3 Supervised Learning

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

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

3.1 Basic Concepts

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

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

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

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

Table 3.1. A loan application data set

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

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

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

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

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

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

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

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

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

We pause here to raises two important questions:

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

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

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

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

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

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

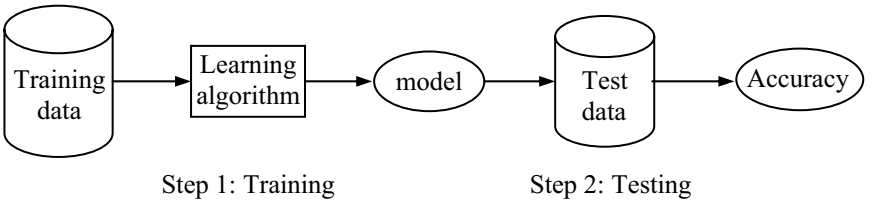


Fig. 3.1. The basic learning process: training and testing

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

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

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

3.2 Decision Tree Induction

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

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

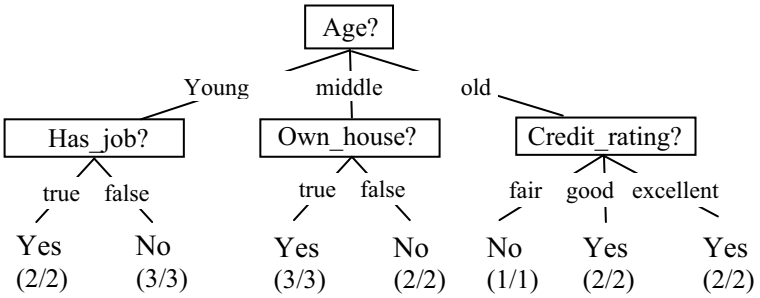


Fig. 3.2. A decision tree for the data in Table 3.1

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

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

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

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

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

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

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

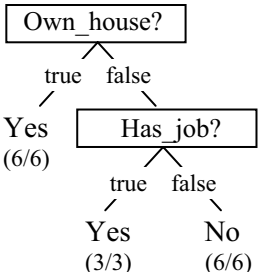


Fig. 3.3. A smaller tree for the data set in Table 3.1

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 3.4. A decision tree learning algorithm

3.2.1 Learning Algorithm

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

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

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

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

3.2.2 Impurity Function

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

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

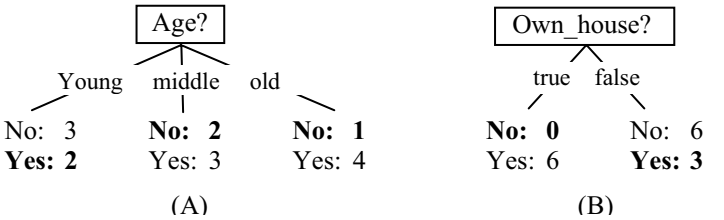


Fig. 3.5. Two possible root nodes or two possible attributes for the root node

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Information Gain

The idea is the following:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Information Gain Ratio

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

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

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

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

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

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

3.2.3 Handling of Continuous Attributes

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

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

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

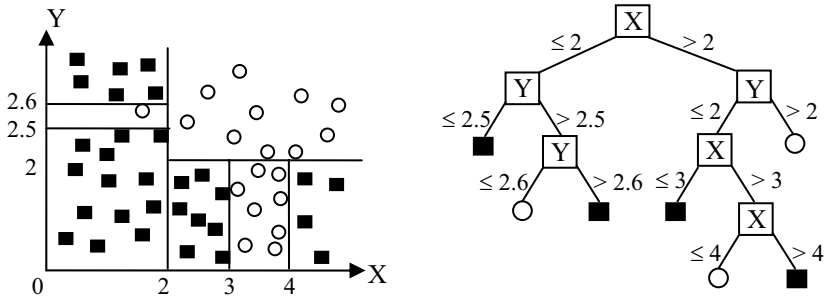
The threshold value can be the middle point between v_i and v_{i+1}, or just on the "right side" of value v_i, which results in two intervals $A_i \leq v_i$ and $A_i > v_i$. This latter approach is used in C4.5. The advantage of this approach is that the values appearing in the tree actually occur in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

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

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

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

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



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

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

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

3.2.4 Some Other Issues

We now discuss several other issues in decision tree learning.

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

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

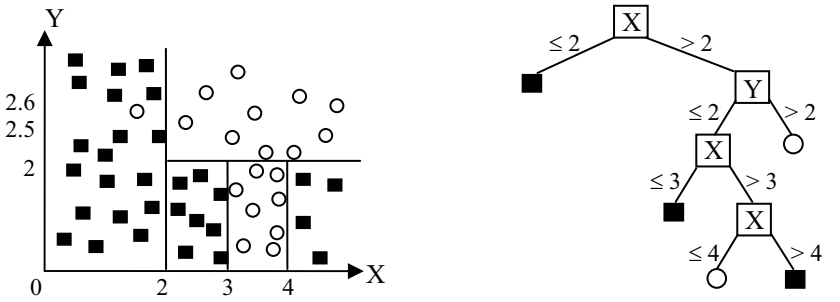
Overfitting is usually caused by noise in the data, i.e., wrong class values/labels and/or wrong values of attributes, but it may also be due to the complexity and randomness of the application domain. These problems cause the decision tree algorithm to refine the tree by extending it to very deep using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

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

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

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

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



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

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

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

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

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

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

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

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

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

$X \leq 2 \rightarrow$ ■

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

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

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

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

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

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

3.3 Classifier Evaluation

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

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

3.3.1 Evaluation Methods

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

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

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

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

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

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

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

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

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

3.3.2 Precision, Recall, F-score and Breakeven Point

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

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

Table 3.2. Confusion matrix of a classifier

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

where

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

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

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

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

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

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

Table 3.3. Confusion matrix of a classifier

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

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

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

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

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

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

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

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

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

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

Assume that the test set has 10 positive examples.

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

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

3.4 Rule Induction

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

3.4.1 Sequential Covering

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

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

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

Algorithm 1 (*Ordered Rules*)

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

This algorithm is given in Fig. 3.8. D is the training data. *RuleList* is the list of rules, which is initialized to empty set (line 1). *Rule* is the best rule found in each iteration. The function learn-one-rule-1() learns the *Rule* (lines 2 and 6). The stopping criteria for the while-loop can be of various kinds. Here we use $D = \varnothing$ 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 rules for each class is unimportant, but the rule subsets for different classes are ordered. Typically, the algorithm finds rules for the least frequent class first, then the second least frequent class and so on. This ensures that some rules are learned for rare classes. Otherwise, they may be dominated by frequent classes and end up with no rules if considered after frequent classes.

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

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

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

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

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

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

Use of Rules for Classification

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

3.4.2 Rule Learning: Learn-One-Rule Function

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

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

Learn-One-Rule-1

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

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

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

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

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

Fig. 3.11. The entropy based evaluation function

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

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

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

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

Learn-One-Rule-2

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

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

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

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

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

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

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

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

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

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

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

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

3.5.1 Classification Using Class Association Rules

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

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

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

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

$$\text{Own\_house} = \text{true} \rightarrow \text{Class} = \text{Yes} \quad [\text{sup}=6/15, \text{conf}=6/6]$$
$$\text{Own\_house} = \text{false}, \text{Has\_job} = \text{true} \rightarrow \text{Class} = \text{Yes} \quad [\text{sup}=3/15, \text{conf}=3/3]$$
$$\text{Own\_house} = \text{false}, \text{Has\_job} = \text{false} \rightarrow \text{Class} = \text{No} \quad [\text{sup}=6/15, \text{conf}=6/6].$$

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

$$\text{Age} = \text{young}, \text{Has\_job} = \text{true} \rightarrow \text{Class} = \text{Yes} \quad [\text{sup}=2/15, \text{conf}=2/2]$$
$$\text{Age} = \text{young}, \text{Has\_job} = \text{false} \rightarrow \text{Class} = \text{No} \quad [\text{sup}=3/15, \text{conf}=3/3]$$
$$\text{Credit\_rating} = \text{fair} \rightarrow \text{Class} = \text{No} \quad [\text{sup}=4/15, \text{conf}=4/5]$$

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

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

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

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

Mining Class Association Rules for Classification

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

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

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

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

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

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

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

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

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

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

Classifier Building

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

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

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

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

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

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

A CBA classifier L is of the form:

$$L = \langle r_1, r_2, \ldots, r_k, \textit{default-class} \rangle$$

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

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

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

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

Fig. 3.13. A simple classifier building algorithm

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

3.5.2 Class Association Rules as Features

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

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

3.5.3 Classification Using Normal Association Rules

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

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

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

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

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

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

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

3.6 Naïve Bayesian Classification

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 3.14. An example of a training data set

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

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

Now we have a test example:

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

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

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

For class $C = f$, we have

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

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

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

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

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

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

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

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

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

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

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

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

3.7 Naïve Bayesian Text Classification

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

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

3.7.1 Probabilistic Framework

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

The generative model is based on two assumptions:

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

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

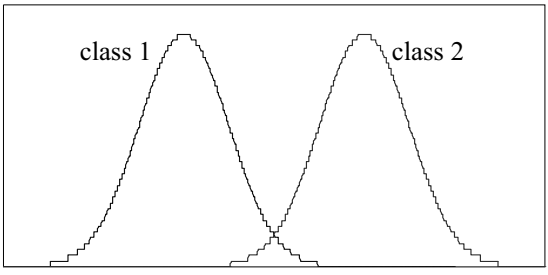


Fig. 3.15. Probability density functions of the two distributions in the mixture model

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

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

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

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

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

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

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

3.7.2 Naïve Bayesian Model

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.7.3 Discussion

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

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

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

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

3.8 Support Vector Machines

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

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

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

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

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

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

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

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

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

$$f(x_1, x_2, \ldots, x_r) = w_1 x_1 + w_2 x_2 + \ldots + w_r x_r + b,$$

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

In essence, SVM finds a hyperplane

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

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

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

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

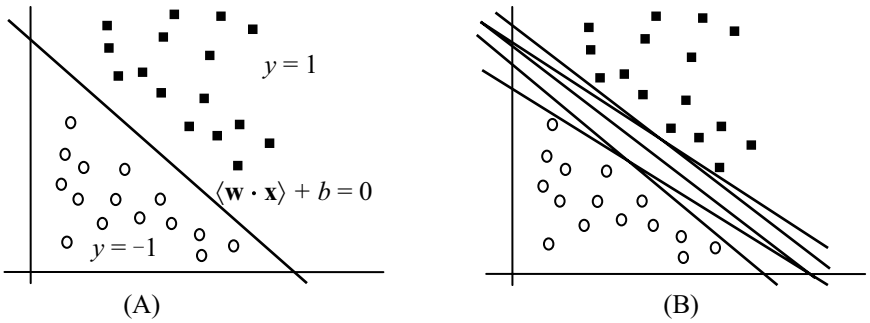


Fig. 3.16. (A) A linearly separable data set and (B) possible decision boundaries

Fig. 3.16(A) raises two interesting questions:

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

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

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

3.8.1 Linear SVM: Separable Case

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

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

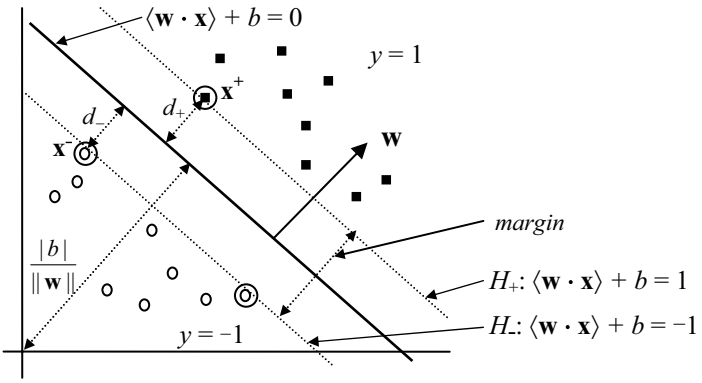


Fig. 3.17. Separating hyperplanes and margin of SVM: Support vectors are circled

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

learning is to solve the following constrained minimization problem,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and (49), which is

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

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

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

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

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

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

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

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

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

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

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

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

3.8.2 Linear SVM: Non-separable Case

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

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

Recall that the primal for the linear separable case was:

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

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

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

Thus we have the new constraints:

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

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

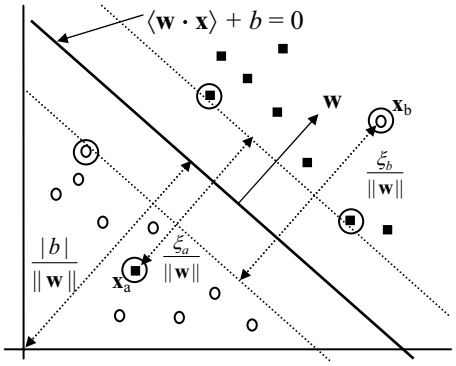


Fig. 3.18. The non-separable case: $\mathbf{x}_a$ and $\mathbf{x}_b$ are error data points

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

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

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

The new optimization problem becomes:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Similar to support vectors for the separable case, (74) shows one of the most important properties of SVM: the solution is sparse in α_i. Most training data points are outside the margin area and their α_i's in the solution are 0. Only those data points that are on the margin (i.e., $y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) = 1$, which are support vectors in the separable case), inside the margin (i.e., α_i

= C and $y_i(\langle \mathbf{w} \cdot \mathbf{x}_i \rangle + b) < 1$), or errors are non-zero. Without this sparsity property, SVM would not be practical for large data sets.

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

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

The decision rule for classification (testing) is the same as the separable case, i.e., $sign(\langle \mathbf{w} \cdot \mathbf{x} \rangle + b)$. We notice that for both Equations (75) and (73), $\mathbf{w}$ does not need to be explicitly computed. This is crucial for using kernel functions to handle nonlinear decision boundaries.

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

3.8.3 Nonlinear SVM: Kernel Functions

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

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

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

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

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

The same linear SVM solution method is then applied to F. Figure 3.19 illustrates the process. In the input space (figure on the left), the training examples cannot be linearly separated. In the transformed feature space (figure on the right), they can be separated linearly.

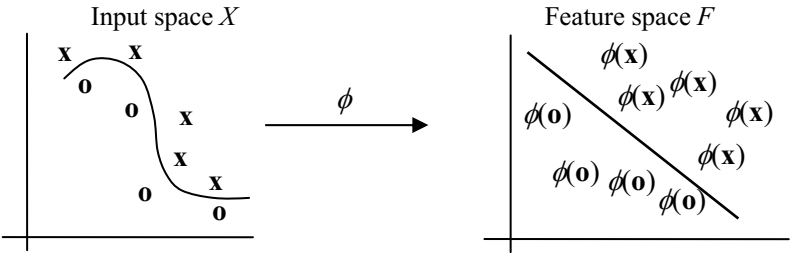


Fig. 3.19. Transformation from the input space to the feature space

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

$$\begin{aligned} &\text{Minimize}: \ \frac{\langle \mathbf{w} \cdot \mathbf{w} \rangle}{2} + C\sum_{i=1}^{n} \xi_i \\ &\text{Subject to}: \ y_i(\langle \mathbf{w} \cdot \phi(\mathbf{x}_i) \rangle + b) \geq 1 - \xi_i, \quad i = 1, 2, ..., n \\ &\qquad\qquad \xi_i \geq 0, \quad i = 1, 2, ..., n \end{aligned} \tag{78}$$

Its corresponding dual is

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

$$\begin{aligned} &\text{Subject to:} \ \sum_{i=1}^{n} y_i \alpha_i = 0 \\ &\qquad\qquad 0 \leq \alpha_i \leq C, \quad i = 1, 2, ..., n. \end{aligned}$$

The final decision rule for classification (testing) is

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

Example 16: Suppose our input space is 2-dimensional, and we choose the following transformation (mapping):

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

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

$$((4, 9, 8.5), -1).$$ ■

The potential problem with this approach of transforming the input data explicitly to a feature space and then applying the linear SVM is that it

may suffer from the curse of dimensionality. The number of dimensions in the feature space can be huge with some useful transformations (see below) even with reasonable numbers of attributes in the input space. This makes it computationally infeasible to handle.

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

Thus, if we have a way to compute the dot product $\langle \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \rangle$ in the feature space F using the input vectors $\mathbf{x}$ and $\mathbf{z}$ directly, then we would not need to know the feature vector $\phi(\mathbf{x})$ or even the mapping function ϕ itself. In SVM, this is done through the use of **kernel functions**, denoted by K,

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

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

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

Example 17: Let us compute this kernel with degree $d = 2$ in a 2-dimensional space. Let $\mathbf{x} = (x_1, x_2)$ and $\mathbf{z} = (z_1, z_2)$.

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

where $\phi(\mathbf{x}) = ({x_1}^2, {x_2}^2, \sqrt{2} x_1 x_2)$, which shows that the kernel $\langle \mathbf{x} \cdot \mathbf{z} \rangle^2$ is a dot product in the transformed feature space. The number of dimensions in the feature space is 3. Note that $\phi(\mathbf{x})$ is actually the mapping function used in Example 16. Incidentally, in general the number of dimensions in the feature space for the polynomial kernel function $K(\mathbf{x}, \mathbf{z}) = \langle \mathbf{x} \cdot \mathbf{z} \rangle^d$ is $\binom{r+d-1}{d}$, which is a huge number even with a reasonable number (r) of attributes in the input space. Fortunately, by using the kernel function in (83), the huge number of dimensions in the feature space does not matter. ■

The derivation in (84) is only for illustration purposes. We do not need to find the mapping function. We can simply apply the kernel function di-

rectly. That is, we replace all the dot products $\langle \phi(\mathbf{x}) \cdot \phi(\mathbf{z}) \rangle$ in (79) and (80) with the kernel function $K(\mathbf{x}, \mathbf{z})$ (e.g., the polynomial kernel in (83)). This strategy of directly using a kernel function to replace dot products in the feature space is called the **kernel trick**. We would never need to explicitly know what ϕ is.

However, the question is, how do we know whether a function is a kernel without performing the derivation such as that in (84)? That is, how do we know that a kernel function is indeed a dot product in some feature space? This question is answered by a theorem called the **Mercer's theorem**, which we will not discuss here. See [118] for details.

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

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

Commonly used kernels include

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

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

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

Summary

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

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

SVM also has some limitations:

1. It works only in real-valued space. For a categorical attribute, we need to convert its categorical values to numeric values. One way to do this is to create an extra binary attribute for each categorical value, and set the attribute value to 1 if the categorical value appears, and 0 otherwise.

2. It allows only two classes, i.e., binary classification. For multiple class classification problems, several strategies can be applied, e.g., one-against-rest, and error-correcting output coding [138].
3. The hyperplane produced by SVM is hard to understand by users. It is difficult to picture where the hyperplane is in a high-dimensional space. The matter is made worse by kernels. Thus, SVM is commonly used in applications that do not required human understanding.

3.9 K-Nearest Neighbor Learning

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

It works as follows: Again let D be the training data set. Nothing will be done on the training examples. When a test instance d is presented, the algorithm compares d with every training example in D to compute the similarity or distance between them. The k most similar (closest) examples in D are then selected. This set of examples is called the ***k* nearest neighbors** of d. d then takes the most frequent class among the k nearest neighbors. Note that $k = 1$ is usually not sufficient for determining the class of d due to noise and outliers in the data. A set of nearest neighbors is needed to accurately decide the class. The general *k*NN algorithm is given in Fig. 3.20.

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

Fig. 3.20. The *k*-nearest neighbor algorithm

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

The number of nearest neighbors k is usually determined by using a validation set, or through cross validation on the training data. That is, a

range of k values are tried, and the k value that gives the best accuracy on the validation set (or cross validation) is selected. Figure 3.21 illustrates the importance of choosing the right k.

Example 18: In Fig. 3.21, we have two classes of data, positive (filled squares) and negative (empty circles). If 1-nearest neighbor is used, the test data point ⊕ will be classified as negative, and if 2-nearest neighbors are used, the class cannot be decided. If 3-nearest neighbors are used, the class is positive as two positive examples are in the 3-nearest neighbors.

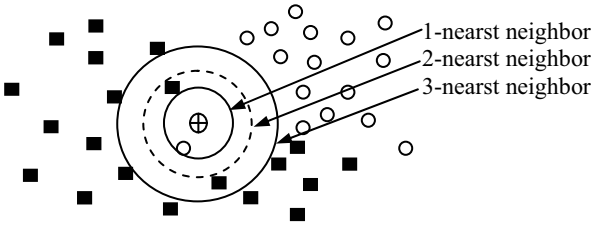


Fig. 3.21. An illustration of k-nearest neighbor classification

Despite its simplicity, researchers have showed that the classification accuracy of kNN can be quite strong and in many cases as accurate as those elaborated methods. For instance, it is showed in [574] that kNN performs equally well as SVM for some text classification tasks. kNN is also very flexible. It can work with any arbitrarily shaped decision boundaries.

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

3.10 Ensemble of Classifiers

So far, we have studied many individual classifier building techniques. A natural question to ask is: can we build many classifiers and then combine them to produce a better classifier? Yes, in many cases. This section describes two well known ensemble techniques, **bagging** and **boosting**. In both these methods, many classifiers are built and the final classification decision for each test instance is made based on some forms of voting of the committee of classifiers.

3.10.1 Bagging

Given a training set D with n examples and a base learning algorithm, bagging (for *B*ootstrap *Agg*regat*ing*) works as follows [63]:

Training:

1. Create k bootstrap samples S_1, S_2, and S_k. Each sample is produced by drawing n examples at random from D with replacement. Such a sample is called a **bootstrap replicate** of the original training set D. On average, each sample S_i contains 63.2% of the original examples in D, with some examples appearing multiple times.
2. Build a classifier based on each sample S_i. This gives us k classifiers. All the classifiers are built using the same base learning algorithm.

Testing: Classify each test (or new) instance by voting of the k classifiers (equal weights). The majority class is assigned as the class of the instance.

Bagging can improve the accuracy significantly for unstable learning algorithms, i.e., a slight change in the training data resulting in a major change in the output classifier. Decision tree and rule induction methods are examples of unstable learning methods. k-nearest neighbor and naïve Bayesian methods are examples of stable techniques. For stable classifiers, Bagging may sometime degrade the accuracy.

3.10.2 Boosting

Boosting is a family of ensemble techniques, which, like bagging, also manipulates the training examples and produces multiple classifiers to improve the classification accuracy [477]. Here we only describe the popular **AdaBoost** algorithm given in [186]. Unlike bagging, AdaBoost assigns a weight to each training example.

Training: AdaBoost produces a sequence of classifiers (also using the same base learner). Each classifier is dependent on the previous one, and focuses on the previous one's errors. Training examples that are incorrectly classified by the previous classifiers are given higher weights.

Let the original training set D be $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$, where $\mathbf{x}_i$ is an input vector, y_i is its class label and $y_i \in Y$ (the set of class labels). With a weight attached to each example, we have, $\{(\mathbf{x}_1, y_1, w_1), (\mathbf{x}_2, y_2, w_2), \ldots, (\mathbf{x}_n, y_n, w_n)\}$, and $\sum_i w_i = 1$. The AdaBoost algorithm is given in Fig. 3.22.

The algorithm builds a sequence of k classifiers (k is specified by the user) using a base learner, called BaseLeaner in line 3. Initially, the weight

AdaBoost(D, Y, BaseLeaner, k)

1. Initialize $D_1(w_i) \leftarrow 1/n$ for all i; // initialize the weights
2. **for** $t = 1$ to k **do**
3. $f_t \leftarrow$ BaseLearner(D_t); // build a new classifier f_t
4. $e_t \leftarrow \sum_{i:f_t(D_t(\mathbf{x}_i))\neq y_i} D_t(w_i)$; // compute the error of f_t
5. **if** $e_t > ½$ **then** // if the error is too large,
6. $k \leftarrow k - 1$; // remove the iteration and
7. exit-loop // exit
8. **else**
9. $\beta_t \leftarrow e_t / (1 - e_t)$;
10. $D_{t+1}(w_i) \leftarrow D_t(w_i) \times \begin{cases} \beta_t & \text{if } f_t(D_t(\mathbf{x}_i)) = y_i \\ 1 & \text{otherwise} \end{cases}$; // update the weights
11. $D_{t+1}(w_i) \leftarrow \frac{D_{t+1}(w_i)}{\sum_{i=1}^{n} D_{t+1}(w_i)}$ // normalize the weights
12. **endif**
13. **endfor**
14. $f_{final}(\mathbf{x}) \leftarrow \underset{y \in Y}{\text{argmax}} \sum_{t:f_t(\mathbf{x})=y} \log\frac{1}{\beta_t}$ // the final output classifier

Fig. 3.22. The AdaBoost algorithm

for each training example is $1/n$ (line 1). In each iteration, the training data set becomes D_t, which is the same as D, but with different weights. Each iteration builds a new classifier f_t (line 3). The error of f_t is calculated in line 4. If it is too large, delete the iteration and exit (lines 5–7). Lines 9–11 update and normalize the weights for building the next classifier.

Testing: For each test case, the results of the series of classifiers are combined to determine the final class of the test case, which is shown in line 14 of Fig. 3.22 (a weighted voting).

Boosting works better than bagging in most cases as shown in [454]. It also tends to improve performance more when the base learner is unstable.

Bibliographic Notes

Supervised learning has been studied extensively by the machine learning community. The book by Mitchell [385] covers most learning techniques and is easy to read. Duda et al.'s pattern classification book is also a great

reference [155]. Most data mining books have one or two chapters on supervised learning, e.g., those by Han and Kamber [218], Hand et al. [221], Tan et al. [512], and Witten and Frank [549].

For decision tree induction, Quinlan's book [453] has all the details and the code of his popular decision tree system C4.5. Other well-known systems include CART by Breiman et al. [62] and CHAD by Kass [270]. Scaling up of decision tree algorithms was also studied in several papers. These algorithms can have the data on disk, and are thus able to run with huge data sets. See [195] for an algorithm and also additional references.

Rule induction algorithms generate rules directly from the data. Well-known systems include AQ by Michalski et al. [381], CN2 by Clark and Niblett [104], FOIL by Quinlan [452], FOCL by Pazzani et al. [438], I-REP by Furnkranz and Widmer [189], and RIPPER by Cohen [106].

Using association rules to build classifiers was proposed by Liu et al. in [343], which also reported the CBA system. CBA selects a small subset of class association rules as the classifier. Other classifier building techniques include combining multiple rules by Li et al. [328], using rules as features by Meretakis and Wüthrich [379], Antonie and Zaiane [23], Deshpande and Karpis [131], Jindal and Liu [255], and Lesh et al. [314], generating a subset of rules by Cong et al. [112, 113], Wang et al. [536], Yin and Han [578], and Zaki and Aggarwal [587]. Other systems include those by Dong et al. [149], Li et al. [319, 320], Yang et al. [570], etc.

The naïve Bayesian classification model described in Sect. 3.6 is based on the papers by Domingos and Pazzani [148], Kohavi et al. [285] and Langley et al [301]. The naïve Bayesian classification for text discussed in Sect. 3.7 is based on the multinomial formulation given by McCallum and Nigam [365]. This model was also used earlier by Lewis and Gale [317], Li and Yamanishi [318], and Nigam et al. [413]. Another formulation of naïve Bayes is based on the multivariate Bernoulli model, which was used in Lewis [316], and Robertson and Sparck-Jones [464].

Support vector machines (SVM) was first introduced by Vapnik and his colleagues in 1992 [59]. Further details were given in his 1995 book [525]. Two other books on SVM and kernel methods are those by Cristianini and Shawe-Taylor [118] and Scholkopf and Smola [479]. The discussion of SVM in this chapter is heavily influenced by Cristianini and Shawe-Taylor's book and the tutorial paper by Burges [74]. Two popular SVM systems are SVM^{Light} (available at http://svmlight.joachims.org/) and LIBSVM (available at http://www.csie.ntu.edu.tw/~cjlin/libsvm/).

Existing classifier ensemble methods include bagging by Breiman [63], boosting by Schapire [477] and Freund and Schapire [186], random forest also by Breiman [65], stacking by Wolpert [552], random trees by Fan [169], and many others.