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

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

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

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

## 8.2 Decision Tree Induction

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

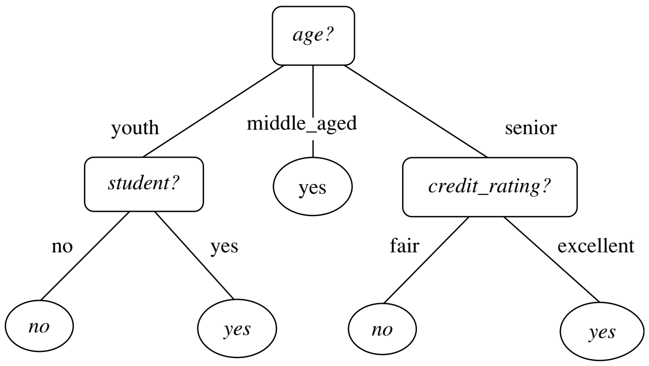


**Figure 8.2** A decision tree for the concept *buys_computer*, indicating whether an *AllElectronics* customer is likely to purchase a computer. Each internal (nonleaf) node represents a test on an attribute. Each leaf node represents a class (either *buys_computer* = *yes* or *buys_computer* = *no*).

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

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

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

In Section 8.2.1, we describe a basic algorithm for learning decision trees. During tree construction, *attribute selection measures* are used to select the attribute that best partitions the tuples into distinct classes. Popular measures of attribute selection are given in Section 8.2.2. When decision trees are built, many of the branches may reflect noise or outliers in the training data. *Tree pruning* attempts to identify and remove such branches, with the goal of improving classification accuracy on unseen data. Tree pruning is described in Section 8.2.3. Scalability issues for the induction of decision trees from large databases are discussed in Section 8.2.4. Section 8.2.5 presents a visual mining approach to decision tree induction.

## 8.2.1 Decision Tree Induction

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

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

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

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

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

**Input:**

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

**Output:** A decision tree.

**Method:**

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

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

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

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

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

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

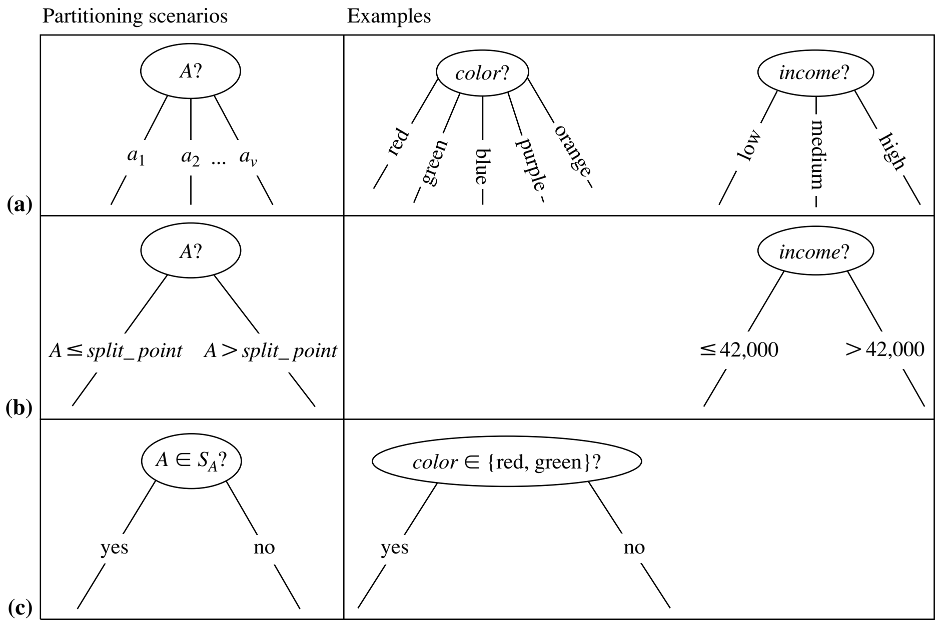


**Figure 8.4** This figure shows three possibilities for partitioning tuples based on the splitting criterion, each with examples. Let $A$ be the splitting attribute. (a) If $A$ is discrete-valued, then one branch is grown for each known value of $A$. (b) If $A$ is continuous-valued, then two branches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

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

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

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

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

  1. All the tuples in partition $D$ (represented at node $N$) belong to the same class (steps 2 and 3).
  2. There are no remaining attributes on which the tuples may be further partitioned (step 4). In this case, **majority voting** is employed (step 5). This involves converting node $N$ into a leaf and labeling it with the most common class in $D$. Alternatively, the class distribution of the node tuples may be stored.
  3. There are no tuples for a given branch, that is, a partition $D_j$ is empty (step 12). In this case, a leaf is created with the majority class in $D$ (step 13).

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

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

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

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

## 8.2.2 Attribute Selection Measures

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

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

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

### Information Gain

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

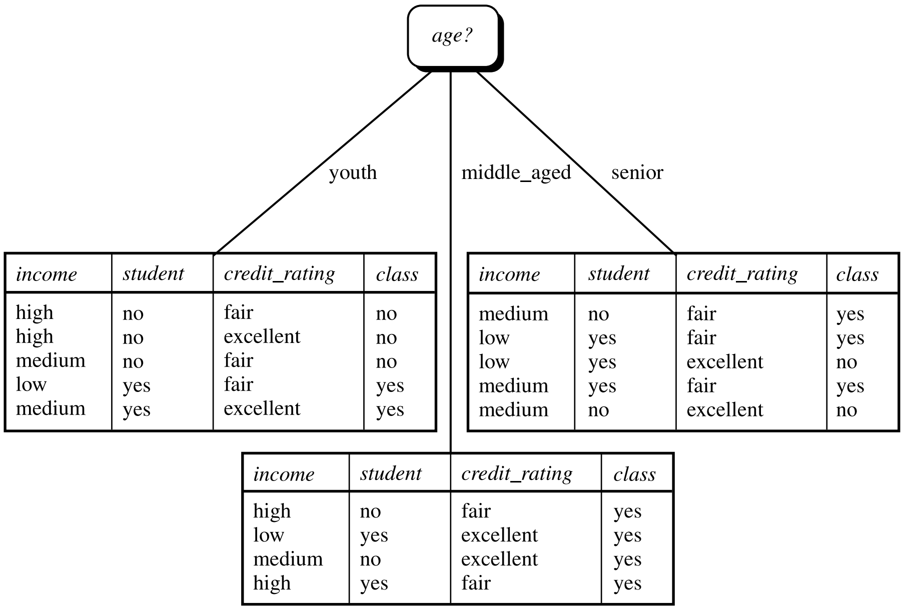
$$Info_{age}(D) = \frac{5}{14} \times \left(-\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5}\right)$$

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

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

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

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



| *income* | *student* | *credit_rating* | *class* |
|---|---|---|---|
| high | no | fair | no |
| high | no | excellent | no |
| medium | no | fair | no |
| low | yes | fair | yes |
| medium | yes | excellent | yes |

| *income* | *student* | *credit_rating* | *class* |
|---|---|---|---|
| medium | no | fair | yes |
| low | yes | fair | yes |
| low | yes | excellent | no |
| medium | yes | fair | yes |
| medium | no | excellent | no |

| *income* | *student* | *credit_rating* | *class* |
|---|---|---|---|
| high | no | fair | yes |
| low | yes | excellent | yes |
| medium | no | excellent | yes |
| high | yes | fair | yes |

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

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

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

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

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

## Gain Ratio

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

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

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

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

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

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

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

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

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

## Gini Index

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Other Attribute Selection Measures

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

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

Attribute selection measures based on the **Minimum Description Length (MDL)** principle have the least bias toward multivalued attributes. MDL-based measures use encoding techniques to define the "best" decision tree as the one that requires the fewest number of bits to both (1) encode the tree and (2) encode the exceptions to the tree (i.e., cases that are not correctly classified by the tree). Its main idea is that the simplest of solutions is preferred.

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

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

### 8.2.3 Tree Pruning

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

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

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

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

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

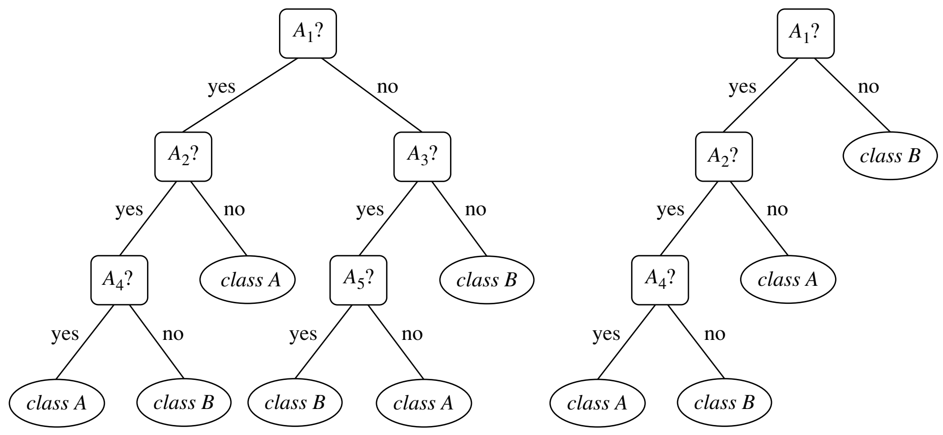


**Figure 8.6** An unpruned decision tree and a pruned version of it.

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

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

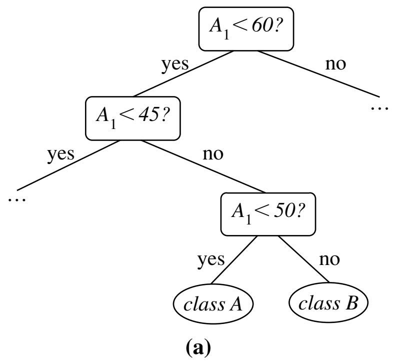
A **pruning set** of class-labeled tuples is used to estimate cost complexity. This set is independent of the training set used to build the unpruned tree and of any test set used for accuracy estimation. The algorithm generates a set of progressively pruned trees. In general, the smallest decision tree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

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

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

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

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



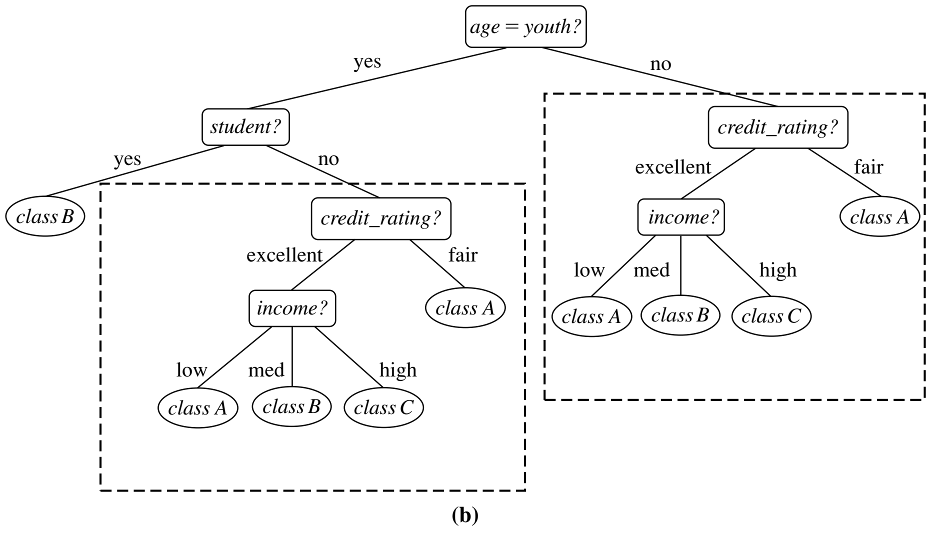


**Figure 8.7** An example of: (a) subtree **repetition**, where an attribute is repeatedly tested along a given branch of the tree (e.g., *age*) and (b) subtree **replication**, where duplicate subtrees exist within a tree (e.g., the subtree headed by the node "*credit_rating?*").

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

### 8.2.4 Scalability and Decision Tree Induction

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

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

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

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

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

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

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

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

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

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

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

### 8.2.5 Visual Mining for Decision Tree Induction

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

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

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

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

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

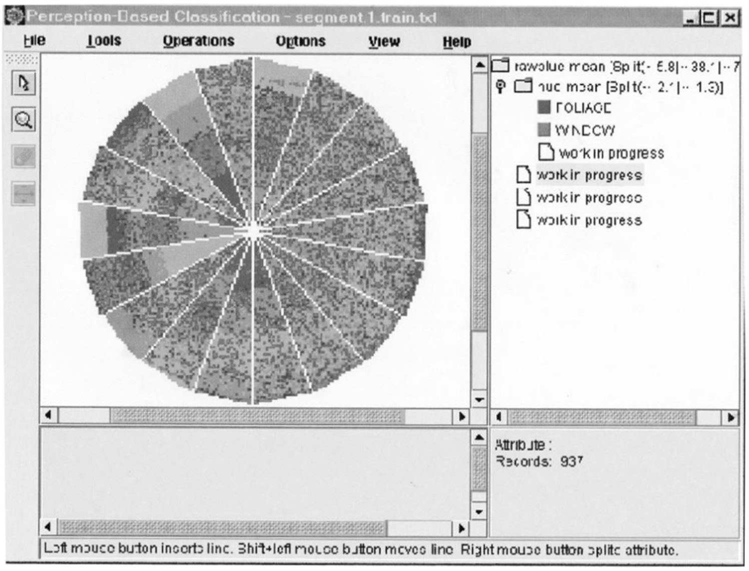

Perception-Based Classification - segment.1.train.txt
File
Tools
Operations
Options
View
Help
rawvalue mean [Split(- 6.8|- 38.'|- 7
hue mean [Split(- 2.'|- 1.3)]
FOLIAGE
WINDOW
work in progress
work in progress
work in progress
work in progress
Attribute :
Records: 937
Left mouse button inserts line. Shift+left mouse button moves line. Right mouse button splits attribute.

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

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

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

## 8.3 Bayes Classification Methods

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

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

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

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

### 8.3.1 Bayes' Theorem

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

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

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

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

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

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

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

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

## 8.3.2 Naïve Bayesian Classification

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

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

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

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

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

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

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

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

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

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

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

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

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

so that

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Using these probabilities, we obtain

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

Similarly,

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

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

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

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

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

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

There is a simple trick to avoid this problem. We can assume that our training database, $D$, is so large that adding one to each count that we need would only make a negligible difference in the estimated probability value, yet would conveniently avoid the case of probability values of zero. This technique for probability estimation is known as the **Laplacian correction** or **Laplace estimator**, named after Pierre Laplace, a French mathematician who lived from 1749 to 1827. If we have, say, $q$ counts to which we each add one, then we must remember to add $q$ to the corresponding denominator used in the probability calculation. We illustrate this technique in Example 8.5.

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

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

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

## 8.4 Rule-Based Classification

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

### 8.4.1 Using IF-THEN Rules for Classification

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

IF *condition* THEN *conclusion*.

An example is rule $R1$,

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

The "IF" part (or left side) of a rule is known as the **rule antecedent** or **precondition**. The "THEN" part (or right side) is the **rule consequent**. In the rule antecedent, the condition consists of one or more *attribute tests* (e.g., *age* = *youth* and *student* = *yes*) that are logically ANDed. The rule's consequent contains a class prediction (in this case, we are predicting whether a customer will buy a computer). $R1$ can also be written as

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 8.4.2 Rule Extraction from a Decision Tree

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

To extract rules from a decision tree, one rule is created for each path from the root to a leaf node. Each splitting criterion along a given path is logically ANDed to form the rule antecedent ("IF" part). The leaf node holds the class prediction, forming the rule consequent ("THEN" part).

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

| | | |
|---|---|---|
| $R1$: IF $age = youth$ | AND $student = no$ | THEN $buys\_computer = no$ |
| $R2$: IF $age = youth$ | AND $student = yes$ | THEN $buys\_computer = yes$ |
| $R3$: IF $age = middle\_aged$ | | THEN $buys\_computer = yes$ |
| $R4$: IF $age = senior$ | AND $credit\_rating = excellent$ | THEN $buys\_computer = yes$ |
| $R5$: IF $age = senior$ | AND $credit\_rating = fair$ | THEN $buys\_computer = no$ |

■

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

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

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

Other problems arise during rule pruning, however, as the rules *will no longer be* mutually exclusive and exhaustive. For conflict resolution, C4.5 adopts a **class-based ordering scheme**. It groups together all rules for a single class, and then determines a ranking of these class rule sets. Within a rule set, the rules are not ordered. C4.5 orders the class rule sets so as to minimize the number of *false-positive errors* (i.e., where a rule predicts a class, $C$, but the actual class is not $C$). The class rule set with the least number of false positives is examined first. Once pruning is complete, a final check is done to remove any duplicates. When choosing a default class, C4.5 does not choose the majority class, because this class will likely have many rules for its tuples. Instead, it selects the class that contains the most training tuples that were not covered by any rule.

### 8.4.3 Rule Induction Using a Sequential Covering Algorithm

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

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

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

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

**Input:**

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

**Output:** A set of IF-THEN rules.

**Method:**

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

**Figure 8.10** Basic sequential covering algorithm.

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

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

IF THEN *loan_decision* = *accept*.

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

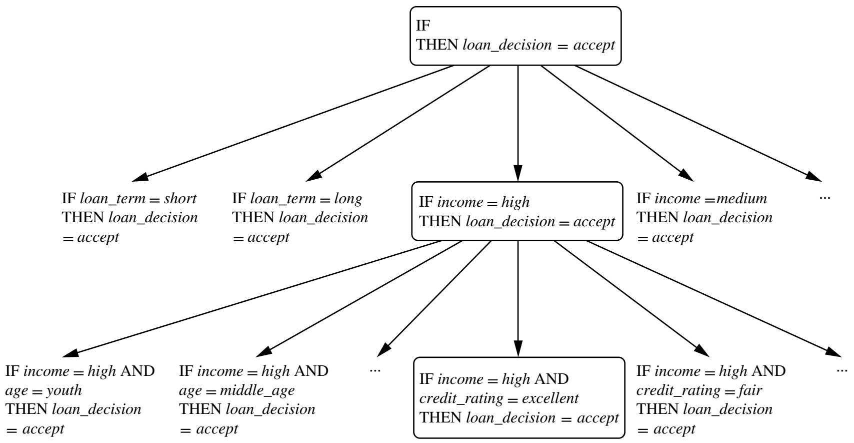

IF
THEN loan_decision = accept
IF loan_term = short THEN loan_decision = accept
IF loan_term = long THEN loan_decision = accept
IF income = high THEN loan_decision = accept
IF income = medium THEN loan_decision = accept
...
IF income = high AND age = youth THEN loan_decision = accept
IF income = high AND age = middle_age THEN loan_decision = accept
...
IF income = high AND credit_rating = excellent THEN loan_decision = accept
IF income = high AND credit_rating = fair THEN loan_decision = accept
...

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

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

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

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

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

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

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

## Rule Quality Measures

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

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

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

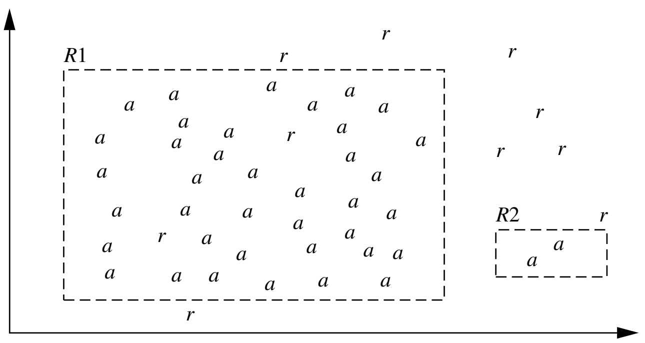


**Figure 8.12** Rules for the class *loan_decision* = *accept*, showing *accept* (*a*) and *reject* (*r*) tuples.

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

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

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

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

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

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

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

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

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

where $m$ is the number of classes.

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

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

## Rule Pruning

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

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

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

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

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

## 8.5 Model Evaluation and Selection

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

Section 8.5.1 describes various evaluation metrics for the predictive accuracy of a classifier. Holdout and random subsampling (Section 8.5.2), cross-validation (Section 8.5.3), and bootstrap methods (Section 8.5.4) are common techniques for assessing accuracy, based on randomly sampled partitions of the given data. What if we have more than one classifier and want to choose the "best" one? This is referred to as **model selection** (i.e., choosing one classifier over another). The last two sections address this issue. Section 8.5.5 discusses how to use tests of statistical significance to assess whether the difference in accuracy between two classifiers is due to chance. Section 8.5.6 presents how to compare classifiers based on cost–benefit and receiver operating characteristic (ROC) curves.

### 8.5.1 Metrics for Evaluating Classifier Performance

This section presents measures for assessing how good or how "accurate" your classifier is at predicting the class label of tuples. We will consider the case of where the class tuples are more or less evenly distributed, as well as the case where classes are unbalanced (e.g., where an important class of interest is rare such as in medical tests). The classifier evaluation measures presented in this section are summarized in Figure 8.13. They include accuracy (also known as recognition rate), sensitivity (or recall), specificity, precision, $F_1$, and $F_\beta$. Note that although accuracy is a specific measure, the word "accuracy" is also used as a general term to refer to a classifier's predictive abilities.

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

Before we discuss the various measures, we need to become comfortable with some terminology. Recall that we can talk in terms of **positive tuples** (tuples of the main class of interest) and **negative tuples** (all other tuples).[6] Given two classes, for example, the positive tuples may be *buys_computer* = *yes* while the negative tuples are

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

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

**Figure 8.13** Evaluation measures. Note that some measures are known by more than one name. $TP, TN, FP, P, N$ refer to the number of true positive, true negative, false positive, positive, and negative samples, respectively (see text).

*buys_computer* = *no*. Suppose we use our classifier on a test set of labeled tuples. $P$ is the number of positive tuples and $N$ is the number of negative tuples. For each tuple, we compare the classifier's class label prediction with the tuple's known class label.

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

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

These terms are summarized in the **confusion matrix** of Figure 8.14.

The confusion matrix is a useful tool for analyzing how well your classifier can recognize tuples of different classes. $TP$ and $TN$ tell us when the classifier is getting things right, while $FP$ and $FN$ tell us when the classifier is getting things wrong (i.e.,

| | | **Predicted class** | | |
|---|---|---|---|---|
| | | *yes* | *no* | Total |
| **Actual class** | *yes* | *TP* | *FN* | *P* |
| | *no* | *FP* | *TN* | *N* |
| | Total | *P′* | *N′* | $P+N$ |

**Figure 8.14** Confusion matrix, shown with totals for positive and negative tuples.

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

**Figure 8.15** Confusion matrix for the classes *buys_computer = yes* and *buys_computer = no,* where an entry in row *i* and column *j* shows the number of tuples of class *i* that were labeled by the classifier as class *j*. Ideally, the nondiagonal entries should be zero or close to zero.

mislabeling). Given $m$ classes (where $m \geq 2$), a **confusion matrix** is a table of at least size $m$ by $m$. An entry, $CM_{i,j}$ in the first $m$ rows and $m$ columns indicates the number of tuples of class $i$ that were labeled by the classifier as class $j$. For a classifier to have good accuracy, ideally most of the tuples would be represented along the diagonal of the confusion matrix, from entry $CM_{1,1}$ to entry $CM_{m,m}$, with the rest of the entries being zero or close to zero. That is, ideally, $FP$ and $FN$ are around zero.

The table may have additional rows or columns to provide totals. For example, in the confusion matrix of Figure 8.14, $P$ and $N$ are shown. In addition, $P'$ is the number of tuples that were labeled as positive ($TP+FP$) and $N'$ is the number of tuples that were labeled as negative ($TN+FN$). The total number of tuples is $TP+TN+FP+TN$, or $P+N$, or $P'+N'$. Note that although the confusion matrix shown is for a binary classification problem, confusion matrices can be easily drawn for multiple classes in a similar manner.

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

$$accuracy = \frac{TP+TN}{P+N}. \tag{8.21}$$

In the pattern recognition literature, this is also referred to as the overall **recognition rate** of the classifier, that is, it reflects how well the classifier recognizes tuples of the various classes. An example of a confusion matrix for the two classes *buys_computer = yes* (positive) and *buys_computer = no* (negative) is given in Figure 8.15. Totals are shown, as well as the recognition rates per class and overall. By glancing at a confusion matrix, it is easy to see if the corresponding classifier is confusing two classes.

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

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

$$error\ rate = \frac{FP + FN}{P + N}. \tag{8.22}$$

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

We now consider the **class imbalance problem**, where the main class of interest is rare. That is, the data set distribution reflects a significant majority of the negative class and a minority positive class. For example, in fraud detection applications, the class of interest (or positive class) is "*fraud*," which occurs much less frequently than the negative "*nonfraudulant*" class. In medical data, there may be a rare class, such as "*cancer*." Suppose that you have trained a classifier to classify medical data tuples, where the class label attribute is "*cancer*" and the possible class values are "*yes*" and "*no*." An accuracy rate of, say, 97% may make the classifier seem quite accurate, but what if only, say, 3% of the training tuples are actually cancer? Clearly, an accuracy rate of 97% may not be acceptable—the classifier could be correctly labeling only the noncancer tuples, for instance, and misclassifying all the cancer tuples. Instead, we need other measures, which access how well the classifier can recognize the positive tuples ($cancer = yes$) and how well it can recognize the negative tuples ($cancer = no$).

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

$$sensitivity = \frac{TP}{P} \tag{8.23}$$

$$specificity = \frac{TN}{N}. \tag{8.24}$$

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

$$accuracy = sensitivity\frac{P}{(P+N)} + specificity\frac{N}{(P+N)}. \tag{8.25}$$

**Example 8.9** **Sensitivity and specificity.** Figure 8.16 shows a confusion matrix for medical data where the class values are *yes* and *no* for a class label attribute, *cancer*. The sensitivity

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

**Figure 8.16** Confusion matrix for the classes *cancer* = *yes* and *cancer* = *no*.

of the classifier is $\frac{90}{300} = 30.00\%$. The specificity is $\frac{9560}{9700} = 98.56\%$. The classifier's overall accuracy is $\frac{9650}{10{,}000} = 96.50\%$. Thus, we note that although the classifier has a high accuracy, it's ability to correctly label the positive (rare) class is poor given its low sensitivity. It has high specificity, meaning that it can accurately recognize negative tuples. Techniques for handling class-imbalanced data are given in Section 8.6.5. ■

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

$$precision = \frac{TP}{TP + FP} \tag{8.26}$$

$$recall = \frac{TP}{TP + FN} = \frac{TP}{P}. \tag{8.27}$$

**Example 8.10** **Precision and recall.** The precision of the classifier in Figure 8.16 for the *yes* class is $\frac{90}{230} = 39.13\%$. The recall is $\frac{90}{300} = 30.00\%$, which is the same calculation for sensitivity in Example 8.9. ■

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

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

$$F = \frac{2 \times precision \times recall}{precision + recall} \tag{8.28}$$

$$F_\beta = \frac{(1+\beta^2) \times precision \times recall}{\beta^2 \times precision + recall}, \tag{8.29}$$

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

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

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

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

- **Speed:** This refers to the computational costs involved in generating and using the given classifier.
- **Robustness:** This is the ability of the classifier to make correct predictions given noisy data or data with missing values. Robustness is typically assessed with a series of synthetic data sets representing increasing degrees of noise and missing values.
- **Scalability:** This refers to the ability to construct the classifier efficiently given large amounts of data. Scalability is typically assessed with a series of data sets of increasing size.
- **Interpretability:** This refers to the level of understanding and insight that is provided by the classifier or predictor. Interpretability is subjective and therefore more difficult to assess. Decision trees and classification rules can be easy to interpret, yet their interpretability may diminish the more they become complex. We discuss some work in this area, such as the extraction of classification rules from a "black box" neural network classifier called backpropagation, in Chapter 9.

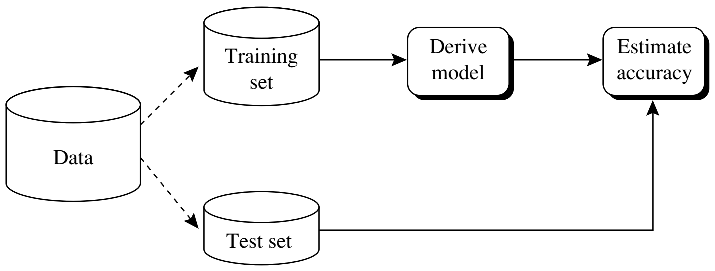


**Figure 8.17** Estimating accuracy with the holdout method.

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

## 8.5.2 Holdout Method and Random Subsampling

The **holdout** method is what we have alluded to so far in our discussions about accuracy. In this method, the given data are randomly partitioned into two independent sets, a *training set* and a *test set*. Typically, two-thirds of the data are allocated to the training set, and the remaining one-third is allocated to the test set. The training set is used to derive the model. The model's accuracy is then estimated with the test set (Figure 8.17). The estimate is pessimistic because only a portion of the initial data is used to derive the model.

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

## 8.5.3 Cross-Validation

In ***k*-fold cross-validation**, the initial data are randomly partitioned into $k$ mutually exclusive subsets or "folds," $D_1, D_2, \ldots, D_k$, each of approximately equal size. Training and testing is performed $k$ times. In iteration $i$, partition $D_i$ is reserved as the test set, and the remaining partitions are collectively used to train the model. That is, in the first iteration, subsets $D_2, \ldots, D_k$ collectively serve as the training set to obtain a first model, which is tested on $D_1$; the second iteration is trained on subsets $D_1, D_3, \ldots, D_k$ and tested on $D_2$; and so on. Unlike the holdout and random subsampling methods, here each sample is used the same number of times for training and once for testing. For classification, the accuracy estimate is the overall number of correct classifications from the $k$ iterations, divided by the total number of tuples in the initial data.

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

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

### 8.5.4 Bootstrap

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

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

"*Where does the figure, 63.2%, come from?*" Each tuple has a probability of $1/d$ of being selected, so the probability of not being chosen is $(1-1/d)$. We have to select $d$ times, so the probability that a tuple will not be chosen during this whole time is $(1-1/d)^d$. If $d$ is large, the probability approaches $e^{-1} = 0.368$.[7] Thus, 36.8% of tuples will not be selected for training and thereby end up in the test set, and the remaining 63.2% will form the training set.

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

$$Acc(M) = \frac{1}{k}\sum_{i=1}^{k}(0.632 \times Acc(M_i)_{test\_set} + 0.368 \times Acc(M_i)_{train\_set}), \tag{8.30}$$

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

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

### 8.5.5 Model Selection Using Statistical Tests of Significance

Suppose that we have generated two classification models, $M_1$ and $M_2$, from our data. We have performed 10-fold cross-validation to obtain a mean error rate[8] for each. How can we determine which model is best? It may seem intuitive to select the model with the lowest error rate; however, the mean error rates are just *estimates* of error on the true population of future data cases. There can be considerable variance between error rates within any given 10-fold cross-validation experiment. Although the mean error rates obtained for $M_1$ and $M_2$ may appear different, that difference may not be statistically significant. What if any difference between the two may just be attributed to chance? This section addresses these questions.

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

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

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

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

model. The $t$-statistic for pairwise comparison is computed as follows:

$$t = \frac{\overline{err}(M_1) - \overline{err}(M_2)}{\sqrt{var(M_1 - M_2)/k}}, \tag{8.31}$$

where

$$var(M_1 - M_2) = \frac{1}{k}\sum_{i=1}^{k}\left[err(M_1)_i - err(M_2)_i - (\overline{err}(M_1) - \overline{err}(M_2))\right]^2. \tag{8.32}$$

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

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

$$var(M_1 - M_2) = \sqrt{\frac{var(M_1)}{k_1} + \frac{var(M_2)}{k_2}}, \tag{8.33}$$

and $k_1$ and $k_2$ are the number of cross-validation samples (in our case, 10-fold cross-validation rounds) used for $M_1$ and $M_2$, respectively. This is also known as the **two sample *t*-test**.[9] When consulting the table of $t$-distribution, the number of degrees of freedom used is taken as the minimum number of degrees of the two models.

### 8.5.6 Comparing Classifiers Based on Cost–Benefit and ROC Curves

The true positives, true negatives, false positives, and false negatives are also useful in assessing the **costs and benefits** (or risks and gains) associated with a classification

[9]This test was used in sampling cubes for OLAP-based mining in Chapter 5.

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

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

**Receiver operating characteristic curves** are a useful visual tool for comparing two classification models. ROC curves come from signal detection theory that was developed during World War II for the analysis of radar images. An ROC curve for a given model shows the trade-off between the *true positive rate* (*TPR*) and the *false positive rate* (*FPR*).[10] Given a test set and a model, *TPR* is the proportion of positive (or "yes") tuples that are correctly labeled by the model; *FPR* is the proportion of negative (or "no") tuples that are mislabeled as positive. Given that *TP*, *FP*, *P*, and *N* are the number of true positive, false positive, positive, and negative tuples, respectively, from Section 8.5.1 we know that $TPR = \frac{TP}{P}$, which is sensitivity. Furthermore, $FPR = \frac{FP}{N}$, which is $1 - specificity$.

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

To plot an ROC curve for a given classification model, *M*, the model must be able to return a probability of the predicted class for each test tuple. With this information, we rank and sort the tuples so that the tuple that is most likely to belong to the positive or "yes" class appears at the top of the list, and the tuple that is least likely to belong to the positive class lands at the bottom of the list. Naïve Bayesian (Section 8.3) and backpropagation (Section 9.2) classifiers return a class probability distribution for each prediction and, therefore, are appropriate, although other classifiers, such as decision tree classifiers (Section 8.2), can easily be modified to return class probability predictions. Let the value

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

that a probabilistic classifier returns for a given tuple $\boldsymbol{X}$ be $f(\boldsymbol{X}) \rightarrow [0,1]$. For a binary problem, a threshold $t$ is typically selected so that tuples where $f(\boldsymbol{X}) \geq t$ are considered positive and all the other tuples are considered negative. Note that the number of true positives and the number of false positives are both functions of $t$, so that we could write $TP(t)$ and $FP(t)$. Both are monotonic descending functions.

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

**Example 8.11** **Plotting an ROC curve.** Figure 8.18 shows the probability value (column 3) returned by a probabilistic classifier for each of the 10 tuples in a test set, sorted by decreasing probability order. Column 1 is merely a tuple identification number, which aids in our explanation. Column 2 is the actual class label of the tuple. There are five positive tuples and five negative tuples, thus $P = 5$ and $N = 5$. As we examine the known class label of each tuple, we can determine the values of the remaining columns, *TP*, *FP*, *TN*, *FN*, *TPR*, and *FPR*. We start with tuple 1, which has the highest probability score, and take that score as our threshold, that is, $t = 0.9$. Thus, the classifier considers tuple 1 to be positive, and all the other tuples are considered negative. Since the actual class label of tuple 1 is positive, we have a true positive, hence $TP = 1$ and $FP = 0$. Among the

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

**Figure 8.18** Tuples sorted by decreasing score, where the score is the value returned by a probabilistic classifier.

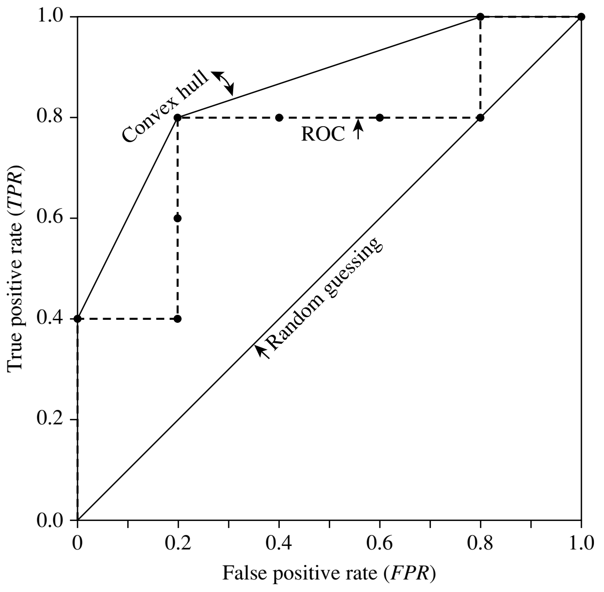


**Figure 8.19** ROC curve for the data in Figure 8.18.

remaining nine tuples, which are all classified as negative, five actually are negative (thus, $TN = 5$). The remaining four are all actually positive, thus, $FN = 4$. We can therefore compute $TPR = \frac{TP}{P} = \frac{1}{5} = 0.2$, while $FPR = 0$. Thus, we have the point (0.2, 0) for the ROC curve.

Next, threshold $t$ is set to 0.8, the probability value for tuple 2, so this tuple is now also considered positive, while tuples 3 through 10 are considered negative. The actual class label of tuple 2 is positive, thus now $TP = 2$. The rest of the row can easily be computed, resulting in the point (0.4, 0). Next, we examine the class label of tuple 3 and let $t$ be 0.7, the probability value returned by the classifier for that tuple. Thus, tuple 3 is considered positive, yet its actual label is negative, and so it is a false positive. Thus, $TP$ stays the same and $FP$ increments so that $FP = 1$. The rest of the values in the row can also be easily computed, yielding the point (0.4, 0.2). The resulting ROC graph, from examining each tuple, is the jagged line shown in Figure 8.19.

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

Figure 8.20 shows the ROC curves of two classification models. The diagonal line representing random guessing is also shown. Thus, the closer the ROC curve of a model is to the diagonal line, the less accurate the model. If the model is really good, initially we are more likely to encounter true positives as we move down the ranked list. Thus,

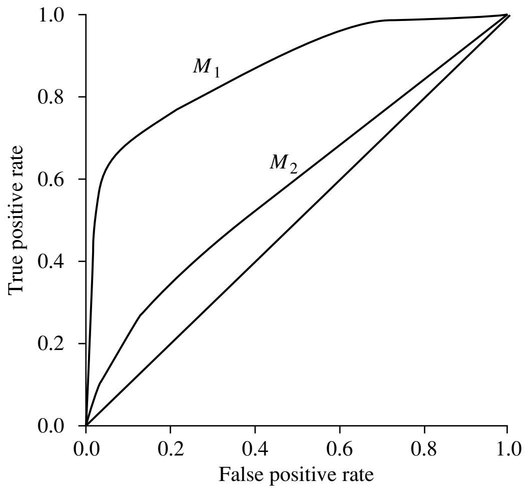


**Figure 8.20** ROC curves of two classification models, $M_1$ and $M_2$. The diagonal shows where, for every true positive, we are equally likely to encounter a false positive. The closer an ROC curve is to the diagonal line, the less accurate the model is. Thus, $M1$ is more accurate here.

the curve moves steeply up from zero. Later, as we start to encounter fewer and fewer true positives, and more and more false positives, the curve eases off and becomes more horizontal.

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

## 8.6 Techniques to Improve Classification Accuracy

In this section, you will learn some tricks for increasing classification accuracy. We focus on *ensemble methods*. An ensemble for classification is a composite model, made up of a combination of classifiers. The individual classifiers vote, and a class label prediction is returned by the ensemble based on the collection of votes. Ensembles tend to be more accurate than their component classifiers. We start off in Section 8.6.1 by introducing ensemble methods in general. Bagging (Section 8.6.2), boosting (Section 8.6.3), and random forests (Section 8.6.4) are popular ensemble methods.

Traditional learning models assume that the data classes are well distributed. In many real-world data domains, however, the data are class-imbalanced, where the main class of interest is represented by only a few tuples. This is known as the *class*