchers, including Manku and Motwani [MM02]; Karp, Papadimitriou, and Shenker [KPS03]; and Metwally, Agrawal, and El Abbadi [MAE05]. These pattern mining methods are considered advanced topics.

Pattern mining has broad applications. Application areas include computer science such as software bug analysis, sensor network mining, and performance improvement of operating systems. For example, CP-Miner by Li, Lu, Myagmar, and Zhou [LLMZ04] uses pattern mining to identify copy-pasted code for bug isolation. PR-Miner by Li and Zhou [LZ05] uses pattern mining to extract application-specific programming rules from source code. Discriminative pattern mining is used for program failure detection to classify software behaviors (Lo, Cheng, Han, et al. [LCH+09]) and for troubleshooting in sensor networks (Khan, Le, Ahmadi, et al. [KLA+08]).

This page intentionally left blank

8 Classification: Basic Concepts

Classification is a form of data analysis that extracts models describing important data classes. Such models, called classifiers, predict categorical (discrete, unordered) class labels. For example, we can build a classification model to categorize bank loan applications as either safe or risky. Such analysis can help provide us with a better understanding of the data at large. Many classification methods have been proposed by researchers in machine learning, pattern recognition, and statistics. Most algorithms are memory resident, typically assuming a small data size. Recent data mining research has built on such work, developing scalable classification and prediction techniques capable of handling large amounts of disk-resident data. Classification has numerous applications, including fraud detection, target marketing, performance prediction, manufacturing, and medical diagnosis.

We start off by introducing the main ideas of classification in Section 8.1. In the rest of this chapter, you will learn the basic techniques for data classification such as how to build decision tree classifiers (Section 8.2), Bayesian classifiers (Section 8.3), and rule-based classifiers (Section 8.4). Section 8.5 discusses how to evaluate and compare different classifiers. Various measures of accuracy are given as well as techniques for obtaining reliable accuracy estimates. Methods for increasing classifier accuracy are presented in Section 8.6, including cases for when the data set is class imbalanced (i.e., where the main class of interest is rare).

8.1 Basic Concepts

We introduce the concept of classification in Section 8.1.1. Section 8.1.2 describes the general approach to classification as a two-step process. In the first step, we build a classification model based on previous data. In the second step, we determine if the model's accuracy is acceptable, and if so, we use the model to classify new data.

8.1.1 What Is Classification?

A bank loans officer needs analysis of her data to learn which loan applicants are "safe" and which are "risky" for the bank. A marketing manager at *AllElectronics* needs data

analysis to help guess whether a customer with a given profile will buy a new computer. A medical researcher wants to analyze breast cancer data to predict which one of three specific treatments a patient should receive. In each of these examples, the data analysis task is **classification**, where a model or **classifier** is constructed to predict *class (categorical) labels*, such as "safe" or "risky" for the loan application data; "yes" or "no" for the marketing data; or "treatment A," "treatment B," or "treatment C" for the medical data. These categories can be represented by discrete values, where the ordering among values has no meaning. For example, the values 1, 2, and 3 may be used to represent treatments A, B, and C, where there is no ordering implied among this group of treatment regimes.

Suppose that the marketing manager wants to predict how much a given customer will spend during a sale at *AllElectronics*. This data analysis task is an example of **numeric prediction**, where the model constructed predicts a *continuous-valued function*, or *ordered value*, as opposed to a class label. This model is a **predictor**. **Regression analysis** is a statistical methodology that is most often used for numeric prediction; hence the two terms tend to be used synonymously, although other methods for numeric prediction exist. Classification and numeric prediction are the two major types of **prediction problems**. This chapter focuses on classification.

8.1.2 General Approach to Classification

"How does classification work?" **Data classification** is a two-step process, consisting of a *learning step* (where a classification model is constructed) and a *classification step* (where the model is used to predict class labels for given data). The process is shown for the loan application data of Figure 8.1. (The data are simplified for illustrative purposes. In reality, we may expect many more attributes to be considered.

In the first step, a classifier is built describing a predetermined set of data classes or concepts. This is the **learning step** (or training phase), where a classification algorithm builds the classifier by analyzing or "learning from" a **training set** made up of database tuples and their associated class labels. A tuple, $\boldsymbol{X}$, is represented by an n-dimensional **attribute vector**, $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$, depicting n measurements made on the tuple from n database attributes, respectively, $A_1, A_2, \ldots, A_n$.[1] Each tuple, $\boldsymbol{X}$, is assumed to belong to a predefined class as determined by another database attribute called the **class label attribute**. The class label attribute is discrete-valued and unordered. It is *categorical* (or nominal) in that each value serves as a category or class. The individual tuples making up the training set are referred to as **training tuples** and are randomly sampled from the database under analysis. In the context of classification, data tuples can be referred to as *samples, examples, instances, data points*, or *objects*.[2]

[1] Each attribute represents a "feature" of $\boldsymbol{X}$. Hence, the pattern recognition literature uses the term *feature vector* rather than *attribute vector*. In our discussion, we use the term attribute vector, and in our notation, any variable representing a vector is shown in bold italic font; measurements depicting the vector are shown in italic font (e.g., $\boldsymbol{X} = (x_1, x_2, x_3)$).

[2] In the machine learning literature, training tuples are commonly referred to as *training samples*. Throughout this text, we prefer to use the term *tuples* instead of *samples*.

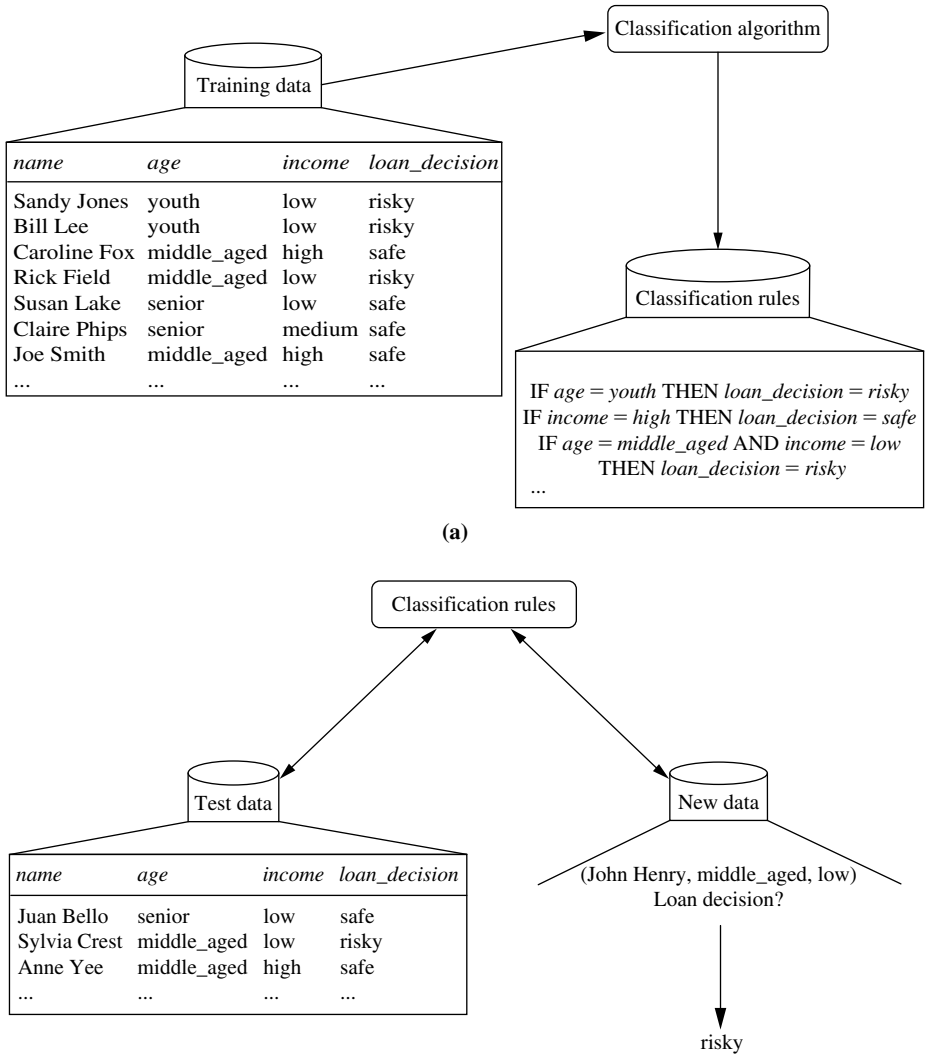

Figure 8.1 The data classification process: (a) *Learning*: Training data are analyzed by a classification algorithm. Here, the class label attribute is *loan_decision*, and the learned model or classifier is represented in the form of classification rules. (b) *Classification*: Test data are used to estimate the accuracy of the classification rules. If the accuracy is considered acceptable, the rules can be applied to the classification of new data tuples.

Because the class label of each training tuple *is provided*, this step is also known as **supervised learning** (i.e., the learning of the classifier is "supervised" in that it is told to which class each training tuple belongs). It contrasts with **unsupervised learning** (or **clustering**), in which the class label of each training tuple is not known, and the number or set of classes to be learned may not be known in advance. For example, if we did not have the *loan_decision* data available for the training set, we could use clustering to try to determine "groups of like tuples," which may correspond to risk groups within the loan application data. Clustering is the topic of Chapters 10 and 11.

This first step of the classification process can also be viewed as the learning of a mapping or function, $y = f(\boldsymbol{X})$, that can predict the associated class label y of a given tuple $\boldsymbol{X}$. In this view, we wish to learn a mapping or function that separates the data classes. Typically, this mapping is represented in the form of classification rules, decision trees, or mathematical formulae. In our example, the mapping is represented as classification rules that identify loan applications as being either safe or risky (Figure 8.1a). The rules can be used to categorize future data tuples, as well as provide deeper insight into the data contents. They also provide a compressed data representation.

"*What about classification accuracy?*" In the second step (Figure 8.1b), the model is used for classification. First, the predictive accuracy of the classifier is estimated. If we were to use the training set to measure the classifier's accuracy, this estimate would likely be optimistic, because the classifier tends to **overfit** the data (i.e., during learning it may incorporate some particular anomalies of the training data that are not present in the general data set overall). Therefore, a **test set** is used, made up of **test tuples** and their associated class labels. They are independent of the training tuples, meaning that they were not used to construct the classifier.

The **accuracy** of a classifier on a given test set is the percentage of test set tuples that are correctly classified by the classifier. The associated class label of each test tuple is compared with the learned classifier's class prediction for that tuple. Section 8.5 describes several methods for estimating classifier accuracy. If the accuracy of the classifier is considered acceptable, the classifier can be used to classify future data tuples for which the class label is not known. (Such data are also referred to in the machine learning literature as "unknown" or "previously unseen" data.) For example, the classification rules learned in Figure 8.1(a) from the analysis of data from previous loan applications can be used to approve or reject new or future loan applicants.

8.2 Decision Tree Induction

Decision tree induction is the learning of decision trees from class-labeled training tuples. A **decision tree** is a flowchart-like tree structure, where each **internal node** (nonleaf node) denotes a test on an attribute, each **branch** represents an outcome of the test, and each **leaf node** (or *terminal node*) holds a class label. The topmost node in a tree is the **root** node. A typical decision tree is shown in Figure 8.2. It represents the concept *buys_computer*, that is, it predicts whether a customer at *AllElectronics* is

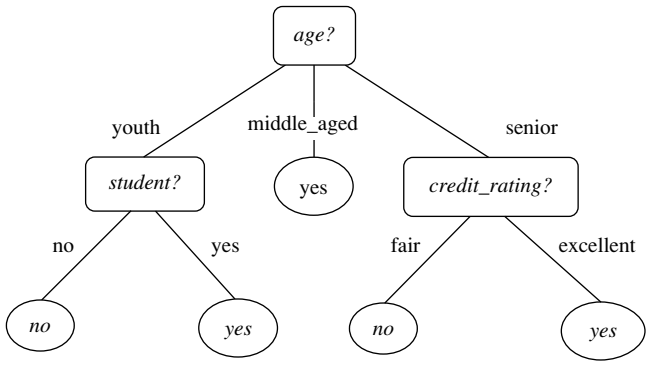

Figure 8.2 A decision tree for the concept *buys_computer*, indicating whether an *AllElectronics* customer is likely to purchase a computer. Each internal (nonleaf) node represents a test on an attribute. Each leaf node represents a class (either *buys_computer* = *yes* or *buys_computer* = *no*).

likely to purchase a computer. Internal nodes are denoted by rectangles, and leaf nodes are denoted by ovals. Some decision tree algorithms produce only *binary* trees (where each internal node branches to exactly two other nodes), whereas others can produce nonbinary trees.

"*How are decision trees used for classification?*" Given a tuple, $\boldsymbol{X}$, for which the associated class label is unknown, the attribute values of the tuple are tested against the decision tree. A path is traced from the root to a leaf node, which holds the class prediction for that tuple. Decision trees can easily be converted to classification rules.

"*Why are decision tree classifiers so popular?*" The construction of decision tree classifiers does not require any domain knowledge or parameter setting, and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle multidimensional data. Their representation of acquired knowledge in tree form is intuitive and generally easy to assimilate by humans. The learning and classification steps of decision tree induction are simple and fast. In general, decision tree classifiers have good accuracy. However, successful use may depend on the data at hand. Decision tree induction algorithms have been used for classification in many application areas such as medicine, manufacturing and production, financial analysis, astronomy, and molecular biology. Decision trees are the basis of several commercial rule induction systems.

In Section 8.2.1, we describe a basic algorithm for learning decision trees. During tree construction, *attribute selection measures* are used to select the attribute that best partitions the tuples into distinct classes. Popular measures of attribute selection are given in Section 8.2.2. When decision trees are built, many of the branches may reflect noise or outliers in the training data. *Tree pruning* attempts to identify and remove such branches, with the goal of improving classification accuracy on unseen data. Tree pruning is described in Section 8.2.3. Scalability issues for the induction of decision trees

from large databases are discussed in Section 8.2.4. Section 8.2.5 presents a visual mining approach to decision tree induction.

8.2.1 Decision Tree Induction

During the late 1970s and early 1980s, J. Ross Quinlan, a researcher in machine learning, developed a decision tree algorithm known as **ID3** (Iterative Dichotomiser). This work expanded on earlier work on *concept learning systems*, described by E. B. Hunt, J. Marin, and P. T. Stone. Quinlan later presented **C4.5** (a successor of ID3), which became a benchmark to which newer supervised learning algorithms are often compared. In 1984, a group of statisticians (L. Breiman, J. Friedman, R. Olshen, and C. Stone) published the book *Classification and Regression Trees* (**CART**), which described the generation of binary decision trees. ID3 and CART were invented independently of one another at around the same time, yet follow a similar approach for learning decision trees from training tuples. These two cornerstone algorithms spawned a flurry of work on decision tree induction.

ID3, C4.5, and CART adopt a greedy (i.e., nonbacktracking) approach in which decision trees are constructed in a top-down recursive divide-and-conquer manner. Most algorithms for decision tree induction also follow a top-down approach, which starts with a training set of tuples and their associated class labels. The training set is recursively partitioned into smaller subsets as the tree is being built. A basic decision tree algorithm is summarized in Figure 8.3. At first glance, the algorithm may appear long, but fear not! It is quite straightforward. The strategy is as follows.

- The algorithm is called with three parameters: D, *attribute_list*, and *Attribute_selection_method*. We refer to D as a data partition. Initially, it is the complete set of training tuples and their associated class labels. The parameter *attribute_list* is a list of attributes describing the tuples. *Attribute_selection_method* specifies a heuristic procedure for selecting the attribute that "best" discriminates the given tuples according to class. This procedure employs an attribute selection measure such as information gain or the Gini index. Whether the tree is strictly binary is generally driven by the attribute selection measure. Some attribute selection measures, such as the Gini index, enforce the resulting tree to be binary. Others, like information gain, do not, therein allowing multiway splits (i.e., two or more branches to be grown from a node).
- The tree starts as a single node, N, representing the training tuples in D (step 1).[3]

[3]The partition of class-labeled training tuples at node N is the set of tuples that follow a path from the root of the tree to node N when being processed by the tree. This set is sometimes referred to in the literature as the *family* of tuples at node N. We have referred to this set as the "tuples represented at node N," "the tuples that reach node N," or simply "the tuples at node N." Rather than storing the actual tuples at a node, most implementations store pointers to these tuples.

Algorithm: Generate_decision_tree. Generate a decision tree from the training tuples of data partition, *D*.

Input:

- Data partition, *D*, which is a set of training tuples and their associated class labels;
- *attribute_list*, the set of candidate attributes;
- *Attribute_selection_method*, a procedure to determine the splitting criterion that "best" partitions the data tuples into individual classes. This criterion consists of a *splitting_attribute* and, possibly, either a *split-point* or *splitting subset*.

Output: A decision tree.

Method:

```
(1)  create a node N;
(2)  if tuples in D are all of the same class, C, then
(3)      return N as a leaf node labeled with the class C;
(4)  if attribute_list is empty then
(5)      return N as a leaf node labeled with the majority class in D; // majority voting
(6)  apply Attribute_selection_method(D, attribute_list) to find the "best" splitting_criterion;
(7)  label node N with splitting_criterion;
(8)  if splitting_attribute is discrete-valued and
         multiway splits allowed then // not restricted to binary trees
(9)      attribute_list ← attribute_list − splitting_attribute; // remove splitting_attribute
(10) for each outcome j of splitting_criterion
     // partition the tuples and grow subtrees for each partition
(11)     let Dj be the set of data tuples in D satisfying outcome j; // a partition
(12)     if Dj is empty then
(13)         attach a leaf labeled with the majority class in D to node N;
(14)     else attach the node returned by Generate_decision_tree(Dj, attribute_list) to node N;
     endfor
(15) return N;
```

Figure 8.3 Basic algorithm for inducing a decision tree from training tuples.

- If the tuples in *D* are all of the same class, then node *N* becomes a leaf and is labeled with that class (steps 2 and 3). Note that steps 4 and 5 are terminating conditions. All terminating conditions are explained at the end of the algorithm.
- Otherwise, the algorithm calls *Attribute_selection_method* to determine the **splitting criterion**. The splitting criterion tells us which attribute to test at node *N* by determining the "best" way to separate or partition the tuples in *D* into individual classes (step 6). The splitting criterion also tells us which branches to grow from node *N* with respect to the outcomes of the chosen test. More specifically, the splitting criterion indicates the **splitting attribute** and may also indicate either a **split-point** or a **splitting subset**. The splitting criterion is determined so that, ideally, the resulting

partitions at each branch are as "pure" as possible. A partition is **pure** if all the tuples in it belong to the same class. In other words, if we split up the tuples in D according to the mutually exclusive outcomes of the splitting criterion, we hope for the resulting partitions to be as pure as possible.

- The node N is labeled with the splitting criterion, which serves as a test at the node (step 7). A branch is grown from node N for each of the outcomes of the splitting criterion. The tuples in D are partitioned accordingly (steps 10 to 11). There are three possible scenarios, as illustrated in Figure 8.4. Let A be the splitting attribute. A has v distinct values, $\{a_1, a_2, \ldots, a_v\}$, based on the training data.

  1. *A is discrete-valued:* In this case, the outcomes of the test at node N correspond directly to the known values of A. A branch is created for each known value, a_j, of A and labeled with that value (Figure 8.4a). Partition D_j is the subset of class-labeled tuples in D having value a_j of A. Because all the tuples in a

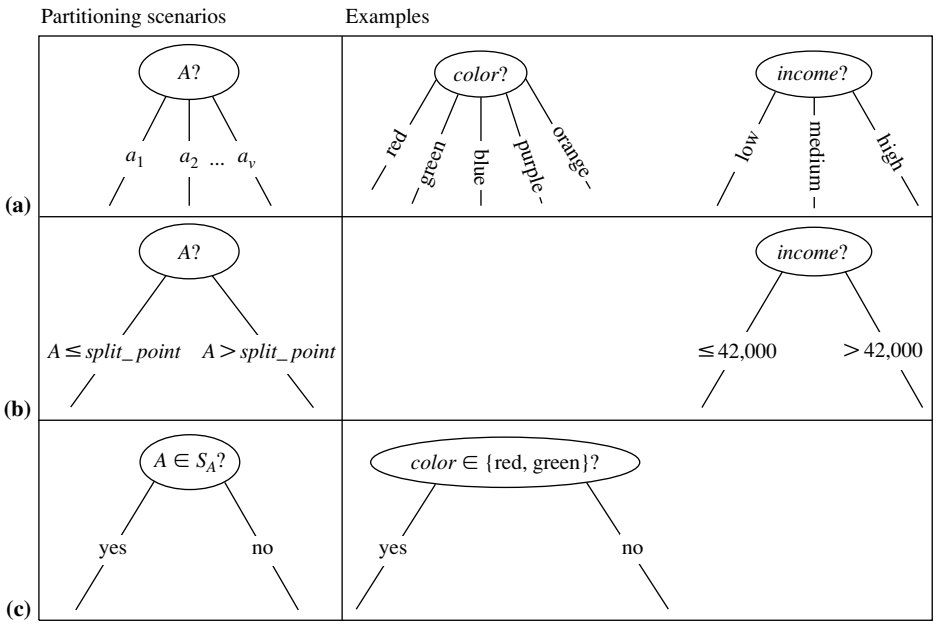

Figure 8.4 This figure shows three possibilities for partitioning tuples based on the splitting criterion, each with examples. Let A be the splitting attribute. (a) If A is discrete-valued, then one branch is grown for each known value of A. (b) If A is continuous-valued, then two branches are grown, corresponding to $A \leq$ *split_point* and $A >$ *split_point*. (c) If A is discrete-valued and a binary tree must be produced, then the test is of the form $A \in S_A$, where S_A is the splitting subset for A.

given partition have the same value for A, A need not be considered in any future partitioning of the tuples. Therefore, it is removed from *attribute_list* (steps 8 and 9).

2. *A is continuous-valued:* In this case, the test at node N has two possible outcomes, corresponding to the conditions $A \leq$ *split_point* and $A >$ *split_point*, respectively, where *split_point* is the split-point returned by *Attribute_selection_method* as part of the splitting criterion. (In practice, the split-point, a, is often taken as the midpoint of two known adjacent values of A and therefore may not actually be a preexisting value of A from the training data.) Two branches are grown from N and labeled according to the previous outcomes (Figure 8.4b). The tuples are partitioned such that D_1 holds the subset of class-labeled tuples in D for which $A \leq$ *split_point*, while D_2 holds the rest.

3. *A is discrete-valued* and a *binary tree* must be produced (as dictated by the attribute selection measure or algorithm being used): The test at node N is of the form "$A \in S_A$?," where S_A is the splitting subset for A, returned by *Attribute_selection_method* as part of the splitting criterion. It is a subset of the known values of A. If a given tuple has value a_j of A and if $a_j \in S_A$, then the test at node N is satisfied. Two branches are grown from N (Figure 8.4c). By convention, the left branch out of N is labeled *yes* so that D_1 corresponds to the subset of class-labeled tuples in D that satisfy the test. The right branch out of N is labeled *no* so that D_2 corresponds to the subset of class-labeled tuples from D that do not satisfy the test.

- The algorithm uses the same process recursively to form a decision tree for the tuples at each resulting partition, D_j, of D (step 14).
- The recursive partitioning stops only when any one of the following terminating conditions is true:

  1. All the tuples in partition D (represented at node N) belong to the same class (steps 2 and 3).

  2. There are no remaining attributes on which the tuples may be further partitioned (step 4). In this case, **majority voting** is employed (step 5). This involves converting node N into a leaf and labeling it with the most common class in D. Alternatively, the class distribution of the node tuples may be stored.

  3. There are no tuples for a given branch, that is, a partition D_j is empty (step 12). In this case, a leaf is created with the majority class in D (step 13).

- The resulting decision tree is returned (step 15).

The computational complexity of the algorithm given training set D is $O(n \times |D| \times log(|D|))$, where n is the number of attributes describing the tuples in D and $|D|$ is the number of training tuples in D. This means that the computational cost of growing a tree grows at most $n \times |D| \times log(|D|)$ with $|D|$ tuples. The proof is left as an exercise for the reader.

Incremental versions of decision tree induction have also been proposed. When given new training data, these restructure the decision tree acquired from learning on previous training data, rather than relearning a new tree from scratch.

Differences in decision tree algorithms include how the attributes are selected in creating the tree (Section 8.2.2) and the mechanisms used for pruning (Section 8.2.3). The basic algorithm described earlier requires one pass over the training tuples in D for each level of the tree. This can lead to long training times and lack of available memory when dealing with large databases. Improvements regarding the scalability of decision tree induction are discussed in Section 8.2.4. Section 8.2.5 presents a visual interactive approach to decision tree construction. A discussion of strategies for extracting rules from decision trees is given in Section 8.4.2 regarding rule-based classification.

8.2.2 Attribute Selection Measures

An **attribute selection measure** is a heuristic for selecting the splitting criterion that "best" separates a given data partition, D, of class-labeled training tuples into individual classes. If we were to split D into smaller partitions according to the outcomes of the splitting criterion, ideally each partition would be pure (i.e., all the tuples that fall into a given partition would belong to the same class). Conceptually, the "best" splitting criterion is the one that most closely results in such a scenario. Attribute selection measures are also known as **splitting rules** because they determine how the tuples at a given node are to be split.

The attribute selection measure provides a ranking for each attribute describing the given training tuples. The attribute having the best score for the measure[4] is chosen as the *splitting attribute* for the given tuples. If the splitting attribute is continuous-valued or if we are restricted to binary trees, then, respectively, either a *split point* or a *splitting subset* must also be determined as part of the splitting criterion. The tree node created for partition D is labeled with the splitting criterion, branches are grown for each outcome of the criterion, and the tuples are partitioned accordingly. This section describes three popular attribute selection measures—*information gain, gain ratio*, and *Gini index*.

The notation used herein is as follows. Let D, the data partition, be a training set of class-labeled tuples. Suppose the class label attribute has m distinct values defining m distinct classes, C_i (for $i = 1, \ldots, m$). Let $C_{i,D}$ be the set of tuples of class C_i in D. Let $|D|$ and $|C_{i,D}|$ denote the number of tuples in D and $C_{i,D}$, respectively.

Information Gain

ID3 uses **information gain** as its attribute selection measure. This measure is based on pioneering work by Claude Shannon on information theory, which studied the value or "information content" of messages. Let node N represent or hold the tuples of partition D. The attribute with the highest information gain is chosen as the splitting attribute for node N. This attribute minimizes the information needed to classify the tuples in the

[4]Depending on the measure, either the highest or lowest score is chosen as the best (i.e., some measures strive to maximize while others strive to minimize).

resulting partitions and reflects the least randomness or "impurity" in these partitions. Such an approach minimizes the expected number of tests needed to classify a given tuple and guarantees that a simple (but not necessarily the simplest) tree is found.

The expected information needed to classify a tuple in D is given by

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i), \tag{8.1}$$

where p_i is the nonzero probability that an arbitrary tuple in D belongs to class C_i and is estimated by $|C_{i,D}|/|D|$. A log function to the base 2 is used, because the information is encoded in bits. $Info(D)$ is just the average amount of information needed to identify the class label of a tuple in D. Note that, at this point, the information we have is based solely on the proportions of tuples of each class. $Info(D)$ is also known as the **entropy** of D.

Now, suppose we were to partition the tuples in D on some attribute A having v distinct values, $\{a_1, a_2, \ldots, a_v\}$, as observed from the training data. If A is discrete-valued, these values correspond directly to the v outcomes of a test on A. Attribute A can be used to split D into v partitions or subsets, $\{D_1, D_2, \ldots, D_v\}$, where D_j contains those tuples in D that have outcome a_j of A. These partitions would correspond to the branches grown from node N. Ideally, we would like this partitioning to produce an exact classification of the tuples. That is, we would like for each partition to be pure. However, it is quite likely that the partitions will be impure (e.g., where a partition may contain a collection of tuples from different classes rather than from a single class).

How much more information would we still need (after the partitioning) to arrive at an exact classification? This amount is measured by

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Info(D_j). \tag{8.2}$$

The term $\frac{|D_j|}{|D|}$ acts as the weight of the jth partition. $Info_A(D)$ is the expected information required to classify a tuple from D based on the partitioning by A. The smaller the expected information (still) required, the greater the purity of the partitions.

Information gain is defined as the difference between the original information requirement (i.e., based on just the proportion of classes) and the new requirement (i.e., obtained after partitioning on A). That is,

$$Gain(A) = Info(D) - Info_A(D). \tag{8.3}$$

In other words, $Gain(A)$ tells us how much would be gained by branching on A. It is the expected reduction in the information requirement caused by knowing the value of A. The attribute A with the highest information gain, $Gain(A)$, is chosen as the splitting attribute at node N. This is equivalent to saying that we want to partition on the attribute A that would do the "best classification," so that the amount of information still required to finish classifying the tuples is minimal (i.e., minimum $Info_A(D)$).

Table 8.1 Class-Labeled Training Tuples from the *AllElectronics* Customer Database

RID	*age*	*income*	*student*	*credit_rating*	*Class: buys_computer*
1	youth	high	no	fair	no
2	youth	high	no	excellent	no
3	middle_aged	high	no	fair	yes
4	senior	medium	no	fair	yes
5	senior	low	yes	fair	yes
6	senior	low	yes	excellent	no
7	middle_aged	low	yes	excellent	yes
8	youth	medium	no	fair	no
9	youth	low	yes	fair	yes
10	senior	medium	yes	fair	yes
11	youth	medium	yes	excellent	yes
12	middle_aged	medium	no	excellent	yes
13	middle_aged	high	yes	fair	yes
14	senior	medium	no	excellent	no

Example 8.1 **Induction of a decision tree using information gain.** Table 8.1 presents a training set, D, of class-labeled tuples randomly selected from the *AllElectronics* customer database. (The data are adapted from Quinlan [Qui86]. In this example, each attribute is discrete-valued. Continuous-valued attributes have been generalized.) The class label attribute, *buys_computer*, has two distinct values (namely, {*yes, no*}); therefore, there are two distinct classes (i.e., $m = 2$). Let class C_1 correspond to *yes* and class C_2 correspond to *no*. There are nine tuples of class *yes* and five tuples of class *no*. A (root) node N is created for the tuples in D. To find the splitting criterion for these tuples, we must compute the information gain of each attribute. We first use Eq. (8.1) to compute the expected information needed to classify a tuple in D:

$$Info(D) = -\frac{9}{14}\log_2\left(\frac{9}{14}\right) - \frac{5}{14}\log_2\left(\frac{5}{14}\right) = 0.940 \text{ bits}.$$

Next, we need to compute the expected information requirement for each attribute. Let's start with the attribute *age*. We need to look at the distribution of *yes* and *no* tuples for each category of *age*. For the *age* category "youth," there are two *yes* tuples and three *no* tuples. For the category "middle_aged," there are four *yes* tuples and zero *no* tuples. For the category "senior," there are three *yes* tuples and two *no* tuples. Using Eq. (8.2), the expected information needed to classify a tuple in D if the tuples are partitioned according to *age* is

$$Info_{age}(D) = \frac{5}{14} \times \left(-\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5}\right)$$

$$+\frac{4}{14}\times\left(-\frac{4}{4}\log_2\frac{4}{4}\right)$$
$$+\frac{5}{14}\times\left(-\frac{3}{5}\log_2\frac{3}{5}-\frac{2}{5}\log_2\frac{2}{5}\right)$$
$$=0.694 \text{ bits}.$$

Hence, the gain in information from such a partitioning would be

$$Gain(age) = Info(D) - Info_{age}(D) = 0.940 - 0.694 = 0.246 \text{ bits}.$$

Similarly, we can compute $Gain(income) = 0.029$ bits, $Gain(student) = 0.151$ bits, and $Gain(credit\_rating) = 0.048$ bits. Because *age* has the highest information gain among the attributes, it is selected as the splitting attribute. Node N is labeled with *age*, and branches are grown for each of the attribute's values. The tuples are then partitioned accordingly, as shown in Figure 8.5. Notice that the tuples falling into the partition for *age* = *middle_aged* all belong to the same class. Because they all belong to class "*yes*," a leaf should therefore be created at the end of this branch and labeled "*yes*." The final decision tree returned by the algorithm was shown earlier in Figure 8.2. ■

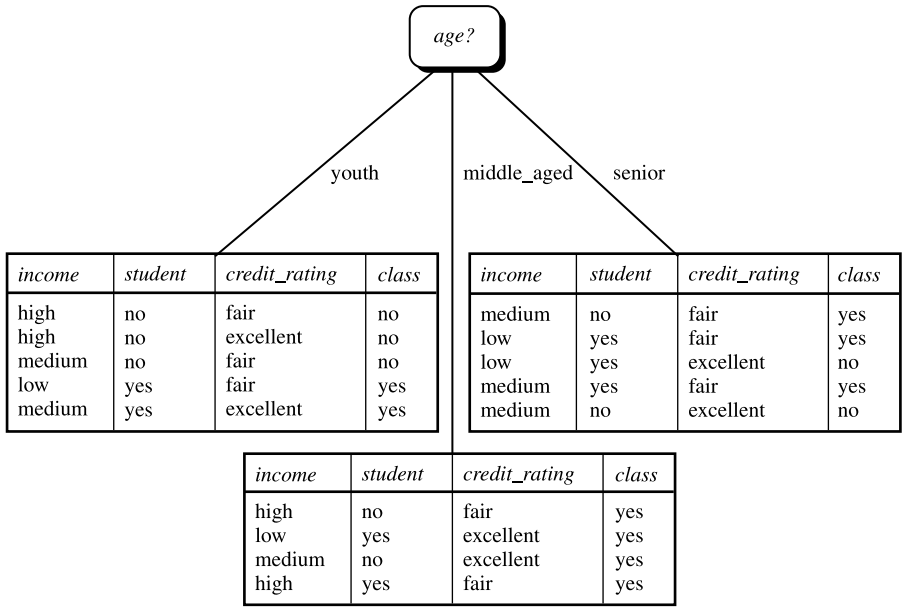

Figure 8.5 The attribute *age* has the highest information gain and therefore becomes the splitting attribute at the root node of the decision tree. Branches are grown for each outcome of *age*. The tuples are shown partitioned accordingly.

"*But how can we compute the information gain of an attribute that is continuous-valued, unlike in the example?*" Suppose, instead, that we have an attribute A that is continuous-valued, rather than discrete-valued. (For example, suppose that instead of the discretized version of *age* from the example, we have the raw values for this attribute.) For such a scenario, we must determine the "best" **split-point** for A, where the split-point is a threshold on A.

We first sort the values of A in increasing order. Typically, the midpoint between each pair of adjacent values is considered as a possible split-point. Therefore, given v values of A, then $v-1$ possible splits are evaluated. For example, the midpoint between the values a_i and a_{i+1} of A is

$$\frac{a_i + a_{i+1}}{2}. \tag{8.4}$$

If the values of A are sorted in advance, then determining the best split for A requires only one pass through the values. For each possible split-point for A, we evaluate $Info_A(D)$, where the number of partitions is two, that is, $v = 2$ (or $j = 1, 2$) in Eq. (8.2). The point with the minimum expected information requirement for A is selected as the *split_point* for A. D_1 is the set of tuples in D satisfying $A \leq$ *split_point*, and D_2 is the set of tuples in D satisfying $A >$ *split_point*.

Gain Ratio

The information gain measure is biased toward tests with many outcomes. That is, it prefers to select attributes having a large number of values. For example, consider an attribute that acts as a unique identifier such as *product_ID*. A split on *product_ID* would result in a large number of partitions (as many as there are values), each one containing just one tuple. Because each partition is pure, the information required to classify data set D based on this partitioning would be $Info_{product\_ID}(D) = 0$. Therefore, the information gained by partitioning on this attribute is maximal. Clearly, such a partitioning is useless for classification.

C4.5, a successor of ID3, uses an extension to information gain known as *gain ratio*, which attempts to overcome this bias. It applies a kind of normalization to information gain using a "split information" value defined analogously with $Info(D)$ as

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2\left(\frac{|D_j|}{|D|}\right). \tag{8.5}$$

This value represents the potential information generated by splitting the training data set, D, into v partitions, corresponding to the v outcomes of a test on attribute A. Note that, for each outcome, it considers the number of tuples having that outcome with respect to the total number of tuples in D. It differs from information gain, which measures the information with respect to classification that is acquired based on the

same partitioning. The gain ratio is defined as

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo_A(D)}. \tag{8.6}$$

The attribute with the maximum gain ratio is selected as the splitting attribute. Note, however, that as the split information approaches 0, the ratio becomes unstable. A constraint is added to avoid this, whereby the information gain of the test selected must be large—at least as great as the average gain over all tests examined.

Example 8.2 **Computation of gain ratio for the attribute *income*.** A test on *income* splits the data of Table 8.1 into three partitions, namely *low*, *medium*, and *high*, containing four, six, and four tuples, respectively. To compute the gain ratio of *income*, we first use Eq. (8.5) to obtain

$$SplitInfo_{income}(D) = -\frac{4}{14} \times \log_2\left(\frac{4}{14}\right) - \frac{6}{14} \times \log_2\left(\frac{6}{14}\right) - \frac{4}{14} \times \log_2\left(\frac{4}{14}\right)$$

$$= 1.557.$$

From Example 8.1, we have $Gain(income) = 0.029$. Therefore, $GainRatio(income) = 0.029/1.557 = 0.019$. ■

Gini Index

The Gini index is used in CART. Using the notation previously described, the Gini index measures the impurity of D, a data partition or set of training tuples, as

$$Gini(D) = 1 - \sum_{i=1}^{m} p_i^2, \tag{8.7}$$

where p_i is the probability that a tuple in D belongs to class C_i and is estimated by $|C_{i,D}|/|D|$. The sum is computed over m classes.

The Gini index considers a binary split for each attribute. Let's first consider the case where A is a discrete-valued attribute having v distinct values, $\{a_1, a_2, \ldots, a_v\}$, occurring in D. To determine the best binary split on A, we examine all the possible subsets that can be formed using known values of A. Each subset, S_A, can be considered as a binary test for attribute A of the form "$A \in S_A$?" Given a tuple, this test is satisfied if the value of A for the tuple is among the values listed in S_A. If A has v possible values, then there are 2^v possible subsets. For example, if *income* has three possible values, namely {*low, medium, high*}, then the possible subsets are {*low, medium, high*}, {*low, medium*}, {*low, high*}, {*medium, high*}, {*low*}, {*medium*}, {*high*}, and {}. We exclude the power set, {*low, medium, high*}, and the empty set from consideration since, conceptually, they do not represent a split. Therefore, there are $2^v - 2$ possible ways to form two partitions of the data, D, based on a binary split on A.

When considering a binary split, we compute a weighted sum of the impurity of each resulting partition. For example, if a binary split on A partitions D into D_1 and D_2, the Gini index of D given that partitioning is

$$Gini_A(D) = \frac{|D_1|}{|D|} Gini(D_1) + \frac{|D_2|}{|D|} Gini(D_2). \tag{8.8}$$

For each attribute, each of the possible binary splits is considered. For a discrete-valued attribute, the subset that gives the minimum Gini index for that attribute is selected as its splitting subset.

For continuous-valued attributes, each possible split-point must be considered. The strategy is similar to that described earlier for information gain, where the midpoint between each pair of (sorted) adjacent values is taken as a possible split-point. The point giving the minimum Gini index for a given (continuous-valued) attribute is taken as the split-point of that attribute. Recall that for a possible split-point of A, D_1 is the set of tuples in D satisfying $A \leq$ *split_point*, and D_2 is the set of tuples in D satisfying $A >$ *split_point*.

The reduction in impurity that would be incurred by a binary split on a discrete- or continuous-valued attribute A is

$$\Delta Gini(A) = Gini(D) - Gini_A(D). \tag{8.9}$$

The attribute that maximizes the reduction in impurity (or, equivalently, has the minimum Gini index) is selected as the splitting attribute. This attribute and either its splitting subset (for a discrete-valued splitting attribute) or split-point (for a continuous-valued splitting attribute) together form the splitting criterion.

Example 8.3 Induction of a decision tree using the Gini index. Let D be the training data shown earlier in Table 8.1, where there are nine tuples belonging to the class *buys_computer* = *yes* and the remaining five tuples belong to the class *buys_computer* = *no*. A (root) node N is created for the tuples in D. We first use Eq. (8.7) for the Gini index to compute the impurity of D:

$$Gini(D) = 1 - \left(\frac{9}{14}\right)^2 - \left(\frac{5}{14}\right)^2 = 0.459.$$

To find the splitting criterion for the tuples in D, we need to compute the Gini index for each attribute. Let's start with the attribute *income* and consider each of the possible splitting subsets. Consider the subset {*low, medium*}. This would result in 10 tuples in partition D_1 satisfying the condition "*income* $\in$ {*low, medium*}." The remaining four tuples of D would be assigned to partition D_2. The Gini index value computed based on

this partitioning is

$$\begin{aligned} &Gini_{income \in \{low,medium\}}(D) \\ &\quad = \frac{10}{14} Gini(D_1) + \frac{4}{14} Gini(D_2) \\ &\quad = \frac{10}{14}\left(1 - \left(\frac{7}{10}\right)^2 - \left(\frac{3}{10}\right)^2\right) + \frac{4}{14}\left(1 - \left(\frac{2}{4}\right)^2 - \left(\frac{2}{4}\right)^2\right) \\ &\quad = 0.443 \\ &\quad = Gini_{income \in \{high\}}(D). \end{aligned}$$

Similarly, the Gini index values for splits on the remaining subsets are 0.458 (for the subsets {*low, high*} and {*medium*}) and 0.450 (for the subsets {*medium, high*} and {*low*}). Therefore, the best binary split for attribute *income* is on {*low, medium*} (or {*high*}) because it minimizes the Gini index. Evaluating *age*, we obtain {*youth, senior*} (or {*middle_aged*}) as the best split for *age* with a Gini index of 0.375; the attributes *student* and *credit_rating* are both binary, with Gini index values of 0.367 and 0.429, respectively.

The attribute *age* and splitting subset {*youth, senior*} therefore give the minimum Gini index overall, with a reduction in impurity of $0.459 - 0.357 = 0.102$. The binary split "*age* $\in$ {*youth, senior*?}" results in the maximum reduction in impurity of the tuples in D and is returned as the splitting criterion. Node N is labeled with the criterion, two branches are grown from it, and the tuples are partitioned accordingly. ■

Other Attribute Selection Measures

This section on attribute selection measures was not intended to be exhaustive. We have shown three measures that are commonly used for building decision trees. These measures are not without their biases. Information gain, as we saw, is biased toward multivalued attributes. Although the gain ratio adjusts for this bias, it tends to prefer unbalanced splits in which one partition is much smaller than the others. The Gini index is biased toward multivalued attributes and has difficulty when the number of classes is large. It also tends to favor tests that result in equal-size partitions and purity in both partitions. Although biased, these measures give reasonably good results in practice.

Many other attribute selection measures have been proposed. CHAID, a decision tree algorithm that is popular in marketing, uses an attribute selection measure that is based on the statistical χ^2 test for independence. Other measures include C-SEP (which performs better than information gain and the Gini index in certain cases) and G-statistic (an information theoretic measure that is a close approximation to χ^2 distribution).

Attribute selection measures based on the **Minimum Description Length (MDL)** principle have the least bias toward multivalued attributes. MDL-based measures use encoding techniques to define the "best" decision tree as the one that requires the fewest number of bits to both (1) encode the tree and (2) encode the exceptions to the tree

(i.e., cases that are not correctly classified by the tree). Its main idea is that the simplest of solutions is preferred.

Other attribute selection measures consider **multivariate splits** (i.e., where the partitioning of tuples is based on a *combination* of attributes, rather than on a single attribute). The CART system, for example, can find multivariate splits based on a linear combination of attributes. Multivariate splits are a form of **attribute** (or feature) **construction**, where new attributes are created based on the existing ones. (Attribute construction was also discussed in Chapter 3, as a form of data transformation.) These other measures mentioned here are beyond the scope of this book. Additional references are given in the bibliographic notes at the end of this chapter (Section 8.9).

"*Which attribute selection measure is the best?*" All measures have some bias. It has been shown that the time complexity of decision tree induction generally increases exponentially with tree height. Hence, measures that tend to produce shallower trees (e.g., with multiway rather than binary splits, and that favor more balanced splits) may be preferred. However, some studies have found that shallow trees tend to have a large number of leaves and higher error rates. Despite several comparative studies, no one attribute selection measure has been found to be significantly superior to others. Most measures give quite good results.

8.2.3 Tree Pruning

When a decision tree is built, many of the branches will reflect anomalies in the training data due to noise or outliers. Tree pruning methods address this problem of *overfitting* the data. Such methods typically use statistical measures to remove the least-reliable branches. An unpruned tree and a pruned version of it are shown in Figure 8.6. Pruned trees tend to be smaller and less complex and, thus, easier to comprehend. They are usually faster and better at correctly classifying independent test data (i.e., of previously unseen tuples) than unpruned trees.

"*How does tree pruning work?*" There are two common approaches to tree pruning: *prepruning* and *postpruning*.

In the **prepruning** approach, a tree is "pruned" by halting its construction early (e.g., by deciding not to further split or partition the subset of training tuples at a given node). Upon halting, the node becomes a leaf. The leaf may hold the most frequent class among the subset tuples or the probability distribution of those tuples.

When constructing a tree, measures such as statistical significance, information gain, Gini index, and so on, can be used to assess the goodness of a split. If partitioning the tuples at a node would result in a split that falls below a prespecified threshold, then further partitioning of the given subset is halted. There are difficulties, however, in choosing an appropriate threshold. High thresholds could result in oversimplified trees, whereas low thresholds could result in very little simplification.

The second and more common approach is **postpruning**, which removes subtrees from a "fully grown" tree. A subtree at a given node is pruned by removing its branches and replacing it with a leaf. The leaf is labeled with the most frequent class among the subtree being replaced. For example, notice the subtree at node "A_3?" in the unpruned

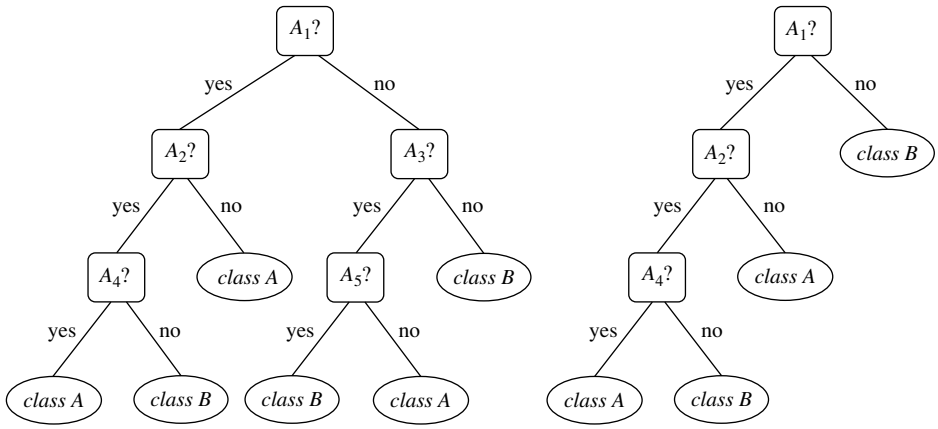

Figure 8.6 An unpruned decision tree and a pruned version of it.

tree of Figure 8.6. Suppose that the most common class within this subtree is "*class B*." In the pruned version of the tree, the subtree in question is pruned by replacing it with the leaf "*class B*."

The **cost complexity** pruning algorithm used in CART is an example of the postpruning approach. This approach considers the cost complexity of a tree to be a function of the number of leaves in the tree and the error rate of the tree (where the **error rate** is the percentage of tuples misclassified by the tree). It starts from the bottom of the tree. For each internal node, N, it computes the cost complexity of the subtree at N, and the cost complexity of the subtree at N if it were to be pruned (i.e., replaced by a leaf node). The two values are compared. If pruning the subtree at node N would result in a smaller cost complexity, then the subtree is pruned. Otherwise, it is kept.

A **pruning set** of class-labeled tuples is used to estimate cost complexity. This set is independent of the training set used to build the unpruned tree and of any test set used for accuracy estimation. The algorithm generates a set of progressively pruned trees. In general, the smallest decision tree that minimizes the cost complexity is preferred.

C4.5 uses a method called **pessimistic pruning**, which is similar to the cost complexity method in that it also uses error rate estimates to make decisions regarding subtree pruning. Pessimistic pruning, however, does not require the use of a prune set. Instead, it uses the training set to estimate error rates. Recall that an estimate of accuracy or error based on the training set is overly optimistic and, therefore, strongly biased. The pessimistic pruning method therefore adjusts the error rates obtained from the training set by adding a penalty, so as to counter the bias incurred.

Rather than pruning trees based on estimated error rates, we can prune trees based on the number of bits required to encode them. The "best" pruned tree is the one that minimizes the number of encoding bits. This method adopts the MDL principle, which was briefly introduced in Section 8.2.2. The basic idea is that the simplest solution is preferred. Unlike cost complexity pruning, it does not require an independent set of tuples.

Alternatively, prepruning and postpruning may be interleaved for a combined approach. Postpruning requires more computation than prepruning, yet generally leads to a more reliable tree. No single pruning method has been found to be superior over all others. Although some pruning methods do depend on the availability of additional data for pruning, this is usually not a concern when dealing with large databases.

Although pruned trees tend to be more compact than their unpruned counterparts, they may still be rather large and complex. Decision trees can suffer from *repetition* and *replication* (Figure 8.7), making them overwhelming to interpret. **Repetition** occurs when an attribute is repeatedly tested along a given branch of the tree (e.g., "*age* < *60?*,"

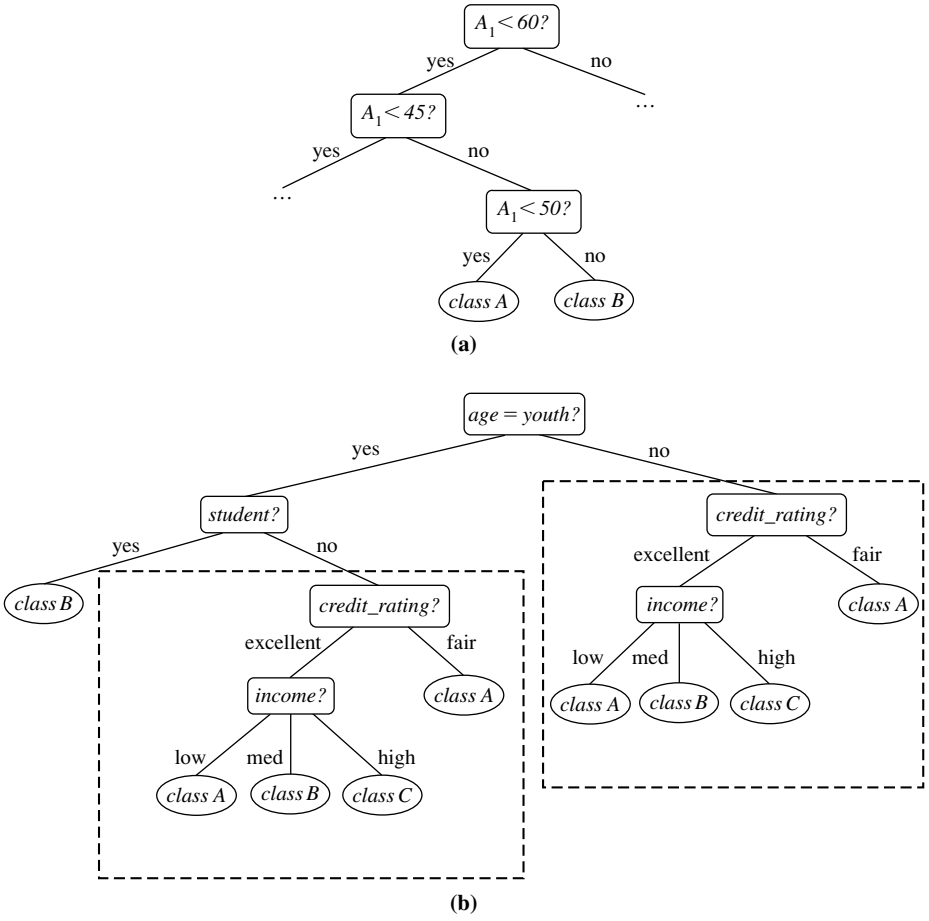

Figure 8.7 An example of: (a) subtree **repetition**, where an attribute is repeatedly tested along a given branch of the tree (e.g., *age*) and (b) subtree **replication**, where duplicate subtrees exist within a tree (e.g., the subtree headed by the node "*credit_rating?*").

followed by *"age < 45?,"* and so on). In **replication**, duplicate subtrees exist within the tree. These situations can impede the accuracy and comprehensibility of a decision tree. The use of multivariate splits (splits based on a combination of attributes) can prevent these problems. Another approach is to use a different form of knowledge representation, such as rules, instead of decision trees. This is described in Section 8.4.2, which shows how a *rule-based classifier* can be constructed by extracting IF-THEN rules from a decision tree.

8.2.4 Scalability and Decision Tree Induction

"What if D, the disk-resident training set of class-labeled tuples, does not fit in memory? In other words, how scalable is decision tree induction?" The efficiency of existing decision tree algorithms, such as ID3, C4.5, and CART, has been well established for relatively small data sets. Efficiency becomes an issue of concern when these algorithms are applied to the mining of very large real-world databases. The pioneering decision tree algorithms that we have discussed so far have the restriction that the training tuples should reside *in memory*.

In data mining applications, very large training sets of millions of tuples are common. Most often, the training data will not fit in memory! Therefore, decision tree construction becomes inefficient due to swapping of the training tuples in and out of main and cache memories. More scalable approaches, capable of handling training data that are too large to fit in memory, are required. Earlier strategies to "save space" included discretizing continuous-valued attributes and sampling data at each node. These techniques, however, still assume that the training set can fit in memory.

Several scalable decision tree induction methods have been introduced in recent studies. RainForest, for example, adapts to the amount of main memory available and applies to any decision tree induction algorithm. The method maintains an **AVC-set** (where "AVC" stands for "*Attribute-Value*, *Classlabel*") for each attribute, at each tree node, describing the training tuples at the node. The AVC-set of an attribute A at node N gives the class label counts for each value of A for the tuples at N. Figure 8.8 shows AVC-sets for the tuple data of Table 8.1. The set of all AVC-sets at a node N is the **AVC-group** of N. The size of an AVC-set for attribute A at node N depends only on the number of distinct values of A and the number of classes in the set of tuples at N. Typically, this size should fit in memory, even for real-world data. RainForest also has techniques, however, for handling the case where the AVC-group does not fit in memory. Therefore, the method has high scalability for decision tree induction in very large data sets.

BOAT (Bootstrapped Optimistic Algorithm for Tree construction) is a decision tree algorithm that takes a completely different approach to scalability—it is not based on the use of any special data structures. Instead, it uses a statistical technique known as "bootstrapping" (Section 8.5.4) to create several smaller samples (or subsets) of the given training data, each of which fits in memory. Each subset is used to construct a tree, resulting in several trees. The trees are examined and used to construct a new tree, T', that turns out to be "very close" to the tree that would have been generated if all the original training data had fit in memory.

age	*buys_computer* yes	no
youth	2	3
middle_aged	4	0
senior	3	2

income	*buys_computer* yes	no
low	3	1
medium	4	2
high	2	2

student	*buys_computer* yes	no
yes	6	1
no	3	4

credit_ratting	*buys_computer* yes	no
fair	6	2
excellent	3	3

Figure 8.8 The use of data structures to hold aggregate information regarding the training data (e.g., these AVC-sets describing Table 8.1's data) are one approach to improving the scalability of decision tree induction.

BOAT can use any attribute selection measure that selects binary splits and that is based on the notion of purity of partitions such as the Gini index. BOAT uses a lower bound on the attribute selection measure to detect if this "very good" tree, T', is different from the "real" tree, T, that would have been generated using all of the data. It refines T' to arrive at T.

BOAT usually requires only two scans of D. This is quite an improvement, even in comparison to traditional decision tree algorithms (e.g., the basic algorithm in Figure 8.3), which require one scan per tree level! BOAT was found to be two to three times faster than RainForest, while constructing exactly the same tree. An additional advantage of BOAT is that it can be used for incremental updates. That is, BOAT can take new insertions and deletions for the training data and update the decision tree to reflect these changes, without having to reconstruct the tree from scratch.

8.2.5 Visual Mining for Decision Tree Induction

"Are there any interactive approaches to decision tree induction that allow us to visualize the data and the tree as it is being constructed? Can we use any knowledge of our data to help in building the tree?" In this section, you will learn about an approach to decision tree induction that supports these options. **Perception-based classification (PBC)** is an interactive approach based on multidimensional visualization techniques and allows the user to incorporate background knowledge about the data when building a decision tree. By visually interacting with the data, the user is also likely to develop a deeper understanding of the data. The resulting trees tend to be smaller than those built using traditional decision tree induction methods and so are easier to interpret, while achieving about the same accuracy.

"*How can the data be visualized to support interactive decision tree construction?*" PBC uses a pixel-oriented approach to view multidimensional data with its class label

information. The circle segments approach is adapted, which maps d-dimensional data objects to a circle that is partitioned into d segments, each representing one attribute (Section 2.3.1). Here, an attribute value of a data object is mapped to one colored pixel, reflecting the object's class label. This mapping is done for each attribute–value pair of each data object. Sorting is done for each attribute to determine the arrangement order within a segment. For example, attribute values within a given segment may be organized so as to display homogeneous (with respect to class label) regions within the same attribute value. The amount of training data that can be visualized at one time is approximately determined by the product of the number of attributes and the number of data objects.

The PBC system displays a split screen, consisting of a Data Interaction window and a Knowledge Interaction window (Figure 8.9). The Data Interaction window displays the circle segments of the data under examination, while the Knowledge Interaction window displays the decision tree constructed so far. Initially, the complete training set is visualized in the Data Interaction window, while the Knowledge Interaction window displays an empty decision tree.

Traditional decision tree algorithms allow only binary splits for numeric attributes. PBC, however, allows the user to specify multiple split-points, resulting in multiple branches to be grown from a single tree node.

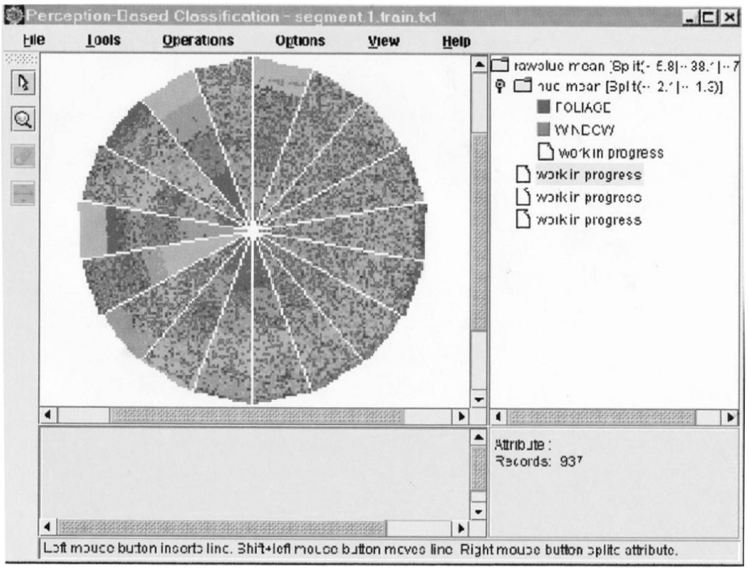

Figure 8.9 A screenshot of PBC, a system for interactive decision tree construction. Multidimensional training data are viewed as circle segments in the Data Interaction window (*left*). The Knowledge Interaction window (*right*) displays the current decision tree. *Source:* From Ankerst, Elsen, Ester, and Kriegel [AEEK99].

A tree is interactively constructed as follows. The user visualizes the multidimensional data in the Data Interaction window and selects a splitting attribute and one or more split-points. The current decision tree in the Knowledge Interaction window is expanded. The user selects a node of the decision tree. The user may either assign a class label to the node (which makes the node a leaf) or request the visualization of the training data corresponding to the node. This leads to a new visualization of every attribute except the ones used for splitting criteria on the same path from the root. The interactive process continues until a class has been assigned to each leaf of the decision tree.

The trees constructed with PBC were compared with trees generated by the CART, C4.5, and SPRINT algorithms from various data sets. The trees created with PBC were of comparable accuracy with the tree from the algorithmic approaches, yet were significantly smaller and, thus, easier to understand. Users can use their domain knowledge in building a decision tree, but also gain a deeper understanding of their data during the construction process.

8.3 Bayes Classification Methods

"What are Bayesian classifiers?" Bayesian classifiers are statistical classifiers. They can predict class membership probabilities such as the probability that a given tuple belongs to a particular class.

Bayesian classification is based on Bayes' theorem, described next. Studies comparing classification algorithms have found a simple Bayesian classifier known as the *naïve Bayesian classifier* to be comparable in performance with decision tree and selected neural network classifiers. Bayesian classifiers have also exhibited high accuracy and speed when applied to large databases.

Naïve Bayesian classifiers assume that the effect of an attribute value on a given class is independent of the values of the other attributes. This assumption is called *class-conditional independence*. It is made to simplify the computations involved and, in this sense, is considered "naïve."

Section 8.3.1 reviews basic probability notation and Bayes' theorem. In Section 8.3.2 you will learn how to do naïve Bayesian classification.

8.3.1 Bayes' Theorem

Bayes' theorem is named after Thomas Bayes, a nonconformist English clergyman who did early work in probability and decision theory during the 18th century. Let $\boldsymbol{X}$ be a data tuple. In Bayesian terms, $\boldsymbol{X}$ is considered "evidence." As usual, it is described by measurements made on a set of n attributes. Let H be some hypothesis such as that the data tuple $\boldsymbol{X}$ belongs to a specified class C. For classification problems, we want to determine $P(H|\boldsymbol{X})$, the probability that the hypothesis H holds given the "evidence" or observed data tuple $\boldsymbol{X}$. In other words, we are looking for the probability that tuple $\boldsymbol{X}$ belongs to class C, given that we know the attribute description of $\boldsymbol{X}$.

$P(H|\boldsymbol{X})$ is the **posterior probability**, or *a posteriori probability*, of H conditioned on $\boldsymbol{X}$. For example, suppose our world of data tuples is confined to customers described by the attributes *age* and *income*, respectively, and that $\boldsymbol{X}$ is a 35-year-old customer with an income of $40,000. Suppose that H is the hypothesis that our customer will buy a computer. Then $P(H|\boldsymbol{X})$ reflects the probability that customer $\boldsymbol{X}$ will buy a computer given that we know the customer's age and income.

In contrast, $P(H)$ is the **prior probability**, or *a priori probability*, of H. For our example, this is the probability that any given customer will buy a computer, regardless of age, income, or any other information, for that matter. The posterior probability, $P(H|\boldsymbol{X})$, is based on more information (e.g., customer information) than the prior probability, $P(H)$, which is independent of $\boldsymbol{X}$.

Similarly, $P(\boldsymbol{X}|H)$ is the posterior probability of $\boldsymbol{X}$ conditioned on H. That is, it is the probability that a customer, $\boldsymbol{X}$, is 35 years old and earns $40,000, given that we know the customer will buy a computer.

$P(\boldsymbol{X})$ is the prior probability of $\boldsymbol{X}$. Using our example, it is the probability that a person from our set of customers is 35 years old and earns $40,000.

"*How are these probabilities estimated?*" $P(H)$, $P(\boldsymbol{X}|H)$, and $P(\boldsymbol{X})$ may be estimated from the given data, as we shall see next. **Bayes' theorem** is useful in that it provides a way of calculating the posterior probability, $P(H|\boldsymbol{X})$, from $P(H)$, $P(\boldsymbol{X}|H)$, and $P(\boldsymbol{X})$. Bayes' theorem is

$$P(H|\boldsymbol{X}) = \frac{P(\boldsymbol{X}|H)P(H)}{P(\boldsymbol{X})}. \tag{8.10}$$

Now that we have that out of the way, in the next section, we will look at how Bayes' theorem is used in the naïve Bayesian classifier.

8.3.2 Naïve Bayesian Classification

The **naïve Bayesian** classifier, or **simple Bayesian** classifier, works as follows:

1. Let D be a training set of tuples and their associated class labels. As usual, each tuple is represented by an n-dimensional attribute vector, $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$, depicting n measurements made on the tuple from n attributes, respectively, $A_1, A_2, \ldots, A_n$.

2. Suppose that there are m classes, $C_1, C_2, \ldots, C_m$. Given a tuple, $\boldsymbol{X}$, the classifier will predict that $\boldsymbol{X}$ belongs to the class having the highest posterior probability, conditioned on $\boldsymbol{X}$. That is, the naïve Bayesian classifier predicts that tuple $\boldsymbol{X}$ belongs to the class C_i if and only if

$$P(C_i|\boldsymbol{X}) > P(C_j|\boldsymbol{X}) \quad \text{for } 1 \leq j \leq m, j \neq i.$$

Thus, we maximize $P(C_i|\boldsymbol{X})$. The class C_i for which $P(C_i|\boldsymbol{X})$ is maximized is called the *maximum posteriori hypothesis*. By Bayes' theorem (Eq. 8.10),

$$P(C_i|\boldsymbol{X}) = \frac{P(\boldsymbol{X}|C_i)P(C_i)}{P(\boldsymbol{X})}. \tag{8.11}$$

3. As $P(\boldsymbol{X})$ is constant for all classes, only $P(\boldsymbol{X}|C_i)P(C_i)$ needs to be maximized. If the class prior probabilities are not known, then it is commonly assumed that the classes are equally likely, that is, $P(C_1) = P(C_2) = \cdots = P(C_m)$, and we would therefore maximize $P(\boldsymbol{X}|C_i)$. Otherwise, we maximize $P(\boldsymbol{X}|C_i)P(C_i)$. Note that the class prior probabilities may be estimated by $P(C_i) = |C_{i,D}|/|D|$, where $|C_{i,D}|$ is the number of training tuples of class C_i in D.

4. Given data sets with many attributes, it would be extremely computationally expensive to compute $P(\boldsymbol{X}|C_i)$. To reduce computation in evaluating $P(\boldsymbol{X}|C_i)$, the naïve assumption of **class-conditional independence** is made. This presumes that the attributes' values are conditionally independent of one another, given the class label of the tuple (i.e., that there are no dependence relationships among the attributes). Thus,

$$P(\boldsymbol{X}|C_i) = \prod_{k=1}^{n} P(x_k|C_i) \tag{8.12}$$
$$= P(x_1|C_i) \times P(x_2|C_i) \times \cdots \times P(x_n|C_i).$$

We can easily estimate the probabilities $P(x_1|C_i), P(x_2|C_i), \ldots, P(x_n|C_i)$ from the training tuples. Recall that here x_k refers to the value of attribute A_k for tuple $\boldsymbol{X}$. For each attribute, we look at whether the attribute is categorical or continuous-valued. For instance, to compute $P(\boldsymbol{X}|C_i)$, we consider the following:

(a) If A_k is categorical, then $P(x_k|C_i)$ is the number of tuples of class C_i in D having the value x_k for A_k, divided by $|C_{i,D}|$, the number of tuples of class C_i in D.

(b) If A_k is continuous-valued, then we need to do a bit more work, but the calculation is pretty straightforward. A continuous-valued attribute is typically assumed to have a Gaussian distribution with a mean μ and standard deviation σ, defined by

$$g(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{8.13}$$

so that

$$P(x_k|C_i) = g(x_k, \mu_{C_i}, \sigma_{C_i}). \tag{8.14}$$

These equations may appear daunting, but hold on! We need to compute μ_{C_i} and σ_{C_i}, which are the mean (i.e., average) and standard deviation, respectively, of the values of attribute A_k for training tuples of class C_i. We then plug these two quantities into Eq. (8.13), together with x_k, to estimate $P(x_k|C_i)$.

For example, let $\boldsymbol{X} = (35, \$40{,}000)$, where A_1 and A_2 are the attributes *age* and *income*, respectively. Let the class label attribute be *buys_computer*. The associated class label for $\boldsymbol{X}$ is *yes* (i.e., *buys_computer* = *yes*). Let's suppose that *age* has not been discretized and therefore exists as a continuous-valued attribute. Suppose that from the training set, we find that customers in D who buy a computer are

38 ± 12 years of age. In other words, for attribute *age* and this class, we have $\mu = 38$ years and $\sigma = 12$. We can plug these quantities, along with $x_1 = 35$ for our tuple $\boldsymbol{X}$, into Eq. (8.13) to estimate $P(age = 35|buys\_computer = yes)$. For a quick review of mean and standard deviation calculations, please see Section 2.2.

5. To predict the class label of $\boldsymbol{X}$, $P(\boldsymbol{X}|C_i)P(C_i)$ is evaluated for each class C_i. The classifier predicts that the class label of tuple $\boldsymbol{X}$ is the class C_i if and only if

$$P(\boldsymbol{X}|C_i)P(C_i) > P(\boldsymbol{X}|C_j)P(C_j) \quad \text{for } 1 \leq j \leq m, j \neq i. \tag{8.15}$$

In other words, the predicted class label is the class C_i for which $P(\boldsymbol{X}|C_i)P(C_i)$ is the maximum.

"How effective are Bayesian classifiers?" Various empirical studies of this classifier in comparison to decision tree and neural network classifiers have found it to be comparable in some domains. In theory, Bayesian classifiers have the minimum error rate in comparison to all other classifiers. However, in practice this is not always the case, owing to inaccuracies in the assumptions made for its use, such as class-conditional independence, and the lack of available probability data.

Bayesian classifiers are also useful in that they provide a theoretical justification for other classifiers that do not explicitly use Bayes' theorem. For example, under certain assumptions, it can be shown that many neural network and curve-fitting algorithms output the *maximum posteriori* hypothesis, as does the naïve Bayesian classifier.

Example 8.4 Predicting a class label using naïve Bayesian classification. We wish to predict the class label of a tuple using naïve Bayesian classification, given the same training data as in Example 8.3 for decision tree induction. The training data were shown earlier in Table 8.1. The data tuples are described by the attributes *age*, *income*, *student*, and *credit_rating*. The class label attribute, *buys_computer*, has two distinct values (namely, {*yes, no*}). Let C_1 correspond to the class *buys_computer* = *yes* and C_2 correspond to *buys_computer* = *no*. The tuple we wish to classify is

$$\boldsymbol{X} = (age = youth, income = medium, student = yes, credit\_rating = fair)$$

We need to maximize $P(\boldsymbol{X}|C_i)P(C_i)$, for $i = 1, 2$. $P(C_i)$, the prior probability of each class, can be computed based on the training tuples:

$$P(buys\_computer = yes) = 9/14 = 0.643$$
$$P(buys\_computer = no) = 5/14 = 0.357$$

To compute $P(\boldsymbol{X}|C_i)$, for $i = 1, 2$, we compute the following conditional probabilities:

$$P(age = youth \mid buys\_computer = yes) = 2/9 = 0.222$$
$$P(age = youth \mid buys\_computer = no) = 3/5 = 0.600$$
$$P(income = medium \mid buys\_computer = yes) = 4/9 = 0.444$$
$$P(income = medium \mid buys\_computer = no) = 2/5 = 0.400$$
$$P(student = yes \mid buys\_computer = yes) = 6/9 = 0.667$$

$$P(student = yes \mid buys\_computer = no) = 1/5 = 0.200$$
$$P(credit\_rating = fair \mid buys\_computer = yes) = 6/9 = 0.667$$
$$P(credit\_rating = fair \mid buys\_computer = no) = 2/5 = 0.400$$

Using these probabilities, we obtain

$$\begin{aligned} P(\mathbf{X}|buys\_computer = yes) &= P(age = youth \mid buys\_computer = yes) \\ &\quad \times P(income = medium \mid buys\_computer = yes) \\ &\quad \times P(student = yes \mid buys\_computer = yes) \\ &\quad \times P(credit\_rating = fair \mid buys\_computer = yes) \\ &= 0.222 \times 0.444 \times 0.667 \times 0.667 = 0.044. \end{aligned}$$

Similarly,

$$P(\mathbf{X}|buys\_computer = no) = 0.600 \times 0.400 \times 0.200 \times 0.400 = 0.019.$$

To find the class, C_i, that maximizes $P(\mathbf{X}|C_i)P(C_i)$, we compute

$$P(\mathbf{X}|buys\_computer = yes)P(buys\_computer = yes) = 0.044 \times 0.643 = 0.028$$
$$P(\mathbf{X}|buys\_computer = no)P(buys\_computer = no) = 0.019 \times 0.357 = 0.007$$

Therefore, the naïve Bayesian classifier predicts *buys_computer* = *yes* for tuple $\mathbf{X}$. ■

"*What if I encounter probability values of zero?*" Recall that in Eq. (8.12), we estimate $P(\mathbf{X}|C_i)$ as the product of the probabilities $P(x_1|C_i)$, $P(x_2|C_i), \ldots, P(x_n|C_i)$, based on the assumption of class-conditional independence. These probabilities can be estimated from the training tuples (step 4). We need to compute $P(\mathbf{X}|C_i)$ for *each* class ($i = 1, 2, \ldots, m$) to find the class C_i for which $P(\mathbf{X}|C_i)P(C_i)$ is the maximum (step 5). Let's consider this calculation. For each attribute–value pair (i.e., $A_k = x_k$, for $k = 1, 2, \ldots, n$) in tuple $\mathbf{X}$, we need to count the number of tuples having that attribute–value pair, per class (i.e., per C_i, for $i = 1, \ldots, m$). In Example 8.4, we have two classes ($m = 2$), namely *buys_computer* = *yes* and *buys_computer* = *no*. Therefore, for the attribute–value pair *student* = *yes* of $\mathbf{X}$, say, we need two counts—the number of customers who are students and for which *buys_computer* = *yes* (which contributes to $P(\mathbf{X}|buys\_computer = yes)$) and the number of customers who are students and for which *buys_computer* = *no* (which contributes to $P(\mathbf{X}|buys\_computer = no)$).

But what if, say, there are no training tuples representing students for the class *buys_computer* = *no*, resulting in $P(student = yes|buys\_computer = no) = 0$? In other words, what happens if we should end up with a probability value of zero for some $P(x_k|C_i)$? Plugging this zero value into Eq. (8.12) would return a zero probability for $P(\mathbf{X}|C_i)$, even though, without the zero probability, we may have ended up with a high probability, suggesting that $\mathbf{X}$ belonged to class C_i! A zero probability cancels the effects of all the other (posteriori) probabilities (on C_i) involved in the product.

There is a simple trick to avoid this problem. We can assume that our training database, D, is so large that adding one to each count that we need would only make a negligible difference in the estimated probability value, yet would conveniently avoid the

case of probability values of zero. This technique for probability estimation is known as the **Laplacian correction** or **Laplace estimator**, named after Pierre Laplace, a French mathematician who lived from 1749 to 1827. If we have, say, q counts to which we each add one, then we must remember to add q to the corresponding denominator used in the probability calculation. We illustrate this technique in Example 8.5.

Example 8.5 **Using the Laplacian correction to avoid computing probability values of zero.** Suppose that for the class *buys_computer* = *yes* in some training database, D, containing 1000 tuples, we have 0 tuples with *income* = *low*, 990 tuples with *income* = *medium*, and 10 tuples with *income* = *high*. The probabilities of these events, without the Laplacian correction, are 0, 0.990 (from 990/1000), and 0.010 (from 10/1000), respectively. Using the Laplacian correction for the three quantities, we pretend that we have 1 more tuple for each income-value pair. In this way, we instead obtain the following probabilities (rounded up to three decimal places):

$$\frac{1}{1003} = 0.001, \frac{991}{1003} = 0.988, \text{ and } \frac{11}{1003} = 0.011,$$

respectively. The "corrected" probability estimates are close to their "uncorrected" counterparts, yet the zero probability value is avoided. ■

8.4 Rule-Based Classification

In this section, we look at rule-based classifiers, where the learned model is represented as a set of IF-THEN rules. We first examine how such rules are used for classification (Section 8.4.1). We then study ways in which they can be generated, either from a decision tree (Section 8.4.2) or directly from the training data using a *sequential covering algorithm* (Section 8.4.3).

8.4.1 Using IF-THEN Rules for Classification

Rules are a good way of representing information or bits of knowledge. A **rule-based classifier** uses a set of IF-THEN rules for classification. An **IF-THEN** rule is an expression of the form

IF *condition* THEN *conclusion*.

An example is rule $R1$,

$R1$: IF *age* = *youth* AND *student* = *yes* THEN *buys_computer* = *yes*.

The "IF" part (or left side) of a rule is known as the **rule antecedent** or **precondition**. The "THEN" part (or right side) is the **rule consequent**. In the rule antecedent, the condition consists of one or more *attribute tests* (e.g., *age* = *youth* and *student* = *yes*)

that are logically ANDed. The rule's consequent contains a class prediction (in this case, we are predicting whether a customer will buy a computer). $R1$ can also be written as

$$R1\text{: } (age = youth) \wedge (student = yes) \Rightarrow (buys\_computer = yes).$$

If the condition (i.e., all the attribute tests) in a rule antecedent holds true for a given tuple, we say that the rule antecedent is **satisfied** (or simply, that the rule is satisfied) and that the rule **covers** the tuple.

A rule R can be assessed by its coverage and accuracy. Given a tuple, $\boldsymbol{X}$, from a class-labeled data set, D, let n_{covers} be the number of tuples covered by R; $n_{correct}$ be the number of tuples correctly classified by R; and $|D|$ be the number of tuples in D. We can define the **coverage** and **accuracy** of R as

$$coverage(R) = \frac{n_{covers}}{|D|} \tag{8.16}$$

$$accuracy(R) = \frac{n_{correct}}{n_{covers}}. \tag{8.17}$$

That is, a rule's coverage is the percentage of tuples that are covered by the rule (i.e., their attribute values hold true for the rule's antecedent). For a rule's accuracy, we look at the tuples that it covers and see what percentage of them the rule can correctly classify.

Example 8.6 **Rule accuracy and coverage.** Let's go back to our data in Table 8.1. These are class-labeled tuples from the *AllElectronics* customer database. Our task is to predict whether a customer will buy a computer. Consider rule $R1$, which covers 2 of the 14 tuples. It can correctly classify both tuples. Therefore, $coverage(R1) = 2/14 = 14.28\%$ and $accuracy(R1) = 2/2 = 100\%$. ■

Let's see how we can use rule-based classification to predict the class label of a given tuple, $\boldsymbol{X}$. If a rule is satisfied by $\boldsymbol{X}$, the rule is said to be **triggered**. For example, suppose we have

$$\boldsymbol{X} = (age = youth,\ income = medium,\ student = yes,\ credit\_rating = fair).$$

We would like to classify $\boldsymbol{X}$ according to *buys_computer*. $\boldsymbol{X}$ satisfies $R1$, which triggers the rule.

If $R1$ is the only rule satisfied, then the rule **fires** by returning the class prediction for $\boldsymbol{X}$. Note that triggering does not always mean firing because there may be more than one rule that is satisfied! If more than one rule is triggered, we have a potential problem. What if they each specify a different class? Or what if no rule is satisfied by $\boldsymbol{X}$?

We tackle the first question. If more than one rule is triggered, we need a **conflict resolution strategy** to figure out which rule gets to fire and assign its class prediction to $\boldsymbol{X}$. There are many possible strategies. We look at two, namely *size ordering* and *rule ordering*.

The **size ordering** scheme assigns the highest priority to the triggering rule that has the "toughest" requirements, where toughness is measured by the rule antecedent *size*. That is, the triggering rule with the most attribute tests is fired.

The **rule ordering** scheme prioritizes the rules beforehand. The ordering may be *class-based* or *rule-based*. With **class-based ordering**, the classes are sorted in order of decreasing "importance" such as by decreasing *order of prevalence*. That is, all the rules for the most prevalent (or most frequent) class come first, the rules for the next prevalent class come next, and so on. Alternatively, they may be sorted based on the misclassification cost per class. Within each class, the rules are not ordered—they don't have to be because they all predict the same class (and so there can be no class conflict!).

With **rule-based ordering**, the rules are organized into one long priority list, according to some measure of rule quality, such as accuracy, coverage, or size (number of attribute tests in the rule antecedent), or based on advice from domain experts. When rule ordering is used, the rule set is known as a **decision list**. With rule ordering, the triggering rule that appears earliest in the list has the highest priority, and so it gets to fire its class prediction. Any other rule that satisfies $\boldsymbol{X}$ is ignored. Most rule-based classification systems use a class-based rule-ordering strategy.

Note that in the first strategy, overall the rules are *unordered*. They can be applied in any order when classifying a tuple. That is, a disjunction (logical OR) is implied between each of the rules. Each rule represents a standalone nugget or piece of knowledge. This is in contrast to the rule ordering (decision list) scheme for which rules must be applied in the prescribed order so as to avoid conflicts. Each rule in a decision list implies the negation of the rules that come before it in the list. Hence, rules in a decision list are more difficult to interpret.

Now that we have seen how we can handle conflicts, let's go back to the scenario where there is no rule satisfied by $\boldsymbol{X}$. How, then, can we determine the class label of $\boldsymbol{X}$? In this case, a fallback or **default rule** can be set up to specify a default class, based on a training set. This may be the class in majority or the majority class of the tuples that were not covered by any rule. The default rule is evaluated at the end, if and only if no other rule covers $\boldsymbol{X}$. The condition in the default rule is empty. In this way, the rule fires when no other rule is satisfied.

In the following sections, we examine how to build a rule-based classifier.

8.4.2 Rule Extraction from a Decision Tree

In Section 8.2, we learned how to build a decision tree classifier from a set of training data. Decision tree classifiers are a popular method of classification—it is easy to understand how decision trees work and they are known for their accuracy. Decision trees can become large and difficult to interpret. In this subsection, we look at how to build a rule-based classifier by extracting IF-THEN rules from a decision tree. In comparison with a decision tree, the IF-THEN rules may be easier for humans to understand, particularly if the decision tree is very large.

To extract rules from a decision tree, one rule is created for each path from the root to a leaf node. Each splitting criterion along a given path is logically ANDed to form the

rule antecedent ("IF" part). The leaf node holds the class prediction, forming the rule consequent ("THEN" part).

Example 8.7 Extracting classification rules from a decision tree. The decision tree of Figure 8.2 can be converted to classification IF-THEN rules by tracing the path from the root node to each leaf node in the tree. The rules extracted from Figure 8.2 are as follows:

*R*1: IF *age* = *youth*	AND *student* = *no*	THEN *buys_computer* = *no*
*R*2: IF *age* = *youth*	AND *student* = *yes*	THEN *buys_computer* = *yes*
*R*3: IF *age* = *middle_aged*		THEN *buys_computer* = *yes*
*R*4: IF *age* = *senior*	AND *credit_rating* = *excellent*	THEN *buys_computer* = *yes*
*R*5: IF *age* = *senior*	AND *credit_rating* = *fair*	THEN *buys_computer* = *no*

■

A disjunction (logical OR) is implied between each of the extracted rules. Because the rules are extracted directly from the tree, they are **mutually exclusive** and **exhaustive**. *Mutually exclusive* means that we cannot have rule conflicts here because no two rules will be triggered for the same tuple. (We have one rule per leaf, and any tuple can map to only one leaf.) *Exhaustive* means there is one rule for each possible attribute–value combination, so that this set of rules does not require a default rule. Therefore, the order of the rules does not matter—they are *unordered*.

Since we end up with one rule per leaf, the set of extracted rules is not much simpler than the corresponding decision tree! The extracted rules may be even more difficult to interpret than the original trees in some cases. As an example, Figure 8.7 showed decision trees that suffer from subtree repetition and replication. The resulting set of rules extracted can be large and difficult to follow, because some of the attribute tests may be irrelevant or redundant. So, the plot thickens. Although it is easy to extract rules from a decision tree, we may need to do some more work by pruning the resulting rule set.

"*How can we prune the rule set?*" For a given rule antecedent, any condition that does not improve the estimated accuracy of the rule can be pruned (i.e., removed), thereby generalizing the rule. C4.5 extracts rules from an unpruned tree, and then prunes the rules using a pessimistic approach similar to its tree pruning method. The training tuples and their associated class labels are used to estimate rule accuracy. However, because this would result in an optimistic estimate, alternatively, the estimate is adjusted to compensate for the bias, resulting in a pessimistic estimate. In addition, any rule that does not contribute to the overall accuracy of the entire rule set can also be pruned.

Other problems arise during rule pruning, however, as the rules *will no longer be* mutually exclusive and exhaustive. For conflict resolution, C4.5 adopts a **class-based ordering scheme**. It groups together all rules for a single class, and then determines a ranking of these class rule sets. Within a rule set, the rules are not ordered. C4.5 orders the class rule sets so as to minimize the number of *false-positive errors* (i.e., where a rule predicts a class, *C*, but the actual class is not *C*). The class rule set with the least number of false positives is examined first. Once pruning is complete, a final check is

done to remove any duplicates. When choosing a default class, C4.5 does not choose the majority class, because this class will likely have many rules for its tuples. Instead, it selects the class that contains the most training tuples that were not covered by any rule.

8.4.3 Rule Induction Using a Sequential Covering Algorithm

IF-THEN rules can be extracted directly from the training data (i.e., without having to generate a decision tree first) using a **sequential covering algorithm**. The name comes from the notion that the rules are learned *sequentially* (one at a time), where each rule for a given class will ideally *cover* many of the class's tuples (and hopefully none of the tuples of other classes). Sequential covering algorithms are the most widely used approach to mining disjunctive sets of classification rules, and form the topic of this subsection.

There are many sequential covering algorithms. Popular variations include AQ, CN2, and the more recent RIPPER. The general strategy is as follows. Rules are learned one at a time. Each time a rule is learned, the tuples covered by the rule are removed, and the process repeats on the remaining tuples. This sequential learning of rules is in contrast to decision tree induction. Because the path to each leaf in a decision tree corresponds to a rule, we can consider decision tree induction as learning a set of rules *simultaneously*.

A basic sequential covering algorithm is shown in Figure 8.10. Here, rules are learned for one class at a time. Ideally, when learning a rule for a class, C, we would like the rule to cover all (or many) of the training tuples of class C and none (or few) of the tuples

Algorithm: Sequential covering. Learn a set of IF-THEN rules for classification.

Input:

- D, a data set of class-labeled tuples;
- *Att_vals*, the set of all attributes and their possible values.

Output: A set of IF-THEN rules.

Method:

```
(1)  Rule_set = {}; // initial set of rules learned is empty
(2)  for each class c do
(3)      repeat
(4)          Rule = Learn_One_Rule(D, Att_vals, c);
(5)          remove tuples covered by Rule from D;
(6)          Rule_set = Rule_set + Rule; // add new rule to rule set
(7)      until terminating condition;
(8)  endfor
(9)  return Rule_Set;
```

Figure 8.10 Basic sequential covering algorithm.

from other classes. In this way, the rules learned should be of high accuracy. The rules need not necessarily be of high coverage. This is because we can have more than one rule for a class, so that different rules may cover different tuples within the same class. The process continues until the terminating condition is met, such as when there are no more training tuples or the quality of a rule returned is below a user-specified threshold. The *Learn_One_Rule* procedure finds the "best" rule for the current class, given the current set of training tuples.

"*How are rules learned?*" Typically, rules are grown in a *general-to-specific* manner (Figure 8.11). We can think of this as a beam search, where we start off with an empty rule and then gradually keep appending attribute tests to it. We append by adding the attribute test as a logical conjunct to the existing condition of the rule antecedent. Suppose our training set, *D*, consists of loan application data. Attributes regarding each applicant include their age, income, education level, residence, credit rating, and the term of the loan. The classifying attribute is *loan_decision*, which indicates whether a loan is accepted (considered safe) or rejected (considered risky). To learn a rule for the class "accept," we start off with the most general rule possible, that is, the condition of the rule antecedent is empty. The rule is

IF THEN *loan_decision* = *accept*.

We then consider each possible attribute test that may be added to the rule. These can be derived from the parameter *Att_vals*, which contains a list of attributes with their associated values. For example, for an attribute–value pair (*att*, *val*), we can consider attribute tests such as $att = val$, $att \leq val$, $att > val$, and so on. Typically, the training data will contain many attributes, each of which may have several possible values. Finding an optimal rule set becomes computationally explosive. Instead, *Learn_One_Rule*

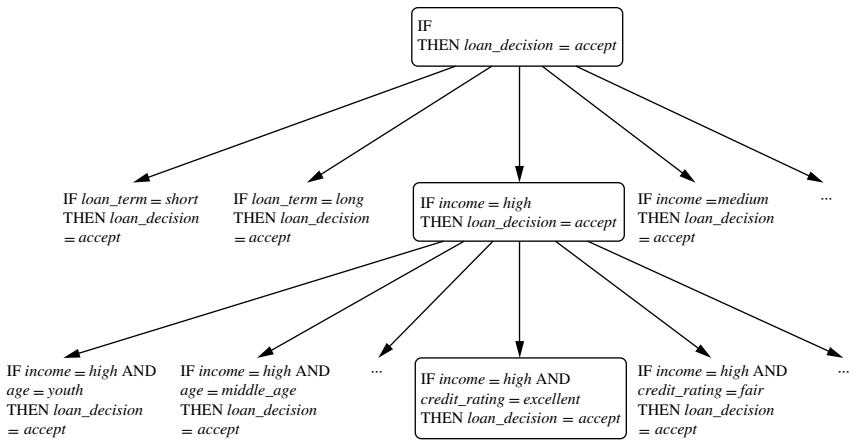

Figure 8.11 A general-to-specific search through rule space.

adopts a greedy depth-first strategy. Each time it is faced with adding a new attribute test (conjunct) to the current rule, it picks the one that most improves the rule quality, based on the training samples. We will say more about rule quality measures in a minute. For the moment, let's say we use rule accuracy as our quality measure. Getting back to our example with Figure 8.11, suppose *Learn_One_Rule* finds that the attribute test *income* = *high* best improves the accuracy of our current (empty) rule. We append it to the condition, so that the current rule becomes

IF *income* = *high* THEN *loan_decision* = *accept*.

Each time we add an attribute test to a rule, the resulting rule should cover relatively more of the "accept" tuples. During the next iteration, we again consider the possible attribute tests and end up selecting *credit_rating* = *excellent*. Our current rule grows to become

IF *income* = *high* AND *credit_rating* = *excellent* THEN *loan_decision* = *accept*.

The process repeats, where at each step we continue to greedily grow rules until the resulting rule meets an acceptable quality level.

Greedy search does not allow for backtracking. At each step, we *heuristically* add what appears to be the best choice at the moment. What if we unknowingly made a poor choice along the way? To lessen the chance of this happening, instead of selecting the best attribute test to append to the current rule, we can select the best k attribute tests. In this way, we perform a beam search of width k, wherein we maintain the k best candidates overall at each step, rather than a single best candidate.

Rule Quality Measures

Learn_One_Rule needs a measure of rule quality. Every time it considers an attribute test, it must check to see if appending such a test to the current rule's condition will result in an improved rule. Accuracy may seem like an obvious choice at first, but consider Example 8.8.

Example 8.8 **Choosing between two rules based on accuracy.** Consider the two rules as illustrated in Figure 8.12. Both are for the class *loan_decision* = *accept*. We use "*a*" to represent the tuples of class "*accept*" and "*r*" for the tuples of class "*reject*." Rule *R*1 correctly classifies 38 of the 40 tuples it covers. Rule *R*2 covers only two tuples, which it correctly classifies. Their respective accuracies are 95% and 100%. Thus, *R*2 has greater accuracy than *R*1, but it is not the better rule because of its small coverage. ■

From this example, we see that accuracy on its own is not a reliable estimate of rule quality. Coverage on its own is not useful either—for a given class we could have a rule that covers many tuples, most of which belong to other classes! Thus, we seek other measures for evaluating rule quality, which may integrate aspects of accuracy and coverage. Here we will look at a few, namely *entropy*, another based on *information gain*, and a *statistical test* that considers coverage. For our discussion, suppose we are learning rules

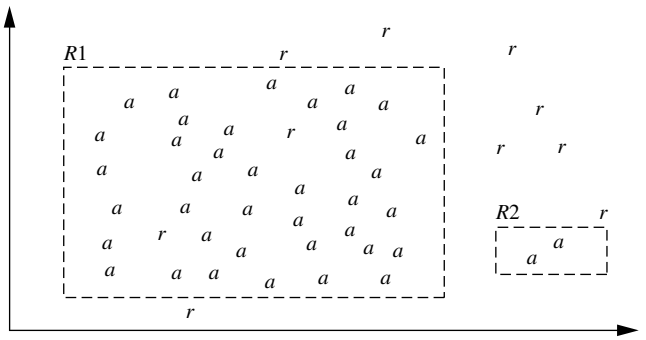

Figure 8.12 Rules for the class *loan_decision* = *accept*, showing *accept* (*a*) and *reject* (*r*) tuples.

for the class *c*. Our current rule is *R*: IF *condition* THEN *class* = *c*. We want to see if logically ANDing a given attribute test to *condition* would result in a better rule. We call the new condition, *condition*′, where *R*′: IF *condition*′ THEN *class* = *c* is our potential new rule. In other words, we want to see if *R*′ is any better than *R*.

We have already seen entropy in our discussion of the information gain measure used for attribute selection in decision tree induction (Section 8.2.2, Eq. 8.1). It is also known as the *expected information* needed to classify a tuple in data set, *D*. Here, *D* is the set of tuples covered by *condition*′ and p_i is the probability of class C_i in *D*. The lower the entropy, the better *condition*′ is. Entropy prefers conditions that cover a large number of tuples of a single class and few tuples of other classes.

Another measure is based on information gain and was proposed in **FOIL** (First Order Inductive Learner), a sequential covering algorithm that learns first-order logic rules. Learning first-order rules is more complex because such rules contain variables, whereas the rules we are concerned with in this section are propositional (i.e., variable-free).[5] In machine learning, the tuples of the class for which we are learning rules are called *positive* tuples, while the remaining tuples are *negative*. Let *pos* (*neg*) be the number of positive (negative) tuples covered by *R*. Let *pos*′ (*neg*′) be the number of positive (negative) tuples covered by *R*′. FOIL assesses the information gained by extending *condition*′ as

$$FOIL\_Gain = pos' \times \left(\log_2 \frac{pos'}{pos' + neg'} - log_2 \frac{pos}{pos + neg} \right). \tag{8.18}$$

It favors rules that have high accuracy and cover many positive tuples.

We can also use a statistical test of significance to determine if the apparent effect of a rule is not attributed to chance but instead indicates a genuine correlation between

[5]Incidentally, FOIL was also proposed by Quinlan, the father of ID3.

attribute values and classes. The test compares the observed distribution among classes of tuples covered by a rule with the expected distribution that would result if the rule made predictions at random. We want to assess whether any observed differences between these two distributions may be attributed to chance. We can use the **likelihood ratio statistic**,

$$Likelihood\_Ratio = 2\sum_{i=1}^{m} f_i \log\left(\frac{f_i}{e_i}\right), \tag{8.19}$$

where m is the number of classes.

For tuples satisfying the rule, f_i is the observed frequency of each class i among the tuples. e_i is what we would expect the frequency of each class i to be if the rule made random predictions. The statistic has a χ^2 distribution with $m-1$ degrees of freedom. The higher the likelihood ratio, the more likely that there is a *significant* difference in the number of correct predictions made by our rule in comparison with a "random guessor." That is, the performance of our rule is not due to chance. The ratio helps identify rules with insignificant coverage.

CN2 uses entropy together with the likelihood ratio test, while FOIL's information gain is used by RIPPER.

Rule Pruning

Learn_One_Rule does not employ a test set when evaluating rules. Assessments of rule quality as described previously are made with tuples from the original training data. These assessments are optimistic because the rules will likely overfit the data. That is, the rules may perform well on the training data, but less well on subsequent data. To compensate for this, we can prune the rules. A rule is pruned by removing a conjunct (attribute test). We choose to prune a rule, R, if the pruned version of R has greater quality, as assessed on an independent set of tuples. As in decision tree pruning, we refer to this set as a *pruning set*. Various pruning strategies can be used such as the pessimistic pruning approach described in the previous section.

FOIL uses a simple yet effective method. Given a rule, R,

$$FOIL\_Prune(R) = \frac{pos - neg}{pos + neg}, \tag{8.20}$$

where *pos* and *neg* are the number of positive and negative tuples covered by R, respectively. This value will increase with the accuracy of R on a pruning set. Therefore, if the *FOIL_Prune* value is higher for the pruned version of R, then we prune R.

By convention, RIPPER starts with the most recently added conjunct when considering pruning. Conjuncts are pruned one at a time as long as this results in an improvement.