Chapter 2: Association Rules and Sequential Patterns

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

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

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

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

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

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

2.1 Basic Concepts of Association Rules

The problem of mining association rules can be stated as follows:

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of **items**. Let $T = (t_1, t_2, \ldots, t_n)$ be a set of

transactions (the database), where each transaction t_i is a set of items such that $t_i \subseteq I$. An **association rule** is an implication of the form,

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

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

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

{Beef, Chicken, Cheese},

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

Beef, Chicken $\rightarrow$ Cheese,

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

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

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

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

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

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

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

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

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

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

Example 2: Fig. 1 shows a set of 7 transactions. Each transaction t_i is a set of items purchased in a basket in a store by a customer. The set I is the set of all items sold in the store.

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

Fig. 1. An example of a transaction set

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

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

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

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

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

We note that the data representation in the transaction form of Fig. 1 is a simplistic view of shopping baskets. For example, the quantity and price of each item are not considered in the model.

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

A large number of association rule mining algorithms have been reported in the literature, which have different mining efficiencies. Their re-

sulting sets of rules are, however, all the same based on the definition of association rules. That is, given a transaction data set *T*, a minimum support and a minimum confidence, the set of association rules existing in *T* is uniquely determined. Any algorithm should find the same set of rules although their computational efficiencies and memory requirements may be different. The best known mining algorithm is the **Apriori Algorithm** proposed in [11], which we study next.

2.2 Apriori Algorithm

The Apriori algorithm works in two steps:

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

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

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

Below, we discuss the two steps in turn.

2.2.1 Frequent Itemset Generation

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

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

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

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

```
Algorithm Apriori(T)
1   C1 ← init-pass(T);                          // the first pass over T
2   F1 ← {f | f ∈ C1, f.count/n ≥ minsup};      // n is the no. of transactions in T
3   for (k = 2; Fk-1 ≠ ∅; k++) do               // subsequent passes over T
4     Ck ← candidate-gen(Fk-1);
5     for each transaction t ∈ T do             // scan the data once
6       for each candidate c ∈ Ck do
7         if c is contained in t then
8           c.count++;
9       end
10    end
11    Fk ← {c ∈ Ck | c.count/n ≥ minsup}
12  end
13  return F ← ⋃k Fk;
```

Fig. 2. The Apriori Algorithm for generating all frequent itemsets

```
Function candidate-gen(Fk-1)
1   Ck ← ∅;                                     // initialize the set of candidates
2   forall f1, f2 ∈ Fk-1                        // traverse all pairs of frequent itemsets
3     with f1 = {i1, … , ik-2, ik-1}            // that differ only in the last item
4     and  f2 = {i1, … , ik-2, i'k-1}
5     and ik-1 < i'k-1 do                       // according to the lexicographic order
6       c ← {i1, …, ik-1, i'k-1};               // join the two itemsets f1 and f2
7       Ck ← Ck ∪ {c};                          // add the new itemset c to the candidates
8       for each (k-1)-subset s of c do
9         if (s ∉ Fk-1) then
10          delete c from Ck;                   // delete c from the candidates
11      end
12  end
13  return Ck;                                  // return the generated candidates
```

Fig. 3. The *candidate-gen* function

…, $w[k]$, where $w[1] < w[2] < \ldots < w[k]$ according to the total order.

The Apriori algorithm for frequent itemset generation, which is given in Fig. 2, is based on **level-wise search**. It generates all frequent itemsets by making multiple passes over the data. In the first pass, it counts the supports of individual items (line 1) and determines whether each of them is frequent (line 2). F_1 is the set of frequent 1-itemsets. In each subsequent pass k, there are three steps:

1. It starts with the seed set of itemsets F_{k-1} found to be frequent in the (k-1)-th pass. It uses this seed set to generate **candidate itemsets** C_k (line 4), which are possible frequent itemsets. This is done using the candidate-gen() function.

2. The transaction database is then scanned and the actual support of each candidate itemset c in C_k is counted (lines 5-10). Note that we do not need to load the whole data into memory before processing. Instead, at any time, only one transaction resides in memory. This is a very important feature of the algorithm. It makes the algorithm scalable to huge data sets, which cannot be loaded into memory.
3. At the end of the pass or scan, it determines which of the candidate itemsets are actually frequent (line 11).

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

Candidate-gen function: The candidate generation function is given in Fig. 3. It consists of two steps, the **join step** and the **pruning step**.

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

Pruning step (lines 8-11 in Fig. 3): A candidate c from the join step may not be a final candidate. This step determines whether all the k-1 subsets (there are k of them) of c are in F_{k-1}. If anyone of them is not in F_{k-1}, c cannot be frequent according to the downward closure property, and is thus deleted from C_k.

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

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

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

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

After the pruning step, we have only:

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

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

Example 4: Let us see a complete running example of the Apriori Algorithm based on the transactions in Fig. 1. We use minsup = 30%.

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

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

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

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

C_3: {{Chicken, Clothes, Milk}}

Note: {Beef, Chicken, Cheese} is also produced in line 6 of Fig. 3. However, {Chicken, Cheese} is not in F_2, and thus the itemset {Beef, Chicken, Cheese} is not included in C_3.

F_3: {{Chicken, Clothes, Milk}:3} ■

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

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

huge number of itemsets (and rules), tens of thousands, or more, which makes it hard for the user to analyze them to find those truly useful ones. This is called the **interestingness** problem. Researchers have proposed several methods to tackle this problem [e.g., 307, 313, 438, 463].

An efficient implementation of the Apriori algorithm involves sophisticated data structures and programming techniques, which are beyond the scope of this book. Apart from the Apriori algorithm, there is a large number of other algorithms, e.g., FP-growth [201] and many others.

2.2.2 Association Rule Generation

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

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

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

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

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

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

Thus, for a given frequent itemset f, if a rule with consequent α holds, then so do rules with consequents that are subsets of α. This is similar to the downward closure property that, if an itemset is frequent, then so are all its subsets. Therefore, from the frequent itemset f, we first generate all

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

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

Fig. 4. The association rule generation algorithm.

rules with one item in the consequent. We then use the consequents of these rules and the function candidate-gen() (Fig. 3) to generate all possible consequents with two items that can appear in a rule, and so on. An algorithm using this idea is given in Fig. 4. Note that all 1-item consequent rules (rules with one item in the consequent) are first generated in line 2 of the function genRules(). The confidence is computed using Equation (3).

Example 5: We again use transactions in Fig. 1, minsup = 30% and minconf = 80%. The frequent itemsets are as follows (see Example 4):

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

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

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

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

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

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

2.3 Data Formats for Association Rule Mining

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

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

Example 6: The table data in Fig. 5(A) can be converted to the transaction data in Fig. 5(B). Each attribute-value pair is considered an **item**. Using only values is not appropriate in the transaction form because different attributes may have the same values. For example, without including attribute names, the value a for Attribute1 and Attribute2 are not distinguishable. After the conversion, Fig. 5(B) can be used in association rule mining. ■

If any attribute takes numerical values, it becomes complex. We need to first discretize its value range into intervals, and treat each interval as a categorical value. For example, an attribute's value range is from 1-100, i.e., [1, 100]. We may want to divide it into 5 equal-sized intervals, 1-20, 21-40, 41-60, 61-80, and 80-100. Each interval is then treated as a categorical value. Discretization can be done manually based on expert knowledge or automatically. There are several existing algorithms [138, 446].

A point to note is that for a table data set, the join step of the candidate generation function (Fig. 3) needs to be slightly modified in order to ensure that it does not join two itemsets to produce a candidate itemset containing two items from the same attribute.

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

Attribute1	Attribute2	Atribute3
a	a	x
b	n	y

(A) Table Data

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

(B) Transaction Data

Fig. 5. From a relational table and a transaction set

2.4 Mining with Multiple Minimum Supports

The key element that makes association rule mining practical is the minsup threshold. It is used to prune the search space and to limit the number of rules generated. However, using only a single minsup implicitly assumes that all items in the data are of the same nature and/or have similar frequencies in the database. This is often not the case in real-life applications. In many applications, some items appear very frequently in the data, while some other items rarely appear. If the frequencies of items vary a great deal, we will encounter two problems [306]:

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

Example 7: In a supermarket transaction data set, in order to find rules involving those infrequently purchased items such as FoodProcessor and CookingPan (they generate more profits per item), we need to set the minsup to very low (say, 0.5%). We may find the following useful rule:

FoodProcessor $\rightarrow$ CookingPan [sup = 0.5%, conf = 60%]

However, this low minsup may also cause the following meaningless rule to be found:

Bread, Cheese, Milk $\rightarrow$ Yogurt [sup = 0.5%, conf = 60%]

Knowing that 0.5% of the customers buy the 4 items together is useless

because all these items are frequently purchased in a supermarket and each of them makes little profit. Worst still, it may cause combinatorial exploration! For the rule to be useful, the support needs to be much higher. In some applications, such a rule may not be useful at all no matter how high its support may be because it is known that customers frequently buy these food items together. Then the question is whether the algorithm can automatically suppress the rule, i.e., not to generate it. ■

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

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

A better solution is to allow the user to specify multiple minimum supports, i.e., to specify a different **minimum item support** (**MIS**) to each item. Thus, different rules need to satisfy different minimum supports depending on what items are in the rules. This model thus enables us to achieve our objective of producing rare item rules without causing frequent items to generate too many meaningless rules.

An important by-product of this model is that it enables the user to easily instruct the algorithm not to generate any rules involving some items by setting their MIS values to 100% (or 101% to be safe). This is useful because in practice the user may only be interested in certain types of rules.

2.4.1 Extended Model

To allow multiple minimum supports, the original model in Section 2.1 needs to be extended. In the extended model, the minimum support of a rule is expressed in terms of **minimum item supports** (**MIS**) of the items that appear in the rule. That is, each item in the data can have a MIS value

specified by the user. By providing different MIS values for different items, the user effectively expresses different support requirements for different rules. It seems that specifying a MIS value for each item is a difficult task. This is not so as we will see at the end of Section 2.4.2.

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

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

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

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

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

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

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

The following rule doesn't satisfy its minimum support:

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

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

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

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

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

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

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

If we find that itemset {1, 2} has 9% of support at level 2, then it does not satisfy either MIS(1) or MIS(2). Using the Apriori algorithm, this itemset is discarded since it is not frequent. Then, the potentially frequent itemsets {1, 2, 3} and {1, 2, 4} will not be generated for level 3. Clearly, itemsets {1, 2, 3} and {1, 2, 4} may be frequent because MIS(3) is only 5% and MIS(4) is 6%. It is thus wrong to discard {1, 2}. However, if we do not discard {1,

2}, the downward closure property is lost. ■

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

2.4.2 Mining Algorithm

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

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

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

Let F_k denote the set of frequent k-itemsets. Each itemset w is of the following form, $\{w[1], w[2], \ldots, w[k]\}$, which consists of items, $w[1]$, $w[2]$, …, $w[k]$, where $MIS(w[1]) \leq MIS(w[2]) \leq \ldots \leq MIS(w[k])$. The algorithm MSapriori is given in Fig. 6. Line 1 performs the sorting on I according to the MIS value of each item (stored in MS). Line 2 makes the first pass over the data using the function init-pass(), which takes two arguments, the data set T and the sorted items M, to produce the seeds L for generating candidate itemsets of length 2, i.e., C_2. init-pass() has two steps:

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

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

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

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

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

Fig. 6. The MSapriori algorithm

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

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

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

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

Level2-candidate-gen function: It takes an argument L, and returns a superset of the set of all frequent 2-itemsets. The algorithm is given in Fig. 7.

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

Fig. 7. The level2-candidate-gen function

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

Fig. 8. The MScandidate-gen function

Example 11: Let us continue with Example 10. Recall the MIS values of the four items are (in Example 9):

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

The level2-candidate-gen() function in Fig. 7 produces

C_2 = {{3, 1}, {3, 2}}

{1, 2} is not a candidate because the support count of item 1 is only 9 (or 9%), less than MIS(1) (= 10%). Hence, {1, 2} cannot be frequent. ■

Note that we must use L rather than F_1 because F_1 does not contain those items that may satisfy the MIS of an earlier item (in the sorted order) but not the MIS of itself, e.g., item 1. Using L, the problem discussed in Section 2.4.2 is solved for C_2.

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

For each (k-1)-subset s of c, if s is not in F_{k-1}, c can be deleted from C_k. However, there is an exception, which is when s does not include $c[1]$ (there is only one such s). That is, the first item of c, which has the lowest MIS value, is not in s. Even if s is not in F_{k-1}, we cannot delete c because we cannot be sure that s does not satisfy MIS(c[1]), although we know that it does not satisfy MIS(c[2]), unless MIS(c[2]) = MIS(c[1]) (line 9).

Example 12: Let F_3 = {{1, 2, 3}, {1, 2, 5}, {1, 3, 4}, {1, 3, 5}, {1, 4, 5}, {1, 4, 6}, {2, 3, 5}}. Items in each itemset are in the sorted order. The join step produces

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

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

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

Example 13: Given the following 7 transactions,

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

and MIS(Milk) = 50%, MIS(Bread) = 70%, and 25% for all other items. We obtain the following frequent itemsets

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

To conclude this sub-section, let us discuss some interesting issues:

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

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

   MIS(Beef) = 101%, and MIS(Chicken) = 101%.

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

   In many applications, this feature is useful because the user is often only interested in certain types of itemsets or rules. Then, those irrelevant itemsets and rules should not be generated.
3. Do not mix very rare items with very frequent items: In some applications, it may not make sense to have very rare items and very frequent items appearing in the same itemset, e.g., bread and television in a store. To restrict this, we can introduce a parameter to limit the difference between the smallest MIS and the largest MIS values of the items in an itemset so that it will not exceed a user-specified threshold. The mining algorithm can be slightly changed to accommodate this parameter.

2.4.3 Rule Generation

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

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

Example 15: Recall in Example 8, we have

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

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

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

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

Clothes, Bread → Shoes

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

Clothes → Shoes, Bread
Bread → Shoes, Clothes

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

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

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

The lemma indicates that we need to record the support count of $f - \{a\}$. This can be easily achieved by lines 13-14 in MSapriori (Fig. 6). A similar rule generation function as genRules() in the Apriori algorithm can be designed to generate rules with multiple minimum supports.

2.5 Mining Class Association Rules

The mining models studied so far do not use any targets. That is, any item can appear as a consequent or a condition of a rule. However, in some applications, the user is only interested in rules with a **target item** on the right-hand-side (consequent). For example, the user has a collection of text documents from some topics (target items). He/she wants to find out what words are correlated with each topic. As another example, the user has a data set that contains attribute settings and modes of operations of cellular phones. The modes of operations are *normal* and *abnormal*. He/she wants to know what attribute values are associated with abnormal operations in order to find possible causes of such undesirable situations.

2.5.1 Problem Definition

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

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

The definitions of **support** and **confidence** are the same as those for normal association rules. In general, a class association rule is different from a normal association rule in two ways:

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

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

Example 16: Fig. 9 shows a data set which has 7 text documents. Each document is a transaction and consists of a set of keywords. Each transaction is also labeled with a topic class (education or sport). We have

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

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

Fig. 9. An example of a data set

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

Student, School $\rightarrow$ Education [sup= 2/7, conf = 2/2]
Game $\rightarrow$ Sport [sup= 2/7, conf = 2/3] ■

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

2.5.2 Mining Algorithm

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

(*condset*, *y*)

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

$condset \rightarrow y$,

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

Ruleitems that satisfy the minsup are called **frequent** ruleitems, while the rest are called infrequent ruleitems. For example, ({Student, School}, Education) is a ruleitem in T of Fig. 9. The support count of the condset

Algorithm CARapriori (*T*)

```
1   C1 ← init-pass(T);                          // the first pass over T
2   F1 ← {f | f ∈ C1, f. rulesupCount / n ≥ minsup};
3   CAR1 ← {f | f ∈ F1, f.rulesupCount / f.condsupCount ≥ minconf};
4   for (k = 2; Fk-1 ≠ ∅; k++) do
5       Ck ← CARcandidate-gen(Fk-1);
6       for each transaction t ∈ T do
7           for each candidate c ∈ Ck do
8               if c.condset is contained in t then  // c is a subset of t
9                   c.condsupCount++;
10                  if t.class = c.class then
11                      c.rulesupCount++
12          end
13      end
14      Fk ← {c ∈ Ck | c.rulesupCount / n ≥ minsup};
15      CARk ← {f | f ∈ Fk, f.rulesupCount / f.condsupCount ≥ minconf};
16  end
17  return CAR ← ⋃k CARk;
```

Fig. 10. The CARapriori algorithm

{Student, School} is 2, and the support count of the ruleitem is also 2. Then the support of the ruleitem is 2/7(= 28.6%), and the confidence of the ruleitem is 100%. If minsup = 10%, then the ruleitem satisfies the minsup threshold. We say that it is frequent. If minconf = 80%, then the ruleitem satisfies the minconf threshold. We say that the ruleitem is **confident**. We thus have the class association rule:

Student, School → Education [sup= 2/7, conf = 2/2]

The rule generation algorithm, called **CARapriori**, is given in Fig. 10, which is based on the Apriori algorithm. Like the Apriori algorithm, CARapriori generates all the frequent ruleitems by making multiple passes over the data. In the first pass, it computes the support count of each 1-ruleitem (containing only one item in its condset) (line 1). The set of all 1-candidate ruleitems considered is:

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

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

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

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

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

Example 17: Let us work on a complete example using our data in Fig. 9. We set minsup = 15%, and minconf = 70%

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

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

CAR_1:

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

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

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

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

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

Class association rules turned out to be quite useful. First, they can be emplyed to build machine learning models (classification models) as we will see in the next chapter. Second, they can be used to find useful or actionable knowledge in an application domain. For example, a deployed data mining system based on such rules is reported in [313]. The system is used in Motorola Inc. for analyzing its engineering and service data sets to identify causes of product issues. Class association rules are particularly suited for such kind of **diagnostic data mining** (see [313] for details).

We note that for the above applications, the data sets used are normally relational tables. They need to be converted to transaction forms before mining. We can use the method described in Section 2.3 for this purpose.

Example 18: In Fig. 11(A), the data set has three data attributes and a class attribute with two possible values, positive and negative. It is converted to the transaction data in Fig. 11(B). Notice that for each class, we only use its original value. There is no need to attach the attribute "Class" because there is no ambiguity. As discussed in Section 2.3, for each numeric attribute, its value range needs to be discretized into intervals either manually or automatically before conversion and rule mining. There are many discretization algorithms. Interested readers are referred to [138]. ■

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

(A) Table Data

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

(B) Transaction Data

Fig. 11. Converting a table data set (A) to a transactional data set (B).

2.5.3 Mining with Multiple Minimum Supports

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

1. **Multiple minimum class supports**: The user can specify different minimum supports for different classes. For example, the user has a data

set with two classes, Yes and No. Based on the application requirement, he/she may want all rules of class Yes to have the minimum support of 5% and all rules of class No to have the minimum support of 20%.

2. **Multiple minimum item supports** The user can specify a minimum item support for every item (either a class item/label or a non-class item). This is more general and is similar to normal association rule mining discussed in Section 2.4.

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

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

2.6 Sequential Pattern Mining

Association rule mining does not consider the order of transactions. However, in many applications such orderings are significant. For example, in market basket analysis, it is interesting to know whether people buy some items in sequence, e.g., buying bed first and then buying bed sheets some time later. In natural language processing or text mining, considering the ordering of words in a sentence is vital in finding language or linguistic patterns. For such applications, association rules are no longer appropriate. Sequential patterns are needed. Sequential patterns have been used extensively in Web usage mining (see Chapter 12) for finding navigational patterns of users in the Web site. They have also been applied to finding linguistic patterns for opinion mining (see Chapter 11).

2.6.1 Problem Definition

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of items. A **sequence** is an ordered list of itemsets. Recall an **itemset** X is a non-empty set of items $X \subseteq I$. We denote a sequence s by $\langle a_1 a_2 \ldots a_r \rangle$, where a_i is an itemset, which is also called an **element** of s. We denote an element (or an itemset) of a sequence by $\{x_1, x_2, \ldots, x_k\}$, where $x_j \in I$ is an item. We assume without loss of generality that items in an element of a sequence are in **lexicographic order**. An item can occur only once in an element of a sequence, but can occur multiple times in different elements. The **size** of a sequence is the number of ele-

ments (or itemsets) in the sequence. The **length** of a sequence is the number of items in the sequence. A sequence of length k is called ***k*-sequence**. If an item occurs multiple times in different elements of a sequence, each occurrence contributes to the value of k. A sequence $s_1 = \langle a_1 a_2 \ldots a_r \rangle$ is a **subsequence** of another sequence $s_2 = \langle b_1 b_2 \ldots b_v \rangle$, or s_2 is a **supersequence** of s_1, if there exist integers $1 \leq j_1 < j_2 < \ldots < j_{r-1} < j_r \leq v$ such that $a_1 \subseteq b_{j_1}$, $a_2 \subseteq b_{j_2}$, …, $a_r \subseteq b_{j_r}$. We also say that s_2 **contains** s_1.

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

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

Example 20: We use the market basket analysis as an example. Each sequence in this context represents an ordered list of transactions of a particular customer. A transaction is a set of items that the customer purchased at a time (called the transaction time). Then transactions in the sequence are ordered by increasing transaction time. Table 1 shows a transaction database which is already sorted according customer ID (the major key) and transaction time (the minor key). Table 2 gives the data sequences (also called **customer sequences**). Table 3 gives the output sequential patterns with the minimum support of 25%, i.e., 2 customers. ■

Table 1. A set of transactions sorted by customer ID and transaction time

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

Table 2. Data sequences produced from the transaction database in Table 1.

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

Table 3. The final output sequential patterns

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

As noted earlier, in text processing, frequent sequences can represent language patterns. To mine such patterns, we treat each sentence as a sequence. For example, the sentence "*this is a great movie*" can be represented with the sequence ⟨{this}{is}{a}{great}{movie}⟩. If we also want to consider the part-of-speech tags of each word in pattern mining, the sentence can be represented with ⟨{PRN, this}{VB, is}{DT, a}{JJ, great}{NN, movie}⟩, where PRN stands for pronoun, VB for verb, DT for determiner, JJ for adjective and NN for noun. A set of such sequences can be used to find language patterns that involve both part-of-speech tags and actual words. We will see an application in Chapter 11.

2.6.2 Mining Algorithm

This section describes an algorithm called Apriori-SPM for mining frequent sequences (or sequential patterns) using a single **minimum support**. It works in almost the same way as the Apriori algorithm. We still use F_k to store the set of all frequent k-sequences, and C_k to store the set of all candidate k-sequences. The algorithm is given in Fig. 12. The main difference is in the candidate generation, candidate-gen-SPM(), which is given in Fig. 13. We use an example to explain the working of the function.

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

Algorithm Apriori-SPM(S)

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

Fig. 12. The Apriori-SPM Algorithm for generating sequential patterns

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

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

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

Fig. 13. The candidate-gen-SPM function

Table 4. Candidate generation: an example

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

There are several sequential pattern mining algorithms. The algorithm described in this section is based on the GSP algorithm in [445]. Some recent algorithms have improved its efficiency [e.g., 27, 200, 394]

2.6.3 Mining with Multiple Minimum Supports

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

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

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

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

The new algorithm, called **MSapriori-SPM**, is given in Fig. 14. It generalizes the Apriori-SPM algorithm in Fig. 12. Like Apriori-SPM, MSapriori-SPM is also based on level-wise search. Line 1 sorts the items in ascending order according to their MIS values stored in *MS*. Line 2 makes the first pass over the sequence data using the function *init-pass*, which performs the same function as that in MSapriori to produce the seeds set L for generating the set of candidate sequences of length 2, i.e., C_2. Frequent 1-sequences (F_1) are obtained from L (line 3).

For each subsequent pass, the algorithm works similarly to MSapriori. The function level2-candidate-gen-SPM() can be designed based on level2-candidate-gen in MSapriori and the join step in Fig. 13. MScandidate-gen-SPM() is, however, complex, which we will discuss shortly.

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

```
Algorithm MSapriori-SPM(S, MS)                    // MS stores all MIS values
1   M ← sort(I, MS);                              // according to MIS(i)'s stored in MS
2   L ← init-pass(S, MS);                         // make the first pass over S
3   F1 ← {⟨{l}⟩ | l ∈ L, l.count/n ≥ MIS(l)};     // n is the size of S
4   for (k = 2; Fk-1 ≠ ∅; k++) do
5     if k = 2 then
6       Ck ← level2-candidate-gen-SPM(L)
7     else Ck ← MScandidate-gen-SPM(Fk-1)
8     end
9     for each data sequence s ∈ S do
10      for each candidate c ∈ Ck do
11        if c is contained in s then
12          c.count++
13        if c' is contained in s, where c' is c after an occurrence of
              c.minMISItem is removed from c then
14          c.tailCount++
15      end
16    end
17    Fk ← {c ∈ Ck | c.count/n ≥ MIS(c.minMISItem)}
18  end
19  Return F = ⋃k Fk;
```

Fig. 14. The MSapriori-SPM algorithm

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

Example 22: Assume we have a sequence $s_1 = \langle\{1, 2\}\{4\}\rangle$ in F_3, from which we want to generate candidate sequences for the next level. Suppose that item 1 has the lowest MIS value in s_1. We use the candidate generation function in Fig. 13. Assume also that the sequence $s_2 = \langle\{2\}\{4, 5\}\rangle$ is not in F_3 because its minimum support is not satisfied. Then we will not generate the candidate $\langle\{1, 2\}\{4, 5\}\rangle$. However, $\langle\{1, 2\}\{4, 5\}\rangle$ can be frequent because items 2, 4, and 5 may have higher MIS values than item 1. ■

To deal with this problem, let us make an observation. The problem only occurs when the first item in the sequence s_1 or the last item in the sequence s_2 is the only item with the lowest MIS value, i.e., no other item in s_1 (or s_2) has the same lowest MIS value. If the item (say x) with the lowest MIS value is not the first item in s_1, then s_2 must contain x, and the candidate generation function in Fig. 13 will still be applicable. The same rea-

soning goes for the last item of s_2. Thus, we only need special treatment for these two cases.

Let us see how to deal with the first case, i.e., the first item is the only item with the lowest MIS value. We use an example to develop the idea. Assume we have the frequent 3-sequence of s_1 = ⟨{1, 2}{4}⟩. Based on the algorithm in Fig. 13, s_1 may be extended to generate two possible candidates using ⟨{2}{4}{x}⟩ and ⟨{2}{4, x}⟩

c_1 = ⟨{1, 2}{4}{x}⟩ and c_2 = ⟨{1, 2}{4, x}⟩,

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

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

Let us see the same problem for the case where the last item has the only lowest MIS value. Again, we use an example to illustrate. Assume we have the frequent 3-sequence s_2 = ⟨{3, 5}{1}⟩. It can be extended to produce two possible candidates based on the algorithm in Fig. 13,

c_1 = ⟨{x}{3, 5}{1}⟩, and c_2 = ⟨{x, 3, 5}{1}⟩.

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

The MScandidate-gen-SPM() function is given in Fig. 15, which is self-explanatory. Some special treatments are needed for 2-sequences because the same s_1 (or s_2) may generate two candidate sequences. We use two examples to show the working of the function.

Example 23: Consider the items 1, 2, 3, 4, 5, and 6 with their MIS values,

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

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

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

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

1 **Join Step**:

2 **if** the MIS value of the first item in a sequence s_1 is less than (<) the MIS value of every other item in s_1 **then**

Sequence s_1 joins with s_2 if (1) the subsequences obtained by dropping the second item of s_1 and the last item of s_2 are the same, (2) the MIS value of the last item of s_2 is greater than that of the first item of s_1. Candidate sequences are generated by extending s_1 with the last item of s_2:

- **if** the last item l in s_2 is a separate element **then**

  $\{l\}$ is appended at the end of s_1 as a separate element to form a candidate sequence c_1.

  if (the length and the size of s_1 are both 2) AND (the last item of s_2 is greater than the last item of s_1) **then** // maintain lexicographic order

  l is added at the end of the last element of s_1 to form another candidate sequence c_2.

- **else if** (the length of s_1 is 2 and the size of s_1 is 1) or (the length of s_1 is greater than 2) **then**

  the last item in s_2 is added at the end of the last element of s_1 to form the candidate sequence c_2.

3 **elseif** the MIS value of the last item in a sequence s_2 is less than (<) the MIS value of every other item in s_2 **then**

A similar method to the one above can be used in the reverse order.

4 **else** use the **Join Step** in Fig. 14

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

Fig. 15. The MScandidate-gen-SPM function

For sequence (a) (= s_1), item 1 has the lowest MIS value. It cannot join with sequence (b) because condition (1) in Fig. 15 is not satisfied. However, (a) can join with (c) to produce the candidate sequence, ⟨{1}{4}{5}{6}⟩. (a) can also join with (d) to produce ⟨{1}{4}{5, 6}⟩. (b) can join with (e) to produce ⟨{1}{4}{6}{3}⟩, which is pruned subsequently because ⟨{1}{4}{3}⟩ is infrequent. (d) and (e) can be joined to give ⟨{1}{5, 6}{3}⟩, but it is pruned because ⟨{1}{5}{3}⟩ does not exist. (e) can join with (f) to produce ⟨{1}{6}{3}{6}⟩ which is done in line 4 because both item 1 and item 3 in (e) have the same MIS value. However, it is pruned because ⟨{1}{3}{6}⟩ is infrequent. We do not join (d) and (g), although they can be joined based on the algorithm in Fig. 13, because the first item of (d) has the lowest MIS value and we use a different join method for such sequences.

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

Example 24: Now we consider generating candidates from frequent 2-sequenes, which is special as we noted earlier. We use the same items and MIS values in Example 23. The following frequent 2-sequences are in F_2 with their actual support counts attached after ":":

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

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

Finally, as in multiple minimum support association rule mining, the user can also instruct the algorithm to generate only certain sequential patterns and not generate others by setting MIS values of items suitably. The algorithm can also be modified so that very rare items and very frequent items will not appear in the same pattern.

2.6.4 Sequential Rules

In classic sequential pattern mining, no rules are generated. It only finds all frequent sequences (or sequential patterns). Let us define sequential rules here, which are analogous to association rules.

A **sequential rule** (**SR**) is an implication of the form

$$X \rightarrow Y,$$

where Y is a sequence and X is a **proper subsequence** of Y, i.e., X is a subsequence of Y and the length Y is greater than the length of X.

Given a minimum support and a minimum confidence, according to the downward closure property, all the rules can be generated from frequent sequences without going to the original sequence data. Let us see an example of a sequential rule found from the data sequences in Table 5.

Table 5. A sequence database for mining sequential rules

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

Example 25: Given the sequence database in Table 5, the minimum support of 30% and the minimum confidence of 30%, one of the sequential

rules found is the following,

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

Data sequences 1, 2 and 3 contain $\langle\{1\}\{7\}\rangle$, and data sequences 1 and 2 contain $\langle\{1\}\{3\}\{7, 8\}\rangle$. ■

If multiple minimum supports are used, we can use the frequent sequences found by MSapriori-SPM to generate all the rules.

2.6.5 Label Sequential Rules

Sequential rules may not be restrictive enough in some applications. We introduce a special kind of sequential rules called **label sequential rules**. A label sequential rule (LSR) is of the following form,

$$X \rightarrow Y,$$

where Y is a sequence and X is a sequence produced from Y by replacing some of its items with wildcards. A wildcard is denoted by an "*" which matches any item.

Example 26: Using the data sequence in Table 5 and the same minimum support and minimum confidence thresholds, we have a similar rule to that in Example 25,

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

Notice the confidence change compared to the rule in Example 25. The supports of the two rules are the same. In this case, data sequences 1, 2, and 4 contain $\langle\{1\}\{*\}\{7, *\}\rangle$, but only data sequences 1 and 2 contain $\langle\{1\}\{3\}\{7, 8\}\rangle$. ■

LSRs are useful because one may be interested in predicting some special items in an input data sequence, e.g., items 3 and 8 in the example. The confidence of the rule simply gives us the probability that the two "*"s are 3 and 8 given that an input sequence contains $\langle\{1\}\{*\}\{7, *\}\rangle$. We will show an application in Chapter 11.

Due to the use of the wildcards, frequent sequences alone are not sufficient for computing rule confidences. We need to scan the data.

2.6.6 Class Sequential Rules

Class sequential rules (CSR) are analogous to class association rules (CAR). Let S be a set of data sequences. Each sequence is also labeled with a class y. Let I be the set of all items in S, and Y be the set of all class

labels, $I \cap Y = \varnothing$. Thus, the input data D for mining is represented with $D = \{(s_1, y_1), (s_2, y_2), \ldots, (s_n, y_n)\}$, where s_i is a sequence and $y_i \in Y$ is its class. A **class sequential rule** (**CSR**) is an implication of the form

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

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

Example 27: Table 6 gives an example of a sequence database with five sequences and two classes, c_1 and c_2. Using the minimum support of 20% and the minimum confidence of 40%, one of the discovered CSRs is:

$$\langle\{1\}\{3\}\{7, 8\}\rangle \rightarrow c_1 \quad [\text{sup} = 2/5, \text{conf} = 2/3]$$

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

Table 6. An example sequence database for mining CSRs

	Data Sequence	Class
1	$\langle\{1\}\{3\}\{5\}\{7, 8, 9\}\rangle$	c_1
2	$\langle\{1\}\{3\}\{6\}\{7, 8\}\rangle$	c_1
3	$\langle\{1, 6\}\{9\}\rangle$	c_2
4	$\langle\{3\}\{5, 6\}\rangle$	c_2
5	$\langle\{1\}\{3\}\{4\}\{7, 8\}\rangle$	c_2

As in class association rule mining, we can modify the Aprior-SPM to produce an algorithm for mining all CSRs. Similarly, we can also use multiple minimum class supports and multiple minimum item supports as in class association rule mining.

Bibliographic Notes

Association rule mining was introduced in 1993 by Agrawal et al. [9]. Since then, numerous research papers have been published on the topic. As given a data set and a minimum support and a minimum confidence, the solution is unique, most papers improve the mining efficiency. The most well-known algorithm is the Apriori algorithm proposed by Agrawal and Srikant [11], which has been studied in this chapter. Another important algorithm is the **FP-growth** algorithm proposed by Han et al. [201]. This algorithm compresses the data and stores it in memory using a frequent pattern tree. It then mines all frequent itemsets without candidate generation.

Other notable algorithms include those by Agarwal et al. [2], Mannila et al. [322], Park et al. [390], Zaki et al. [520] and among others. An efficiency comparison of various algorithms is reported by Zheng et al. [541].

Apart from performance improvements, several variations of the original model were also proposed. Srikant and Agrawal [444], and Han and Fu [198] proposed two algorithms for mining **generalized association rules** or **multi-level association rules**. Liu et al. [306] extended the original model to take **multiple minimum supports**, which was also studied by Wang et al. [475], and Seno and Karypis [432]. Srikant et al. [447] proposed to mine association rules with **item constraints**, which restrict the rules that should be generated. Ng et al. [364] generalized the idea, which was followed by many subsequent papers in the area of **constrained rule mining**.

It is well known that association rule mining typically generates a huge number of rules. Bayado [39], and Lin and Kedem [297] introduced the problem of mining **maximal frequent itemsets**, which are itemsets with no frequent superset. Improved algorithms were reported in [2] and [67]. Since maximal pattern mining only finds longest patterns, the support information of their subsets, which are obvious also frequent, is not found. The next significant development was the mining of **closed frequent itemsets** given by Pasquier et al [391], Zaki and Hsiao [517], and Wang et al. [470]. Closed itemsets are better than maximal itemsets because closed itemsets provide a lossless concise representation of all frequent itemsets.

Other developments on association rules include **cyclic association rules** proposed by Ozden et al. [376] and **periodic patterns** by Yang et al. [502], **negative association rules** by Savasere [426] and Wu et al. [497], weighted association rules by Wang et al [479], association **rules with numerical variables** by Webb [481], **class association rules** by Liu et al. [305] and many others. Recently, Cong et al [100, 101] introduced association rule mining from **bioinformatics data**, which typically have a very large number of attributes (more than ten thousands) but only a very small number of records or transactions (less than 100).

Regarding sequential pattern mining, the first algorithm was proposed by Agrawal and Srikant [12], which was a direct application of the Apriori algorithm. Improvements were made subsequently by several researchers, e.g., Ayres et al. [27], Pei et al. [394], Srikant and Agrawal [445], etc. Mining sequential patterns and rules with multiple minimum supports is introduced in this book. Both label and class sequential rules are used in [232, 233] for mining comparative sentences and relations from text documents for opinion extraction.

Chapter 3: Supervised Learning

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

There are several types of supervised learning tasks. In this chapter, we focus on one particular type, namely, learning a target function that can be used to predict the values of a discrete class attribute. This type of learning has been the focus of the machine learning research and is perhaps also the most widely used learning paradigm in practice. This chapter introduces a number of such supervised learning techniques.

3.1 Basic Concepts

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

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

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

Example 1: Table 1 shows a small loan application data set. It has 4 attributes. The first attribute is Age, which has three possible values, young, middle and old. The second attribute is Has_Job, which indicates whether an applicant has a job. Its values are true (has a job), or false (does not have a job). The third attribute is Own_house, which shows whether an applicant owns a house. The fourth attribute is Credit_rating, which has three values, fair, good and excellent. The last column is the Class attribute, which shows whether each loan application was approved (denoted by Yes) or not (denoted by No) in the past.

Table 1: A loan application data set

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

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

Our learning task is called the **supervision learning** because the class labels (e.g., Yes and No values of the class attribute in Table 1) are pro-

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

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

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

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

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

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

We pause here to raises two important questions:

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

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

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

the future data is 9/15 = 0.6 as there are 9 `Yes` class examples out of the total of 15 examples in Table 1. The question is: can we do better with learning? If the learned model can indeed improve the accuracy, then the learning is said to be effective.

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

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

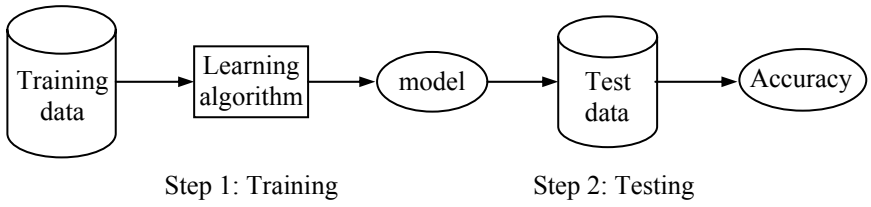


Fig. 1. The basic learning process: training and testing

So far we have assumed that the training and test data are available for learning. However, in many text and Web page related learning tasks, this is not true. Usually, we need to collect raw data, design attributes and compute attribute values from the raw data. The reason is that the raw data

in text and Web applications are often not suitable for learning either because their formats are not right or because there are no obvious attributes in the raw text documents and Web pages.

3.2 Decision Tree Induction

Decision tree learning is one of the most widely used techniques for classification. Its classification accuracy is competitive with other learning methods, and it is very efficient. The learned classification model is represented as a tree, called a **decision tree**. The techniques presented in this section are based on the C4.5 system from Quinlan [406].

Example 2: Fig. 2 shown a possible decision tree learned from the data in Table 1. The decision tree has two types of nodes, **decision nodes** (which are internal nodes) and **leaf nodes**. A decision node specifies some test (i.e., asks a question) on a single attribute. A leaf node indicates a class.

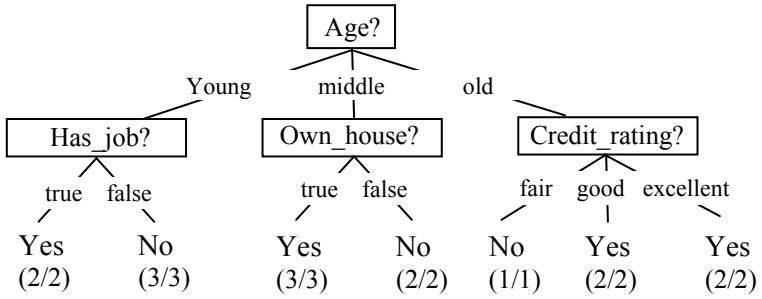


Fig. 2. A decision tree for the data in Table 1.

The root node of the decision tree in Fig. 2 is Age, which basically asks the question: what is the age of the applicant? It has three possible answers or **outcomes**, which are the three possible values of Age. These three values form three tree branches/edges. The other internal nodes have the same meaning. Each leaf node gives a class value (Yes or No). (*x*/*y*) below each class means that *x* out of *y* training examples that reach this leaf node have the class of the leaf. For instance, the class of the left most leaf node is Yes. Two training examples (examples 3 and 4 in Table 1) reach here and both of them are of class Yes. ■

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

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

Age	Has_Job	Own_house	Credit-Rating	Class
young	false	false	good	?

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

A decision tree is constructed by partitioning the training data so that the resulting subsets are as pure as possible. A **pure subset** is one that contains only training examples of a single class. If we apply all the training data in Table 1 on the tree in Fig. 2, we will see that the training examples reaching each leaf node form a subset of examples that have the same class as the class of the leaf. In fact, we can see that from the *x* and *y* values in (*x*/y). We will discuss the decision tree building algorithm in Section 3.2.1.

An interesting question is: Is the tree in Fig. 2 unique for the data in Table 1? The answer is no. In fact, there are many possible trees that can be learned from the data. For example, Fig. 3 gives another decision tree, which is much smaller and is also able to partition the training data perfectly according to their classes.

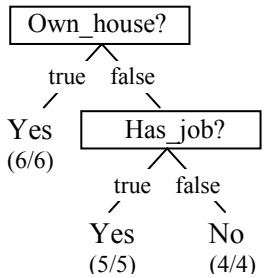


Fig. 3. A smaller tree for the data set in Table 1.

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

It is useful to note that in both Fig. 2 and Fig. 3, the training examples

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

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

Example 4: The tree in Fig. 3 generates three rules. "," means "and".

Own_house = true → Class =Yes [sup=6/15, conf=6/6]
Own_house = false, Has_job = true → Class = Yes [sup=5/15, conf=5/5]
Own_house = false, Has_job = false → Class = No [sup=4/15, conf=4/4]

We can see that these rules are of the same format as association rules. However, the rules above are only a small subset of the rules that can be found in the data of Table 1. For instance, the decision tree in Fig. 3 does not find the following rule:

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

Thus, we say that a decision tree only finds a subset of rules that exist in data, which is sufficient for classification. The objective of association rule mining is to find all rules subject to some minimum support and minimum confidence constraints. Thus, the two methods have different objectives. We will discuss these issues again in Section 3.4 when we show that association rules, more precisely, class association rules, can be used for classification as well, which is obvious.

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

We also say that a decision tree **generalizes** the data as a tree is a smaller (compact) description of the data, i.e., it captures the key regularities in the data. Then, the problem becomes building the best tree that is small and accurate. It turns out that finding the best tree that models the data is a NP-complete problem [226]. All existing algorithms use heuristic methods for tree building. Below, we study one of the most successful techniques.

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

Fig. 4. A decision tree learning algorithm

3.2.1 Learning Algorithm

As indicated earlier, a decision tree T simply partitions the training data set D into disjoint subsets so that each subset is as pure as possible (of the same class). The learning of a tree is typically done using the **divide-and-conquer** strategy that recursively partitions the data to produce the tree. At the beginning, all the examples are at the root. As the tree grows, the examples are sub-divided recursively. A decision tree learning algorithm is given in Fig. 4. For now, we assume that every attribute in D takes discrete values. This assumption is not necessary as we will see later.

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

path. In tree learning, each successive recursion chooses the **best attribute** to partition the data at the current node according to the values of the attribute. The best attribute is selected based on a function that aims to minimize the impurity after the partitioning (lines 7-11). In other words, it maximizes the purity. The key in decision tree learning is thus the choice of the **impurity function**, which is used in lines 7, 9 and 11 in Fig. 4. The recursive recall of the algorithm is in line 20, which takes the subset of training examples at the node for further partitioning to extend the tree.

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

3.2.2 Impurity Function

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

Example 5: Fig. 5 shows two possible root nodes for the data in Table 1.

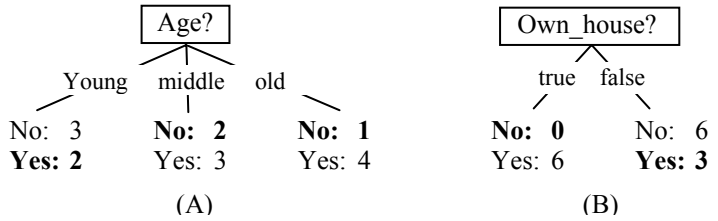


Fig. 5. Two possible root nodes or two possible attributes for the root node

Fig. 5(A) uses Age as the root node, and Fig. 5(B) uses Own_house as the root node. Their possible values (or outcomes) are the branches. At each branch, we listed the number of training examples of each class (No or Yes) that land or reach there. Fig. 5(B) is obviously a better choice for the root. From a prediction or classification point of view, Fig. 5(B) makes fewer mistakes than Fig. 5(A). In Fig. 5(B), when Own_house=true every example has the class Yes. When Own_house=false, if we take majority class (the most frequent class), which is No, we make 3 mistakes/errors. If we look at Fig. 5(A), the situation is worse. If we take the majority class for each branch, we make 5 mistakes (marked in bold). Thus, we say that the impurity of the tree in Fig. 5(A) is higher than the tree in Fig. 5(B). To learn a decision tree, we prefer Own_house to Age to be the root node. Instead of counting the number of mistakes or errors, C4.5 uses a more principled approach to perform this evaluation on every attribute in order to choose the best attribute to build the tree. ■

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

The information gain measure is based on the **entropy** function from **information theory** [433]:

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

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

where $\Pr(c_j)$ is the probability of class c_j in data set D, which is the number of examples of class c_j in D divided by the total number of examples in D. In the entropy computation, we define $0log0 = 0$. The unit of entropy is **bit**. Let us use an example to get a feeling of what this function does.

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

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

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

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

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

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

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

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

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

Information gain

The idea is the following:

1. Given a data set D, we first use the entropy function (Equation (2)) to compute the impurity value of D, which is *entropy*(D). The **impurityEval-1** function in line 7 of Fig. 4 performs this task.
2. Then, we want to know which attribute can improve the impurity most if it is used to partition D. To find out, every attribute is evaluated (lines 8-10 in Fig. 4). Let the number of possible values of the attribute A_i be v. If we are going to use A_i to partition the data D, we will divide D into v disjoint subsets $D_1, D_2, \ldots, D_v$. The entropy after the partition is

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

   The **impurityEval-2** function in line 9 of Fig. 4 performs this task.
3. The information gain of attribute A_i is computed with:

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

Clearly, the gain criterion measures the reduction in impurity or disorder. The *gain* measure is used in line 11 of Fig. 4, which chooses the attribute A_g resulting in the largest reduction in impurity. If the gain of A_g is too small, the algorithm stops for the branch (line 12). Normally a threshold is used here. If choosing A_g is able to reduce impurity significantly, A_g is employed to partition the data to extend the tree further, and so on (lines 15-21 in Fig. 4). The process goes on recursively by building sub-trees using $D_1, D_2, \ldots, D_m$ (line 20). For subsequent tree extensions, we do not need A_g any more, as all training examples in each branch has the same A_g value.

Example 7: Let us compute the gain values for attributes Age, Own_house and Credit_Rating using the whole data set D in Table 1, i.e., we evaluate for the root node of a decision tree.

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

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

We then try Age, which partitions the data into 3 subsets (as Age has three possible values) D_1 (with Age=young), D_2 (with Age=middle), and D_3 (with Age=old). Each subset has 5 training examples. In Fig. 5, we also see

the number of No class examples and the number of Yes examples in each subset (or in each branch).

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

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

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

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

$gain(D,$ Age$) = 0.971 - 0.888 = 0.083$
$gain(D,$ Own_house$) = 0.971 - 0.551 = 0.420$
$gain(D,$ Has_Job$) = 0.971 - 0.647 = 0.324$
$gain(D,$ Credit_Rating$) = 0.971 - 0.608 = 0.363$

Own_house is the best attribute for the root node. Fig. 5 shows the root node using Own_house. Since the left branch has only one class (Yes) of data, it results in a leaf node (line 1 in Fig. 4). For Own_house = false, further extension is needed. The process is the same as above, but we only use the subset of the data with Own_house = false, i.e., D_2. ■

Information gain ratio

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

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

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

where s is the number of possible values of A_i, and D_j is the subset of data that has the jth value of A_i. $|D_j|/|D|$ simply corresponds to the probability in Equation (2). Using Equation (5), we simply choose the attribute with the highest gainRatio value to extend the tree.

This method works because if A_i has too many values the denominator will be large. For instance, in our above example of the *ID* attribute, the denominator will be $\log_2|D|$. The denominator is called the **split info** in C4.5. One note is that gainRatio can be 0 or very small. Some heuristic solutions can be devised to deal with them [404].

3.2.3 Handling of Continuous Attributes

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

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

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

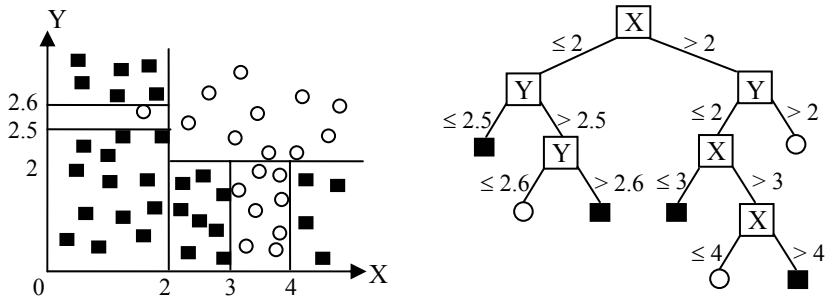
The threshold value can be the middle point between v_i and v_{i+1}, or just on the "right side" of value v_i, which results in two intervals $A_i \le v_i$ and $A_i > v_i$. This latter approach is used in C4.5. The advantage of this approach is

that the values appearing in the tree actually occur in the data. The threshold value that maximizes the gain (gainRatio) value is selected. We can modify the algorithm in Fig. 4 (lines 8-11) easily to accommodate this computation so that both discrete and continuous attributes are considered.

A change to line 20 of the algorithm in Fig. 4 is also needed. For a continuous attribute, we do not remove attribute A_g because an interval can be further split recursively in subsequent tree extensions. Thus, the same continuous attribute may appear multiple times in a tree path (see Example 9), which does not happen for a discrete attribute.

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

Example 8: The hyper-rectangular regions in Fig. 6(A), which partitions the data space, are produced by the decision tree in Fig. 6(B). There are two classes in the data, represented by empty circles and filled rectangles.



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

Fig. 6. A partitioning of the data space and its corresponding decision tree ■

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

3.2.4 Some Other Issues

We now discuss several other issues in decision tree learning.

Tree pruning and overfitting: A decision tree algorithm recursively parti-

tions the data until there is no impurity or there is no attribute left. This process may result in trees that are very deep and many tree leaves may cover very few training examples. If we use such a tree to predict the training set, the accuracy will be very high. However, when it is used to classify unseen test set, the accuracy may be very low. The learning is thus not effective, i.e., the decision tree does not **generalize** the data well. This phenomenon is called **overfitting**. More specifically, we say that a classifier f_1 **overfits** the data if there is another classifier f_2 such that f_1 achieves a higher accuracy on the training data than f_2, but a lower accuracy on the unseen test data than f_2 [344].

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

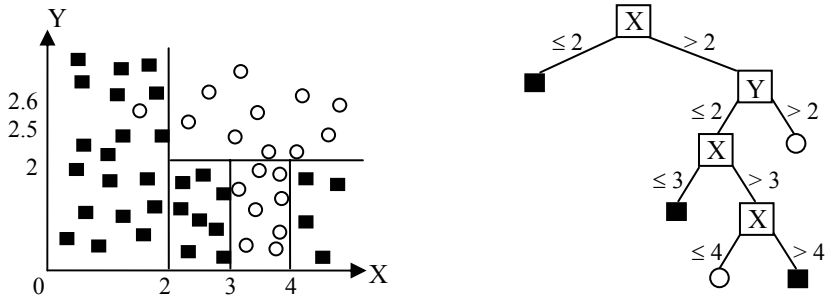
To reduce overfitting in the context of decision tree learning, we perform pruning of the tree, i.e., to delete some branches or sub-trees and replace them with leaves of majority classes. There are two main methods to do this, **stopping early** (which is also called **pre-pruning**) in tree building and **pruning** the tree after it is built (which is called **post-pruning**). Both approaches have been experimented by researchers. Post-pruning has been shown more effective. Early-stopping can be dangerous because it is not clear what will happen if the tree is extended further (without stopping). Post-pruning is more effective because after we have extended the tree to the fullest, it becomes clearer what branches/sub-trees may not be useful (may overfit the data). The general idea of post-pruning is to estimate the error of each tree node. If the estimated error for a node is less than the estimated error of its extended sub-tree, then the sub-tree is pruned. Most existing tree learning algorithms takes this approach. See [406] for a technique called the pessimistic error based pruning.

Example 9: In Fig. 6(B), the sub-tree represents the rectangular region

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

in Fig.6(A) is very likely to be overfitting. The region is very small and contains only a single data point, which may be an error (or noise) in the data collection. If it is pruned, we obtain Fig.7(A) and (B). ■

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



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

Fig. 7. The data space partition and the decision tree after pruning

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

Example 10: The sub-tree below $X \le 2$ in Fig. 6(B) produces these rules:

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

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

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

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

$X \le 2 \rightarrow$ ■

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

A useful point to note is that after pruning the resulting set of rules may no longer be **mutually exclusive** and **exhaustive**. There may be data points that satisfy the conditions of more than one rule, and if inaccurate rules are discarded, of no rules. A sorting of the rules is needed to ensure

that when classifying a test case only one rule (the first one) will be applied to determine the class of the test case. To deal with the situation that a test case does not satisfy the conditions of any rule, a **default class** is used, which is the majority class.

Handling missing attribute values: In many practical data sets, some attribute values are missing or not available due to various reasons. There are many ways to deal with the problem. For example, we can fill each missing value with the special value "unknown" or the most frequent value of the attribute if the attribute is discrete. If the attribute is continuous, use the mean of the attribute for each missing value.

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

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

3.3 Classifier Evaluation

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

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

sent several common methods for classifier evaluation, and then introduce some other evaluation measures.

3.3.1 Evaluation Methods

Holdout set: The available data set D is divided into two disjoint subsets, the *training set* D_{train} and the *test set* D_{test}, $D = D_{train} \cup D_{test}$ and $D_{train} \cap D_{test} = \varnothing$. The test set is also called the holdout set. Note that the examples in the original data set D are all labeled with classes. This method is mainly used when the data set D is large.

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

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

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

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

Cross-validation: When the data set is small, the ***n*-fold cross-validation** method is also (more) commonly used. In this method, the available data is partitioned into n equal-size disjoint subsets. We then use each subset as the test set and combine the rest n-1 subsets as the training set to learn a classifier. This procedure is then run n times, which give n accuracies. The

final estimated accuracy of learning from this data set is the average of the *n* accuracies. 10-fold and 5-fold cross-validations are often used.

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

In Section 3.2.4, we mentioned that a validation set can be used to prune a decision tree or a set of rules. If **a validation set** is employed for that purpose, it should not be used in testing. In that case, the available data is divided into three subsets, a training set, a validation set and a test set. Apart from using a validation set to help tree or rule pruning, a validation set is also used frequently to estimate parameters in learning algorithms. In such cases, the values that give the best accuracy on the validation set are used as the final values of the parameters. Cross-validation can be used for parameter estimating as well. Then, there is no need to have a separate validation set. Instead, the whole training set is used in cross validation.

3.3.2 Precision, Recall, F-score and Breakeven Point

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

Precision and **recall** are more suitable in such applications because they measure how precise and how complete the classification is on the positive class. It is convenient to introduce these measures using a **confusion matrix** (Table 2). A confusion matrix contains information about actual and predicted results given by a classifier.

Table 2. Confusion Matrix of a classifier

	Classified Positive	Classified Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

where

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

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

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

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

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

Example 11: A test data set has 100 positive examples and 1000 negative examples. After classification using a classifier, we have the following confusion matrix (Table 3),

Table 3. Confusion Matrix of a classifier

	Classified Positive	Classified Negative
Actual Positive	1	99
Actual Negative	0	1000

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

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

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

F-score (also called $\mathbf{F_1}$**-score**) is the harmonic mean of precision and recall.

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

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

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

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

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

Assume that the test set has 10 positive examples.

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

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

3.4 Rule Induction

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

3.4.1 Sequential Covering

Most rule induction systems use an algorithm called **sequential covering**. A classifier built with this algorithm consists of a list of rules, which is also called a **decision list** [414]. In the list, the ordering of the rules is significant.

The basic idea of sequential covering is to learn a list of rules sequentially, one at a time, to cover the training data. After each rule is learned, the training examples covered by the rule are removed. Only the remaining data are used to find subsequent rules. Recall that a rule covers an example if the example satisfies the conditions of the rule. We study two specific algorithms based on this general strategy. The first algorithm is based on the CN2 system [92], and the second algorithm is based on the ideas in FOIL [405], I-REP [171], REP [65], and RIPPER [94] systems.

Algorithm 1 (ordered rules)

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

This algorithm is given in Fig. 8. D is the training data. *RuleList* is the list of rules, which is initialized to empty set (line 1). *Rule* is the best rule found in each iteration. The function learn-one-rule-1() learns the *Rule* (lines 2 and 6). The stopping criteria for the while-loop can be of various kinds. Here we use $D = \varnothing$ or *Rule* is NULL (a rule is not learned). Once a rule is learned from the data, it is inserted into *RuleList* at the end (line 4). All the training examples that are covered by the rule are removed from the data (line 5). The remaining data is used to find the next rule and so on. After rule learning ends, a **default class** is inserted at the end of *RuleList*. This is because there may still be some training examples that are not covered by any rule as no good rule can be found from them, or because some test cases may not be covered by any rule and thus cannot be classified. The final list of rules is as follows:

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

where r_i is a rule.

Algorithm 2 (ordered classes)

This algorithm learns all rules for each class together. After rule learning for one class is completed, it moves to the next class. Thus all rules for each class appear together in the rule list. The sequence of the rules for

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

Fig. 8. The first rule learning algorithm based on sequential covering

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

Fig. 9. The second rule learning algorithm based on sequential covering

each class is unimportant, but the rule subsets for different classes are ordered. Typically, the algorithm finds rules for the least frequent class first, then the second least frequent class and so on. This ensures that some rules are learned for rare classes. Otherwise, they may be dominated by frequent classes and end up with no rules if considered after frequent classes.

The algorithm is given in Fig. 9. The data set D is split into two subsets, *Pos* and *Neg*, where *Pos* contains all the examples of class c from D, and *Neg* the rest of the examples in D (line 3). Two stopping conditions for rule learning of each class are in line 4 and line 6. The other parts of the algorithm are quite similar to those of the first algorithm in Fig. 8. Both learn-one-rule-1() and learn-one-rule-2() functions are described in Section 3.4.2.

Use of rules for classification

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

3.4.2 Rule learning: Learn-one-rule Function

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

This is a heuristic and greedy algorithm in that after a condition is added, it will not be changed or removed through backtracking. Ideally, we would try all possible combinations of attribute-value pairs. However, this is not practical as the number of possibilities grows exponentially. Hence, in practice, the above greedy algorithm is used. However, instead of keeping only the best set of conditions, we can improve the procedure a little by keeping k best sets of conditions ($k > 1$) in each iteration. This is called the **beam search** (k beams), which ensures that a larger space is explored.

We present two specific implementations of the algorithm, namely learn-one-rule-1() and learn-one-rule-2(). learn-one-rule-1() is used in the sequential-covering-1 algorithm, and learn-one-rule-2() is used the sequential-covering-2 algorithm.

Learn-one-rule-1

This function uses beam search (Fig. 10). The number of beams is k. *BestCond* stores the conditions of the rule to be returned. The class is omitted as it is the majority class of the data covered by *BestCond*. *candidateCondSet* stores the current best condition sets (which are the frontier beams) and its size is less than or equal to k. Each condition set contains a set of conditions connected by "and" (conjunction). *newCandidateCondSet* stores all the new candidate condition sets after adding each attribute-value

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

Fig. 10. The learn-one-rule-1() function

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

Fig. 11. The entropy based evaluation function

pair (a possible condition) to every candidate in *candidateCondSet* (lines 5-11). Lines 13-17 update the *BestCond*. Specifically, an evaluation function is used to assess whether each new candidate condition set is better than the existing best condition set *BestCond* (line 14). If so, it replaces the current *BestCond* (line 15). Line 18 updates *candidateCondSet*, which selects *k* new best condition sets (new beams).

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

Fig. 12. The learn-one-rule-2() function

Once the final *BestCond* is found, it is evaluated to see if it is significantly better than without any condition (∅) using a *threshold* (line 20). If yes, a rule will be formed using *BestCond* and the most frequent (or the majority) class of the data covered by *BestCond* (line 21). If not, NULL is returned to indicate that no significant rule is found.

The evaluation() function (Fig. 11) uses the entropy function as in the decision tree learning. Other evaluation functions are possible too. Note that when *BestCond* = ∅, it covers every example in *D*, i.e., *D* = *D'*.

Learn-one-rule-2

In the learn-one-rule-2() function (Fig. 12), a rule is first generated and then it is pruned. This method first splits the positive and negative training data *Pos* and *Neg*, into a growing and pruning sets respectively. The growing sets, *GrowPos* and *GrowNeg*, are used to generate a rule, called *BestRule*. The pruning sets, *PrunePos* and *PruneNeg* are used to prune the rule because *BestRule* may overfit the data. Note that *PrunePos* and *PruneNeg* are actually validation sets discussed in Sections 3.2.4 and 3.3.1.

growRule() function: growRule() generates a rule (called *BestRule*) by repeatedly adding a condition to its condition set that maximizes an evaluation function until the rule covers only some positive examples in *GrowPos* but no negative examples from *GrowNeg*. This is basically the same as lines 4-17 in Fig.10, but without beam search (i.e., only the best rule is kept in each iteration). Let the current partially developed rule be *R*:

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

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

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

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

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

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

where p (respectively n) is the number of examples in *PrunePos* (*PruneNeg*) covered by the current rule (after a deletion).

3.4.3 Discussion

Separate-and-conquer vs. divide-and-conquer: Decision tree learning is said to use the *divide-and-conquer* strategy. At each step, all attributes are evaluated and one is selected to partition/divide the data into m disjoint subsets, where m is the number of values of the attribute. Rule induction discussed in this section is said to use the *separate-and-conquer* strategy, which evaluates all attribute-value pairs (conditions) (which are much larger in number than the number of attributes) and selects only one. Thus, each step of divide-and-conquer expands m rules, while each step of separate-and-conquer expands only one rule. Due to both effects, the separate-and-conquer strategy is much slower than the divide-and-conquer strategy.

Rule understandability: If-then rules are easy to understand by human users. However, a word of caution about rules generated by sequential covering is in order. Such rules can be misleading because the covered data are removed after each rule is generated. Thus the rules in the rule list are not independent of each other. A rule r may be of high quality in the context of the data D' from which r was generated. However, it may be a weak rule with a very low accuracy (confidence) in the context of the whole data set D ($D' \subseteq D$) because many training examples that can be covered by r have already been removed by rules generated before r. If you want to understand the rules and possibly use them in some real-world tasks, you should be aware of this fact.

3.5 Classification Based on Associations

In Section 3.2, we show that a decision tree can be converted to a set of rules, and in Section 3.4, we see that a set of rules may also be found directly for classification. It is thus only natural to expect that association rules, in particular **class association rules** (**CAR**), may be used for classification as well. Recall that a CAR is an association rule with a single class value/label on the right-hand-side of the rule as its consequent. For instance, from the data in Table 1, the following rule can be found:

Own_house = false, Has_job = true → Class = Yes [sup=5/15, conf=5/5]

which was also a rule from the decision tree in Fig. 3. In fact, syntactically there is no difference between rules from a decision tree (or a rule induction system) and CARs if we consider only categorical (or discrete) attributes (more on this later). The differences are in the mining processes, objectives and the final rule sets.

Class association rule mining finds all rules in data that satisfy some user-specified minimum support (minsup) and the minimum confidence (minconf) constraints. A decision tree or a rule induction system finds only a **subset** of the rules (expressed as a tree or a list of rules) for classification

Example 11: Recall the decision tree in Fig. 3 gives the following rules:

Own_house = true → Class =Yes [sup=6/15, conf=6/6]
Own_house = false, Has_job = true → Class=Yes [sup=5/15, conf=5/5]
Own_house = false, Has_job = false → Class=No [sup=4/15, conf=4/4]

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

Age = young, Has_job = true → Class=Yes [sup=2/15, conf=2/2]
Age = young, Has_job = false → Class=No [sup=3/15, conf=3/3]
Credit_Rating = fair → Class=No [sup=4/15, conf=4/4]
Credit_Rating = good → Class=Yes [sup=5/15, conf=5/6]

and many more, if we use minsup = 2/15 = 13.3% and minconf = 80%. ■

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

The complete set of rules from CAR mining is also beneficial from a rule usage point of view. In some applications, the user wants to act on some interesting rules. For example, in an application of finding causes of product problems, more rules are preferred to fewer rules because with

more rules the user is more likely to find rules that indicate causes of the problems. Such rules may not be generated by a decision tree or a rule induction system. A deployed data mining system based on CARs is reported in [313]. However, we should also bear in mind of the following:

1. Decision tree learning and rule induction do not use minimum support (minsup) or minimum confidence (minconf) constraints. Thus, some rules that they find can have very low supports, which of course are likely to be pruned in tree pruning or rule pruning because the chance that they overfit the training data is high. Although we can set a low minsup for CAR mining, it may cause combinatorial explosion and generates too many rules (the mining process may not be able to complete). In practice, a limit on the total number of rules to be generated may be used to control the CAR generation process. When the number of generated rules reaches the limit, the algorithm stops. However, with this limit, we may not be able to generate long rules (rules with many conditions). In some applications, this might not be a problem. Recall that the Apriori algorithm works in a level-wise fashion, i.e., short rules are generated before long rules. For practical applications, short rules are preferred and are often sufficient for classification and for action. Long rules normally have very low supports and tend to overfit the data.
2. CAR mining does not use continuous (numeric) attributes, while decision trees deal with continuous attributes naturally. Rule induction can use continuous attributes as well. There is still no satisfactory method to deal with such attributes directly in association rule mining. Fortunately, many attribute discretization algorithms exist that can automatically discretize the value range of a continuous attribute into suitable intervals [e.g., 138, 156], which are then considered as discrete values.

There are several methods that use CARs to build classifiers. We study one of them here, which is a simplified version of the method in the CBA system [305]. CBA builds a classifier in two steps. It first mines all CARs and then identifies a subset of rules to build a classifier.

3.5.1 Mining Class Association Rules for Classification

Mining of CARs has been discussed in Chapter 2, so we will not repeat the algorithm here. Here we only highlight some issues on rule mining that affect the constructed classifiers.

Multiple minimum class supports: As discussed in Chapter 2, the most important parameter in association rule mining is the minimum support (or minsup for short), which greatly impacts on the number rules generated

and the kinds of rules generated. Similar to normal association rule mining, using a single minsup is inadequate for mining of CARs because many practical classification data sets have uneven class distributions, i.e., some classes cover a large proportion of data, while others cover only a small proportion of data (which are called **rare** or **infrequent classes**). As we discussed in Chapter 2, using a single minsup may not be adequate.

Example 12: Suppose we have a dataset with 2 classes, *Y* and *N*. 99% of the data belong to the *Y* class, and only 1% of the data belong to the *N* class. If we set minsup = 1.5%, we will not find any rule for class *N*. To solve the problem, we need to lower down the minsup. Suppose we set minsup = 0.2%. Then, we may find a huge number of overfitting rules for class *Y* because minsup = 0.2% is too low for class *Y*. ■

Multiple class minimum class supports can be applied to deal with the problem. We can assign a minimum class support $minsup_i$ for each class c_i.

***minsup*$_i$**: For each class c_i, we set a different **minimum class support** $minsup_i$ according to the frequency of the class and other application specific information. All the final rules for class c_i must satisfy $minsup_i$. Alternatively, we can provide one single total minsup, denoted by *t_minsup*, which is then distributed to each class according to the class distribution:

$$minsup_i = t\_minsup \times freq(c_i) \quad (12)$$

where $freq(c_i)$ is the proportion of training examples with class c_i. The formula gives frequent classes higher minsups and infrequent classes lower minsups. This ensures that we will generate sufficient rules for infrequent classes without producing many overfitting rules for frequent classes.

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

Data formats: The algorithm for CAR mining given in Chapter 2 is for mining a transaction data set. However, most classification data sets are in the table format. As we discussed in Chapter 2, a tabular data set can be easily converted to a transaction data set.

Finally, rule pruning may also be performed to remove overfitting rules before classifier building. CBA uses a similar method to that in C4.5. Other methods can be used as well, e.g., [292] uses chi-square test.

3.5.2 Classifier Building

After all rules are found, a classifier is built using the rules. Clearly, there are many possible methods to build/learn a classifier from CARs. For instance, a simple and lazy approach is "do-nothing". That is, we simply use CARs directly for classification. For each test instance, we find the most confident rule (the rule with the highest confidence) that covers the instance. Recall that a rule covers an instance if the instance satisfies the conditions of the rule. The class of the rule is assigned as the class of the test instance. This simple method performs in fact quite well. However, on average CBA performs better. Let us study the method in CBA, which is similar to the sequential covering method, but applied to class association rules with additional enhancements as discussed in the last sub-section.

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

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

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

A CBA classifier L is of the form:

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

where $r_i \in S$, $r_a \succ r_b$ if $b > a$. In classifying an unseen or test case, the first rule that satisfies the case classifies it. If no rule applies to the case, it takes the default class (*default_class*). A simplified version of the algorithm for building such a classifier is given in Fig. 13. The classifier is the *RuleList*.

This algorithm can be easily implemented by making one pass through the training data for every rule. However, this is extremely inefficient for large data sets. An efficient algorithm which makes at most two passes over the data is given in [305].

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

Fig. 13. A simple classifier building algorithm

3.6 Naïve Bayesian Classification

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

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

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

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

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

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

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

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

$\Pr(C=c_j)$ is the class **prior** probability in D, which can be easily estimated

from the training data. It is simply the percentage of data in D with class c_i.

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

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

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

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

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

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

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

$$\begin{aligned}&\Pr(C = c_j \mid A_1 = a_1,...,A_{|A|} = a_{|A|})\\ &= \frac{\Pr(C = c_j)\prod_{i=1}^{|A|} \Pr(A_i = a_i \mid C = c_j)}{\sum_{r=1}^{|C|} \Pr(C = c_r)\prod_{i=1}^{|A|} \Pr(A_i = a_i \mid C = c_r)}\end{aligned} \tag{18}$$

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

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

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

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

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

Example 13: Suppose that we have the training data set in Fig.14, which has two attributes *A* and *B*, and the class *C*. We can compute all the probability values required to learn a naïve Bayesian classifier.

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

Fig. 14. An example training data set.

Pr(C = t) = 1/2, Pr(C= f) = 1/2

Pr(A=m | C=t) = 2/5	Pr(A=g | C=t) = 2/5	Pr(A=h | C=t) = 1/5
Pr(A=m | C=f) = 1/5	Pr(A=g | C=f) = 2/5	Pr(A=h | C=n) =2/5
Pr(B=b | C=t) = 1/5	Pr(B=s | C=t) = 2/5	Pr(B=q | C=t) = 2/5
Pr(B=b | C=f) = 2/5	Pr(B=s | C=f) = 1/5	Pr(B=q | C=f) = 2/5

Now we have a test example:

A = m B = q C = ?

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

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

For class C = f, we have

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

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

It is easy to see that the probabilities (i.e., $\Pr(C=c_j)$ and $\Pr(A_i=a_i \mid C=c_j)$) required to build a naïve Bayesian classifier can be found in one scan of

the data. Thus, the algorithm is linear in the number of training examples, which is one of the great strengths of the naïve Bayes, i.e., it is extremely efficient. In terms of classification accuracy, although the algorithm makes the strong assumption of conditional independence, several researchers have shown that its classification accuracies are surprisingly strong. See experimental comparisons of various techniques in [135, 255, 310].

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

Numeric attributes: The above formulation of the naïve Bayesian learning assumes that all attributes are categorical. However, most real life data sets have numeric attributes. Therefore, in order to use the naïve Bayeisan algorithm, each numeric attribute needs to be discretized into intervals. This is the same as for class association rule mining. Existing discretization algorithms in [e.g., 138, 156] can be used.

Zero counts: It is possible that a particular attribute value never occurs together with a class in the training set. This is problematic because it will result in a 0 probability, which wipes out all the other probabilities $\Pr(A_i=a_i \mid C=c_j)$ when they are multiplied according to Equation (21) or Equation (18). A principled solution to this problem is to incorporate a small-sample correction into all probabilities.

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

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

where m_i is the number of values of attribute A_i (e.g., 2 for a Boolean attribute), and λ is a multiplicative factor, which is commonly set to $\lambda = 1/n$, where n is the number of examples in the training set D [135, 255]. When $\lambda = 1$, we get the well known **Laplace's law of succession** [185]. The general form of Equation (22) is called the **Lidstone's law of succession** [294]. Applying the correction $\lambda = 1/n$, the probabilities of Example 13 are revised. For example,

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

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

3.7 Naïve Bayesian Text Classification

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

Due to the rapid growth of online documents in organizations and on the Web, automated document classification becomes an important problem. Although the techniques discussed in the previous sections can be applied to text classification, it has been shown that they are not as effective as the methods presented in this section and in the next two sections. In this section, we study a naïve Bayesian learning method that is specifically formulated for texts, which makes use of some text specific features. However, the ideas are similar to those in Section 3.6. We first present a probabilistic framework of texts, and then study the naïve Bayesian equations for their classification. There are some slight variations of the model. This section is mainly based on the technique given in [326].

3.7.1 Probabilistic Framework

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

The generative model is based on two assumptions:

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

A **mixture model** models the data with a number of statistical distributions. Intuitively, each distribution corresponds to a data cluster and the parameters of the distribution provide a description of the corresponding cluster. Each distribution in a mixture model is also called a **mixture component** (the distribution can be of any kind). Fig. 15 shows a plot of

the **probability density function** of a 1-dimensional data set (with two classes) generated by a mixture of two Gaussian distributions, one per class, whose parameters (denoted by θ_i) are the mean (μ_i) and the standard deviation (σ_i), i.e., $\theta_i = (\mu_i, \sigma_i)$.

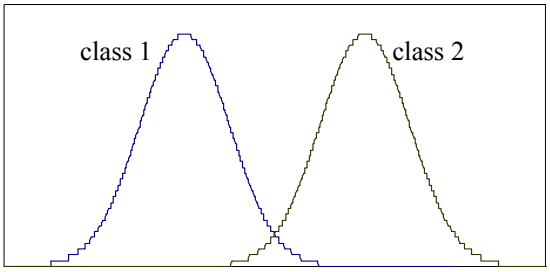


Fig. 15. A mixture of two Gaussian distributions that generates the data set

Let the number of mixture components (or distributions) in a mixture model be K, and the jth distribution have the parameters θ_j. Let $\mathbf{\Theta}$ be the set of parameters of all components, $\mathbf{\Theta} = \{\varphi_1, \varphi_2, \ldots, \varphi_K, \theta_1, \theta_2, \ldots, \theta_K\}$, where φ_j is the **mixture weight** (or **mixture probability**) of the mixture component j and θ_j is the parameters of component j. The mixture weights are subject to the constraint $\sum_{j=1}^{K} \varphi_j = 1$. The meaning of mixture weights (or probabilities) will be clear below.

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

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

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

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

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

3.7.2 Naïve Bayesian Model

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

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

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

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

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

Now assume n independent trials are conducted, each with the k possible outcomes and with the same probabilities, $p_1, p_2, \ldots, p_k$. Let us number

the outcomes 1, 2, 3, .., k. For each outcome, let X_t denote the number of trials that result in that outcome. Then, X_1, X_2, …, X_k are discrete random variables. The collection of X_1, X_2, …, X_k is said to have the **multinomial distribution** with parameters, n, p_1, p_2, …, p_k.

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

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

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

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

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

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

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

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

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

times that word w_t occurs in document d_i.

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

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

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

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

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

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

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

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

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

3.7.3 Discussions

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

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

The naïve Bayesian formulation presented here is based on a mixture of **multinomial distributions**. There is also a formulation based on **multivariate Bernoulli distributions** in which each word in the vocabulary is a binary feature, i.e., it either appears or does not appear in the document. Thus, it does not consider the number of times that a word occurs in a document. Experimental comparisons show that multinomial formulation consistently produces more accurate classifiers [326].

3.8 Support Vector Machines

Support vector machines (SVM) is another type of learning system [466], which has many desirable qualities that make it one of most popular algorithms. It not only has a solid theoretical foundation, but also performs classification more accurately than most other systems in many applications, especially those applications involving very high dimensional data. For instance, it has been shown by several researchers that SVM is perhaps the most accurate algorithm for text classification. It is also widely used in Web page classification and bioinformatics applications.

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

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

where $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{ir})$ is a r-dimensional **input vector** in a real-valued space $X \subseteq \Re^r$, y_i is its **class label** (output value) and $y_i \in \{1, -1\}$. 1 denotes the positive class and -1 denotes the negative class. Note that we use

slightly different notations in this section. For instance, we use y instead of c to represent a class because y is commonly used to represent classes in the SVM literature. Similarly, each data instance is called an **input vector** and denoted a bold face letter. In the following, we use bold face letters for all vectors.

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

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

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

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

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

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

where x_i is the variable representing the ith coordinate of the vector $\mathbf{x}$. For convenience, we will use the vector notation from now on.

In essence, SVM finds a hyperplane

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

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

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

Fig. 16(A) shows an example in a 2-dimensional space. Positive instances (also called positive data points or simply positive points) are represented with small filled rectangles, and negative examples are represented with small empty circles. The thick line in the middle is the decision boundary hyperplane (a line in this case), which separates positive (above the line) and negative (below the line) data points. Equation (31), which is also called the **decision rule** of the SVM classifier, is used to make classification decisions on test instances.

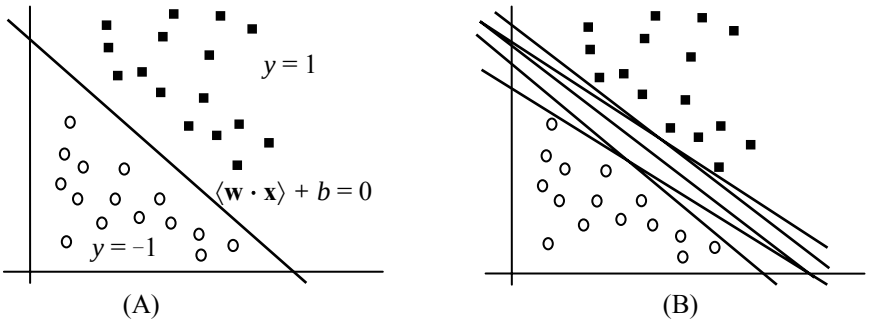


Fig. 16. (A) A linearly separable data set and (B) possible decision boundaries

Fig. 16(A) raises two interesting questions:

1. There are an infinite number of lines that can separate the positive and negative data points as illustrated by Fig. 16(B). Which line should we choose?
2. A hyperplane classifier is only applicable if the positive and negative data can be linearly separated. How can we deal with nonlinear separations or data sets that require nonlinear decision boundaries?

The SVM framework provides good answers to both questions. Briefly, for question 1, SVM chooses the hyperplane that maximizes the margin (the gap) between positive and negative data points, which will be defined formally below. For question 2, SVM uses **kernel** functions. Before we dive into the details, we should note that SVM requires numeric data and only builds two-class classifiers. At the end of the section, we will discuss how these limitations may be addressed.

3.8.1 Linear SVM: Separable Case

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

From the linear algebra, we know that in $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$, $\mathbf{w}$ defines a direction perpendicular to the hyperplane (see Fig. 17). $\mathbf{w}$ is also called the **normal vector** (or simply **normal**) of the hyperplane. Without changing the normal vector $\mathbf{w}$, varying b moves the hyperplane parallel to itself. Note also that $\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = 0$ has an inherent degree of freedom. We can rescale the hyperplane to $\langle \lambda\mathbf{w} \cdot \mathbf{x} \rangle + \lambda b = 0$ for $\lambda \in \Re^+$ (positive real numbers) without changing the function/hyperplane.

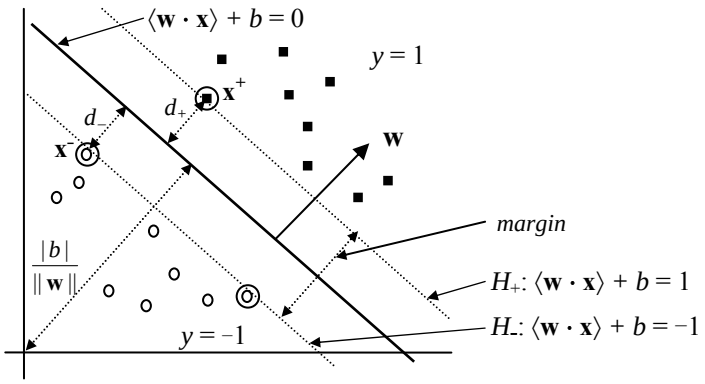


Fig. 17. Separating hyperplanes and margin of SVM: Support vectors are circled

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Learning is to solve the following constrained minimization problem,

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

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

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

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

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

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

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

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

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

The optimization theory says that an optimal solution to (41) must satisfy certain conditions, called **Kuhn-Tucker conditions**. Kuhn-Tucker conditions play a central role in constrained optimization. Here, we give a brief introduction to these conditions. Let the general optimization problem be

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

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

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

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

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

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

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

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

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

If $\alpha_i > 0$ then $g_i(\mathbf{x}) = b_i$.
If $g_i(\mathbf{x}) > b_i$ then $\alpha_i = 0$.

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

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

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

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

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

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

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

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

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

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

The concept of duality is widely used in the optimization literature. The aim is to provide an alternative formulation of the problem which is more convenient to solve computationally and/or has some theoretical significance. In the context of SVM, the dual problem is not only easy to solve computationally, but also crucial for using kernel functions to deal with nonlinear decision boundaries.

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

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

and (49), which is

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

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

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

where $\sum_{i=1}^{r} y_i \alpha_i = 0$. L_D contains only **dual variables** and must be maximized under simpler constraints, (48) and (49), and $\alpha_i \geq 0$. Note that (48) is not needed as it has already been substituted into the objective function L_D. Hence, the **dual** of the primal Equation (40) is

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

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

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

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

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

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

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

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

3.8.2 Linear SVM: Non-separable Case

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

For SVM to be practically useful, it must allow errors or noise in the training data. However, with noisy data the linear separable SVM will not find a solution because the constraints cannot be satisfied. For example, in Fig. 18, there is a negative point (which is circled) in the positive region, and a positive point in the negative region. Clearly, no feasible solution can be found for this problem.

Recall that the primal for the linear separable case was:

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

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

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

Thus we have the new constraints:

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

The geometric interpretation is shown in Fig. 18, which has two error data points $\mathbf{x}_a$ and $\mathbf{x}_b$ (circled) in wrong regions.

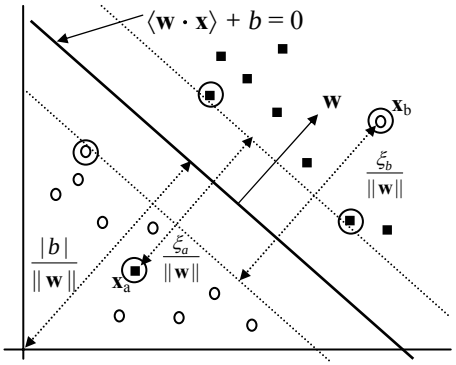


Fig. 18. The non-separable case: $\mathbf{x}_a$ and $\mathbf{x}_b$ are error data points.

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

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

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

The new optimization problem becomes:

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

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

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

where α_i, $\mu_i \geq 0$ are the **Lagrange multipliers**.

Kuhn-Tucker conditions for optimality are the following:

$$\frac{\partial L_P}{\partial w_j} = w_j - \sum_{i=1}^{r} y_i \alpha_i x_{ij} = 0, \ \ j = 1, 2, ..., r \tag{63}$$
$$\frac{\partial L_P}{\partial b} = -\sum_{i=1}^{r} y_i \alpha_i = 0 \tag{64}$$
$$\frac{\partial L_P}{\partial \xi_i} = C - \alpha_i - \mu_i = 0, \ \ i = 1, 2, ..., n \tag{65}$$
$$y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) - 1 + \xi_i \geq 0, \ \ i = 1, 2, ..., n \tag{66}$$
$$\xi_i \geq 0, \ \ i = 1, 2, ..., n \tag{67}$$
$$\alpha_i \geq 0, \ \ i = 1, 2, ..., n \tag{68}$$
$$\mu_i \geq 0, \ \ i = 1, 2, ..., n \tag{69}$$
$$\alpha_i(y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) - 1 + \xi_i) = 0, \ \ i = 1, 2, ..., n \tag{70}$$
$$\mu_i \xi_i = 0, \ \ i = 1, 2, ..., n \tag{71}$$

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

$$\text{Maximize:} \ L_D(\boldsymbol{\alpha}) = \sum_{i=1}^{r} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{r} y_i y_j \alpha_i \alpha_j \langle \mathbf{x}_i \cdot \mathbf{x}_j \rangle. \tag{72}$$

$$\text{Subject to: } \sum_{i=1}^{r} y_i \alpha_i = 0$$
$$0 \le \alpha_i \le C, \quad i = 1, 2, ..., n.$$

Interestingly, ξ_i and its Lagrange multipliers μ_i are not in the dual and the objective function is identical to that for the separable case. The only difference is the constraint $\alpha_i \le C$ (inferred from $C-\alpha_i-\mu_i = 0$ and $\mu_i \ge 0$).

The dual problem (72) can also be solved numerically, and the resulting α_i values are then used to compute **w** and b. **w** is computed using Equation (63) and b is computed using the Kuhn-Tucker complementarity conditions (70) and (71). Since we do not have values for ξ_i, we need to get around it. From Equations (65), (70) and (71), we observe that if $0 < \alpha_i < C$ then both $\xi_i = 0$ and $y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) - 1 + \xi_i) = 0$. Thus, we can use any training data point for which $0 < \alpha_i < C$ and Equation (69) (with $\xi_i = 0$) to compute b.

$$b = \frac{1}{y_i} - \sum_{i=1}^{r} y_i \alpha_i \langle \mathbf{x}_i \cdot \mathbf{x}_j \rangle = 0. \tag{73}$$

Again, due to numerical errors, we can compute all possible b's and then take their average as the final b value.

Note that Equations (65), (70) and (71) in fact tell us more:

$$\begin{array}{lll} \alpha_i = 0 & \Rightarrow & y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) \ge 1 \text{ and } \xi_i = 0 \\ 0 < \alpha_i < C & \Rightarrow & y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) = 1 \text{ and } \xi_i = 0 \\ \alpha_i = C & \Rightarrow & y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) \le 1 \text{ and } \xi_i \ge 0 \end{array} \tag{74}$$

Similar to support vectors for the separable case, (74) shows one of the most important properties of SVM: the solution is sparse in α_i. Most training data points are outside the margin area and their α_i's in the solution are 0. Only those data points that are on the margin (i.e., $y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) = 1$, which are support vectors in the separable case), inside the margin (i.e., $\alpha_i = C$ and $y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) < 1$), or errors are non-zero. Without this sparsity property, SVM would not be practical for large data sets.

The final decision boundary is (we note that many α_i's are 0)

$$\langle \mathbf{w} \cdot \mathbf{x} \rangle + b = \sum_{i=1}^{r} y_i \alpha_i \langle \mathbf{x}_i \cdot \mathbf{x} \rangle + b = 0 \tag{75}$$

The decision rule for classification (testing) is the same as the separable case, i.e., $sign(\langle \mathbf{w} \cdot \mathbf{x} \rangle + b)$.

Finally, we still have the problem of determining the parameter C. The

value of C is usually chosen by trying a range of values on the training set to build multiple classifiers and then to test them on a validation set before selecting the one that gives the best classification result on the validation set. Cross-validation is commonly used as well.

3.8.3 Nonlinear SVM: Kernel Functions

The SVM formulations discussed so far require that positive and negative examples can be linearly separated, i.e., the decision boundary must be a hyperplane. However, for many real-life data sets, the decision boundaries are nonlinear. To deal with non-linearly separable data, the same formulation and solution techniques as for the linear case are still used. We only transform the input data from its original space into another space (usually of a much higher dimensional space) so that a linear decision boundary can separate positive and negative examples in the transformed space, which is called the **feature space**. The original data space is called the **input space**.

Thus, the basic idea is to map the data in the input space X to a feature space F via a nonlinear mapping ϕ,

$$\begin{aligned} &\phi : X \to F \\ &\mathbf{x} \mapsto \phi(\mathbf{x}) \end{aligned} \tag{76}$$

After the mapping, the original training data set $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$ becomes:

$$\{(\phi(\mathbf{x}_1), y_1), (\phi(\mathbf{x}_2), y_2), \ldots, (\phi(\mathbf{x}_n), y_n)\} \tag{77}$$

The same linear SVM solution method is then applied to F. Fig. 19 illustrates the process. In the input space (figure on the left), the training examples cannot be linearly separated. In the transformed feature space (figure on the right), they can be separated linearly.

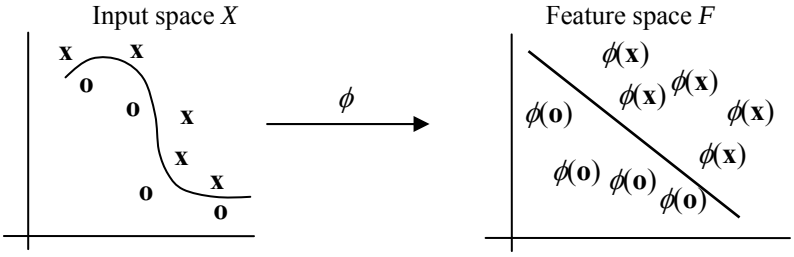


Fig. 19. Transformation from the input space to the feature space.

With the transformation, the optimization problem in (61) becomes

$$\text{Minimize}: \ \frac{\langle \mathbf{w} \cdot \mathbf{w} \rangle}{2} + C\sum_{i=1}^{r} \xi_i \tag{78}$$

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

$$\xi_i \geq 0, \quad i = 1, 2, ..., n$$

The dual is

$$\text{Maximize}: \ L_D = \sum_{i=1}^{r} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{r} y_i y_j \alpha_i \alpha_j \langle \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j) \rangle. \tag{79}$$

$$\text{Subject to}: \ \sum_{i=1}^{r} y_i \alpha_i = 0$$

$$0 \leq \alpha_i \leq C, \quad i = 1, 2, ..., n.$$

The final decision rule for classification (testing) is

$$\sum_{i=1}^{r} y_i \alpha_i \langle \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}) \rangle + b \tag{80}$$

Example 21: Suppose our input space is 2-dimensional, and we choose the following transformation (mapping):

$$(x_1, x_2) \mapsto (x_1^2, x_2^2, \sqrt{2}x_1 x_2) \tag{81}$$

The training example ((2, 3), -1) in the input space is transformed to the following training example in the feature space:

((4, 9, 8.5), -1) ■

The potential problem with this approach of transforming the input data explicitly to a feature space and then applying the linear SVM is that it may suffer from the curse of dimensionality. The number of dimensions in the feature space can be huge with some useful transformations (see below) even with reasonable numbers of attributes in the input space. This makes it computationally infeasible to handle.

Fortunately, explicit transformations can be avoided if we notice that in the dual representation both the construction of the optimal hyperplane (79) in F and the evaluation of the corresponding decision/classification function (80) only require the evaluation of dot products $\langle \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \rangle$ and never the mapped vector $\phi(\mathbf{x})$ in its explicit form. This is a crucial point.

Thus, if we have a way to compute the dot product $\langle \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \rangle$ in the feature space F using the input vectors $\mathbf{x}$ and $\mathbf{z}$ directly, then we would not

need to know the feature vector $\phi(\mathbf{x})$ or even the mapping function ϕ itself. In SVM, this is done through the use of **kernel functions**, denoted by K,

$$K(\mathbf{x}, \mathbf{z}) = \langle \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \rangle, \tag{82}$$

which are exactly the functions for computing dot products in the transformed feature space using input vectors $\mathbf{x}$ and $\mathbf{z}$. An example kernel function is the **polynomial kernel**,

$$K(\mathbf{x}, \mathbf{z}) = \langle \mathbf{x} \cdot \mathbf{z} \rangle^d \tag{83}$$

Example 22: Let us compute this kernel with degree $d = 2$ in a 2-dimensional space. Let $\mathbf{x} = (x_1, x_2)$ and $\mathbf{z} = (z_1, z_2)$.

$$\begin{aligned} \langle \mathbf{x} \cdot \mathbf{z} \rangle^2 &= (x_1 z_1 + x_2 z_2)^2 \\ &= {x_1}^2 {z_1}^2 + 2 x_1 z_1 x_2 z_2 + {x_2}^2 {z_2}^2 \\ &= \langle ({x_1}^2, {x_2}^2, \sqrt{2} x_1 x_2) \cdot ({z_1}^2, {z_2}^2, \sqrt{2} z_1 z_2) \rangle \\ &= \langle \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \rangle, \end{aligned} \tag{84}$$

where $\phi(\mathbf{x}) = ({x_1}^2, {x_2}^2, \sqrt{2} x_1 x_2)$, which shows that the kernel $\langle \mathbf{x} \cdot \mathbf{z} \rangle^2$ is a dot product in the transformed feature space. The number of dimensions in the feature space is 3. Note that $\phi(\mathbf{x})$ is actually the mapping function used in Example 21. Incidentally, in general the number of dimensions in the feature space for the polynomial kernel function $K(\mathbf{x}, \mathbf{z}) = \langle \mathbf{x} \cdot \mathbf{z} \rangle^d$ is $\binom{n+d-1}{d}$, which is a huge number even with a reasonable number of attributes in the input space. Fortunately, by using the kernel function in (83), the huge number of dimensions in the feature space does not matter. ■

The derivation in (84) is only for illustration purposes. We do not need to find the mapping function. We can simply apply the kernel function directly. That is, we replace all the dot products $\langle \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \rangle$ in (79) and (80) with the kernel function $K(\mathbf{x}, \mathbf{z})$ (e.g., the polynomial kernel in (83)). This strategy of directly using a kernel function to replace dot products in the feature space is called the **kernel trick**. We would never need to explicitly know what ϕ is.

However, the question is, how do we know whether a function is a kernel without performing the derivation such as that in (84)? That is, how do we know that a kernel function is indeed a dot product in some feature space? This question is answered by a theorem called the **Mercer's theorem**, which we will not discuss here. See [106] for details.

It is clear that the idea of kernel generalizes the dot product in the input space. This dot product is also a kernel with the feature map being the identity

$$K(\mathbf{x}, \mathbf{z}) = \langle \mathbf{x} \cdot \mathbf{z} \rangle. \tag{85}$$

Commonly used kernels include

$$\text{Polynomial:} \quad K(\mathbf{x}, \mathbf{z}) = (\langle \mathbf{x} \cdot \mathbf{z} \rangle + \theta)^d \tag{86}$$

$$\text{Gaussian RBF:} \quad K(\mathbf{x}, \mathbf{z}) = e^{-\|\mathbf{x}-\mathbf{z}\|^2/2\sigma} \tag{87}$$

where $\theta \in \Re$, $d \in N$, and $\sigma > 0$.

Summary

SVM is a linear learning system that finds the maximal margin decision boundary to separate positive and negative examples. Learning is formulated as a quadratic optimization problem. Non-linear decision boundaries are found via a transformation of the original data to a much higher dimensional feature space. However, this transformation is never explicitly done. Instead, kernel functions are used to compute dot products required in learning without the need to even know the transformation function.

Due to the separation of the learning algorithm and kernel functions, kernels can be studied independently from the learning algorithm. One can design and experiment with different kernel functions without touching the underlying learning algorithm.

SVM also has some limitations:

1. It works only in the real-valued space. For a categorical attribute, we need to convert its categorical values to numeric values. One way to do that is to create an extra binary attribute for each categorical value, and set the attribute value to 1 if the categorical value appears, and 0 otherwise.
2. It allows only two classes, i.e., binary classification. For multiple class classification problems, several strategies can be applied, e.g., one-against-rest, and error-correcting output coding [125].
3. The hyperplane produced by SVM is hard to understand by users. It is difficult to picture where the hyperplane is in a high dimensional space. The matter is made worse by kernels. Thus, SVM is commonly used in applications that do not required human understanding.

3.9 K-Nearest Neighbor Learning

All the previous learning methods learn some kinds of models from the data, e.g., decision trees, a set of rules, and posteriori probabilities. These learning methods are called **eager learning** methods as they learn models of the data before testing. In contrast, *k*-nearest neighbor (*k*NN) is a **lazy learning** method in the sense that no model is learned from the training data. Learning only occurs when a test example needs to be classified. The idea of *k*NN is extremely simple and yet quite effective in many applications, e.g., text classification.

It works as follows: Again let D be the training data set. Nothing will be done on the training examples. When a test instance d is presented, the algorithm compares d with every training example in D to compute the similarity or distance between them. The k most similar (closest) examples in D are then selected. This set of examples is called the ***k* nearest neighbors** of d. d then takes the most frequent class among the k nearest neighbors. Note that $k = 1$ is usually not sufficient for determining the class of d due to noises and outliers in the data. A set of nearest neighbors is needed to accurately decide the class. The general *k*NN algorithm is given in Fig. 20.

Algorithm kNN(D, d, k)
1 Compute the distance between d and every example in D;
2 Choose the k examples in D that are nearest to d, denote the set by P ($\subseteq D$);
3 Assign d the class that is the most frequent class in P (or the majority class);

Fig. 20. The *k*-nearest neighbor algorithm

The key component of a *k*NN algorithm is the **distance/similarity function**, which is chosen based on applications and the nature of the data. For relational data, the Euclidean distance is commonly used. For text documents, cosine similarity is a popular choice. We will introduce these distance functions and many others in the next Chapter.

The number of nearest neighbors k is usually determined by using a validation set, or through cross validation on the training data. That is, a range of k values are tried, and the k value that gives the best accuracy on the validation set (or cross validation) is selected. Fig. 21 illustrates the importance of choosing the right k.

Example 20: In Fig. 21, we have two classes of data, positive (filled squares) and negative (empty circles). If 1-nearest neighbor is used, the test data point $\oplus$ will be classified as negative, and if 2-nearest neighbor are used, the class cannot be decided. If 3-nearest neighbors are used, the class is positive as 2 positive examples are in the 3-nearest neighbors.

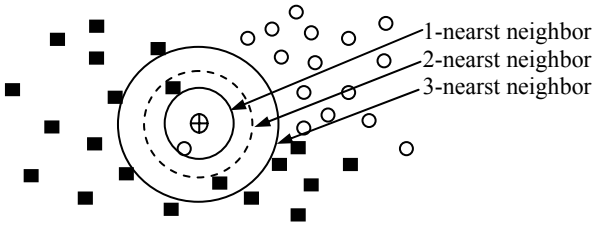


Fig. 21. An illustration of *k*-nearest neighbor classification.

Despite its simplicity, researchers have showed that the classification accuracy of *k*NN can be quite strong and in many cases as accurate as those elaborated methods. For instance, In [505], Yang and Liu showed that *k*NN performs equally well as SVM for some text classification tasks. *k*NN classification can produce arbitrarily shaped decision boundary, which makes it very flexible.

*k*NN is, however, slow at the classification time. Due to the fact that there is no model building, each test instance is compared with every training example at the classification time, which can be quite time consuming especially when the training set *D* and the test set are large. Another disadvantage is that *k*NN does not produce an understandable model. It is thus not applicable if an understandable model is required in the application.

Bibliographic Notes

Supervised learning has been studied extensively by the machine learning community. The book by Mitchell [344] covers most learning techniques and is easy to read. Duda, Hart and Stork's pattern classification book is also a great reference [142]. Additionally, most data mining books have one or two chapters on supervised learning, e.g., those by Han and Kamber [199], Hand et al. [202], Tan et al. [454], and Witten and Frank [487].

For decision tree induction, Quinlan's book [406] has all the details and the code of his decision tree system C4.5, which is what the algorithm described in this chapter is based on. Readers are referred to the book for additional information on handling of missing values, tree pruning and rule generation. The rule generation algorithm of C4.5, which includes pruning, is very slow for large data sets. C5.0, which is the commercial version of C4.5, is much more efficient. However, its algorithm is not published. Other well known decision tree systems include CART by Breiman et al. [58] and CHAD by Kass [243], which use different splitting criteria in tree

building. CART uses the Gini index and CHAD uses the χ^2 test.

Rule induction algorithms generate rules directly from the data. Well known systems include AQ by Michalski et al. [340], CN2 by Clark and Niblett [92], FOIL by Quinlan [405], FOCL by Pazzani et al. [393], I-REP by Furnkranz and Widmer [171], and RIPPER by Cohen [94].

Using association rules to build classifiers was proposed by Liu et al. in [305], which also reported the CBA system. CBA selects a small subset of class association rules as the classifier. Other classifier building techniques include generating only a subset of accurate rules proposed by Cong et al [100, 101], Wang et al [476], and Yin and Han [509], combining multiple rules by Li et al. [292], combining rules with probabilities by Meretakis and Wüthrich [338], and several others by Antonie and Zaiane [21], Dong et al [136], Li et al [284, 258], and Zaki and Aggarwal [518].

The naïve Bayesian classification model described in Section 3.6 is based on the papers by Domingos and Pazzani [135], Kohavi et al. [255] and Langley et al [267]. The naïve Bayesian classification for text discussed in Section 3.7 is based on the multinomial model given by McCallum and Nigam [326]. This model was also used earlier by Lewis and Gale [282], Li and Yamanishi [283], and Nigam et al. [369]. Another formulation of naïve Baye is based on the multi-variate Bernoulli model, which was used in Lewis [281], and Robertson and Sparck-Jones [415].

There are also several classifier combination methods (or ensemble methods) which use multiple classifiers to achieve a better accuracy. Well known methods are bagging by Breiman [59], boosting by Freund and Schapire [168]. Apart from these two, there are several other techniques such as Arcing by Breiman [60], which is a variant of boosting, random forest also by Breiman [61], and stacking by Wolpert [490].

Support vector machines (SVM) was first introduced by Vapnik and his colleagues in 1992 [55]. Further details were given in his 1995 book "The Nature of Statistical Learning Theory" [466]. Two other books on SVM and kernel methods are those by Cristianini and Shawe-Taylor [106] and Scholkopf and Smola [429]. The discussion of SVM in this chapter is heavily influenced by Cristianini and Shawe-Taylor's book and the tutorial paper by Burges [68]. One can find many other SVM tutorials, survey articles (published or unpublished) and presentation slides on the Web. Joachims [234, 235] showed superb accuracy of SVM on text classification. Yang and Liu [505] compares SVM with other methods and confirm it. Two popular SVM systems are SVM^{Light} by Joachim (available at http://svmlight.joachims.org/) and LIBSVM (available at http://www.csie.ntu.edu.tw/~cjlin/libsvm/).

Chapter 4: Unsupervised Learning

Supervised learning discovers patterns in the data that relate data attributes to a class attribute. These patterns are then utilized to predict the values of the class attribute of future data instances. These classes indicate some real-world predictive or classification tasks such as determining whether a news article belongs to the category of sports or politics, or whether a patient has a particular disease. However, in some other applications, the data have no class attribute. The user wants to explore the data to find some intrinsic structures in them. Clustering is one technique for finding such structures, which organizes data instances into **similarity groups**, called **clusters** such that the data instances in the same cluster are similar to each other and data instances in different clusters are very different from each other. Clustering is often called **unsupervised learning**, because unlike supervised learning, no class values denoting an *a priori* partition or grouping of the data are given. Note that according to this definition, we can also say that association rule mining is an unsupervised learning task. However, due to historical reasons, clustering is closely associated and even synonymous with unsupervised learning while association rule mining is not. We follow this convention, and describe some main clustering techniques in this chapter.

Clustering has been shown to be one of the most commonly used data analysis techniques. It also has a long history, and has been used in almost every field, e.g., medicine, psychology, botany, sociology, biology, archeology, marketing, insurance, libraries, etc. In recent years, due to the rapid increase of online documents and the expansion of the Web, text document clustering too has become a very important task.

4.1 Basic Concepts

Clustering is the process of organizing data instances into groups whose members are similar in some way. A **cluster** is therefore a collection of data instances which are "similar" to each other and are "dissimilar" to data instances in other clusters. In the clustering literature, a data instance

is also called an **object** as the instance may represent an object in the real-world. It is also called a **data point** as it can be seen as a point in an r-dimension space, where r is the number of attributes in the data.

Fig. 1 shows a 2-dimensional data set. We can clearly see three groups of data points. Each group is a cluster. The task of clustering is to find the three clusters hidden in the data. Although it is easy for a human user to visually detect clusters in a 2-dimensional or even 3-demensional space, it becomes very hard, if not impossible, to detect clusters through human eyes as the number of dimensions increases. Additionally, in many applications, clusters are not as clear-cut or well separated as the three clusters in Fig. 1. Automatic techniques are thus needed for clustering.

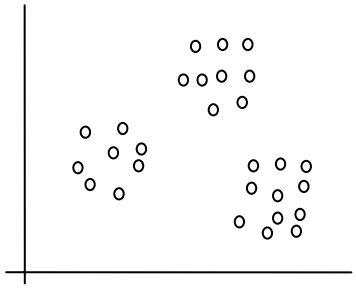


Fig. 1. Three natural groups or clusters of data points

After seeing the example in Fig. 1, you may ask the question: What is clustering for? To answer it, let us see some application examples from different domains.

Example 1: A company wants to conduct a marketing campaign to promote its products. The most effective strategy is to design a set of personalized marketing materials for each individual customer according to his/her profile and financial situation. However, this is too expensive for a large number of customers. On the other extreme, the company designs only one set of marketing materials to be used for all customers. This one-size-fits-all approach, however, may not be effective. The most cost-effective approach is to segment the customers into a small number of groups according to their similarities and design some targeted marketing materials for each group. This segmentation task is commonly done using clustering algorithms, which **partitions** customers into similarity groups. In marketing research, clustering is often called **segmentation**. ■

Example 2: A company wants to produce and sell T-shirts. Similar to the case above, on one extreme, for each customer it can measure his/her sizes and has a T-shirt tailor-made for him/her. Obviously, this T-shirt is going to be expensive. On the other extreme, only one size of T-shirts is made.

Since this size may not fit most people, the company will not be able to sell many such T-shirts. Again, the most cost effective way is to group people based on their sizes and make a different generalized size of T-shirts for each group. This is why we see small, medium and large size T-shirts in shopping malls, and seldom see T-Shirts with only a single size. The method used to group people according to their sizes is clustering. The process is usually as follows: The T-shirt manufacturer first samples a large number of people and measure their sizes to produce a measurement database. It then clusters the data, which **partitions** the data into some similarity subsets, i.e., clusters. For each cluster, it computes the averages of the sizes and then uses the averages to mass-produce T-shirts for all people of similar sizes. ■

Example 3: Everyday, news agencies around the world generate a large number of news articles. If a Web site wants to collect these news articles to provide an integrated news service, it has to organize the collected articles according to some topic hierarchy. The question is: What should the topics be, and how should they be organized? One possibility is to employ a group of human editors to do the job. However, the manual organization is costly and very time consuming, which makes it unsuitable for news and other time sensitive information. Throwing all the news articles to the readers with no organization is clearly not an option. Although classification is able to classify news articles according to predefined topics, it is not applicable here because classification needs training data, which have to be manually labeled with topic classes. Since news topics change constantly and rapidly, the training data need to change constantly as well, which is infeasible via manual labeling. Clustering is clearly a solution for this problem because it automatically groups a stream of news articles based on their content similarities. **Hierarchical clustering algorithms** can also organize documents hierarchically, i.e., each topic may contain sub-topics and so on. Topic hierarchies are particularly useful for texts. ■

Example 4: A general question facing researchers in many areas of inquiry is how to organize observed objects into taxonomic structures. For example, biologists have to organize different species of animals before a meaningful description of the differences between animals is possible. According to the system employed in biology, man belongs to the primates, the mammals, the amniotes, etc. In this organization, the higher the level of aggregation, the less similar are the members in the respective cluster. Man has more in common with all other primates (e.g., apes) than it does with the more "distant" members of the mammals (e.g., dogs). This **hierarchical organization** can be generated by a hierarchical clustering algorithm using features of the objects. ■

The above four examples indicate two types of clustering, **partitional** and **hierarchical**. Indeed, these are the two most important types of clustering approaches. We will study some specific algorithms of these two types of clustering.

Our discussion and examples above also indicate that clustering needs a similarity function to measure how similar two data points (or objects) are, or alternatively a **distance function** to measure the distance between two data points. We will use distance functions in this chapter. The goal of clustering is thus to discover the intrinsic grouping of the input data through the use of a clustering algorithm and a distance function.

4.2 K-means Clustering

The *k*-means algorithm is the best known **partitional algorithm.** It is also perhaps the most widely used among all clustering algorithms due to its simplicity and efficiency. Given a set of data points and the required number *k* of clusters (*k* is specified by the user), this algorithm iteratively partitions the data into *k* clusters based on a distance function.

4.2.1 K-means Algorithm

Let the set of data points (or instances) *D* be

$$\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\},$$

where $\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{ir})$ is a vector in a real-valued space $X \subseteq \Re^r$, and *r* is the number of attributes in the data (or the number of dimensions of the **data space**). The *k*-means algorithm partitions the given data into *k* clusters. Each cluster has a cluster **center**, which is often called the cluster **centroid**. The centroid, usually used to represent a cluster, is simply the mean of all the data points in the cluster, which gives the name to the algorithm, i.e., since there are *k* clusters and thus *k* means. Fig. 2 gives the *k*-means clustering algorithm.

At the beginning, the algorithm randomly selects *k* data points as the **seed** centroids. It then computes the distance between each seed centroid and every other data point, and each data point is assigned to the centroid that is closest to it. A centroid and its data points therefore represent a cluster. Once all the data points in the data are assigned, the centroid for each cluster is re-computed using the data points in the current cluster. This process repeats until a stopping criterion is met. The stopping (or convergence) criterion can be anyone of the following:

Algorithm *k*-means(*k*, *D*)
1 Choose *k* data points as the initial centroids (cluster centers)
2 **repeat**
3 **for** each data point $\mathbf{x} \in D$ do
4 compute the distance from **x** to each centroid;
5 assign **x** to the closest centroid // a centroid represents a cluster
6 **endfor**
7 re-compute the centroid using the current cluster memberships
8 **until** the stopping criterion is met

Fig. 2. The *k*-means algorithm

1. no (or minimum) re-assignments of data points to different clusters,
2. no (or minimum) change of centroids, or
3. minimum decrease in the **sum of squared error** (SSE),

$$SSE = \sum_{j=1}^{k} \sum_{\mathbf{x} \in C_j} dist(\mathbf{x}, \mathbf{m}_j)^2 \tag{1}$$

where k is the number of required clusters, C_j is the jth cluster, $\mathbf{m}_j$ is the centroid of cluster C_j (the mean vector of all the data points in C_j), and $dist(\mathbf{x}, \mathbf{m}_j)$ is the distance between data point **x** and centroid $\mathbf{m}_j$.

The *k*-means algorithm can be used for any application data set where the **mean** can be defined and computed. In the **Euclidean space**, the mean of a cluster is computed with:

$$\mathbf{m}_j = \frac{1}{|C_j|} \sum_{\mathbf{x}_i \in C_j} \mathbf{x}_i \tag{2}$$

where $|C_j|$ is the number of data points in cluster C_j. The distance from a data point $\mathbf{x}_i$ to a cluster mean (centroid) $\mathbf{m}_j$ is computed with

$$\begin{aligned} dist(\mathbf{x}_i, \mathbf{m}_j) &= \| \mathbf{x}_i - \mathbf{m}_j \| \\ &= \sqrt{(x_{i1} - m_{j1})^2 + (x_{i2} - m_{j2})^2 + ... + (x_{ir} - m_{jr})^2} \end{aligned} \tag{3}$$

Example 5: Figure 3(A) shows a set of data points in a 2-dimensional space. We want to find 2 clusters from the data, i.e., $k = 2$. First, two data points (each marked with a cross) are randomly selected to be the initial centroids (or seeds) shown in Fig. 3(A). The algorithm then goes to the first iteration (the repeat-loop).

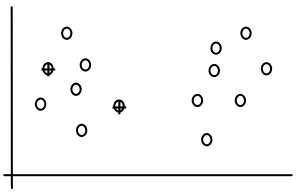
Iteration 1: Each data point is assigned to its closest centroid to form 2 clusters. The resulting clusters are given in Fig. 3(B). Then the cen-

troids are re-computed based on the data points in the current clusters (Fig. 3(C). This leads to iteration 2.

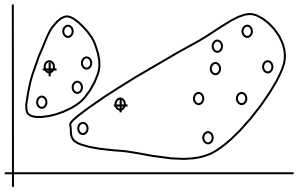
Iteration 2: Again, each data point is assigned to its closest new centroid to form 2 new clusters shown in Fig. 3(D). The centroids are then re-computed. The new centroids are shown in Fig. 3(E).

Iteration 3: The same operations are performed as in the first two iterations. Since there is no re-assignment of data points to different clusters in this iteration, the algorithm ends.

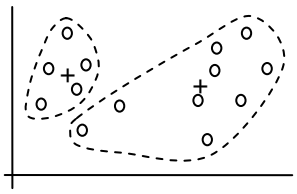
The final clusters are those given in Fig. 3(G). The set of data points in each cluster and its centroid are output to the user.



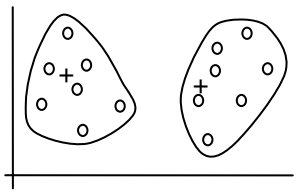
(A). Random selection of *k* seeds (or centroids)



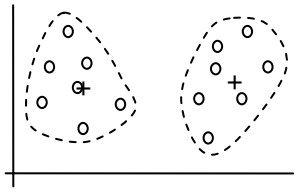
Iteration 1: (B). Cluster assignment



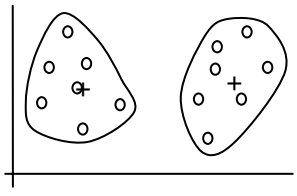
(C). Re-compute centroids



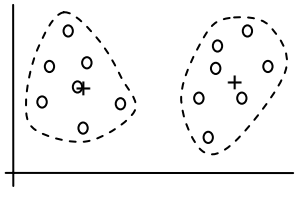
Iteration 2: (D). Cluster assignment



(E). Re-compute centroids



Iteration 3: (F). Cluster assignment



(G). Re-compute centroids

Fig. 3. The working of the *k*-means algorithm through an example ■

One problem with the *k*-means algorithm is that some clusters may become empty during the clustering process since no data point is assigned to them. Such clusters are called **empty clusters**. To deal with an empty cluster, we can choose a data point as the replacement centroid, e.g., a data point that is furthest from the centroid of a large cluster. If the sum of the square error (SSE) is used as the stopping criterion, the cluster with the largest SSE value may be used to find another centroid.

4.2.2 Disk Version of the *K*-means Algorithm

The *k*-means algorithm may be implemented in such a way that it does not need to load the entire data set into the main memory, which is useful for large data sets. Notice that the centroids for the *k* clusters can be computed incrementally in each iteration because the summation in Equation (2) can be calculated separately first. During the clustering process, the number of data points in each cluster can be counted incrementally as well. This gives us a disk based implementation of the algorithm (Fig. 4), which produces exactly the same clusters as that in Fig. 2, but with the data on disk. In each for-loop, the algorithm simply scans the data once.

The whole clustering process thus scans the data *t* times, where *t* is the number of iterations before convergence, which is usually not very large (< 50). In applications, it is quite common to set a limit on the number of iterations because later iterations typically result in only minor changes to the clusters. Thus, this algorithm may be used to cluster large data sets which cannot be loaded into the main memory. Although there are several special algorithms that scale-up clustering algorithms to large data sets, they all require sophisticated techniques.

Algorithm disk-*k*-means(*k*, *D*)
1 Choose *k* data points as the initial centriods $\mathbf{m}_j$, $j = 1, \ldots, k$;
2 **repeat**
3 initialize $\mathbf{s}_j = \mathbf{0}$, $j = 1, \ldots, k$; // **0** is a vector with all 0's
4 initialize $n_j = 0$, $j = 1, \ldots, k$; // n_j is the number points in cluster j
5 **for** each data point $\mathbf{x} \in D$ **do**
6 $j = \arg\min_j dist(\mathbf{x}, \mathbf{m}_j)$;
7 assign **x** to the cluster *j*;
8 $\mathbf{s}_j = \mathbf{s}_j + \mathbf{x}$;
9 $n_j = n_j + 1$;
10 **endfor**
11 $\mathbf{m}_{ji} = \mathbf{s}_j / n_j$, $ji = 1, \ldots, k$;
12 **until** the stopping criterion is met

Fig. 4. A simple disk version of the *k*-means algorithm

Let us give some explanations to this algorithm. Line 1 does exactly the same thing as the algorithm in Fig. 2. Line 3 initializes $\mathbf{s}_j$ which is used to incrementally compute the sum in Equation (2) (line 8). Line 4 initializes n_j which records the number of data points assigned to cluster j (line 9). Lines 6 and 7 perform exactly the same tasks as lines 4 and 5 in the original algorithm in Fig. 2. Line 11 re-computes the centroids, which are used in the next iteration. Any of the three stopping criteria may be used here. If the sum of square error is applied, we can modify the algorithm slightly to compute the sum of square error incrementally.

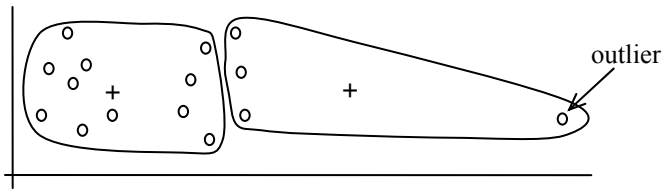
4.2.3 Strengths and Weaknesses

The main strengths of the *k*-means algorithm are its simplicity and efficiency. It is easy to understand and easy to implement. Its time complexity is $O(tkn)$, where n is the number of data points, k is the number of clusters, and t is the number of iterations. Since both k and t are normally much smaller than n. The *k*-means algorithm is considered a linear algorithm in the number of data points.

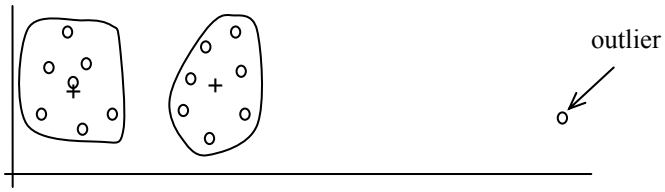
The weaknesses and ways to address them are as follows:

1. The algorithm is only applicable to data sets where the notion of the *mean* is defined. Thus, it is difficult to apply to categorical data sets. There is, however, a variation of the *k*-means algorithm called ***k*-modes**, which clusters categorical data. The algorithm uses the mode instead of the mean as the centroid. Assuming that the data instances are described by r categorical attributes, the mode of a cluster C_j is a tuple $\mathbf{m}_j = (m_{j1}, m_{j2}, \ldots, m_{jr})$ where m_{ji} is the most frequent value of the ith attribute of the data instances in cluster C_j. The similarity (distance) between a data instance and a mode is the number of values that they match (do not match).
2. The user needs to specify the number of clusters k in advance. In practice, several k values are tried and the one that gives the most desirable result is selected. We will discuss the evaluation of clusters later.
3. The algorithm is sensitive to **outliers**. Outliers are data points that are very far away from other data points. Outliers could be errors in the data recording or some special data points with very different values. For example, in an employee data set, the salary of the Chief-Executive-Officer (CEO) of the company may be considered as an outlier because its value could be many times larger than everyone else. Since the *k*-means algorithm uses the mean as the centroid of each cluster, outliers may result in undesirable clusters as the following example shows.

Example 6: In Fig. 5(A), due to an outlier data point, the resulting two clusters do not reflect the natural groupings in the data. The ideal clusters are shown in Fig. 5(B). The outlier should be identified and reported to the user.



(A): Undesirable clusters



(B): Ideal clusters

Fig. 5. Clustering with and without the effect of outliers ■

There are several methods for dealing with outliers. One simple method is to remove some data points in the clustering process that are much further away from the centroids than other data points. To be safe, we may want to monitor these possible outliers over a few iterations and then decide whether to remove them. It is possible that a very small cluster of data points may be outliers. Usually, a threshold value is used to make the decision.

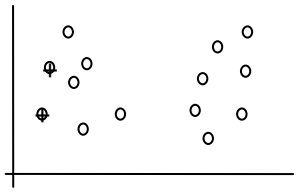
Another method is to perform random sampling. Since in sampling we only choose a small subset of the data points, the chance of selecting an outlier is very small. We can use the sample to do a pre-clustering and then assign the rest of the data points to these clusters, which may be done in any of the three ways.

- Assign each remaining data point to the centroid closest to it. This is the simplest method.
- Use the clusters produced from the sample to perform supervised learning (classification). Each cluster is regarded as a class. The clustered sample is thus treated as the training data for learning. The resulting classifier is then applied to classify the remaining data points into appropriate classes or clusters.

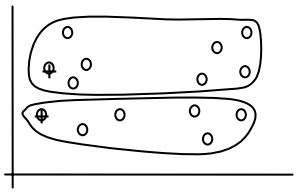
- Use the clusters produced from the sample as seeds to perform **semi-supervised learning**. Semi-supervised learning is a new learning model that learns from a small set of labeled examples (with classes) and a large set of unlabeled examples (without classes). In our case, the clustered sample data are used as the labeled set and the remaining data points are used as the unlabeled set. The results of the learning naturally cluster all the remaining data points. We will study this technique in the next Chapter.

4. The algorithm is sensitive to **initial seeds**, which are the initially selected centroids. Different initial seeds may result in different clusters. Thus, if the sum of square error is used as the stopping criterion, the algorithm only achieves **local optimal**. The global optimal is computationally infeasible for large data sets.

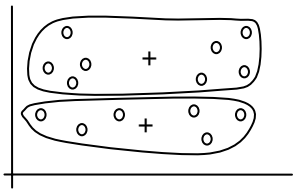
Example 7: Fig. 6 shows the clustering process of a 2-dimensional data set. The goal is to find two clusters. The randomly selected initial seeds are marked with crosses in Fig. 6(A). Fig. 6(B) gives the clustering result of the first iteration. Fig. 6(C) gives the result of the second iteration. Since there is no re-assignment of data points, the algorithm stops.



(A). Random selection of seeds (centroids)



(B). Iteration 1

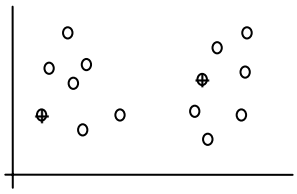


(C). Iteration 2

Fig. 6. Poor initial seeds (centroids)

If the initial seeds are different, we may obtain entirely different clusters as Fig. 7 shows. Fig. 7 uses the same data as Fig. 6, but different initial seeds (Fig. 7(A)). After two iterations, the algorithm ends, and the final clusters are given in Fig. 7(C). These two clusters are more reason-

able than the two clusters in Fig. 6(C), which indicates that the choice of the initial seeds in Fig. 6(A) is poor.



(A). Random selection of k seeds (centroids)

(B). Iteration 1 (C). Iteration 2

Fig. 7. Good initial seed (centroids) ■

To select good initial seeds, researchers have proposed several methods. One simple method is to first compute the mean **m** (the centroid) of the entire data set (any random data point can be used as well). Then the first seed data point $\mathbf{x}_1$ is selected that is furthest from the mean **m**. The second data point $\mathbf{x}_2$ is selected that is furthest from $\mathbf{x}_1$. Each subsequent data point $\mathbf{x}_i$ is selected such that the sum of distances from $\mathbf{x}_i$ to those already selected data points is the largest. However, if the data has outliers, the method will not work well. To deal with outliers, again, we can randomly select a small sample of the data and perform the same operation on the sample. As we discussed above, since the number of outliers is small, the chance that they show up in the sample is very small. Using sampling also reduces the computation time because computing the mean and selecting each initial seed needs one scan of the data.

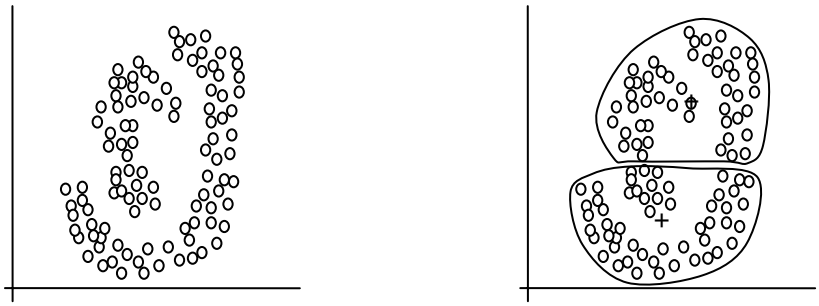
Another method is to sample the data and use the sample to perform hierarchical clustering, which we will discuss in Section 4.4. The centroids of the resulting k clusters are used as the initial seeds.

Yet another approach is to manually select seeds. This may not be a difficult task for text clustering applications because it is easy for human users to read some documents and pick some good seeds. These seeds may help improve the clustering result significantly and also enable the system to produce clusters that meet the user's needs.

5. The *k*-means algorithm is not suitable for discovering clusters that are not hyper-ellipsoids (or hyper-spheres).

Example 8: Fig. 8(A) shows a 2-dimensional data set. There are two irregular shaped clusters. However, the two clusters are not hyper-ellipsoids, which means that the *k*-means algorithm will not be able to find them. Instead, it may find the two clusters shown in Fig. 8(B).

The question is: are the two clusters in Fig. 8(B) necessarily bad? The answer is "no", and that it depends on the application. It is not true that a clustering algorithm that is able to find arbitrarily shaped clusters is always better. We will discuss this issue in Section 4.3.2.



(A): Two natural clusters (B): *k*-means clusters

Fig. 8. Natural (but irregular) clusters and *k*-means clusters ■

Despite these weaknesses, *k*-means is still the most popular algorithm in practice due to its simplicity, efficiency and the fact that other clustering algorithms have their own lists of weaknesses. There is no clear evidence showing that any other clustering algorithm performs better than the *k*-means algorithm in general, although they may be more suitable for some specific types of data or applications. Note that comparing different clustering algorithms is a very difficult task because unlike supervised learning, nobody knows what the correct clusters are, especially in high dimensional spaces. There are several cluster evaluation methods but they all have drawbacks. We will discuss the evaluation issue in Section 4.9.

4.3 Representation of Clusters

Once a set of clusters is found, the next task is to find a way to represent the clusters. In some applications, outputting the set of data points that makes up the cluster to the user is sufficient. However, in other applications that involve decision making, the resulting clusters need to be repre-

sented in a compact and understandable way, which also facilitates the evaluation of the resulting clusters.

4.3.1 Common Ways to Represent Clusters

There are three main ways to represent clusters:

1. Use the centroid of each cluster to represent the cluster. This is the most popular way. The centroid tells where the center of the cluster is. One may also compute the radius and standard deviation of the cluster to determine its spread in each dimension. The centroid representation alone works well if the clusters are of the hyper-spherical shape. If clusters are elongated or are of other shapes, centroids may not be suitable.
2. Use classification models to represent clusters. In this method, we treat each cluster as a class. That is, all the data points in a cluster are regarded to have the same class label, e.g., the cluster ID. We then run a supervised learning algorithm on the data to find a classification model. For example, we may use the decision tree learning to distinguish the clusters. The resulting tree or set of rules provide an understandable representation of the clusters.

   Fig. 9 shows a partitioning produced by a decision tree algorithm. The original clustering produced three clusters. Data points in cluster 1 are represented by 1's, data points in cluster 2 are represented by 2's, and data points in cluster 3 are represented by 3's. We can see that the three clusters are separated and each can be represented with a rule.

   $x \le 2 \rightarrow$ cluster 1
   $x > 2, y > 1.5 \rightarrow$ cluster 2
   $x > 2, y \le 1.5 \rightarrow$ cluster 3

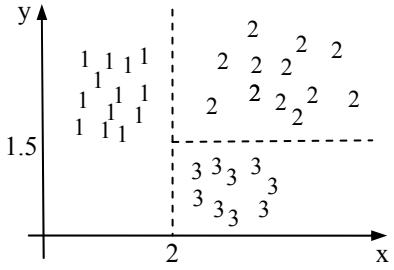


Fig. 9. Description of clusters using rules

We make two remarks about this representation method:

- The partitioning in Fig. 9 is an ideal case as each cluster is repre-

sented by a single rectangle (or rule). However, in most applications, the situation may not be so ideal. A cluster may be split into a few hyper-rectangles or rules. However, there is usually a dominant or large rule which covers most of the data points in the cluster.

- One can use the set of rules to evaluate the clusters to see whether they conform to some existing domain knowledge or intuition.

3. Use frequent values in each cluster to represent it. This method is mainly for clustering of categorical data (e.g., in the *k*-modes clustering). It is also the key method used in text clustering, where a small set of frequent words in each cluster is selected to represent the cluster.

4.3.2 Clusters of Arbitrary Shapes

Hyper-elliptical and hyper-spherical clusters are usually easy to represent, using their centroid together with spreads (e.g., standard deviations), a rule, or a combination of both. However, other arbitrary shape clusters, like the natural clusters show in Fig 8(A), are hard to represent especially in high dimensional spaces.

A common criticism about an algorithm like *k*-means is that it is not able to find arbitrarily shaped clusters. However, this criticism may not be as bad as it sounds because whether one type of clustering is desirable or not depends on its application. Let us use the natural clusters in Fig. 8(A) to discuss this issue together with an artificial application.

Example 9: Assume that the data shown in Fig. 8(A) is the measurement data of people's physical sizes. We want to group people based on their sizes into only two groups in order to mass-produce T-shirts of only 2 sizes (say large and small). Even if the measurement data indicate two natural clusters as in Fig. 8(A), it is difficult to use the clusters because we need centroids of the clusters to design T-shirts. The clusters in Fig. 8(B) are in fact better because they provide us the centroids that are representative of the surrounding data points. If we use the centroids of the two natural clusters as shown in Fig. 10 to make T-shirts, it is clearly inappropriate because they are too near to each other in this case. In general, it does not make good sense to define the concept of center or centroid for an irregularly shaped cluster. ■

Note that clusters of arbitrary shapes can be found by neighborhood search algorithms such as some hierarchical clustering methods, and density-based clustering methods [150]. Due to the difficulty of representing an arbitrarily shaped cluster, an algorithm that finds such clusters may only output a list of data points in each cluster, which are not easy to use. Such kinds of clusters are more useful in spatial and image processing applica-

tions, but less useful in others. This partially explains why the *k*-means algorithm is so popular while most methods that find arbitrarily shaped (natural) clusters are not.

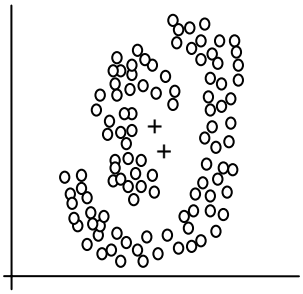


Fig. 10. Two natural clusters and their centroids (two classes)

4.4 Hierarchical Clustering

Hierarchical clustering is another major clustering approach. It has a number of desirable properties which make it popular. It clusters by producing a nested sequence of clusters like a **tree** (also called a **dendrogram**). Singleton clusters (individual data points) are at the bottom of the tree and one root cluster is at the top, which covers all data points. Each internal cluster node contains child cluster nodes. Sibling clusters partition the data points covered by their common parent. Fig. 11 shows an example.

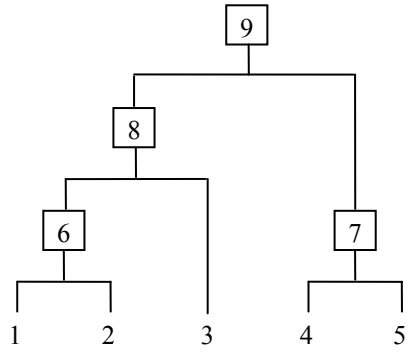


Fig. 11. Hierarchical clustering

At the bottom of the tree, there are 5 clusters (5 data points). At the next level, cluster 6 contains data points 1 and 2, and cluster 7 contains data points 4 and 5. As we move up the tree, we have fewer and fewer clusters.

Since the whole clustering tree is stored, the user can choose to view clusters at any level of the tree.

There are two types of hierarchical clustering methods:

Agglomerative (**bottom up**) **clustering**: It builds the dendrogram (tree) from the bottom level, and merges the most similar (or nearest) pair of clusters at each level to go one level up. The process continues until all the data points are merged into a single cluster (i.e., the root cluster).

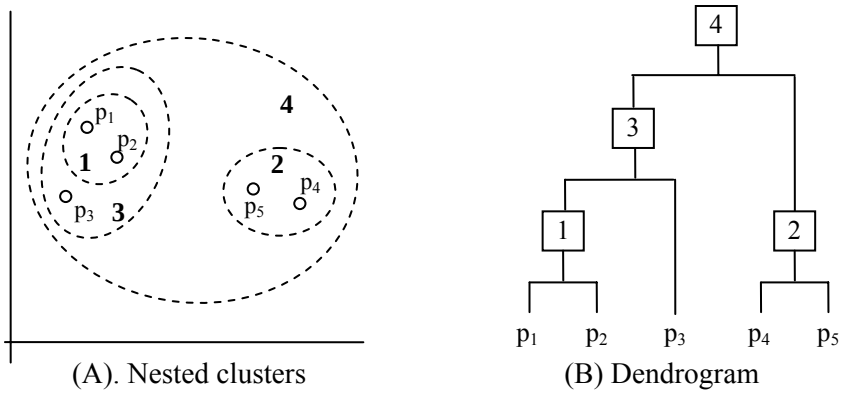
Divisive (**top down**) **clustering**: It starts with all data points in one cluster, the root. It then splits the root into a set of child clusters. Each child cluster is recursively divided further until only singleton clusters of individual data points remain, i.e., each cluster with only a single point.

Agglomerative methods are much more popular than divisive methods. We will focus on agglomerative hierarchical clustering. The general agglomerative algorithm is given in Fig. 12.

Algorithm Agglomerative(D)
1 Make each data point in the data set D a cluster,
2 Compute all pair-wise distances of $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in D$;
2 **repeat**
3 find two clusters that are nearest to each other;
4 merge the two clusters form a new cluster c;
5 compute the distance from c to all other clusters;
12 **until** there is only one cluster left

Fig. 12. The agglomerative hierarchical clustering algorithm

Example 10: Fig. 13 illustrates the working of the algorithm. The data points are in a 2-dimensional space. Fig. 13(A) shows the sequence of nested clusters, and Fig. 13(B) gives the dendrogram.



(A). Nested clusters (B) Dendrogram

Fig. 13. The working of the agglomerative hierarchical clustering algorithm ■

Note that centroids are not used in this case because unlike the *k*-means algorithm, hierarchical clustering may use several methods to determine the distance between two clusters. We introduce these methods next.

4.4.1 Single-Link Method

In **single-link** (or **single linkage**) hierarchical clustering, the distance between two clusters is the distance between two closest data points in the two clusters (one data point from each cluster). In other words, the single link clustering merges the two clusters in each step whose two nearest data points (or members) have the smallest distance, i.e., the two clusters with the **smallest minimum** pair-wise distance. The single-link method is suitable for finding non-elliptical shape clusters. However, it can be sensitive to noise in the data, which may cause a **chain effect** and produce straggly clusters. Fig. 14 illustrates this situation. The noisy data points (represented with filled circles) in the middle connect two natural clusters and split one of them.

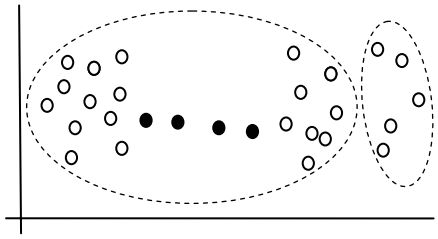


Fig. 14. Chain effect of the single link method

With suitable data structures, single-link hierarchical clustering can be done in $O(n^2)$ time, where n is the number of data points. This is much slower than the *k*-means method, which performs clustering in linear time.

4.4.2 Complete-Link Method

In **complete-link** (or **complete linkage**) clustering, the distance between two clusters is the **maximum** of all pair-wise distances between the data points in the two clusters. In other words, the complete link clustering merges the two clusters in each step whose two furthest data points have the smallest distance, i.e., the two clusters with the smallest maximum pair-wise distance. Fig. 15 shows the clusters produced by complete-link clustering using the same data as in Fig. 14.

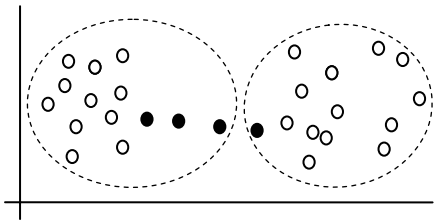


Fig. 15. Clustering using the complete link method

Although the complete-link method does not have the problem of chain effect, it can be sensitive to outliers. Despite this limitation, it has been observed that the complete-link method usually produces better clusters than the single-link method. The worse case time complexity of the complete-link clustering is $O(n^2 \log n)$, where n is the number of data points.

4.4.3 Average-Link Method

This is a compromise between the sensitivity of complete-link clustering to outliers and the tendency of single-link clustering to form long chains that do not correspond to the intuitive notion of clusters as compact, spherical objects. In this method, the distance between two clusters is the average distance of all pair-wise distances between the data points in two clusters. The time complexity of this method is also $O(n^2 \log n)$.

Apart from the above three popular methods, there are several others. The following two methods are also commonly used:

Centroid method: In this method, the distance between two clusters is the distance between their centroids.

Ward's method: In this method, the distance between two clusters is defined as the increase in the sum of squared error (distances) that results when two clusters are merged. Thus, the clusters to be merged in the next step are the ones that will increase the sum the least. Recall the sum of squared error (SSE) is one of the measures used in the *k*-means clustering.

4.4.4. Strengths and Weaknesses

Hierarchical clustering has several advantages compared to the *k*-means and other partitioning clustering methods. It is able to take any forms of distance or similarity functions. Unlike the *k*-means algorithm which only gives *k* clusters at the end, the nested sequence of clusters enables the user to explore clusters at any level of detail (or granularity). Hierarchical clus-

tering can also find clusters of arbitrary shapes, e.g., using the single-link method. Furthermore, the resulting hierarchy can be very useful in its own right. For example, in text document clustering, the cluster hierarchy may represent a topic hierarchy in the documents. Some studies showed that agglomerative hierarchical clustering often produces better clusters than the *k*-means method, but it is considerably slower.

Hierarchical clustering also has several weaknesses. As we discussed with the individual methods, the single-link method may suffer from the chain effect, and the complete-link method is sensitive to outliers. The main shortcomings of all the hierarchical clustering methods are their computation complexities and space requirements, which are at least quadratic. Compared to the *k*-means algorithm, this is very slow and not practical for large data sets. One can use sampling to deal with the efficiency problem. A small sample is taken to do clustering and then the rest of the data points are assigned to each cluster either by distance comparison or by supervised learning (see Section 4.3.1). Some **scale-up methods** may also be applied to large data sets. The main idea of the scale-up methods is to find many small clusters first using an efficient algorithm, and then use the centroids of these small clusters to represent the small clusters to perform the final hierarchical clustering (see the BIRCH method in [535]).

4.5 Distance Functions

Distance or similarity functions play central roles in all clustering algorithms. Numerous distance functions have been reported in the literature and used in applications. Different distance functions are used for different types of attributes (also called **variables**).

4.5.1 Numeric Attributes

The most commonly used distance functions for numeric attributes are the **Euclidean distance** and **Manhattan** (**city block**) **distance**. Both distance measures are special cases of a more general distance function called the **Minkowski distance**. We use $dist(\mathbf{x}_i, \mathbf{x}_j)$ to denote the distance between two data points of r dimensions. The Minkowski distance is:

$$dist(\mathbf{x}_i, \mathbf{x}_j) = ((x_{i1} - x_{j1})^h + (x_{i2} - x_{j2})^h + \ldots + (x_{ir} - x_{jr})^h)^{\frac{1}{h}} \quad (4)$$

where h is a positive integer.

If $h = 2$, it is the **Euclidean distance**,

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{(x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2 + ... + (x_{ir} - x_{jr})^2} \quad (5)$$

If $h = 1$, it is the **Manhattan distance**,

$$dist(\mathbf{x}_i, \mathbf{x}_j) = | x_{i1} - x_{j1} | + | x_{i2} - x_{j2} | + ... + | x_{ir} - x_{jr} | \quad (6)$$

Other common distance functions include:

Weighted Euclidean distance: A weight is associated with each attribute to express its importance in relation to other attributes.

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{w_1(x_{i1} - x_{j1})^2 + w_2(x_{i2} - x_{j2})^2 + ... + w_r(x_{ir} - x_{jr})^2} \quad (7)$$

Squared Euclidean distance: We square the standard Euclidean distance in order to place progressively greater weight on data points that are further apart. The distance is thus

$$dist(\mathbf{x}_i, \mathbf{x}_j) = (x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2 + ... + (x_{ir} - x_{jr})^2 \quad (8)$$

Chebychev distance: This distance measure may be appropriate in cases when one wants to define two data points as "different" if they are different on any one of the attributes. The Chebychev distance is defined as:

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \max(| x_{i1} - x_{j1} |, | x_{i2} - x_{j2} |, ..., | x_{ir} - x_{jr} |) \quad (9)$$

4.5.2 Binary and Nominal Attributes

The above distance measures are only appropriate for numeric attributes. For binary and **nominal** attributes (also called **unordered categorical** attributes), we need different functions. Let us discuss binary attributes first.

A **binary attribute** has two states or values, usually represented by 1 and 0. The two states have no logical numerical ordering. For example, Gender has two values, "male" and "female", which have no ordering relations but are just different. Existing distance functions for binary attributes are based on the proportion of value matches in two data points. A match means that, for a particular attribute, both data points have the same value. It is convenient to use a confusion matrix to introduce these measures. Given the ith and jth data points, $\mathbf{x}_i$ and $\mathbf{x}_j$, we can construct the following confusion matrix:

		Data point $\mathbf{x}_j$ 1	0	
Data point $\mathbf{x}_i$	1	a	b	$a+b$
	0	c	d	$c+d$
		$a+c$	$b+d$	$a+b+c+d$

(10)

- a: the number of attributes with the value of 1 for both data points.
- b: the number of attributes for which $x_{if} = 1$ and $x_{jf} = 0$, where x_{if} (x_{jf}) is the value of the fth attribute of the data point $\mathbf{x}_i$ ($\mathbf{x}_j$).
- c: the number of attributes for which $x_{if} = 0$ and $x_{jf} = 1$.
- d: the number of attributes with the value of 0 for both data points.

To give the distance functions, we further divide binary attribute into symmetric and asymmetric attributes. For different types of attributes, different distance functions need to be used [244]:

Symmetric attributes: A binary attribute is **symmetric** if both of its states (0 and 1) have equal importance, and carry the same weights, e.g., male and female of the attribute Gender. The most commonly used distance function for symmetric attributes is the **Simple Matching Coefficient**, which is the proportion of mismatches (Equation (11)) of their values. We assume that every attribute in the data set is a symmetric attribute.

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{b + c}{a + b + c + d} \tag{11}$$

We can also weight some components in Equation (11) according to application needs. For example, we may want mismatches to carry twice the weight of matches, or vice versa.

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{2(b + c)}{a + d + 2(b + c)} \tag{12}$$

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{b + c}{2(a + d) + b + c} \tag{13}$$

Example 11: Given the following two data points, where each attribute is a symmetric binary attribute**.**

$\mathbf{x}_1$	1	1	1	0	1	0	0
$\mathbf{x}_2$	0	1	1	0	0	1	0

The distance computed based on the simple matching coefficient is

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{2+1}{2+2+1+2} = \frac{3}{7} = 0.429 \quad (14)$$

■

Asymmetric attributes: A binary attribute is asymmetric if one of the states is more important or valuable than the other. By convention, we use state 1 to represent the more important state, which is typically the rare or infrequent state. The most commonly used distance measure for asymmetric attributes is the **Jaccard coefficient**:

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{b+c}{a+b+c} \quad (15)$$

Similarly, we can vary the Jaccard distance by giving more weight to a or more weight to $(b+c)$ to express different emphases,

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{2(b+c)}{a+2(b+c)} \quad (16)$$

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{b+c}{2a+b+c} \quad (17)$$

For general **nominal attributes** with more than two states or values, the commonly used distance measure is also based on the simple matching method. Given two data points $\mathbf{x}_i$ and $\mathbf{x}_j$, let the number of attributes be r, and the number of values that match in $\mathbf{x}_i$ and $\mathbf{x}_j$ be q:

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{r-q}{r} \quad (18)$$

As that for binary attributes, we can give higher weights to different components in Equation (18) according to different application characteristics.

4.5.3 Text Documents

Although a text document consists of a sequence of sentences and each sentence consists of a sequence of words, a document is usually considered a "bag" of words in document clustering. The sequence and the position information of words are ignored. Thus a document is represented by a vector just like a normal data point. However, we use similarity to compare two documents rather than distance. The most commonly used similarity function is the **cosine similarity**. We will study this similarity measure in Section 6.2 when we discuss information retrieval and Web search.

4.6 Data Standardization

One of the most important steps in data pre-processing for clustering is to standardize the data. For example, using the Euclidean distance, standardization of attributes is highly recommended so that all attributes can have equal impact on the distance computation. This is to avoid obtaining clusters that are dominated by attributes with the largest amounts of variation.

Example 12: In a 2-dimensional data set, the value range of one attribute is from 0 to 1, while the value range of the other attribute is from 0 to 1000. Consider the following pair of data points $\mathbf{x}_i$: (0.1, 20) and $\mathbf{x}_j$: (0.9, 720). The distance between the two points is

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{(0.9-0.1)^2 + (720-20)^2} = 700.000457, \tag{19}$$

which is almost completely dominated by (720-20) = 700. To deal with the problem, we standardize the attributes, i.e., to force the attributes to have a common value range. If both attributes are forced to have a scale within the range 0 - 1, the values 20 and 720 become 0.02 and 0.72. The distance on the first dimension becomes 0.8 and the distance on the second dimension 0.7, which are more equitable. Then, $dist(\mathbf{x}_i, \mathbf{x}_j)$ becomes 1.063. ■

This example shows that standardizing attributes is important. In fact, different types of attributes require different treatments. We list these treatments below:

Interval-scaled attributes: These are numeric/continuous attributes. Their values are real numbers following a linear scale. Examples of such attributes are age, height, weight, cost, etc. For example, the difference in Age between 10 and 20 is the same as that between 40 and 50. The key idea is that intervals keep the same importance through out the scale.

There are two main approaches to standardize interval scaled attributes, **range** and **z-score**. The range method divides each value by the range of valid values of the attribute so that the transformed value ranges between 0 and 1. Given the value x_{if} of the fth attribute of the ith data point, the new value $range(x_{if})$ is,

$$range(x_{if}) = \frac{x_{if} - \min(f)}{\max(f) - \min(f)}, \tag{20}$$

where min(f) and max(f) are the minimum value and maximum value of attribute f respectively. max(f) - min(f) is the value range of valid values of attribute f.

z-score transforms the attribute values so that they have a mean of zero and a **mean absolute deviation** of 1. The mean absolute deviation of attribute f, denoted by s_f, is computed as follows:

$$s_f = \frac{1}{n}(|x_{1f} - m_f| + |x_{2f} - m_f| + ... + |x_{nf} - m_f|), \tag{21}$$

where n is the number data points/instances in the data set, x_{if} is the value of attribute f in the ith data point, and m_f is the mean/average of attribute f, which is computed with:

$$m_f = \frac{1}{n}(x_{1f} + x_{2f} + ... + x_{nf}), \tag{22}$$

Given the value x_{if} of attribute f from data point i, its z-score (the new value after transformation) is $z(x_{if})$,

$$z(x_{if}) = \frac{x_{if} - m_f}{s_f}. \tag{23}$$

Ratio-scaled attributes: These are also numeric attributes taking real values. However, unlike interval-scaled attributes, their scales are not linear. For example, the total amount of microorganisms that evolve in a time t is approximately given by

$$Ae^{Bt},$$

where A and B are some positive constants. This formula is usually referred to as exponential growth. If we have such attributes in a data set for clustering, we have one of the following two options:

1. Treat it as an interval-scaled attribute. This is often not recommended due to scale distortion.
2. Perform logarithmic transformation to each value, x_{if}, i.e.,

$$\log(x_{if}) \tag{24}$$

   After the transformation, the attribute can be treated as an interval-scaled attribute.

Nominal (unordered categorical) attributes: As we discussed in Section 4.5.2, the value of such an attribute can take anyone of a set of states (also called categories). The states have no logical or numerical ordering. For example, the attribute *fruit* may have the possible values, Apple, Orange, and Pear, which has no ordering. A **binary attribute** is a special case of a nominal attribute with only two states or values.

Although nominal attributes are not standardized as numeric attributes, it is sometime useful to convert a nominal attribute to a set of binary attributes. Let the number of values of a nominal attribute be v. We can then create v binary attributes to represent them, i.e., one binary attribute for each value. If a data instance for the nominal attribute takes a particular value, the value of its corresponding binary attribute is set to 1, otherwise it is set to 0. The resulting binary attributes can be used as numeric attributes. We will discuss this again in Section 4.7.

Example 13: For the nominal attribute *fruit*, we create three binary attributes called, Apple, Orange, and Pear in the new data. If a particular data instance in the original data has Apple as the value for *fruit*, then in the transformed data, we set the value of the attribute Apple to 1, and the values of attributes Orange and Pear to 0. ■

Ordinal (ordered categorical) attributes: An ordinal attribute is like a nominal attribute, but its values have a numerical ordering. For example, the Age attribute may have the values, Young, Middle-Age and Old. The common approach to distance computation is to treat ordinal attributes as interval-scaled attributes and use the same methods as for interval-scaled attributes to standardize values of ordinal attributes.

4.7 Handling of Mixed Attributes

So far, we assumed that a data set contains only one type of attributes. However, in practice, a data set may contain mixed attributes. That is, it may contain any subset of the 6 types of attributes, **interval-scaled**, **symmetric binary**, **asymmetric binary**, **ratio-scaled**, **ordinal** and **nominal** attributes. Clustering a data set involving mixed attributes is a challenging problem.

One way to deal with such a data set is to choose a dominant attribute type and then convert the attributes of other types to this type. For example, if most attributes in a data set are interval-scaled, we can convert ordinal attributes and ratio-scaled attributes to interval-scaled attributes as discussed above. It is also appropriate to treat symmetric binary attributes as interval-scaled attributes. However, it does not make much sense to convert a nominal attribute with more than two values or an asymmetric binary attribute to an interval-scaled attribute, but it is still frequently done in practice by assigning some numbers to them according to some hidden ordering. For instance, in the example of Apple, Orange, and Pear, one may order them according to their prices, and thus make the attribute *fruit* an

ordinal attribute or even an interval-scaled attribute. In the previous section, we also see that a nominal attribute can be converted to a set of (symmetric) binary attributes, which in turn can be regarded as interval-scaled attributes.

Another method of handling mixed attributes is to compute the distance of each attribute of the two data points separately and then combine all the individual distances to produce an overall distance. We describe one such method, which is due to Gower [186] and is also described in [199, 244]. We describe the combination formula first (Equation (25)) and then present the methods to compute individual distances.

$$dist(\mathbf{x}_i, \mathbf{x}_j) = \frac{\sum_{f=1}^{r} \delta_{ij}^f d_{ij}^f}{\sum_{f=1}^{r} \delta_{ij}^f} \tag{25}$$

This distance value is between 0 and 1. r is the number of attributes in the data set. The indicator δ_{ij}^f is 1 if both values x_{if} and x_{jf} for attribute f are non-missing, and it is set to 0 otherwise. It is also set to 0 if attribute f is asymmetric and the match is 0-0. Equation (25) cannot be computed if all δ_{ij}^f's are 0. In such a case, some default value may be used or one of the data points is removed.

d_{ij}^f is the distance contributed by attribute f, and it is in the 0-1 range. If f is a binary or nominal attribute,

$$d_{ij}^f = \begin{cases} 1 & if\ x_{if} \neq x_{jf} \\ 0 & otherwise \end{cases} \tag{26}$$

If all the attributes are nominal, Equation (25) reduces to Equation (18). The same is true for symmetric binary attributes, in which we recover the simple matching coefficient (Equation 11). When the attributes are all asymmetric, we obtain the Jaccard coefficient (Equation (15)).

If attribute k is interval-scaled, we use

$$d_{ij}^f = \frac{| x_{if} - x_{jf} |}{R_f} \tag{27}$$

where R_f is the value range of attribute f, which is

$$R_f = \max(f) - \min(f) \tag{28}$$

Ordinal attributes and ratio-scaled attributes are handled in the same way after conversion.

If all the attributes are interval-scaled, Equation (25) becomes the Manhattan distance assuming that all attribute values are standardized by dividing their values with the ranges of their corresponding attributes.

4.8 Which Clustering Algorithm to Use?

Clustering research and application has a long history. Over the years, a vast collection of clustering algorithms has been designed. This chapter only introduced several main algorithms. Many other algorithms are variations and extensions of these algorithms.

Given an application data set, choosing the "best" clustering algorithm to cluster the data is a challenge. Every clustering algorithm has limitations and works well with only certain data distributions. However, it is very hard, if not impossible, to know what distribution the application data follows. Worse still, the application data set may not fully follow any "ideal" structure or distribution required by the algorithms. Apart from choosing a suitable clustering algorithm from a large collection of algorithms, deciding how to standardize the data, to choose a suitable distance function and to select other parameter values (e.g., k in the k-means algorithm) are complex as well. Due to these complexities, the common practice is to run several algorithms using different distance functions and parameter settings, and then carefully analyze and compare the results.

The interpretation of the results must be based on insight into the meaning of the original data together with knowledge of the algorithms used. Thus, it is crucial that the user of a clustering algorithm fully understands the algorithm and its limitations. He/she also needs to know the data well and has the domain expertise to interpret the clustering results. In many cases, generating cluster descriptions using a supervised learning method (e.g., decision tree induction) can be particularly helpful to the analysis and comparison.

4.9 Cluster Evaluation

After a set of clusters is found, we need to assess the goodness of the clusters. Unlike classification, where it is easy to measure accuracy using labeled test data, for clustering nobody knows what the correct clusters are given a data set. Thus, the quality of a clustering is much harder to evaluate. We introduce a few commonly used evaluation methods below.

User inspection: A panel of experts is asked to inspect the resulting clusters and to score them. Since this process is subjective, we take the average of the scores from all the experts as the final score of the clustering. This manual inspection is obviously a labor intensive and time consuming task. It is subjective as well. However, in most applications, some level of manual inspection is necessary because no other existing evaluation methods are able to guarantee the quality of the final clusters. It should be noted that direct user inspection may be easy for certain types of data, but not for others. For example, user inspection is not hard for text documents because one can read them easily. However, for a relational table with only numbers, staring at the data instances in each cluster makes no sense. The user can only meaningfully study the centroids of the clusters. Alternatively, the clusters can be represented as rules or a decision tree through supervised learning before giving to the user for inspection (see Section 4.3.1).

Ground truth: In this method, classification data sets are used to evaluate clustering algorithms. Recall that a classification data set has several classes, and each data instance/point is labeled with one class. Using such a data set for cluster evaluation, we make the assumption that each class corresponds to a cluster. For example, if a data set has 3 classes, we assume that it has three clusters, and we request the clustering algorithm to also produce three clusters. After clustering, we compare the cluster memberships with the class memberships to determine how good the clustering is. A variety of measures can be used to assess the clustering quality, e.g., entropy, purity, precision, recall, and F-score.

To facilitate evaluation, a confusion matrix can be constructed from the resulting clusters. From the matrix, various measurements are computed. Let the classes in the data set D be $C = (c_1, c_2, \ldots, c_k)$. The clustering method produces k clusters, which partition D into k disjoint subsets, D_1, $D_2, \ldots, D_k$.

Entropy: For each cluster, we can measure its entropy as follows:

$$entropy(D_i) = -\sum_{j=1}^{k} \Pr_i(c_j) \log_2 \Pr_i(c_j), \tag{29}$$

where $\Pr_i(c_j)$ is the proportion of class c_j data points in cluster i or D_i. The total entropy of the whole clustering (which considers all clusters) is

$$entropy_{total}(D) = \sum_{i=1}^{k} \frac{|D_i|}{|D|} \times entropy(D_i) \tag{30}$$

Purity: This again measures the extent that a cluster contains only one

class of data. The purity of each cluster is computed with

$$purity(D_i) = \max_j(\Pr_i(c_j)) \tag{31}$$

The total purity of the whole clustering (considering all clusters) is

$$purity_{total}(D) = \sum_{i=1}^{k} \frac{|D_i|}{|D|} \times purity(D_i) \tag{32}$$

Precision, recall, and F-score can be computed as well for each cluster based on the class that is the most frequent in the cluster. Note that these measures are based on a single class (see Section 3.3.2).

Example 14: Assume we have a text collection *D* of 900 documents from three topics (or three classes), Science, Sports, and Politics. Each class has 300 documents, and each document is labeled with one of the topics (classes). We use this collection to perform clustering to find three clusters. Class/topic labels are not used in clustering. After clustering, we want to measure the effectiveness of the clustering algorithm.

First, a confusion matrix (Fig. 16) is constructed based on the clustering results. From Fig. 16, we see that cluster 1 has 250 Science documents, 20 Sports documents, and 10 Politics documents. The entries of the other rows have similar meanings. The last two columns list the entropy and purity values of each cluster and also the total entropy and purity of the whole clustering (last row). We observe that cluster 1, which contains mainly Science documents, is a much better (or purer) cluster than the other two. This fact is also reflected by both their entropy and purity values.

Cluster	Science	Sports	Politics	Entropy	Purity
1	250	20	10	0.589	0.893
2	20	180	80	1.198	0.643
3	30	100	210	1.257	0.617
Total	300	300	300	1.031	0.711

Fig. 16. The confusion matrix with entropy and purity values

Obviously, we can use the total entropy or the total purity to compare different clustering results from the same algorithm with different parameter settings or from different algorithms.

Precision and recall may be computed similarly for each cluster. For example, the precision of Science documents in cluster 1 is 0.89. The recall of Science documents in cluster 1 is 0.83. F-score for Science documents in cluster 1 is thus 0.86. ■

A final remark about this evaluation method is that it is commonly used to compare different clustering algorithms based on some labeled data sets. However, a real-life data set for clustering has no class labels. Thus although an algorithm may perform very well on some labeled data sets (which may not even be in the same domain as the data that the user is working on), there is no guarantee that it will perform well on the actual application data at hand. The fact that it performs well on some label data sets does give us some confidence on the quality of the algorithm. This evaluation method is said to be based on **external data** or information.

There are methods that evaluate clusters based on the **internal information** in the clusters (without using external data with class labels). These methods measure **intra-cluster cohesion** (compactness) and **inter-cluster separation** (isolation). Cohesion measures how near the data points in a cluster are to the cluster centroid. Sum of squared error (SSE) is a commonly used measure. Separation means that different cluster centroids should be far away from one another. We need to note that good values for these measurements do not always mean good clusters, and that in most applications, expert judgments are still the key. Clustering evaluation remains to be a very difficult problem.

Indirect evaluation: In some applications, clustering is not the primary task. Instead, it is used to help perform another more important task. Then, we can use the performance on the primary task to determine which clustering method is the best for the task. For instance, in a Web usage mining application, the primary task is to provide recommendations on book purchasing to online shoppers. If shoppers can be clustered according to their profiles and their past purchasing history, we may be able to provide better recommendations. A few clustering methods can be tried, and their clustering results are then evaluated based on how well they help with the recommendation task. Of course, here we assume that the recommendation results can be reliably evaluated.

4.10 Discovering Holes and Data Regions

In this section, we wander a little to discuss something related but quite different from the preceding algorithms. We show that unsupervised learning tasks may be performed by using supervised learning techniques [311].

In clustering, data points are grouped into clusters according to their distances (or similarities). However, clusters only represent one aspect of the hidden knowledge in data. Another aspect that we have not studied is the **holes**. If we treat data instances as points in an r-dimensional space, a hole

is simply a region in the space that contains no or few data points. The existence of holes is due to the following two reasons:

1. insufficient data in certain areas, and/or
2. certain attribute-value combinations are not possible or seldom occur.

Although clusters are important, holes in the space can be quite useful as well. For example, in a disease database we may find that certain symptoms and/or test values do not occur together, or when a certain medicine is used, some test values never go beyond certain ranges. Discovery of such information can be of great importance in medical domains because it could mean the discovery of a cure to a disease or some biological laws.

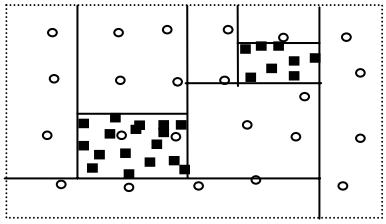
The technique discussed in this section aims to divide the data space into two types of regions, **data regions** (also called **dense regions**) and **empty regions** (also called **sparse regions**). A data region is an area in the space that contains a concentration of data points and can be regarded as a cluster. An empty region is a hole. A supervised learning technique similar to **decision tree induction** is used to separate the two types of regions. The algorithm (called CLTree) works for numeric attributes, but can be easily extended to discrete or categorical attributes.

Decision tree learning is a popular technique for classifying data of various classes. For a decision tree algorithm to work, we need at least two classes of data. A clustering data set, however, has no class label for each data point. Thus, the technique is not directly applicable. However, the problem can be dealt with by a simple idea.

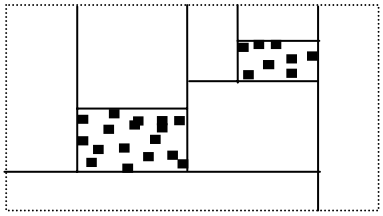
We can regard each data instance/point in the data set to have a class label *Y*. We assume that the data space is uniformly distributed with another type of points, called **non-existing points**, which we will label *N*. With the *N* points added to the original data space, our problem of partitioning the data space into data regions and empty regions becomes a supervised classification problem. The decision tree algorithm can be applied to solve the problem. However, for the technique to work several issues need to be addressed. Let us use an example to illustrate the idea.

Example 15: Fig. 17(A) gives a 2-dimensional space with 24 data (*Y*) points. Two data regions (clusters) exist in the space. We then add some uniformly distributed *N* points (represented by "o") to the data space (Fig. 17(B)). With the augmented data set, we can run a decision tree algorithm to obtain the partitioning of the space in Fig. 17(B). Data regions and empty regions are separated. Each region is a rectangle, which can be expressed as a rule.

(A): The original data space



(B). Partitioning with added *N* points



(C). Partitioning without adding *N* points.

Fig. 17. Separating data and empty regions using a decision tree ■

The reason that this technique works is that if there are clusters (or dense data regions) in the data, the data points cannot be uniformly distributed in the entire space. By adding some uniformly distributed *N* points, we can isolate data regions because within each data region there are significantly more *Y* points than *N* points. The decision tree technique is well known for this partitioning task.

An interesting question is: can the task be performed without physically adding the *N* points to the original data? The answer is yes. Physically adding *N* points increases the size of the data and thus the running time. A more important issue is that it is unlikely that we can have points truly uniformly distributed in a high dimensional space as we would need an exponential number of them. Fortunately, we do not need to physically add any *N* points. We can compute them when needed. The CLTree method is able to produce the partitioning in Fig. 17(C) with no *N* point added. This method has some interesting characteristics:

- It provides descriptions or representations of the resulting data regions and empty regions in terms of hyper-rectangles, which are expressed as rules as we show in Section 3.2 of Chapter 3. Many applications require such descriptions that can be interpreted by users.
- It detects outliers, which are data points in an empty region, automatically. The algorithm can separate outliers from clusters because it natu-

rally identifies empty and data regions.

- It may not use all attributes in the data just as in a normal decision tree for supervised learning. Thus, it can automatically determine what attributes are important, which means that it performs sub-space clustering, i.e., a cluster is only represented by a subset of the attributes, or a cluster only appear in a subspace (see Bibliographic notes on other related work in the area).

This method also has limitations. The main limitation is that data regions of irregular shapes are hard to handle since decision tree learning only generates hyper-rectangles (formed by axis-parallel hyper-planes), which are rules. Hence, an irregularly shaped data or empty region may be split into several hyper-rectangles. Post-processing is needed to join them if desired (see [311] for additional details).

Bibliographic Notes

Clustering or unsupervised learning has a long history and a very large body of work. This chapter described only some widely used core algorithms. Most other algorithms are variations or extensions of these methods. For a comprehensive coverage of clustering, please refer to several books dedicated to clustering, e.g., those by Everitt [153], Hartigan [203], Jain and Dubes [229], Kaufman and Rousseeuw [244], Mirkin [342]. Most data mining texts also have excellent coverage of clustering techniques, e.g., Han and Kamber [199] and Tan et al. [454], which have influenced the writing of this chapter. Below, we review some more recent developments and further readings.

A density-based clustering algorithm based on local data densities was proposed by Ester et al [150] and Xu et al. [499] to find clusters of arbitrary shapes. Hinneburg and Keim [218], Sheikholeslami et al [434] and Wang et al. [478] proposed several grid-based clustering methods which first partition the space into small grids. A popular Neural Networks clustering algorithm is the Self-Organizing Maps (SOMs) by Kohonen [257]. Fuzzy clustering (e.g., fuzzy c-means) was studied by Bezdek [46] and Dunn [144]. Cheeseman et al. [83] and Moore [354] studied clustering using mixture models. The method assumes that clusters are a mixture of Gaussians and uses the EM algorithm [115] to learn a mixture density. We will see in Chapter 4 that EM based partial-supervised learning algorithms are basically clustering methods with some given initial seeds.

Most of clustering algorithms work on numeric data. Categorical data and/or transaction data clustering were investigated by Barbará et al. [34],

Ganti et al. [175], Gibson et al. [178], Guha et al. [193], Wang et al [477], etc. A related area is the conceptual clustering in Artificial Intelligence, which were studied by Fisher [162], Mishra et al. [343] and many others.

Many clustering algorithms, e.g., hierarchical clustering algorithms, have high time complexities and are thus not suitable for large datasets. Scaling up such algorithms becomes an important issue for large applications. Several researchers have designed techniques to scale up clustering algorithms, e.g., Bradley et al. [57], Guha et al [192], Ng and Han [362], and Zhang et al. [535].

In recent years, there were quite a few new developments in clustering. The first one is **subspace clustering**. Traditional clustering algorithms use the whole space to find clusters, but natural clusters may only exist in some sub-spaces. That is, some clusters may only use certain subsets of attributes. This problem was investigated by Agrawal et al. [8], Aggarwal et al. [4], Aggarwal and Yu [5], Cheng et al. [84], Liu et al. [311], Zaki et al [521], etc.

The second new research is **semi-supervised clustering**, which means that the user can provide some initial information to guide the clustering process. For example, the user can select some initial seeds [36] and/or specify some constraints, e.g., **must-link** (two points must be in the same cluster) and **cannot-link** (two points cannot be in the same cluster) [469].

The third is the **spectral clustering**, which emerges from several fields, e.g., in VLSI [16] and computer vision [431, 436, 482]. It clusters data points by computing eigenvectors of the similarity matrix. Recently, it was also studied in machine learning and data mining [128, 363, 525].

Yet another new research is **co-clustering**, which simultaneously clusters both rows and columns. This approach was studied by Cheng and Church [87], Dhillon [121], Dhillon et al. [122], and Hartigan [204].

Regarding document and Web page clustering, most implementations are still based on *k*-means and hierarchical clustering methods, or their variations but using text specific similarity or distance functions. Steinbach et al. [451], and Zhao and Karypis [540, 539] experimented with *k*-means and agglomerative hierarchical clustering methods and also proposed some improvements. Many researchers also worked on **clustering of search engine results** (or snippets) to organize search results into different topics, e.g., Hearst and Pedersen [212], Kummamuru et al. [262], Leouski and Croft [277], Zamir and Etzioni [522, 523], and Zeng et al. [524].