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

B. Liu, *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*,
Data-Centric Systems and Applications, DOI 10.1007/978-3-642-19460-3_3,

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.

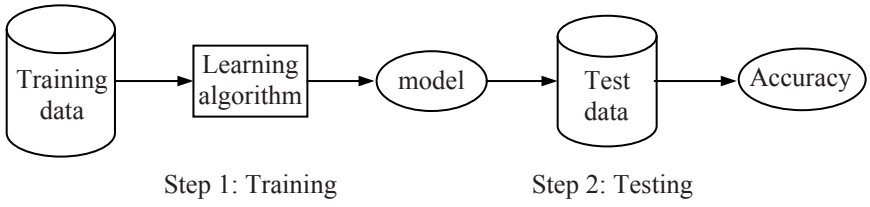

Fig. 3.1. The basic learning process: training and testing

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 [49].

Example 2: Fig. 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.

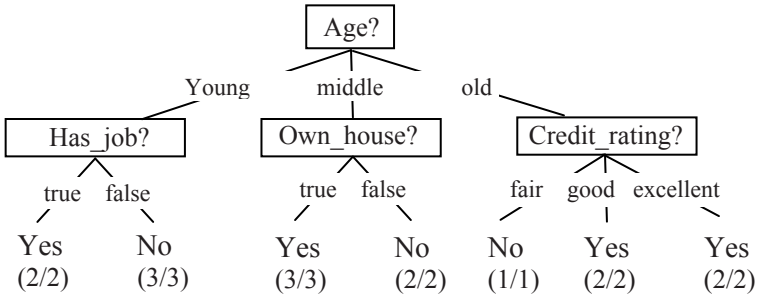

Fig. 3.2. A decision tree for the data in Table 3.1

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.

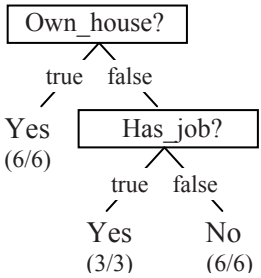

Fig. 3.3. A smaller tree for the data set in Table 3.1

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 exam-

ples 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 [26]. 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 ∈ C then
2       make T a leaf node labeled with class c_j;
3   elseif A = ∅ 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 ∈ A (={A_1, A_2, …, A_k}) do
9           p_i = impurityEval-2(A_i, D)
10      endfor
11      Select A_g ∈ {A_1, A_2, …, 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, …, v_m. Partition D into m
                disjoint subsets D_1, D_2, …, D_m based on the m values of A_g.
17          for each D_j in {D_1, D_2, …, D_m} do
18              if D_j ≠ ∅ 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: Fig. 3.5 shows two possible root nodes for the data in Table 3.1.

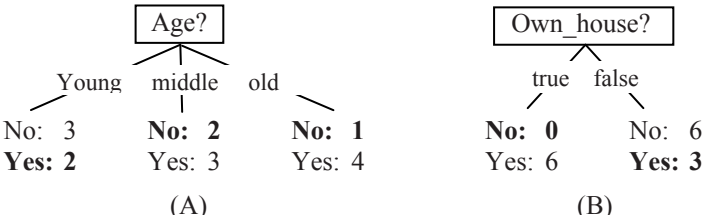

Fig. 3.5. Two possible root nodes or two possible attributes for the root node

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** [55]:

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

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

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. Fig. 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 [49]).

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