CHAPTER

Pattern mining: basic concepts and methods

4

Frequent patterns are patterns (e.g., itemsets, subsequences, or substructures) that appear frequently in a data set. For example, a set of items, such as milk and bread, that appear frequently together in a transaction data set is a *frequent itemset*. A subsequence, such as buying first a smartphone, then a smart TV, and then a smart home device, if it occurs frequently in a shopping history database, is a (*frequent*) *sequential pattern*. A *substructure* can refer to different structural forms, such as subgraphs, subtrees, or sublattices. If a substructure occurs frequently, it is called a (*frequent*) *structured pattern*. Finding frequent patterns plays an essential role in mining associations, correlations, and many other interesting relationships among data. Moreover, it helps in data classification, clustering, and other data mining tasks. Thus, frequent pattern mining has become an important data mining task and a focused theme in data mining research.

In this chapter, we introduce the basic concepts of frequent patterns, associations, and correlations (Section 4.1) and study how they can be mined efficiently (Section 4.2). We also discuss how to judge whether the patterns found are interesting (Section 4.3). In the subsequent chapter, we extend our discussion to advanced frequent pattern mining, including mining more complex forms of frequent patterns, and their applications.

4.1 Basic concepts

Frequent pattern mining uncovers recurring relationships in a given data set. This section introduces the basic concepts of frequent pattern mining for the discovery of interesting associations and correlations between itemsets in transactional and relational databases. We begin in Section 4.1.1 by presenting an example of market basket analysis, the earliest form of frequent pattern mining for association rules. The basic concepts of mining frequent patterns and associations are discussed in Section 4.1.2.

4.1.1 Market basket analysis: a motivating example

A set of items is referred to as an **itemset**.[1] Frequent itemset mining leads to the discovery of associations and correlations among items in large transactional or relational data sets. With massive amounts of data continuously being collected and stored, many industries are interested in mining such patterns from their databases. The discovery of interesting correlation relationships among huge amounts

[1] In the data mining research literature, "itemset" is more commonly used than "item set."

Data Mining. https://doi.org/10.1016/B978-0-12-811760-6.00014-X

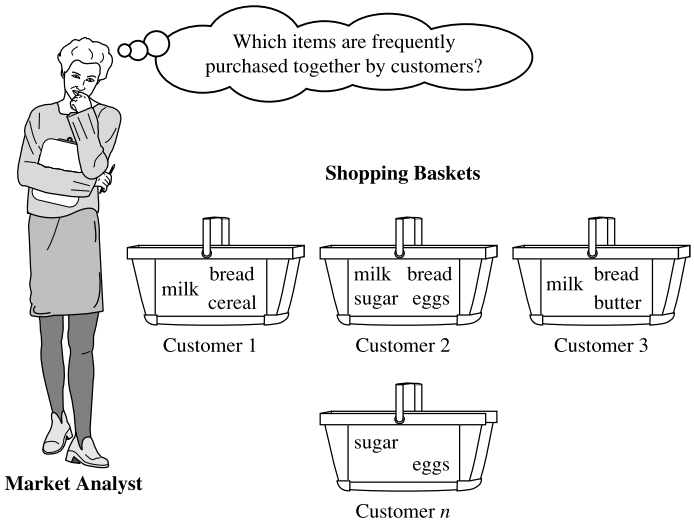


FIGURE 4.1

Market basket analysis.

of business transaction records can help in many business decision-making processes such as catalog design, cross-marketing, and customer shopping behavior analysis.

A typical example of frequent itemset mining is **market basket analysis**. This process analyzes customer buying habits by finding associations between the different items that customers place in their "shopping baskets" (Fig. 4.1). The discovery of these associations can help retailers develop marketing strategies by gaining insight into which items are frequently purchased together by customers. For instance, if customers are buying milk, how likely are they to also buy bread (and what kind of bread) on the same trip to the supermarket? This information can lead to increased sales, revenue, and customer acquisition by helping retailers do selective marketing and planned shelf space.

Let's look at an example of how market basket analysis can be useful.

Example 4.1. Market basket analysis. Suppose, as manager of a retail company, you would like to learn more about the buying habits of your customers. Specifically, you wonder, *"Which groups or sets of items are customers likely to purchase on a given trip to the store?"* To answer your question, market basket analysis may be performed on the retail data of customer transactions at your store. You can then use these results to choose marketing strategies and help create a new catalog. For instance, market basket analysis may help you design different store layouts. In one strategy, items that are frequently purchased together can be placed in proximity to further encourage the combined sale of such items. If customers who purchase computers also tend to buy antivirus software at the same time, then placing the hardware display close to the software display may help increase the sales of both items.

In an alternative strategy, placing hardware and software at opposite ends of the store may entice customers who purchase such items to pick up other items along the way. For instance, after deciding on an expensive computer, a customer may observe security systems for sale while heading toward the software display to purchase antivirus software and may decide to purchase a home security system as

well. Market basket analysis can also help retailers plan which items to put on sale at reduced prices. If customers tend to purchase computers and printers together, then reducing the prices on printers may encourage the sale of printers *as well as* computers. □

If we think of the universe as the set of items available at the store, then each item has a Boolean variable representing the presence or absence of that item. Each basket can then be represented by a Boolean vector of values assigned to these variables. The Boolean vectors can be analyzed to extract buying patterns that reflect items that are frequently *associated* or purchased together. These patterns can be represented in the form of **association rules**. For example, the information that customers who purchase computers also tend to buy antivirus software at the same time is represented in the following association rule:

$$computer \Rightarrow antivirus\_software \quad [support = 2\%, confidence = 60\%]. \tag{4.1}$$

Rule **support** and **confidence** are two measures of rule interestingness. They reflect the usefulness and certainty of discovered rules, respectively. A support of 2% for Rule (4.1) means that 2% of all the transactions under analysis show that computer and antivirus software are purchased together. A confidence of 60% means that 60% of the customers who purchased a computer also bought the software. Typically, association rules are considered interesting if they satisfy a **minimum support threshold** and a **minimum confidence threshold**. These thresholds can be set by users or domain experts. Additional analysis can be performed to discover interesting statistical correlations between associated items.

4.1.2 Frequent itemsets, closed itemsets, and association rules

Let $\mathcal{I} = \{I_1, I_2, \ldots, I_m\}$ be an itemset. Let D, the task-relevant data, be a set of database transactions where each transaction T is a nonempty itemset such that $T \subseteq \mathcal{I}$. Each transaction is associated with an identifier, called a *TID*. Let A be a set of items. A transaction T is said to contain A if $A \subseteq T$. An association rule is an implication of the form $A \Rightarrow B$, where $A \subset \mathcal{I}$, $B \subset \mathcal{I}$, $A \neq \emptyset$, $B \neq \emptyset$, and $A \cap B = \phi$. The rule $A \Rightarrow B$ holds in the transaction set D with **support** s, where s is the percentage of transactions in D that contain $A \cup B$ (i.e., the *union* of sets A and B say, or, both A and B). This is taken to be the probability, $P(A \cup B)$.[2] The rule $A \Rightarrow B$ has **confidence** c in the transaction set D, where c is the percentage of transactions in D containing A that also contain B. This is taken to be the conditional probability, $P(B|A)$. That is,

$$support\,(A \Rightarrow B) = P(A \cup B) \tag{4.2}$$

$$confidence\,(A \Rightarrow B) = P(B|A). \tag{4.3}$$

Rules that satisfy both a minimum support threshold (*min_sup*) and a minimum confidence threshold (*min_conf*) are called **strong**. By convention, support and confidence values are represented as percentages.

[2] Notice that the notation $P(A \cup B)$ indicates the probability that a transaction contains the *union* of sets A and B (i.e., it contains every item in A and B). This should not be confused with $P(A \; or \; B)$, which indicates the probability that a transaction contains either A or B.

An itemset that contains k items is a ***k*-itemset**. The set {*computer, antivirus_software*} is a 2-itemset. The **occurrence frequency of an itemset** is the number of transactions that contain the itemset. Occurrence frequency is also referred as the **frequency**, **support count**, or **count** of the itemset. Note that the itemset support defined in Eq. (4.2) is sometimes referred to as *relative support*, whereas the occurrence frequency is called the **absolute support**. If the relative support of an itemset I satisfies a prespecified **minimum support threshold** (i.e., the absolute support of I satisfies the corresponding **minimum support count threshold**), then I is a **frequent** itemset.[3] The set of frequent k-itemsets is commonly denoted by L_k.[4]

From Eq. (4.3), we have

$$confidence\,(A \Rightarrow B) = P(B|A) = \frac{support\,(A \cup B)}{support\,(A)} = \frac{support\_count\,(A \cup B)}{support\_count\,(A)}. \tag{4.4}$$

Eq. (4.4) shows that the confidence of rule $A \Rightarrow B$ can be easily derived from the support counts of A and $A \cup B$. That is, once the support counts of A, B, and $A \cup B$ are found, it is straightforward to derive the corresponding association rules $A \Rightarrow B$ and $B \Rightarrow A$ and check whether they are strong. Thus the problem of mining association rules can be reduced to that of mining frequent itemsets.

In general, association rule mining can be viewed as a two-step process:

1. **Find all frequent itemsets.** By definition, each of these itemsets will occur at least as frequently as a predetermined minimum support count, *min_sup*.
2. **Generate strong association rules from the frequent itemsets.** By definition, these rules must satisfy minimum support and minimum confidence.

Additional interestingness measures that can be applied for the discovery of correlation relationships between associated items will be discussed in Section 4.3. The overall performance of mining association rules is determined by the first step since the second step is much less costly than the first.

A major challenge in mining frequent itemsets from a large data set is the fact that such mining often generates a huge number of itemsets satisfying the minimum support (*min_sup*) threshold, especially when *min_sup* is set low. This is because if an itemset is frequent, each of its subsets is frequent as well. A long itemset will contain a combinatorial number of shorter frequent subitemsets. For example, a frequent itemset of length 100, such as $\{a_1, a_2, \ldots, a_{100}\}$, contains $\binom{100}{1} = 100$ frequent 1-itemsets: $\{a_1\}$, $\{a_2\}$, $\ldots$, $\{a_{100}\}$; $\binom{100}{2}$ frequent 2-itemsets: $\{a_1, a_2\}$, $\{a_1, a_3\}$, $\{a_1, a_4\}$, $\ldots$, $\{a_2, a_3\}$, $\{a_2, a_4\}$, $\ldots$, $\{a_{99}, a_{100}\}$; and so on. The total number of frequent itemsets that it contains is thus

$$\binom{100}{1} + \binom{100}{2} + \cdots + \binom{100}{100} = 2^{100} - 1 \approx 1.27 \times 10^{30}. \tag{4.5}$$

This is too huge a number of itemsets for any computer to compute or store. To overcome this difficulty, we introduce the concepts of *closed frequent itemset* and *maximal frequent itemset*.

---

[3] In early work, itemsets satisfying minimum support were referred to as **large**. This term, however, is somewhat confusing as it has connotations of the number of items in an itemset rather than the frequency of occurrence of the set. Hence, we use the more recent term **frequent**.

[4] Although the term **frequent** is preferred over **large**, for historic reasons frequent k-itemsets are still denoted as L_k.

An itemset X is **closed** in a data set D if there exists no proper superitemset Y[5] such that Y has the same support count as X in D. An itemset X is a **closed frequent itemset** in set D if X is both closed and frequent in D. An itemset X is a **maximal frequent itemset** (or **max-itemset**) in a data set D if X is frequent, and there exists no superitemset Y such that $X \subset Y$ and Y is frequent in D.

Let $\mathcal{C}$ be the set of closed frequent itemsets for a data set D satisfying a minimum support threshold, min_sup. Let $\mathcal{M}$ be the set of maximal frequent itemsets for D satisfying min_sup. Suppose that we have the support count of each itemset in $\mathcal{C}$ and $\mathcal{M}$. Notice that $\mathcal{C}$ and its count information can be used to derive the whole set of frequent itemsets. Thus we say that $\mathcal{C}$ contains complete information regarding its corresponding frequent itemsets. On the other hand, $\mathcal{M}$ registers only the support of the maximal itemsets. It usually does not contain the complete support information regarding its corresponding frequent itemsets. We illustrate these concepts with Example 4.2.

Example 4.2. Closed and maximal frequent itemsets. Suppose that a transaction database has only two transactions: $\{\langle a_1, a_2, \ldots, a_{100}\rangle; \langle a_1, a_2, \ldots, a_{50}\rangle\}$. Let the minimum support count threshold be $min\_sup = 1$. We find two closed frequent itemsets and their support counts, that is, $\mathcal{C} = \{\{a_1, a_2, \ldots, a_{100}\} : 1;\ \{a_1, a_2, \ldots, a_{50}\} : 2\}$. There is only one maximal frequent itemset: $\mathcal{M} = \{\{a_1, a_2, \ldots, a_{100}\} : 1\}$. Notice that we cannot include $\{a_1, a_2, \ldots, a_{50}\}$ as a maximal frequent itemset because it has a frequent superset, $\{a_1, a_2, \ldots, a_{100}\}$. Compare this to the preceding where we determined that there are $2^{100} - 1$ frequent itemsets, which are too many to be enumerated!

The set of closed frequent itemsets contains complete information regarding the frequent itemsets. For example, from $\mathcal{C}$, we can derive, say, (1) $\{a_2, a_{45} : 2\}$ since $\{a_2, a_{45}\}$ is a subitemset of the itemset $\{a_1, a_2, \ldots, a_{50} : 2\}$; and (2) $\{a_8, a_{55} : 1\}$ since $\{a_8, a_{55}\}$ is not a subitemset of the previous itemset but of the itemset $\{a_1, a_2, \ldots, a_{100} : 1\}$. However, from the maximal frequent itemset, we can only assert that both itemsets ($\{a_2, a_{45}\}$ and $\{a_8, a_{55}\}$) are frequent, but we cannot assert their actual support counts. □

4.2 Frequent itemset mining methods

In this section, you will learn methods for mining the simplest form of frequent patterns such as those discussed for market basket analysis in Section 4.1.1. We begin by presenting **Apriori**, the basic algorithm for finding frequent itemsets in Section 4.2.1. In Section 4.2.2, we look at how to generate strong association rules from frequent itemsets. Section 4.2.3 describes several variations to the Apriori algorithm for improved efficiency and scalability. Section 4.2.4 presents pattern-growth methods for mining frequent itemsets that confine the subsequent search space to only the data sets containing the current frequent itemsets. Section 4.2.5 presents methods for mining frequent itemsets that take advantage of the vertical data format.

[5] Y is a proper superitemset of X if X is a proper subitemset of Y, that is, if $X \subset Y$. In other words, every item of X is contained in Y, but there is at least one item of Y that is not in X.

4.2.1 Apriori algorithm: finding frequent itemsets by confined candidate generation

Apriori is a seminal algorithm proposed by R. Agrawal and R. Srikant in 1994 for mining frequent itemsets for Boolean association rules [AS94b]. The name of the algorithm is based on the fact that the algorithm uses *prior knowledge* of frequent itemset properties, as we shall see later. Apriori employs an iterative approach known as a *level-wise* search, where k-itemsets are used to explore $(k+1)$-itemsets. First, the set of frequent 1-itemsets is found by scanning the database to accumulate the count for each item, and collecting those items that satisfy minimum support. The resulting set is denoted by L_1. Next, L_1 is used to find L_2, the set of frequent 2-itemsets, which is used to find L_3, and so on, until no more frequent k-itemsets can be found. The finding of each L_k requires one full scan of the database.

To improve the efficiency of the level-wise generation of frequent itemsets, an important property called the **Apriori property** is used to reduce the search space.

Apriori property: *all nonempty subsets of a frequent itemset must also be frequent.*

The Apriori property is based on the following observation. By definition, if an itemset I does not satisfy the minimum support threshold, *min_sup*, then I is not frequent, that is, $P(I) < min\_sup$. If an item A is added to the itemset I, then the resulting itemset (i.e., $I \cup A$) cannot occur more frequently than I. Therefore $I \cup A$ is not frequent either, that is, $P(I \cup A) < min\_sup$.

This property belongs to a special category of properties called **antimonotonicity** in the sense that *if a set cannot pass a test, all of its supersets will fail the same test as well*. It is called *antimonotonicity* because the property is monotonic in the context of failing a test.

"How is the Apriori property used in the algorithm?" To understand this, let us look at how L_{k-1} is used to find L_k for $k \geq 2$. A two-step process is followed, consisting of **join** and **prune** actions.

1. **The join step**. To find L_k, a set of **candidate** k-itemsets is generated by joining L_{k-1} with itself. This set of candidates is denoted C_k. Let l_1 and l_2 be itemsets in L_{k-1}. The notation $l_i[j]$ refers to the jth item in l_i (e.g., $l_1[k-2]$ refers to the second to the last item in l_1). For efficient implementation, Apriori assumes that items within a transaction or itemset are sorted in lexicographic order. For the $(k-1)$-itemset, l_i, this means that the items are sorted such that $l_i[1] < l_i[2] < \cdots < l_i[k-1]$. The join, $L_{k-1} \bowtie L_{k-1}$, is performed, where members of L_{k-1} are joinable if their first $(k-2)$ items are in common. That is, members l_1 and l_2 of L_{k-1} are joined if $(l_1[1] = l_2[1]) \wedge (l_1[2] = l_2[2]) \wedge \cdots \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] < l_2[k-1])$. The condition $l_1[k-1] < l_2[k-1]$ simply ensures that no duplicates are generated. The resulting itemset formed by joining l_1 and l_2 is $\{l_1[1], l_1[2], \ldots, l_1[k-2], l_1[k-1], l_2[k-1]\}$.
2. **The prune step**. C_k is a superset of L_k, that is, its members may or may not be frequent, but all of the frequent k-itemsets are included in C_k. A database scan to determine the count of each candidate in C_k would result in the determination of L_k (i.e., all candidates having a count no less than the minimum support count are frequent by definition and therefore belong to L_k). C_k, however, can be huge, and so this could involve heavy computation. To reduce the size of C_k, the Apriori property is used as follows. Any $(k-1)$-itemset that is not frequent cannot be a subset of a frequent k-itemset. Hence, if any $(k-1)$-subset of a candidate k-itemset is not in L_{k-1}, then the candidate cannot be frequent either and so can be removed from C_k. This **subset testing** can be done quickly by maintaining a hash tree of all frequent itemsets.

Example 4.3. Apriori. Let's look at a concrete example, based on the transaction database, D, of Table 4.1. There are nine transactions in this database, that is, $|D| = 9$. We use Fig. 4.2 to illustrate the Apriori algorithm for finding frequent itemsets in D.

1. In the first iteration of the algorithm, each item is a member of the set of candidate 1-itemsets, C_1. The algorithm simply scans all of the transactions to count the number of occurrences of each item.
2. Suppose that the minimum support count required is 2, that is, $min\_sup = 2$. (Here, we are referring to *absolute* support because we are using a support count. The corresponding relative support is $2/9 = 22\%$.) The set of frequent 1-itemsets, L_1, can then be determined. It consists of the candidate 1-itemsets satisfying minimum support. In our example, all of the candidates in C_1 satisfy minimum support.
3. To discover the set of frequent 2-itemsets, L_2, the algorithm uses the join $L_1 \bowtie L_1$ to generate a candidate set of 2-itemsets, C_2.[6] C_2 consists of $\binom{|L_1|}{2}$ 2-itemsets. Note that no candidates are removed from C_2 during the prune step because each subset of the candidates is also frequent.
4. Next, the transactions in D are scanned and the support count of each candidate itemset in C_2 is accumulated, as shown in the middle table of the second row in Fig. 4.2.
5. The set of frequent 2-itemsets, L_2, is then determined, consisting of those candidate 2-itemsets in C_2 having minimum support.
6. The generation of the set of the candidate 3-itemsets, C_3, is detailed in Fig. 4.3. From the join step, we first get $C_3 = L_2 \bowtie L_2 = \{\{I1, I2, I3\}, \{I1, I2, I5\}, \{I1, I3, I5\}, \{I2, I3, I4\}, \{I2, I3, I5\}, \{I2, I4, I5\}\}$. Based on the Apriori property that all subsets of a frequent itemset must also be frequent, we can determine that the four latter candidates cannot possibly be frequent. We therefore remove them from C_3, thereby saving the effort of unnecessarily obtaining their counts during the subsequent scan of D to determine L_3. Note that when given a candidate k-itemset, we only need to check if its $(k-1)$-subsets are frequent since the Apriori algorithm uses a level-wise search strategy. The resulting pruned version of C_3 is shown in the first table of the bottom row of Fig. 4.2.
7. The transactions in D are scanned to determine L_3, consisting of those candidate 3-itemsets in C_3 having minimum support (Fig. 4.2).

Table 4.1 A transactional data set.

TID	List of item_IDs
T100	I1, I2, I5
T200	I2, I4
T300	I2, I3
T400	I1, I2, I4
T500	I1, I3
T600	I2, I3
T700	I1, I3
T800	I1, I2, I3, I5
T900	I1, I2, I3

[6] $L_1 \bowtie L_1$ is equivalent to $L_1 \times L_1$, since the definition of $L_k \bowtie L_k$ requires the two joining itemsets to share $k-1 = 0$ items.

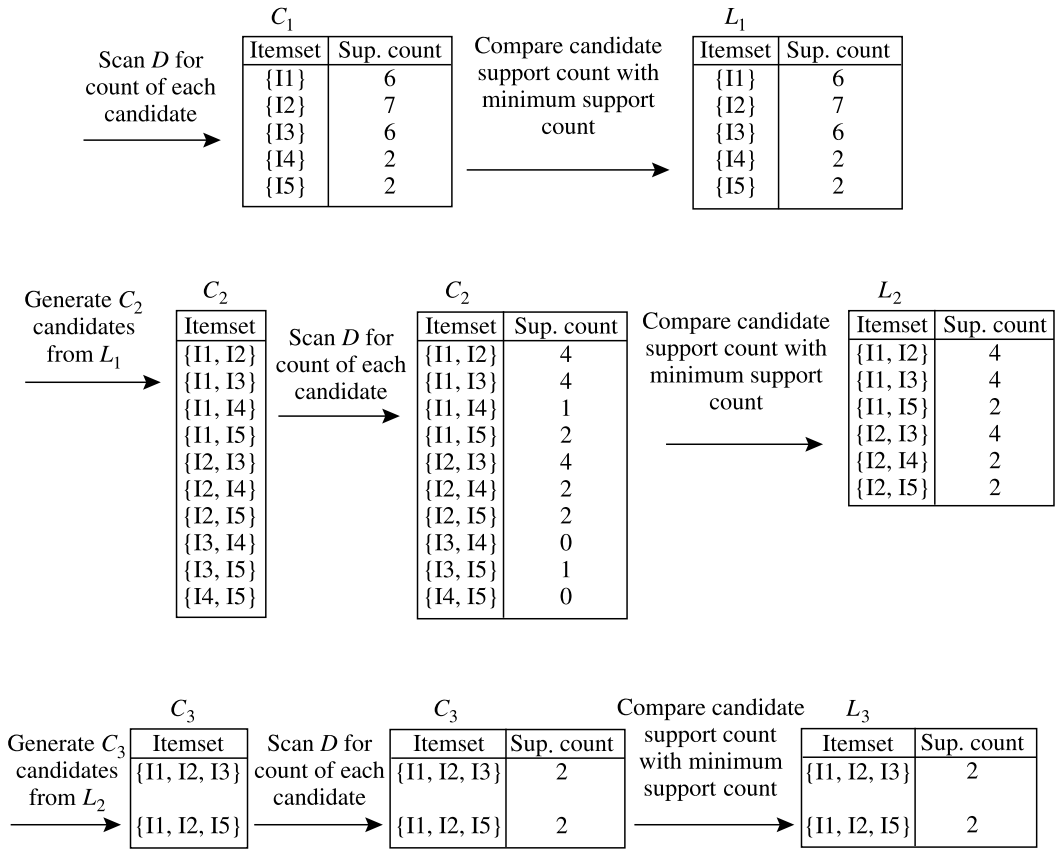


FIGURE 4.2

Generation of the candidate itemsets and frequent itemsets, where the minimum support count is 2.

8. The algorithm uses $L_3 \bowtie L_3$ to generate a candidate set of 4-itemsets, C_4. Although the join results in $\{\{I1, I2, I3, I5\}\}$, itemset $\{I1, I2, I3, I5\}$ is pruned because its subset $\{I2, I3, I5\}$ is not frequent. Thus, $C_4 = \phi$, and the algorithm terminates, having found all of the frequent itemsets. □

Fig. 4.4 shows pseudocode for the Apriori algorithm and its related procedures. Step 1 of Apriori finds the frequent 1-itemsets, L_1. In steps 2 through 10, L_{k-1} is used to generate candidates C_k to find L_k for $k \geq 2$. The `apriori_gen` procedure generates the candidates and then uses the Apriori property to eliminate those having a subset that is not frequent (step 3). Once all of the candidates have been generated, the database is scanned (step 4). For each transaction, a subset function is used to find all subsets of the transaction that are candidates (step 5), and the count for each of these candidates is accumulated (steps 6 and 7). Finally, all the candidates satisfying the minimum support (step 9) form the set of frequent itemsets, L (step 11). A procedure can then be called to generate association rules from the frequent itemsets. Such a procedure is described in Section 4.2.2.

a. Join: $C_3 = L_2 \bowtie L_2 = \{\{I1, I2\}, \{I1, I3\}, \{I1, I5\}, \{I2, I3\}, \{I2, I4\}, \{I2, I5\}\}$
$\bowtie \{\{I1, I2\}, \{I1, I3\}, \{I1, I5\}, \{I2, I3\}, \{I2, I4\}, \{I2, I5\}\}$
$= \{\{I1, I2, I3\}, \{I1, I2, I5\}, \{I1, I3, I5\}, \{I2, I3, I4\}, \{I2, I3, I5\}, \{I2, I4, I5\}\}$.

b. Prune using the Apriori property: *all nonempty subsets of a frequent itemset must also be frequent*. Do any of the candidates have a subset that is not frequent?

- The 2-item subsets of {I1, I2, I3} are {I1, I2}, {I1, I3}, and {I2, I3}. All 2-item subsets of {I1, I2, I3} are members of L_2. Therefore, keep {I1, I2, I3} in C_3.
- The 2-item subsets of {I1, I2, I5} are {I1, I2}, {I1, I5}, and {I2, I5}. All 2-item subsets of {I1, I2, I5} are members of L_2. Therefore, keep {I1, I2, I5} in C_3.
- The 2-item subsets of {I1, I3, I5} are {I1, I3}, {I1, I5}, and {I3, I5}. {I3, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I1, I3, I5} from C_3.
- The 2-item subsets of {I2, I3, I4} are {I2, I3}, {I2, I4}, and {I3, I4}. {I3, I4} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I3, I4} from C_3.
- The 2-item subsets of {I2, I3, I5} are {I2, I3}, {I2, I5}, and {I3, I5}. {I3, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I3, I5} from C_3.
- The 2-item subsets of {I2, I4, I5} are {I2, I4}, {I2, I5}, and {I4, I5}. {I4, I5} is not a member of L_2, and so it is not frequent. Therefore, remove {I2, I4, I5} from C_3.

c. Therefore, $C_3 = \{\{I1, I2, I3\}, \{I1, I2, I5\}\}$ after pruning.

FIGURE 4.3

Generation and pruning of candidate 3-itemsets, C_3, from L_2 using the Apriori property.

The `apriori_gen` procedure performs two kinds of actions, namely, **join** and **prune**, as described before. In the join component, L_{k-1} is joined with L_{k-1} to generate potential candidates (steps 1–4). The prune component (steps 5–7) employs the Apriori property to remove candidates that have a subset that is not frequent. The test for infrequent subsets is shown in procedure `has_infrequent_subset`.

4.2.2 Generating association rules from frequent itemsets

Once the frequent itemsets from transactions in a database D have been found, it is straightforward to generate strong association rules from them (where *strong* association rules satisfy both minimum support and minimum confidence). This can be done using Eq. (4.4) for confidence, which we show again here for completeness:

$$confidence\,(A \Rightarrow B) = P(B|A) = \frac{support\_count\,(A \cup B)}{support\_count\,(A)}.$$

The conditional probability is expressed in terms of itemset support count, where $support\_count\,(A \cup B)$ is the number of transactions containing the itemsets $A \cup B$, and $support\_count\,(A)$ is the number of transactions containing the itemset A. Based on this equation, association rules can be generated as follows.

- For each frequent itemset l, generate all nonempty subsets of l.
- For every nonempty subset s of l, output the rule "$s \Rightarrow (l - s)$" if $\frac{support\_count\,(l)}{support\_count\,(s)} \geq min\_conf$, where min_conf is the minimum confidence threshold.

Algorithm: Apriori. Find frequent itemsets using an iterative level-wise approach based on candidate generation.

Input:

- D, a database of transactions;
- *min_sup*, the minimum support count threshold.

Output: L, frequent itemsets in D.

Method:

```
(1)   L1 = find_frequent_1-itemsets(D);
(2)   for (k = 2; Lk−1 ≠ φ; k++) {
(3)       Ck = apriori_gen(Lk−1);
(4)       for each transaction t ∈ D { // scan D for counts
(5)           Ct = subset(Ck, t); // get the subsets of t that are candidates
(6)           for each candidate c ∈ Ct
(7)               c.count++;
(8)       }
(9)       Lk = {c ∈ Ck|c.count ≥ min_sup}
(10)  }
(11)  return L = ∪k Lk;

procedure apriori_gen(Lk−1:frequent (k − 1)-itemsets)
(1)   for each itemset l1 ∈ Lk−1
(2)       for each itemset l2 ∈ Lk−1
(3)           if (l1[1] = l2[1]) ∧ (l1[2] = l2[2])
                  ∧... ∧ (l1[k − 2] = l2[k − 2]) ∧ (l1[k − 1] < l2[k − 1]) then {
(4)               c = l1 ⋈ l2; // join step: generate candidates
(5)               if has_infrequent_subset(c, Lk−1) then
(6)                   delete c; // prune step: remove unfruitful candidate
(7)               else add c to Ck;
(8)           }
(9)   return Ck;

procedure has_infrequent_subset(c: candidate k-itemset;
          Lk−1: frequent (k − 1)-itemsets); // use prior knowledge
(1)   for each (k − 1)-subset s of c
(2)       if s ∉ Lk−1 then
(3)           return TRUE;
(4)   return FALSE;
```

FIGURE 4.4

Apriori algorithm for discovering frequent itemsets for mining Boolean association rules.

Because the rules are generated from frequent itemsets, each one automatically satisfies the minimum support. Frequent itemsets can be stored ahead of time in hash tables along with their counts so that they can be accessed quickly.

Example 4.4. Generating association rules. Let's try an example based on the transactional data shown before in Table 4.1. The data contain frequent itemset $X = \{\text{I1}, \text{I2}, \text{I5}\}$. What are the association rules that can be generated from X? The nonempty subsets of X are $\{\text{I1}, \text{I2}\}$, $\{\text{I1}, \text{I5}\}$, $\{\text{I2}, \text{I5}\}$, $\{\text{I1}\}$, $\{\text{I2}\}$, and $\{\text{I5}\}$. The resulting association rules are as shown below, each listed with its confidence:

$\{I1, I2\} \Rightarrow I5$, $confidence = 2/4 = 50\%$
$\{I1, I5\} \Rightarrow I2$, $confidence = 2/2 = 100\%$
$\{I2, I5\} \Rightarrow I1$, $confidence = 2/2 = 100\%$
$I1 \Rightarrow \{I2, I5\}$, $confidence = 2/6 = 33\%$
$I2 \Rightarrow \{I1, I5\}$, $confidence = 2/7 = 29\%$
$I5 \Rightarrow \{I1, I2\}$, $confidence = 2/2 = 100\%$

If the minimum confidence threshold is, say, 70%, then only the second, third, and last rules are output, because these are the only ones generated that are strong. Note that, unlike conventional classification rules, association rules can contain more than one conjunct in the right side of the rule. □

4.2.3 Improving the efficiency of Apriori

"How can we further improve the efficiency of Apriori-based mining?" Many variations of the Apriori algorithm have been proposed that focus on improving the efficiency of the original algorithm. Several of these variations are summarized as follows.

Hash-based technique (hashing itemsets into corresponding buckets). A hash-based technique can be used to reduce the size of the candidate k-itemsets, C_k, for $k > 1$. For example, when scanning each transaction (e.g., let $t = \{i_1, i_2, i_4\}$) in the database to generate the frequent 1-itemsets, L_1, we can generate all the 2-itemsets for each transaction (e.g., three 2-itemsets $\{i_1, i_2\}$, $\{i_1, i_4\}$, and $\{i_2, i_4\}$ for transaction t), hash (i.e., map) them into the different *buckets* of a *hash table* structure, and increase the corresponding bucket counts as shown in Fig. 4.5. A 2-itemset with a corresponding bucket count in the hash table that is below the support threshold cannot be frequent and thus should be removed from the candidate set. Such a hash-based technique may substantially reduce the number of candidate k-itemsets examined (especially when $k = 2$).

Transaction reduction (reducing the number of transactions scanned in future iterations). A transaction that does not contain any frequent k-itemsets cannot contain any frequent $(k + 1)$-itemsets. Therefore such a transaction can be marked or removed from further consideration because subsequent database scans for j-itemsets, where $j > k$, will not need to consider such a transaction.

Create hash table H_2 using hash function $h(x, y) = ((order\ of\ x) \times 10 + (order\ of\ y))\ mod\ 7$

H_2

bucket address	0	1	2	3	4	5	6
bucket count	2	2	4	2	2	4	4
bucket contents	{I1, I4} {I3, I5}	{I1, I5} {I1, I5}	{I2, I3} {I2, I3} {I2, I3} {I2, I3}	{I2, I4} {I2, I4}	{I2, I5} {I2, I5}	{I1, I2} {I1, I2} {I1, I2} {I1, I2}	{I1, I3} {I1, I3} {I1, I3} {I1, I3}

FIGURE 4.5

Hash table, H_2, for candidate 2-itemsets. This hash table was generated by scanning Table 4.1's transactions while determining L_1. If the minimum support count is, say, 3, then the itemsets in buckets 0, 1, 3, and 4 cannot be frequent and so they should not be included in C_2.

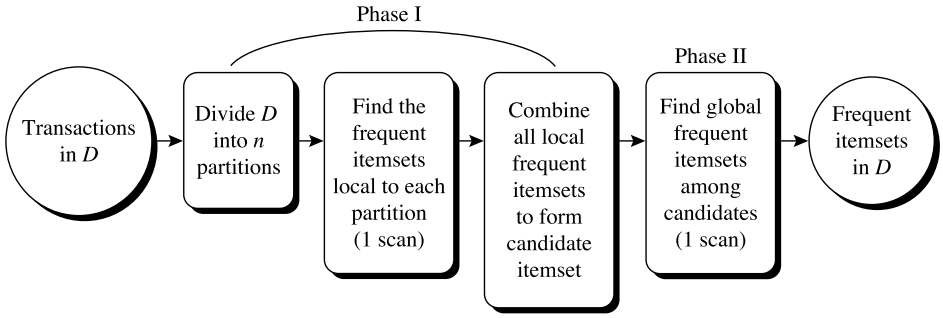


FIGURE 4.6

Mining by partitioning the data.

Partitioning (partitioning the data to find candidate itemsets). A partitioning technique can be used that requires just two database scans to mine the frequent itemsets (Fig. 4.6). It consists of two phases. In phase I, the algorithm divides the transactions of D into n nonoverlapping partitions. If the minimum relative support threshold for transactions in D is min_sup, then the minimum support count for a partition is $min\_sup \times$ *the number of transactions in that partition*. For each partition, all the *local frequent itemsets* (i.e., the itemsets frequent within the partition) are found.

A local frequent itemset may or may not be frequent with respect to the entire database, D. However, *any itemset that is potentially frequent with respect to D must occur as a frequent itemset in at least one of the partitions*.[7] Therefore all local frequent itemsets are candidate itemsets with respect to D. The collection of frequent itemsets from all partitions forms the *global candidate itemsets* with respect to D. In phase II, a second scan of D is conducted in which the actual support of each candidate is assessed to determine the global frequent itemsets. Partition size and the number of partitions are set so that each partition can fit into main memory and therefore be read only once in each phase.

Sampling (mining on a subset of the given data). The basic idea of the sampling approach is to pick a random sample S of the given data D, and then search for frequent itemsets in S instead of D. In this way, we trade off some degree of accuracy against efficiency. The S sample size is such that the search for frequent itemsets in S can be done in main memory, and so only one scan of the transactions in S is required overall. Because we are searching for frequent itemsets in S rather than in D, it is possible that we will miss some of the global frequent itemsets.

To reduce this possibility, we use a lower support threshold than the minimum support to find the frequent itemsets local to S (denoted L_S). The rest of the database is then used to compute the actual frequencies of each itemset in L_S. A mechanism is used to determine whether all the global frequent itemsets are included in L_S. If L_S actually contains all the frequent itemsets in D, then only one scan of D is required. Otherwise, a second pass can be done to find the frequent itemsets that were missed in the first pass. The sampling approach is especially beneficial when efficiency is of utmost importance such as in computationally intensive applications that must be run frequently.

[7] The proof of this property is left as an exercise (see Exercise 4.3d).

Dynamic itemset counting (adding candidate itemsets at different points during a scan). A dynamic itemset counting technique is proposed in which the database is partitioned into blocks marked by start points. In this variation, new candidate itemsets can be added at any start point, unlike in Apriori, which determines new candidate itemsets only after each complete database scan. The technique uses the count-so-far as the lower bound of the actual count. If the count-so-far passes the minimum support, the itemset is added into the frequent itemset collection and can be used to generate longer candidates. This leads to fewer database scans than with Apriori for finding all the frequent itemsets.

Other variations are discussed in the next chapter or left as exercises.

4.2.4 A pattern-growth approach for mining frequent itemsets

As we have seen, in many cases the Apriori candidate generate-and-test method significantly reduces the size of candidate sets, leading to good performance gain. However, it can suffer from two nontrivial costs.

- *It may still need to generate a huge number of candidate sets.* For example, if there are 10^4 frequent 1-itemsets, the Apriori algorithm will need to generate more than 10^7 candidate 2-itemsets.
- *It may need to repeatedly scan the whole database and check a large set of candidates by pattern matching.* It is costly to go over each transaction in the database to determine the support of the candidate itemsets.

"Can we design a method that mines the complete set of frequent itemsets without such a costly candidate generation process?" An interesting method in this attempt is called **frequent pattern growth,** or simply **FP-growth**, which adopts a *divide-and-conquer* strategy as follows. First, it compresses the database representing frequent items into a **frequent pattern tree,** or **FP-tree**, which retains the itemset association information. It then divides the compressed database into a set of *conditional databases* (a special kind of projected database), each associated with one itemset found so far, or "pattern fragment," and mines each database separately. For each "pattern fragment," only its associated data sets need to be examined. Therefore this approach may substantially reduce the size of the data sets to be searched, along with the "growth" of patterns being examined. You will see how it works in Example 4.5.

Example 4.5. FP-growth (finding frequent itemsets without candidate generation). We reexamine the mining of transaction database, D, of Table 4.1 in Example 4.3 using the frequent pattern growth approach.

The first scan of the database is the same as Apriori, which derives the set of frequent items (1-itemsets) and their support counts (frequencies). Let the minimum support count be 2. The set of frequent items is sorted in the order of descending support count. This resulting set or *list* is denoted by L. Thus, we have $L = \{\{\text{I2: 7}\}, \{\text{I1: 6}\}, \{\text{I3: 6}\}, \{\text{I4: 2}\}, \{\text{I5: 2}\}\}$.

An FP-tree is then constructed as follows. First, create the root of the tree, labeled with "null." Scan database D a second time. The items in each transaction are processed in L order (i.e., sorted according to descending support count), and a branch is created for each transaction. For example, the scan of the first transaction, "T100: I1, I2, I5," which contains three items (I2, I1, I5 in L order), leads to the construction of the first branch of the tree with three nodes, $\langle$I2: 1$\rangle$, $\langle$I1: 1$\rangle$, and $\langle$I5: 1$\rangle$, where I2 is linked as a child to the root, I1 is linked to I2, and I5 is linked to I1. The second transaction, T200,

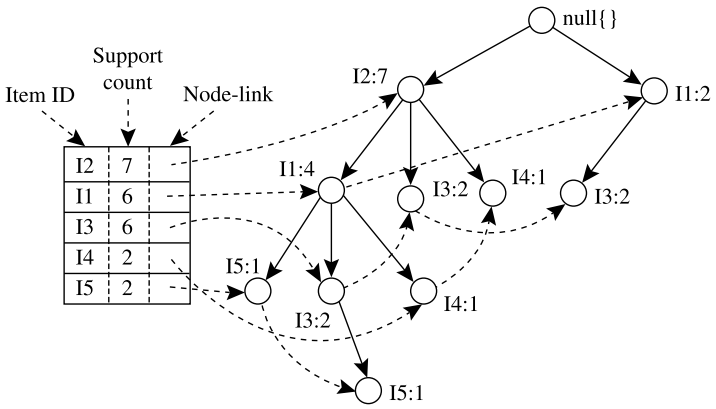


FIGURE 4.7

An FP-tree registers compressed frequent pattern information.

contains the items I2 and I4 in L order, which would result in a branch where I2 is linked to the root and I4 is linked to I2. However, this branch would share a common **prefix,** I2, with the existing path for T100. Therefore, we instead increment the count of the I2 node by 1, and create a new node, ⟨I4: 1⟩, which is linked as a child to ⟨I2: 2⟩. In general, when considering the branch to be added for a transaction, the count of each node along a common prefix is incremented by 1, and nodes for the items following the prefix are created and linked accordingly.

To facilitate tree traversal, an item header table is built so that each item points to its occurrences in the tree via a chain of **node-links**. The tree obtained after scanning all the transactions is shown in Fig. 4.7 with the associated node-links. In this way, the problem of mining frequent patterns in databases is transformed into that of mining the FP-tree.

The FP-tree is mined as follows. Start from each frequent length-1 pattern (as an initial **suffix pattern**), construct its **conditional pattern base** (a "subdatabase," which consists of the set of *prefix paths* in the FP-tree cooccurring with the suffix pattern), then construct its (*conditional*) FP-tree, and perform mining recursively on the tree. The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree.

Mining of the FP-tree is summarized in Table 4.2 and detailed as follows.

- We first consider I5, which is the last item in L, rather than the first. The reason for starting at the end of the list will become apparent as we explain the FP-tree mining process. I5 occurs in two FP-tree branches of Fig. 4.7. (The occurrences of I5 can easily be found by following its chain of node-links.) The paths formed by these branches are ⟨I2, I1, I5: 1⟩ and ⟨I2, I1, I3, I5: 1⟩. Therefore, considering I5 as a suffix, its corresponding two prefix paths are ⟨I2, I1: 1⟩ and ⟨I2, I1, I3: 1⟩, which form its conditional pattern base. Using this conditional pattern base as a transaction database, we build an I5-conditional FP-tree, which contains only a single path, ⟨I2: 2, I1: 2⟩; I3 is not included because its support count of 1 is less than the minimum support count. The single path generates all the combinations of frequent patterns: {I2, I5: 2}, {I1, I5: 2}, {I2, I1, I5: 2}.

Table 4.2 Mining the FP-tree by creating conditional (sub-)pattern bases.

Item	Conditional Pattern Base	Conditional FP-tree	Frequent Patterns Generated
I5	{{I2, I1: 1}, {I2, I1, I3: 1}}	⟨I2: 2, I1: 2⟩	{I2, I5: 2}, {I1, I5: 2}, {I2, I1, I5: 2}
I4	{{I2, I1: 1}, {I2: 1}}	⟨I2: 2⟩	{I2, I4: 2}
I3	{{I2, I1: 2}, {I2: 2}, {I1: 2}}	⟨I2: 4, I1: 2⟩, ⟨I1: 2⟩	{I2, I3: 4}, {I1, I3: 4}, {I2, I1, I3: 2}
I1	{{I2: 4}}	⟨I2: 4⟩	{I2, I1: 4}

Algorithm: **FP_growth.** Mine frequent itemsets using an FP-tree by pattern fragment growth.

Input:

- D, a transaction database;
- *min_sup*, the minimum support count threshold.

Output: The complete set of frequent patterns.
Method:

1. The FP-tree is constructed in the following steps:
   a. Scan the transaction database D once. Collect F, the set of frequent items, and their support counts. Sort F in support count descending order as L, the *list* of frequent items.
   b. Create the root of an FP-tree, and label it as "null." For each transaction *Trans* in D do the following. Select and sort the frequent items in *Trans* according to the order of L. Let the sorted frequent item list in *Trans* be $[p|P]$, where p is the first element and P is the remaining list. Call insert_tree($[p|P]$, T), which is performed as follows. If T has a child N such that $N.item\text{-}name = p.item\text{-}name$, then increment N's count by 1; else create a new node N, and let its count be 1, its parent link be linked to T, and its node-link to the nodes with the same *item-name* via the node-link structure. If P is nonempty, call insert_tree(P, N) recursively.
2. The FP-tree is mined by calling FP_growth(*FP_tree*, *null*), which is implemented as follows.

```
procedure FP_growth(Tree, α)
(1)  if Tree contains a single path P then
(2)    for each combination (denoted as β) of the nodes in the path P
(3)      generate pattern β ∪ α with support_count = minimum support count of nodes in β;
(4)  else for each a_i in the header of Tree {
(5)    generate pattern β = a_i ∪ α with support_count = a_i.support_count;
(6)    construct β's conditional pattern base and then β's conditional FP_tree Tree_β;
(7)    if Tree_β ≠ ∅ then
(8)      call FP_growth(Tree_β, β); }
```

FIGURE 4.8

FP-growth algorithm for discovering frequent itemsets without candidate generation.

- For I4, its two prefix paths form the conditional pattern base, {{I2 I1: 1}, {I2: 1}}, which generates a single-node conditional FP-tree, ⟨I2: 2⟩, and derives one frequent pattern, {I2, I4: 2}.
- Similar to the preceding analysis, I3's conditional pattern base is {{I2, I1: 2}, {I2: 2}, {I1: 2}}. Its conditional FP-tree has two branches, ⟨I2: 4, I1: 2⟩ and ⟨I1: 2⟩, as shown in Fig. 4.9, which generates the set of patterns {{I2, I3: 4}, {I1, I3: 4}, {I2, I1, I3: 2}}.
- Finally, I1's conditional pattern base is {{I2: 4}}, with an FP-tree that contains only one node, ⟨I2: 4⟩, which generates one frequent pattern, {I2, I1: 4}.

This mining process is summarized in Fig. 4.8. □

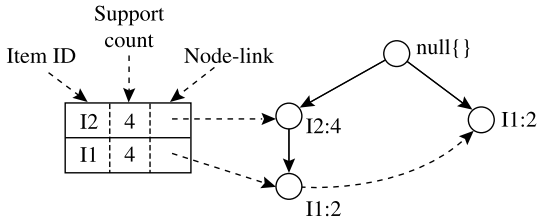


FIGURE 4.9

The conditional FP-tree associated with the conditional node I3.

The FP-growth method transforms the problem of finding long frequent patterns into searching for shorter ones in much smaller conditional databases recursively and then concatenating the suffix. It uses the least frequent items as a suffix, offering good selectivity. The method substantially reduces the search costs.

When the database is large, it is sometimes unrealistic to construct a main memory-based FP-tree. An interesting alternative is to first partition the database into a set of projected databases and then construct an FP-tree and mine it in each projected database. This process can be recursively applied to any projected database if its FP-tree still cannot fit in main memory.

4.2.5 Mining frequent itemsets using the vertical data format

Both the Apriori and FP-growth methods mine frequent patterns from a set of transactions in *TID-itemset* format (i.e., $\{TID : itemset\}$), where *TID* is a transaction ID and *itemset* is the set of items bought in transaction *TID*. This is known as the **horizontal data format**. Alternatively, data can be presented in *item-TID_set* format (i.e., $\{item : TID\_set\}$), where *item* is an item name, and *TID_set* is the set of transaction identifiers containing the item. This is known as the **vertical data format**.

In this subsection, we look at how frequent itemsets can also be mined efficiently using vertical data format, which is the essence of the **Eclat** (Equivalence Class Transformation) algorithm.

Example 4.6. Mining frequent itemsets using the vertical data format. Consider the horizontal data format of the transaction database, *D*, of Table 4.1 in Example 4.3. This can be transformed into the vertical data format shown in Table 4.3 by scanning the data set once.

Table 4.3 The vertical data format of the transaction data set *D* of Table 4.1.

itemset	TID_set
I1	{T100, T400, T500, T700, T800, T900}
I2	{T100, T200, T300, T400, T600, T800, T900}
I3	{T300, T500, T600, T700, T800, T900}
I4	{T200, T400}
I5	{T100, T800}

Table 4.4 2-Itemsets in vertical data format.

itemset	TID_set
{I1, I2}	{T100, T400, T800, T900}
{I1, I3}	{T500, T700, T800, T900}
{I1, I4}	{T400}
{I1, I5}	{T100, T800}
{I2, I3}	{T300, T600, T800, T900}
{I2, I4}	{T200, T400}
{I2, I5}	{T100, T800}
{I3, I5}	{T800}

Table 4.5 3-Itemsets in vertical data format.

itemset	TID_set
{I1, I2, I3}	{T800, T900}
{I1, I2, I5}	{T100, T800}

Mining can be performed on this data set by intersecting the TID_sets of every pair of frequent single items. The minimum support count is 2. Because every single item is frequent in Table 4.3, there are 10 intersections performed in total, which lead to eight nonempty 2-itemsets, as shown in Table 4.4. Notice that because the itemsets {I1, I4} and {I3, I5} each contain only one transaction, they do not belong to the set of frequent 2-itemsets.

Based on the Apriori property, a given 3-itemset is a candidate 3-itemset only if every one of its 2-itemset subsets is frequent. The candidate generation process here will generate only two 3-itemsets: {I1, I2, I3} and {I1, I2, I5}. By intersecting the TID_sets of any two corresponding 2-itemsets of these candidate 3-itemsets, it derives Table 4.5, where there are only two frequent 3-itemsets: {I1, I2, I3: 2} and {I1, I2, I5: 2}. □

Example 4.6 illustrates the process of mining frequent itemsets by exploring the vertical data format. First, we transform the horizontally formatted data into the vertical format by scanning the data set once. The support count of an itemset is simply the length of the TID_set of the itemset. Starting with $k = 1$, the frequent k-itemsets can be used to construct the candidate $(k + 1)$-itemsets based on the Apriori property. The computation is done by intersection of the TID_sets of the frequent k-itemsets to compute the TID_sets of the corresponding $(k + 1)$-itemsets. This process repeats, with k incremented by 1 each time, until no frequent itemsets or candidate itemsets can be found.

Besides taking advantage of the Apriori property in the generation of candidate $(k + 1)$-itemset from frequent k-itemsets, another merit of this method is that there is no need to scan the database to find the support of $(k + 1)$-itemsets (for $k \geq 1$). This is because the TID_set of each k-itemset carries the complete information required for counting such support. However, the TID_sets can be quite long, taking substantial memory space as well as computation time for intersecting the long sets.

To further reduce the cost of registering long TID_sets, as well as the subsequent costs of intersections, we can use a technique called *diffset*, which keeps track of only the differences of the TID_sets of

a $(k+1)$-itemset and a corresponding k-itemset. For instance, in Example 4.6 we have $\{I1\} = \{T100, T400, T500, T700, T800, T900\}$ and $\{I1, I2\} = \{T100, T400, T800, T900\}$. The *diffset* between the two is $diffset(\{I1, I2\}, \{I1\}) = \{T500, T700\}$. Thus rather than recording the four TIDs that make up the intersection of $\{I1\}$ and $\{I2\}$, we can instead use *diffset* to record just two TIDs, indicating the difference between $\{I1\}$ and $\{I1, I2\}$. With such compressed bookkeeping, itemset frequency can still be calculated correctly. Experiments show that in certain situations, such as when the data set contains many dense and long patterns, this technique can substantially reduce the total cost of vertical format mining of frequent itemsets.

4.2.6 Mining closed and max patterns

In Section 4.1.2 we saw how frequent itemset mining may generate a huge number of frequent itemsets, especially when the *min_sup* threshold is set low or when there exist long patterns in the data set. Example 4.2 showed that closed frequent itemsets[8] can substantially reduce the number of patterns generated in frequent itemset mining while preserving the complete information regarding the set of frequent itemsets. That is, from the set of closed frequent itemsets, we can easily derive the set of frequent itemsets and their support. Thus in practice, it is more desirable to mine the set of closed frequent itemsets rather than the set of all frequent itemsets in most cases.

"How can we mine closed frequent itemsets?" A naïve approach would be to first mine the complete set of frequent itemsets and then remove every frequent itemset that is a proper subset of, and carries the same support as, an existing frequent itemset. However, this is quite costly. As shown in Example 4.2, this method would have to first derive $2^{100} - 1$ frequent itemsets to obtain a length-100 frequent itemset, all before it could begin to eliminate redundant itemsets. This is prohibitively expensive. In fact, there exist only a very small number of closed frequent itemsets in Example 4.2's data set.

A recommended methodology is to prune the search space as soon as we can identify the case of closed itemsets during mining. For example, an itemset merging method is introduced as follows.

Itemset merging. *If every transaction containing a frequent itemset X also contains an itemset Y but not any proper superset of Y, then $X \cup Y$ forms a frequent closed itemset and there is no need to search for any itemset containing X but no Y.*

For example, in Table 4.2 of Example 4.5, the projected conditional database for prefix itemset $\{I5:2\}$ is $\{\{I2, I1\}, \{I2, I1, I3\}\}$, from which we can see that each of its transactions contains itemset $\{I2, I1\}$ but no proper superset of $\{I2, I1\}$. Itemset $\{I2, I1\}$ can be merged with $\{I5\}$ to form the closed itemset $\{I5, I2, I1: 2\}$, and we do not need to mine for closed itemsets that contain I5 but not $\{I2, I1\}$.

Many search space pruning and closure checking methods have been developed for mining frequent closed itemsets. Moreover, because maximal frequent itemsets share many similarities with closed frequent itemsets, many of the optimization techniques developed for mining closed itemset can be extended to mining maximal frequent itemsets. Interested readers may like to dig deeper by studying related research papers.

---

[8] Remember that X is a *closed frequent* itemset in a data set S if there exists no proper superitemset Y such that Y has the same support count as X in S, and X satisfies minimum support.

4.3 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 many users. 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 4.3.1). We then discuss how the support–confidence framework can be supplemented with additional interestingness measures based on *correlation analysis* (Section 4.3.2). Section 4.3.3 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.

4.3.1 Strong rules are not necessarily interesting

The interestingness of a rule 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 4.7. A misleading "strong" association rule. Suppose we are interested in analyzing transactions 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, whereas 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:

$$buys(X, \text{"}computer\ games\text{"}) \Rightarrow buys(X, \text{"}videos\text{"})$$
$$[support = 40\%, confidence = 66\%]. \tag{4.6}$$

Rule (4.6) 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 (4.6) 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 (4.6). □

Example 4.7 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.

4.3.2 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 augmented to the support–confidence framework for association rules. This leads to *correlation rules* of the form

$$A \Rightarrow B \ [support,\ confidence,\ correlation]. \tag{4.7}$$

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 for us 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**. 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

$$lift(A, B) = \frac{P(A \cup B)}{P(A)P(B)}. \tag{4.8}$$

If the resulting value of Eq. (4.8) 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.

Eq. (4.8) is equivalent to $P(B|A)/P(B)$, or $conf(A \Rightarrow B)/P(B)$, 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. (4.8).

Let's go back to the computer game and video data of Example 4.7.

Example 4.8. Correlation analysis using lift. To help filter out misleading "strong" associations of the form $A \Rightarrow B$ from the data of Example 4.7, we need to study how the two itemsets, A and B, are correlated. Let $\overline{game}$ refer to the transactions of Example 4.7 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 4.6.

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. (4.8), the lift of Rule (4.6) 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, whereas 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. □

Table 4.6 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 4.7 Table 4.6 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

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 4.8.

Example 4.9. 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 4.7. 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 4.8. □

4.3.3 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. In the data mining community, several other pattern evaluation measures have attracted interest. In this subsection, we present four such measures: *all_confidence*, *max_confidence*, *Kulczynski*, and *cosine*. Each of these four measures has an interesting property: the value of each measure 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.

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

$$all\_conf(A, B) = \frac{sup(A \cup B)}{max\{sup(A), sup(B)\}} = min\{P(A|B), P(B|A)\}, \tag{4.9}$$

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

$$max\_conf(A, B) = max\{P(A \mid B), P(B \mid A)\}. \tag{4.10}$$

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

$$Kulc(A, B) = \frac{1}{2}(P(A|B) + P(B|A)). \tag{4.11}$$

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

$$\begin{aligned} cosine(A, B) &= \frac{P(A \cup B)}{\sqrt{P(A) \times P(B)}} = \frac{sup(A \cup B)}{\sqrt{sup(A) \times sup(B)}} \\ &= \sqrt{P(A|B) \times P(B|A)}. \end{aligned} \tag{4.12}$$

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.

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.

Example 4.10. 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 4.8, a 2×2 contingency table, where an entry such as *mc* represents the number of transactions containing both milk and coffee.

Table 4.8 2 × 2 contingency table for two items.

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

Table 4.9 Comparison of six pattern evaluation measures using contingency tables for a variety of data sets.

Data Set	mc	$\overline{m}c$	$m\overline{c}$	$\overline{m}\,\overline{c}$	χ^2	*lift*	*all_conf.*	*max_conf.*	*Kulc.*	*cosine*
D_1	10,000	1000	1000	100,000	90,557	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	24,740	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

Table 4.9 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 $m\overline{c}$ (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 mc 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, $\overline{mc}$ rep-

resents 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 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

$$IR(A, B) = \frac{|sup(A) - sup(B)|}{sup(A) + sup(B) - sup(A \cup B)}, \tag{4.13}$$

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 4.10.

Example 4.11. 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 4.9, 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?*" Actually, in this case, it is difficult to argue whether the two data sets have positive or negative association. From one point of view, only $mc/(mc + m\overline{c}) = 1000/(1000 + 10{,}000) = 9.09\%$ 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, a very different conclusion.

In this case, it is fair to treat it as neutral, as *Kulc* does. In the meantime, it will be good to also indicate its skewness using the *imbalance ratio (IR)*. According to Eq. (4.13), 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.

4.4 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.

4.5 Exercises

4.1. 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.

4.2. An itemset X is called a *generator* on a data set D if there does not exist a proper subitemset $Y \subset X$ 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?

4.3. 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.

4.4. 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 Fig. 4.4?

4.5. Section 4.2.2 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 4.3(b), (c) into your design.)

4.6. 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,"*):

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

4.7. (**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.

4.8. 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).

4.9. 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. Your algorithm should not require shipping all the data to one site and should not cause excessive network communication overhead.

4.10. 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*?

4.11. 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.

4.12. (**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 4.2.4 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 subbranches. 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.

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

4.14. 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.

	hotdogs	$\overline{hot\ dogs}$	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.

4.15. (**Implementation project**) The DBLP data set *(https://dblp.uni-trier.de/xml/)* consists of over three 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.

4.6 Bibliographic notes

Association rule mining was first proposed by Agrawal, Imielinski, and Swami [AIS93]. The Apriori algorithm discussed in Section 4.2.1 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 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 4.2.3 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 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 4.2.4). An exploration of hyper structure mining of frequent patterns, called H-Mine, was proposed by Pei et al. [PHL01]. 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 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 [PS91]. 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.

CHAPTER

Classification: basic concepts and methods

6

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, or identify the early sign of cognitive impairment based on a patient's functional magnetic resonance imaging (fMRI) scan, or help a self-driving car automatically recognize various road signs. 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. Traditional classification algorithms typically assume a small or medium data size. Modern classification techniques have built on such work, developing scalable classification and prediction techniques capable of handling very large amounts of data. Classification belongs to supervised learning and is closely connected to many other data mining tasks. Classification has numerous applications, including fraud detection, target marketing, performance prediction, manufacturing, medical diagnosis, and many more.

We start off by introducing the main ideas of classification in Section 6.1. In the rest of this chapter, you will learn the basic techniques for data classification such as how to build decision tree classifiers (Section 6.2), Bayes classifiers (Section 6.3), lazy learners (Section 6.4), and linear classifiers (Section 6.5). Section 6.6 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 improving classifier accuracy are presented in Section 6.7, including ensemble methods and class-imbalanced data (i.e., where the main class of interest is rare).

6.1 Basic concepts

We introduce the concept of classification in Section 6.1.1. Section 6.1.2 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.

6.1.1 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, and her colleague from the risk management department wishes to detect fraudulent transactions. A marketing manager at an electronics store needs data analysis to help guess whether a customer with a given profile will buy a new computer, or understand the *sentiment* of social media posts regarding a newly released product, or detect *fake reviews* about a new product from an

Data Mining. https://doi.org/10.1016/B978-0-12-811760-6.00016-3

online review site, or identify a subscribed customer who is likely to switch to a competitive electronics store (i.e., churn prediction). An IT security analyst wants to know if the network system is under attack (intrusion detection) or if a given application is contaminated with malware (malware detection). A teacher wishes to know if a student enrolled in an online course will drop out before she completes the course. A talent recruiter wants to know if an individual is looking for the next career move. A medical researcher wants to analyze breast cancer data to predict which one of three specific treatments a patient should receive, a cardiologist wants to identify the patient who is likely to have a congestive heart failure based on her chronic medical history, a neuroscientist wants to identify the early sign of cognitive impairment (which could lead to, say Alzheimer's disease) based on a patient's functional magnetic resonance imaging (fMRI) scan. An intelligent question-answering system needs to understand what type of question the user is asking (question classification), as the first step to automatically provide a high-quality answer. A self-driving car needs to automatically recognize various road signs (e.g., 'stop,' 'detour,' etc.). A physicist needs to identify *high energy event* from massive experiment data, which might lead to new discoveries. Law enforcement wishes to predict the crime hot spot so that the precaution measures can be taken proactively.

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; or "positive" or "negative" for sentiment classification; or "yes" or "no" for the marketing data; or "dropout" or "stay" for online course enrollment, or "treatment A," "treatment B," or "treatment C" for the medical data; or various question types for a question-answering system. 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; or a realtor might be interested in knowing the average house pricing of the next year in different residential areas; or a career planner wants to forecast the average yearly income of students immediately after graduating from the college in different majors. This kind of 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. **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. **Ranking** is another type of numerical prediction where the model predicts the ordered values (i.e., ranks), for example, a web search engine (e.g., Google) ranks the relevant webpages with respect to a given query, with the higher-ranked webpages being more relevant to the query. Classification and numeric prediction are the two major types of **prediction problems**. This chapter primarily focuses on classification. It is worth pointing out that classification and numerical prediction (e.g., regression) are closely related to each other. Many classification techniques can be modified for the purpose of regression. We will see some examples, including regression trees (Section 6.2), lazy learners (Section 6.4.1), linear regression (Section 6.5), and gradient tree boosting (Section 6.7.1).

6.1.2 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

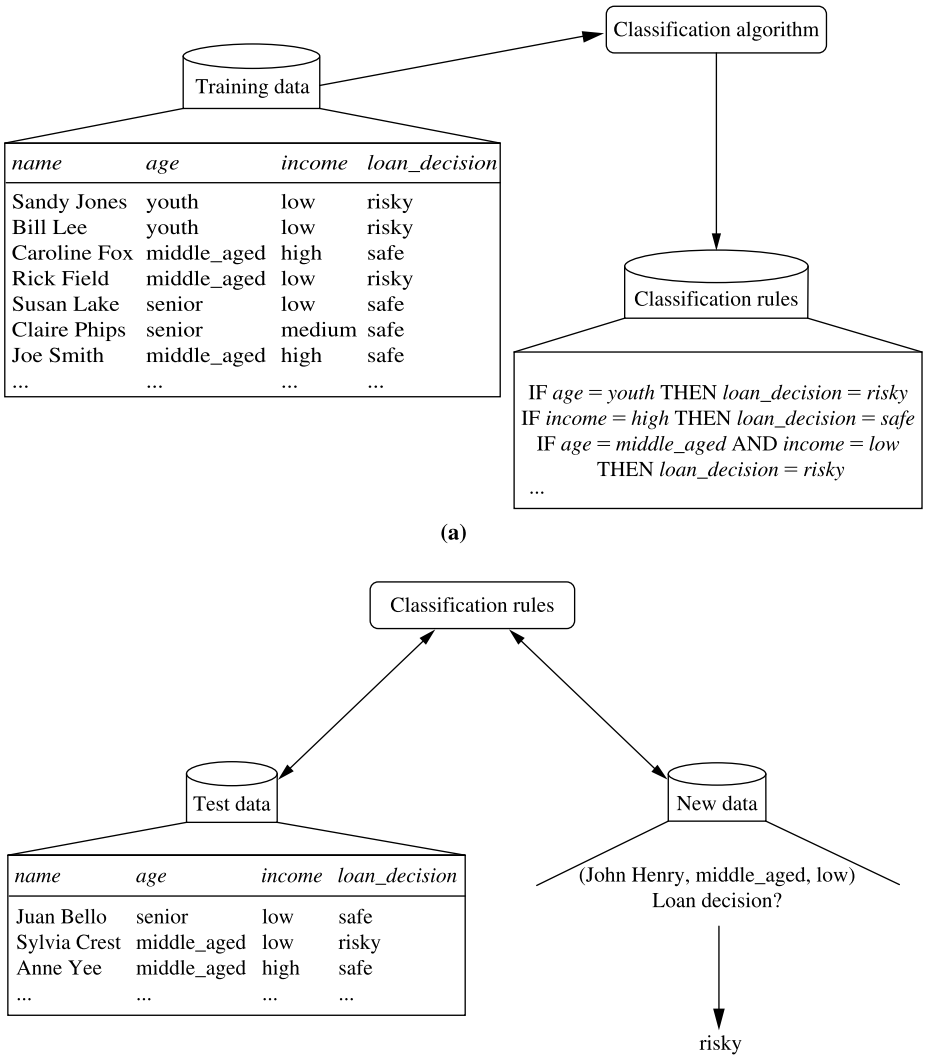


FIGURE 6.1

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 acceptable, the rules can be applied to the classification of new data tuples.

to predict class labels for given data). The process is shown for the loan application data in Fig. 6.1. 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** (also known as the 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]

Because the class label of each training tuple *is provided*, this step belongs to **supervised learning** (i.e., the learning of the classifier is "supervised" in that it is told to which class each training tuple belongs). The scope of supervised learning is larger than classification, and it broadly encompasses learning methods for training a numerical prediction model (e.g., regression, ranking) if the true target values of training tuples are known during the learning step. Supervised learning contrasts with **unsupervised learning** (e.g., **clustering**), in which the true target value (e.g., 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. Likewise, we could use clustering techniques to find social media posts sharing similar topics without knowing their actual class labels. Clustering is the topic of Chapters 8 and 9. The landscape of the prediction problem (e.g., classification, regression, ranking) has gone beyond supervised vs. unsupervised learning. To name a few, in **semisupervised classification**, it builds a classifier based on a limited number of labeled training tuples (whose true class labels are given during training) and a large number of unlabeled training tuples (whose class labels are unknown during training); in **zero-shot learning**, some class label might appear *after* the classification model has been built. In other words, during the training phase, there are no (i.e., zero) labeled training tuples for such a class label. Both semisupervised learning and zero-shot learning belong to **weakly supervised learning** in that the supervision information for training the model is weaker than the standard supervised learning. For the classification task, this means that the supervision (i.e., the true class labels of training tuples) is known only for a small fraction of the entire training set in semisupervised learning; or is absent for certain class label(s) in zero-shot learning. *Classification with weak supervision* will be introduced in Chapter 7.

The 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 map-

[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 these two terms interchangeably. In our notation, any variable representing a vector is typically 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*.

ping is represented as classification rules that identify loan applications as being either safe or risky (Fig. 6.1(a)). 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 (Fig. 6.1(b)), 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 too optimistic, because the classifier tends to **overfit** the data (i.e., during learning it may incorporate some particular anomalies of the training data that do not represent the general data set). 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 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 6.6 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 Fig. 6.1(a) from the analysis of data from previous loan applications can be used to approve or reject new or future loan applicants.

6.2 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 Fig. 6.2. It represents the concept *buys_computer*; that is, it predicts whether a customer at an electronics store is

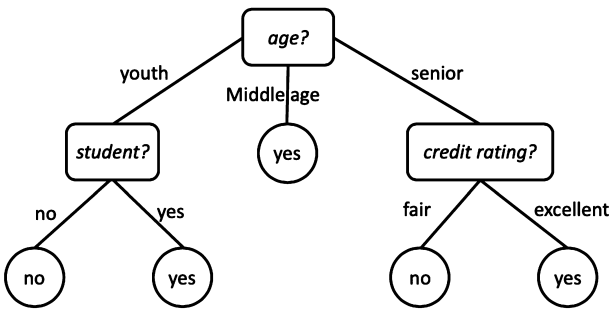


FIGURE 6.2

A decision tree for the concept *buys_computer*, indicating whether a 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 (or circles). 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 6.2.1, 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 6.2.2. 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 6.2.3.

6.2.1 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 Dichotomizer). 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 independent 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 Fig. 6.3. 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*. D is 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

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 6.3

Basic algorithm for inducing a decision tree from training tuples.

according to class. This procedure employs an attribute selection measure such as information gain or the Gini impurity. (We will introduce these measures in the next subsection.) Whether the tree is strictly binary is generally driven by the attribute selection measure. Some attribute selection measures, such as the Gini impurity, 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]
- 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

[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.

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 10–11). There are three possible scenarios, as illustrated in Fig. 6.4. 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 directly correspond to the known values of A. A branch is created for each known value, a_j, of A and labeled with that value (Fig. 6.4(a)). Partition D_j is the subset of class-labeled tuples in D having value a_j of A. Because all the tuples in a given partition have the same value for A, A does not need to be

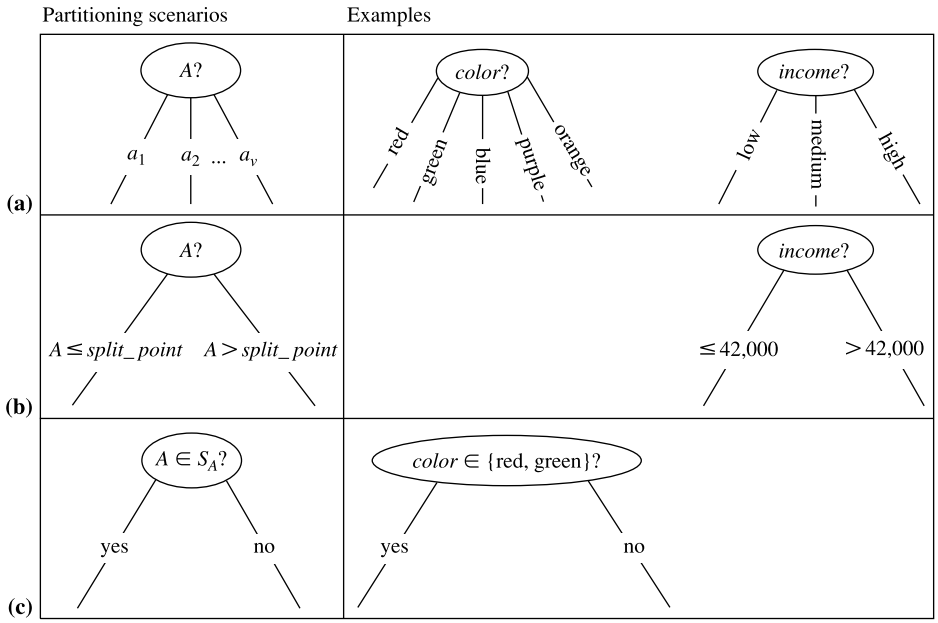


FIGURE 6.4

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.

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 (Fig. 6.4(b)). 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 (Fig. 6.4(c)). 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, it restructures 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 6.2.2) and the mechanisms used for pruning (Section 6.2.3).

Decision tree is closely related to another type of tree, called **regression tree**, which is used to predict the continuous output value. A regression tree is very similar to a decision tree in that it also partitions the entire attribute space into multiple subregions, each corresponding to a leaf node. The main difference is as follows. In a regression tree, a leaf node holds a *continuous value* instead of a

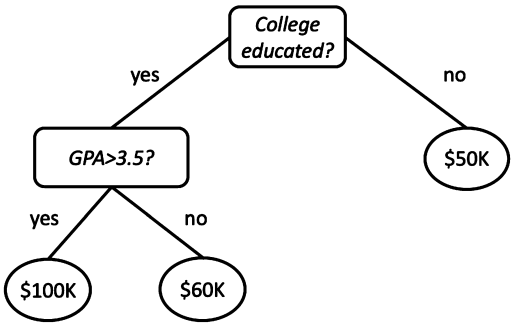


FIGURE 6.5

A regression tree for predicting the average yearly income based on an individual's education. The values of the three leaf nodes are calculated as follows. $50K is the average yearly income of all training individuals who do not have a college degree; $60K is the average yearly income of all training individuals who have a college degree with a GPA less than or equal to 3.5; and $100K is the average yearly income of all training individuals who have a college degree with a GPA higher than 3.5. The leaf node values ($50K, $60K, and $100K) are used to predict the yearly income of any test individual who falls into the corresponding leaf nodes.

categorical value (i.e., class label) in a decision tree. The continuous value of a leaf node is learned during the training phase, which is set as the average output value of all training tuples fallen in the corresponding subregions. CART uses **residual sum of squares** (RSS) as the objective function, which is the sum of the squared difference between the actual and predicted output values of training tuples

$$\text{RSS} = \sum_i (y_i - \hat{y}_i)^2, \tag{6.1}$$

where y_i is the actual output value of the ith training tuple, and $\hat{y}_i$ is the predicted output by the regression tree. Choosing the average output of all training tuples in the corresponding subregion is optimal in that it minimizes the RSS in Eq. (6.1). Each leaf node value is then used to predict the output of a test tuple which falls into it. Fig. 6.5 presents an example of a regression tree for predicting the average yearly income based on an individual's education (e.g., whether or not the individual attended the college, the average GPA at college, etc.).

6.2.2 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 impurity*.

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 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

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i), \tag{6.2}$$

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).

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

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

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Info(D_j). \tag{6.3}$$

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. $Info_A(D)$ is also known as the conditional entropy of D (conditioned on the attribute A).

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,

$$Gain(A) = Info(D) - Info_A(D). \tag{6.4}$$

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 $Info_A(D)$).

Example 6.1. Induction of a decision tree using information gain. Table 6.1 presents a training set, D, of class-labeled tuples randomly selected from the customer database of an electronics store. (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

Table 6.1 Class-labeled training tuples from the customer database of an electronics store.

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

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. (6.2) 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. (6.3), the expected information needed to classify a tuple in D if the tuples are partitioned according to *age* is

$$\begin{aligned} Info_{age}(D) &= \frac{5}{14} \times \left(-\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5}\right) \\ &+ \frac{4}{14} \times \left(-\frac{4}{4}\log_2\frac{4}{4}\right) \\ &+ \frac{5}{14} \times \left(-\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5}\right) \\ &= 0.694 \text{ bits}. \end{aligned}$$

Hence, the gain in information from such 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 Fig. 6.6. 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 Fig. 6.2. □

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

$$\frac{a_i + a_{i+1}}{2}. \tag{6.5}$$

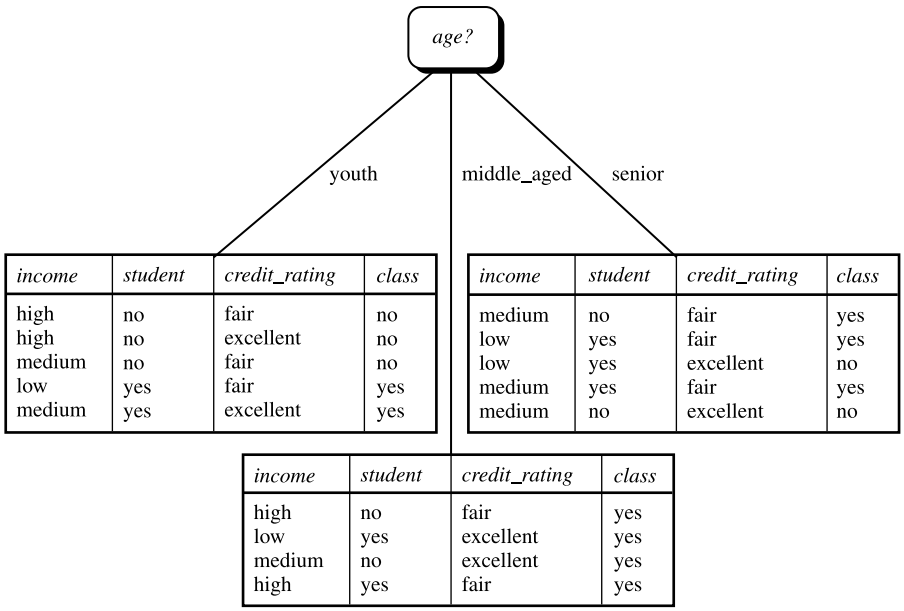


FIGURE 6.6

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.

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. (6.3). 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

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2\left(\frac{|D_j|}{|D|}\right). \tag{6.6}$$

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

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo_A(D)}. \tag{6.7}$$

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 6.2. Computation of gain ratio for the attribute *income*. A test on *income* splits the data of Table 6.1 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. (6.6) to obtain

$$\begin{aligned} SplitInfo_{income}(D) &= -\frac{4}{14} \times \log_2\left(\frac{4}{14}\right) - \frac{6}{14} \times \log_2\left(\frac{6}{14}\right) - \frac{4}{14} \times \log_2\left(\frac{4}{14}\right) \\ &= 1.557. \end{aligned}$$

From Example 6.1, we have $Gain(income) = 0.029$. Therefore $GainRatio(income) = 0.029/1.557 = 0.019$. □

Gini impurity

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

$$Gini(D) = 1 - \sum_{i=1}^{m} p_i^2, \tag{6.8}$$

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 impurity 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 $(2^v - 2)/2$ 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 impurity of D given that partitioning is

$$Gini_A(D) = \frac{|D_1|}{|D|}Gini(D_1) + \frac{|D_2|}{|D|}Gini(D_2). \tag{6.9}$$

For each attribute, each of the possible binary splits is considered. For a discrete-valued attribute, the subset that gives the minimum Gini impurity 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 impurity 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

$$\Delta Gini(A) = Gini(D) - Gini_A(D). \tag{6.10}$$

The attribute that maximizes the reduction in impurity (or, equivalently, has the minimum Gini impurity) 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 6.3. Induction of a decision tree using the Gini impurity. Let D be the training data shown earlier in Table 6.1, 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. (6.8) for the Gini impurity 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 impurity 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 impurity 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 impurity 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 impurity. Evaluating *age*, we obtain {*youth, senior*} (or {*middle_aged*}) as the best split for *age* with a Gini impurity of 0.375; the attributes *student* and *credit_rating* are both binary, with Gini impurity values of 0.367 and 0.429, respectively.

The attribute *age* and splitting subset {*youth, senior*} therefore give the minimum Gini impurity 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. □

"So, what is the relationship between Gini impurity and information gain?" Intuitively, both measures aim to quantify to what extent the impurity will be reduced if we split the current node based on the given attribute. Information gain, rooted in information theory, measures the impurity based on (the change of) the average amount of information needed to identify the class label of a tuple. Gini impurity is related to *mis-classification* in the following way. Based on the class label distribution in the current node, it tells how likely a randomly chosen tuple will be mis-classified if it is assigned to a random class label. Gini impurity is always used for binary split, whereas information gain allows multiway split. In terms of computation, Gini impurity is slightly more efficient than information gain, since the latter involves the logarithm computation. In practice, however, both measures often lead to very similar decision trees.

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 impurity 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 Gini impurity in certain cases) and G-statistic (an information theoretic measure that is a close approximation to χ^2 distribution).

For regression tree, it is natural to use RSS (Eq. (6.1)) as the splitting criteria. That is, the best split point for a given attribute is the one that leads the smallest RSS. We choose the attribute with the minimum RSS to split the tree node into two nodes, including left leaf node and right leaf node.

Example 6.4. Let us look at an example in Table 6.2 on how to use RSS to find the best split point. Suppose there are five training tuples at a regression tree node, and each training tuple has a true output value y_i and a continuous attribute $x_i (i = 1, ..., 5)$. We want to find the best split point for attribute x_i to split the tree node into two leaf nodes. More specifically, all the tuples whose x_i is less than or equal to the split point will go to the left leaf node, and the remaining training tuples will go to the right leaf node.

Table 6.2 Training data for regression.

attribute x_i	1	2	3	4	5
output y_i	10	12	8	20	22

Given five training tuples at a regression tree node, each with a true output value y_i and a continuous attribute $x_i (i = 1, ..., 5)$. We want to find the best split point for attribute x_i to split the tree node into two nodes (left node and right node).

Table 6.3 Using RSS to choose the best split point for data tuples in Table 6.2.

candidate split point x_i	1.5	2.5	3.5	4.5
predicted value of left leaf node y_l	10	11	10	12.5
predicted value of right leaf node y_r	15.5	16.7	21	22
RSS	131	116.67	10	83

Since x_i is a continuous attribute with five possible values, there are four candidate split points, including $x_i = 1.5$, $x_i = 2.5$, $x_i = 3.5$ and $x_i = 4.5$. For each candidate split point, we partition the current tree node into two leaf nodes. The average output value y_l of the training tuples in the left leaf node is used to predict the output of all tuples residing in the left leaf node. Likewise, the average output value y_r of the training tuples in the right leaf node is used to predict the output of all tuples residing in the right leaf node. For example, if the split point $x_i = 1.5$, only the first training tuple goes to the left leaf node, and we have that $y_l = y_1 = 10$; and $y_r = (y_2 + y_3 + y_4 + y_5)/4 = (12 + 8 + 20 + 22)/4 = 15.5$. Using the predicted output values for all five training tuples (y_l or y_r), we can use Eq. (6.1) to calculate RSS. Again, if the split point $x_i = 1.5$, we have that $\text{RSS} = \sum_{i=1}^{5}(y_i - \hat{y}_i)^2 = (y_1 - y_l)^2 + (y_2 - y_r)^2 + (y_3 - y_r)^2 + (y_4 - y_r)^2 + (y_5 - y_r)^2 = 122.25$. The computation results for all four possible split points are summarized in Table 6.3. Since $x_i = 3.5$ has the smallest RSS, it is chosen as the split point. □

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 solution is preferred. The philosophy underlying the MLD principle is **Occam's razor**, also known as *law of parsimony*. In data mining and machine learning, Occam's razor is often translated into a design principle that one should favor a model with a shorter description (hence minimum description length) for the data over a lengthier model, provided that everything else is equal (e.g., both shorter and lengthier models share the same training set errors).

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 2 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 6.10).

"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 single attribute selection measure has been found to be significantly superior to others. Most measures give quite good results.

6.2.3 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 Fig. 6.7. 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 label among the subset tuples or the probability distribution of the class labels of those tuples.

When constructing a tree, measures such as statistical significance, information gain, Gini impurity, 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.

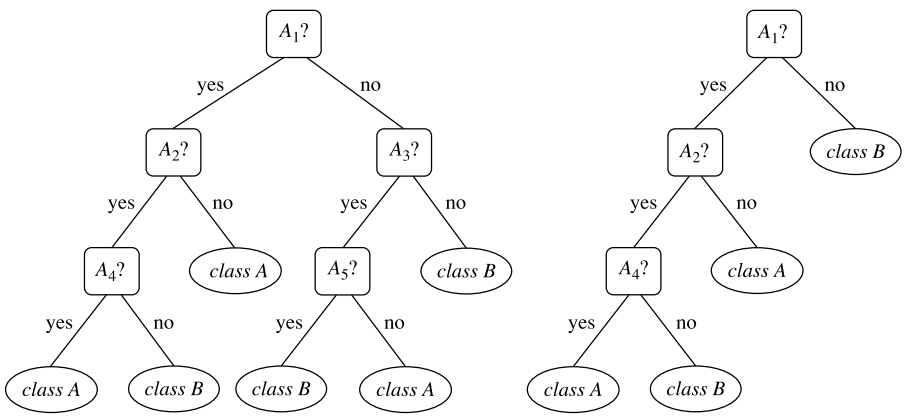


FIGURE 6.7

An unpruned decision tree (left) and a pruned version of it (right).

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 label among the subtree being replaced. For example, notice the subtree at node "A_3?" in the unpruned tree of Fig. 6.7. 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 the cost complexity. This set is independent (1) of the training set used to build the unpruned tree and (2) 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 pruning 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 6.2.2. The basic idea is that the simplest solution is preferred. Unlike cost complexity pruning, it does not require an independent set of tuples (i.e., the pruning set).

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* (Fig. 6.8), making them overwhelming to interpret. **Repetition** occurs when an attribute is repeatedly tested along a given branch of the tree (e.g., *"age < 60?,"* 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 Chapter 7, which shows how a *rule-based classifier* can be constructed by extracting IF-THEN rules from a decision tree.

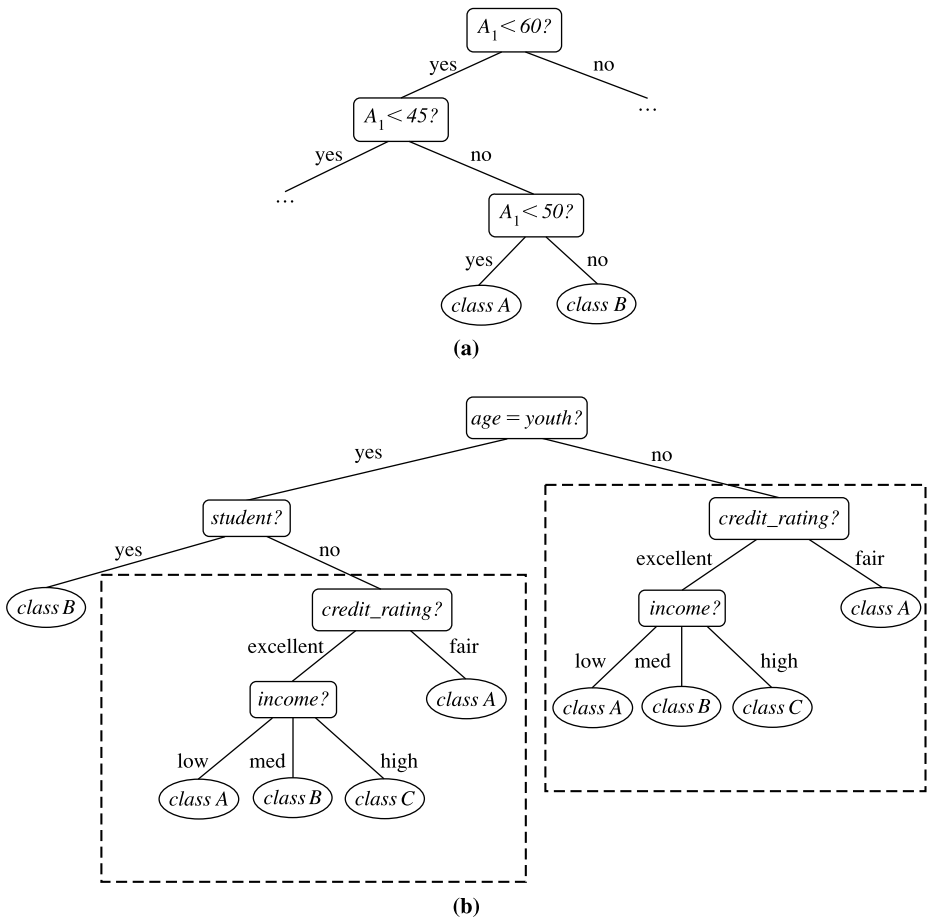


FIGURE 6.8

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?*").

6.3 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 trees 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 6.3.1 reviews basic probability notation and Bayes' theorem. In Section 6.3.2, you will learn how to do naïve Bayesian classification.

6.3.1 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 conditional 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. In classification, $P(\boldsymbol{X}|H)$ is also often referred to as *likelihood*.

$P(\boldsymbol{X})$ 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. In classification, $P(\boldsymbol{X})$ is also often referred to as *marginal probability*.

"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

$$P(H|\boldsymbol{X}) = \frac{P(\boldsymbol{X}|H)P(H)}{P(\boldsymbol{X})}. \tag{6.11}$$

"What does Bayes classifier look like?" Suppose that there are m classes, $C_1, C_2, \ldots, C_m$. Given a tuple, $\boldsymbol{X}$, we want to predict which class it belongs to. In Bayes classifier, it first calculates the posterior probabilities for each of the m classes, $P(C_i|\boldsymbol{X})$ $(i = 1, \ldots, m)$, and then predicts that tuple $\boldsymbol{X}$ belongs to the class with the highest posterior probability. In the above example, given a customer, $\boldsymbol{X}$, of 35 years old and earning $40,000, we want to predict if the customer will buy a computer. So, in this task, there are two possible classes (buy computer vs. not buy computer). Suppose $P(\text{buy computer}|\boldsymbol{X}) = 0.8$ and $P(\text{not buy computer}|\boldsymbol{X}) = 0.2$. Bayes classifier will predict that the customer $\boldsymbol{X}$ will buy a computer.

"So, how good is Bayes classifier?" In theory, Bayes classifier is *optimal* in the sense that it has the smallest classification error rate compared to *all* other classifiers. Since Bayes classifier is a probabilistic method, it could make a wrong prediction for any given tuple. In the above example, Bayes classifier predicts the customer will buy a computer. Since $P(\text{not buy computer}|\boldsymbol{X}) = 0.2$, there is 20% chance that the prediction the Bayes classifier makes is incorrect. However, since Bayes classifier always predicts the class with the maximum posterior probability, the probability that its prediction is wrong for a given tuple $\boldsymbol{X}$ (which is often called *risk*) is the lowest in comparison to all other classifiers. In our example, the risk for the given customer is 0.2. In other words, there is 20% probability that the prediction by Bayes classifier is wrong. Therefore, the overall classification error of Bayes classifier, which is the expectation (i.e., the weighted average) of the risk of all possible tuples, is the lowest in all possible classifiers. Given its theoretic optimality, Bayes classifier plays a foundational role in the statistical machine learning community. For example, many classifiers (e.g., naïve Bayesian classifier, k-Nearest-Neighbor classifier, logistic regression, Bayesian network, etc.) can be viewed as approximated Bayes classifiers. Bayes classifier is also useful in that it provides 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 Bayes classifier.

"Then, why do not we just use Bayes classifier?" According to Bayes' theorem (Eq. (6.11)), in order to calculate the posterior probabilities $P(C_i|\boldsymbol{X})$ $(i = 1, \ldots, m)$, we need to know the conditional probabilities $P(\boldsymbol{X}|C_i)$ $(i = 1, \ldots, m)$, the priors $P(C_i)$ $(i = 1, \ldots, m)$ and the marginal probability $P(\boldsymbol{X})$. In Bayes classifier, we only need to know which class has the highest posterior probability and for a given tuple $\boldsymbol{X}$, its marginal probability is independent of different classes. In other words, different posterior probabilities $P(C_i|\boldsymbol{X})$ $(i = 1, \ldots, m)$ share the same marginal probability $P(\boldsymbol{X})$. Therefore for the purpose of predicting which class a given tuple belongs to, we only need to estimate the conditional probabilities $P(\boldsymbol{X}|C_i)$ $(i = 1, \ldots, m)$, and the priors $P(C_i)$ $(i = 1, \ldots, m)$.[5]

It is relatively easy to estimate the priors $P(C_i)$ $(i = 1, \ldots, m)$ from the training data set (the details will be introduced in the next section). On the other hand, it is usually very challenging to directly estimate the conditional probabilities $P(\boldsymbol{X}|C_i)$ $(i = 1, \ldots, m)$. To see this, let us assume there are n binary attributes $A_1, A_2, \ldots, A_n$. Then, the n-dimensional attribute vector $\boldsymbol{X}$ has 2^n possible values and we need to estimate the conditional probability of each possible value of the attribute vector with respect to each class label.[6] In other words, the attribute value space is exponential! It is very difficult to estimate such a large number of parameters for the conditional probabilities.[7]

Therefore the main difficulty for Bayes classifier lies in how to efficiently estimate the conditional probabilities, often with some approximation. Many solutions have been developed. One of such efforts, probably the simplest yet quite effective solution, is the naïve Bayesian classifier, which we introduce next.

---

[5] The marginal probability $P(\boldsymbol{X})$ itself can be calculated based on the conditional probabilities $P(\boldsymbol{X}|C_i)$ $(i = 1, \ldots, m)$, and the priors $P(C_i)$ $(i = 1, \ldots, m)$ based on the law of total probability, that is, $P(\boldsymbol{X}) = \sum_{i=1}^{m} P(\boldsymbol{X}|C_i)P(C_i)$. It is necessary to calculate the marginal probability in some scenarios (e.g., to estimate the risk of Bayes classifier).

[6] The total number of the parameters we need to estimate for conditional probabilities in this case is $m(2^n - 1)$. The details are left as an exercise.

[7] In statistics, it means that the estimation results bear high variance, which are not reliable.

6.3.2 Naïve Bayesian classification

The **naïve Bayesian** classifier, or **simple Bayesian** classifier, follows the same procedure as Bayes classifier, except the way it estimates the conditional probabilities. In detail, it 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. (6.11)),

$$P(C_i|\boldsymbol{X}) = \frac{P(\boldsymbol{X}|C_i)P(C_i)}{P(\boldsymbol{X})}. \tag{6.12}$$

3. As $P(\boldsymbol{X})$ is constant for all classes, we only need to find out which class maximizes $P(\boldsymbol{X}|C_i)P(C_i)$. 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 a data set with many attributes, it would be extremely computationally expensive to compute $P(\boldsymbol{X}|C_i)$ for the aforementioned reasons. 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., there are no dependence relationships among the attributes, *if* we know which class the tuple belongs to.). Thus

$$\begin{aligned} P(\boldsymbol{X}|C_i) &= \prod_{k=1}^{n} P(x_k|C_i) \\ &= P(x_1|C_i) \times P(x_2|C_i) \times \cdots \times P(x_n|C_i). \end{aligned} \tag{6.13}$$

   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.[8]

[8] In statistics, this is the classic maximum likelihood estimation (MLE) method.

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

$$g(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{6.14}$$

so that

$$P(x_k|C_i) = g(x_k, \mu_{C_i}, \sigma_{C_i}). \tag{6.15}$$

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. (6.14), 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 (i.e., *buys_computer* = *yes*), 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. (6.14) 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

$$P(\boldsymbol{X}|C_i)P(C_i) > P(\boldsymbol{X}|C_j)P(C_j) \quad \text{for } 1 \le j \le m, j \ne i. \tag{6.16}$$

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 is naïve Bayesian classifier?" Notice that the only difference between naïve Bayesian classifier and Bayes classifier is the class-conditional independence assumption. Therefore if such an assumption indeed holds, naïve Bayesian classifier would be optimal with the smallest possible classification error. 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. Nonetheless, various empirical studies of this classifier in comparison to decision trees and selected neural network classifiers have found it to be comparable in some domains. Another advantage of naïve Bayesian classifier is that it can naturally handle the missing attribute(s).

Example 6.5. 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 6.3 for decision tree induction. The training data were shown earlier in Table 6.1. 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 find out which class maximizes $P(\boldsymbol{X}|C_i)P(C_i)$, 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:

$$\begin{aligned}
P(age = youth \mid buys_computer = yes) &= 2/9 = 0.222\\
P(age = youth \mid buys_computer = no) &= 3/5 = 0.600\\
P(income = medium \mid buys_computer = yes) &= 4/9 = 0.444\\
P(income = medium \mid buys_computer = no) &= 2/5 = 0.400\\
P(student = yes \mid buys_computer = yes) &= 6/9 = 0.667\\
P(student = yes \mid buys_computer = no) &= 1/5 = 0.200\\
P(credit_rating = fair \mid buys_computer = yes) &= 6/9 = 0.667\\
P(credit_rating = fair \mid buys_computer = no) &= 2/5 = 0.400.
\end{aligned}$$

Using these probabilities, we obtain

$$\begin{aligned}
P(\boldsymbol{X}|buys_computer = yes) = {} & P(age = youth \mid buys_computer = yes)\\
& \times P(income = medium \mid buys_computer = yes)\\
& \times P(student = yes \mid buys_computer = yes)\\
& \times P(credit_rating = fair \mid buys_computer = yes)\\
= {} & 0.222 \times 0.444 \times 0.667 \times 0.667 = 0.044.
\end{aligned}$$

Similarly,

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

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

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

Therefore the naïve Bayesian classifier predicts $buys\_computer = yes$ for tuple $\boldsymbol{X}$. □

"What if I encounter probability values of zero?" Recall that in Eq. (6.13), we estimate $P(\boldsymbol{X}|C_i)$ 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 $P(\boldsymbol{X}|C_i)$ for *each* class ($i = 1, 2, \ldots, m$) to find the class C_i for which $P(\boldsymbol{X}|C_i)P(C_i)$ 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 $\boldsymbol{X}$, 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 6.5, we have two classes ($m = 2$), namely $buys\_computer = yes$ and $buys\_computer = no$. Therefore, for the attribute–value pair $student = yes$ of $\boldsymbol{X}$, say,

we need two counts—the number of customers who are students and for which $buys\_computer = yes$ (which contributes to $P(\boldsymbol{X}|buys\_computer = yes)$) and the number of customers who are students and for which $buys\_computer = no$ (which contributes to $P(\boldsymbol{X}|buys\_computer = no)$).

However, 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. (6.13) would return a zero probability for $P(\boldsymbol{X}|C_i)$, even though, without the zero probability, we may have ended up with a high probability, suggesting that $\boldsymbol{X}$ 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.[9] 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 6.6.

Example 6.6. 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 one 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. □

The main idea of naïve Bayesian classifier lies in the class-conditional independence assumption, which significantly simplifies the estimation of the conditional probabilities $P(\boldsymbol{X}|C_i)$ $(i = 1, ..., m)$. However, this (class-conditional independence assumption) is also one major limitation of naïve Bayesian classifier, since it might not be true for some applications. To address this issue, we need more sophisticated ways to approximate the conditional probabilities, such as Bayesian networks, which will be introduced in the next chapter.

[9] In statistics, this belongs to the Maximum a Posteriori (MAP) method. This can also be viewed as a smoothing technique (i.e., to "smooth" the zero probabilities). In practice, we can also replace 1 by a small integer k. The intuition is that we have k (instead of 1) more tuples for each attribute-value pair.

6.4 Lazy learners (or learning from your neighbors)

The classification methods discussed so far in this book—decision tree induction and Bayesian classification—are both examples of *eager learners*. **Eager learners**, when given a set of training tuples, will construct a generalization (i.e., classification) model before receiving new (e.g., test) tuples to classify. We can think of the learned model as being ready and eager to classify previously unseen tuples.

Imagine a contrasting lazy approach, in which the learner instead waits until the last minute before doing any model construction to classify a given test tuple. That is, when given a training tuple, a **lazy learner** simply stores it (or does only a little minor processing) and waits until it is given a test tuple. Only when it sees the test tuple does it perform generalization to classify the tuple based on its similarity to the stored training tuples. Unlike eager learning methods, lazy learners do less work when a training tuple is presented and more work when making a classification or numeric prediction. Because lazy learners store the training tuples or "instances," they are also referred to as **instance-based learners**, even though all learning is essentially based on instances.

When making a classification or numeric prediction, lazy learners can be computationally expensive. They require efficient storage techniques and are well suited to implementation on parallel hardware. They offer little explanation or insight into the data's structure. Lazy learners, however, naturally support incremental learning. They are able to model complex decision spaces having hyperpolygonal shapes that may not be as easily describable by other learning algorithms (such as hyperrectangular shapes modeled by decision trees). In this section, we look at two examples of lazy learners: *k-nearest-neighbor classifiers* (Section 6.4.1) and *case-based reasoning classifiers* (Section 6.4.2).

6.4.1 *k*-nearest-neighbor classifiers

The k-nearest-neighbor method was first described in the early 1950s. The method is labor-intensive when given a large training set, and did not gain popularity until the 1960s when increased computing power became available. It has since been widely used in the area of pattern recognition.

Suppose you want to make a decision on whether or not you should buy a computer. What would you do? One possible way to make such a decision is to find out your friends' decision on this (whether or not to buy a computer). If most of your close friends buy a computer, maybe you will decide to buy a computer as well. Nearest-neighbor classifiers follow a very similar idea of learning by analogy, that is, by comparing a given test tuple with training tuples that are similar to it. The training tuples are described by n attributes. Each tuple represents a point in an n-dimensional space. In this way, all the training tuples are stored in an n-dimensional attribute space. When given an unknown tuple, a ***k*-nearest-neighbor classifier** searches the attribute space for the k training tuples that are closest to the unknown tuple (i.e., to find your close friends in the above example). These k training tuples are the k "nearest neighbors" of the unknown tuple. Then *k-nearest-neighbor classifier* chooses the most common class label among the k nearest neighbors as the predicted class label of the unknown tuple (i.e., to follow the majority decision of your friends in the above example).

"Closeness" is defined in terms of a distance metric, such as Euclidean distance. The Euclidean distance between two points or tuples, say, $\boldsymbol{X}_1 = (x_{11}, x_{12}, \ldots, x_{1n})$ and $\boldsymbol{X}_2 = (x_{21}, x_{22}, \ldots, x_{2n})$, is

$$dist(\boldsymbol{X}_1, \boldsymbol{X}_2) = \sqrt{\sum_{i=1}^{n} (x_{1i} - x_{2i})^2}. \tag{6.17}$$

In other words, for each numeric attribute, we take the difference between the corresponding values of that attribute in tuple $\boldsymbol{X}_1$ and in tuple $\boldsymbol{X}_2$, square this difference, and accumulate it. The square root is taken of the total accumulated distance count. Typically, we normalize the values of each attribute before using Eq. (6.17). This helps prevent attributes with initially large ranges (e.g., *income*) from outweighing attributes with initially smaller ranges (e.g., binary attributes). Min-max normalization, for example, can be used to transform a value v of a numeric attribute A to v' in the range [0, 1] by computing

$$v' = \frac{v - min_A}{max_A - min_A}, \tag{6.18}$$

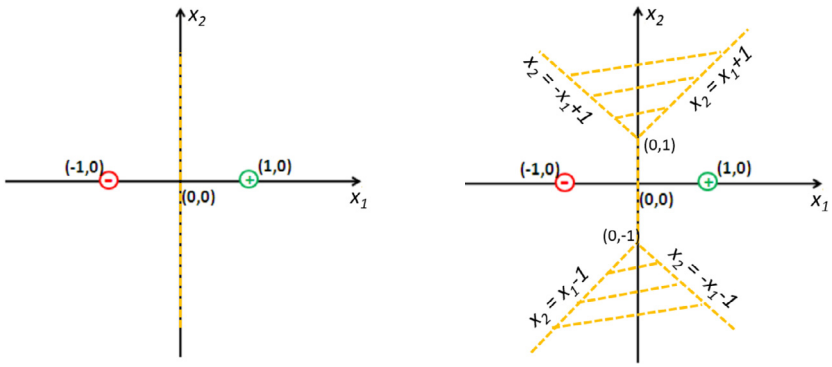
where min_A and max_A are the minimum and maximum values of attribute A. Chapter 2 describes other methods for data normalization as a form of data transformation.

For k-nearest-neighbor classification, the unknown tuple is assigned the most common class label among its k-nearest neighbors. When $k = 1$, the unknown tuple is assigned the class of the training tuple that is closest to it in the attribute space. When $k > 1$, we can take a (weighted) majority voting on the class labels among its k-nearest neighbors. Nearest-neighbor classifiers can also be used for numeric prediction, that is, to return a real-valued prediction for a given unknown tuple. In this case, the classifier returns the (weighted) average value of the real-valued labels associated with the k-nearest neighbors of the unknown tuple.

"*But how can distance be computed for attributes that are not numeric, but nominal (or categorical) such as color?*" The previous discussion assumes that the attributes used to describe the tuples are all numeric. For nominal attributes, a simple method is to compare the corresponding value of the attribute in tuple $\boldsymbol{X}_1$ with that in tuple $\boldsymbol{X}_2$. If the two are identical (e.g., tuples $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ both have the color blue), then the difference between the two is taken as 0. If the two are different (e.g., tuple $\boldsymbol{X}_1$ is blue but tuple $\boldsymbol{X}_2$ is red), then the difference is considered to be 1. Other methods may incorporate more sophisticated schemes for differential grading (e.g., where a larger difference score is assigned, say, for blue and white than for blue and black).

"*What about missing values?*" In general, if the value of a given attribute A is missing in tuple $\boldsymbol{X}_1$ or in tuple $\boldsymbol{X}_2$, we assume the maximum possible difference. Suppose that each of the attributes has been mapped to the range [0, 1]. For nominal attributes, we take the difference value to be 1 if either one or both of the corresponding values of A are missing. If A is numeric and missing from both tuples $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$, then the difference is also taken to be 1. If only one value is missing and the other (which we will call v') is present and normalized, then we can take the difference to be either $|1 - v'|$ or $|0 - v'|$ (i.e., $1 - v'$ or v'), whichever is greater.

"*How can I determine a good value for k, the number of neighbors?*" This can be determined experimentally. Starting with $k = 1$, we use a test set to estimate the error rate of the classifier. This process can be repeated each time by incrementing k to allow for one more neighbor. The k value that gives the minimum error rate may be selected. In general, the larger the number of training tuples, the larger the value of k will be (so that classification and numeric prediction decisions can be based on a larger portion of the stored tuples). As the number of training tuples approaches infinity and $k = 1$, the error rate can be no worse than twice the Bayes error rate (the latter being the theoretical minimum). In



(a) decision boundary using L_2 norm

(b) decision boundary using L_∞ norm.

FIGURE 6.9

The impact of distance metrics on 1-nearest-neighbor classifier. Given two training examples, including a positive example at $(1, 0)$ and a negative example at $(-1, 0)$. The decision boundaries of 1-nearest-neighbor classifier using different distance metrics are quite different from each other. Using L_2 norm (on the left), the decision boundary is a vertical line at $x_2 = 0$. Using L_∞ norm (on the right), the decision boundary includes a line segment between $(0, -1)$ and $(0, 1)$ and two shaded areas.

other words, 1-nearest-neighbor classifier is asymptotically near-optimal. If k approaches infinity, the error rate approaches the Bayes error rate.

Nearest-neighbor classifiers use distance-based comparisons that intrinsically assign equal weight to each attribute. They, therefore, can suffer from poor accuracy when given noisy or irrelevant attributes. The method, however, has been modified to incorporate attribute weighting and the pruning of noisy data tuples. The choice of a distance metric can be critical. The Manhattan (city block) distance (Section 2.3), or other distance measurements, may also be used. Fig. 6.9 presents an illustrative example in terms of the impact of distance metrics on the decision boundary of k-nearest-neighbor classifier.

Nearest-neighbor classifiers can be extremely slow when classifying test tuples. If D is a training database of $|D|$ tuples and $k = 1$, then $O(|D|)$ comparisons are required to classify a given test tuple. By presorting and arranging the stored tuples into search trees, the number of comparisons can be reduced to $O(log(|D|)$. Parallel implementation can reduce the running time to a constant, that is, $O(1)$, which is independent of $|D|$.

Other techniques to speed up classification time include the use of *partial distance* calculations and *editing* the stored tuples. In the **partial distance** method, we compute the distance based on a subset of the n attributes. If this distance exceeds a threshold, then further computation for the given stored tuple is halted, and the process moves on to the next stored tuple. The **editing** method removes training tuples that are proven useless. This method is also referred to as **pruning** or **condensing** because it reduces the total number of tuples stored. Another technique to speed up nearest-neighbor search is via **locality-sensitive-hashing** (LSH). The key idea is to hash the similar tuples into the same bucket with a high probability via *locality-preserving hash functions*. Then, given a test tuple, we first identify which bucket it belongs to, and then we only search the training tuples in the same bucket to identify its nearest neighbors.

6.4.2 Case-based reasoning

Case-based reasoning (CBR) classifiers use a database of problem solutions to solve new problems. Unlike *k*-nearest-neighbor classifiers, which store training tuples as points in Euclidean space, CBR stores the tuples or "cases" for problem solving as complex symbolic descriptions. Business applications of CBR include problem resolution for customer service help desks, where cases describe product-related diagnostic problems. CBR has also been applied to areas, such as engineering and law, where cases are either technical designs or legal rulings in the common law system, respectively. Medical education is another area for CBR, where patient case histories and treatments are used to help diagnose and treat new patients.

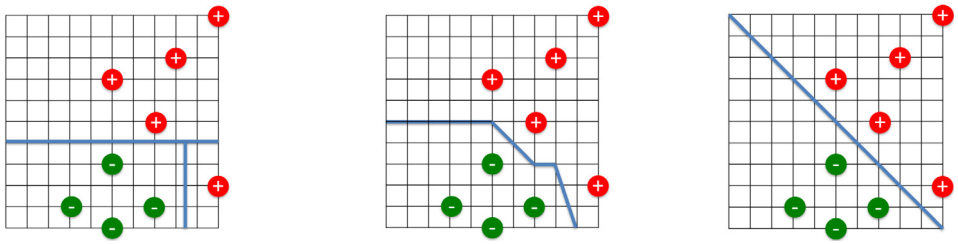
When given a new case to classify, a case-based reasoner will first check if an identical training case exists. If one is found, then the accompanying solution to that case is returned. If no identical case is found, then the case-based reasoner will search for training cases having components that are similar to those of the new case. Conceptually, these training cases may be considered as neighbors of the new case. If cases are represented as graphs, this involves searching for subgraphs that are similar to subgraphs within the new case. The case-based reasoner tries to combine the solutions of the neighboring training cases to propose a solution for the new case. If incompatibilities arise with the individual solutions, then backtracking to search for other solutions may be necessary. The case-based reasoner may employ background knowledge and problem-solving strategies to propose a feasible combined solution.

Key challenges in case-based reasoning include finding a good similarity metric (e.g., for matching subgraphs) and suitable methods for combining solutions. Other challenges include the selection of salient features for indexing training cases and the development of efficient indexing techniques. A trade-off between accuracy and efficiency evolves as the number of stored cases becomes very large. As this number increases, the case-based reasoner becomes more intelligent. After a certain point, however, the system's efficiency will suffer as the time required to search for and process relevant cases increases. As with nearest-neighbor classifiers, one solution is to edit the training database. Cases that are redundant or those that have not proved useful may be discarded for the sake of improved performance. These decisions, however, are not clear-cut, and their automation remains an active area of research.

6.5 Linear classifiers

So far, we have learned a few classifiers which are capable of generating complex decision boundaries. For example, a decision tree classifier might output a hyperrectangular-shaped decision boundary (Fig. 6.10(a)), and a *k*-nearest-neighbor classifier might output a hyperpolygonal-shaped decision boundary (Fig. 6.10(b)). However, what about a simple, linear decision boundary? For the example in Fig. 6.10, intuitively, a linear decision boundary (the straight line in Fig. 6.10(c)) is (almost) as good as decision tree classifiers and *k*-nearest-neighbor classifier in separating the positive training tuples from the negative training ones. Yet, such a linear decision boundary might offer additional advantages, such as efficient computation for training the classifier, better generalization performance, and better interpretability.

In this section, we introduce basic techniques to learn such linear classifiers. We will start with **linear regression**, which forms the basis for linear classifiers. Then, we will introduce two linear classi-



(a) Decision Tree Classifier (b) 1−Nearest-Neighbor Classifier (c) Linear Classifier

FIGURE 6.10

Decision boundaries by different classifiers. Note that this example is *linearly separable*, meaning that a linear classifier (c) can perfectly separate all the positive training tuples from all the negative training tuples. If the training set is *linearly inseparable*, we could still use a linear classifier, at the expense that some training tuples are on the 'wrong' side of the decision boundary. In Chapter 7, we will introduce techniques (e.g., support vector machines) to handle linearly inseparable case.

fiers, including (1) **perception**, which is one of the earliest linear classifiers, and (2) **logistic regression** which is one of the most widely used linear classifiers. Additional linear classifiers will be introduced in Chapter 7, such as linear support vector machines.

6.5.1 Linear regression

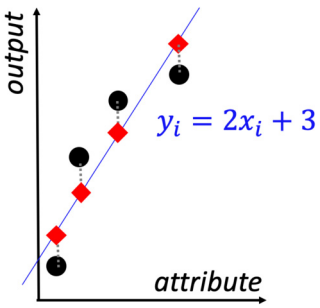
Linear regression is a statistical technique that predicts a continuous value based on one or more independent attributes. For example, we might want to predict the housing price based on the living area or to predict the future income of a student based on which college she attended, in which major and the overall GPA, etc. Since linear regression aims to predict a *continuous* value, it cannot be directly applied to the classification task, where the output is a categorical variable. Nonetheless, the core techniques in linear regression form the basis of linear classifiers. Therefore, let us first briefly introduce linear regression.

Suppose we have n tuples, each of which is represented by p attributes $x_i = (x_{i,1}, ..., x_{i,p})^T$ and a continuous output value y_i $(i = 1, ..., n)$. In linear regression, we want to learn a linear function that maps the p input attributes x_i to the output variable y_i, that is, $\hat{y}_i = w^T x_i + b = \sum_{j=1}^{p} w_j x_{i,j} + b$, where $\hat{y}_i$ is the predicted output value for the ith tuple, $w = (w_1, ..., w_p)^T$ is a p-dimensional weight vector and b is the bias scalar. In other words, linear regression assumes that the output value is a linear weighted summation of the p input attribute values, offset by the bias scalar b. The entries in the weight vector w_j $(j = 1, ..., p)$ tell how important the corresponding attribute $x_{i,j}$ is in predicting the output variable $\hat{y}_i$. In the aforementioned examples, a linear regression model would assume that the housing prices are linearly correlated with the living area; the future income of a student can be predicted by a linear weighted combination of the college she attended, the major, and the overall GPA (plus a bias scalar b). If we know the weight vector w and the bias scalar b, we can make a prediction of the output value based on its p input attribute values.

"So, how can we determine the weight vector w and the bias scalar b?" Intuitively, we want to learn the "best" weight vector w and the "best" bias scalar b from the training data, so that the lin-

Index (i)	1	2	3	4
Attribute (x_i)	1	3	5	7
Output (y_i)	4	10	14	16

(a) Training tuples



(b) Least square regression

FIGURE 6.11

An example of least square regression. (a) Four training tuples. (b) Scatter-plot of the training tuples (black dots) and least square regression model (the blue line). Red diamonds are the predicted output $\hat{y}_i$ $(i = 1, 2, 3, 4)$ and dashed lines indicate the prediction errors ($|y_i - \hat{y}_i|$) of the corresponding training tuples.

ear regression model can make the "best" prediction. That is, the predicted value $\hat{y}_i = w^T x_i + b$ is as close as possible to the actual observed value y_i $(i = 1, ..., n)$. One of the most common linear regression methods is called **least square regression**, which aims to minimize the following loss function $L(w, b) = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{n}(y_i - (w^T x_i + b))^2$. Therefore the best weight vector w and the bias scalar b are the ones that minimize the loss function $L(w, b)$, which measures the sum of the squared difference between the predicted output value $\hat{y}_i$ and the actual observed value y_i. For example, if there is only one input attribute (i.e., $p = 1$), the optimal weight $w = \frac{\sum_{i=1}^{n} x_i (y_i - \bar{y})}{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}(\sum_{i=1}^{n} x_i)^2}$ and the optimal bias scalar $b = \frac{1}{n}\sum_{i=1}^{n}(y_i - wx_i)$, where $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ is the average observed output value among all n training tuples.

Example 6.7. Let us look at an example of least square regression in Fig. 6.11. There are four training tuples, each represented by a single-dimensional attribute x_i and an output variable y_i $(i = 1, 2, 3, 4)$. We want to find least square regression model $y = wx + b$ that predicts the output y based on the input attribute x. We use the two equations mentioned above to find the optimal weight w and the optimal bias scalar b. We first find the optimal weight w as follows. The average output of four training tuples is $\bar{y} = (y_1 + y_2 + y_3 + y_4)/4 = (4 + 10 + 14 + 16)/4 = 11$. Therefore we have that $\sum_{i=1}^{4} x_i (y_i - \bar{y}) = 1(4 - 11) + 3(10 - 11) + 5(14 - 11) + 7(16 - 11) = 40$. In the meanwhile, we have that $\sum_{i=1}^{4} x_i^2 = 1^2 + 3^2 + 5^2 + 7^2 = 84$ and $1/4(\sum_{i=1}^{4} x_i)^2 = (1 + 3 + 5 + 7)^2/4 = 64$. Therefore the optimal weight $w = \frac{\sum_{i=1}^{n} x_i (y_i - \bar{y})}{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}(\sum_{i=1}^{n} x_i)^2} = \frac{40}{84-64} = 2$. Based on the optimal weight w, the optimal bias scalar $b = \frac{\sum_{i=1}^{4}(y_i - wx_i)}{4} = \frac{(4-2\times1)+(10-2\times3)+(14-2\times5)+(16-2\times7)}{4} = 3$. □

"But, what if there are multiple p $(p > 1)$ attributes?" In this case (which is called **multilinear regression**), let us first change our notation a little bit. We assume there is an additional "dummy" attribute which always takes the value of 1 for any tuple. Let the weight for this dummy attribute be w_0. Then the overall weight vector $w = (w_0, w_1, ..., w_p)$ and the new input attribute vector $x_i = (1, x_{i,1}, ..., x_{i,p})$ are both $(p+1)$-dimensional vectors. The multilinear regression model can be re-written as $\hat{y}_i = w^T x_i = w_0 + w_1 x_{i,1} + ... + w_p x_{i,p}$. We use the same loss function as before, that is, $L(w) = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{n}(y_i - (w^T x_i))^2$. It turns out the optimal weight vector w can be computed as $w = (XX^T)^{-1}Xy$, where $X = [x_1, x_2, ..., x_n]$ is a $(p+1) \times n$ matrix, and $y = [y_1, ..., y_n]^T$ is an $n \times 1$ vector. (How to derive the closed form solutions for single linear regression as well as multilinear regression are left as exercises.)

In least square regression, we measure the "goodness" of the learned regression model by the sum of the squared difference between predicted and actual output values. The squared loss might be sensitive to the outliers in the training set. In *robust regression*, it uses alternative loss functions that are less sensitive to such outliers. For example, the Huber method in robust regression uses the following loss: $L(w) = \sum_{i=1}^{n} l_H(y_i - \hat{y}_i)$, where $l_H(y_i - \hat{y}_i) = (y_i - \hat{y}_i)^2$ if $|y_i - \hat{y}_i| < \theta$, $l_H(y_i - \hat{y}_i) = 2\theta|y_i - \hat{y}_i| - \theta^2$ otherwise, and $\theta > 0$ is a user-specified parameter. Notice that the optimal weight vector w for multilinear regression involves a matrix inverse (i.e., $(XX^T)^{-1}$). In case $p > n$ (i.e., the number of attributes is more than the number of training tuples), such a matrix inverse does not exist. An effective way to address this issue is to introduce a *regularization term* regarding the norm of the weight vector w. For example, if we use l_2 norm of the weight vector w, the corresponding regression model is called Ridge regression; if we use l_1 norm of the weight vector w instead, the corresponding regression model is called Lasso regression which often learns a *sparse* weight vector. This means that some entries of the learned weight vector w are zeros, which indicates that those attributes are not used in the regression model. In Section 7.1, we will use Lasso regression for feature selection.

6.5.2 Perceptron: turning linear regression to classification

"How can we modify a linear regression model to perform classification task?" Suppose we have a binary classification task.[10] The output value y_i for a given tuple is a binary variable: $y_1 = +1$ indicates the ith tuple is a positive tuple (e.g., *buy computer*) and $y_i = 0$ indicates the ith tuple is a negative one (e.g., *not buy computer*). One way to modify the linear regression model for such a binary classification task is to use the *sign* of the output of the linear regression model as the predicted class label, that is, $\hat{y}_i = \text{sign}(w^T x_i)$, where $\hat{y}_i$ is the predicted class label for ith tuple, $\text{sign}(z) = 1$ if $z > 0$ and $\text{sign}(z) = 0$ otherwise. Notice that we use the same notation as multilinear regression where we have introduced a "dummy" attribute which always takes the value of 1 for any tuple. Therefore if we know the weight vector w, we can use it to predict the class label of a given tuple as follows. We compute a linear combination of the attribute values of the given tuple, weighted by the corresponding entries of the weight vector w. If the resulting value of such a linear combination is positive, we predict that the given tuple is a positive tuple. Otherwise, we predict that it is a negative one.

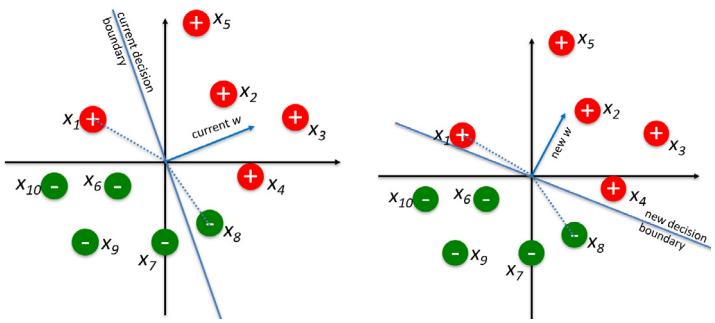
"How can we find the optimal weight vector w from a set of training tuples?" The classic learning algorithm to train a perceptron is as follows. We start with an initial guess of the weight vector w (e.g.,

[10] For both perceptron and logistic regression classifiers that we will introduce next, we focus on binary classification task. However, the techniques we introduce can be generalized to handle multiclass classification task for both classifiers.

we can simply set $w = 0$). Then, the learning algorithm will iterate until it converges, or the maximum iteration number or some other preset stopping criteria are met. In each iteration, we do the following for each training tuple x_i. We try to predict the class label of x_i using the current weight vector w, that is, $\hat{y}_i = \text{sign}(w^T x_i)$. If the prediction is correct (i.e., $\hat{y}_i = y_i$), we do nothing about the weight vector. However, if the prediction is incorrect (i.e., $\hat{y}_i \neq y_i$), we update the current weight vector in one of the following two ways. If $y_i = +1$ (i.e., the ith tuple is a positive tuple, but the current classifier predicts it is a negative tuple), we update weight vector as $w \leftarrow w + \eta x_i$. If $y_i = 0$ (i.e., the ith tuple is a negative tuple, which is wrongly predicted by the current classifier as a positive tuple), we update weight vector as $w \leftarrow w - \eta x_i$, where $\eta > 0$ is the user-specified learning rate. So, the intuition is that in each iteration of the training process, the algorithm will focus on those wrongly predicted training tuples by the current weight vector w. If the wrongly predicted training tuple x_i is a positive tuple, we update the weight vector w by moving it *towards* the attribute vector x_i of this training tuple (i.e., $w \leftarrow w + \eta x_i$). On the other hand, if the wrongly predicted training tuple x_i is a negative tuple, we update the weight vector w by moving it *away from* the attribute vector x_i of this training tuple (i.e., $w \leftarrow w - \eta x_i$).

Example 6.8. Let us look at an example in Fig. 6.12 for training a perceptron classifier. In Fig. 6.12, we assume the bias $w_0 = 0$ for illustration clarity. Fig. 6.12(a) (left) shows the current decision boundary and the weight vector w, where two training tuples are wrongly classified, including a positive tuple x_1 and a negative tuple x_8. Therefore only these two tuples are used to update the weight vector in the current iteration, that is, $w \leftarrow w + \eta x_1 - \eta x_8$. The updated weight vector w and the corresponding decision boundary are shown in Fig. 6.12(b) (right), where all training tuples are correctly classified. □

"How effective is the perceptron learning algorithm?" If the training tuples are linearly separable (e.g., the example in Fig. 6.12), the perceptron algorithm is guaranteed to find a weight vector (i.e., a hyperplane decision boundary) that perfectly separates all the positive training tuples from all the negative training tuples. However, if the training tuples are not linearly separable, this algorithm will fail to converge.



(a) current weight vector w (b) updated weight vector w

FIGURE 6.12

Training a perceptron classifier.

Perceptron, one of the earliest linear classifiers, was first invented back in 1958. It can also be used as a building block (called a "neuron") in deep neural networks that will be introduced in Chapter 10.

6.5.3 Logistic regression

Perceptron that we have just introduced in the previous section is capable of predicting the binary class label of a given tuple. However, can we also tell how confident such a prediction is? Again, let us consider a binary classification task, and we assume that there are two possible class labels, that is, $y = 1$ for a positive tuple and $y = 0$ for a negative tuple. Recall that in (naïve) Bayes classifier, we can estimate the posterior probability $P(y_i = 1|x_i)$, which can be directly used to indicate how confident the predicted classification result is. For example, if $P(y_i = 1|x_i)$ is close to 1, the classifier is highly confident that the tuple x_i is a positive example.

How can we make a linear classifier not only predict which class label a tuple has, but also tell how confident it is in making such a prediction? An effective way to this end is via **logistic regression** classifier. Let us first introduce an important function called **sigmoid** function, which is defined as $\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z}$. From Fig. 6.13, we can see that the sigmoid function maps a real number in $(-\infty, +\infty)$ (i.e., the x-axis of Fig. 6.13) to an output value in the range of $(0, 1)$ (i.e., the y-axis of Fig. 6.13). Therefore if we leverage the sigmoid function to map the output of a linear regression model to a number between 0 and 1, we can interpret the mapping result as the posterior probability of observing a positive class label. This is exactly what logistic regression classifier tries to do!

Formally, we have $P(\hat{y}_i = 1|x_i, w) = \sigma(w^T x_i) = \frac{1}{1+e^{-w^T x_i}}$, where $\hat{y}_i$ is the predicted class label for the tuple with attributes x_i, and w is the weight vector. Notice that we have absorbed the bias term b into the weight vector w by introducing a dummy attribute to simplify the notation, as we did in the multilinear regression model and in perceptron. Naturally, if $P(\hat{y}_i = 1|x_i, w) > 0.5$, the classifier predicts that the tuple x_i is a positive tuple (i.e., $\hat{y}_i = 1$), otherwise, it predicts a negative tuple (i.e., $\hat{y}_i = 0$). This (details are left as an exercise) is equivalent to the following linear classifier: predict $\hat{y}_i = 1$ (i.e., positive tuple) if $w^T x_i > 0$, and predict $\hat{y}_i = 0$ (i.e., negative tuple) if $w^T x_i < 0$. Therefore if we know the weight vector w, the classification task for a given tuple is quite simple. That is, we

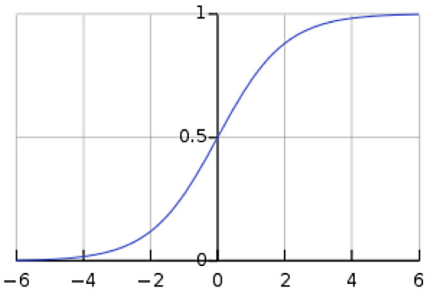


FIGURE 6.13

Illustration of sigmoid function. The sigmoid function "squashes" an input from a larger range $(-\infty, +\infty)$ to a smaller range $(0, 1)$. For this reason, sigmoid function is also called *squash function*. In Chapter 10, we will see other types of squash functions, which are called activation functions in the deep learning terminology.

only need to multiply the attribute vector x_i of the given tuple with the weight vector w, and then make a prediction based on the sign of $w^T x_i$. If $w^T x_i$ is a positive number, we predict that the given tuple is a positive tuple. Otherwise, we predict that it is a negative tuple.

"How can we determine the optimal weight vector w from a set of training tuples?" The classic method to train a logistic regression classifier (i.e., to determine the best weight vector w from the training set) is via *maximum likelihood estimation* (MLE). Again, let us assume there are n training tuples (x_i, y_i) $(i = 1, ..., n)$. Since we have a binary classification task, we can view the predicted class label $\hat{y}_i$ as a Bernoulli random variable, which can only take two possible values, including $P(\hat{y}_i = 1|x_i, w) = p_i$ and $P(\hat{y}_i = 0|x_i, w) = 1 - p_i$, where $p_i = \sigma(w^T x_i) = \frac{1}{1+e^{-w^T x_i}}$ is determined by the sigmoid function and it describes the probability of observing a positive outcome for the predicted class label (i.e., $\hat{y}_i = 1$). Notice that the true class label y_i for the ith tuple is a binary variable. Therefore we have that $P(\hat{y}_i = y_i) = p_i^{y_i}(1 - p_i)^{1-y_i}$. The maximum likelihood estimation method aims to solve the following optimization problem, which says that we should choose the best weight vector w that maximizes the likelihood of the training set. The intuition is that we want to find the optimal model parameter (i.e., the weight vector w) so that there is the highest "chance" (i.e., the likelihood or the probability) of observing the entire training set.

$$w^* = \text{argmax}_w \; L(w) = \Pi_{i=1}^n p_i^{y_i}(1 - p_i)^{1-y_i} = \Pi_{i=1}^n (\frac{e^{w^T x_i}}{1 + e^{w^T x_i}})^{y_i} (\frac{1}{1 + e^{w^T x_i}})^{1-y_i} \tag{6.19}$$

"But, how can we develop an algorithm to solve this optimization problem to find the optimal weight vector w?" First, we notice that the likelihood function $L(w)$ has many nonnegative terms that are multiplied with each other. In practice, it is often more convenient to work with the logarithm of such a complicated function. Thus we have the following equivalent optimization problem, where $l(w)$ is called the log likelihood

$$w^* = \text{argmax}_w \; l(w) = \sum_{i=1}^{n} y_i x_i^T w - \log(1 + e^{w^T x_i}). \tag{6.20}$$

From the optimization perspective, the good news is that the log likelihood function in Eq. (6.20) is a strictly concave function, and therefore its maximum (the optimal solution) uniquely exists. However, the bad news is that the closed-form solution for the above optimization problem does not exist. In this case, a common strategy is to find the optimal solution w^* iteratively as follows. In each iteration, we try to improve the current weight vector w so that the objective function we wish to maximize (the log likelihood function $l(w)$) is improved most. In order to increase the current objective function $l(w)$ most, it turns out the best direction to update the current estimation of the weight vector w is to follow its *gradient*. This leads to the following algorithm to learn the optimal weight vector w^* from the training set. We start with an initial guess of the weight vector w (e.g., we can simply set $w = 0$). Then, the learning algorithm will iterate until it converges, or the maximum iteration number or some other preset stopping criteria are met. In each iteration, it updates the weight vector w as follows $w \leftarrow w + \eta \sum_{i=1}^n (y_i - P(\hat{y}_i = 1|x_i, w))x_i$, where $\eta > 0$ is the user-specified learning rate.

"So, what is the intuition of the above algorithm?" Let us analyze the impact of each training tuple (x_i, y_i) on updating the estimation of the weight vector w. We consider two situations depending on whether it is a positive tuple (i.e., $y_i = 1$) or a negative tuple (i.e., $y_i = 0$). For the former, the

impact of the given tuple on updating the weight vector w can be calculated as $w \leftarrow w + \eta(1 - P(\hat{y}_i = 1|x_i, w))x_i$. The intuition is that we want update the current weight vector w *towards* the direction of the attribute vector x_i of this positive tuple. For the latter case (i.e., $y_i = 0$), the impact of the given tuple on updating the weight vector w can be calculated as $w \leftarrow w - \eta P(\hat{y}_i = 1|x_i, w)x_i$. The intuition is that we want update the current weight vector w *away from* the direction of the attribute vector x_i of this negative tuple. From this perspective, the learning algorithm for training a logistic regression classifier bears some similarities to the perceptron algorithm. That is, both algorithms try to update the current weight vector w so that is (1) more aligned with the attribute vectors of positive tuples and (2) more mis-aligned with (i.e., towards the opposite direction of) the attribute vectors of negative tuples.

However, the two algorithms (perceptron vs. logistic regression) differ regarding to what extent the algorithms update the weight vector w. In perceptron, it uses a *fixed* learning rate η for all wrongly predicted tuples by the current weight vector w. On the other hand, in logistic regression, it depends on the learning rate η as well as $P(\hat{y}_i = 1|x_i, w)$ (i.e., the probability that the given tuple belongs to the positive class based on the current weight vector w). This makes the logistic regression algorithm *adaptive* in the following sense. For example, if $P(\hat{y}_i = 1|x_i, w)$ is high for a positive tuple, it means that the prediction by the current weight vector w for this positive tuple is not only correct (i.e., $P(\hat{y}_i = 1|x_i, w) > 0.5$), but also quite confident (i.e., $P(\hat{y}_i = 1|x_i, w)$ is close to 1). Then, the impact of this positive tuple (i.e., $\eta(1 - P(\hat{y}_i = 1|x_i, w))$) on updating the weight vector is relatively small. On the other hand, if $P(\hat{y}_i = 1|x_i, w)$ is high for a negative tuple, it means that the prediction by the current weight vector w for this negative tuple is either wrong (i.e., $P(\hat{y}_i = 1|x_i, w) > 0.5$), or correct but with low confidence (i.e., $P(\hat{y}_i = 1|x_i, w)$ is barely below 0.5). Then, the impact of this negative tuple (i.e., $\eta P(\hat{y}_i = 1|x_i, w)$) on updating the weight vector will be relatively large. In other words, the logistic regression learning algorithm pays more attention to those "hard" training tuples, which are either wrongly predicted or correctly predicted with a low confidence by the current weight vector w. Recall that for the example in Fig. 6.12(a), perceptron only uses x_1 and x_8 to update the current weight vector w since these two tuples are wrongly classified by the current w. In contrast, logistic regression uses *all* training tuples to update the weight vector w. Among them, x_1 and x_8 have the highest impact on updating w since they are both wrongly classified by the current classifier; x_2, x_3, x_5, x_9 and x_{10} have the least impact since they are all correctly classified by the current weight vector w with a high confidence; x_4, x_6 and x_7 have the moderate impact since they are correctly classified but with a relatively low confidence.

"How good is the logistic regression algorithm? What are the potential limitations and how to mitigate?" Since the log likelihood function $l(w)$ is a concave function, the algorithm for training a logistic regression classifier described above is guaranteed to converge to its optimal solution. However, if the training set is linearly separable, the algorithm might converge to a weight vector w with an infinitely large norm. (See an illustrative example in Fig. 6.14.) A "large" weight vector w could make the trained classifier prone to the noise of certain attributes of a given tuple. This will, in turn, lead to a poor generalization performance of the learned logistic regression classifier. In other words, the learned logistic regression classifier *overfits* the training set. An effective way to mitigate the overfitting is to introduce a regularization term $\|w\|_2^2$ into the objective function $l(w)$ to prevent the learned weight vector from

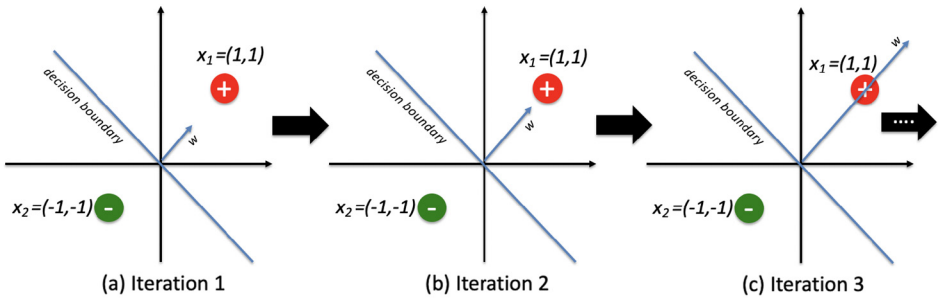


FIGURE 6.14

Illustration of the infinitely large weight vector of logistic regression in linearly separable case. There are two training tuples in 2d space, with one positive training tuple $x_1 = (1, 1)$ and one negative training tuple $x_2 = (-1, -1)$. For simplicity, we let the bias scalar $b = 0$ and the learning rate $\eta = 1$. Suppose that at iteration 1, the weight vector $w = (1, 1)$. Then, logistic regression algorithm introduced above will update the weight vector as $w_{\text{new}} = w_{\text{old}} + (1 - P(\hat{y}_1 = 1|x_1, w_{\text{old}}))x_1 - P(\hat{y}_2 = 1|x_2, w_{\text{old}})x_2 = (0.5, 0.5) + a(1, 1) = (1 + 2a)w_{\text{old}}$, where $a = 1 - P(\hat{y}_1 = 1|x_1, w_{\text{old}}) + P(\hat{y}_2 = 1|x_2, w_{\text{old}}) > 0$. As such, the new weight vector w_{new} shares the same direction as the old w_{old}. Therefore, the decision boundary remains the same, but new weight vector w_{new} grows in the magnitude by a factor of $(1 + 2a)$. This trend will continue as the logistic regression algorithm progresses, leading to a weight vector with an infinitely large magnitude.

becoming "too large."[11] The second potential limitation is the *independence* assumption behind logistic regression. Recall that when we calculate the likelihood $L(w)$ of the training set, we simply multiply the likelihood of each training tuple together (Eq. (6.19)). This means that we have implicitly assumed that different training tuples are independent of each other. However, this assumption might be violated in some applications (e.g., users on a social network are interconnected with each other). The graph-based classification might provide a natural remedy for this issue. The third potential limitation lies in the computational challenge. Notice that in the updating rule $w \leftarrow w + \eta \sum_{i=1}^{n}(y_i - P(\hat{y}_i = 1|x_i, w))x_i$ described above, we need to calculate the gradients $(y_i - P(\hat{y}_i = 1|x_i, w))$ for *all* training tuples and then sum them up to update the weight vector w. If there are millions of training tuples, it is computationally very expensive to perform such computation. An efficient way to address this issue is to use *stochastic gradient descent* method to train a logistic regression classifier. That is, at each iteration, we will randomly sample a small subset of training tuples (this is often referred to as a *minibatch*) and *only* use the sampled tuples (instead of all training tuples) to update the weight vector. It is worth pointing out that the stochastic gradient descent is extensively used in many other data mining algorithms, such as deep learning methods, which will be introduced in Chapter 10.

[11] From the statistical parameter estimation perspective, we are switching from the maximum likelihood estimation (MLE) to maximum a posterior estimation (MAP). Adding a regularization term $\|w\|_2^2$ into $l(w)$ is equivalent to imposing a Gaussian prior with the mean vector at the origin for the weight vector w.

6.6 Model evaluation and selection

Now that you may have built a classification model, there may be many questions going through your mind. For example, suppose you have used data from previous sales to build a classifier to predict customer purchasing behavior. You would like an estimate of how accurately the classifier can predict the purchasing behavior of future customers, that is, future customer data on which the classifier has not been trained. You may even have tried different methods to build more than one classifier and now wish to compare their accuracy. But what is accuracy? How can we estimate it? Are some measures of a classifier's accuracy more appropriate than others? How can we obtain a *reliable* accuracy estimate? These questions are addressed in this section.

Section 6.6.1 describes various evaluation metrics for the predictive accuracy of a classifier. Based on randomly sampled partitions of the given data, holdout and random subsampling (Section 6.6.2), cross-validation (Section 6.6.3), and bootstrap methods (Section 6.6.4) are common techniques for assessing accuracy. What if we have more than one classifier and want to choose the "best" one? This is referred to as **model selection** (i.e., choosing one classifier over another). The last two sections address this issue. Section 6.6.5 discusses how to use tests of statistical significance to assess whether the difference in accuracy between two classifiers is due to chance. Section 6.6.6 presents how to compare classifiers based on cost–benefit and receiver operating characteristic (ROC) curves.

6.6.1 Metrics for evaluating classifier performance

This section presents measures for assessing how good or how "accurate" your classifier is at predicting the class label of tuples. We will consider the case where the class tuples are more or less evenly distributed, as well as the case where classes are unbalanced (e.g., where an important class of interest is rare such as in medical tests). The classifier evaluation measures presented in this section are summarized in Fig. 6.15. They include accuracy (also known as recognition rate), sensitivity (or recall), specificity, precision, F_1, and F_β. Note that although accuracy is a specific measure, the word "accuracy" is also used as a general term to refer to a classifier's predictive abilities.

Using training data to derive a classifier and then estimate the accuracy of the learned model can result in misleading overoptimistic estimates due to overspecialization of the learning algorithm to the data. (We will say more on this in a moment!) Instead, it is better to measure the classifier's accuracy on a *test set* consisting of class-labeled tuples that were not used to train the model.

Before we discuss the various measures, we need to become comfortable with some terminology. Recall that we can talk in terms of **positive tuples** (tuples of the main class of interest) and **negative tuples** (all other tuples).[12] Given two classes, for example, the positive tuples may be *buys_computer* = *yes* while the negative tuples are *buys_computer* = *no*. Suppose we use our classifier on a test set of labeled tuples. P is the number of positive tuples, and N is the number of negative tuples. For each tuple, we compare the classifier's class label prediction with the tuple's known class label.

There are four additional terms we need to know that are the "building blocks" used in computing various evaluation measures. Understanding them will make it easy to grasp the meaning of the various measures.

[12] In the machine learning and pattern recognition literature, these are referred to as *positive samples* and *negative samples*, respectively.

Measure	Formula
accuracy, recognition rate	$\frac{TP+TN}{P+N}$
error rate, misclassification rate	$\frac{FP+FN}{P+N}$
sensitivity, true positive rate, recall	$\frac{TP}{P}$
specificity, true negative rate	$\frac{TN}{N}$
precision	$\frac{TP}{TP+FP}$
F, F_1, F-score, harmonic mean of precision and recall	$\frac{2 \times precision \times recall}{precision+recall}$
F_β, where β is a nonnegative real number	$\frac{(1+\beta^2) \times precision \times recall}{\beta^2 \times precision+recall}$

FIGURE 6.15

Evaluation measures. Note that some measures are known by more than one name. TP, TN, FP, FN, P, N refer to the number of true positive, true negative, false positive, false negative, positive, and negative samples, respectively (see text).

		Predicted class		
		yes	*no*	Total
Actual class	*yes*	TP	FN	P
	no	FP	TN	N
	Total	P'	N'	$P+N$

FIGURE 6.16

Confusion matrix, shown with totals for positive and negative tuples.

- **True positives** (TP): These refer to the positive tuples that were correctly labeled by the classifier. Let TP be the number of true positives.
- **True negatives** (TN): These are the negative tuples that were correctly labeled by the classifier. Let TN be the number of true negatives.
- **False positives** (FP): These are the negative tuples that were incorrectly labeled as positive (e.g., tuples of class *buys_computer* = *no* for which the classifier predicted *buys_computer* = *yes*). Let FP be the number of false positives.
- **False negatives** (FN): These are the positive tuples that were mislabeled as negative (e.g., tuples of class *buys_computer* = *yes* for which the classifier predicted *buys_computer* = *no*). Let FN be the number of false negatives.

These terms are summarized in the **confusion matrix** of Fig. 6.16.

A confusion matrix is a useful tool for analyzing how well your classifier can recognize tuples of different classes. TP and TN tell us when the classifier is getting things right, whereas FP and FN tell us when the classifier is getting things wrong (i.e., mislabeling). Given m classes (where $m \geq 2$), a **confusion matrix** is a table of at least size m by m. An entry, $CM_{i,j}$ at the ith row and the jth column indicates the number of tuples of class i that were labeled by the classifier as class j. For a classifier to have good accuracy, ideally most of the tuples would be represented along the diagonal of the confusion matrix, from entry $CM_{1,1}$ to entry $CM_{m,m}$, with the rest of the entries being zero or close to zero. That is, ideally, FP and FN are around zero.

Classes	*buys_computer = yes*	*buys_computer = no*	Total	Recognition (%)
buys_computer = yes	**6954**	**46**	7000	99.34
buys_computer = no	**412**	**2588**	3000	86.27
Total	7366	2634	10,000	95.42

FIGURE 6.17

Confusion matrix for the classes *buys_computer = yes* and *buys_computer = no,* where an entry in row i and column j shows the number of tuples of class i that were labeled by the classifier as class j. Ideally, the nondiagonal entries should be zero or close to zero.

The table may have additional rows or columns to provide totals. For example, in the confusion matrix of Fig. 6.16, P and N are shown. In addition, P' is the number of tuples that were labeled as positive ($TP + FP$), and N' is the number of tuples that were labeled as negative ($TN + FN$). The total number of tuples is $TP + TN + FP + TN$, or $P + N$, or $P' + N'$. Note that although the confusion matrix shown is for a binary classification problem, confusion matrices can be easily drawn for multiple classes in a similar manner.

Now let's look at the evaluation measures, starting with accuracy. The **accuracy** of a classifier on a given test set is the percentage of test set tuples that are correctly classified by the classifier. That is,

$$accuracy = \frac{TP + TN}{P + N}. \tag{6.21}$$

In the pattern recognition literature, this is also referred to as the overall **recognition rate** of the classifier; that is, it reflects how well the classifier recognizes tuples of the various classes. An example of a confusion matrix for the two classes *buys_computer = yes* (positive) and *buys_computer = no* (negative) is given in Fig. 6.17. Totals are shown, as well as the recognition rates per class and overall. By glancing at a confusion matrix, it is easy to see if the corresponding classifier is confusing two classes.

For example, we see that it mislabeled 412 *"no"* tuples as *"yes."* Accuracy is most effective when the class distribution is relatively balanced.

We can also speak of the **error rate** or **misclassification rate** of a classifier, M, which is simply $1 - accuracy(M)$, where $accuracy(M)$ is the accuracy of M. This also can be computed as

$$error\ rate = \frac{FP + FN}{P + N}. \tag{6.22}$$

If we were to use the training set (instead of a test set) to estimate the error rate of a model, this quantity is known as the **resubstitution error**.[13] This error estimate is optimistic of the true error rate (and similarly, the corresponding accuracy estimate is optimistic) because the model is not tested on any samples that it has not already seen.

We now consider the **class imbalance problem**, where the main class of interest is rare. That is, the data set distribution reflects a significant majority of the negative class and a minority positive class. For example, in fraud detection applications, the class of interest (or positive class) is *"fraud"* which occurs much less frequently than the negative *"nonfraudulant"* class. In medical data, there may

[13] In machine learning literature, it is often referred to as the training error.

Classes	*yes*	*no*	Total	Recognition (%)
yes	**90**	**210**	300	30.00
no	**140**	**9560**	9700	98.56
Total	230	9770	10,000	96.40

FIGURE 6.18

Confusion matrix for the classes *cancer* = *yes* and *cancer* = *no*.

be a rare class, such as *"cancer."* Suppose that you have trained a classifier to classify medical data tuples, where the class label attribute is *"cancer"* and the possible class values are *"yes"* and *"no."* An accuracy rate of, say, 97% may make the classifier seem quite accurate, but what if only, say, 3% of the training tuples are actually cancer? Clearly, an accuracy rate of 97% may not be acceptable—the classifier could be correctly labeling only the noncancer tuples, for instance, and misclassifying all the cancer tuples. Instead, we need other measures, which assess how well the classifier can recognize the positive tuples (*cancer* = *yes*) and how well it can recognize the negative tuples (*cancer* = *no*).

The **sensitivity** and **specificity** measures can be used, respectively, for this purpose. Sensitivity is also referred to as the *true positive* (*recognition*) *rate* (i.e., the proportion of positive tuples that are correctly identified), whereas specificity is the *true negative rate* (i.e., the proportion of negative tuples that are correctly identified). These measures are defined as

$$sensitivity = \frac{TP}{P} \tag{6.23}$$

$$specificity = \frac{TN}{N}. \tag{6.24}$$

It can be shown that accuracy is a function of sensitivity and specificity:

$$accuracy = sensitivity\frac{P}{(P+N)} + specificity\frac{N}{(P+N)}. \tag{6.25}$$

Example 6.9. Sensitivity and specificity. Fig. 6.18 shows a confusion matrix for medical data where the class values are *yes* and *no* for a class label attribute, *cancer*. The sensitivity of the classifier is $\frac{90}{300} = 30.00\%$. The specificity is $\frac{9560}{9700} = 98.56\%$. The classifier's overall accuracy is $\frac{9650}{10,000} = 96.50\%$. Thus we note that although the classifier has a high accuracy, it's ability to correctly label the positive (rare) class is poor given its low sensitivity. It has high specificity, meaning that it can accurately recognize negative tuples. Techniques for handling class-imbalanced data are given in Section 6.7.5. □

The *precision* and *recall* measures are also widely used in classification. **Precision** can be thought of as a measure of *exactness* (i.e., what percentage of tuples labeled as positive are actually such), whereas **recall** is a measure of *completeness* (what percentage of positive tuples are labeled as such). If recall seems familiar, that's because it is the same as sensitivity (or the *true positive rate*). These measures can be computed as

$$precision = \frac{TP}{TP + FP} \tag{6.26}$$

$$recall = \frac{TP}{TP + FN} = \frac{TP}{P}. \tag{6.27}$$

Example 6.10. Precision and recall. The precision of the classifier in Fig. 6.18 for the *yes* class is $\frac{90}{230} = 39.13\%$. The recall is $\frac{90}{300} = 30.00\%$, which is the same calculation for sensitivity in Example 6.9. □

A perfect precision score of 1.0 for a class C means that every tuple that the classifier labeled as belonging to class C does indeed belong to class C. However, it does not tell us anything about the number of class C tuples that the classifier mislabeled. A perfect recall score of 1.0 for C means that every item from class C was labeled as such, but it does not tell us how many other tuples were incorrectly labeled as belonging to class C. There tends to be an inverse relationship between precision and recall, where it is possible to increase one at the cost of reducing the other. For example, our medical classifier may achieve high precision by labeling all cancer tuples that present a certain way as *cancer* but may have low recall if it mislabels many other instances of *cancer* tuples. Precision and recall scores are typically used together, where precision values are compared for a fixed value of recall, or vice versa. For example, we may compare precision values at a recall value of, say, 0.75.

An alternative way to use precision and recall is to combine them into a single measure. This is the approach of the F measure (also known as the F_1 score or F-score) and the F_β measure. They are defined as

$$F = \frac{2 \times precision \times recall}{precision + recall} \tag{6.28}$$

$$F_\beta = \frac{(1 + \beta^2) \times precision \times recall}{\beta^2 \times precision + recall}, \tag{6.29}$$

where β is a nonnegative real number. The F measure is the *harmonic mean* of precision and recall (the proof of which is left as an exercise). It gives equal weights to precision and recall. The F_β measure is a weighted measure of precision and recall. It assigns β times as much weight to recall as to precision. Commonly used F_β measures are F_2 (which weights recall twice as much as precision) and $F_{0.5}$ (which weights precision twice as much as recall).

"Are there other cases where accuracy may not be appropriate?" In classification problems, it is commonly assumed that all tuples are uniquely classifiable, that is, each training tuple can belong to only one class. Yet, owing to the wide diversity of data in large databases, it is not always reasonable to assume that all tuples are uniquely classifiable. Rather, it is more probable to assume that each tuple may belong to more than one class. How then can the accuracy of classifiers on large databases be measured? The accuracy measure is not appropriate, because it does not take into account the possibility of tuples belonging to more than one class.

Rather than returning a class label, it is useful to return a class probability distribution. Accuracy measures may then use a **second guess** heuristic, whereby a class prediction is judged as correct if it agrees with the first or second most probable class. Although this does take into consideration, to some degree, the nonunique classification of tuples, it is not a complete solution.

In addition to accuracy-based measures, classifiers can also be compared with respect to the following additional aspects:

- **Speed:** This refers to the computational cost involved in generating and using the given classifier.

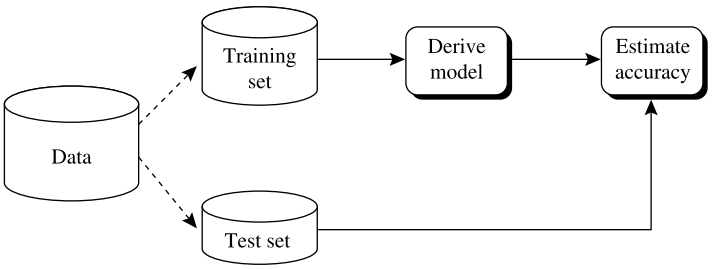


FIGURE 6.19

Estimating accuracy with the holdout method.

- **Robustness:** This is the ability of the classifier to make correct predictions given noisy data or data with missing values. Robustness is typically assessed with a series of synthetic data sets representing increasing degrees of noise and missing values.
- **Scalability:** This refers to the ability to construct the classifier efficiently given large amounts of data. Scalability is typically assessed with a series of data sets of increasing size.
- **Interpretability:** This refers to the level of understanding and insight that is provided by the classifier or predictor. Interpretability could be subjective and therefore more difficult to assess. Decision trees and classification rules can be easy to interpret, yet their interpretability may diminish the more they become complex. We will introduce some basic techniques to improve the interpretability of classification models in Chapter 7.

In summary, we have presented several evaluation measures. The accuracy measure works best when the data classes are fairly evenly distributed. Other measures, such as sensitivity (or recall), specificity, precision, F, and F_β, are better suited to the class imbalance problem, where the main class of interest is rare. The remaining subsections focus on obtaining reliable classifier accuracy estimates.

6.6.2 Holdout method and random subsampling

The **holdout** method is what we have alluded to so far in our discussions about accuracy. In this method, the given data are randomly partitioned into two independent sets, a *training set* and a *test set*. Typically, two-thirds of the data are allocated to the training set, and the remaining one-third is allocated to the test set. The training set is used to derive the model. The model's accuracy is then estimated with the test set (Fig. 6.19). The estimate is pessimistic because only a portion of the initial data is used to derive the model.

Random subsampling is a variation of the holdout method in which the holdout method is repeated k times. The overall accuracy estimate is taken as the average of the accuracies obtained from each iteration.

6.6.3 Cross-validation

In ***k*-fold cross-validation**, the initial data are randomly partitioned into k mutually exclusive subsets or "folds" $D_1, D_2, \ldots, D_k$, each of approximately equal size. Training and testing are performed k

times. In iteration i, partition D_i is reserved as the test set, and the remaining partitions are collectively used to train the model. That is, in the first iteration, subsets $D_2, \ldots, D_k$ collectively serve as the training set to obtain the first model, which is tested on D_1; the second iteration is trained on subsets $D_1, D_3, \ldots, D_k$ and tested on D_2; and so on. Unlike the holdout and random subsampling methods, here each sample is used the same number of times for training and once for testing. For classification, the accuracy estimate is the overall number of correct classifications from the k iterations, divided by the total number of tuples in the initial data.

Leave-one-out-cross-validation is a special case of k-fold cross-validation where k is set to the number of initial tuples. That is, only one sample is "left out" at a time for the test set. Leave-one-out-cross-validation is often used when the initial data set is small. In **stratified cross-validation**, the folds are stratified so that the class distribution of the tuples in each fold is approximately the same as that in the initial data.

In practice, stratified 10-fold cross-validation is recommended for estimating accuracy (even if computation power allows using more folds) due to its relatively low bias and variance.

6.6.4 Bootstrap

Unlike the accuracy estimation methods just mentioned, the **bootstrap method** samples the given training tuples uniformly *with replacement*. That is, each time a tuple is selected, it is equally likely to be selected again and re-added to the training set. For instance, imagine a machine that randomly selects tuples for our training set. In *sampling with replacement*, the machine is allowed to select the same tuple more than once.

There are several bootstrap methods. A commonly used one is the **.632 bootstrap**, which works as follows. Suppose we are given a data set of d tuples. The data set is sampled d times, with replacement, resulting in a *bootstrap sample* or training set of d samples. Some of the original data tuples will likely occur more than once in this sample. The data tuples that did not make it into the training set end up forming the test set. Suppose we were to try this out several times. As it turns out, on average, 63.2% of the original data tuples will end up in the bootstrap sample, and the remaining 36.8% will form the test set (hence, the name, .632 bootstrap).

"Where does the figure, 63.2%, come from?" Each tuple has a probability of $1/d$ of being selected, so the probability of not being chosen is $(1 - 1/d)$. We have to select d times, so the probability that a tuple will not be chosen during this whole time is $(1 - 1/d)^d$. If d is large, the probability approaches $e^{-1} = 0.368$.[14] Thus 36.8% of tuples will not be selected for training and thereby end up in the test set, and the remaining 63.2% will form the training set.

We can repeat the sampling procedure k times, wherein each iteration, we use the current test set to obtain an estimated accuracy of the model obtained from the current bootstrap sample. The overall accuracy of the model, M, is then estimated as

$$Acc(M) = \frac{1}{k} \sum_{i=1}^{k} (0.632 \times Acc(M_i)_{test\_set} + 0.368 \times Acc(M_i)_{train\_set}), \tag{6.30}$$

[14] e is the base of natural logarithms, that is, $e = 2.718$.

where $Acc(M_i)_{test\_set}$ is the accuracy of the model obtained with bootstrap sample i when it is applied to test set i. $Acc(M_i)_{train\_set}$ is the accuracy of the model obtained with bootstrap sample i when it is applied to the original set of data tuples. Bootstrapping tends to be overly optimistic. It works best with small data sets.

6.6.5 Model selection using statistical tests of significance

Suppose that we have generated two classification models, M_1 and M_2, from our data. We have performed 10-fold cross-validation to obtain a mean error rate[15] for each. How can we determine which model is best? It may seem intuitive to select the model with the lowest error rate; however, the mean error rates are just *estimates* of the error on the true population of future data cases. There can be considerable variance between error rates within any given 10-fold cross-validation experiment. Although the mean error rates obtained for M_1 and M_2 may appear different, that difference may not be statistically significant. What if any difference between the two may just be attributed to chance? This section addresses these questions.

To determine if there is any "real" difference in the mean error rates of two models, we need to employ a *test of statistical significance*. In addition, we want to obtain some confidence limits for our mean error rates so that we can make statements like, *"Any observed mean will not vary by ± two standard errors 95% of the time for future samples"* or *"One model is better than the other by a margin of error of ± 4%."*

What do we need to perform the statistical test? Suppose that for each model, we did 10-fold cross-validation, say, 10 times, each time using a different 10-fold data partitioning. Each partitioning is independently drawn. We can average the 10 error rates obtained each for M_1 and M_2, respectively, to obtain the mean error rate for each model. For a given model, the individual error rates calculated in the cross-validations may be considered different, independent samples from a probability distribution. In general, they follow a *t-distribution with* $k - 1$ *degrees of freedom* where, here, $k = 10$. (This distribution looks very similar to a normal, or Gaussian, distribution even though the functions defining the two are quite different. Both are unimodal, symmetric, and bell-shaped.) This allows us to do hypothesis testing where the significance test used is the ***t*-test**, or **Student's *t*-test**. Our hypothesis is that the two models are the same, or in other words, that the difference in mean error rate between the two is zero. If we can reject this hypothesis (referred to as the *null hypothesis*), then we can conclude that the difference between the two models is statistically significant, in which case we can select the model with the lower error rate.

In data mining practice, we may often employ a single test set, that is, the same test set can be used for both M_1 and M_2. In such cases, we do a **pairwise comparison** of the two models *for each* 10-fold cross-validation round. That is, for the ith round of 10-fold cross-validation, the same cross-validation partitioning is used to obtain an error rate for M_1 and M_2. Let $err(M_1)_i$ (or $err(M_2)_i$) be the error rate of model M_1 (or M_2) on round i. The error rates for M_1 are averaged to obtain a mean error rate for M_1, denoted $\overline{err}(M_1)$. Similarly, we can obtain $\overline{err}(M_2)$. The variance of the difference between the two models is denoted $var(M_1 - M_2)$. The t-test computes the *t-statistic with* $k - 1$ *degrees of freedom* for k samples. In our example, we have $k = 10$ since, here, the k samples are our error rates obtained

[15] Recall that the error rate of a model, M, is $1 - accuracy(M)$.

from ten 10-fold cross-validations for each model. The t-statistic for pairwise comparison is computed as follows:

$$t = \frac{\overline{err}(M_1) - \overline{err}(M_2)}{\sqrt{var(M_1 - M_2)/k}}, \tag{6.31}$$

where

$$var(M_1 - M_2) = \frac{1}{k}\sum_{i=1}^{k}[err(M_1)_i - err(M_2)_i - (\overline{err}(M_1) - \overline{err}(M_2))]^2. \tag{6.32}$$

To determine whether M_1 and M_2 are significantly different, we compute t and select a **significance level**, sig. In practice, a significance level of 5% or 1% is typically used. We then consult a table for the *t-distribution*, available in standard textbooks on statistics. This table is usually shown arranged by degrees of freedom as rows and significance levels as columns. Suppose we want to ascertain whether the difference between M_1 and M_2 is significantly different for 95% of the population, that is, $sig = 5\%$ or 0.05. We need to find the t-distribution value corresponding to $k - 1$ degrees of freedom (or 9 degrees of freedom for our example) from the table. However, because the t-distribution is symmetric, typically only the upper percentage points of the distribution are shown. Therefore we look up the table value for $z = sig/2$, which, in this case, is 0.025, where z is also referred to as a **confidence limit**. If $t > z$ or $t < -z$, then our value of t lies in the rejection region, within the distribution's tails. This means that we can reject the null hypothesis that the means of M_1 and M_2 are the same and conclude that there is a statistically significant difference between the two models. Otherwise, if we cannot reject the null hypothesis, we conclude that any difference between M_1 and M_2 can be attributed to chance.

If two test sets are available instead of a single test set, then a nonpaired version of the t-test is used, where the variance between the means of the two models is estimated as

$$var(M_1 - M_2) = \frac{var(M_1)}{k_1} + \frac{var(M_2)}{k_2}, \tag{6.33}$$

and k_1 and k_2 are the number of cross-validation samples (in our case, 10-fold cross-validation rounds) used for M_1 and M_2, respectively. This is also known as the **two sample *t*-test**. When consulting the table of t-distribution, the number of degrees of freedom used is taken as the minimum number of degrees of the two models.

6.6.6 Comparing classifiers based on cost–benefit and ROC curves

The true positives, true negatives, false positives, and false negatives are also useful in assessing the **costs and benefits** (or risks and gains) associated with a classification model. The cost associated with a false negative (such as incorrectly predicting that a cancerous patient is not cancerous) is far greater than those of a false positive (incorrectly yet conservatively labeling a noncancerous patient as cancerous). In such cases, we can outweigh one type of error over another by assigning a different cost to each. These costs may consider the danger to the patient, financial costs of resulting therapies, and other hospital costs. Similarly, the benefits associated with a true positive decision may be different from those of a true negative. Up to now, to compute the classifier's accuracy, we have assumed equal costs and essentially divided the sum of true positives and true negatives by the total number of test tuples.

Alternatively, we can incorporate costs and benefits by computing the average cost (or benefit) per decision. Other applications involving cost–benefit analysis include loan application decisions and target marketing mailouts. For example, the cost of loaning to a defaulter greatly exceeds that of the lost business incurred by denying a loan to a nondefaulter. Similarly, in an application that tries to identify households that are likely to respond to mailouts of certain promotional material, the cost of mailouts to numerous households that do not respond may outweigh the cost of lost business from not mailing to households that would have responded. Other costs to consider in the overall analysis include the costs to collect the data and to develop the classification tools.

Receiver operating characteristic curves are a useful visual tool for comparing two classification models. ROC curves come from signal detection theory that was developed during World War II for the analysis of radar images. A ROC curve for a given model shows the trade-off between the *true positive rate* (TPR) and the *false positive rate* (FPR).[16] Given a test set and a model, TPR is the proportion of positive (or "yes") tuples that are correctly labeled by the model; FPR is the proportion of negative (or "no") tuples that are mislabeled as positive. Recall that TP, FP, P, and N are the number of true positive, false positive, positive, and negative tuples, respectively. From Section 6.6.1, we know that $TPR = \frac{TP}{P}$, which is sensitivity. Furthermore, $FPR = \frac{FP}{N}$, which is $1 -$ specificity.

For a two-class problem, a ROC curve allows us to visualize the trade-off between the rate at which the model can accurately recognize positive cases vs. the rate at which it mistakenly identifies negative cases as positive for different portions of the test set. Any increase in TPR occurs at the cost of an increase in FPR. The area under the ROC curve is a measure of the accuracy of the model.

To plot a ROC curve for a given classification model, M, the model must be able to return a probability of the predicted class for each test tuple. With this information, we rank and sort the tuples so that the tuple that is most likely to belong to the positive or "yes" class appears at the top of the list, and the tuple that is least likely to belong to the positive class lands at the bottom of the list. Naïve Bayesian (Section 6.3) and logistic regression (Section 6.5) classifiers return a class probability distribution for each prediction and, therefore, are appropriate, although other classifiers, such as decision tree classifiers (Section 6.2), can easily be modified to return class probability predictions. Let the value that a probabilistic classifier returns for a given tuple $\boldsymbol{X}$ be $f(\boldsymbol{X}) \rightarrow [0, 1]$. For a binary problem, a threshold t is typically selected so that tuples where $f(\boldsymbol{X}) \geq t$ are considered positive and all the other tuples are considered negative. Note that the number of true positives and the number of false positives are both functions of t, so that we could write $TP(t)$ and $FP(t)$. Both are monotonic nonincreasing functions.

We first describe the general idea behind plotting a ROC curve and then follow up with an example. The vertical axis of a ROC curve represents TPR. The horizontal axis represents FPR. To plot a ROC curve for M, we begin as follows. Starting at the bottom left corner (where $TPR = FPR = 0$), we check the tuple's actual class label at the top of the list. If we have a true positive (i.e., a positive tuple that was correctly classified), then TP and thus TPR increase. On the graph, we move up and plot a point. If, instead, the model classifies a negative tuple as positive, we have a false positive, and so both FP and FPR increase. On the graph, we move right and plot a point. This process is repeated for each of the test tuples in ranked order, each time moving up on the graph for a true positive or toward the right for a false positive.

[16] TPR and FPR are the two operating characteristics being compared.

Tuple #	Class	Prob.	*TP*	*FP*	*TN*	*FN*	*TPR*	*FPR*
1	*P*	0.90	1	0	5	4	0.2	0
2	*P*	0.80	2	0	5	3	0.4	0
3	*N*	0.70	2	1	4	3	0.4	0.2
4	*P*	0.60	3	1	4	2	0.6	0.2
5	*P*	0.55	4	1	4	1	0.8	0.2
6	*N*	0.54	4	2	3	1	0.8	0.4
7	*N*	0.53	4	3	2	1	0.8	0.6
8	*N*	0.51	4	4	1	1	0.8	0.8
9	*P*	0.50	5	4	1	0	1.0	0.8
10	*N*	0.40	5	5	0	0	1.0	1.0

FIGURE 6.20

Tuples sorted by decreasing score, where the score is the value returned by a probabilistic classifier.

Example 6.11. Plotting a ROC curve. Fig. 6.20 shows the probability value (column 3) returned by a probabilistic classifier for each of the 10 tuples in a test set, sorted in the decreasing probability order. Column 1 is merely a tuple identification number, which aids in our explanation. Column 2 is the actual class label of the tuple. There are five positive tuples and five negative tuples; thus $P = 5$ and $N = 5$. As we examine the known class label of each tuple, we can determine the values of the remaining columns, TP, FP, TN, FN, TPR, and FPR. We start with tuple 1, which has the highest probability score, and take that score as our threshold, that is, $t = 0.9$. Thus the classifier considers tuple 1 to be positive, and all the other tuples are considered negative. Since the actual class label of tuple 1 is positive, we have a true positive, hence $TP = 1$ and $FP = 0$. Among the remaining nine tuples, which are all classified as negative, five actually are negative (thus, $TN = 5$). The remaining four are all actually positive; thus, $FN = 4$. We can therefore compute $TPR = \frac{TP}{P} = \frac{1}{5} = 0.2$, whereas $FPR = 0$. Thus we have the point $(0.2, 0)$ for the ROC curve.

Next, threshold t is set to 0.8, the probability value for tuple 2, so this tuple is now also considered positive, whereas tuples 3 through 10 are considered negative. The actual class label of tuple 2 is positive, thus now $TP = 2$. The rest of the row can easily be computed, resulting in the point $(0.4, 0)$. Next, we examine the class label of tuple 3 and let t be 0.7, the probability value returned by the classifier for that tuple. Thus tuple 3 is considered positive, yet its actual label is negative, and so it is a false positive. Thus TP stays the same and FP increments so that $FP = 1$. The rest of the values in the row can also be easily computed, yielding the point $(0.4, 0.2)$. The resulting ROC graph, from examining each tuple, is the jagged line shown in Fig. 6.21.

There are many methods to obtain a curve out of these points, the most common of which is to use a convex hull. The plot also shows a diagonal line where for every true positive of such a model, we are just as likely to encounter a false positive. For comparison, this line represents random guessing. □

Fig. 6.22 shows the ROC curves of two classification models. The diagonal line representing random guessing is also shown. Thus the closer the ROC curve of a model is to the diagonal line, the less accurate the model. If the model is really good, initially we are more likely to encounter true positives as we move down the ranked list. Thus the curve moves steeply up from zero. Later, as we start to encounter fewer and fewer true positives, and more and more false positives, the curve eases off and becomes more horizontal.

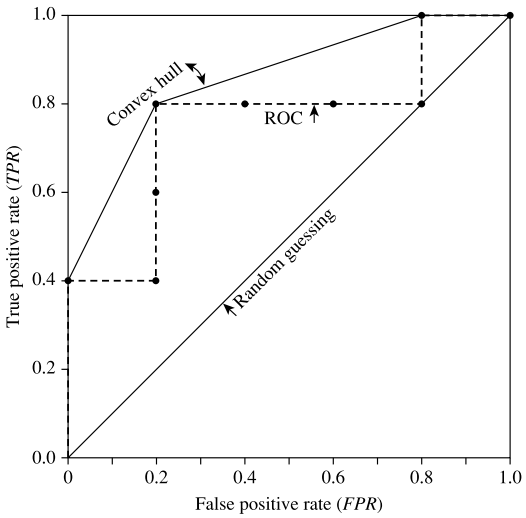


FIGURE 6.21

ROC curve for the data in Figure 6.20.

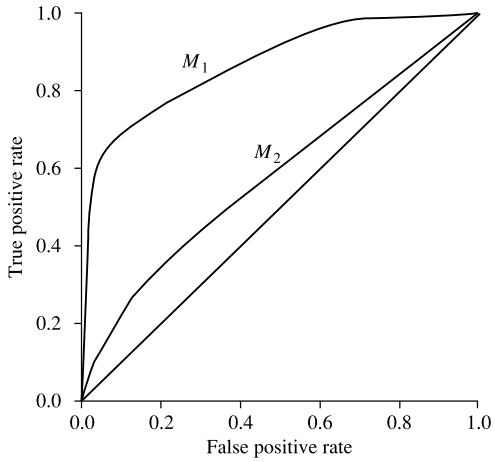


FIGURE 6.22

ROC curves of two classification models, M_1 and M_2. The diagonal shows where, for every true positive, we are equally likely to encounter a false positive. The closer a ROC curve is to the diagonal line, the less accurate the model is. Thus $M1$ is more accurate here.

To assess the accuracy of a model, we can measure the area under the curve (AUC). Several software packages are able to perform such calculation. The closer the area is to 0.5, the less accurate the corresponding model is. A model with perfect accuracy will have an AUC of 1.0.

6.7 Techniques to improve classification accuracy

In this section, you will learn some tricks for increasing classification accuracy. We focus on *ensemble methods*. An ensemble for classification is a composite model, made up of a combination of classifiers. The individual classifiers vote, and a class label prediction is returned by the ensemble based on the collection of votes. Ensembles tend to be more accurate than their component classifiers. We start off in Section 6.7.1 by introducing ensemble methods in general. Bagging (Section 6.7.2), boosting (Section 6.7.3), and random forests (Section 6.7.4) are popular ensemble methods.

Traditional learning models assume that the data classes are well distributed. In many real-world data domains, however, the data are class-imbalanced, where the main class of interest is represented by only a few tuples. This is known as the *class imbalance problem*. We also study techniques for improving the classification accuracy of class-imbalanced data. These are presented in Section 6.7.5.

6.7.1 Introducing ensemble methods

Bagging, *boosting*, and *random forests* are examples of **ensemble methods** (Fig. 6.23). An ensemble combines a series of k learned models (or *base classifiers*), $M_1, M_2, \ldots, M_k$, with the aim of creating an improved composite classification model, M^*. A given data set, D, is used to create k training sets, $D_1, D_2, \ldots, D_k$, where D_i $(1 \leq i \leq k)$ is used to generate classifier M_i. Given a new data tuple to classify, the base classifiers each vote by returning a class prediction. The ensemble returns a class prediction based on the votes of the base classifiers.

An ensemble tends to be more accurate than its base classifiers. For example, consider an ensemble that performs majority voting. That is, given a tuple $\boldsymbol{X}$ to classify, it collects the class label predictions returned from the base classifiers and outputs the class in the majority. The base classifiers may make mistakes, but the ensemble will misclassify $\boldsymbol{X}$ only if over half of the base classifiers are in error. Ensembles yield better results when there is significant diversity among the models. That is, ideally,

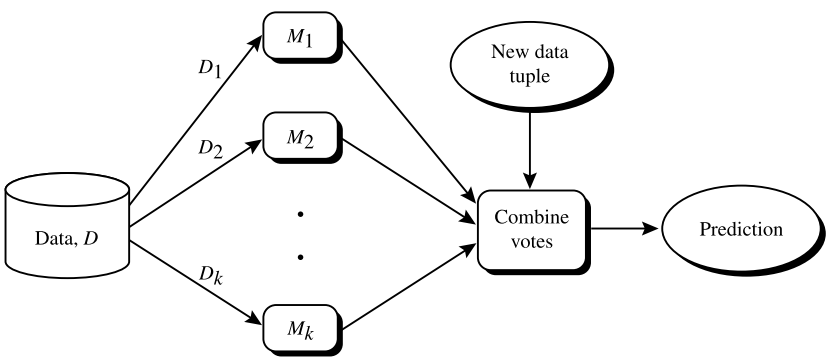


FIGURE 6.23

Increasing classifier accuracy. Ensemble methods generate a set of classification models, $M_1, M_2, \ldots, M_k$. Given a new data tuple to classify, each classifier "votes" for the class label of that tuple. The ensemble combines the votes to return a class prediction.

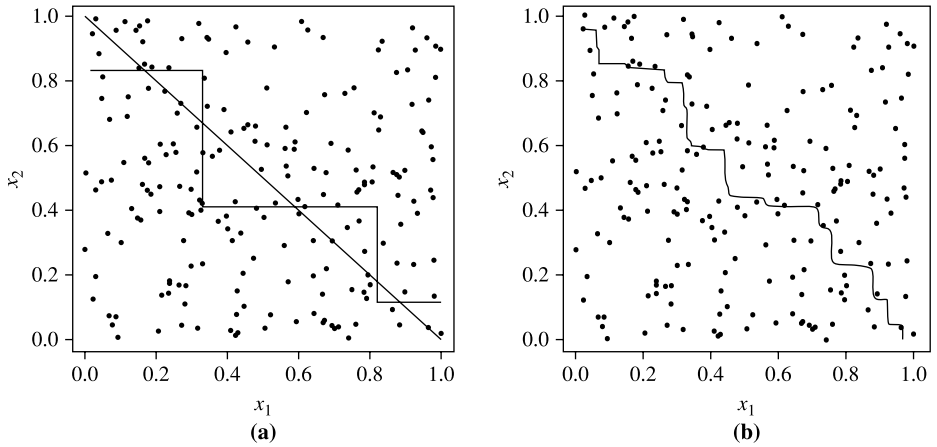


FIGURE 6.24

Decision boundary by (a) a single decision tree and (b) an ensemble of decision trees for a linearly separable problem (i.e., where the actual decision boundary is a straight line). The decision tree struggles with approximating a linear boundary. The decision boundary of the ensemble is closer to the true boundary. *Source:* From Seni and Elder [SE10]. © 2010 Morgan & Claypool Publishers; used with permission.

there is little correlation among classifiers. The base classifiers should also perform better than random guessing. Each base classifier can be allocated to a different CPU and so ensemble methods are parallelizable.

To help illustrate the power of an ensemble, consider a simple two-class problem described by two attributes, x_1 and x_2. The problem has a linear decision boundary. Fig. 6.24(a) shows the decision boundary of a decision tree classifier on the problem. Fig. 6.24(b) shows the decision boundary of an ensemble of decision tree classifiers on the same problem. Although the ensemble's decision boundary is still piecewise constant, it has a finer resolution and is better than that of a single tree.

6.7.2 Bagging

We now take an intuitive look at how bagging works as a method of increasing accuracy. Suppose that you are a patient and would like to have a diagnosis made based on your symptoms. Instead of asking one doctor, you may choose to ask several. If a certain diagnosis occurs more than any other, you may choose this as the final or best diagnosis. That is, the final diagnosis is made based on a majority vote, where each doctor gets an equal vote. Now replace each doctor by a classifier, and you have the basic idea behind bagging. Intuitively, a majority vote made by a large group of doctors may be more reliable than a majority vote made by a small group.

Given a set, D, of d tuples, **bagging** works as follows. For iteration i ($i = 1, 2, \ldots, k$), a training set, D_i, of d tuples is sampled with replacement from the original set of tuples, D. Note that the term *bagging* stands for *bootstrap aggregation*. Each training set is a bootstrap sample, as described in Section 6.6.4. Because sampling with replacement is used, some of the original tuples of D may not

Algorithm: Bagging. The bagging algorithm—create an ensemble of classification models for a learning scheme where each model gives an equally weighted prediction.

Input:

- D, a set of d training tuples;
- k, the number of models in the ensemble;
- a classification learning scheme (e.g., decision tree algorithm, naïve Bayesian, etc.).

Output: The ensemble—a composite model, M^*.
Method:

(1) **for** $i = 1$ to k **do** // create k models:
(2) create bootstrap sample, D_i, by sampling D with replacement;
(3) use D_i and the learning scheme to derive a model, M_i.
(4) **endfor**

To use the ensemble to classify a tuple, X:

let each of the k models classify $\boldsymbol{X}$ and return the majority vote;

FIGURE 6.25

Bagging.

be included in D_i, whereas others may occur more than once. A classifier model, M_i, is learned for each training set, D_i. To classify an unknown tuple, $\boldsymbol{X}$, each classifier, M_i, returns its class prediction, which counts as one vote. The bagged classifier, M^*, counts the votes and assigns the class with the most votes to $\boldsymbol{X}$. Bagging can be applied to the prediction of continuous values by taking the average value of each prediction for a given test tuple. The algorithm is summarized in Fig. 6.25.

The bagged classifier often has significantly greater accuracy than a single classifier derived from D, the original training data. It is often more robust to the effects of noisy data and overfitting. The increased accuracy occurs because the composite model reduces the variance of the individual classifiers.

6.7.3 Boosting

We now look at the ensemble method of boosting. As in the previous section, suppose that as a patient, you have certain symptoms. Instead of consulting one doctor, you choose to consult several. Suppose you assign weights to the value or worth of each doctor's diagnosis based on the accuracies of previous diagnoses they have made. The final diagnosis is then a combination of the weighted diagnoses. This is the essence behind boosting.

In **boosting**, weights are also assigned to each training tuple. A series of k classifiers is iteratively learned. After a classifier, M_i, is learned, the weights are updated to allow the subsequent classifier, M_{i+1}, to "pay more attention" to the training tuples that were misclassified by M_i. The final boosted classifier, M^*, combines the votes of each individual classifier, where the weight of each classifier's vote is a function of its accuracy.

AdaBoost (short for Adaptive Boosting) is a popular boosting algorithm. Suppose we want to boost the accuracy of a learning method. We are given D, a data set of d class-labeled tuples, $(\boldsymbol{X_1}, y_1), (\boldsymbol{X_2}, y_2), \ldots, (\boldsymbol{X_d}, y_d)$, where y_i is the class label of tuple $\boldsymbol{X_i}$. Initially, AdaBoost assigns each training tuple an equal weight of $1/d$. Generating k classifiers for the ensemble requires k rounds

Algorithm: AdaBoost. A boosting algorithm—create an ensemble of classifiers. Each one gives a weighted vote.

Input:

- D, a set of d class-labeled training tuples;
- k, the number of rounds (one classifier is generated per round);
- a classification learning scheme.

Output: A composite model.
Method:

(1) initialize the weight of each tuple in D to $1/d$;
(2) **for** $i = 1$ to k **do** // for each round:
(3) sample D with replacement according to the tuple weights to obtain D_i;
(4) use training set D_i to derive a model, M_i;
(5) compute $error(M_i)$, the error rate of M_i (Eq. (6.34))
(6) **if** $error(M_i) > 0.5$ **then**
(7) abort the loop;
(8) **endif**
(9) **for** each tuple in D that was correctly classified **do**
(10) multiply the weight of the tuple by $error(M_i)/(1 - error(M_i))$; // update weights
(11) normalize the weight of each tuple.
(12) **endfor**

To use the ensemble to classify tuple, X:

(1) initialize weight of each class to 0;
(2) **for** $i = 1$ to k **do** // for each classifier:
(3) $w_i = log \frac{1-error(M_i)}{error(M_i)}$; // weight of the classifier's vote
(4) $c = M_i(\boldsymbol{X})$; // get class prediction for $\boldsymbol{X}$ from M_i
(5) add w_i to the weight for class c
(6) **endfor**
(7) return the class with the largest weight.

FIGURE 6.26

AdaBoost, a boosting algorithm.

through the rest of the algorithm. We can sample to form any sized training set, not necessarily of size d. Sampling with replacement is used—the same tuple may be selected more than once. Each tuple's chance of being selected is based on its weight. A classifier model, M_i, is derived from the training tuples of D_i. Its error is then calculated using D as the test set. The weights of the tuples are then adjusted according to how they were classified.

If a tuple was incorrectly classified, its weight is increased. If a tuple was correctly classified, its weight is decreased. A tuple's weight reflects how difficult it is to classify—the higher the weight, the more often it has been misclassified. These weights will be used to generate the training samples for the classifier of the next round. The basic idea is that when we build a classifier, we want it to focus more on the misclassified tuples of the previous round. Some classifiers may be better at classifying some "difficult" tuples than others. In this way, we build a series of classifiers that complement each other. The algorithm is summarized in Fig. 6.26.

Now, let's look at some of the math that's involved in the algorithm. To compute the error rate of model M_i, we sum the weights of each of the tuples in D that M_i misclassified. That is,

$$error(M_i) = \sum_{j=1}^{d} w_j \times err(\boldsymbol{X}_j), \tag{6.34}$$

where $err(\boldsymbol{X}_j)$ is the misclassification error of tuple $\boldsymbol{X}_j$: If the tuple was misclassified, then $err(\boldsymbol{X}_j)$ is 1; otherwise, it is 0. If the performance of classifier M_i is so poor that its error exceeds 0.5, then we abandon it. Instead, we try again by generating a new D_i training set, from which we derive a new M_i.

The error rate of M_i affects how the weights of the training tuples are updated. If a tuple in the round i was correctly classified, its weight is multiplied by $error(M_i)/(1 - error(M_i))$. Once the weights of all the correctly classified tuples are updated, the weights for all tuples (including the misclassified ones) are normalized so that their sum remains the same as it was before. To normalize a weight, we multiply it by the sum of the old weights, divided by the sum of the new weights. As a result, the weights of misclassified tuples are increased, and the weights of correctly classified tuples are decreased, as described before.

"Once boosting is complete, how is the ensemble of classifiers used to predict the class label of a tuple, $\boldsymbol{X}$?" Unlike bagging, where each classifier was assigned an equal vote, boosting assigns a weight to each classifier's vote, based on how well the classifier performed. The lower a classifier's error rate, the more accurate it is, and therefore, the higher its weight for voting should be. The weight of classifier M_i's vote is

$$log \frac{1 - error(M_i)}{error(M_i)}. \tag{6.35}$$

For each class, c, we sum the weights of each classifier that assigned class c to $\boldsymbol{X}$. The class with the highest sum is the "winner" and is returned as the class prediction for tuple $\boldsymbol{X}$.

Gradient boosting is another powerful boosting technique, which can be used for classification, regression, and ranking. If we use a tree (e.g., decision tree for classification, regression tree for regression) as the base model (i.e., the weak learner), it is called **gradient tree boosting**, or **gradient boosted tree**. Fig. 6.27 presents the gradient tree boosting algorithm for the regression task. It works as follows.

Gradient tree boosting algorithm starts with a simple regression model $F(x)$ (line 1), which outputs a constant (i.e., the average output of all training tuples). Then, similar to Adaboost, it tries to find a new base model (i.e., weak leaner) $M_t(x)$ at each round (line 3). The newly constructed base model $M_t(x)$ is added to the regression model $F(x)$ (line 8). In other words, the composite regression model $F(x)$ consists of k *additive* base models $M_t(x)$ $(t = 1, ..., k)$. When we search for a new base model $M_t(x)$, all the previously constructed base models (i.e., $M_1(x), ...M_{t-1}(x)$) are kept unchanged.

In order to construct a new base model $M_t(x)$, we first compute the predicted output $\hat{y}_i$ of each training tuple by the current regression model $F(x)$ (line 4) and calculate the **negative gradient** r_i of the loss function with respect to the predicted output $\hat{y}_i$ (line 5). Then, we fit a regression tree model for the training set $\{(x_1, r_1), ..., (x_n, r_n)\}$, where the negative gradient r_i is treated as the targeted output value of the ith training tuple. Since the negative gradient r_i $(i = 1, ..., n)$ changes in different rounds, we end up with different base models $M_t(x)$ $(t = 1, ..., k)$.

"But, why do we use the negative gradients to construct the new base model?" Suppose the loss function $L(y_i, F(x_i)) = \frac{1}{2}(y_i - \hat{y}_i)^2$ (recall that we have used the similar loss function for the regression tree and the least square linear regression model). Then, we can show that the negative gradient

Algorithm: Gradient Tree Boosting for Regression.

Input:

- D, a set of n training tuples $\{(x_1, y_1), ..., (x_n, y_n)\}$, where x_i is the attribute vector of the ith training tuple and y_i is its true target output value;
- k, the number of rounds (one base regression model is generated per round);
- a differential loss function Loss $= \sum_{i=1}^{n} L(y_i, F(x_i))$.

Output: A composite regression model $F(x)$.
Method:

(1) initialize the regression model $F(x) = \frac{\sum_{i=1}^{n} y_i}{n}$;
(2) **for** $t = 1$ to k **do** // construct a new weak learner $M_t(x)$ for each round:
(3) **for** $i = 1$ to n //each training tuple:
(4) calculate $\hat{y}_i = F(x_i)$; //predicted value by the current model $F(x)$
(5) calculate the negative gradient $r_i = -\frac{\partial L(y_i, \hat{y}_i)}{\partial \hat{y}_i}$;
(6) **endfor**
(7) fit a regression tree model $M_t(x)$ for the training set $\{(x_1, r_1), ..., (x_n, r_n)\}$;
(8) update the composite regression model $F(x) \leftarrow F(x) + M_t(x)$.
(9) **endfor**

FIGURE 6.27

Gradient tree boosting for regression.

$r_i = y_i - \hat{y}_i$, which is the difference between the actual output value and predicted output value by the current regression model $F(x)$ (i.e., the residual). In other words, the negative gradient r_i reveals the "shortcoming" of the current regression model $F(x)$ (i.e., how far away the predicted output is from its actual output value). If we use other loss functions (e.g., the Huber loss in robust regression), the negative gradient is no longer equal to the residual $y_i - \hat{y}_i$, but still provides a good indicator in terms of the prediction quality of the current regression model $F(x)$ on the ith training tuple. For this reason, the negative gradients are also referred to as *pseudo residuals*. By fitting a regression tree model with respect to the negative gradients (i.e., where the "shortcoming" of the current regression model $F(x)$ is), the newly constructed base model, $M_t(x)$, is expected to dramatically improve the composite regression model $F(x)$.

In addition to the algorithm in Fig. 6.27, several alternative design choices for gradient tree boosting exist. For example, similar to Adaboost, we can learn a weight for each base model $M_t(x)$, and then the composite regression model $F(x)$ becomes the *weighted sum* of the k base models. In practice, it was found that shrinking the newly constructed base model helps improve the generalization performance of the composite model $F(x)$ (i.e. $F(x) \leftarrow F(x) + \eta M_t(x)$ in line 8, where $0 < \eta < 1$ is the shrinkage constant.). The number of leaf nodes T of the regression tree $M_t(x)$ plays an important role in the learning performance of the composite model $F(x)$. That is, $F(x)$ might underfit the training set if T is too small, but could overfit the training set with a large T. The typical choice for T is between 4 and 8. At a given round t, we could use a subsample of the entire training set to construct the base model $M_t(x)$. Gradient tree boosting equipped with such a subsampling strategy is referred to as *stochastic gradient (tree) boosting* and it was found to significantly improve the accuracy of the composite model $F(x)$. A highly scalable end-to-end gradient tree boosting system is called **XGBoost**, which is capable to handle a billion-scale training set. XGBoost has made a number of innovations for training gradient

tree boosting, including a new tree construction algorithm designed for sparse data, feature subsampling (as opposed to training tuple subsampling in stochastic gradient boosting), and a highly efficient cache-aware block structure. XGBoost has been successfully used by data scientists in many data mining challenges, often leading to top competitive results.

"How does boosting compare with bagging?" Because of the way boosting focuses on the misclassified tuples, it risks overfitting the resulting composite model to such data. Therefore sometimes the resulting "boosted" model may be less accurate than a single model derived from the same data. Bagging is less susceptible to model overfitting. While both can significantly improve accuracy in comparison to a single model, boosting tends to achieve greater accuracy.

6.7.4 Random forests

We now present another ensemble method called **random forests**. Imagine that each of the classifiers in the ensemble is a *decision tree* classifier so that the collection of classifiers is a "forest." The individual decision trees are generated using a random selection of attributes at each node to determine the split. More formally, each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. During classification, each tree votes, and the most popular class is returned.

Random forests can be built using bagging (Section 6.7.2) in tandem with random attribute selection. A training set, D, of d tuples is given. The general procedure to generate k decision trees for the ensemble is as follows. For each iteration, i ($i = 1, 2, \ldots, k$), a training set, D_i, of d tuples is sampled with replacement from D. That is, each D_i is a bootstrap sample of D (Section 6.6.4), so that some tuples may occur more than once in D_i, while others may be excluded. Let F be the number of attributes to be used to determine the split at each node, where F is much smaller than the number of available attributes. To construct a decision tree classifier, M_i, randomly select, at each node, F attributes as candidates for the split at the node. The CART methodology is used to grow the trees. The trees are grown to maximum size and are not pruned. Random forests formed this way, with *random input selection*, are called Forest-RI.

Another form of random forest, called Forest-RC, uses *random linear combinations* of the input attributes. Instead of randomly selecting a subset of the attributes, it creates new attributes (or features) that are a linear combination of the existing attributes. That is, an attribute is generated by specifying L, the number of original attributes to be combined. At a given node, L attributes are randomly selected and added together with coefficients that are uniform random numbers on $[-1, 1]$. F linear combinations are generated, and a search is made over these for the best split. This form of random forest is useful when there are only a few attributes available, so as to reduce the correlation between individual classifiers.

Random forests are comparable in accuracy to AdaBoost, yet are more robust to errors and outliers. The generalization error for a forest converges as long as the number of trees in the forest is large. Thus, overfitting is less likely to be a problem. The accuracy of a random forest depends on the strength of the individual classifiers and a measure of the dependence between them. The ideal is to maintain the strength of individual classifiers without increasing their correlation. Random forests are insensitive to the number of attributes selected for consideration at each split. Typically, up to $log_2 d + 1$ are chosen. (An interesting empirical observation was that using a single random input attribute may result in good accuracy that is often higher than when using several attributes.) Because random forests consider much

fewer attributes for each split, they are efficient on very large databases. They can be faster than either bagging or boosting. Random forests give internal estimates of variable importance.

6.7.5 Improving classification accuracy of class-imbalanced data

In this section, we revisit the *class imbalance problem*. In particular, we study approaches to improving the classification accuracy of class-imbalanced data.

Given two-class data, the data are class-imbalanced if the main class of interest (the positive class) is represented by only a few tuples, while the majority of tuples represent the negative class. For multiclass-imbalanced data, the data distribution of each class differs substantially where, again, the main class or classes of interest are rare. The class imbalance problem is closely related to cost-sensitive learning, wherein the costs of errors per class are not equal. In medical diagnosis, for example, it is much more costly to falsely diagnose a cancerous patient as healthy (a false negative) than to misdiagnose a healthy patient as having cancer (a false positive). A false negative error could lead to the loss of life and therefore is much more expensive than a false positive error. Other applications involving class-imbalanced data include fraud detection, the detection of oil spills from satellite radar images, and fault monitoring.

Traditional classification algorithms aim to minimize the number of errors made during classification. They assume that the costs of false positive and false negative errors are equal. By assuming a balanced distribution of classes and equal error costs, they are therefore not suitable for class-imbalanced data. Earlier parts of this chapter presented ways of addressing the class imbalance problem. Although the accuracy measure assumes that the cost of classes are equal, alternative evaluation metrics can be used that consider the different types of classifications. Section 6.6.1, for example, presented *sensitivity* or recall (the true positive rate) and *specificity* (the true negative rate), which help to assess how well a classifier can predict the class label of imbalanced data. Additional relevant measures discussed include F_1 and F_β. Section 6.6.6 showed how ROC curves plot *sensitivity* vs. $1 - specificity$ (i.e., the false positive rate). Such curves can provide insight when studying the performance of classifiers on class-imbalanced data.

In this section, we look at general approaches for *improving* the classification accuracy of class-imbalanced data. These approaches include (1) oversampling, (2) undersampling, (3) threshold moving, and (4) ensemble techniques. The first three do not involve any changes to the construction of the classification model. That is, oversampling and undersampling change the distribution of tuples in the training set; threshold moving affects how the model makes decisions when classifying new data. Ensemble methods follow the techniques described in Section 6.7.2 through Section 6.7.4. For ease of explanation, we describe these general approaches with respect to the two-class imbalanced data problem, where the higher-cost classes are rarer than the lower-cost classes.

Both oversampling and undersampling change the training data distribution so that the rare (positive) class is well represented. **Oversampling** works by resampling the positive tuples so that the resulting training set contains an equal number of positive and negative tuples. **Undersampling** works by decreasing the number of negative tuples. It randomly eliminates tuples from the majority (negative) class until there are an equal number of positive and negative tuples.

Example 6.12. Oversampling and undersampling. Suppose the original training set contains 100 positive and 1000 negative tuples. In oversampling, we replicate tuples of the rare class to form a new training set containing 1000 positive tuples and 1000 negative tuples. In undersampling, we randomly

eliminate negative tuples so that the new training set contains 100 positive tuples and 100 negative tuples. □

Several variations to oversampling and undersampling exist. They may vary, for instance, in how tuples are added or eliminated. For example, the SMOTE algorithm uses oversampling where synthetic tuples are added, which are "close to" the given positive tuples in tuple space.

The **threshold-moving** approach to the class imbalance problem does not involve any sampling. It applies to classifiers that, given an input tuple, return a continuous output value (just like in Section 6.6.6, where we discussed how to construct ROC curves). That is, for an input tuple, $\boldsymbol{X}$, such a classifier returns as output a mapping, $f(\boldsymbol{X}) \rightarrow [0, 1]$. Rather than manipulating the training tuples, this method returns a classification decision based on the output values. In the simplest approach, tuples for which $f(\boldsymbol{X}) \geq t$, for some threshold, t, are considered positive, while all other tuples are considered negative. Other approaches may involve manipulating the outputs by weighting. In general, threshold moving moves the threshold, t, so that the rare class tuples are easier to classify (and hence, there is less chance of costly false negative errors). Examples of such classifiers include naïve Bayesian classifiers (Section 6.3) and neural networks (Chapter 10). The threshold-moving method, although not as popular as over- and undersampling, is simple and has shown some success for the two-class-imbalanced data.

Ensemble methods (Section 6.7.2 through Section 6.7.4) have also been applied to the class imbalance problem. The individual classifiers making up the ensemble may include versions of the approaches described here, such as oversampling and threshold moving.

These methods work relatively well for the class imbalance problem on two-class tasks. Threshold-moving and ensemble methods were empirically observed to outperform oversampling and undersampling. Threshold moving works well even on extremely imbalanced data sets. The class imbalance problem on multiclass tasks is much more difficult where oversampling and threshold moving are less effective. Although threshold-moving and ensemble methods show promise, finding a solution for the multiclass imbalance problem remains an area of future work.

6.8 Summary

- **Classification** is a form of data analysis that extracts models describing data classes. A classifier, or classification model, predicts categorical labels (classes). **Numeric prediction** models continuous-valued functions. Classification and numeric prediction are the two major types of prediction problems.
- **Decision tree induction** is a top-down recursive tree induction algorithm, which uses an attribute selection measure to select the attribute tested for each nonleaf node in the tree. **ID3**, **C4.5**, and **CART** are examples of such algorithms using different attribute selection measures. **Tree pruning** algorithms attempt to improve accuracy by removing tree branches reflecting noise in the data.
- **Naïve Bayesian classification** is based on Bayes' theorem of the posterior probability. It assumes class-conditional independence—that the effect of an attribute value on a given class is independent of the values of other attributes.
- **Linear classifiers** compute a linear weighted combination of the input attribute values, based on which it predicts the class label for a given tuple. **Perceptron** and **logistic regression** are two classic examples of linear classifiers.

- Decision tree classifiers, Bayesian classifiers, and linear classifiers are all examples of **eager learners** in that they use training tuples to construct a generalization model and in this way are ready for classifying new tuples. This contrasts with **lazy learners** or **instance-based** methods of classification, such as nearest-neighbor classifiers and case-based reasoning classifiers, which store all of the training tuples in pattern space and wait until presented with a test tuple before performing generalization. Hence lazy learners require efficient indexing techniques.
- A **confusion matrix** can be used to evaluate a classifier's quality. For a two-class problem, it shows the *true positives, true negatives, false positives*, and *false negatives*. Measures that assess a classifier's predictive ability include **accuracy**, **sensitivity** (also known as **recall**), **specificity**, **precision**, F, and F_β. Reliance on the accuracy measure can be deceiving when the main class of interest is in the minority.
- Construction and evaluation of a classifier require partitioning labeled data into a training set and a test set. **Holdout, random sampling, cross-validation**, and **bootstrapping** are typical methods used for such partitioning.
- Significance tests and ROC curves are useful tools for model selection. **Significance tests** can be used to assess whether the difference in accuracy between two classifiers is due to chance. **ROC curves** plot the true positive rate (or sensitivity) vs. the false positive rate (or $1 - specificity$) of one or more classifiers.
- **Ensemble methods** can be used to increase overall accuracy by learning and combining a series of individual (base) classifier models. **Bagging, boosting**, and **random forests** are popular ensemble methods.
- The **class imbalance problem** occurs when the main class of interest is represented by only a few tuples. Strategies to address this problem include **oversampling, undersampling, threshold moving,** and **ensemble techniques**.

6.9 Exercises

6.1. Briefly outline the major steps of *decision tree classification*.

6.2. Why is *tree pruning* useful in decision tree induction? What is a drawback of using a separate set of tuples to evaluate pruning?

6.3. Given a decision tree, you have the option of (a) *converting* the decision tree to rules and then pruning the resulting rules, or (b) *pruning* the decision tree and then converting the pruned tree to rules. What advantage does (a) have over (b)?

6.4. It is important to calculate the worst-case computational complexity of the decision tree algorithm. Given data set, D, the number of attributes, n, and the number of training tuples, $|D|$, analyze the computational complexity in terms of n and $|D|$.

6.5. Given a 5-GB data set with 50 attributes (each containing 100 distinct values) and 512 MB of main memory in your laptop, outline an efficient method that constructs decision trees in such large data sets. Justify your answer by a rough calculation of your main memory usage.

6.6. Why is *naïve Bayesian classification* called "naïve"? Briefly outline the major ideas of naïve Bayesian classification.

6.7. The following table consists of training data from an employee database. The data have been generalized. For example, "31 ... 35" for *age* represents the age range of 31 to 35. For a given

row entry, *count* represents the number of data tuples having the values for *department, status, age*, and *salary* given in that row.

department	*status*	*age*	*salary*	*count*
sales	senior	31...35	46K...50K	30
sales	junior	26...30	26K...30K	40
sales	junior	31...35	31K...35K	40
systems	junior	21...25	46K...50K	20
systems	senior	31...35	66K...70K	5
systems	junior	26...30	46K...50K	3
systems	senior	41...45	66K...70K	3
marketing	senior	36...40	46K...50K	10
marketing	junior	31...35	41K...45K	4
secretary	senior	46...50	36K...40K	4
secretary	junior	26...30	26K...30K	6

Let *status* be the class label attribute.

- **a.** How would you modify the basic decision tree algorithm to take into consideration the *count* of each generalized data tuple (i.e., of each row entry)?
- **b.** Use your algorithm to construct a decision tree from the given data.
- **c.** Given a data tuple having the values *"systems," "26...30,"* and *"46–50K"* for the attributes *department, age*, and *salary*, respectively, what would a naïve Bayesian classification of the *status* for the tuple be?

6.8. Compare the advantages and disadvantages of *eager* classification (e.g., decision tree, Bayesian, neural network) vs. *lazy* classification (e.g., k-nearest neighbor, case-based reasoning).

6.9. Write an algorithm for *k-nearest-neighbor classification* given k, the nearest number of neighbors, and n, the number of attributes describing each tuple.

6.10. RainForest is a scalable algorithm for decision tree induction. Develop a scalable naïve Bayesian classification algorithm that requires just a single scan of the entire data set for most databases. Discuss whether such an algorithm can be refined to incorporate *boosting* to further enhance its classification accuracy.

6.11. Design an efficient method that performs effective naïve Bayesian classification over an *infinite* data stream (i.e., you can scan the data stream only once). If we wanted to discover the *evolution* of such classification schemes (e.g., comparing the classification scheme at this moment with earlier schemes such as one from a week ago), what modified design would you suggest?

6.12. The perceptron model $y = f(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x} + b)$ can be used to learn a binary classifier from training data.

- **a.** Assume there are two training samples. The positive one is $\mathbf{x}_1 = (2, 1)^T$; the negative one is $\mathbf{x}_2 = (1, 0)^T$. The learning rate $\eta = 1$. Starting from $\mathbf{w} = (1, 1)^T$ and $b = 0$, solve the parameters of the classifier.
- **b.** Assume there are four training samples. The positive samples are $\mathbf{x}_1 = (1, 1)^T$ and $\mathbf{x}_2 = (0, 0)^T$; the negative samples are $\mathbf{x}_3 = (1, 0)^T$ and $\mathbf{x}_4 = (0, 1)^T$. Can we classify all training samples correctly using the perceptron model? Why?

6.13. Suppose we have three positive examples $x_1 = (1, 0, 0)$, $x_2 = (0, 0, 1)$ and $x_3 = (0, 1, 0)$ and three negative examples $x_4 = (-1, 0, 0)$, $x_5 = (0, -1, 0)$ and $x_6 = (0, 0, -1)$. Apply standard gradient ascent method to train a logistic regression classifier (without regularization terms).

Tuple #	Class	Probability
1	P	0.95
2	N	0.85
3	P	0.78
4	P	0.66
5	N	0.60
6	P	0.55
7	N	0.53
8	N	0.52
9	N	0.51
10	P	0.40

FIGURE 6.28

Tuples sorted by decreasing score, where the score is the value returned by a probabilistic classifier.

Initialize the weight vector with two different values and set $w_0^0 = 0$ (e.g. $w_0 = (0, 0, 0, 0)'$, $w_0 = (0, 0, 1, 0)'$). Would the final weight vector (w^*) be the same for the two different initial values? What are the values? Please explain your answer in detail. You may assume the learning rate to be a positive real constant η.

6.14. Suppose that we are training a naïve Bayes classifier and a logistic regression classifier: $f : \mathbf{X} \rightarrow Y$, which maps a d-dimensional real-valued feature vector $\mathbf{X} \in \mathbb{R}^d$ to a binary class label $Y \in \{0, 1\}$. In the naïve Bayes classifier, we assume that all $\mathbf{X}_i$ where $i = 1, ..., n$ are conditionally independent given the class label Y and the class prior $P(Y)$ follow the Bernoulli distribution with $P(Y = 1) = \theta$. Now, prove the equivalence of logistic regression and naïve Bayes under these two assumptions.

- **a.** For each $\mathbf{X}_i$, we assume it is drawn from the Gaussian distribution $P(\mathbf{X}_i|Y = k) \sim \mathcal{N}(\mu_{ik}, \sigma_{ik})$ where $k = 0, 1$. We also assume that $\sigma_{i0} = \sigma_{i1} = \sigma_i$.
- **b.** For each $\mathbf{X}_i$, we assume it is drawn from the Bernoulli distribution $P(\mathbf{X}_i = 1|Y = k) = p_k$ where $k = 0, 1$.

6.15. Show that accuracy is a function of *sensitivity* and *specificity*, that is, prove Eq. (6.25).

6.16. The harmonic mean is one of several kinds of averages. Chapter 2 discussed how to compute the *arithmetic mean*, which is what most people typically think of when they compute an average. The **harmonic mean**, H, of the positive real numbers, $x_1, x_2, \ldots, x_n$, is defined as

$$H = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}.$$

The F measure is the harmonic mean of precision and recall. Use this fact to derive Eq. (6.28) for F. In addition, write F_β as a function of true positives, false negatives, and false positives.

6.17. The data tuples of Fig. 6.28 are sorted by decreasing probability value, as returned by a classifier. For each tuple, compute the values for the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). Compute the true positive rate (TPR) and false positive rate (FPR). Plot the ROC curve for the data.

6.18. It is difficult to assess classification *accuracy* when individual data objects may belong to more than one class at a time. In such cases, comment on what criteria you would use to compare different classifiers modeled after the same data.

6.19. Suppose that we want to *select between two prediction models*, M_1 and M_2. We have performed 10 rounds of 10-fold cross-validation on each model, where the same data partitioning in round i is used for both M_1 and M_2. The error rates obtained for M_1 are 30.5, 32.2, 20.7, 20.6, 31.0, 41.0, 27.7, 26.0, 21.5, 26.0. The error rates for M_2 are 22.4, 14.5, 22.4, 19.6, 20.7, 20.4, 22.1, 19.4, 16.2, 35.0. Comment on whether one model is significantly better than the other considering a significance level of 1%.

6.20. What is *boosting*? State why it may improve the accuracy of decision tree induction.

6.21. Outline methods for addressing the *class imbalance problem*. Suppose a bank wants to develop a classifier that guards against fraudulent credit card transactions. Illustrate how you can induce a quality classifier based on a large set of legitimate examples and a very small set of fraudulent cases.

6.22. XGBoost is a scalable machine learning system for tree boosting. Its objective function has a training loss and a regularization term: $\mathcal{L} = \sum_i l(y_i, \hat{y}_i) + \sum_k \Omega(f_k)$. Read the XGBoost paper and answer the following questions:

- **a.** What is $\hat{y}_i$? At the tth iteration, XGBoost fixes $f_1, ..., f_{t-1}$ and trains the tth tree model f_t. How does XGBoost approximate the training loss $l(y_i, \hat{y}_i)$ here?
- **b.** What is $\Omega(f_k)$? Which part in the regularization term needs to be considered at the tth iteration?

6.10 Bibliographic notes

Classification is a fundamental topic in machine learning, statistics, and pattern recognition. Many textbooks from these fields highlight classification methods such as Mitchell [Mit97]; Bishop [Bis06a]; Duda, Hart, and Stork [DHS01]; Theodoridis and Koutroumbas [TK08]; Hastie, Tibshirani, and Friedman [HTF09]; Alpaydin [Alp11]; Marsland [Mar09]; and Aggarwal [Agg15a].

For decision tree induction, the C4.5 algorithm is described in a book by Quinlan [Qui93]. The CART system is detailed in *Classification and Regression Trees* by Breiman, Friedman, Olshen, and Stone [BFOS84]. Both books give an excellent presentation of many of the issues regarding decision tree induction. C4.5 has a commercial successor, known as C5.0, which can be found at *http://www.rulequest.com*. ID3, a predecessor of C4.5, is detailed in Quinlan [Qui86]. It expands on pioneering work on concept learning systems, described by Hunt, Marin, and Stone [HMS66].

Other algorithms for decision tree induction include FACT (Loh and Vanichsetakul [LV88]), QUEST (Loh and Shih [LS97]), PUBLIC (Rastogi and Shim [RS98]), and CHAID (Kass [Kas80] and Magidson [Mag94]). INFERULE (Uthurusamy, Fayyad, and Spangler [UFS91]) learns decision trees from inconclusive data, where probabilistic rather than categorical classification rules are obtained. KATE (Manago and Kodratoff [MK91]) learns decision trees from complex structured data. Incremental versions of ID3 include ID4 (Schlimmer and Fisher [SF86]) and ID5 (Utgoff [Utg88]), the latter of which is extended in Utgoff, Berkman, and Clouse [UBC97]. An incremental version of CART is described in Crawford [Cra89]. BOAT (Gehrke, Ganti, Ramakrishnan, and Loh [GGRL99]), a decision tree algorithm that addresses the scalability issue in data mining, is also incremental. Other decision tree

algorithms that address scalability include SLIQ (Mehta, Agrawal, and Rissanen [MAR96]), SPRINT (Shafer, Agrawal, and Mehta [SAM96]), RainForest (Gehrke, Ramakrishnan, and Ganti [GRG98]), and earlier approaches, such as Catlet [Cat91] and Chan and Stolfo [CS93a,CS93b].

For a comprehensive survey of many salient issues relating to decision tree induction, such as attribute selection and pruning, see Murthy [Mur98]. Perception-based classification (PBC), a visual and interactive approach to decision tree construction, is presented in Ankerst, Elsen, Ester, and Kriegel [AEEK99].

For a detailed discussion on attribute selection measures, see Kononenko and Hong [KH97]. Information gain was proposed by Quinlan [Qui86] and is based on pioneering work on information theory by Shannon and Weaver [SW49]. The gain ratio, proposed as an extension to information gain, is described as part of C4.5 (Quinlan [Qui93]). The Gini impurity was proposed for CART by Breiman, Friedman, Olshen, and Stone [BFOS84]. The G-statistic, based on information theory, is given in Sokal and Rohlf [SR81]. Comparisons of attribute selection measures include Buntine and Niblett [BN92], Fayyad and Irani [FI92], Kononenko [Kon95], Loh and Shih [LS97], and Shih [Shi99]. Fayyad and Irani [FI92] show limitations of impurity-based measures, such as information gain and the Gini impurity. They propose a class of attribute selection measures called C-SEP (Class SEParation), which outperform impurity-based measures in certain cases.

Kononenko [Kon95] notes that attribute selection measures based on the minimum description length principle have the least bias toward multivalued attributes. Martin and Hirschberg [MH95] proved that the time complexity of decision tree induction increases exponentially with respect to tree height in the worst case, and under fairly general conditions in the average case. Fayad and Irani [FI90] found that shallow decision trees tend to have many leaves and higher error rates for a large variety of domains. Attribute (or feature) construction is described in Liu and Motoda [LM98,Le98].

There are numerous algorithms for decision tree pruning, including cost complexity pruning (Breiman, Friedman, Olshen, and Stone [BFOS84]), reduced error pruning (Quinlan [Qui87]), and pessimistic pruning (Quinlan [Qui86]). PUBLIC (Rastogi and Shim [RS98]) integrates decision tree construction with tree pruning. MDL-based pruning methods can be found in Quinlan and Rivest [QR89]; Mehta, Agrawal, and Rissanen [MAR96]; and Rastogi and Shim [RS98]. Other methods include Niblett and Bratko [NB86] and Hosking, Pednault, and Sudan [HPS97]. For an empirical comparison of pruning methods, see Mingers [Min89] and Malerba, Floriana, and Semeraro [MFS95]. For a survey on simplifying decision trees, see Breslow and Aha [BA97].

Thorough presentations of Bayesian classification can be found in Duda, Hart, and Stork [DHS01], Weiss and Kulikowski [WK91], and Mitchell [Mit97]. For an analysis of the predictive power of naïve Bayesian classifiers when the class-conditional independence assumption is violated, see Domingos and Pazzani [DP96]. Experiments with kernel density estimation for continuous-valued attributes, rather than Gaussian estimation, have been reported for naïve Bayesian classifiers in John [Joh97].

Nearest-neighbor classifiers were introduced in 1951 by Fix and Hodges [FH51]. A comprehensive collection of articles on nearest-neighbor classification can be found in Dasarathy [Das91]. Additional references can be found in many texts on classification, such as Duda, Hart, and Stork [DHS01] and James [Jam85], as well as articles by Cover and Hart [CH67] and Fukunaga and Hummels [FH87]. Their integration with attribute weighting and the pruning of noisy instances is described in Aha [Aha92]. The use of search trees to improve nearest-neighbor classification time is detailed in Friedman, Bentley, and Finkel [FBF77]. The partial distance method was proposed by researchers in vector quantization and compression. It is outlined in Gersho and Gray [GG92]. The editing method for re-

moving "useless" training tuples was first proposed by Hart [Har68]. For speeding-up the computation of k-nearest neighbor based on locality sensitive hashing, see Pan and Manocha [PM12,PM11]; and Zhang, Huang, Geng and Liu [ZHGL13].

The computational complexity of nearest-neighbor classifiers is described in Preparata and Shamos [PS85]; and Haghani, Sebastian, and Karl [HMA09]. References on case-based reasoning include the texts by Riesbeck and Schank [RS89] and Kolodner [Kol93], as well as Leake [Lea96] and Aamodt and Plazas [AP94]. For a list of business applications, see Allen [All94]. Examples in medicine include CASEY by Koton [Kot88] and PROTOS by Bareiss, Porter, and Weir [BPW88], whereas Rissland and Ashley [RA87] is an example of CBR for law. CBR is available in several commercial software products.

Linear regression and its numerous variants, such as RIDGE regression, robust regression are covered in most statistics textbooks, such as Freedman [Fre09]; Draper and Smith [DS98]; and Fox [Fox97]. For LASSO, see Tibshirani [Tib11]. Perceptron was first invented by Rosenblatt [Ros58], and Novikoff [Nov63] analyzed its convergence property. The Perceptron is one of the earliest linear classifiers, proposed in 1958 by Rosenblatt [Ros58], which became a landmark in early machine learning history. In 1969, Minsky and Papert [MP69] showed that Perceptrons are incapable of learning concepts that are linearly inseparable. A general introduction to logistic regression can be found in most machine learning textbooks, such as Mitchell [Mit97]; Hastie, Tibshirani, and Friedman [HTF09]; and Aggarwal [Agg15a]. Ng and Jordan [NJ02] conducted a thorough comparison between naïve Bayesian classifier and logistic regression. The relationship between logistic regression and log-linear model can be found in Christensen [Chr06].

Issues involved in estimating classifier accuracy are described in Weiss and Kulikowski [WK91] and Witten and Frank [WF05]. Sensitivity, specificity, and precision are discussed in most information retrieval textbooks. For the F and F_β measures, see van Rijsbergen [vR90]. The use of stratified 10-fold cross-validation for estimating classifier accuracy is recommended over the holdout, cross-validation, leave-one-out (Stone [Sto74]), and bootstrapping (Efron and Tibshirani [ET93]) methods, based on a theoretical and empirical study by Kohavi [Koh95]. See Freedman, Pisani, and Purves [FPP07] for the confidence limits and statistical tests of significance.

For ROC analysis, see Egan [Ega75], Swets [Swe88], and Vuk and Curk [VC06]. Bagging is proposed in Breiman [Bre96]. Freund and Schapire [FS97] proposed AdaBoost. This boosting technique has been applied to several different classifiers, including decision tree induction (Quinlan [Qui96]) and naïve Bayesian classification (Elkan [Elk97]). Friedman [Fri01] proposed the gradient boosting machine. Chen and Guestrin designed a highly scalable system called Xgboost [CG16]. The ensemble technique of random forests is described by Breiman [Bre01]. Seni and Elder [SE10] proposed the Importance Sampling Learning Ensembles (ISLE) framework, which views bagging, AdaBoost, random forests, and gradient boosting as special cases of a generic ensemble generation procedure. There are many online software packages for ensemble routines, including bagging, AdaBoost, gradient boosting, and random forests. Studies on the class imbalance problem and/or cost-sensitive learning include Weiss [Wei04], Zhou and Liu [ZL06], Zapkowicz and Stephen [ZS02], Elkan [Elk01], Domingos [Dom99], and Huang, Li, Loy and Tang [HLLT16].

The University of California at Irvine (UCI) maintains a Machine Learning Repository of data sets for the development and testing of classification algorithms. It also maintains a Knowledge Discovery in Databases (KDD) Archive, an online repository of large data sets that encompasses a wide vari-

ety of data types, analysis tasks, and application areas. For information on these two repositories, see *http://www.ics.uci.edu/~mlearn/MLRepository.html* and *http://kdd.ics.uci.edu*.

No classification method is superior to all others for all data types and domains. Empirical comparisons of classification methods include Quinlan [Qui88]; Shavlik, Mooney, and Towell [SMT91]; Brown, Corruble, and Pittard [BCP93]; Curram and Mingers [CM94]; Michie, Spiegelhalter, and Taylor [MST94]; Brodley and Utgoff [BU95]; and Lim, Loh, and Shih [LLS00].

CHAPTER

Cluster analysis: basic concepts and methods

8

Imagine that you are the director of Customer Relationships at a retail company. Managing millions of customers one by one is inefficient and ineffective. You would like to organize all customers of the company into a small number of groups so that each group can be assigned to a different manager. Strategically, you would like that the customers in each group are as similar as possible. Two customers having very different business patterns should not be placed in the same group. Your intention behind this business strategy is to develop customer relationship campaigns that specifically target each group, based on common features shared by the customers in the group. What kind of data mining techniques can help you accomplish this task?

Unlike in classification, the class label (i.e., the group-id in this context) of each customer is unknown in this new task. You need to *discover* these groupings. Given a large number of customers and many attributes describing customer profiles, it can be costly or even infeasible to manually study the data and come up with a way to partition the customers into strategic groups. You need a *clustering* tool to help.

Clustering is the process of grouping a set of data objects into multiple groups or *clusters* so that objects within a cluster have high similarity, but are dissimilar to objects in other clusters. Dissimilarities and similarities are assessed based on the attribute values describing the objects and often involve distance measures.[1] Clustering as a data mining tool has its roots in many application areas, such as biology, security, business intelligence, and Web search.

This chapter presents the basic concepts and methods of cluster analysis. In Section 8.1, we introduce the basic concept of clustering and study the requirements of clustering methods for massive amounts of data and various applications. You will learn several basic clustering techniques, organized into several categories, namely *partitioning methods* (Section 8.2), *hierarchical methods* (Section 8.3), and *density-based and grid-based methods* (Section 8.4). In Section 8.5, we discuss how to evaluate clustering methods. A discussion of advanced methods of clustering is reserved for Chapter 9.

8.1 Cluster analysis

This section sets up the groundwork for studying cluster analysis. Section 8.1.1 defines cluster analysis and presents examples where clustering is useful. In Section 8.1.2, you will learn aspects for comparing

[1] Data similarity and dissimilarity are discussed in detail in Chapter 2. You may want to refer to the corresponding section for a quick review.

Data Mining. https://doi.org/10.1016/B978-0-12-811760-6.00018-7

clustering methods, as well as requirements for clustering. An overview of basic clustering techniques is presented in Section 8.1.3.

8.1.1 What is cluster analysis?

Cluster analysis or simply **clustering** is the process of partitioning a set of data objects (or observations) into subsets. Each subset is a **cluster**, such that objects in a cluster are similar to one another, yet dissimilar to objects in other clusters. The set of clusters resulting from a cluster analysis can be referred to as a **clustering**. In this context, different clustering methods may generate different clusterings on the same data set. The same clustering method equipped with different parameters or even different initializations may also produce different clusterings. Such partitioning is not performed by humans, but by a clustering algorithm. Hence, clustering is useful in that it can lead to the discovery of previously unknown groups within the data.

Cluster analysis has been widely used in many applications such as business intelligence, image pattern recognition, Web search, biology, and security. For example, in business intelligence, clustering can be used to organize a large number of customers into groups, where customers within a group share strong similar characteristics. This facilitates the development of business strategies for enhanced customer relationship management. Moreover, consider a consultant company with a large number of projects. To improve project management, such as project delivery and outcome quality control, clustering can be applied to partition projects into categories based on similarities in, for example, business scenarios, customers, expertise required, period and size, so that project auditing and diagnosis can be conducted effectively.

In image recognition, as another example, clustering can be used to discover clusters or "subclasses" in photos. One application is to automatically group photos according to faces recognized in the images so that the photos of the same person may likely come together into a group. Here, we do not have to specify and label the persons in the photos beforehand, and thus a classification method cannot be applied. A clustering method can use faces as features and partition photos into groups so that the faces in the same group are similar and the faces in different groups are dissimilar. Moreover, typically there are many different ways to organize photos. Clustering can help automatically identify significant features and suggest meaningful ways to organize photos into groups accordingly. For example, a group of scenic pictures may be formed using the features of blue sky and beach, whereas another group may share the theme of snow, and a third group highlights group photos with many faces.

Clustering has also found many applications in Web search. For example, a keyword search may often return a large number of hits (i.e., pages relevant to the search) due to the extremely large number of web pages. Clustering can be used to organize the search results into groups and present the results in a concise and easily accessible way. Moreover, clustering techniques have been developed to cluster documents into topics, which are commonly used in information retrieval practice.

As a data mining function, cluster analysis can be used as a standalone tool to gain insight into the distribution of data, to observe the characteristics of each cluster, and to focus on a particular set of clusters for further analysis. Alternatively, it may serve as a preprocessing step for other algorithms, such as characterization, attribute subset selection, and classification, which would then operate on the detected clusters and the selected attributes or features.

Because a cluster is a collection of data objects that are similar to one another within the cluster and dissimilar to objects in other clusters, a cluster of data objects can be treated as an implicit class.

In this sense, clustering is sometimes called **automatic classification** or **unsupervised classification**. Again, a critical difference here is that clustering can automatically find the groupings. This is a distinct advantage of cluster analysis.

Clustering is also called **data segmentation** in some applications because clustering partitions large data sets into groups according to their *similarity*. Clustering can also be used for **outlier detection**, where outliers (values that are "far away" from any cluster) may be more interesting than common cases. Applications of outlier detection include the detection of credit card frauds and the monitoring of criminal activities in electronic commerce. For example, exceptional cases in credit card transactions, such as very expensive and infrequent purchases at unusual locations, may be of interest as possible fraudulent activities. Outlier detection is the subject of Chapter 11.

Data clustering is under vigorous development. Contributing areas of research include data mining, statistics, machine learning and deep learning, spatial database technology, information retrieval, Web search, biology, marketing, and many other application areas. Owing to the huge amounts of data collected in databases, cluster analysis has become a highly active topic in data mining research.

As a branch of statistics, cluster analysis has been extensively studied, with the main focus on *distance-based cluster analysis*. Cluster analysis tools based on k-means, k-medoids, and several other methods also have been built into many statistical analysis software packages or systems, such as S-Plus, SPSS, and SAS. In machine learning, recall that classification is known as supervised learning because the class label information is given, that is, the learning algorithm is supervised in that it is told the class membership of each training tuple. Clustering is known as **unsupervised learning** because the class label information is not present. For this reason, clustering is a form of **learning by observation**, rather than *learning by examples*. In data mining, efforts have focused on finding methods for efficient and effective cluster analysis in *large data sets*. Active themes of research focus on the *scalability* of clustering methods, the effectiveness of methods for clustering *complex shapes* (e.g., nonconvex) and *types of data* (e.g., text, graphs, and images), *high-dimensional* clustering techniques (e.g., clustering objects with thousands or even millions of features), and methods for clustering *mixed numerical and nominal data* in large data sets.

8.1.2 Requirements for cluster analysis

Clustering is a challenging research field. In this section, you will learn about the requirements for clustering as a data mining tool, as well as aspects that can be used for comparing clustering methods.

When we think about employing a clustering method, what requirements should we consider for the method? The following are typical requirements of clustering in data mining.

- **Ability to deal with various kinds of data objects**: Many algorithms are designed to cluster numeric (interval-based) data objects. However, applications may require clustering objects based on a mixed data types, such as binary, nominal (categorical), ordinal, and numerical data, as well as data objects of various kinds, such as text, graphs, sequences, images, and videos.
- **Scalability**: Many clustering algorithms work well on small data sets containing fewer than several hundred data objects; however, a large database may contain millions or even billions of objects, such as in Web search scenarios. Clustering on only a sample of a given large data set may lead to biased results. Therefore highly scalable clustering algorithms are needed.
- **Discovery of clusters with arbitrary shape**: Many clustering algorithms determine clusters based on Euclidean or Manhattan distance measures (Chapter 2). Algorithms based on such distance mea-

sures tend to find spherical clusters with similar size and density. However, a cluster could be of any shape. For example, we may want to use clustering to find the frontier of a running forest fire in a satellite image, which is often not spherical. It is important to develop algorithms that can detect clusters of arbitrary shape.

- **Requirements for domain knowledge to determine input parameters**: Many clustering algorithms require users to provide domain knowledge in the form of input parameters such as the desired number of clusters. Consequently, the clustering results may be sensitive to such parameters. Parameters are often hard to determine, especially for data sets of high dimensionality where users have yet to grasp a deep understanding of their data. Requiring the specification of domain knowledge not only burdens users, but also makes the quality of clustering difficult to control. Clustering algorithms that do not heavily rely on domain knowledge input or can help users explore domain knowledge are highly preferable.
- **Ability to deal with noisy data**: Most real-world data sets contain outliers and/or missing, unknown, or erroneous data. For example, data collected by physical sensors is often noisy. Clustering algorithms can be sensitive to such noise and may produce poor-quality clusters. Therefore, we need clustering methods that are robust to noise.
- **Incremental clustering and insensitivity to input order**: In many applications, incremental updates (representing newer data) may arrive at any time. Some clustering algorithms cannot incorporate incremental updates into existing clustering structures and, instead, have to recompute a new clustering from scratch. Clustering algorithms may also be sensitive to the input data order. That is, given a set of data objects, clustering algorithms may return dramatically different clusterings depending on the order in which the objects are presented. Incremental clustering algorithms and algorithms that are insensitive to the input order are needed.
- **Capability of clustering high-dimensionality data**: A data set can contain numerous dimensions or attributes. When clustering documents, for example, each keyword can be regarded as a dimension, and there are often thousands of keywords. Most clustering algorithms are good at handling low-dimensional data such as data sets involving only two or three dimensions. Finding clusters of data objects in a high-dimensional space is challenging, especially considering that such data can be very sparse and highly skewed.
- **Constraint-based clustering**: Real-world applications may need to perform clustering under various kinds of constraints. Suppose that your task is to choose the locations for a given number of new electric vehicle charging stations in a city. To decide upon this, you may cluster potential charging needs while considering constraints such as available spaces, electricity networks, and the river and highway networks in a city. A challenging task is to find data groups with good clustering behaviors that satisfy specified constraints.
- **Interpretability and usability**: Users want clustering results to be interpretable, comprehensible, and usable. That is, clustering may need to be tied with specific semantic interpretations and applications. It is important to study how an application goal may influence the selection of clustering features and clustering methods.

Given one data set, using different clustering methods or using different parameters or initializations, we may be able to obtain different clusterings. How can we evaluate and compare the clusterings? The following are the orthogonal aspects with which clustering methods can be compared.

- **Single vs. multilevel clustering**: In many clustering methods, all the objects are partitioned so that no hierarchy exists among the clusters. That is, all the clusters are at the same level conceptually. Such a method is useful, for example, for partitioning customers into groups so that each group has its own manager. Alternatively, other methods partition data objects hierarchically, where clusters can be formed at different semantic levels. For example, in text mining, we may want to organize a corpus of documents into multiple general topics, such as "politics" and "sports," each of which may have subtopics. For instance, "football," "basketball," "baseball," and "hockey" can exist as subtopics of "sports." The latter four subtopics are at a lower level in the hierarchy than "sports."
- **Separation of clusters**: Some methods partition data objects into mutually exclusive clusters. In some other situations, the clusters may not be exclusive, that is, a data object may belong to more than one cluster. For example, when clustering documents into topics, a document may be related to multiple topics. Thus the topics as clusters may not be exclusive.
- **Similarity measure**: Some methods determine the similarity between two objects by the distance between them. Such a distance can be defined on a Euclidean space, a road network, a vector space, or some other space. For some applications, similarity may also be defined by other means such as connectivity based on density or contiguity and thus may or may not rely on the absolute distance between two objects. Similarity measures play a fundamental role in the design of clustering methods. While distance-based methods can often take advantage of some computation and optimization techniques, density- and continuity-based methods can often find clusters of arbitrary shape.
- **Clustering in full space vs. subspace**: Many clustering methods search for clusters within the entire given data space. These methods are useful for low-dimensionality data sets. With high-dimensional data, however, there can be many irrelevant attributes, which can make similarity measurements unreliable. Consequently, clusters found in the full space are often meaningless. It is often better to instead search for clusters within different subspaces of the same data set. *Subspace clustering* discovers both clusters and subspaces (often of low dimensionality) containing interesting clusters.

To conclude, clustering algorithms have a series of requirements. These factors include the ability to deal with different kinds of data objects, scalability, robustness to noisy data, incremental updates, clusters of arbitrary shape, and constraints. Interpretability and usability are also important. In addition, clustering methods can differ with respect to single vs. multilevel clustering, whether or not clusters are mutually exclusive, the similarity measures used, and whether or not subspace clustering is performed.

8.1.3 Overview of basic clustering methods

There are many clustering algorithms in the literature. It is difficult to provide a crisp categorization of clustering methods because these categories may overlap so that a method may have features from several categories. Nevertheless, it is useful to present a relatively organized picture of clustering methods. In general, the major fundamental clustering methods can be classified into the following categories, which are discussed in the rest of this chapter.

Partitioning methods: Given a set of n objects, a partitioning method constructs k ($k \leq n$) partitions of the data, where each partition represents a cluster. That is, it divides the data into k groups such that each group must contain at least one object. Typically, k is set to a small number, that is, $k \ll n$. In other words, a partitioning method conducts one-level partitioning on data sets. The basic partitioning methods typically adopt *exclusive cluster separation*. That is, each object must

belong to exactly one group. This requirement may be relaxed. For example, in fuzzy partitioning techniques an object may take probabilities to belong to more than one cluster. References to such techniques are given in the bibliographic notes (Section 8.8).

Most partitioning methods are distance-based. Given k, the number of partitions to construct, a partitioning method creates an initial partitioning. It then uses an **iterative relocation technique** that attempts to improve the partitioning by moving objects from one group to another. The general criterion of a good partitioning is that objects in the same cluster are "close" or related to each other, whereas objects in different clusters are "far apart" or very different. There are various kinds of other criteria for judging the quality of partitions. Traditional partitioning methods can be extended for subspace clustering rather than searching the full data space. This is useful when there are many attributes and the data is sparse.

Achieving global optimality in partitioning-based clustering is often computationally prohibitive, potentially requiring an exhaustive enumeration of all the possible partitions. Instead, most applications adopt popular heuristic methods, such as greedy approaches like the *k-means* and the *k-medoids* algorithms, which progressively improve the clustering quality and approach a local optimum. These heuristic clustering methods work well for finding spherical-shaped clusters in small- to medium-sized data sets. To find clusters with complex shapes and for very large data sets, partitioning-based methods need to be extended. Partitioning-based clustering methods are studied in depth in Section 8.2.

Hierarchical methods: A hierarchical method creates a hierarchical decomposition of a given set of data objects. A hierarchical method can be classified as being either *agglomerative* or *divisive*, based on how the hierarchical decomposition is formed. The *agglomerative approach*, also called the *bottom-up* approach, starts with each object forming a separate group. It successively merges the objects or groups close to one another, until all the groups are merged into one (the topmost level of the hierarchy), or a termination condition holds. The *divisive approach*, also called the *top-down* approach, starts with all the objects in the same cluster. In each successive iteration, a cluster is split into smaller clusters, until eventually each object is in one cluster, or a termination condition holds.

Hierarchical clustering methods can be distance-, density-, or continuity-based. Various extensions of hierarchical methods consider clustering in subspaces as well.

Hierarchical methods suffer from the fact that once a step (merge or split) is done, it can never be reverted. This rigidity is useful in that it leads to smaller computation costs by not having to worry about a combinatorial number of different choices. Such techniques cannot correct erroneous decisions; however, methods for improving the quality of hierarchical clustering have been proposed. Hierarchical clustering methods are studied in Section 8.3.

Density-based and grid-based methods: Most partitioning methods cluster objects based on the distance between objects. Such methods can find only spherical-shaped clusters and encounter difficulty in discovering clusters of arbitrary shapes. Other clustering methods have been developed based on the notion of *density*. Their general idea is to continue growing a given cluster as long as the density (number of objects or data points) in the "neighborhood" exceeds some threshold. For example, for each data point within a given cluster, the neighborhood of a given radius has to contain at least a minimum number of points. Such a method can be used to filter out noise or outliers and discover clusters of arbitrary shape.

Density-based methods can divide a set of objects into multiple exclusive clusters, or a hierarchy of clusters. Typically, density-based methods consider exclusive clusters only, and do not consider fuzzy clusters. Moreover, density-based methods can be extended from full space to subspace clustering.

One way to implement the idea of density-based clustering is grid-based methods, which quantize the object space into a finite number of cells that form a grid structure. All the clustering operations are performed on the grid structure (i.e., on the quantized space). For example, the dense cells, that is, those cells each containing a sufficient number of data points, are considered components of clusters, and are used to assemble clusters. The main advantage of the grid-based methods is the fast processing time, which is typically independent of the number of data objects and dependent only on the number of cells in each dimension in the quantized space. Using grids is often an efficient approach to many spatial data mining problems, including clustering. In addition to density-based clustering, grid-based methods can be integrated with other clustering methods, such as hierarchical methods. Density-based and grid-based clustering methods are studied in Section 8.4.

Some clustering algorithms integrate the ideas of several clustering methods, so that it is sometimes difficult to classify a given algorithm as uniquely belonging to only one clustering method category. Furthermore, some applications may have clustering criteria that require an integration of several clustering techniques.

In the following sections, we examine representative clustering methods in detail. Advanced clustering methods and related issues are discussed in Chapter 9.

8.2 Partitioning methods

The simplest and most fundamental version of cluster analysis is partitioning. In partitioning clustering, we organize the objects in a given set into several exclusive groups or clusters. Each cluster can be typified by a representative. In other words, each object o can be assigned to the cluster whose representative that o is the closest or most similar to. To keep the problem specification concise, we can assume that the number of expected clusters is given. This parameter is the starting point for partitioning methods. There are two foremost technical issues in partitioning methods. First, how can we decide the representatives of clusters? Second, how can we measure the distance or similarity between objects or between objects and representatives.

Formally, given a data set, D, of n objects, and k, the number of clusters to form, a **partitioning algorithm** organizes the objects into k partitions ($k \leq n$), where each partition represents a cluster. The clusters are formed to optimize an objective partitioning criterion, such as a dissimilarity function based on distance, so that the objects within a cluster are "similar" to one another and "dissimilar" to the objects in other clusters in terms of the data set attributes.

In this section you will learn the partitioning methods. We will start with the most prominent partitioning method, k-means (Section 8.2.1). Then, in Section 8.2.2 we will look at a series of variations of partitioning methods to address different types of data and different application scenarios. Last, we will discuss kernel k-means, an advanced version of partitioning method that can explore nonlinear separability between clusters in high dimensional data.

8.2.1 *k*-Means: a centroid-based technique

Suppose a data set, D, contains n objects in Euclidean space. Partitioning methods distribute the objects in D into k clusters, $C_1, \ldots, C_k$, that is, $C_i \subset D$, $|C_i| \geq 1$, and $C_i \cap C_j = \emptyset$ for $(1 \leq i, j \leq k, i \neq j)$. Each cluster is required to have at least one object. An objective function is used to assess the partitioning quality so that objects within a cluster are similar to one another but dissimilar to objects in other clusters. This is, the objective function aims for high intracluster similarity and low intercluster similarity.

A centroid-based partitioning technique uses the *centroid* of a cluster, C_i, as the representative of that cluster. Conceptually, the centroid of a cluster is its center point. The centroid can be defined in various ways such as by the mean or medoid of the objects (or points) assigned to the cluster. The difference between an object $\boldsymbol{p} \in C_i$ and $\boldsymbol{c_i}$, the representative of the cluster, is measured by $dist(\boldsymbol{p}, \boldsymbol{c_i})$, where $dist(\boldsymbol{x}, \boldsymbol{y})$ is the Euclidean distance between two points $\boldsymbol{x}$ and $\boldsymbol{y}$. The quality of cluster C_i can be measured by the **within-cluster variation**, which is the sum of *squared error* between all objects in C_i and the centroid $\boldsymbol{c_i}$, defined as

$$E = \sum_{i=1}^{k} \sum_{\boldsymbol{p} \in C_i} dist(\boldsymbol{p}, \boldsymbol{c_i})^2, \tag{8.1}$$

where E is the sum of the squared error for all objects in the data set; $\boldsymbol{p}$ is the point in space representing a given object; and $\boldsymbol{c_i}$ is the centroid of cluster C_i (both $\boldsymbol{p}$ and $\boldsymbol{c_i}$ are multidimensional). In other words, for each object in each cluster, the distance from the object to its cluster center is squared, and the distances are summed. This objective function tries to make the resulting k clusters as compact and separate as possible. The task of partitioning clustering can be modeled as to minimize the within-cluster variation (Eq. (8.1)) among all possible assignments of objects into clusters.

Minimizing the within-cluster variation is computationally challenging. In the worst case, we would have to enumerate the number of all possible partitionings. It is easy to see that the number of all possible partitionings is exponential to the number of objects. (This is left to be an exercise.) It has been shown that the problem is NP-hard in the general Euclidean space even for two clusters (i.e., $k = 2$). To overcome the prohibitive computational cost for the exact solution, greedy approaches are often used in practice. A prime example is the k-means algorithm, which is simple and commonly used.

"How does the k*-means algorithm work?"* The k-means algorithm defines the centroid of a cluster as the mean value of the points within the cluster. It proceeds as follows. First, it randomly selects k objects from D, each of which initially represents a cluster mean or center. For each of the remaining objects, an object is assigned to the cluster to which it is the most similar, based on the Euclidean distance between the object and the chosen means. The k-means algorithm then iteratively improves the within-cluster variation. For each cluster, it computes the new mean using the objects assigned to the cluster in the previous iteration. All the objects are then reassigned using the updated means as the new cluster centers. The iterations continue until the assignment is stable, that is, the clusters formed in the current round are the same as those formed in the previous round. The k-means procedure is summarized in Fig. 8.1.

Example 8.1. Clustering by *k*-means partitioning. Consider a set of objects located in 2-D space, as depicted in Fig. 8.2(a). Let $k = 3$, that is, the user would like to partition the objects into three clusters.

According to the algorithm in Fig. 8.1, we arbitrarily choose three objects as the three initial cluster centers, where cluster centers are marked by a $+$. Each object is assigned to a cluster based on the cluster

Algorithm: *k*-means. The *k*-means algorithm for partitioning, where each cluster's center is represented by the mean value of the objects in the cluster.

Input:

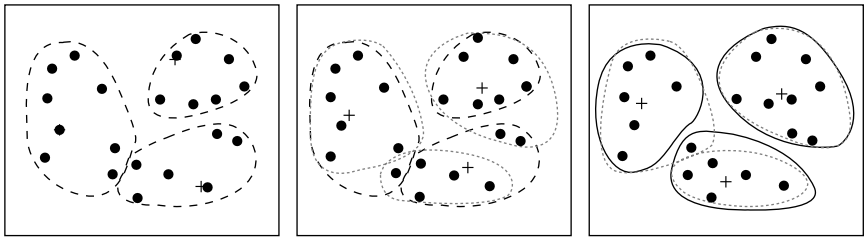
- k: the number of clusters,
- D: a data set containing n objects.

Output: A set of k clusters.
Method:

(1) arbitrarily choose k objects from D as the initial cluster centers;
(2) **repeat**
(3) (re)assign each object to the cluster to which the object is the most similar;
(4) update the cluster centers, that is, calculate the mean value of the objects for each cluster;
(5) **until** no change;

FIGURE 8.1

The *k*-means partitioning algorithm.



(a) Initial clustering **(b)** Iterate **(c)** Final clustering

FIGURE 8.2

Clustering of a set of objects using the *k*-means method; for (b) update cluster centers and reassign objects accordingly (the mean of each cluster is marked by a +).

center to which it is the nearest. Such an assignment forms silhouettes encircled by dotted curves, as shown in Fig. 8.2(a).

Next, the cluster centers are updated. That is, the mean value of each cluster is recalculated based on the current objects in the cluster. Using the new cluster centers, the objects are reassigned to the clusters based on which cluster center is the nearest. Such a reassignment forms new silhouettes encircled by dashed curves, as shown in Fig. 8.2(b).

This process iterates, leading to Fig. 8.2(c). The process of iteratively reassigning objects to clusters to improve the partitioning is referred to as *iterative relocation*. Eventually, no reassignment of the objects in any cluster occurs and so the process terminates. The resulting clusters are returned by the clustering process. □

The *k*-means method is not guaranteed to converge to the global optimum and often terminates at a local optimum. The results may depend on the initial random selection of cluster centers. (You will be asked to give an example to show this as an exercise.) To obtain good results in practice, it is common

to run the k-means algorithm multiple times with different initial cluster centers. The clustering with the smallest within-cluster variation should be returned as the final result.

The time complexity of the k-means algorithm is $O(nkt)$, where n is the total number of objects, k is the number of clusters, and t is the number of iterations. Normally, $k \ll n$ and $t \ll n$. Therefore the method is relatively scalable and efficient in processing large data sets.

As the simplest version of partitioning methods, the k-means method has several advantages. First, the k-means method is conceptually intuitive and relatively simple to implement. Indeed, the k-means method is included in many software toolkits and open source suites in statistics, data mining, and machine learning. Second, as analyzed, the k-means method is scalable to large data sets. The runtime is linear with respect to the data set size (i.e., the number of data objects), the number of clusters, and the number of iterations. Third, the k-means method is guaranteed to converge to some local optimum, and thus likely does not produce very poor results. Fourth, if a user has some domain knowledge about the possible locations of the clusters, the user can set the initial means and then run the iteration steps. In other words, the k-means method can take a warm-start. Last, the k-means method can take new observed data easily. That is, if some new data objects arrive after a certain number of iterations, the k-means method still can easily take those data into the next iteration, and the updated clustering can adapt to the new data.

The k-means method also has some limitations. First, a user has to manually specify the number of clusters. When a user is not familiar with a data set, it is not easy to set this parameter properly. Second, the effect of result clustering heavily depends on the choice of initial means. When the number of clusters is small, to overcome this limitation, one may run the k-means method multiple times with different initial means. However, when the number of clusters is large, even running the k-means method multiple times many not help to mitigate the issue, since it is unlikely all clusters produced in a run are all good. Third, as to be illustrated later (see Fig. 8.16 as an example), the k-means method may meet difficulty in finding clusters of substantially different sizes and density. Moreover, outliers may distract the centers of clusters (see Example 8.2 for demonstration). Last, since the Euclidean distance is used in the k-means method, when the dimensionality increases, the distance measure is mainly dominated by noise. In expectation, the distance between any two data objects in a high dimensional space is the same. Thus the k-means method cannot scale up to high dimensional data straightforwardly.

In the rest of the section, we will discuss some variations of the k-means method to address some of the above limitations. Some other limitations are unfortunately shared by the partitioning methods, and thus have to be tackled by introducing other types of clustering methods.

8.2.2 Variations of k-means

In order to tackle various limitations, there are multiple variants of the k-means method. In this subsection, we will study some of them that are popularly used in various applications.

k-*Medoids: a representative object-based technique*

The k-means algorithm is sensitive to outliers because such objects are far away from the majority of the data, and thus, when assigned to a cluster, they can dramatically distort the mean value of the cluster. This inadvertently affects the assignment of other objects to clusters. This effect is particularly exacerbated due to the use of the *squared*-error function (Eq. (8.1)), as observed in Example 8.2.

Example 8.2. A drawback of *k*-means. Consider six points in a 1-D space having values 1, 2, 3, 8, 9, 10, and 25, respectively. Intuitively, by visual inspection we may imagine the points partitioned into the clusters $\{1, 2, 3\}$ and $\{8, 9, 10\}$, where point 25 is excluded because it appears to be an outlier. How would k-means partition the values? If we apply k-means using $k = 2$ and Eq. (8.1), the partitioning $\{\{1, 2, 3\}, \{8, 9, 10, 25\}\}$ has the within-cluster variation

$$(1-2)^2 + (2-2)^2 + (3-2)^2 + (8-13)^2 + (9-13)^2 + (10-13)^2 + (25-13)^2 = 196,$$

given that the mean of cluster $\{1, 2, 3\}$ is 2 and the mean of $\{8, 9, 10, 25\}$ is 13. Compare this to the partitioning $\{\{1, 2, 3, 8\}, \{9, 10, 25\}\}$, for which k-means computes the within-cluster variation as

$$\begin{aligned}(1-3.5)^2 + (2-3.5)^2 + (3-3.5)^2 + (8-3.5)^2 + (9-14.67)^2 \\ + (10-14.67)^2 + (25-14.67)^2 = 189.67,\end{aligned}$$

given that 3.5 is the mean of cluster $\{1, 2, 3, 8\}$ and 14.67 is the mean of cluster $\{9, 10, 25\}$. The latter partitioning has the lower within-cluster variation; therefore the k-means method assigns the value 8 to a cluster different from that containing values 9 and 10 due to the outlier point 25. Moreover, the center of the second cluster, 14.67, is substantially far from all the members in the cluster. □

"How can we modify the k-means algorithm to diminish such sensitivity to outliers?" Instead of taking the mean value of the objects in a cluster as a reference point, we can pick actual objects to represent the clusters, using one representative object per cluster. Each remaining object is assigned to the cluster of which the representative object is the most similar. The partitioning method is then performed based on the principle of minimizing the sum of the dissimilarities between each object $\boldsymbol{p}$ and its corresponding representative object. That is, an **absolute-error criterion** is used, defined as

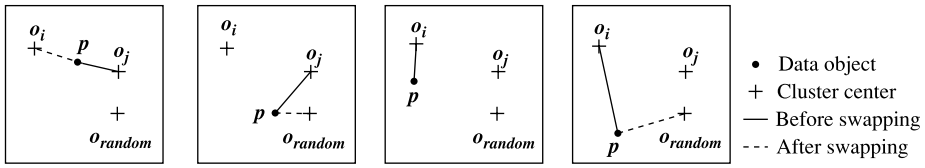
$$E = \sum_{i=1}^{k} \sum_{\boldsymbol{p} \in C_i} dist(\boldsymbol{p}, \boldsymbol{o}_i), \tag{8.2}$$

where E is the sum of the absolute error for all objects $\boldsymbol{p}$ in the data set, and $\boldsymbol{o_i}$ is the representative object of C_i. This is the basis for the ***k*-medoids method**, which groups n objects into k clusters by minimizing the absolute error (Eq. (8.2)).

When $k = 1$, we can find the exact median in $O(n^2)$ time. However, when k is a general positive number, the k-medoid problem is NP-hard.

The **Partitioning Around Medoids (PAM)** algorithm (see Fig. 8.4 later) is a popular realization of k-medoids clustering. It tackles the problem in an iterative, greedy way. Like the k-means algorithm, the initial representative objects (called seeds) are chosen arbitrarily. We consider whether replacing a representative object by a nonrepresentative object would improve the clustering quality. All the possible replacements are tried out. The iterative process of replacing representative objects by other objects continues until the quality of the resulting clustering cannot be improved by any replacement. This quality is measured by a cost function of the average dissimilarity between an object and the representative object of its cluster.

Specifically, let $\boldsymbol{o_1}, \ldots, \boldsymbol{o_k}$ be the current set of representative objects (i.e., medoids). To determine whether a nonrepresentative object, denoted by $\boldsymbol{o_{random}}$, is a good replacement for a current medoid



(a) Reassigned to o_i **(b)** Reassigned to o_{random} **(c)** No change **(d)** Reassigned to o_{random}

FIGURE 8.3

Four cases of the cost function for k-medoids clustering.

o_j $(1 \leq j \leq k)$, we calculate the distance from every object p to the closest object in the set $\{o_1, \ldots, o_{j-1}, o_{random}, o_{j+1}, \ldots, o_k\}$, and use the distance to update the cost function. The reassignments of objects to $\{o_1, \ldots, o_{j-1}, o_{random}, o_{j+1}, \ldots, o_k\}$ are simple. Suppose object p is currently assigned to a cluster represented by medoid o_j (Fig. 8.3a or b). Do we need to reassign p to a different cluster if o_j is being replaced by o_{random}? Object p needs to be reassigned to either o_{random} or some other cluster represented by o_i $(i \neq j)$, whichever is the closest. For example, in Fig. 8.3(a), p is closest to o_i and therefore is reassigned to o_i. In Fig. 8.3(b), however, p is closest to o_{random} and so is reassigned to o_{random}. What if, instead, p is currently assigned to a cluster represented by some other object o_i, $i \neq j$? Object p remains assigned to the cluster represented by o_i as long as p is still closer to o_i than to o_{random} (Fig. 8.3c). Otherwise, p is reassigned to o_{random} (Fig. 8.3d).

Each time a reassignment occurs, a difference in absolute error, E, is contributed to the cost function. Therefore the cost function calculates the *difference* in absolute-error value if a current representative object is replaced by a nonrepresentative object. The total cost of swapping is the sum of costs incurred by all nonrepresentative objects. If the total cost is negative, then o_j is replaced or swapped with o_{random} because the actual absolute-error E is reduced. If the total cost is positive, the current representative object, o_j, is considered acceptable, and nothing is changed in the iteration.

"Which method is more robust—k-means or k-medoids?" The k-medoids method is more robust than k-means in the presence of noise and outliers because a medoid is less influenced by outliers or other extreme values than a mean. However, the complexity of each iteration in the k-medoids algorithm is $O(k(n-k))$. For large values of n and k, such computation becomes very costly and much more costly than the k-means method. Both methods require the user to specify k, the number of clusters.

k-Modes: clustering nominal data

One limitation of the k-means method is that it can be applied only when the mean of a set of objects is defined. This may not be the case in some applications such as when data with nominal attributes is involved. The ***k*-modes method** is a variant of k-means, which extends the k-means paradigm to cluster nominal data by replacing the means of clusters with modes.

Recall that the mode for a set of data is the value that occurs most frequently in the set. In order to use modes in clustering, we need a new way to compute the distance between two objects. Given two

Algorithm: k-medoids. PAM, a k-medoids algorithm for partitioning based on medoid or central objects.

Input:

- k: the number of clusters,
- D: a data set containing n objects.

Output: A set of k clusters.
Method:

(1) arbitrarily choose k objects in D as the initial representative objects or seeds;
(2) **repeat**
(3) assign each remaining object to the cluster with the nearest representative object;
(4) randomly select a nonrepresentative object, o_{random};
(5) compute the total cost, S, of swapping representative object, o_j, with o_{random};
(6) **if** $S < 0$ **then** swap o_j with o_{random} to form the new set of k representative objects;
(7) **until** no change;

FIGURE 8.4

PAM, a k-medoids partitioning algorithm.

objects $\boldsymbol{x} = (x_1, \ldots, x_l)$ and $\boldsymbol{y} = (y_1, \ldots, y_l)$, we define the distance

$$dist(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{l} d(x_i, y_i), \tag{8.3}$$

where $d(x, y) = 1$ if $x \neq y$ and otherwise 0. With this change, the sum of squared error (Eq. (8.1)) remains valid, where $\boldsymbol{c_i}$ is the representative of cluster i.

The k-modes method works largely the same way as the k-means method. First, it selects k initial modes, one for each cluster. Second, it allocates an object to the cluster whose mode is the closest to the object using the distance function in Eq. (8.3). Third, it updates the mode of each cluster. For a cluster i and dimension j, the mode is updated to the most frequent value on the dimension of all objects assigned to this cluster. If there are more than one such a value, that is, if two or more values of the same frequency happen most frequently in the cluster, we can randomly choose one. The k-modes method iterates the object allocation and mode update steps until either the sum of squared error (Eq. (8.1)) stabilizes or a given number of iterations are conducted.

The k-means and the k-modes methods can be integrated to cluster data with mixed numeric and nominal values. This is known as the k-prototype method. On each dimension, according to whether it is a numeric attribute or a nominal attribute, we can use either the absolute error $dist(x - y)$ or the mode difference $d(x, y) = 1$ if $x \neq y$ and otherwise 0. Since numeric attributes may have a much larger range in absolute error than the mode difference on nominal attributes, we can associate with each dimension a weight balancing the effect of each dimension. You will have an opportunity to explore the details of the k-prototype method in the exercise.

Initialization in partitioning methods

It is interesting that by choosing the initial cluster centers carefully, we may be able to not only speed up the convergence of the k-means algorithm, but also guarantee the quality of the final clustering results. For example, the k-means++ algorithm chooses the initial centers in the following steps. First, it chooses one center uniformly at random from the objects in the data set. Iteratively, for each object

p other than the chosen centers, it chooses the object as the new center at random with probability proportional to $D(p)^2$, where $D(p)$ is the distance from p to the closest center that has already been chosen. The iteration continues until k centers are chosen.

Extensive experimental results have shown that the k-means++ algorithm can speed up the clustering process by a factor from 2 in most cases. Moreover, the k-means++ algorithm guarantees an approximation ratio of $O(\log k)$; that is, the within-cluster variation obtained by k-means++ is not more than $O(\log k)$ times larger than the global optimum.

Estimating the number of clusters

The necessity for users to specify k, the number of clusters, in advance can be seen as a disadvantage. The desired number of k is often dependent on the shape and scale of the distribution of points in a data set and the desired clustering resolution of the user. There have been studies on how to estimate a desired number of clusters. For example, given a data set of n objects, let $B(k)$ and $W(k)$ be the sum of squares of the distances between and within clusters, respectively, when there are k clusters. The *Calinski-Harabasz index* is defined by

$$CH(k) = \frac{\frac{B(k)}{k-1}}{\frac{W(k)}{n-k}}. \tag{8.4}$$

The number of clusters k can be estimated by maximizing the Calinski-Harabasz index.

Gap statistic is another method to estimate the number of clusters. The sum of the pairwise distances for all points in a cluster C_i is

$$SD_{C_i} = \sum_{\boldsymbol{p}, \boldsymbol{q} \in C_i} dist(\boldsymbol{p}, \boldsymbol{q}).$$

If the data set is divided into k clusters, define

$$W_k = \sum_{i=1}^{k} \frac{SD_{C_i}}{2|C_i|},$$

which is the pooled within-cluster sum of squares around the cluster means. The *gap statistic* is

$$Gap_n(k) = E_n^*\{\log(W_k)\} - \log(W_k), \tag{8.5}$$

where E_n^* is the expectation under a sample of size n from the reference distribution, that is, the distribution producing the data set to be clustered. We can choose the value k that maximizes the gap statistic as the estimation of number of clusters.

In Section 8.5.2, we will introduce additional methods to estimate the number of clusters.

Applying feature transformation

The k-means method employing the Euclidean distance or any metric measures in general can only output convex clusters. Here, a cluster is convex if for any two points a and b belonging to the cluster, every point between the two points on the line connecting a and b also belongs to the cluster. Moreover, the k-means method employing any metric measure can only detect clusters that are linearly separable. That is, two clusters can be separated by a linear hyperplane.

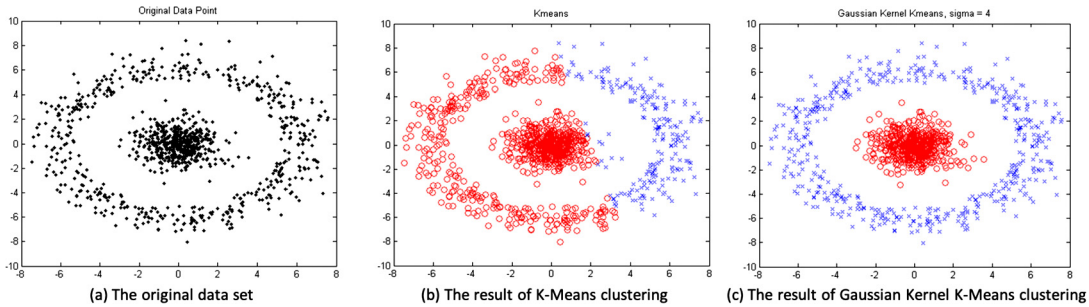


FIGURE 8.5

Concave and not linearly separable clusters can be detected by kernel k-means.

In many applications, clusters may not be convex or linearly separable. In Fig. 8.5(a), one can easily see that there are two clusters, the points at the center form a cluster, which is convex. The other points form another cluster in a "ring" shape. If we apply k-means on the data set, specify the number of clusters $k = 2$ and use the Euclidean distance, the output is in Fig. 8.5(b). As you can see, k-means cuts the data set into two parts using a line. However, those two parts do not match the visual intuition.

Can we still use the k-means method to find clusters that are concave and not linearly separable? Indeed, kernel k-means is such a method. The general idea of kernel k-means is to map the data points in the original input space to a feature space of higher dimensionality where the points belonging to the same cluster are close to each other in the feature space. Explicitly defining a space of high dimensionality and mapping the points into that space is subtle and may be costly. Instead, a convenient way is to apply a kernel function to measure the distance between points.

Recall that Section 7.3.2 introduces the concept of kernel functions. For example, using the Gaussian radial basis function (RBF) kernel, we can calculate the distance between two points $\mathbf{x}$ and $\mathbf{y}$ by

$$K(\mathbf{x}, \mathbf{y}) = e^{\frac{-\|\mathbf{x}-\mathbf{y}\|^2}{2\sigma^2}},$$

where $\|\mathbf{x} - \mathbf{y}\|^2$ is indeed the squared Euclidean distance between the two points, and σ is a free parameter. Clearly, the RBF kernel has the range between 0 and 1 and decreases with respect to the Euclidean distance.

How does a kernel function, such as the RBF kernel, transform the similarity among data points? Consider the five points in Fig. 8.6. The Euclidean distance matrix is

$$\begin{bmatrix} 0 & 5.66 & 5.66 & 5.66 & 5.66 \\ 5.66 & 0 & 8 & 11.31 & 8 \\ 5.66 & 8 & 0 & 8 & 11.31 \\ 5.66 & 11.31 & 8 & 0 & 8 \\ 5.66 & 8 & 11.31 & 8 & 0 \end{bmatrix}$$

where the value at the ith row and the jth column is the Euclidean distance between $\mathbf{x}_i$ and $\mathbf{x}_j$. Let $\sigma = 4$. We can apply the RBF kernel to the same five points. The corresponding RBF kernel similarity

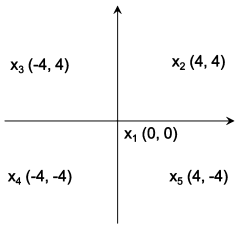


FIGURE 8.6

An example of five points.

matrix is

$$\begin{bmatrix} 0 & e^{-1} & e^{-1} & e^{-1} & e^{-1} \\ e^{-1} & 0 & e^{-2} & e^{-4} & e^{-2} \\ e^{-1} & e^{-2} & 0 & e^{-2} & e^{-4} \\ e^{-1} & e^{-4} & e^{-2} & 0 & e^{-2} \\ e^{-1} & e^{-2} & e^{-4} & e^{-2} & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0.37 & 0.37 & 0.37 & 0.37 \\ 0.37 & 1 & 0.135 & 0.02 & 0.135 \\ 0.37 & 0.135 & 1 & 0.135 & 0.02 \\ 0.37 & 0.02 & 0.135 & 1 & 0.135 \\ 0.37 & 0.135 & 0.02 & 0.135 & 1 \end{bmatrix}.$$

The magic here is that the RBF kernel indeed reduces the similarity between two points in a super-linear manner as the Euclidean distance between them increases. This nonlinear allocation of similarity enables k-means to assemble clusters using points that are not linearly separable and form clusters that are not convex. For example, if we apply the RBF kernel and k-means on the data set in Fig. 8.5(a), the output is Fig. 8.5(c), where the points in red (gray in print version) form a cluster and the points in blue (dark gray in print version) form another cluster. The output matches the visual intuition nicely.

8.3 Hierarchical methods

While partitioning methods meet the basic clustering requirement of organizing a set of objects into a number of exclusive groups, in some situations we may want to partition our data into groups at different levels, or in general a hierarchy. A **hierarchical clustering method** works by grouping data objects into a hierarchy or "tree" of clusters.

In this section, you will study hierarchical clustering methods. Section 8.3.1 begins with a discussion about the basic concepts of hierarchical clustering. Then, Section 8.3.2 introduces the agglomerative, bottom-up approaches for hierarchical clustering. Section 8.3.3 presents the divisive, top-down approaches. Hierarchical clustering methods may be combined with other methods. Section 8.3.4 discusses BIRCH, a scalable hierarchical clustering method for large amounts of numeric data. Last, Section 8.3.5 describes the probabilistic hierarchical clustering methods.

8.3.1 Basic concepts of hierarchical clustering

Representing data objects in the form of a hierarchy is useful for data summarization and visualization. For example, as a manager of human resources in a company, you may organize your employees

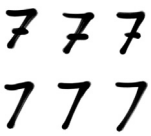


FIGURE 8.7

Two clusters of handwritten digit 7s.

into major groups such as executives, managers, and staff. You can further partition these groups into smaller subgroups. For instance, the general group of staff can be further divided into subgroups of senior officers, officers, and trainees. All these groups form a hierarchy. We can easily summarize or characterize the data that is organized into a hierarchy, which can be used to find, say, the average salary of managers and of officers.

Consider handwritten character recognition as another example. A set of handwriting samples may be first partitioned into general groups where each group corresponds to a unique character. Some groups can be further partitioned into subgroups since a character may be written in multiple substantially different ways. For example, Fig. 8.7 shows a group of handwritten digit 7s. The group can be further divided into two subgroups, the first row being a subgroup where a short horizontal line is used in each writing, and the second row being another subgroup. If necessary, the hierarchical partitioning can be continued recursively until a desired granularity is reached.

In the previous examples, although we partition the data hierarchically, we do not assume that the data has a hierarchical structure. Our use of a hierarchy here is just to summarize and represent the underlying data in a compressed way. Such a hierarchy is particularly useful for data visualization.

Alternatively, in some applications we may believe that the data bear an underlying hierarchical structure that we want to discover. For example, hierarchical clustering may uncover a hierarchy for the employees in a company structured on, say, salary. In the study of biological evolution, hierarchical clustering may group living creatures according to their biological features to uncover evolutionary paths, which are a hierarchy of species. As another example, grouping configurations of a strategic game (e.g., chess or checkers) in a hierarchical way may help to develop game strategies that can be used to train players.

A hierarchical clustering method can be either *agglomerative* or *divisive*, depending on whether the hierarchical decomposition is formed in a bottom-up (merging) or top-down (splitting) fashion. Let us have a closer look at these strategies.

An **agglomerative hierarchical clustering method** uses a bottom-up strategy. It typically starts by letting each object form its own cluster and iteratively merges clusters into larger and larger clusters, until all the objects are in a single cluster or certain termination conditions are satisfied. The single cluster becomes the hierarchy's root. For the merging step, it finds the two clusters that are closest to each other (according to some similarity measure) and combines the two to form one cluster. Because two clusters are merged per iteration, where each cluster contains at least one object, an agglomerative method requires at most n iterations.

A **divisive hierarchical clustering method** employs a top-down strategy. It starts by placing all objects in one cluster, which is the hierarchy's root. It then divides the root cluster into several smaller subclusters and recursively partitions those clusters into smaller ones. The partitioning process contin-

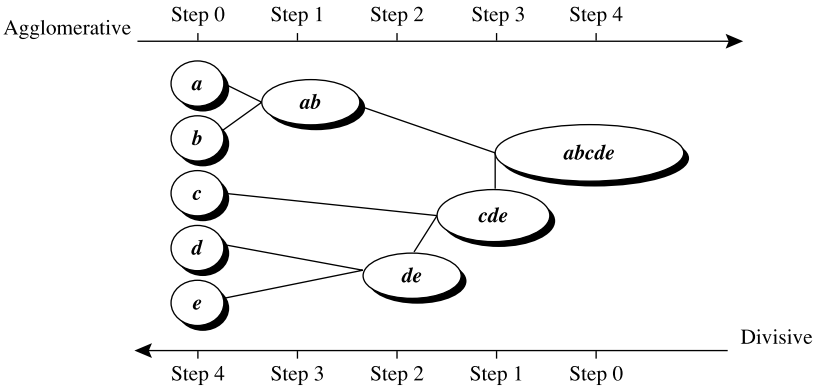


FIGURE 8.8

Agglomerative and divisive hierarchical clustering on data objects $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}, \boldsymbol{e}\}$.

ues until each cluster at the lowest level is coherent enough—either containing only one object, or the objects within a cluster are sufficiently similar to each other.

In either agglomerative or divisive hierarchical clustering, a user can specify the desired number of clusters as a termination condition.

Example 8.3. Agglomerative vs. divisive hierarchical clustering. Fig. 8.8 shows the application of an agglomerative hierarchical clustering method and a divisive hierarchical clustering method on a data set of five objects, $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}, \boldsymbol{e}\}$. Initially, the agglomerative method places each object into a cluster of its own. The clusters are then merged step-by-step according to some criterion. For example, clusters C_1 and C_2 may be merged if an object in C_1 and an object in C_2 form the minimum Euclidean distance between any two objects from different clusters. This is a **single-linkage** approach in that each cluster is represented by all the objects in the cluster, and the similarity between two clusters is measured by the similarity of the *closest* pair of data points belonging to different clusters. The cluster-merging process repeats until all the objects are eventually merged to form one cluster.

The divisive method proceeds in the contrasting way. All the objects are used to form one initial cluster. The cluster is split according to some principle such as the maximum Euclidean distance between the closest neighboring objects in the cluster. The cluster-splitting process repeats until, eventually, each new cluster contains only a single object. □

The selection of merge or split points is critical for hierarchical clustering methods, because once a group of objects is merged or split, the process at the next step will operate on the newly generated clusters. It will neither undo what was done previously, nor perform object swapping between clusters. Thus merge or split decisions, if not well chosen, may lead to low-quality clusters. Moreover, the methods do not scale well because each decision of merge or split needs to examine and evaluate many objects or clusters.

A promising direction for improving the clustering quality of hierarchical methods is to integrate hierarchical clustering with other clustering techniques, resulting in **multiple-phase clustering**. We introduce BIRCH as a representative method in Section 8.3.4. BIRCH begins by partitioning objects

hierarchically using tree structures, where the leaf or low-level nonleaf nodes can be viewed as "microclusters" depending on the resolution scale. It then applies other clustering algorithms to perform macroclustering on the microclusters.

There are several orthogonal ways to categorize hierarchical clustering methods. For instance, they may be categorized into *deterministic* methods and *probabilistic* methods. Agglomerative, divisive, and multiphase methods are *deterministic*, since they consider data objects as deterministic and compute clusters according to the deterministic distances between objects. Probabilistic methods use probabilistic models to capture clusters and measure the quality of clusters by the fitness of models. We discuss probabilistic hierarchical clustering in Section 8.3.5.

8.3.2 Agglomerative hierarchical clustering

In this section, we discuss some important issues in agglomerative hierarchical clustering methods.

Similarity measures in hierarchical clustering

How can we choose which objects and clusters to merge in an agglomerative step? The core is to measure the similarity between two clusters, where each cluster is generally a set of objects.

Four widely used measures for distance between clusters are as follows, where $|\boldsymbol{p} - \boldsymbol{p'}|$ is the distance between two objects or points, $\boldsymbol{p}$ and $\boldsymbol{p'}$; $\boldsymbol{m_i}$ is the mean for cluster C_i; and n_i is the number of objects in C_i. They are also known as *linkage measures*.

$$\textbf{Minimum distance}: \quad dist_{min}(C_i, C_j) = \min_{\boldsymbol{p} \in C_i, \boldsymbol{p'} \in C_j} \{\|\boldsymbol{p} - \boldsymbol{p'}\|\} \tag{8.6}$$

$$\textbf{Maximum distance}: \quad dist_{max}(C_i, C_j) = \max_{\boldsymbol{p} \in C_i, \boldsymbol{p'} \in C_j} \{\|\boldsymbol{p} - \boldsymbol{p'}\|\} \tag{8.7}$$

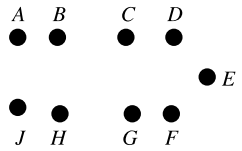
$$\textbf{Mean distance}: \quad dist_{mean}(C_i, C_j) = \|\boldsymbol{m_i} - \boldsymbol{m_j}\| \tag{8.8}$$

$$\textbf{Average distance}: \quad dist_{avg}(C_i, C_j) = \frac{1}{n_i n_j} \sum_{\boldsymbol{p} \in C_i, \boldsymbol{p'} \in C_j} \|\boldsymbol{p} - \boldsymbol{p'}\| \tag{8.9}$$

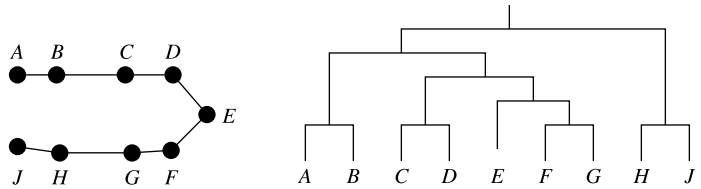
When an algorithm uses the *minimum distance*, $d_{min}(C_i, C_j)$, to measure the distance between clusters, it is sometimes called a **nearest-neighbor clustering algorithm** or **single-linkage algorithm**. If we view the data points as nodes of a graph, with edges forming a path between the nodes in a cluster, then the merging of two clusters, C_i and C_j, corresponds to adding an edge between the nearest pair of nodes in C_i and C_j.

When an algorithm uses the *maximum distance*, $d_{max}(C_i, C_j)$, to measure the distance between clusters, it is sometimes called a **farthest-neighbor clustering algorithm** or **complete-linkage algorithm**. By viewing data points as nodes of a graph, with edges linking nodes, we can think of each cluster as a *complete* subgraph, that is, with edges connecting all the nodes in the clusters. The distance between two clusters is determined by the most distant nodes in the two clusters.

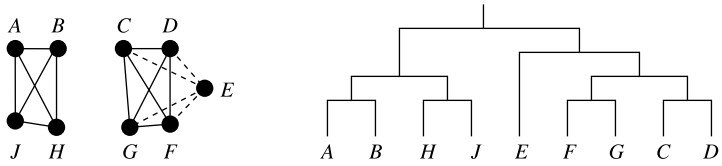
Similar to the situation in the k-means method, single-linkage and complete-linkage methods are sensitive to outliers. The use of *mean* or *average distance* is a compromise between the minimum and maximum distances and overcomes the outlier sensitivity problem. Whereas the *mean distance* is the simplest to compute, the *average distance* is advantageous in that it can handle categoric data and



(a) Data set



(b) Clustering using single linkage



(c) Clustering using complete linkage

FIGURE 8.9

Hierarchical clustering using single and complete linkages.

numeric data. The computation of the mean vector for categoric data can be difficult or impossible to define.

Example 8.4. Single vs. complete linkages. Let us apply hierarchical clustering to the data set of Fig. 8.9(a). Fig. 8.9(b) shows the hierarchy of clusters using single-linkage. Fig. 8.9(c) shows the case using complete linkage, where the edges between clusters $\{A, B, J, H\}$ and $\{C, D, G, F, E\}$ are omitted for ease of presentation. This example shows that by using single linkages we can find hierarchical clusters defined by local proximity, whereas complete linkage tends to find clusters opting for global closeness. □

There are variations of the four essential linkage measures just discussed. For example, we can measure the distance between two clusters by the distance between the centroids (i.e., the central objects) of the clusters.

Connecting agglomerative hierarchical clustering and partitioning methods

Are there any connections between (agglomerative) hierarchical clustering and partitioning methods (Section 8.2)? Partitioning methods use the sum of squared errors (SSE) (Eq. (8.1)) to measure the compactness and quality of a possible clustering, that is a partitioning of points into clusters. Heuristically,

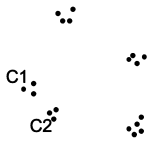


FIGURE 8.10

Ward's criterion.

agglomerative hierarchical clustering methods may also use the sum of squared errors (SSE) to guide the selection of clusters to merge.

For a data set of n points, if we set the number of clusters to n, a partitioning method naturally assigns each point into a cluster. This corresponds to the starting point of agglomerative hierarchical clustering. When we merge clusters in agglomerative clustering, we reduce the number of clusters. Which two clusters should we choose to merge? Heuristically, we may want to merge two clusters so that the resulting clustering also minimizes the sum of squared errors (SSE) (Eq. (8.1)), which is used as the criterion in partitioning methods like k-means.

Consider the five clusters in Fig. 8.10 as an illustrative example. Suppose we want to merge two of them into one so that we can have a hierarchy of clusters. Among all the possible pairs of clusters, merging C_1 and C_2 minimizes the SSE, and thus C_1 and C_2 should be merged in the next step in building the hierarchy.

The above intuition connecting agglomerative hierarchical clustering and partitioning methods gives us an alternative way to measure the similarity between two clusters. We can look at the increase of the SSE (Eq. (8.1)) if the two clusters are merged into one, the smaller the better. This is formulated by J.H. Ward and thus is known as **Ward's criterion**.

Suppose two disjoint clusters C_i and C_j are merged, and $m_{(ij)}$ is the mean of the new cluster. Then, Ward's criterion is defined as

$$\begin{aligned} W(C_i, C_j) &= \sum_{\mathbf{x} \in C_i \cup C_j} \|\mathbf{x} - \boldsymbol{m}_{(ij)}\|^2 - \sum_{\mathbf{x} \in C_i} \|\mathbf{x} - \boldsymbol{m}_i\|^2 - \sum_{\mathbf{x} \in C_j} \|\mathbf{x} - \boldsymbol{m}_j\|^2 \\ &= \frac{n_i n_j}{n_i + n_j} \|\boldsymbol{m}_i - \boldsymbol{m}_j\|^2. \end{aligned}$$

The Lance-Williams algorithm

We discuss a few different proximity measures for clusters in agglomerative hierarchical clustering, is there a way to generalize them? Indeed, the **Lance-Williams formula** generalizes different measures in a uniform way. Suppose two exclusive clusters C_i and C_j are merged. We need to specify the distance between the merged cluster, denoted by $C_{(ij)}$, and every other cluster C_k. The similarity between the merged clusters $C_{(ij)}$ and cluster C_k is given by

$$d(C_{(ij)}, C_k) = \alpha_i d(C_i, C_k) + \alpha_j d(C_j, C_k) + \beta d(C_i, C_j) + \gamma |d(C_i, C_k) - d(C_j, C_k)|,$$

where α_i, α_j, β, and γ are parameters that together with the similarity function $d(C_i, C_j)$ determine the hierarchical clustering algorithm. As shown in the formula, the similarity between $C_{(ij)}$ and C_k is decided by four terms. The first two terms are the similarities between C_i and C_j to C_k, respectively.

The third term relies on the similarity between C_i and C_j. The last term represents how the difference of the original similarities between C_i and C_j to C_k may contribute to the new similarity.

For example, in the exercise, you are asked to verify that the single-linkage method is equivalent to taking $\alpha_i = \alpha_j = 0.5$, $\beta = 0$, and $\gamma = -0.5$ in the Lance-Williams formula. Record that in the single link method, the similarity between two clusters is determined by the similarity between of the closest pair of data points belonging to different clusters. Thus the above parameters simply pick the smaller one between $d(C_i, C_k)$ and $d(C_j, C_k)$. You can also verify that the complete-linkage method is equivalent to $\alpha_i = \alpha_j = 0.5$, $\beta = 0$, and $\gamma = 0.5$. Moreover, to implement the Ward's criterion, we can take $\alpha_i = \frac{n_i+n_k}{n_i+n_j+n_k}$, $\alpha_j = \frac{n_j+n_k}{n_i+n_j+n_k}$, $\beta = -\frac{n_k}{n_i+n_j+n_k}$, and $\gamma = 0$ in the Lance-Williams formula.

Using the Lance-Williams formula, the *Lance-Williams algorithm* generalizes agglomerative hierarchical clustering. It takes an agglomerative way and minimizes the sum of distance in each iteration until all points are merged into one cluster.

8.3.3 Divisive hierarchical clustering

A divisive hierarchical clustering method partitions a set of objects step by step into clusters. To design a divisive hierarchical clustering method, there are three important issues to consider.

First, a set of objects can be split in many different ways. A splitting criterion is needed to determine which splitting is the best. Technically, given two splittings, the splitting criterion should be able to tell which one is better. For example, the SSE (Eq. (8.1)) may be used on numeric data. If two splittings have the same number of clusters, then the one with a less SSE value is preferred. On nominal data, the Gini index (Chapter 6) may be used. The choice of splitting criteria is one of the most important decisions in designing a divisive hierarchical clustering method, since it determines what clustering results that the method may lead to.

Second, when we decide to split a cluster, we need to design a splitting method. While in general the splitting method should optimize the splitting criterion, one major consideration is the computational cost. For example, enumerating all possible splittings and finding the best one is likely computationally prohibitive. Thus some heuristic or approximate methods, such as bisecting k-means, that is, setting $k = 2$, may be used.

Third, in the middle of divisive hierarchical clustering, there are in general multiple clusters. Then, which cluster should be split next? An intuitive idea is to choose the "loosest" cluster. More concretely, we may compute the average SSE of each cluster, $E_{C_i} = \frac{1}{|C_i|}\sum_{\mathbf{x}\in C_i}(\mathbf{x} - \boldsymbol{m_i})^2$, and choose the one with the largest average SSE to split.

The minimum spanning tree–based approach

Let us use the minimum spanning tree–based approach to illustrate the basic idea of divisive hierarchical clustering. In a weighted graph G, a minimum spanning tree is an acyclic subgraph that contains all nodes in G, and the sum of edge weights of the tree is minimized. For example, consider the weighted graph in Fig. 8.11, where the edges of weights up to nine are shown in the figure, and the edges of weights over nine are omitted. The minimum spanning tree consists of the edges in solid lines, whereas the edges in dashed lines and those omitted are not included in the minimum spanning tree. Minimum spanning tree can be computed using, for example, Prim's algorithm and Kruskal's algorithm.

Given a set of points, we can construct a weighted graph such that each point is represented by a node in the graph, and the distance between two points is the weight of the edge connecting the two

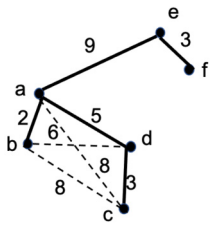


FIGURE 8.11

A weighted graph and a minimum spanning tree.

corresponding nodes in the graph. Then, we can compute the minimum spanning tree of the weighted graph. Intuitively, the minimum spanning tree can be regarded as the most compact way that the points are connected into one cluster. In general, in the process of divisive hierarchical clustering based on minimum spanning tree, every cluster is a subset of nodes and is represented by the minimum spanning tree, which is a subtree of the minimum spanning tree of the whole data set. The splitting criterion is the total weights of all the edges in the spanning tree(s) of all the clusters, the smaller the better.

Based on this spanning tree, we can progressively divide the set of points in one cluster into smaller clusters. Suppose we want to conduct bisecting splitting, that is, every time we split one cluster into two smaller clusters. At each step, we consider all edges in the spanning trees of the current clusters and delete the edge of the largest weight. Deleting an edge in a tree divides one cluster into two. Thus the splitting method is to divide a cluster by deleting the edge in the minimum spanning tree of the largest weight.

For example, consider a set of points $\{a, b, c, d, e, f\}$ as shown in Fig. 8.11. A weighted edge and the corresponding distance are plotted in the figure if the distance between two points is smaller than 10. Based on the minimum spanning tree method, the edge (a, e), which has the largest weight in the minimum spanning tree, is first deleted, which divides the data set into two clusters, $\{a, b, c, d\}$ and $\{e, f\}$. Next, by deleting edge (a, d), which has the largest weight in the remaining minimum spanning trees, the cluster $\{a, b, c, d\}$ is split into two smaller clusters $\{a, b\}$ and $\{c, d\}$. The process continues until each cluster has only one point and all the edges in the minimum spanning tree are deleted.

Dendrogram

A tree structure called a **dendrogram** is commonly used to represent the process of hierarchical clustering. It shows how objects are grouped together (in an agglomerative method) or partitioned (in a divisive method) step-by-step. Fig. 8.12 shows a dendrogram for the five objects presented in Fig. 8.8, where $l = 0$ shows the five objects as singleton clusters at level 0. At $l = 1$, objects $\boldsymbol{a}$ and $\boldsymbol{b}$ are grouped together to form the first cluster, and they stay together at all subsequent levels. We can also use a vertical axis to show the similarity scale between clusters. For example, when the similarity of two groups of objects, $\{\boldsymbol{a}, \boldsymbol{b}\}$ and $\{\boldsymbol{c}, \boldsymbol{d}, \boldsymbol{e}\}$, is roughly 0.16, they are merged together to form a single cluster.

A complete dendrogram shows every data point as a node at the leaf level and the whole data set as one cluster at the root. In practice, however, too small clusters may not be very meaningful and too many such small clusters can be overwhelming. Thus a data analyst often shows and considers only the portion close to the root of a dendrogram. Moreover, when the graph contains a sufficiently small

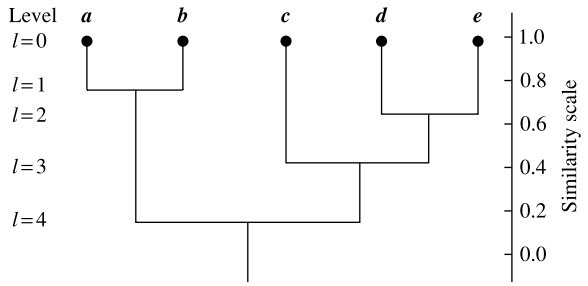


FIGURE 8.12

Dendrogram representation for hierarchical clustering of data objects $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}, \boldsymbol{e}\}$.

number of clusters, further merging them into even bigger ones may counter the objective of clustering analysis. Therefore the very root part of a dendrogram may also be ignored in analysis.

8.3.4 BIRCH: scalable hierarchical clustering using clustering feature trees

There are two major difficulties in the agglomerative and divisive hierarchical clustering methods discussed so far. First, all those methods cannot revisit any merge or split decisions made before. Thus an improper decision based on limited information may lead to low quality final clustering results. Moreover, scalability is a major bottleneck, since each merge or split needs to examine many possible options. To overcome those difficulties, we may conduct hierarchical clustering in multiple phases, so that clustering results can be improved over phases. Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH) is such a method and is designed for clustering a large amount of numeric data by integrating hierarchical clustering (at the initial *microclustering* stage) and other clustering methods such as iterative partitioning (at the later *macroclustering* stage).

BIRCH uses the notions of *clustering feature* to summarize a cluster, and *clustering feature tree* (*CF-tree*) to represent a cluster hierarchy. These structures help the clustering method achieve good speed and scalability in large or even streaming databases and also make it effective for incremental and dynamic clustering of incoming objects.

Consider a cluster of n d-D data objects or points. The **clustering feature** (**CF**) of the cluster is a 3-D vector summarizing information about clusters of objects. It is defined as

$$\boldsymbol{CF} = \langle n, \boldsymbol{LS}, SS \rangle, \tag{8.10}$$

where $\boldsymbol{LS}$ is the linear sum of the n points (i.e., $\boldsymbol{LS} = \sum_{i=1}^{n} \boldsymbol{x_i}$) and SS is the square sum of the data points (i.e., $SS = \sum_{i=1}^{n} \|\boldsymbol{x}_i\|^2$).

A clustering feature is essentially a summary of the statistics for the given cluster. Using a clustering feature, we can easily derive many useful statistics of a cluster. For example, the cluster's centroid, $\boldsymbol{x_0}$, radius, R, and diameter, D, are

$$\boldsymbol{x}_0 = \frac{\sum_{i=1}^{n} \boldsymbol{x}_i}{n} = \frac{\boldsymbol{LS}}{n}, \tag{8.11}$$

$$R = \sqrt{\frac{\sum_{i=1}^{n}(\boldsymbol{x_i} - \boldsymbol{x_0})^2}{n}} = \sqrt{\frac{SS}{n} - (\frac{\|\boldsymbol{LS}\|}{n})^2}, \tag{8.12}$$

$$D = \sqrt{\frac{\sum_{i=1}^{n}\sum_{j=1}^{n}(\boldsymbol{x_i} - \boldsymbol{x_j})^2}{n(n-1)}} = \sqrt{\frac{2nSS - 2\|\boldsymbol{LS}\|^2}{n(n-1)}}. \tag{8.13}$$

Here, R is the average distance from member objects to the centroid, and D is the average pairwise distance within a cluster. Both R and D reflect the tightness of the cluster around the centroid.

Summarizing a cluster using the clustering feature can avoid storing the detailed information about individual objects or points. Instead, we only need a constant size of space to store the clustering feature. This is the key to the efficiency of BIRCH in space. Moreover, clustering features are *additive*. That is, for two disjoint clusters, C_1 and C_2, with the clustering features $\boldsymbol{CF}_1 = \langle n_1, \boldsymbol{LS}_1, SS_1\rangle$ and $\boldsymbol{CF}_2 = \langle n_2, \boldsymbol{LS}_2, SS_2\rangle$, respectively, the clustering feature for the cluster that formed by merging C_1 and C_2 is simply

$$\boldsymbol{CF}_1 + \boldsymbol{CF}_2 = \langle n_1 + n_2, \boldsymbol{LS}_1 + \boldsymbol{LS}_2, SS_1 + SS_2\rangle. \tag{8.14}$$

Example 8.5. Clustering feature. Suppose there are three points, (2, 5), (3, 2), and (4, 3), in a cluster, C_1. The clustering feature of C_1 is

$$\boldsymbol{CF}_1 = \langle 3, (2+3+4, 5+2+3), (2^2+3^2+4^2) + (5^2+2^2+3^2)\rangle = \langle 3, (9, 10), 67\rangle.$$

Suppose that C_1 is disjoint to a second cluster, C_2, where $\boldsymbol{CF}_2 = \langle 3, (35, 36), 857\rangle$. The clustering feature of a new cluster, C_3, that is formed by merging C_1 and C_2, is derived by adding $\boldsymbol{CF}_1$ and $\boldsymbol{CF}_2$. That is,

$$\boldsymbol{CF}_3 = \langle 3+3, (9+35, 10+36), 67+857\rangle = \langle 6, (44, 46), 924\rangle.$$

□

A **CF-tree** is a height-balanced tree that stores the clustering features for a hierarchical clustering. An example is shown in Fig. 8.13. By definition, a nonleaf node in a tree has descendants or "children." The nonleaf nodes store sums of the CFs of their children and thus summarize clustering information about their children. A CF-tree has two parameters: *branching factor*, B, and *threshold*, T. The branching factor specifies the maximum number of children per nonleaf node. The threshold parameter specifies the maximum diameter of subclusters stored at the leaf nodes of the tree. These two parameters implicitly control the size of the resulting CF-tree.

Given a limited amount of main memory, an important consideration in BIRCH is to minimize the time required for input/output (I/O). BIRCH applies a *multiphase* clustering technique: A single scan of the data set yields a basic, good clustering, and one or more additional scans can optionally be used to further improve the quality. The primary phases are as follows:

- **Phase 1:** BIRCH scans the database to build an initial in-memory CF-tree, which can be viewed as a multilevel compression of the data that tries to preserve the inherent clustering structure of the data.

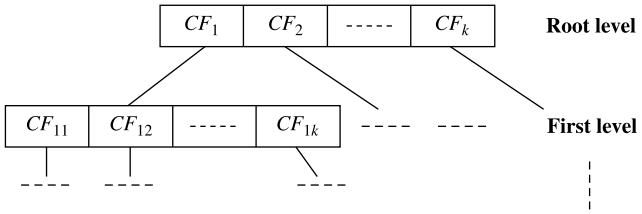


FIGURE 8.13

CF-tree structure.

- **Phase 2:** BIRCH applies a (selected) clustering algorithm to cluster the leaf nodes of the CF-tree, which removes sparse clusters as outliers and groups dense clusters into larger ones.

For Phase 1, the CF-tree is built dynamically as objects are inserted. Thus the method is incremental. An object is inserted into the closest leaf entry (subcluster). If the diameter of the subcluster stored in the leaf node after insertion is larger than the threshold value, then the leaf node and possibly other nodes are split. After the insertion of the new object, information about the object is passed toward the root of the tree. The size of the CF-tree can be changed by modifying the threshold. If the size of the memory that is needed for storing the CF-tree is larger than the size of the main memory available, then a larger threshold value can be specified and the CF-tree is rebuilt.

The rebuild process is performed by building a new tree from the leaf nodes of the old tree. Thus the process of rebuilding the tree is done without the necessity of rereading all the objects or points. This is similar to the insertion and node split in the construction of B+-trees. Therefore for building the tree, data have to be read just once. Some heuristics and methods have been introduced to deal with outliers and improve the quality of CF-trees by additional scans of the data. Once the CF-tree is built, any clustering algorithm, such as a typical partitioning algorithm, can be used with the CF-tree in Phase 2.

"How effective is BIRCH?" The time complexity of the algorithm is $O(n)$, where n is the number of objects to be clustered. Experiments have shown the linear scalability of the algorithm with respect to the number of objects, and good quality of clustering of data. However, since each node in a CF-tree can hold only a limited number of entries due to its size, a CF-tree node does not always correspond to what a user may consider a natural cluster. Moreover, if the clusters are not spherical in shape, BIRCH does not perform well because it uses the notion of radius or diameter to control the boundary of a cluster.

The ideas of clustering features and CF-trees have been applied beyond BIRCH. The ideas have been borrowed by many others to tackle problems of clustering streaming and dynamic data.

8.3.5 Probabilistic hierarchical clustering

Hierarchical clustering methods using linkage measures tend to be easy to understand and are often efficient in clustering. They are commonly used in many clustering analysis applications. However, hierarchical clustering methods can suffer from several drawbacks. First, choosing a good distance measure for hierarchical clustering is often far from trivial. Second, to apply such a method, the data

objects cannot have any missing attribute values. In the case where data is partially observed (i.e., some attribute values of some objects are missing), it is not easy to apply a hierarchical clustering method because the distance computation cannot be conducted. Third, most of the hierarchical clustering methods are heuristic and search locally at each step for a good merging/splitting decision. Consequently, the optimization goal of the resulting cluster hierarchy can be unclear.

Probabilistic hierarchical clustering aims to overcome some of these disadvantages by using probabilistic models to measure distances between clusters.

One way to look at the clustering problem is to regard the set of data objects to be clustered as a sample of the underlying data generation mechanism to be analyzed or, formally, the *generative model*. For example, when we conduct clustering analysis on a set of marketing surveys, we assume that the surveys collected are a sample of the opinions of all possible customers. Here, the data generation mechanism is a probability distribution of opinions with respect to different customers, which cannot be obtained directly and completely. The task of clustering is to estimate the generative model as accurately as possible using the observed data objects to be clustered.

In practice, we can assume that the data generative models adopt common distribution functions, such as Gaussian distribution or Bernoulli distribution, which are governed by parameters. The task of learning a generative model is then reduced to finding the parameter values for which the model best fits the observed data set.

Example 8.6. Generative model. Suppose we are given a set of 1-D points $X = \{x_1, \ldots, x_n\}$ for clustering analysis. Let us assume that the data points are generated by a Gaussian distribution,

$$\mathcal{N}(\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{8.15}$$

where the parameters are μ (the mean) and σ^2 (the variance).

The probability that a point $x_i \in X$ is then generated by the model is

$$P(x_i|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}. \tag{8.16}$$

Consequently, the likelihood that the data set X observed is generated by the model is

$$L(\mathcal{N}(\mu, \sigma^2) : X) = P(X|\mu, \sigma^2) = \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}. \tag{8.17}$$

The task of learning the generative model is to find the parameters μ and σ^2 such that the likelihood $L(\mathcal{N}(\mu, \sigma^2) : X)$ is maximized, that is, finding

$$\mathcal{N}(\mu_0, \sigma_0^2) = \arg\max\{L(\mathcal{N}(\mu, \sigma^2) : X)\}, \tag{8.18}$$

where $\max\{L(\mathcal{N}(\mu, \sigma^2) : X)\}$ is called the *maximum likelihood.* □

Given a set of objects, the quality of a cluster formed by all the objects can be measured by the maximum likelihood. For a set of objects partitioned into m clusters $C_1, \ldots, C_m$. Then the quality can

be measured by

$$Q(\{C_1, \ldots, C_m\}) = \prod_{i=1}^{m} P(C_i), \tag{8.19}$$

where $P()$ is the maximum likelihood. To calculate $P(C_i)$, we can fit each cluster C_i $(1 \leq i \leq m)$ by a generative model M_i, and estimate the probability by $P(C_i) = \prod_{x \in C_i} P(x|M_i)$. If we merge two clusters, C_{j_1} and C_{j_2}, into a cluster, $C_{j_1} \cup C_{j_2}$, then, the change in quality of the overall clustering is

$$\begin{aligned} &Q((\{C_1, \ldots, C_m\} - \{C_{j_1}, C_{j_2}\}) \cup \{C_{j_1} \cup C_{j_2}\}) - Q(\{C_1, \ldots, C_m\}) \\ &= \frac{\prod_{i=1}^{m} P(C_i) \cdot P(C_{j_1} \cup C_{j_2})}{P(C_{j_1})P(C_{j_2})} - \prod_{i=1}^{m} P(C_i) \\ &= \prod_{i=1}^{m} P(C_i) \left(\frac{P(C_{j_1} \cup C_{j_2})}{P(C_{j_1})P(C_{j_2})} - 1 \right). \end{aligned} \tag{8.20}$$

When choosing to merge two clusters in hierarchical clustering, $\prod_{i=1}^{m} P(C_i)$ is constant for any pair of clusters. Therefore given clusters C_1 and C_2, the dissimilarity between them can be measured by

$$dist(C_1, C_2) = -\log \frac{P(C_1 \cup C_2)}{P(C_1)P(C_2)}. \tag{8.21}$$

A probabilistic hierarchical clustering method can adopt the agglomerative clustering framework, but use probabilistic models (Eq. (8.21)) to measure the similarity between clusters.

Upon close observation of Eq. (8.20), we see that merging two clusters may not always lead to an improvement in clustering quality, that is, $\frac{P(C_{j_1} \cup C_{j_2})}{P(C_{j_1})P(C_{j_2})}$ may be less than 1. For example, assume that Gaussian distribution functions are used in the model of Fig. 8.14. Although merging clusters C_1 and C_2 results in a cluster that better fits a Gaussian distribution, merging clusters C_3 and C_4 lowers the clustering quality because no Gaussian functions can fit the merged cluster well.

Based on this observation, a probabilistic hierarchical clustering scheme can start with one cluster per object and merge two clusters, C_i and C_j, if the distance between them is negative. In each iteration, we try to find C_i and C_j so as to maximize $\log \frac{P(C_i \cup C_j)}{P(C_i)P(C_j)}$. The iteration continues as long as $\log \frac{P(C_i \cup C_j)}{P(C_i)P(C_j)} > 0$, that is, as long as there is an improvement in clustering quality. The pseudocode is given in Fig. 8.15.

Probabilistic hierarchical clustering methods are easy to understand and generally have the same efficiency as agglomerative hierarchical clustering methods; in fact, they share the same framework. Probabilistic models are more interpretable, but sometimes less flexible than distance metrics. Probabilistic models can handle partially observed data. For example, given a multidimensional data set where some objects have missing values on some dimensions, we can learn a Gaussian model on each dimension independently using the observed values on the dimension. The resulting cluster hierarchy accomplishes the optimization goal of fitting data to the selected probabilistic models.

A drawback of using probabilistic hierarchical clustering is that it outputs only one hierarchy with respect to a chosen probabilistic model. It cannot handle the uncertainty of cluster hierarchies. Given a

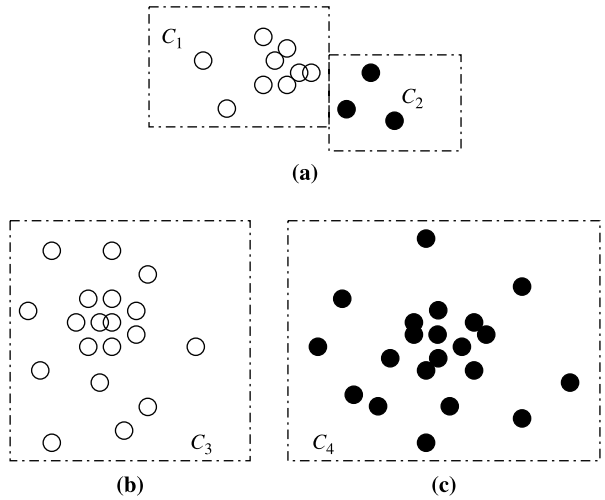


FIGURE 8.14

Merging clusters in probabilistic hierarchical clustering: (a) Merging clusters C_1 and C_2 leads to an increase in overall cluster quality, but merging clusters (b) C_3 and (c) C_4 does not.

Algorithm: A probabilistic hierarchical clustering algorithm.

Input:

- $D = \{\boldsymbol{o_1}, \ldots, \boldsymbol{o_n}\}$: a data set containing n objects;

Output: A hierarchy of clusters.
Method:

(1) **create** a cluster for each object $C_i = \{\boldsymbol{o_i}\}$, $1 \leq i \leq n$;
(2) **for** $i = 1$ to n
(3) **find** pair of clusters C_i and C_j such that $C_i, C_j = \arg\max_{i \neq j} \log \frac{P(C_i \cup C_j)}{P(C_i)P(C_j)}$;
(4) **if** $\log \frac{P(C_i \cup C_j)}{P(C_i)P(C_j)} > 0$ **then** merge C_i and C_j;
(5) **else** stop;

FIGURE 8.15

A probabilistic hierarchical clustering algorithm.

data set, there may exist multiple hierarchies that fit the observed data. Neither algorithmic approaches nor probabilistic approaches can find the distribution of such hierarchies. Recently, Bayesian tree-structured models have been developed to handle such problems. Bayesian and other sophisticated probabilistic clustering methods are considered advanced topics and are not covered in this book.

8.4 Density-based and grid-based methods

Most of the partitioning and hierarchical methods are designed to find spherical-shaped clusters. They have difficulty finding clusters of arbitrary shape such as the "S" shape and oval clusters in Fig. 8.16.

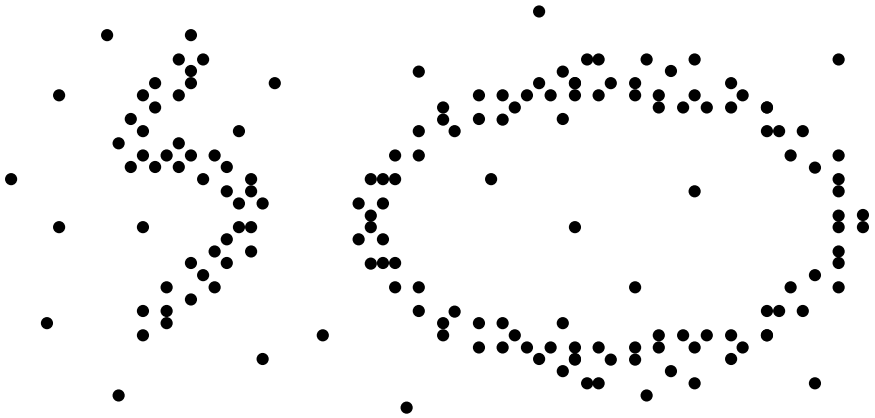


FIGURE 8.16

Clusters of arbitrary shape.

Although some feature transformation methods, such as kernel k-means, may help, it is often tricky to choose appropriate kernel functions. Given such data, they would likely inaccurately identify convex regions, where noises or outliers are included in the clusters.

To find clusters of arbitrary shape, alternatively, we can model clusters as dense regions in the data space, separated by sparse regions. This is the main strategy behind *density-based clustering methods*, which can discover clusters of nonspherical shape. In this section, you will learn the basic techniques of density-based clustering by studying two representative methods, namely, DBSCAN (Section 8.4.1) and DENCLUE (Section 8.4.2). To tackle the computational cost in density-based clustering, the data space may be partitioned into a grid. This idea motivates the grid-based clustering methods (Section 8.4.3).

8.4.1 DBSCAN: density-based clustering based on connected regions with high density

"How can we find dense regions in density-based clustering?" The *density* of an object $\boldsymbol{o}$ can be measured by the number of objects close to $\boldsymbol{o}$. **DBSCAN** (Density-Based Spatial Clustering of Applications with Noise) finds *core objects*, that is, objects that have dense neighborhoods. It connects core objects and their neighborhoods to form dense regions as clusters. In this section, let us explain **DBSCAN***, an improved version of the original **DBSCAN**.

"How does **DBSCAN*** *quantify the neighborhood of an object?"* **DBSCAN*** employs a user-specified parameter $\epsilon > 0$ to specify the radius of a neighborhood that is considered for every object. The **ϵ-neighborhood** of an object $\boldsymbol{o}$ is the space within a radius ϵ centered at $\boldsymbol{o}$.

Due to the fixed neighborhood size parameterized by ϵ, the **density of a neighborhood** can be measured simply by the number of objects in the neighborhood. To determine whether a neighborhood is dense or not, DBSCAN* uses another user-specified parameter, $MinPts$, which specifies the density

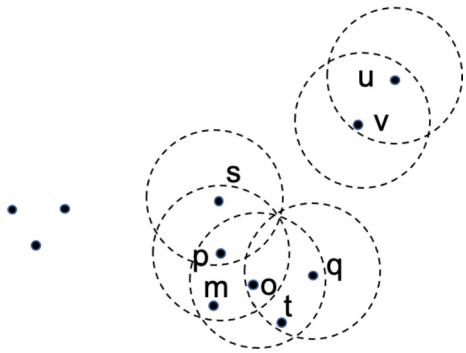


FIGURE 8.17

Density-reachability and density-connectivity in DBSCAN.

threshold of dense regions. An object is a **core object** if the ϵ-neighborhood of the object contains at least $MinPts$ objects, otherwise, it is **noise**. Core objects are the pillars of dense regions.

Given a set, D, of objects, we can identify all core objects with respect to the given parameters, ϵ and $MinPts$. The clustering task is therein reduced to using core objects and their neighborhoods to form dense regions, where the dense regions are clusters.

Two core objects $\boldsymbol{p}$ and $\boldsymbol{q}$ are **ϵ-reachable** if $d(\boldsymbol{p}, \boldsymbol{q}) \leq \epsilon$, that is, $\boldsymbol{p}$ is in the ϵ-neighbor of $\boldsymbol{q}$ and vice versa. Two core objects $\boldsymbol{p}$ and $\boldsymbol{q}$ are **density-connected** if $\boldsymbol{p}$ and $\boldsymbol{q}$ are ϵ-reachable or transitively ϵ-reachable, where $\boldsymbol{p}$ and $\boldsymbol{q}$ are transitive ϵ-reachable if there exist one or multiple core objects $\boldsymbol{r}_1, \ldots, \boldsymbol{r}_l$ such that $\boldsymbol{p}$ and $\boldsymbol{r}_1$ are ϵ-reachable, $\boldsymbol{r}_i$ and $\boldsymbol{r}_{i+1}$ $(1 \leq i < l)$ are ϵ-reachable, and $\boldsymbol{r}_l$ and $\boldsymbol{q}$ are ϵ-reachable. Then, a cluster C with respect to parameters ϵ and $MinPts$ is simply a nonempty maximal subset of core objects that every pair of objects in C is density connected.

In DBSCAN*, a cluster contains only core objects. However, it is easy to assign a noncore object that is in the ϵ-neighborhood of a core object to the cluster containing that core object. DBSCAN explicitly identifies such noncore objects as **border objects**, and DBSCAN* can use a postprocessing step to pick up those border objects. Those objects that do not belong to any ϵ-neighborhood are outliers.

Example 8.7. Density-reachability and density-connectivity. Consider Fig. 8.17 for a given ϵ represented by the radius of the circles, and, say, let $MinPts = 3$.

Of the labeled objects, $\boldsymbol{p}$, $\boldsymbol{m}$, $\boldsymbol{o}$, $\boldsymbol{q}$, and $\boldsymbol{t}$ are core objects, since each of the ϵ-neighborhoods (dashed circles in the figure) of them contains at least three objects. Objects $\boldsymbol{p}$ and $\boldsymbol{o}$ are ϵ-reachable, so are $\boldsymbol{o}$ and $\boldsymbol{q}$. Thus $\boldsymbol{p}$ and $\boldsymbol{q}$ are density-connected.

It can be verified that the core objects $\boldsymbol{p}$, $\boldsymbol{m}$, $\boldsymbol{o}$, $\boldsymbol{q}$, and $\boldsymbol{t}$ form a cluster, since each two among them are density-connected and no other core objects can be added into this group so that the pairwise density-connectivity is maintained.

Object $\boldsymbol{s}$ is not a core object, since the ϵ-neighborhood of s contains only two objects. However, $\boldsymbol{s}$ is in the ϵ-neighborhood of core object $\boldsymbol{p}$, thus $\boldsymbol{s}$ is a border object.

Objects $\boldsymbol{u}$ and $\boldsymbol{v}$ are not core objects, and they do not belong to the ϵ-neighborhood of any core objects. Thus they are outliers. □

Algorithm: DBSCAN*: a density-based clustering algorithm.

Input:

- D: a data set containing n objects,
- ϵ: the radius parameter, and
- $MinPts$: the neighborhood density threshold.

Output: A set of density-based clusters.
Method:

```
(1)  mark all objects as unvisited;
(2)  do
(3)      randomly select an unvisited object p;
(4)      mark p as visited;
(5)      if the ε-neighborhood of p has at least MinPts objects
(6)          create a new cluster C, and add p to C;
(7)          let N be the set of objects in the ε-neighborhood of p;
(8)          for each point p' in N
(9)              if p' is unvisited
(10)                 mark p' as visited;
(11)                 if the ε-neighborhood of p' has at least MinPts points,
                         add those points to N and add p' to C;
(12)         end for
(13)         output C;
(14)     else mark p as noise;
(15) until no object is unvisited;
```

FIGURE 8.18

DBSCAN* algorithm.

"How does DBSCAN find clusters?"* Initially, all objects in a given data set D are marked as "`unvisited`." DBSCAN* randomly selects an unvisited object $\boldsymbol{p}$, marks $\boldsymbol{p}$ as "`visited`," and checks whether the ϵ-neighborhood of $\boldsymbol{p}$ contains at least $MinPts$ objects. If not, $\boldsymbol{p}$ is marked as a `noise` point. Otherwise, a new cluster C is created for $\boldsymbol{p}$, and all the objects in the ϵ-neighborhood of $\boldsymbol{p}$ are added to a candidate set, N.

DBSCAN* iteratively adds to C those core objects in N that do not belong to any cluster. In this process, for an object $\boldsymbol{p'}$ in N that carries the label "`unvisited`," DBSCAN* marks it as "`visited`" and checks its ϵ-neighborhood. If the ϵ-neighborhood of $\boldsymbol{p'}$ has at least $MinPts$ objects, $\boldsymbol{p'}$ is labeled as a core object and added into C, those objects in the ϵ-neighborhood of $\boldsymbol{p'}$ are added to N. DBSCAN* continues adding objects to C until C can no longer be expanded, that is, N is empty. At this time, cluster C is completed, and thus is output.

To find the next cluster, DBSCAN* randomly selects an unvisited object from the remaining ones. The clustering process continues until all objects are visited. The pseudocode of the DBSCAN* algorithm is given in Fig. 8.18.

If a spatial index is used, the computational complexity of DBSCAN* is $O(n \log n)$, where n is the number of database objects. Otherwise, the complexity is $O(n^2)$. With appropriate settings of the user-defined parameters, ϵ and $MinPts$, the algorithm is effective in finding arbitrary-shaped clusters.

It is not easy to specify two parameters, ϵ and $MinPts$, in DBSCAN. Moreover, density-based clusters may also have hierarchies. For example, within a dense area there may be a sub-area is sub-*

stantially denser. Can we find hierarchical density-based clusters? Indeed, DBSCAN* can be extended to HDBSCAN*, which achieves density-based hierarchical clustering.

HDBSCAN* only takes one parameter, $MinPts$. For an object $\boldsymbol{p}$, the **core distance** of $\boldsymbol{p}$, denoted by $d_{core}(\boldsymbol{p})$ is the distance from $\boldsymbol{p}$ to its $MinPts$th nearest neighbor (including $\boldsymbol{p}$ itself). In other words, $d_{core}(\boldsymbol{p})$ is the minimum radius with respect to which $\boldsymbol{p}$ is a core object in DBSCAN*. For two objects $\boldsymbol{p}$ and $\boldsymbol{q}$, the **mutual reachability distance** between them is $d_{mreach}(\boldsymbol{p}, \boldsymbol{q}) = \max\{d_{core}(\boldsymbol{p}), d_{core}(\boldsymbol{q}), d(\boldsymbol{p}, \boldsymbol{q})\}$. In other words, $d_{mreach}(\boldsymbol{p}, \boldsymbol{q})$ is the minimum radius ϵ such that $\boldsymbol{p}$ and $\boldsymbol{q}$ are ϵ-reachable in DBSCAN*.

Given a set of objects as the input to HDBSCAN*, we can construct a **mutual reachability graph** G_{MinPts}, which is a complete graph. Every object in the input is a node in the mutual reachability graph. The weight of the edge between $\boldsymbol{p}$ and $\boldsymbol{q}$ is the mutual reachability distance $d_{mreach}(\boldsymbol{p}, \boldsymbol{q})$. We can apply the minimum spanning tree approach (Section 8.3.3) on the mutual reachability graph to find density-based hierarchical clusters.

To further reduce the demand of setting parameters, a cluster analysis method called **OPTICS** was proposed. OPTICS does not explicitly produce a data set clustering. Instead, it outputs a **cluster ordering**, a linear list of all objects under analysis, representing the *density-based clustering structure* of the data. Objects in a denser cluster are listed closer to each other in the cluster ordering. This ordering is equivalent to density-based clustering obtained from a wide range of parameter settings. Thus, OPTICS does not require the user to provide a specific density threshold. The cluster ordering can be used to extract basic clustering information (e.g., cluster centers, or arbitrary-shaped clusters), derive the intrinsic clustering structure, and provide a visualization of the clustering.

To construct the different clusterings simultaneously, the objects are processed in a specific order. This order selects an object that is density-reachable with respect to the lowest ϵ value so that clusters with higher density (i.e., lower ϵ) will be finished first. For example, Fig. 8.19 shows the reachability plot for a simple 2-D data set, which presents a general overview of how the data are structured and clustered. The data objects are plotted in the clustering order (horizontal axis) together with their respective reachability-distances (vertical axis). The three Gaussian "bumps" in the plot reflect three clusters in the data set.

OPTICS can be seen as a generalization of DBSCAN that replaces the ϵ parameter with a maximum value that mostly affects performance. $MinPts$ then essentially becomes the minimum cluster size to find. While the algorithm is much easier to parameterize than DBSCAN, it usually produces a hierarchical clustering instead of the simple data partitioning that DBSCAN produces.

8.4.2 DENCLUE: clustering based on density distribution functions

Density estimation is a core issue in density-based clustering. **DENCLUE** (DENsity-based CLUsteRing) is a clustering method based on a set of density distribution functions. We first give some background on density estimation and then describe the DENCLUE algorithm.

In probability and statistics, **density estimation** is the estimation of an unobservable underlying probability density function based on a set of observed data. In the context of density-based clustering, the unobservable underlying probability density function is the true distribution of the population of all possible objects to be analyzed. The observed data set is regarded as a random sample from that population.

In DENCLUE, **kernel density estimation** is used, which is a nonparametric density estimation approach from statistics. The general idea behind kernel density estimation is simple. We treat an observed

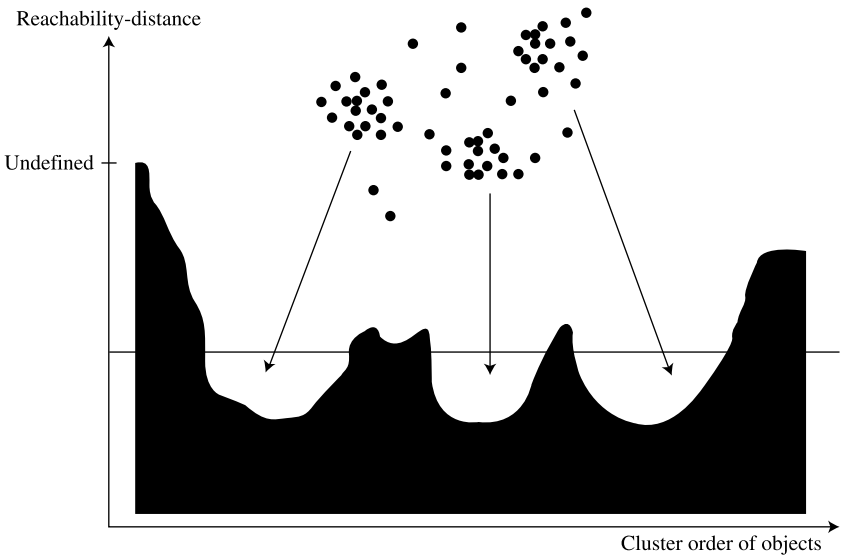


FIGURE 8.19

Cluster ordering in OPTICS. *Source:* Adapted from Ankerst, Breunig, Kriegel, and Sander [ABKS99].

object as an indicator of high-probability density in the surrounding region. The probability density at a point depends on the distances from this point to the observed objects.

Formally, let $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ be an independent and identically distributed sample of a random variable f. The *kernel density approximation of the probability density function* is

$$\hat{f}_h(x) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{\boldsymbol{x} - \boldsymbol{x_i}}{h}\right), \tag{8.22}$$

where $K()$ is a kernel function and h is the bandwidth serving as a smoothing parameter. A **kernel** can be regarded as a function modeling the influence of a sample point within its neighborhood. Technically, a kernel $K()$ is a nonnegative real-valued integrable function that should satisfy two requirements: $\int_{-\infty}^{+\infty} K(u)du = 1$ and $K(-u) = K(u)$ for all values of u. A frequently used kernel is a standard Gaussian function with a mean of 0 and a variance of 1:

$$K\left(\frac{\boldsymbol{x} - \boldsymbol{x_i}}{h}\right) = \frac{1}{\sqrt{2\pi}} e^{-\frac{(x - x_i)^2}{2h^2}}. \tag{8.23}$$

DENCLUE uses a Gaussian kernel to estimate density based on the given set of objects to be clustered. A point $\boldsymbol{x^*}$ is called a **density attractor** if it is a local maximum of the estimated density function. To avoid trivial local maximum points, DENCLUE uses a noise threshold, ξ, and only considers those density attractors $\boldsymbol{x^*}$ such that $\hat{f}(\boldsymbol{x^*}) \geq \xi$. These nontrivial density attractors are the centers of clusters.

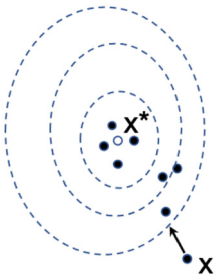


FIGURE 8.20

Hill-climbing in DENCLUE.

The objects under analysis are assigned to clusters through density attractors using a stepwise hill-climbing procedure. For an object, $\boldsymbol{x}$, the hill-climbing procedure starts from $\boldsymbol{x}$ and is guided by the gradient of the estimated density function. That is, the density attractor for $\boldsymbol{x}$ is computed as

$$\begin{aligned} \boldsymbol{x^0} &= \boldsymbol{x} \\ \boldsymbol{x^{j+1}} &= \boldsymbol{x^j} + \delta \frac{\nabla \hat{f}(\boldsymbol{x^j})}{|\nabla \hat{f}(\boldsymbol{x^j})|}, \end{aligned} \tag{8.24}$$

where δ is a parameter to control the speed of convergence, and

$$\nabla \hat{f}(\boldsymbol{x}) = \frac{1}{h^{d+2} n \sum_{i=1}^{n} K\left(\frac{\boldsymbol{x}-\boldsymbol{x_i}}{h}\right)(\boldsymbol{x_i}-\boldsymbol{x})}. \tag{8.25}$$

The hill-climbing procedure stops at step $k > 0$ if $\hat{f}(\boldsymbol{x^{k+1}}) < \hat{f}(\boldsymbol{x^k})$, and assigns $\boldsymbol{x}$ to the density attractor $\boldsymbol{x^*} = \boldsymbol{x^k}$. An object $\boldsymbol{x}$ is an outlier or noise if it converges in the hill-climbing procedure to a local maximum $\boldsymbol{x^*}$ with $\hat{f}(\boldsymbol{x^*}) < \xi$.

Fig. 8.20 illustrates the hill-climbing idea. For a point $\boldsymbol{x}$, the density attractor for x is initialized to $\boldsymbol{x^0} = \boldsymbol{x}$. In the next iteration, the density attractor moves a small step towards the direction indicated by the gradient of the density function, shown by the arrow in the figure, until the point $\boldsymbol{x}^*$ where density is stable (the white circle point in the figure), which is a local optimal.

A cluster in DENCLUE is a set of density attractors X and a set of input objects C such that each object in C is assigned to a density attractor in X, and there exists a path between every pair of density attractors where the density is above ξ. By using multiple density attractors connected by paths, DENCLUE can find clusters of arbitrary shape.

DENCLUE has several advantages. It can be regarded as a generalization of several well-known clustering methods such as single-linkage approaches and DBSCAN. Moreover, DENCLUE is invariant against noise. The kernel density estimation can effectively reduce the influence of noise by uniformly distributing noise into the input data.

8.4.3 Grid-based methods

As analyzed in the previous subsections, computing density for density-based clustering may be costly, particularly on large data sets and data sets of high dimensionality. To tackle the efficiency and scalability challenges, one idea is to partition the data space into cells using a grid. This motivates the grid-based clustering methods.

The *grid-based clustering* approach uses a multiresolution grid data structure. It quantizes the object space into a finite number of cells that form a grid structure on which all of the operations for clustering are performed. The main advantage of the approach is its fast processing time, which is typically independent of the number of data objects, yet dependent on only the number of cells in each dimension in the quantized space.

Typically, grid-based clustering takes three steps.

1. We create a grid structure so that the data space is partitioned into a finite number of cells.
2. For each cell, we calculate the cell density. By a carefully designed method, we may be able to scan the data once and derive the densities for all cells. This step is a key to gain efficiency and scalability.
3. We use the dense cells to assemble clusters and optionally summarize dense cells and the corresponding clusters.

Let us illustrate this using an example. **CLIQUE** (CLustering In QUEst) is a simple grid-based method for finding density-based clusters in subspaces. CLIQUE partitions each dimension into nonoverlapping intervals, thereby partitioning the entire data space into cells. It uses a density threshold to identify *dense* cells and *sparse* ones. A cell is dense if the number of objects mapped to it exceeds the density threshold.

The main strategy behind CLIQUE for identifying a candidate search space uses the monotonicity of dense cells with respect to dimensionality. This is based on the *Apriori property* used in frequent pattern and association rule mining (Chapter 4). In the context of clusters in subspaces, the monotonicity says the following. A k-dimensional cell c ($k > 1$) can have at least l points only if every $(k-1)$-dimensional projection of c, which is a cell in a $(k-1)$-dimensional subspace, has at least l points. Consider Fig. 8.21, where the data space contains three dimensions: *age, salary*, and *vacation*. A 2-D

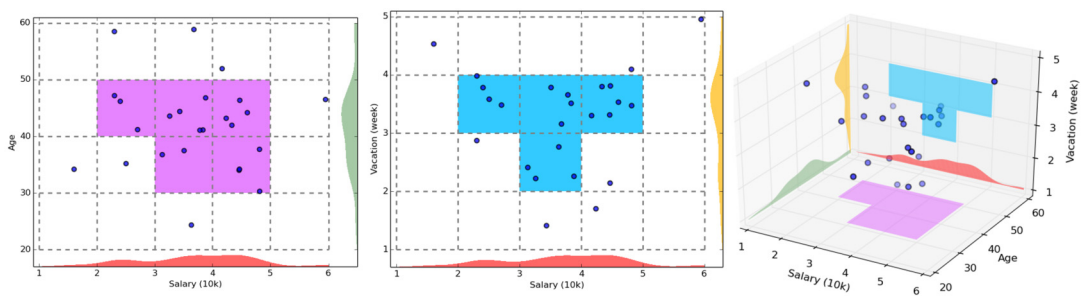


FIGURE 8.21

Dense units found with respect to *age* for the dimensions *salary* and *vacation* are intersected to provide a candidate search space for dense units of higher dimensionality.

cell, say in the subspace formed by *age* and *salary*, contains l points only if the projection of this cell in every dimension, that is, *age* and *salary*, respectively, contains at least l points.

CLIQUE performs clustering in three steps. In the first step, CLIQUE partitions the d-dimensional data space into nonoverlapping rectangular units.

In the second step, CLIQUE identifies the dense units among these. CLIQUE finds dense cells in all of the subspaces. To do so, CLIQUE partitions every dimension into intervals, and identifies intervals containing at least l points, where l is the density threshold. CLIQUE then iteratively joins two k-dimensional dense cells, c_1 and c_2, in subspaces $(D_{i_1}, \ldots, D_{i_k})$ and $(D_{j_1}, \ldots, D_{j_k})$, respectively, if $D_{i_1} = D_{j_1}, \ldots, D_{i_{k-1}} = D_{j_{k-1}}$, and c_1 and c_2 share the same intervals in those dimensions. The join operation generates a new $(k+1)$-dimensional candidate cell c in space $(D_{i_1}, \ldots, D_{i_{k-1}}, D_{i_k}, D_{j_k})$. CLIQUE checks whether the number of points in c passes the density threshold. The iteration terminates when no candidates can be generated or no candidate cells are dense.

In the last step, CLIQUE uses the dense cells in each subspace to assemble clusters, which can be of arbitrary shape. The idea is to apply the Minimum Description Length (MDL) principle (Chapter 7) to use the *maximal regions* to cover connected dense cells, where a maximal region is a hyperrectangle where every cell falling into this region is dense, and the region cannot be extended further in any dimension in the subspace. Finding the best description of a cluster in general is NP-hard. Thus CLIQUE adopts a simple greedy approach. It starts with an arbitrary dense cell, finds a maximal region covering the cell, and then works on the remaining dense cells that have not yet been covered. The greedy method terminates when all dense cells are covered.

"How effective is CLIQUE?" CLIQUE automatically finds subspaces of the highest dimensionality such that high-density clusters exist in those subspaces. It is insensitive to the order of input objects and does not presume any canonical data distribution. It scales linearly with the size of the input and has good scalability as the number of dimensions in the data increases. However, obtaining a meaningful clustering is dependent on proper tuning of the grid size (which is a stable structure here) and the density threshold. This can be difficult in practice because the grid size and density threshold are used across all combinations of dimensions in the data set. Thus the accuracy of the clustering results may be degraded at the expense of the method's simplicity. Moreover, for a given dense region, all projections of the region onto lower-dimensionality subspaces will also be dense. This can result in a large overlap among the reported dense regions. Furthermore, it is difficult to find clusters of rather different densities within different dimensional subspaces.

STING is another representative grid-based multiresolution clustering technique. In STING, the spatial area of the input objects is divided into rectangular cells. The space can be divided in a hierarchical and recursive way. Several levels of such rectangular cells correspond to different levels of resolution and form a hierarchical structure: Each cell at a high level is partitioned to form a number of cells at the next lower level. Statistical information regarding the attributes in each grid cell, such as the mean, maximum, and minimum values, is precomputed and stored as *statistical parameters*. These statistical parameters are useful for query processing and for other data analysis tasks.

Fig. 8.22 shows a hierarchical structure for STING clustering. The statistical parameters of higher-level cells can easily be computed from the parameters of the lower-level cells. These parameters include the following: the attribute-independent parameter, *count*; and the attribute-dependent parameters, *mean*, $stdev$ (standard deviation), min (minimum), max (maximum), and the type of *distribution* that the attribute value in the cell follows such as *normal, uniform*, *exponential*, or *none* (if the distribution is unknown). Here, the attribute is a selected measure for analysis such as *price* for house objects.

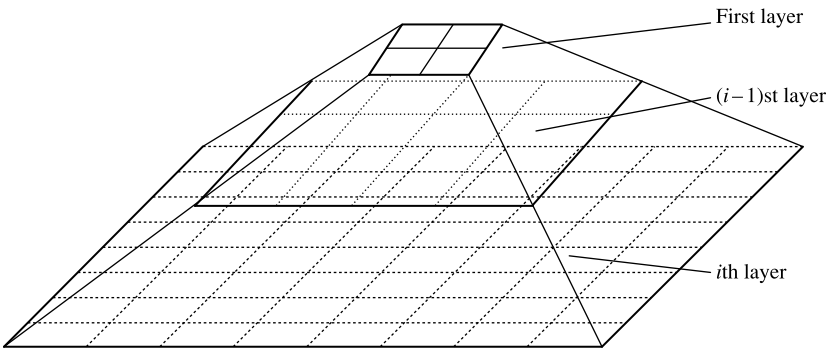


FIGURE 8.22

Hierarchical structure for STING clustering.

When the data are loaded into the database, the parameters *count*, $mean$, $stdev$, min, and max of the bottom-level cells are calculated directly from the data. The value of *distribution* may either be assigned by the user if the distribution type is known beforehand or obtained by hypothesis tests such as the χ^2 test. The type of distribution of a higher-level cell can be computed based on the majority of distribution types of its corresponding lower-level cells in conjunction with a threshold filtering process. If the distributions of the lower-level cells disagree with each other and fail the threshold test, the distribution type of the high-level cell is set to *none*.

"How is this statistical information useful for query answering?" The statistical parameters can be used in a top-down, grid-based manner as follows. First, a layer within the hierarchical structure is determined from which the query-answering process is to start. This layer typically contains a small number of cells. For each cell in the current layer, we compute the confidence interval (or estimated probability range) reflecting the relevancy of the cell to the given query. The irrelevant cells are removed from further consideration. Processing of the next lower level examines only the remaining relevant cells. This process repeats until the bottom layer is reached. At this time, if the query specification is met, the regions of relevant cells that satisfy the query are returned. Otherwise, the data points that fall into the relevant cells are retrieved and further processed until they meet the query's requirements.

An interesting property of STING is that it approaches the clustering result of DBSCAN if the granularity approaches 0 (i.e., toward very low-level data). In other words, using the count and cell size information, dense clusters can be identified approximately using STING. Therefore STING can also be regarded as a density-based clustering method.

"What advantages does STING offer over other clustering methods?" STING offers several advantages. First, the grid-based computation is *query-independent* because the statistical information stored in each cell represents the summary information of the data in the grid cell, independent of the query. Moreover, the grid structure facilitates parallel processing and incremental updating. Last, the efficiency of STING is a major advantage: STING goes through the database once to compute the statistical parameters of the cells and hence the time complexity of generating clusters is $O(n)$, where n is the total number of objects. After generating the hierarchical structure, the query processing time

is $O(g)$, where g is the total number of grid cells at the lowest level, which is usually much smaller than n.

Because STING uses a multiresolution approach to cluster analysis, the quality of STING clustering depends on the granularity of the lowest level of the grid structure. If the granularity is very fine, the cost of processing increases substantially; however, if the bottom level of the grid structure is too coarse, it may reduce the quality of cluster analysis. Moreover, STING does not consider the spatial relationship between the children and their neighboring cells for construction of a parent cell. As a result, the shapes of the resulting clusters are isothetic, that is, all the cluster boundaries are either horizontal or vertical, and no diagonal boundary is detected. This may lower the quality and accuracy of the clusters despite the fast processing time of the technique.

8.5 Evaluation of clustering

By now you have learned what clustering is and know several popular clustering methods. You may ask, "*When I try out a clustering method on a data set, how can I evaluate whether the clustering results are good?*" In general, *cluster evaluation* assesses the feasibility of clustering analysis on a data set and the quality of the results generated by a clustering method. The major tasks of clustering evaluation include the following:

- *Assessing clustering tendency*. In this task, for a given data set, we assess whether a nonrandom structure exists in the data. Blindly applying a clustering method on a data set will return clusters; however, the clusters mined may be misleading. Clustering analysis on a data set is meaningful only when there is a nonrandom structure in the data.
- *Determining the number of clusters in a data set*. A few algorithms, such as k-means, require the number of clusters in a data set as the parameter. Moreover, the number of clusters can be regarded as an interesting and important summary statistic of a data set. Therefore it is desirable to estimate this number even before a clustering algorithm is used to derive detailed clusters.
- *Measuring clustering quality*. After applying a clustering method on a data set, we want to assess how good the resulting clusters are. A number of measures can be used. Some methods measure how well the clusters fit the data set, while others measure how well the clusters match the ground truth, if such truth is available. There are also measures that score clusterings and thus can compare two sets of clustering results on the same data set.

In this section, we discuss these three topics one by one.

8.5.1 Assessing clustering tendency

Clustering tendency assessment determines whether a given data set has a nonrandom structure, which may lead to meaningful clusters. Consider a data set that does not have any nonrandom structure, such as a set of uniformly distributed points in a data space. Even though a clustering algorithm may return clusters for the data, those clusters are random and thus are not meaningful.

Example 8.8. Clustering requires nonuniform distribution of data. Fig. 8.23 shows a data set that is uniformly distributed in 2-D data space. Although a clustering algorithm may still artificially partition

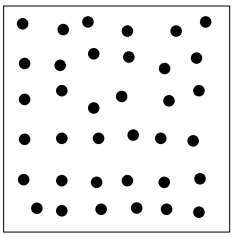


FIGURE 8.23

A data set that is uniformly distributed in the data space.

the points into groups, the groups will unlikely mean anything significant to the application due to the uniform distribution of the data. □

"How can we assess the clustering tendency of a data set?" Intuitively, we can try to measure the probability that the data set is generated by a uniform data distribution. This can be achieved using statistical tests for spatial randomness. To illustrate this idea, let us look at a simple yet effective statistic called the Hopkins statistic.

The **Hopkins Statistic** is a spatial statistic that tests the spatial randomness of a variable as distributed in a space. Given a data set, D, which is regarded as a sample of a random variable, o, we want to determine how far away o is from being uniformly distributed in the data space. We calculate the Hopkins Statistic as follows:

1. Sample n points, $\boldsymbol{p_1}, \ldots, \boldsymbol{p_n}$ from the data space. For each point, $\boldsymbol{p_i}$ $(1 \leq i \leq n)$, we find the nearest neighbor in D, and let x_i be the distance between $\boldsymbol{p_i}$ and its nearest neighbor in D. That is,

$$x_i = \min_{\boldsymbol{v} \in D} \{dist(\boldsymbol{p_i}, \boldsymbol{v})\}. \tag{8.26}$$

2. Sample n points, $\boldsymbol{q_1}, \ldots, \boldsymbol{q_n}$ uniformly from D without replacement. That is, each point in D has the same probability of being included in this sample, and one point can only be included in the sample at most once. For each $\boldsymbol{q_i}$ $(1 \leq i \leq n)$, we find the nearest neighbor of $\boldsymbol{q_i}$ in $D - \{\boldsymbol{q_i}\}$, and let y_i be the distance between $\boldsymbol{q_i}$ and its nearest neighbor in $D - \{\boldsymbol{q_i}\}$. That is,

$$y_i = \min_{\boldsymbol{v} \in D, \boldsymbol{v} \neq \boldsymbol{q_i}} \{dist(\boldsymbol{q_i}, \boldsymbol{v})\}. \tag{8.27}$$

3. Calculate the Hopkins statistic, H, as

$$H = \frac{\sum_{i=1}^{n} x_i^d}{\sum_{i=1}^{n} x_i^d + \sum_{i=1}^{n} y_i^d}, \tag{8.28}$$

where d is the dimensionality of the data set D.

"What does the Hopkins statistic tell us about how likely data set D *follows a uniform distribution in the data space?"* If D is uniformly distributed, then $\sum_{i=1}^{n} y_i^d$ and $\sum_{i=1}^{n} x_i^d$ are close to each other,

and thus H tends to be about 0.5. However, if D is highly skewed, then the points in D are closer to their nearest neighbors than the random points $\boldsymbol{p_1}, \ldots, \boldsymbol{p_n}$ are, and thus $\sum_{i=1}^{n} x_i^d$ shall be substantially larger than $\sum_{i=1}^{n} y_i^d$ in expectation, and H tends to be close to 1.

Example 8.9. Hopkins statistic. Consider a 1-D data set $D = \{0.9, 1, 1.3, 1.4, 1.5, 1.8, 2, 2.1, 4.1, 7, 7.4, 7.5, 7.7, 7.8, 7.9, 8.1\}$ in the data space $[0, 10]$. We draw a sample of four points from D without replacement, say, 1.3, 1.8, 7.5, and 7.9. We also draw a sample of four points uniformly from the data space $[0, 10]$, say, 1.9, 4, 6, 8. Then, the Hopkins statistic can be calculated as

$$H = \frac{|1.9-2| + |4-4.1| + |6-7| + |8-8.1|}{(|1.9-2| + |4-4.1| + |6-7| + |8-8.1|) + (|1.3-1.4| + |1.8-2| + |7.5-7.4| + |7.9-7.8|)}$$
$$= \frac{1.3}{1.3+0.5} = \frac{1.3}{1.8} = 0.72.$$

Since the Hopkins statistic is substantially larger than 0.5 and is close to 1, the data set D has a strong clustering tendency. Indeed, there are two clusters, one around 1.5 and the other one around 7.8. □

In addition to Hopkins statistic, there are some other methods, such as spatial histogram and distance distribution, comparing statistics between a data set under clustering tendency analysis and the corresponding uniform distribution. For example, distance distribution compares the distribution of pairwise distance in the target data set and that in a random uniform sample from the data space.

8.5.2 Determining the number of clusters

Determining the "right" number of clusters in a data set is important, not only because some clustering algorithms like k-means require such a parameter, but also because the appropriate number of clusters controls the proper granularity of cluster analysis. It can be regarded as finding a good balance between *compressibility* and *accuracy* in cluster analysis. Consider two extreme cases. What if you were to treat the entire data set as a cluster? This would maximize the compression of the data, but such a cluster analysis has no value. In contrast, treating each object in a data set as a cluster gives the finest clustering resolution (i.e., most accurate due to the zero distance between an object and the corresponding cluster center). In some methods like k-means, this even achieves the minimum cost. However, having one object per cluster does not enable any data summarization.

Determining the number of clusters is far from easy, often because the "right" number is ambiguous. Figuring out the right number of clusters often depends on the distribution's shape and scale in the data set, as well as the clustering resolution required by the user. There are many possible ways to estimate the number of clusters.

For example, a simple method is to set the number of clusters to about $\sqrt{\frac{n}{2}}$ for a data set of n points. In expectation, each cluster has $\sqrt{2n}$ points. Section 8.2.2 introduces the Calinski-Harabasz index, which estimates the number of clusters for k-means.

Let us look at two more alternative methods.

The **elbow method** is based on the observation that increasing the number of clusters can help to reduce the sum of within-cluster variance of each cluster. This is because having more clusters allows one to capture finer groups of data objects that are more similar to each other. However, the marginal

effect of reducing the sum of within-cluster variances may drop if too many clusters are formed, because splitting a cohesive cluster into two gives only a small reduction. Consequently, a heuristic for selecting the right number of clusters is to use the turning point in the curve of the sum of within-cluster variances with respect to the number of clusters.

Technically, given a number, $k > 0$, we can form k clusters on the data set in question using a clustering algorithm like k-means, and calculate the sum of within-cluster variances, $var(k)$. We can then plot the curve of var with respect to k. The first (or most significant) turning point of the curve suggests the "right" number.

More advanced methods can determine the number of clusters using information criteria or information theoretic approaches. Please refer to the bibliographic notes for further information (Section 8.8).

The "right" number of clusters in a data set can also be determined by **cross-validation**, a technique often used in classification (Chapter 6). First, we divide the given data set, D, into m parts. Next, we use $m - 1$ parts to build a clustering model, and use the remaining part to test the quality of the clustering. For example, for each point in the test set, we can find the closest centroid. Consequently, we can use the sum of the squared distances between all points in the test set and the closest centroids to measure how well the clustering model fits the test set. For any integer $k > 0$, we repeat this process m times to derive clusterings of k clusters, using each part in turn as the test set. The average of the quality measure is taken as the overall quality measure. We can then compare the overall quality measure with respect to different values of k and find the number of clusters that best fits the data.

8.5.3 Measuring clustering quality: extrinsic methods

Suppose you have assessed the clustering tendency of a given data set. You may have also tried to predetermine the number of clusters in the set. You can now apply one or multiple clustering methods to obtain clusterings of the data set. "*How good is the clustering generated by a method, and how can we compare the clusterings generated by different methods?*"

Extrinsic vs. intrinsic methods

We have a few methods to choose from for measuring the quality of a clustering. In general, these methods can be categorized into two groups according to whether ground truth is available. Here, *ground truth* is the ideal clustering that is often built using human experts.

If ground truth is available, it can be used by the **extrinsic methods**, which compare the clustering against the ground truth and measure. If the ground truth is unavailable, we can use the **intrinsic methods**, which evaluate the goodness of a clustering by considering how well the clusters are separated. Ground truth can be considered as supervision in the form of "cluster labels." Hence, extrinsic methods are also known as *supervised methods*, whereas intrinsic methods are *unsupervised methods*.

In this section, we focus on extrinsic methods. We will discuss intrinsic methods in the next section.

Desiderata of extrinsic methods

When the ground truth is available, we can compare it with a clustering to assess the quality of the clustering. Thus the core task in extrinsic methods is to assign a score, $Q(\mathcal{C}, \mathcal{C}_g)$, to a clustering, $\mathcal{C}$, given the ground truth, $\mathcal{C}_g$. Whether an extrinsic method is effective largely depends on the measure, Q, it uses.

In general, a measure Q on clustering quality is effective if it satisfies the following four essential criteria:

- **Cluster homogeneity.** This requires that the purer the clusters in a clustering are, the better the clustering. Suppose that the ground truth says that the objects in a data set, $D = \{a, b, c, d, e, f, g, h\}$, can belong to three categories. Objects a and b are in category L_1, objects c and d belong to category L_2, and the others are in category L_3. Consider clustering, $\mathcal{C}_1 = \{\{a, b, c, d\}, \{e, f, g, h\}\}$, wherein a cluster $\{a, b, c, d\} \in \mathcal{C}_1$ contains objects from two categories, L_1 and L_2. Also consider clustering $\mathcal{C}_2 = \{\{a, b\}, \{c, d\}, \{e, f, g, h\}\}$, which is identical to $\mathcal{C}_1$ except that $\mathcal{C}_2$ is split into two clusters containing the objects in L_1 and L_2, respectively. A clustering quality measure, Q, respecting cluster homogeneity should give a higher score to $\mathcal{C}_2$ than $\mathcal{C}_1$, that is, $Q(\mathcal{C}_2, \mathcal{C}_g) > Q(\mathcal{C}_1, \mathcal{C}_g)$.
- **Cluster completeness.** This is the counterpart of cluster homogeneity. Cluster completeness requires that for a clustering, if any two objects belong to the same category according to the ground truth, then they should be assigned to the same cluster. Cluster completeness requires that a clustering should assign objects belonging to the same category (according to the ground truth) to the same cluster. Continue our previous example. Suppose clustering $\mathcal{C}_3 = \{\{a, b\}, \{c, d\}, \{e, f\}, \{g, h\}\}$. $\mathcal{C}_3$ and $\mathcal{C}_2$ are identical except that $\mathcal{C}_3$ split the objects in category L_3 into two clusters. Then, a clustering quality measure, Q, respecting cluster completeness should give a higher score to $\mathcal{C}_2$, that is, $Q(\mathcal{C}_2, \mathcal{C}_g) > Q(\mathcal{C}_1, \mathcal{C}_g)$.
- **Rag bag.** In many practical scenarios, there is often a "rag bag" category containing objects that cannot be merged with other objects. Such a category is often called "miscellaneous," "other," and so on. The rag bag criterion states that putting a heterogeneous object into a pure cluster should be penalized more than putting it into a rag bag. Consider a clustering $\mathcal{C}_1$ and a cluster $C \in \mathcal{C}_1$ such that all objects in C except for one, denoted by $\boldsymbol{o}$, belong to the same category according to the ground truth. Consider a clustering $\mathcal{C}_2$ identical to $\mathcal{C}_1$ except that $\boldsymbol{o}$ is assigned to a cluster $C' \neq C$ in $\mathcal{C}_2$ such that C' contains objects from various categories according to ground truth, and thus is noisy. In other words, C' in $\mathcal{C}_2$ is a rag bag. Then, a clustering quality measure Q respecting the rag bag criterion should give a higher score to $\mathcal{C}_2$, that is, $Q(\mathcal{C}_2, \mathcal{C}_g) > Q(\mathcal{C}_1, \mathcal{C}_g)$.
- **Small cluster preservation.** If a small category is split into small pieces in a clustering, those small pieces may likely become noise and thus the small category cannot be discovered from the clustering. The small cluster preservation criterion states that splitting a small category into pieces is more harmful than splitting a large category into pieces. Consider an extreme case. Let D be a data set of $n + 2$ objects such that, according to ground truth, n objects, denoted by $\boldsymbol{o_1}, \ldots, \boldsymbol{o_n}$, belong to one category and the other two objects, denoted by $\boldsymbol{o_{n+1}}, \boldsymbol{o_{n+2}}$, belong to another category. Suppose clustering $\mathcal{C}_1$ has three clusters, $C_1^1 = \{\boldsymbol{o_1}, \ldots, \boldsymbol{o_n}\}$, $C_2^1 = \{\boldsymbol{o_{n+1}}\}$, and $C_3^1 = \{\boldsymbol{o_{n+2}}\}$. Let clustering $\mathcal{C}_2$ have three clusters, too, namely $C_1^2 = \{\boldsymbol{o_1}, \ldots, \boldsymbol{o_{n-1}}\}$, $C_2^2 = \{\boldsymbol{o_n}\}$, and $C_3^2 = \{\boldsymbol{o_{n+1}}, \boldsymbol{o_{n+2}}\}$. In other words, $\mathcal{C}_1$ splits the small category and $\mathcal{C}_2$ splits the big category. A clustering quality measure Q preserving small clusters should give a higher score to $\mathcal{C}_2$, that is, $Q(\mathcal{C}_2, \mathcal{C}_g) > Q(\mathcal{C}_1, \mathcal{C}_g)$.

Categories of extrinsic methods

The ground truth may be used in different ways to evaluate clustering quality, which lead to different extrinsic methods. In general, the extrinsic methods can be categorized according to how the ground truth is used as follows.

- **The matching-based methods** examine how well the clustering results match the ground truth in partitioning the objects in the data set. For example, the purity methods assess how a cluster matches only those objects in one group in the ground truth.
- **The information theory–based methods** compare the distribution of the clustering results and that of the ground truth. Entropy or other measures in information theory are often employed to quantify the comparison. For example, we can measure the conditional entropy between the clustering results and the ground truth to measure whether there exists dependency between the information of the clustering results and the ground truth. The higher the dependency, the better the clustering results.
- **The pairwise comparison–based methods** treat each group in the ground truth as a class and then check the pairwise consistency of the objects in the clustering results. The clustering results are good if more pairs of objects of the same class are put into the same cluster, less pairs of objects of different classes are put into the same cluster, and less pairs of objects of the same class are put into different clusters.

Next, let us use some examples to illustrate the above categories of extrinsic methods.

Matching-based methods

The matching-based methods compare clusters in the clustering results and the groups in the ground truth. Let us use an example to explain the ideas.

Suppose a clustering method partitions a set of objects $D = \{o_1, \ldots, o_n\}$ into clusters $\mathcal{C} = \{C_1, \ldots, C_m\}$. The ground truth $\mathcal{G}$ also partitions the same set of objects into groups $\mathcal{G} = \{G_1, \ldots, G_l\}$. Let $C(o_x)$ and $G(o_x)$ $(1 \leq x \leq n)$ be the cluster-id and the group-id of object o_x in the clustering results and the ground truth, respectively.

For a cluster C_i $(1 \leq i \leq m)$, how well C_i matches group G_j in the ground truth can be measured by $|C_i \cap G_j|$, the larger the better. $\frac{|C_i \cap G_j|}{|C_i|}$ can be regarded as the purity of cluster C_i, where G_j matching C_i maximizes $|C_i \cap G_j|$. The purity of the whole clustering results can be calculated as the weighted sum of the purity of the clusters. That is,

$$purity = \sum_{i=1}^{m} \frac{|C_i|}{n} \max_{j=1}^{l} \{\frac{|C_i \cap G_j|}{|C_i|}\} = \frac{1}{n} \sum_{i=1}^{m} \max_{j=1}^{l} \{|C_i \cap G_j|\}. \tag{8.29}$$

The higher the purity, the purer are the clusters, that is, the more objects in each cluster belong to the same group in the ground truth. When the purity is 1, each cluster either matches a group perfectly or is a subset of a group. In other words, no two objects belong to two groups are mixed in one cluster. However, it is possible that multiple clusters partition a group in the ground truth.

Example 8.10. Purity. Consider the set of objects $D = \{a, b, c, d, e, f, g, h, i, j, k\}$. The clustering ground truth and two clusterings $\mathcal{C}_1$ and $\mathcal{C}_2$ output by two methods are shown in Table 8.1.

The purity of clustering $\mathcal{C}_1$ is calculated by $\frac{1}{11} \times (4 + 2 + 4 + 1) = \frac{11}{11} = 1$ and that of clustering $\mathcal{C}_2$ is $\frac{1}{11}(2 + 3 + 1) = \frac{6}{11}$. In terms of purity, $\mathcal{C}_1$ is better than $\mathcal{C}_2$. Please note that, although $\mathcal{C}_1$ has purity 1, it splits G_1 in the ground truth into two clusters, C_1 and C_2. □

There are some other matching based methods further refine the measurement of matching quality, such as maximum matching and using F-measure.

Table 8.1 A set of objects, the clustering ground truth, and two clusterings.

Object	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*
Ground truth $\mathcal{G}$	G_1	G_1	G_1	G_1	G_1	G_1	G_2	G_2	G_2	G_2	G_3
Clustering $\mathcal{C}_1$	C_1	C_1	C_1	C_1	C_2	C_2	C_3	C_3	C_3	C_3	C_4
Clustering $\mathcal{C}_2$	C_1	C_1	C_2	C_2	C_2	C_3	C_1	C_2	C_2	C_1	C_3

Information theory–based methods

A clustering assigns objects to clusters and thus can be regarded as a compression of the information carried by the objects. In other words, a clustering can be regarded as a compressed representation of a given set of objects. Therefore we can use information theory to compare a clustering and the ground truth as representations. This is the general idea behind the information theory–based methods.

For example, we can measure the amount of information needed to describe the ground truth given the distribution of a clustering output by a method. Better the clustering results approach the ground truth, less amount information is needed. This leads to a natural approach using conditional entropy.

Concretely, according to information theory, the entropy of a clustering $\mathcal{C}$ is

$$H(\mathcal{C}) = -\sum_{i=1}^{m} \frac{|C_i|}{n} \log \frac{|C_i|}{n},$$

and the entropy of the ground truth is

$$H(\mathcal{G}) = -\sum_{i=1}^{l} \frac{|G_i|}{n} \log \frac{|G_i|}{n}.$$

The conditional entropy of $\mathcal{G}$ given cluster C_i is

$$H(\mathcal{G}|C_i) = -\sum_{j=1}^{l} \frac{|C_i \cap G_j|}{|C_i|} \log \frac{|C_i \cap G_j|}{|C_i|}.$$

The conditional entropy of $\mathcal{G}$ given clustering $\mathcal{C}$ is

$$H(\mathcal{G}|\mathcal{C}) = \sum_{i=1}^{m} \frac{|C_i|}{n} H(\mathcal{G}|C_i) = -\sum_{i=1}^{m} \sum_{j=1}^{l} \frac{|C_i \cap G_j|}{n} \log \frac{|C_i \cap G_j|}{|C_i|}.$$

In addition to the simple conditional entropy, more sophisticated information theory–based measures may be used, such as normalized mutual information and variation of information.

Taking the case in Table 8.1 as an example, we can calculate

$$H(\mathcal{G}|\mathcal{C}_1) = -(\frac{4}{11} \log \frac{4}{4} + \frac{2}{11} \log \frac{2}{2} + \frac{4}{11} \log \frac{4}{4} + \frac{1}{11} \log \frac{1}{1}) = 0$$

and

$$H(\mathcal{G}|\mathcal{C}_2) = -(\frac{2}{11}\log\frac{2}{4} + \frac{2}{11}\log\frac{2}{4} + \frac{3}{11}\log\frac{3}{5} + \frac{2}{11}\log\frac{2}{5} + \frac{1}{11}\log\frac{1}{2} + \frac{1}{11}\log\frac{1}{2})$$
$$= 0.297.$$

Clustering $\mathcal{C}_1$ has better quality than $\mathcal{C}_2$ in terms of conditional entropy. Again, although $H(\mathcal{G}|\mathcal{C}_1) = 0$, conditional entropy cannot detect the issue that $\mathcal{C}_1$ splits the objects in G_1 into two clusters.

Pairwise comparison–based methods

The pairwise comparison–based methods treat each group in the ground truth as a class. For each pair of objects $o_i, o_j \in D$ $(1 \leq i, j \leq n, i \neq j)$, if they are assigned to the same cluster/group, the assignment is regarded as positive, and otherwise, negative. Then, depending on assignments of o_i and o_j into clusters $C(o_i)$, $C(o_j)$, $G(o_i)$, and $G(o_j)$, we have four possible cases.

	$C(o_i) = C(o_j)$	$C(o_i) \neq C(o_j)$
$G(o_i) = G(o_j)$	true positive	false negative
$G(o_i) \neq G(o_j)$	false positive	true negative

Using the statistics on pairwise comparison, we can assess the quality of the clustering results approaching the ground truth. For example, we can use the Jaccard coefficient, which is defined as

$$J = \frac{\text{true positive}}{\text{true positive} + \text{false negative} + \text{false positive}}.$$

Many other measures can be built based on the pairwise comparison statistics, such as Rand statistic, fowlkes-Mallows measure, BCubed precision, and recall. The pairwise comparison results can be further used to conduct correlation analysis. For example, we can form a binary matrix $\boldsymbol{G}$ according to the ground truth, where element $v_{ij} = 1$ if $G(o_i) = G(o_j)$, and otherwise 0. A binary matrix $\boldsymbol{C}$ can also be constructed in a similar way based on a clustering $\mathcal{C}$. We can analyze the element-wise correlation between the two matrixes and use the correlation to measure the quality of the clustering results. Clearly, the more correlated the two matrixes, the better the clustering results.

8.5.4 Intrinsic methods

When the ground truth of a data set is not available, we have to use an intrinsic method to assess the clustering quality. Unable to reference any external supervision information, the intrinsic methods have to come back to the fundamental intuition in clustering analysis, that is, examining how compact clusters are and how well clusters are separated. Many intrinsic methods take the advantage of a similarity or distance measure between objects in the data set.

For example, the **Dunn index** measures the compactness of clusters by the maximum distance between two points that belong to the same cluster, that is, $\Delta = \max_{C(o_i)=C(o_j)}\{d(o_i, o_j)\}\}$. It measures the degree of separation among different clusters by the minimum distance between two points that belong to different clusters, that is $\delta = \min_{C(o_i)\neq C(o_j)}\{d(o_i, o_j)\}$. Then, the Dunn index is simply the ration $DI = \frac{\delta}{\Delta}$. The larger the ratio, the farther away the clusters are separated comparing to the compactness of the clusters.

The Dunn index uses the extreme distances to measure the cluster compactness and intercluster separation. The measures δ and Δ may be affected by the outliers. Many methods consider the average situations. The **silhouette coefficient** is such a measure. For a data set, D, of n objects, suppose D is partitioned into k clusters, $C_1, \ldots, C_k$. For each object $\boldsymbol{o} \in D$, we calculate $a(\boldsymbol{o})$ as the average distance between $\boldsymbol{o}$ and all other objects in the cluster to which $\boldsymbol{o}$ belongs. Similarly, $b(\boldsymbol{o})$ is the minimum average distance from $\boldsymbol{o}$ to all clusters to which $\boldsymbol{o}$ does not belong. Formally, suppose $\boldsymbol{o} \in C_i$ $(1 \leq i \leq k)$. Then,

$$a(\boldsymbol{o}) = \frac{\sum_{\boldsymbol{o}' \in C_i, \boldsymbol{o} \neq \boldsymbol{o}'} dist(\boldsymbol{o}, \boldsymbol{o}')}{|C_i| - 1} \tag{8.30}$$

and

$$b(\boldsymbol{o}) = \min_{C_j : 1 \leq j \leq k, j \neq i} \left\{ \frac{\sum_{\boldsymbol{o}' \in C_j} dist(\boldsymbol{o}, \boldsymbol{o}')}{|C_j|} \right\}. \tag{8.31}$$

The **silhouette coefficient** of $\boldsymbol{o}$ is then defined as

$$s(\boldsymbol{o}) = \frac{b(\boldsymbol{o}) - a(\boldsymbol{o})}{\max\{a(\boldsymbol{o}), b(\boldsymbol{o})\}}. \tag{8.32}$$

The value of the silhouette coefficient is between -1 and 1. The value of $a(\boldsymbol{o})$ reflects the compactness of the cluster to which $\boldsymbol{o}$ belongs. The smaller the value, the more compact the cluster. The value of $b(\boldsymbol{o})$ captures the degree to which $\boldsymbol{o}$ is separated from other clusters. The larger $b(\boldsymbol{o})$ is, the more separated $\boldsymbol{o}$ is from other clusters. Therefore when the silhouette coefficient value of $\boldsymbol{o}$ approaches 1, the cluster containing $\boldsymbol{o}$ is compact and $\boldsymbol{o}$ is far away from other clusters, which is the preferable case. However, when the silhouette coefficient value is negative (i.e., $b(\boldsymbol{o}) < a(\boldsymbol{o})$, this means that, in expectation, $\boldsymbol{o}$ is closer to the objects in another cluster than to the objects in the same cluster as $\boldsymbol{o}$. In many cases, this is a bad situation and should be avoided.

To measure a cluster's fitness within a clustering, we can compute the average silhouette coefficient value of all objects in the cluster. To measure the quality of a clustering, we can use the average silhouette coefficient value of all objects in the data set. The silhouette coefficient and other intrinsic measures can also be used in the elbow method to heuristically derive the number of clusters in a data set by replacing the sum of within-cluster variances.

8.6 Summary

- A **cluster** is a collection of data objects that are *similar* to one another within the same cluster and are *dissimilar* to the objects in other clusters. The process of grouping a set of physical or abstract objects into classes of *similar* objects is called **clustering**.
- Cluster analysis has extensive **applications**, including business intelligence, image pattern recognition, Web search, biology, and security. Cluster analysis can be used as a standalone data mining tool to gain insight into the data distribution or as a preprocessing step for other data mining algorithms operating on the detected clusters.
- Clustering is a dynamic field of research in data mining. It is related to **unsupervised learning** in machine learning.

- Clustering is a challenging field. Typical **requirements** of it include scalability, the ability to deal with different types of data and attributes, the discovery of clusters in arbitrary shape, minimal requirements for domain knowledge to determine input parameters, the ability to deal with noisy data, incremental clustering and insensitivity to input order, the capability of clustering high-dimensionality data, constraint-based clustering, and interpretability and usability.
- Many clustering algorithms have been developed. These can be categorized from several **orthogonal aspects** such as those regarding partitioning criteria, separation of clusters, similarity measures used, and clustering space. This chapter discusses major fundamental clustering methods of the following categories: *partitioning methods*, *hierarchical methods*, and *density-based and grid-based methods*. Some algorithms may belong to more than one category.
- A **partitioning method** first creates an initial set of k partitions, where parameter k is the number of partitions to construct. It then uses an *iterative relocation technique* that attempts to improve the partitioning by moving objects from one group to another. Typical partitioning methods include k-means, k-medoids, and k-modes.
- The **centroid-based partitioning technique** uses the **within-cluster variation** to measure the quality of clusters, which is the sum of squared error between all objects in a cluster and the centroid of the cluster. Minimizing the within-cluster variation is computationally challenging and thus some greedy approaches are often used. The **k-means method** uses the mean value of the points within a cluster as the centroid. It randomly selects k objects as the initial centroids of the clusters and then iteratively conducts object assignment and mean update steps until the assignment becomes stable or a certain number of iterations are reached.
- As a variation of k-means to overcome the effect of outliers, the **k-medoids method** uses actual objects to represent clusters. While k-medoids is more robust against noise and outliers, it incurs higher computational cost in each iteration. As another variation of k-means, the **k-modes** method uses modes to measure the similarity on nominal data. We can also use kernel functions, such as the Gaussian radial basis function, in k-means to find clusters that are concave and not linearly separable.
- A **hierarchical method** creates a hierarchical decomposition of the given set of data objects. The method can be classified as being either *agglomerative* (*bottom-up*) or *divisive* (*top-down*), based on how the hierarchical decomposition is formed. **Linkage measures** can be used to assess the distance between clusters in hierarchical clustering. Some widely used measures include *minimum distance* (*single-linkage*), *maximum distance* (*complete-linkage*), *mean distance*, and *average distance*. The **Lance-Williams algorithm** generalizes different measures and the agglomerative hierarchical clustering framework. The **minimum spanning tree based approach** is a representative method for divisive hierarchical clustering. Hierarchical clustering results can be represented using a **dendrogram**.
- **BIRCH** is a method combining hierarchical clustering and other clustering methods. In BIRCH, clusters are represented using **clustering features** (CF for short) and a hierarchical clustering is represented by a **CF-tree**.
- To overcome some of the drawbacks of hierarchical clustering methods, **probabilistic hierarchical clustering** uses probabilistic models to measure distances between clusters. It shares the same framework and thus has the same efficiency as agglomerative hierarchical clustering methods.
- A **density-based method** clusters objects based on the notion of density. It grows clusters either according to the density of neighborhood objects (e.g., in DBSCAN) or according to a density

function (e.g., in DENCLUE). OPTICS is a density-based method that generates an augmented ordering of the data's clustering structure.

- A **grid-based method** first quantizes the object space into a finite number of cells that form a grid structure, and then performs clustering on the grid structure. STING is a typical example of a grid-based method based on statistical information stored in grid cells. CLIQUE is a grid-based and subspace clustering algorithm.
- **Clustering evaluation** assesses the feasibility of clustering analysis on a data set and the quality of the results generated by a clustering method. The major tasks include assessing clustering tendency, determining the number of clusters, and measuring clustering quality.
- Some statistics, such as **Hopkins statistic**, can be used to assess clustering tendency. In addition to the Calinski-Harabasz index, the **elbow method**, and the **cross-validation** technique can be used to decide the number of clusters in a data set. Depending on whether the ground truth is available, the methods measuring clustering quality can be divided into **extrinsic methods** and **intrinsic methods**. The extrinsic methods try to address the desiderata of *cluster homogeneity*, *cluster completeness*, *rag bag*, and *small cluster preservation*. The extrinsic methods can be divided into the **matching-based methods** (e.g., *purity*), the **information theory–based methods** (e.g., *entropy*), and the **pairwise comparison–based methods** (e.g., using Jaccard coefficient). Examples of the intrinsic methods are the **Dunn index** and the **silhouette coefficient**.

8.7 Exercises

8.1. Briefly describe and give examples of each of the following approaches to clustering: *partitioning* methods, *hierarchical* methods, *density-based and grid-based* methods, and *bi-clustering* methods.

8.2. Suppose that the data mining task is to cluster points (with (x, y) representing location) into three clusters, where the points are

$$A_1(2, 10), A_2(2, 5), A_3(8, 4), B_1(5, 8), B_2(7, 5), B_3(6, 4), C_1(1, 2), C_2(4, 9).$$

The distance function is Euclidean distance. Suppose initially we assign A_1, B_1, and C_1 as the center of each cluster, respectively. Use the *k-means* algorithm to show *only*

a. The three cluster centers after the first round of execution.

b. The final three clusters.

8.3. Use an example to show why the k-means algorithm may not find the global optimum, that is, optimizing the within-cluster variation.

8.4. For the k-means algorithm, it is interesting to note that by choosing the initial cluster centers carefully, we may be able to not only speed up the algorithm's convergence, but also guarantee the quality of the final clustering. The ***k*-means++** algorithm is a variant of k-means, which chooses the initial centers as follows. First, it selects one center uniformly at random from the objects in the data set. Iteratively, for each object $\boldsymbol{p}$ other than the chosen center, it chooses an object as the new center. This object is chosen at random with probability proportional to $dist(\boldsymbol{p})^2$, where $dist(\boldsymbol{p})$ is the distance from $\boldsymbol{p}$ to the closest center that has already been chosen. The iteration continues until k centers are selected.

Explain why this method will not only speed up the convergence of the k-means algorithm, but also guarantee the quality of the final clustering results.

8.5. Provide the pseudocode of the object reassignment step of the PAM algorithm.

8.6. Both *k-means* and *k-medoids* algorithms can perform effective clustering.

- **a.** Illustrate the strength and weakness of *k-means* in comparison with *k-medoids*.
- **b.** Illustrate the strength and weakness of these schemes in comparison with a hierarchical clustering scheme.

8.7. Show that the single-linkage method is equivalent to taking $\alpha_i = \alpha_j = 0.5$, $\beta = 0$, and $\gamma = -0.5$ in the Lance-Williams formula; the complete-linkage method is equivalent to $\alpha_i = \alpha_j = 0.5$, $\beta = 0$, and $\gamma = 0.5$; and the Ward's criterion is equivalent to $\alpha_i = \frac{n_i+n_k}{n_i+n_j+n_k}$, $\alpha_j = \frac{n_j+n_k}{n_i+n_j+n_k}$, $\beta = -\frac{n_k}{n_i+n_j+n_k}$, and $\gamma = 0$.

8.8. Prove that in DBSCAN*, the density-connectedness is an equivalence relation.

8.9. Prove that in DBSCAN*, for a fixed $MinPts$ value and two neighborhood thresholds, $\epsilon_1 < \epsilon_2$, a cluster C with respect to ϵ_1 and $MinPts$ must be a subset of a cluster C' with respect to ϵ_2 and $MinPts$.

8.10. Provide the pseudocode of the OPTICS algorithm.

8.11. Why is it that BIRCH encounters difficulties in finding clusters of arbitrary shape but OPTICS does not? Propose modifications to BIRCH to help it find clusters of arbitrary shape.

8.12. Provide the pseudocode of the step in CLIQUE that finds dense cells in all subspaces.

8.13. Present conditions under which density-based clustering is more suitable than partitioning-based clustering and hierarchical clustering. Give application examples to support your argument.

8.14. Give an example of how specific clustering methods can be *integrated*, for example, where one clustering algorithm is used as a preprocessing step for another. In addition, provide reasoning as to why the integration of two methods may sometimes lead to improved clustering quality and efficiency.

8.15. Clustering is recognized as an important data mining task with broad applications. Give one application example for each of the following cases:

- **a.** An application that uses clustering as a major data mining function.
- **b.** An application that uses clustering as a preprocessing tool for data preparation for other data mining tasks.

8.16. Data cubes and multidimensional databases contain nominal, ordinal, and numeric data in hierarchical or aggregate forms. Based on what you have learned about the clustering methods, design a clustering method that finds clusters in large data cubes effectively and efficiently.

8.17. Describe each of the following clustering algorithms in terms of the following criteria: (1) shapes of clusters that can be determined; (2) input parameters that must be specified; and (3) limitations.

- **a.** k-means
- **b.** k-medoids
- **c.** BIRCH
- **d.** DBSCAN*

8.18. Human eyes are fast and effective at judging the quality of clustering methods for 2-D data. Can you design a data visualization method that may help humans visualize data clusters and judge the clustering quality for 3-D data? What about for even higher-dimensional data?

8.19. Discuss how well purity, entropy, and the method using Jacaard coefficient satisfy the four essential requirements for extrinsic clustering evaluation methods.

8.8 Bibliographic notes

Clustering has been extensively studied for over 40 years and across many disciplines due to its broad applications. Most books on pattern classification and machine learning contain chapters on cluster analysis or unsupervised learning. Several textbooks are dedicated to the methods of cluster analysis, including Hartigan [Har75]; Jain and Dubes [JD88]; Kaufman and Rousseeuw [KR90]; and Arabie, Hubert, and De Sorte [AHS96]. There are also many survey articles on different aspects of clustering methods. Recent ones include Jain, Murty, and Flynn [JMF99]; Parsons, Haque, and Liu [PHL04]; Xu and Wunsch [XW05]; Jain [Jai10]; Greenlaw and Kantabutra [GK13]; Xu and Tian [XT15]; and Berkhin [Ber06].

For partitioning methods, the k-means algorithm was first introduced by Lloyd [Llo57], and then by MacQueen [Mac67]. Arthur and Vassilvitskii [AV07] presented the k-means++ algorithm. A filtering algorithm, which uses a spatial hierarchical data index to speed up the computation of cluster means, is given in Kanungo et al. [KMN+02].

The k-medoids algorithms of PAM and CLARA were proposed by Kaufman and Rousseeuw [KR90]. The k-modes (for clustering nominal data) and k-prototypes (for clustering hybrid data) algorithms were proposed by Huang [Hua98]. The k-modes clustering algorithm was also proposed independently by Chaturvedi, Green, and Carroll [CGC94,CGC01]. The CLARANS algorithm was proposed by Ng and Han [NH94]. Ester, Kriegel, and Xu [EKX95] proposed techniques for further improvement of the performance of CLARANS using efficient spatial access methods such as R$*$-tree and focusing techniques. A k-means–based scalable clustering algorithm was proposed by Bradley, Fayyad, and Reina [BFR98]. The kernel k-means method was developed by Dhillon, Guan, and Kulis [DGK04].

An early survey of agglomerative hierarchical clustering algorithms was conducted by Day and Edelsbrunner [DE84]. Murtagh and Contreras [MC12] provided a more recent survey on hierarchical clustering. Zhao, Karypis, and Fayyad [ZKF05] surveyed the hierarchical clustering algorithms for document databases. Agglomerative hierarchical clustering, such as AGNES, and divisive hierarchical clustering, such as DIANA, were introduced by Kaufman and Rousseeuw [KR90]. Rohlf [Roh73] developed the essential idea of hierarchical clustering using the minimum spanning tree. An interesting direction for improving the clustering quality of hierarchical clustering methods is to integrate hierarchical clustering with distance-based iterative relocation or other nonhierarchical clustering methods. For example, BIRCH, by Zhang, Ramakrishnan, and Livny [ZRL96], first performs hierarchical clustering with a CF-tree before applying other techniques. Hierarchical clustering can also be performed by sophisticated linkage analysis, transformation, or nearest-neighbor analysis, such as CURE by Guha, Rastogi, and Shim [GRS98]; ROCK (for clustering nominal attributes) by Guha, Rastogi, and Shim [GRS99]; and Chameleon by Karypis, Han, and Kumar [KHK99].

Ward [War63] proposed the Ward's criterion. Murtagh and Legendre [ML14] surveyed how the Ward's criterion is implemented. Lance and Williams [LW67] proposed the Lance-Williams algorithm. A probabilistic hierarchical clustering framework following normal linkage algorithms and using probabilistic models to define cluster similarity was developed by Friedman [Fri03] and Heller and Ghahramani [HG05].

For density-based clustering methods, DBSCAN was proposed by Ester, Kriegel, Sander, and Xu [EKSX96]. Campello, Moulavi, Zimek, and Sander [CMZS15] developed both DBSCAN* and HDBSCAN. Ankerst, Breunig, Kriegel, and Sander [ABKS99] developed OPTICS, a cluster-ordering method that facilitates density-based clustering without worrying about parameter specification. The DENCLUE algorithm, based on a set of density distribution functions, was proposed by Hinneburg and Keim [HK98]. Hinneburg and Gabriel [HG07] developed DENCLUE 2.0, which includes a new hill-climbing procedure for Gaussian kernels that adjusts the step size automatically.

STING, a grid-based multiresolution approach that collects statistical information in grid cells, was proposed by Wang, Yang, and Muntz [WYM97]. WaveCluster, developed by Sheikholeslami, Chatterjee, and Zhang [SCZ98], is a multiresolution clustering approach that transforms the original feature space by wavelet transform.

Scalable methods for clustering nominal data were studied by Gibson, Kleinberg, and Raghavan [GKR98]; Guha, Rastogi, and Shim [GRS99]; and Ganti, Gehrke, and Ramakrishnan [GGR99]. There are also many other clustering paradigms. For example, fuzzy clustering methods are discussed in Kaufman and Rousseeuw [KR90], Bezdek [Bez81], and Bezdek and Pal [BP92].

For high-dimensional clustering, an Apriori-based dimension-growth subspace clustering algorithm called CLIQUE was proposed by Agrawal, Gehrke, Gunopulos, and Raghavan [AGGR98]. It integrates density-based and grid-based clustering methods.

Recent studies have proceeded to clustering stream data (Babcock et al. [BBD+02]). A k-median–based data stream clustering algorithm was proposed by Guha, Mishra, Motwani, and O'Callaghan [GMMO00] and by O'Callaghan et al. [OMM+02]. A method for clustering evolving data streams was proposed by Aggarwal, Han, Wang, and Yu [AHWY03]. A framework for projected clustering of high-dimensional data streams was proposed by Aggarwal, Han, Wang, and Yu [AHWY04].

Clustering evaluation is discussed in a few monographs and survey articles such as Jain and Dubes [JD88] and Halkidi, Batistakis, and Vazirgiannis [HBV01]. The Hopkins statistic was proposed by Hopkins and Skellam [HS54]. The problem of determining the number of clusters in a data set was discussed by Sugar and James [SJ03] and Cordeiro De Amorim and Hennig [CH15b], for example.

The extrinsic methods for clustering quality evaluation are extensively explored. Some recent studies include Meilă [Mei03,Mei05] and Amigó, Gonzalo, Artiles, and Verdejo [AGAV09]. The four essential criteria introduced in this chapter are formulated in Amigó, Gonzalo, Artiles, and Verdejo [AGAV09], whereas some individual criteria were also mentioned earlier, for example, in Meilă [Mei03] and Rosenberg and Hirschberg [RH07]. Bagga and Baldwin [BB98] introduced the BCubed metrics. The silhouette coefficient is described in Kaufman and Rousseeuw [KR90].