6 Classification and Prediction

Databases are rich with hidden information that can be used for intelligent decision making. Classification and prediction are two forms of data analysis that can be used to extract models describing important data classes or to predict future data trends. Such analysis can help provide us with a better understanding of the data at large. Whereas *classification* predicts categorical (discrete, unordered) labels, *prediction* models continuous-valued functions. For example, we can build a classification model to categorize bank loan applications as either safe or risky, or a prediction model to predict the expenditures in dollars of potential customers on computer equipment given their income and occupation. Many classification and prediction methods have been proposed by researchers in machine learning, pattern recognition, and statistics. Most algorithms are memory resident, typically assuming a small data size. Recent data mining research has built on such work, developing scalable classification and prediction techniques capable of handling large disk-resident data.

In this chapter, you will learn basic techniques for data classification, such as how to build decision tree classifiers, Bayesian classifiers, Bayesian belief networks, and rule-based classifiers. Backpropagation (a neural network technique) is also discussed, in addition to a more recent approach to classification known as support vector machines. Classification based on association rule mining is explored. Other approaches to classification, such as *k*-nearest-neighbor classifiers, case-based reasoning, genetic algorithms, rough sets, and fuzzy logic techniques, are introduced. Methods for prediction, including linear regression, nonlinear regression, and other regression-based models, are briefly discussed. Where applicable, you will learn about extensions to these techniques for their application to classification and prediction in *large* databases. Classification and prediction have numerous applications, including fraud detection, target marketing, performance prediction, manufacturing, and medical diagnosis.

What Is Classification? What Is Prediction?

A bank loans officer needs analysis of her data in order to learn which loan applicants are "safe" and which are "risky" for the bank. A marketing manager at *AllElectronics* needs data

analysis to help guess whether a customer with a given profile will buy a new computer. A medical researcher wants to analyze breast cancer data in order to predict which one of three specific treatments a patient should receive. In each of these examples, the data analysis task is **classification**, where a model or **classifier** is constructed to predict *categorical labels*, such as "safe" or "risky" for the loan application data; "yes" or "no" for the marketing data; or "treatment A," "treatment B," or "treatment C" for the medical data. These categories can be represented by discrete values, where the ordering among values has no meaning. For example, the values 1, 2, and 3 may be used to represent treatments A, B, and C, where there is no ordering implied among this group of treatment regimes.

Suppose that the marketing manager would like to predict how much a given customer will spend during a sale at *AllElectronics*. This data analysis task is an example of **numeric prediction**, where the model constructed predicts a *continuous-valued function*, or *ordered value*, as opposed to a categorical label. This model is a **predictor. Regression analysis** is a statistical methodology that is most often used for numeric prediction, hence the two terms are often used synonymously. We do not treat the two terms as synonyms, however, because several other methods can be used for numeric prediction, as we shall see later in this chapter. Classification and numeric prediction are the two major types of **prediction problems**. For simplicity, when there is no ambiguity, we will use the shortened term of *prediction* to refer to *numeric prediction*.

"*How does classification work?* **Data classification** is a two-step process, as shown for the loan application data of Figure 6.1. (The data are simplified for illustrative purposes. In reality, we may expect many more attributes to be considered.) In the first step, a classifier is built describing a predetermined set of data classes or concepts. This is the **learning step** (or training phase), where a classification algorithm builds the classifier by analyzing or "learning from" a **training set** made up of database tuples and their associated class labels. A tuple, $\boldsymbol{X}$, is represented by an n-dimensional **attribute vector**, $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$, depicting n measurements made on the tuple from n database attributes, respectively, $A_1, A_2, \ldots, A_n$.[1] Each tuple, $\boldsymbol{X}$, is assumed to belong to a predefined class as determined by another database attribute called the **class label attribute**. The class label attribute is discrete-valued and unordered. It is *categorical* in that each value serves as a category or class. The individual tuples making up the training set are referred to as **training tuples** and are selected from the database under analysis. In the context of classification, data tuples can be referred to as *samples, examples, instances, data points*, or *objects*.[2]

Because the class label of each training tuple *is provided*, this step is also known as **supervised learning** (i.e., the learning of the classifier is "supervised" in that it is told

[1] Each attribute represents a "feature" of $\boldsymbol{X}$. Hence, the pattern recognition literature uses the term *feature vector* rather than *attribute vector*. Since our discussion is from a database perspective, we propose the term "attribute vector." In our notation, any variable representing a vector is shown in bold italic font; measurements depicting the vector are shown in italic font, e.g., $\boldsymbol{X} = (x_1, x_2, x_3)$.

[2] In the machine learning literature, training tuples are commonly referred to as *training samples*. Throughout this text, we prefer to use the term *tuples* instead of *samples*, since we discuss the theme of classification from a database-oriented perspective.

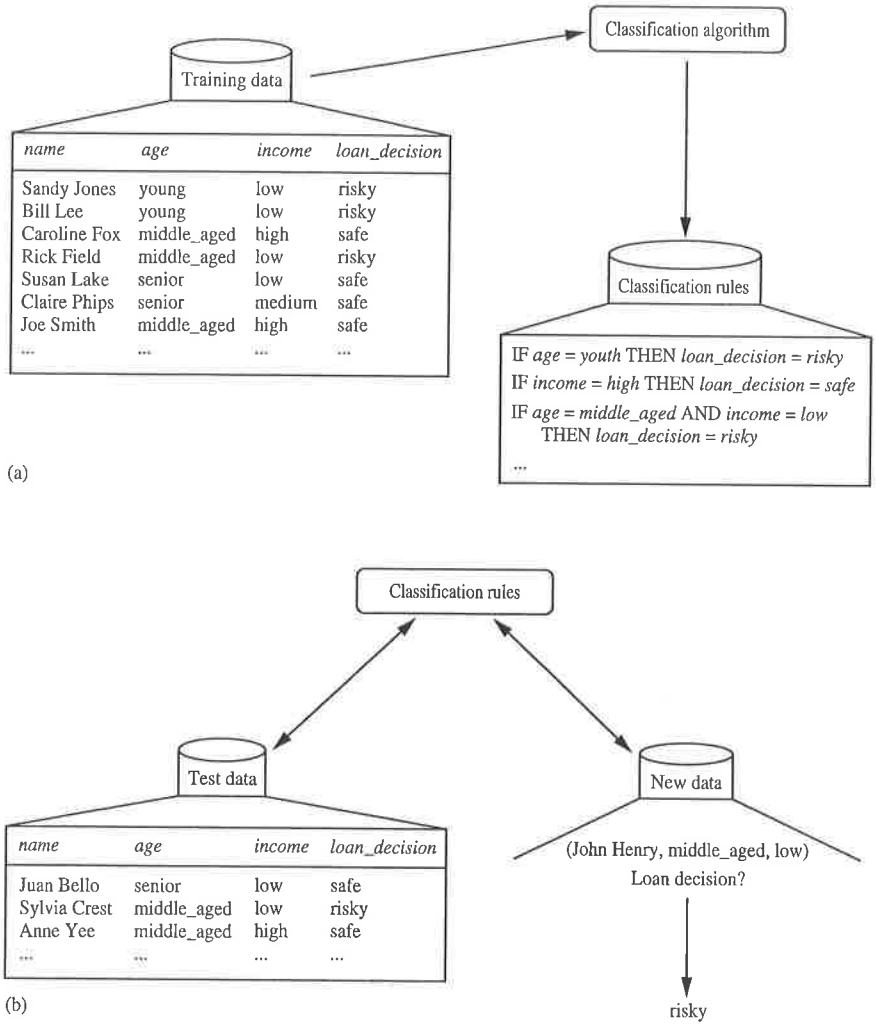


Figure 6.1 The data classification process: (a) *Learning*: Training data are analyzed by a classification algorithm. Here, the class label attribute is *loan_decision*, and the learned model or classifier is represented in the form of classification rules. (b) *Classification*: Test data are used to estimate the accuracy of the classification rules. If the accuracy is considered acceptable, the rules can be applied to the classification of new data tuples.

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 Chapter 7.

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 6.1(a)). The rules can be used to categorize future data tuples, as well as provide deeper insight into the database contents. They also provide a compressed representation of the data.

"What about classification accuracy?" In the second step (Figure 6.1(b)), the model is used for classification. First, the predictive accuracy of the classifier is estimated. If we were to use the training set to measure the accuracy of the classifier, 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. These tuples are randomly selected from the general data set. They are independent of the training tuples, meaning that they are 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 6.13 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 6.1(a) from the analysis of data from previous loan applications can be used to approve or reject new or future loan applicants.

"How is (numeric) prediction different from classification?" Data prediction is a two-step process, similar to that of data classification as described in Figure 6.1. However, for prediction, we lose the terminology of "class label attribute" because the attribute for which values are being predicted is continuous-valued (ordered) rather than categorical (discrete-valued and unordered). The attribute can be referred to simply as the **predicted attribute**.[3] Suppose that, in our example, we instead wanted to predict the amount (in dollars) that would be "safe" for the bank to loan an applicant. The data mining task becomes prediction, rather than classification. We would replace the categorical attribute, *loan_decision*, with the continuous-valued *loan_amount* as the predicted attribute, and build a predictor for our task.

Note that prediction can also be viewed as a mapping or function, $y = f(\boldsymbol{X})$, where $\boldsymbol{X}$ is the input (e.g., a tuple describing a loan applicant), and the output y is a continuous or

[3]We could also use this term for classification, although for that task the term "class label attribute" is more descriptive.

ordered value (such as the predicted amount that the bank can safely loan the applicant); That is, we wish to learn a mapping or function that models the relationship between X and y.

Prediction and classification also differ in the methods that are used to build their respective models. As with classification, the training set used to build a predictor should not be used to assess its accuracy. An independent test set should be used instead. The accuracy of a predictor is estimated by computing an error based on the difference between the predicted value and the actual known value of y for each of the test tuples, X. There are various predictor error measures (Section 6.12.2). General methods for error estimation are discussed in Section 6.13.

6.2 Issues Regarding Classification and Prediction

This section describes issues regarding preprocessing the data for classification and prediction. Criteria for the comparison and evaluation of classification methods are also described.

6.2.1 Preparing the Data for Classification and Prediction

The following preprocessing steps may be applied to the data to help improve the accuracy, efficiency, and scalability of the classification or prediction process.

- **Data cleaning:** This refers to the preprocessing of data in order to remove or reduce *noise* (by applying smoothing techniques, for example) and the treatment of *missing values* (e.g., by replacing a missing value with the most commonly occurring value for that attribute, or with the most probable value based on statistics). Although most classification algorithms have some mechanisms for handling noisy or missing data, this step can help reduce confusion during learning.
- **Relevance analysis:** Many of the attributes in the data may be *redundant*. **Correlation analysis** can be used to identify whether any two given attributes are statistically related. For example, a strong correlation between attributes A_1 and A_2 would suggest that one of the two could be removed from further analysis. A database may also contain *irrelevant* attributes. **Attribute subset selection**[4] can be used in these cases to find a reduced set of attributes such that the resulting probability distribution of the data classes is as close as possible to the original distribution obtained using all attributes. Hence, relevance analysis, in the form of correlation analysis and attribute subset selection, can be used to detect attributes that do not contribute to the classification or prediction task. Including such attributes may otherwise slow down, and possibly mislead, the learning step.

[4] In machine learning, this is known as *feature subset selection.*

Ideally, the time spent on relevance analysis, when added to the time spent on learning from the resulting "reduced" attribute (or feature) subset, should be less than the time that would have been spent on learning from the original set of attributes. Hence, such analysis can help improve classification efficiency and scalability.

Data transformation and reduction: The data may be transformed by normalization, particularly when neural networks or methods involving distance measurements are used in the learning step. **Normalization** involves scaling all values for a given attribute so that they fall within a small specified range, such as -1.0 to 1.0, or 0.0 to 1.0. In methods that use distance measurements, for example, this would prevent attributes with initially large ranges (like, say, *income*) from outweighing attributes with initially smaller ranges (such as binary attributes).

The data can also be transformed by *generalizing* it to higher-level concepts. Concept hierarchies may be used for this purpose. This is particularly useful for continuous-valued attributes. For example, numeric values for the attribute *income* can be generalized to discrete ranges, such as *low*, *medium*, and *high*. Similarly, categorical attributes, like *street*, can be generalized to higher-level concepts, like *city*. Because generalization compresses the original training data, fewer input/output operations may be involved during learning.

Data can also be reduced by applying many other methods, ranging from wavelet transformation and principle components analysis to discretization techniques, such as binning, histogram analysis, and clustering.

Data cleaning, relevance analysis (in the form of correlation analysis and attribute subset selection), and data transformation are described in greater detail in Chapter 2 of this book.

6.2.2 Comparing Classification and Prediction Methods

Classification and prediction methods can be compared and evaluated according to the following criteria:

Accuracy: The accuracy of a classifier refers to the ability of a given classifier to correctly predict the class label of new or previously unseen data (i.e., tuples without class label information). Similarly, the **accuracy of a predictor** refers to how well a given predictor can guess the value of the predicted attribute for new or previously unseen data. Accuracy measures are given in Section 6.12. Accuracy can be estimated using one or more test sets that are independent of the training set. Estimation techniques, such as cross-validation and bootstrapping, are described in Section 6.13. Strategies for improving the accuracy of a model are given in Section 6.14. Because the accuracy computed is only an estimate of how well the classifier or predictor will do on new data tuples, confidence limits can be computed to help gauge this estimate. This is discussed in Section 6.15.

- **Speed:** This refers to the computational costs involved in generating and using the given classifier or predictor.
- **Robustness:** This is the ability of the classifier or predictor to make correct predictions given noisy data or data with missing values.
- **Scalability:** This refers to the ability to construct the classifier or predictor efficiently given large amounts of data.
- **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. We discuss some work in this area, such as the extraction of classification rules from a "black box" neural network classifier called backpropagation (Section 6.6.4).

These issues are discussed throughout the chapter with respect to the various classification and prediction methods presented. Recent data mining research has contributed to the development of scalable algorithms for classification and prediction. Additional contributions include the exploration of mined "associations" between attributes and their use for effective classification. Model selection is discussed in Section 6.15.

6.3 Classification by 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.

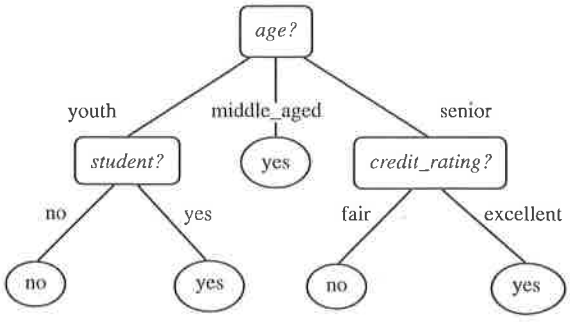


Figure 6.2 A decision tree for the concept *buys_computer*, indicating whether a customer at *AllElectronics* is likely to purchase a computer. Each internal (nonleaf) node represents a test on an attribute. Each leaf node represents a class (either *buys_computer = yes* or *buys_computer = no*).

A typical decision tree is shown in Figure 6.2. It represents the concept *buys_computer*, that is, it predicts whether a customer at *AllElectronics* is 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 high dimensional data. Their representation of acquired knowledge in tree form is intuitive and generally easy to assimilate by humans. The learning and classification steps of decision tree induction are simple and fast. In general, decision tree classifiers have good accuracy. However, successful use may depend on the data at hand. Decision tree induction algorithms have been used for classification in many application areas, such as medicine, manufacturing and production, financial analysis, astronomy, and molecular biology. Decision trees are the basis of several commercial rule induction systems.

In Section 6.3.1, we describe a basic algorithm for learning decision trees. During tree construction, *attribute selection measures* are used to select the attribute that best partitions the tuples into distinct classes. Popular measures of attribute selection are given in Section 6.3.2. When decision trees are built, many of the branches may reflect noise or outliers in the training data. *Tree pruning* attempts to identify and remove such branches, with the goal of improving classification accuracy on unseen data. Tree pruning is described in Section 6.3.3. Scalability issues for the induction of decision trees from large databases are discussed in Section 6.3.4.

6.3.1 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 such a top-down approach, which

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

Input:

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

Output: A decision tree.

Method:

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

Figure 6.3 Basic algorithm for inducing a decision tree from training tuples.

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 6.3. At first glance, the algorithm may appear long, but fear not! It is quite straightforward. The strategy is as follows.

- The algorithm is called with three parameters: D, *attribute_list*, and *Attribute_selection_method*. 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).[5]

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 of the 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 of the tuples in it belong to the same class. In other words, if we were to 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 6.4. Let A be the splitting attribute. A has v distinct values, $\{a_1, a_2, \ldots, a_v\}$, based on the training data.

1. *A is discrete-valued*: In this case, the outcomes of the test at node N 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 6.4(a)). Partition D_j is the subset of class-labeled tuples in D having value a_j of A. Because all of the tuples in a given partition have the same value for A, then A need not be considered in any future partitioning of the tuples. Therefore, it is removed from *attribute_list* (steps 8 to 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,

[5]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.

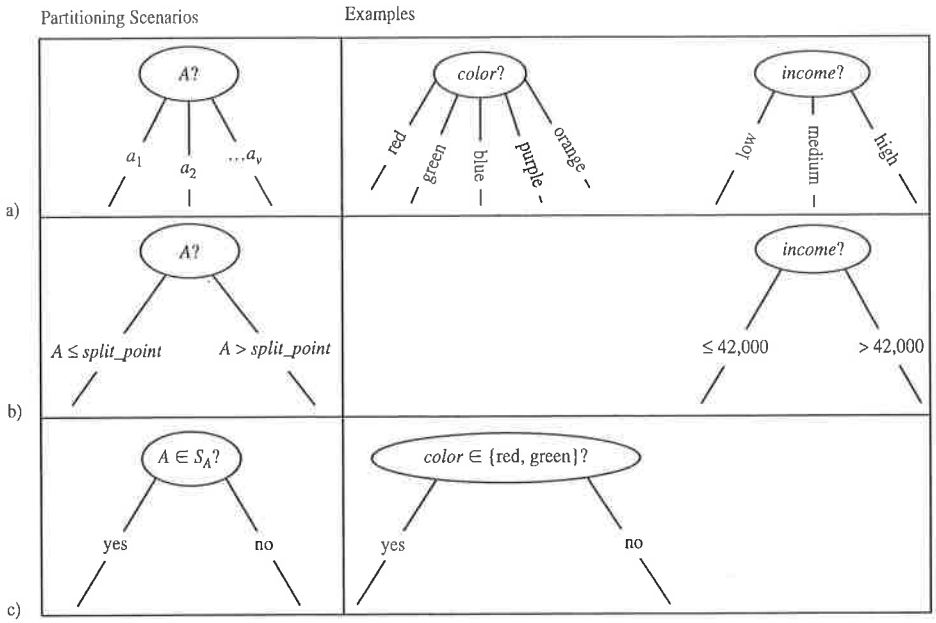


Figure 6.4 Three possibilities for partitioning tuples based on the splitting criterion, shown with examples. Let A be the splitting attribute. (a) If A is discrete-valued, then one branch is grown for each known value of A. (b) If A is continuous-valued, then two branches are grown, corresponding to $A \leq split\_point$ 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.

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 pre-existing value of A from the training data.) Two branches are grown from N and labeled according to the above outcomes (Figure 6.4(b)). The tuples are partitioned such that D_1 holds the subset of class-labeled tuples in D for which $A \leq split\_point$, while D_2 holds the rest.

3. *A is discrete-valued* and a *binary tree* must be produced (as dictated by the attribute selection measure or algorithm being used): The test at node N is of the form "$A \in S_A$?". 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 6.4(c)). By convention, the left branch out of N is labeled *yes* so that D_1 corresponds to the subset of class-labeled tuples in D

that satisfy the test. The right branch out of N is labeled *no* so that D_2 corresponds to the subset of class-labeled tuples from D that do not satisfy the test.

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

1. All of the tuples in partition D (represented at node N) belong to the same class (steps 2 and 3), or
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 6.3.2) and the mechanisms used for pruning (Section 6.3.3). The basic algorithm described above 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 6.3.4. A discussion of strategies for extracting rules from decision trees is given in Section 6.5.2 regarding rule-based classification.

6.3.2 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 of 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[6] 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 resulting partitions and reflects the least randomness or "impurity" in these partitions. Such an approach minimizes the expected number of tests needed to classify a given tuple and guarantees that a simple (but not necessarily the simplest) tree is found.

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

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i), \tag{6.1}$$

where p_i is the 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

[6] 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).

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) in order to arrive at an exact classification? This amount is measured by

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Info(D_j). \tag{6.2}$$

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

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

$$Gain(A) = Info(D) - Info_A(D). \tag{6.3}$$

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 $Info_A(D)$).

Example 6.1 **Induction of a decision tree using information gain.** Table 6.1 presents a training set, D, of class-labeled tuples randomly selected from the *AllElectronics* customer database. (The data are adapted from [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 (that is, $m = 2$). 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 Equation (6.1) 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 Equation (6.2),

Table 6.1 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

the expected information needed to classify a tuple in D if the tuples are partitioned according to *age* is

$$\begin{aligned} Info_{age}(D) &= \frac{5}{14} \times \left(-\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5}\right) \\ &\quad + \frac{4}{14} \times \left(-\frac{4}{4}\log_2\frac{4}{4} - \frac{0}{4}\log_2\frac{0}{4}\right) \\ &\quad + \frac{5}{14} \times \left(-\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5}\right) \\ &= 0.694 \text{ bits.} \end{aligned}$$

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 6.5. 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 with "*yes*." The final decision tree returned by the algorithm is shown in Figure 6.2. ■

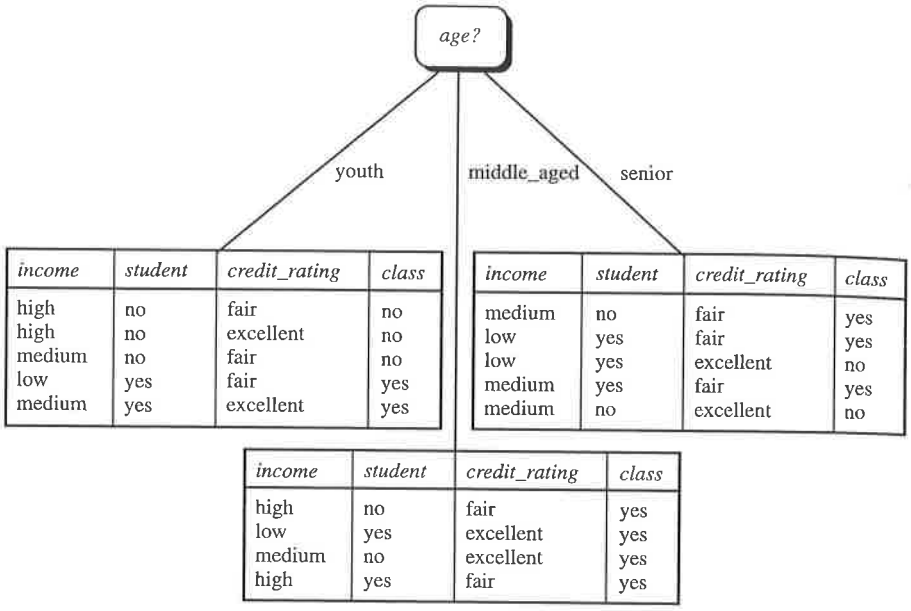


Figure 6.5 The attribute *age* has the highest information gain and therefore becomes the splitting attribute at the root node of the decision tree. Branches are grown for each outcome of *age*. The tuples are shown partitioned accordingly.

"But how can we compute the information gain of an attribute that is continuous-valued, unlike above?" 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* above, we instead 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

$$\frac{a_i + a_{i+1}}{2}. \tag{6.4}$$

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 Equation (6.2). 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 $A \leq$ *split_point*, and D_2 is the set of tuples in D satisfying $A >$ *split_point*.

Gain ratio

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

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

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2\left(\frac{|D_j|}{|D|}\right). \tag{6.5}$$

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

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo(A)}. \tag{6.6}$$

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 6.2 **Computation of gain ratio for the attribute *income*.** A test on *income* splits the data of Table 6.1 into three partitions, namely *low*, *medium*, and *high*, containing four, six, and four tuples, respectively. To compute the gain ratio of *income*, we first use Equation (6.5) to obtain

$$SplitInfo_A(D) = -\frac{4}{14} \times \log_2\left(\frac{4}{14}\right) - \frac{6}{14} \times \log_2\left(\frac{6}{14}\right) - \frac{4}{14} \times \log_2\left(\frac{4}{14}\right).$$

$$= 0.926.$$

From Example 6.1, we have $Gain(income) = 0.029$. Therefore, $GainRatio(income) = 0.029/0.926 = 0.031$. ■

Gini index

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

$$Gini(D) = 1 - \sum_{i=1}^{m} p_i^2, \tag{6.7}$$

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 of 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

$$Gini_A(D) = \frac{|D_1|}{|D|} Gini(D_1) + \frac{|D_2|}{|D|} Gini(D_2). \tag{6.8}$$

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 above 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

$$\Delta Gini(A) = Gini(D) - Gini_A(D). \tag{6.9}$$

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 6.3 **Induction of a decision tree using gini index.** Let D be the training data of Table 6.1 where there are nine tuples belonging to the class *buys_computer* = *yes* and the remaining five tuples belong to the class *buys_computer* = *no*. A (root) node N is created for the tuples in D. We first use Equation (6.7) for 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

$$\begin{aligned} &Gini_{income \in \{low,medium\}}(D) \\ &= \frac{10}{14}Gini(D_1) + \frac{4}{14}Gini(D_2) \\ &= \frac{10}{14}\left(1 - \left(\frac{6}{10}\right)^2 - \left(\frac{4}{10}\right)^2\right) + \frac{4}{14}\left(1 - \left(\frac{1}{4}\right)^2 - \left(\frac{3}{4}\right)^2\right) \\ &= 0.450 \\ &= Gini_{income \in \{high\}}(D). \end{aligned}$$

Similarly, the Gini index values for splits on the remaining subsets are: 0.315 (for the subsets {*low, high*} and {*medium*}) and 0.300 (for the subsets {*medium, high*} and {*low*}). Therefore, the best binary split for attribute *income* is on {*medium, high*} (or {*low*}) because it minimizes the gini index. Evaluating the attribute, 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 *income* and splitting subset {*medium, high*} therefore give the minimum gini index overall, with a reduction in impurity of 0.459 − 0.300 = 0.159. The binary split "*income* $\in$ {*medium, high*}" 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. Hence, the Gini index has selected *income* instead of *age* at the root node, unlike the (nonbinary) tree created by information gain (Example 6.1). ■

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-sized partitions and purity in both partitions. Although biased, these measures give reasonably good results in practice.

Many other attribute selection measures have been proposed. CHAID, a decision tree algorithm that is popular in marketing, uses an attribute selection measure that is based on the statistical χ^2 test for independence. Other measures include C-SEP (which performs better than information gain and Gini index in certain cases) and G-statistic (an information theoretic measure that is a close approximation to χ^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 is also discussed in Chapter 2, as a form of data transformation.) These other measures mentioned here are beyond the scope of this book. Additional references are given in the Bibliographic Notes at the end of this chapter.

"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.

6.3.3 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 6.6. 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).

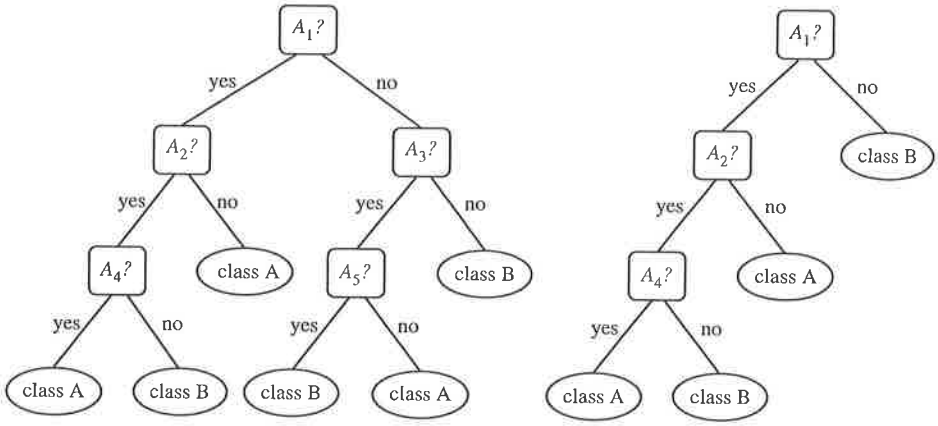


Figure 6.6 An unpruned decision tree and a pruned version of it.

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 tree of Figure 6.6. 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 Minimum Description Length (MDL) principle, which was briefly introduced in Section 6.3.2. 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 6.7), making them overwhelming to interpret. **Repetition** occurs when an attribute is repeatedly tested along a given branch of the tree (such as *"age < 60?"*, followed by *"age < 45"?*, and so on). In **replication**, duplicate subtrees exist within the tree. These situations can impede the accuracy and comprehensibility of a decision tree. The use of multivariate splits (splits based on a combination of attributes) can prevent these problems. Another approach is to use a different form of knowledge representation, such as rules, instead of decision trees. This is described in Section 6.5.2, which shows how a *rule-based classifier* can be constructed by extracting IF-THEN rules from a decision tree.

6.3.4 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! Decision tree construction therefore 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.

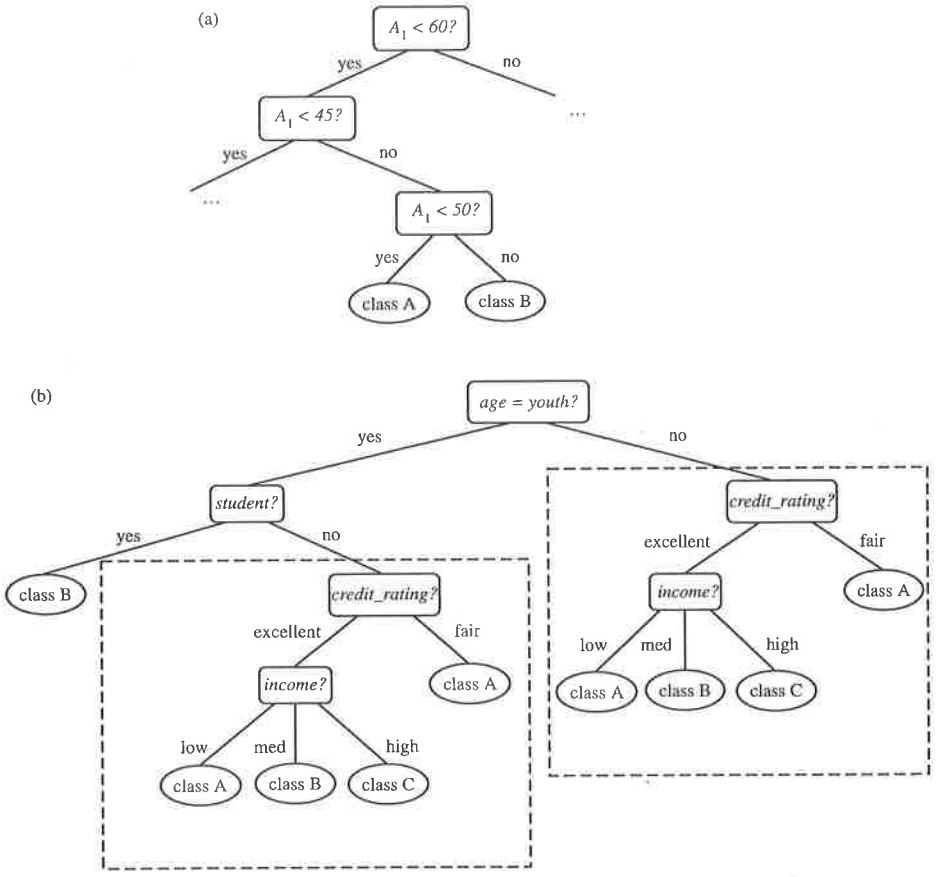


Figure 6.7 An example of subtree (a) **repetition** (where an attribute is repeatedly tested along a given branch of the tree, e.g., *age*) and (b) **replication** (where duplicate subtrees exist within a tree, such as the subtree headed by the node "*credit_rating?*").

More recent decision tree algorithms that address the scalability issue have been proposed. Algorithms for the induction of decision trees from very large training sets include SLIQ and SPRINT, both of which can handle categorical and continuous-valued attributes. Both algorithms propose presorting techniques on disk-resident data sets that are too large to fit in memory. Both define the use of new data structures to facilitate the tree construction. SLIQ employs disk-resident *attribute lists* and a single memory-resident *class list*. The attribute lists and class list generated by SLIQ for the tuple data of Table 6.2 are shown in Figure 6.8. Each attribute has an associated attribute list, indexed by *RID* (a record identifier). Each tuple is represented by a linkage of one entry from each attribute list to an entry in the class list (holding the class label of the given tuple), which in turn is linked to its corresponding leaf node

Table 6.2 Tuple data for the class *buys_computer*.

RID	*credit_rating*	*age*	*buys_computer*
1	excellent	38	yes
2	excellent	26	yes
3	fair	35	no
4	excellent	49	no
...	...	...	...

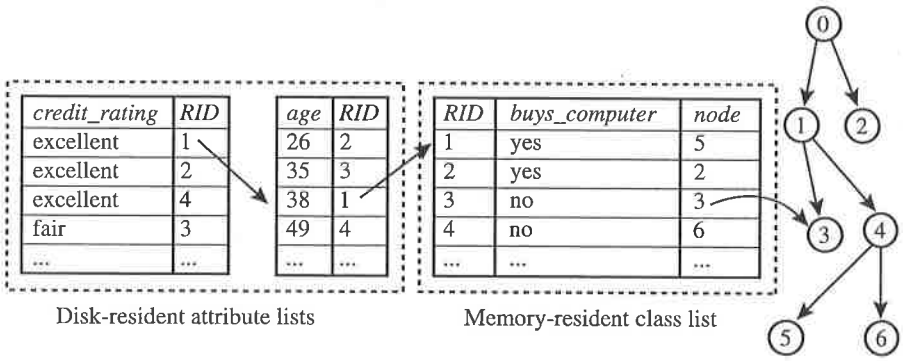


Figure 6.8 Attribute list and class list data structures used in SLIQ for the tuple data of Table 6.2.

credit_rating	*buys_computer*	*RID*
excellent	yes	1
excellent	yes	2
excellent	no	4
fair	no	3
...	...	...

age	*buys_computer*	*RID*
26	yes	2
35	no	3
38	yes	1
49	no	4
...	...	...

Figure 6.9 Attribute list data structure used in SPRINT for the tuple data of Table 6.2.

in the decision tree. The class list remains in memory because it is often accessed and modified in the building and pruning phases. The size of the class list grows proportionally with the number of tuples in the training set. When a class list cannot fit into memory, the performance of SLIQ decreases.

SPRINT uses a different *attribute list* data structure that holds the class and *RID* information, as shown in Figure 6.9. When a node is split, the attribute lists are partitioned and distributed among the resulting child nodes accordingly. When a list is

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

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

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

credit_rating	*buys_computer*	
	yes	no
fair	6	2
excellent	3	3

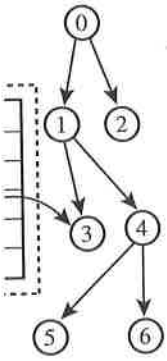
Figure 6.10 The use of data structures to hold aggregate information regarding the training data (such as these AVC-sets describing the data of Table 6.1) are one approach to improving the scalability of decision tree induction.

partitioned, the order of the records in the list is maintained. Hence, partitioning lists does not require resorting. SPRINT was designed to be easily parallelized, further contributing to its scalability.

While both SLIQ and SPRINT handle disk-resident data sets that are too large to fit into memory, the scalability of SLIQ is limited by the use of its memory-resident data structure. SPRINT removes all memory restrictions, yet requires the use of a hash tree proportional in size to the training set. This may become expensive as the training set size grows.

To further enhance the scalability of decision tree induction, a method called RainForest was proposed. It 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 6.10 shows AVC-sets for the tuple data of Table 6.1. 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 has techniques, however, for handling the case where the AVC-group does not fit in memory. RainForest can use any attribute selection measure and was shown to be more efficient than earlier approaches employing aggregate data structures, such as SLIQ and SPRINT.

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 6.13.3) 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 of the original training data had fit in memory. BOAT can use any attribute selection measure that selects



of Table 6.2.

uter	*RID*
	2
	3
	1
	4
	...

.2.

t is often accessed
he class list grows
a class list cannot

the class and *RID*
ibute lists are par-
gly. When a list is

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 in order to detect if this "very good" tree, T', is different from the "real" tree, T, that would have been generated using the entire data. It refines T' in order 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 (such as the basic algorithm in Figure 6.3), which require one scan per level of the tree! 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.

6.4 Bayesian Classification

"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 below. Studies comparing classification algorithms have found a simple Bayesian classifier known as the *naive 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." *Bayesian belief networks* are graphical models, which unlike naïve Bayesian classifiers, allow the representation of dependencies among subsets of attributes. Bayesian belief networks can also be used for classification.

Section 6.4.1 reviews basic probability notation and Bayes' theorem. In Section 6.4.2 you will learn how to do naïve Bayesian classification. Bayesian belief networks are described in Section 6.4.3.

6.4.1 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

such as the
e in order to
t would have

nent, even in
algorithm in
nd to be two
ree. An addi-
'hat is, BOAT
decision tree
h.

They can pre-
ole belongs to

dies compar-
n as the *naive*
selected neu-
cy and speed

a given class
d *class condi-*
in this sense,
unlike naïve
of attributes.

Section 6.4.2
orks are des-

rgyman who
y. Let X be a
described by
such as that
s, we want to
'evidence" or
that tuple X

nditioned on
described by

the attributes *age* and *income*, respectively, and that 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 $P(H|X)$ reflects the probability that customer 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, $P(H|X)$, is based on more information (e.g., customer information) than the prior probability, $P(H)$, which is independent of X.

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

$P(X)$ is the prior probability of 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?" $P(H)$, $P(X|H)$, and $P(X)$ may be estimated from the given data, as we shall see below. **Bayes' theorem** is useful in that it provides a way of calculating the posterior probability, $P(H|X)$, from $P(H)$, $P(X|H)$, and $P(X)$. Bayes' theorem is

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)}. \tag{6.10}$$

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

6.4.2 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, $X = (x_1, x_2, \ldots, x_n)$, 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, X, the classifier will predict that X belongs to the class having the highest posterior probability, conditioned on X. That is, the naïve Bayesian classifier predicts that tuple X belongs to the class C_i if and only if

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

Thus we maximize $P(C_i|X)$. The class C_i for which $P(C_i|X)$ is maximized is called the *maximum posteriori hypothesis*. By Bayes' theorem (Equation (6.10)),

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

3. As $P(X)$ is constant for all classes, only $P(X|C_i)P(C_i)$ need 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 $P(\boldsymbol{X}|C_i)$. In order to reduce computation in evaluating $P(\boldsymbol{X}|C_i)$, the naive assumption of **class conditional independence** is made. This presumes that the values of the attributes 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,

$$\begin{aligned} P(\boldsymbol{X}|C_i) &= \prod_{k=1}^{n} P(x_k|C_i) \\ &= P(x_1|C_i) \times P(x_2|C_i) \times \cdots \times P(x_n|C_i). \end{aligned} \tag{6.12}$$

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 μ and standard deviation σ, defined by

$$g(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{6.13}$$

so that

$$P(x_k|C_i) = g(x_k, \mu_{C_i}, \sigma_{C_i}). \tag{6.14}$$

These equations may appear daunting, but hold on! We need to compute μ_{C_i} and σ_{C_i}, which are the mean (i.e., average) and standard deviation, respectively, of the values of attribute A_k for training tuples of class C_i. We then plug these two quantities into Equation (6.13), together with x_k, in order to estimate $P(x_k|C_i)$. For example, let $\boldsymbol{X} = (35, \$40{,}000)$, where A_1 and A_2 are the attributes *age* and *income*, respectively. Let the class label attribute be *buys_computer*. The associated class label for $\boldsymbol{X}$ is *yes* (i.e., *buys_computer* = *yes*). Let's suppose that *age* has not been discretized and therefore exists as a continuous-valued attribute. Suppose that from the training set, we find that customers in D who buy a computer are 38 ± 12 years of age. In other words, for attribute *age* and this class, we have $\mu = 38$ years and $\sigma = 12$. We can plug these quantities, along with $x_1 = 35$ for our tuple $\boldsymbol{X}$ into Equation (6.13) in order to estimate $P(age = 35|buys\_computer = yes)$. For a quick review of mean and standard deviation calculations, please see Section 2.2.

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

$$P(\mathbf{X}|C_i)P(C_i) > P(\mathbf{X}|C_j)P(C_j) \quad \text{for } 1 \leq j \leq m, j \neq i. \tag{6.15}$$

In other words, the predicted class label is the class C_i for which $P(\mathbf{X}|C_i)P(C_i)$ 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 6.4 **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 6.3 for decision tree induction. The training data are in Table 6.1. The data tuples are described by the attributes *age*, *income*, *student*, and *credit_rating*. The class label attribute, *buys_computer*, has two distinct values (namely, {*yes*, *no*}). Let C_1 correspond to the class *buys_computer* = *yes* and C_2 correspond to *buys_computer* = *no*. The tuple we wish to classify is

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

We need to maximize $P(\mathbf{X}|C_i)P(C_i)$, 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 $P\mathbf{X}|C_i)$, for $i = 1, 2$, we compute the following conditional probabilities:

$$P(age = youth \mid buys\_computer = yes) = 2/9 = 0.222$$
$$P(age = youth \mid buys\_computer = no) = 3/5 = 0.600$$
$$P(income = medium \mid buys\_computer = yes) = 4/9 = 0.444$$
$$P(income = medium \mid buys\_computer = no) = 2/5 = 0.400$$
$$P(student = yes \mid buys\_computer = yes) = 6/9 = 0.667$$
$$P(student = yes \mid buys\_computer = no) = 1/5 = 0.200$$
$$P(credit\_rating = fair \mid buys\_computer = yes) = 6/9 = 0.667$$
$$P(credit\_rating = fair \mid buys\_computer = no) = 2/5 = 0.400$$

Using the above probabilities, we obtain

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

Similarly,

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

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

$$P(\mathbf{X}|buys\_computer = yes)P(buys\_computer = yes) = 0.044 \times 0.643 = 0.028$$

$$P(\mathbf{X}|buys\_computer = no)P(buys\_computer = no) = 0.019 \times 0.357 = 0.007$$

Therefore, the naïve Bayesian classifier predicts *buys_computer = yes* for tuple $\mathbf{X}$. ■

"*What if I encounter probability values of zero?*" Recall that in Equation (6.12), we estimate $P(\mathbf{X}|C_i)$ 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 $P(\mathbf{X}|C_i)$ for *each* class ($i = 1, 2, \ldots, m$) in order to find the class C_i for which $P(\mathbf{X}|C_i)P(C_i)$ 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 $\mathbf{X}$, we need to count the number of tuples having that attribute-value pair, per class (i.e., per C_i, for $i = 1, \ldots, m$). In Example 6.4, we have two classes ($m = 2$), namely *buys_computer = yes* and *buys_computer = no*. Therefore, for the attribute-value pair *student = yes* of $\mathbf{X}$, say, we need two counts—the number of customers who are students and for which *buys_computer = yes* (which contributes to $P(\mathbf{X}|buys\_computer = yes)$) and the number of customers who are students and for which *buys_computer = no* (which contributes to $P(\mathbf{X}|buys\_computer = no)$). 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 Equation (6.12) would return a zero probability for $P(\mathbf{X}|C_i)$, even though, without the zero probability, we may have ended up with a high probability, suggesting that $\mathbf{X}$ belonged to class C_i! A zero probability cancels the effects of all of 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 the following example.

Example 6.5 **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 1,000 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 999/1000), and 0.010 (from 10/1,000), 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):

$$\frac{1}{1,003} = 0.001, \frac{991}{1,003} = 0.988, \text{ and } \frac{11}{1,003} = 0.011,$$

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

6.4.3 Bayesian Belief Networks

The naïve Bayesian classifier makes the assumption of class conditional independence, that is, given the class label of a tuple, the values of the attributes are assumed to be conditionally independent of one another. This simplifies computation. When the assumption holds true, then the naïve Bayesian classifier is the most accurate in comparison with all other classifiers. In practice, however, dependencies can exist between variables. **Bayesian belief networks** specify joint conditional probability distributions. They allow class conditional independencies to be defined between subsets of variables. They provide a graphical model of causal relationships, on which learning can be performed. Trained Bayesian belief networks can be used for classification. Bayesian belief networks are also known as **belief networks**, **Bayesian networks**, and **probabilistic networks**. For brevity, we will refer to them as belief networks.

A belief network is defined by two components—a *directed acyclic graph* and a set of *conditional probability tables* (Figure 6.11). Each node in the directed acyclic graph represents a random variable. The variables may be discrete or continuous-valued. They may correspond to actual attributes given in the data or to "hidden variables" believed to form a relationship (e.g., in the case of medical data, a hidden variable may indicate a syndrome, representing a number of symptoms that, together, characterize a specific disease). Each arc represents a probabilistic dependence. If an arc is drawn from a node Y to a node Z, then Y is a **parent** or **immediate predecessor** of Z, and Z is a **descendant** of Y. *Each variable is conditionally independent of its nondescendants in the graph, given its parents.*

Figure 6.11 is a simple belief network, adapted from [RBKK95] for six Boolean variables. The arcs in Figure 6.11(a) allow a representation of causal knowledge. For example, having lung cancer is influenced by a person's family history of lung cancer, as well as whether or not the person is a smoker. Note that the variable *PositiveXRay* is independent of whether the patient has a family history of lung cancer or is a smoker, given that we know the patient has lung cancer. In other words, once we know the outcome of the variable *LungCancer*, then the variables *FamilyHistory* and *Smoker* do not provide

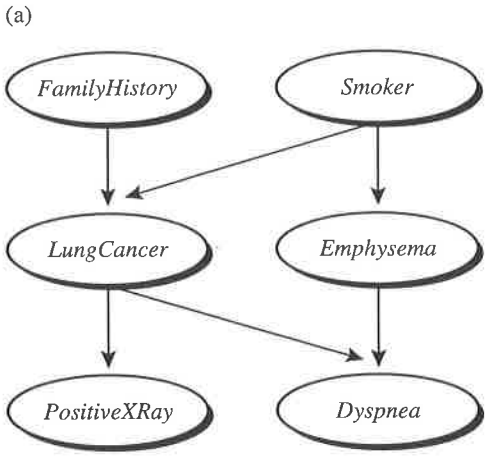


Figure 6.11 A simple Bayesian belief network: (a) A proposed causal model, represented by a directed acyclic graph. (b) The conditional probability table for the values of the variable *LungCancer (LC)* showing each possible combination of the values of its parent nodes, *FamilyHistory (FH)* and *Smoker (S)*. Figure is adapted from [RBKK95].

any additional information regarding *PositiveXRay*. The arcs also show that the variable *LungCancer* is conditionally independent of *Emphysema*, given its parents, *FamilyHistory* and *Smoker*.

A belief network has one **conditional probability table** (CPT) for each variable. The CPT for a variable Y specifies the conditional distribution $P(Y|Parents(Y))$, where $Parents(Y)$ are the parents of Y. Figure 6.11(b) shows a CPT for the variable *LungCancer*. The conditional probability for each known value of *LungCancer* is given for each possible combination of values of its parents. For instance, from the upper leftmost and bottom rightmost entries, respectively, we see that

$$P(LungCancer = yes \mid FamilyHistory = yes, Smoker = yes) = 0.8$$
$$P(LungCancer = no \mid FamilyHistory = no, Smoker = no) = 0.9$$

Let $X = (x_1, \ldots, x_n)$ be a data tuple described by the variables or attributes $Y_1, \ldots, Y_n$, respectively. Recall that each variable is conditionally independent of its nondescendants in the network graph, given its parents. This allows the network to provide a complete representation of the existing joint probability distribution with the following equation:

$$P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i|Parents(Y_i)), \tag{6.16}$$

where $P(x_1, \ldots, x_n)$ is the probability of a particular combination of values of X, and the values for $P(x_i|Parents(Y_i))$ correspond to the entries in the CPT for Y_i.

A node within the network can be selected as an "output" node, representing a class label attribute. There may be more than one output node. Various algorithms for learning can be applied to the network. Rather than returning a single class label, the classification process can return a probability distribution that gives the probability of each class.

6.4.4 Training Bayesian Belief Networks

"How does a Bayesian belief network learn?" In the learning or training of a belief network, a number of scenarios are possible. The network **topology** (or "layout" of nodes and arcs) may be given in advance or inferred from the data. The network variables may be *observable* or *hidden* in all or some of the training tuples. The case of hidden data is also referred to as *missing values* or *incomplete data*.

Several algorithms exist for learning the network topology from the training data given observable variables. The problem is one of discrete optimization. For solutions, please see the bibliographic notes at the end of this chapter. Human experts usually have a good grasp of the direct conditional dependencies that hold in the domain under analysis, which helps in network design. Experts must specify conditional probabilities for the nodes that participate in direct dependencies. These probabilities can then be used to compute the remaining probability values.

If the network topology is known and the variables are observable, then training the network is straightforward. It consists of computing the CPT entries, as is similarly done when computing the probabilities involved in naive Bayesian classification.

When the network topology is given and some of the variables are hidden, there are various methods to choose from for training the belief network. We will describe a promising method of gradient descent. For those without an advanced math background, the description may look rather intimidating with its calculus-packed formulae. However, packaged software exists to solve these equations, and the general idea is easy to follow.

Let D be a training set of data tuples, $\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_{|D|}$. Training the belief network means that we must learn the values of the CPT entries. Let w_{ijk} be a CPT entry for the variable $Y_i = y_{ij}$ having the parents $U_i = u_{ik}$, where $w_{ijk} \equiv P(Y_i = y_{ij} | U_i = u_{ik})$. For example, if w_{ijk} is the upper leftmost CPT entry of Figure 6.11(b), then Y_i is *LungCancer*; y_{ij} is its value, *"yes"*; U_i lists the parent nodes of Y_i, namely, {*FamilyHistory, Smoker*}; and u_{ik} lists the values of the parent nodes, namely, {*"yes", "yes"*}. The w_{ijk} are viewed as weights, analogous to the weights in hidden units of neural networks (Section 6.6). The set of weights is collectively referred to as $\boldsymbol{W}$. The weights are initialized to random probability values. A *gradient descent* strategy performs greedy hill-climbing. At each iteration, the weights are updated and will eventually converge to a local optimum solution.

A **gradient descent** strategy is used to search for the w_{ijk} values that best model the data, based on the assumption that each possible setting of w_{ijk} is equally likely. Such a strategy is iterative. It searches for a solution along the negative of the gradient (i.e., steepest descent) of a criterion function. We want to find the set of weights, $\boldsymbol{W}$, that maximize this function. To start with, the weights are initialized to random probability values.

The gradient descent method performs greedy hill-climbing in that, at each iteration or step along the way, the algorithm moves toward what appears to be the best solution at the moment, without backtracking. The weights are updated at each iteration. Eventually, they converge to a local optimum solution.

For our problem, we maximize $P_w(D) = \prod_{d=1}^{|D|} P_w(\boldsymbol{X}_d)$. This can be done by following the gradient of $\ln P_w(S)$, which makes the problem simpler. Given the network topology and initialized w_{ijk}, the algorithm proceeds as follows:

1. **Compute the gradients:** For each i, j, k, compute

$$\frac{\partial \ln P_w(D)}{\partial w_{ijk}} = \sum_{d=1}^{|D|} \frac{P(Y_i = y_{ij}, U_i = u_{ik} | \boldsymbol{X}_d)}{w_{ijk}}. \tag{6.17}$$

   The probability in the right-hand side of Equation (6.17) is to be calculated for each training tuple, $\boldsymbol{X}_d$, in D. For brevity, let's refer to this probability simply as p. When the variables represented by Y_i and U_i are hidden for some $\boldsymbol{X}_d$, then the corresponding probability p can be computed from the observed variables of the tuple using standard algorithms for Bayesian network inference such as those available in the commercial software package HUGIN (*http://www.hugin.dk*).

2. **Take a small step in the direction of the gradient:** The weights are updated by

$$w_{ijk} \leftarrow w_{ijk} + (l) \frac{\partial \ln P_w(D)}{\partial w_{ijk}}, \tag{6.18}$$

   where l is the **learning rate** representing the step size and $\frac{\partial \ln P_w(D)}{\partial w_{ijk}}$ is computed from Equation (6.17). The learning rate is set to a small constant and helps with convergence.

3. **Renormalize the weights:** Because the weights w_{ijk} are probability values, they must be between 0.0 and 1.0, and $\sum_j w_{ijk}$ must equal 1 for all i, k. These criteria are achieved by renormalizing the weights after they have been updated by Equation (6.18).

Algorithms that follow this form of learning are called *Adaptive Probabilistic Networks*. Other methods for training belief networks are referenced in the bibliographic notes at the end of this chapter. Belief networks are computationally intensive. Because belief networks provide explicit representations of causal structure, a human expert can provide prior knowledge to the training process in the form of network topology and/or conditional probability values. This can significantly improve the learning rate.

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.

We then study ways in which they can be generated, either from a decision tree or directly from the training data using a *sequential covering algorithm.*

6.5.1 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-hand side) of a rule is known as the **rule antecedent** or **precondition.** The "THEN"-part (or right-hand side) is the **rule consequent.** In the rule antecedent, the condition consists of one or more *attribute tests* (such as *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 (that is, all of 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

$$coverage(R) = \frac{n_{covers}}{|D|} \tag{6.19}$$

$$accuracy(R) = \frac{n_{correct}}{n_{covers}}. \tag{6.20}$$

That is, a rule's coverage is the percentage of tuples that are covered by the rule (i.e., whose 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 6.6 **Rule accuracy and coverage.** Let's go back to our data of Table 6.1. These are class-labeled tuples from the *AllElectronics* customer database. Our task is to predict whether a customer will buy a computer. Consider rule $R1$ above, 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 of 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 highest priority, and so it gets to fire its class prediction. Any other rule that satisfies $\boldsymbol{X}$ 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 stand-alone 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 $\boldsymbol{X}$. How, then, can we determine the class label of $\boldsymbol{X}$? 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.

6.5.2 Rule Extraction from a Decision Tree

In Section 6.3, 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 6.7 **Extracting classification rules from a decision tree.** The decision tree of Figure 6.2 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 6.2 are

$R1$: 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**. By *mutually exclusive*, this 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.) By *exhaustive*, 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 6.7 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 all rules for a single class together, 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.

6.5.3 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 tuples of that class (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. Note that in a newer alternative approach, classification rules can be generated using *associative classification algorithms*, which search for attribute-value pairs that occur frequently in the data. These pairs may form association rules, which can be analyzed and used in classification. Since this latter approach is based on association rule mining (Chapter 5), we prefer to defer its treatment until later, in Section 6.8.

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 6.12. Here, rules are learned for one class at a time. Ideally, when learning a rule for a class, C_i, we would like the rule to cover all (or many) of the training tuples of class C and none (or few) of the tuples 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

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

Input:

- D, a data set 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)      until terminating condition;
(7)      Rule_set = Rule_set + Rule; // add new rule to rule set
(8)  endfor
(9)  return Rule_Set;
```

Figure 6.12 Basic sequential covering algorithm.

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 6.13). 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

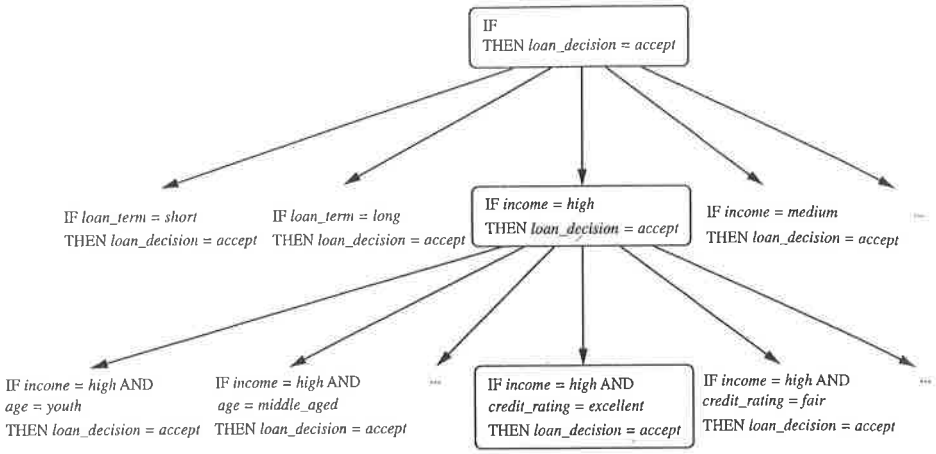


Figure 6.13 A general-to-specific search through rule space.

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* 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 6.13, 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 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 the following example.

Example 6.8 **Choosing between two rules based on accuracy.** Consider the two rules as illustrated in Figure 6.14. 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 the above 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 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*.

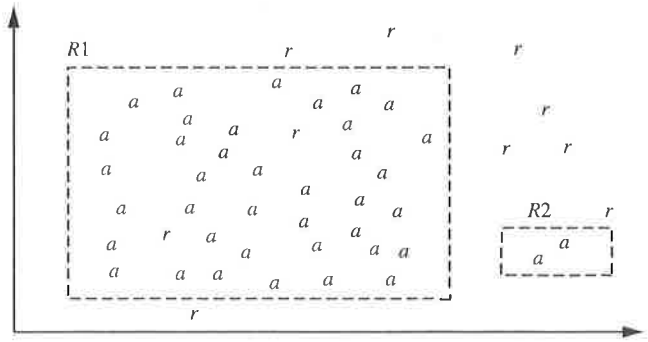


Figure 6.14 Rules for the class *loan_decision* = *accept*, showing *accept* (*a*) and *reject* (*r*) tuples.

We have already seen entropy in our discussion of the information gain measure used for attribute selection in decision tree induction (Section 6.3.2, Equation 6.1). 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).[7] 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

$$FOIL\_Gain = pos' \times \left(\log_2 \frac{pos'}{pos' + neg'} - log_2 \frac{pos}{pos + neg}\right). \tag{6.21}$$

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 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**,

$$Likelihood\_Ratio = 2 \sum_{i=1}^{m} f_i \log\left(\frac{f_i}{e_i}\right), \tag{6.22}$$

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 χ^2 distribution with $m - 1$ degrees of freedom. The higher the likelihood ratio is, 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 above are made with tuples from the original training data.

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

Such assessment is 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,

$$FOIL\_Prune(R) = \frac{pos - neg}{pos + neg}, \tag{6.23}$$

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.

Classification by Backpropagation

"What is backpropagation?" Backpropagation is a neural network learning algorithm. The field of neural networks was originally kindled by psychologists and neurobiologists who sought to develop and test computational analogues of neurons. Roughly speaking, a **neural network** is a set of connected input/output units in which each connection has a weight associated with it. During the learning phase, the network learns by adjusting the weights so as to be able to predict the correct class label of the input tuples. Neural network learning is also referred to as **connectionist learning** due to the connections between units.

Neural networks involve long training times and are therefore more suitable for applications where this is feasible. They require a number of parameters that are typically best determined empirically, such as the network topology or "structure." Neural networks have been criticized for their poor interpretability. For example, it is difficult for humans to interpret the symbolic meaning behind the learned weights and of "hidden units" in the network. These features initially made neural networks less desirable for data mining.

Advantages of neural networks, however, include their high tolerance of noisy data as well as their ability to classify patterns on which they have not been trained. They can be used when you may have little knowledge of the relationships between attributes and classes. They are well-suited for continuous-valued inputs *and outputs*, unlike most decision tree algorithms. They have been successful on a wide array of real-world data, including handwritten character recognition, pathology and laboratory medicine, and training a computer to pronounce English text. Neural network algorithms are inherently parallel; parallelization techniques can be used to speed up the computation process. In addition, several techniques have recently been developed for the extraction of rules from trained neural networks. These factors contribute toward the usefulness of neural networks for classification and prediction in data mining.

There are many different kinds of neural networks and neural network algorithms. The most popular neural network algorithm is *backpropagation*, which gained repute in the 1980s. In Section 6.6.1 you will learn about multilayer feed-forward networks, the type of neural network on which the backpropagation algorithm performs. Section 6.6.2 discusses defining a network topology. The backpropagation algorithm is described in Section 6.6.3. Rule extraction from trained neural networks is discussed in Section 6.6.4.

6.6.1 A Multilayer Feed-Forward Neural Network

The backpropagation algorithm performs learning on a *multilayer feed-forward* neural network. It iteratively learns a set of weights for prediction of the class label of tuples. A **multilayer feed-forward** neural network consists of an *input layer*, one or more *hidden layers*, and an *output layer*. An example of a multilayer feed-forward network is shown in Figure 6.15.

Each layer is made up of units. The inputs to the network correspond to the attributes measured for each training tuple. The inputs are fed simultaneously into the units making up the **input layer**. These inputs pass through the input layer and are then weighted and fed simultaneously to a second layer of "neuronlike" units, known as a **hidden layer**. The outputs of the hidden layer units can be input to another hidden layer, and so on. The number of hidden layers is arbitrary, although in practice, usually only one is used. The weighted outputs of the last hidden layer are input to units making up the **output layer**, which emits the network's prediction for given tuples.

The units in the input layer are called **input units**. The units in the hidden layers and output layer are sometimes referred to as **neurodes**, due to their symbolic biological basis, or as **output units**. The multilayer neural network shown in Figure 6.15 has two layers

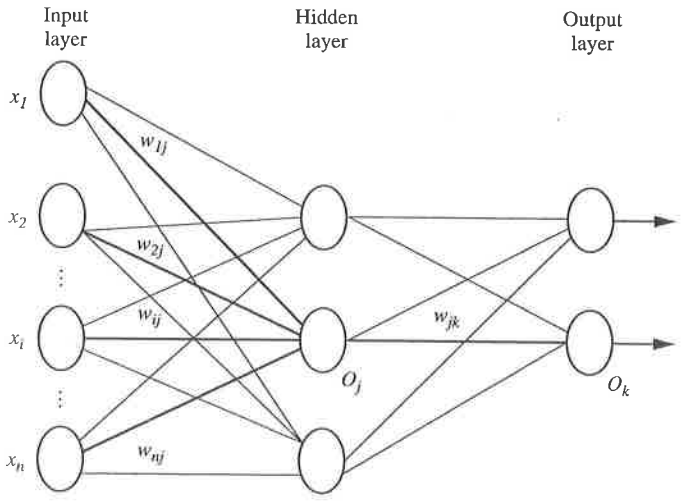


Figure 6.15 A multilayer feed-forward neural network.

of output units. Therefore, we say that it is a **two-layer** neural network. (The input layer is not counted because it serves only to pass the input values to the next layer.) Similarly, a network containing two hidden layers is called a *three-layer* neural network, and so on. The network is **feed-forward** in that none of the weights cycles back to an input unit or to an output unit of a previous layer. It is **fully connected** in that each unit provides input to each unit in the next forward layer.

Each output unit takes, as input, a weighted sum of the outputs from units in the previous layer (see Figure 6.17). It applies a nonlinear (activation) function to the weighted input. Multilayer feed-forward neural networks are able to model the class prediction as a nonlinear combination of the inputs. From a statistical point of view, they perform nonlinear regression. *Multilayer feed-forward networks, given enough hidden units and enough training samples, can closely approximate any function.*

6.6.2 Defining a Network Topology

"How can I design the topology of the neural network?" Before training can begin, the user must decide on the network topology by specifying the number of units in the input layer, the number of hidden layers (if more than one), the number of units in each hidden layer, and the number of units in the output layer.

Normalizing the input values for each attribute measured in the training tuples will help speed up the learning phase. Typically, input values are normalized so as to fall between 0.0 and 1.0. Discrete-valued attributes may be encoded such that there is one input unit per domain value. For example, if an attribute A has three possible or known values, namely $\{a_0, a_1, a_2\}$, then we may assign three input units to represent A. That is, we may have, say, I_0, I_1, I_2 as input units. Each unit is initialized to 0. If $A = a_0$, then I_0 is set to 1. If $A = a_1$, I_1 is set to 1, and so on. Neural networks can be used for both classification (to predict the class label of a given tuple) or prediction (to predict a continuous-valued output). For classification, one output unit may be used to represent two classes (where the value 1 represents one class, and the value 0 represents the other). If there are more than two classes, then one output unit per class is used.

There are no clear rules as to the "best" number of hidden layer units. Network design is a trial-and-error process and may affect the accuracy of the resulting trained network. The initial values of the weights may also affect the resulting accuracy. Once a network has been trained and its accuracy is not considered acceptable, it is common to repeat the training process with a different network topology or a different set of initial weights. Cross-validation techniques for accuracy estimation (described in Section 6.13) can be used to help decide when an acceptable network has been found. A number of automated techniques have been proposed that search for a "good" network structure. These typically use a hill-climbing approach that starts with an initial structure that is selectively modified.

6.6.3 Backpropagation

"How does backpropagation work?" Backpropagation learns by iteratively processing a data set of training tuples, comparing the network's prediction for each tuple with the

actual known *target* value. The target value may be the known class label of the training tuple (for classification problems) or a continuous value (for prediction). For each training tuple, the weights are modified so as to minimize the mean squared error between the network's prediction and the actual target value. These modifications are made in the "backwards" direction, that is, from the output layer, through each hidden layer down to the first hidden layer (hence the name *backpropagation*). Although it is not guaranteed, in general the weights will eventually converge, and the learning process stops. The algorithm is summarized in Figure 6.16. The steps involved are expressed in terms of inputs, outputs, and errors, and may seem awkward if this is your first look at neural network learning. However, once you become familiar with the process, you will see that each step is inherently simple. The steps are described below.

Algorithm: Backpropagation. Neural network learning for classification or prediction, using the backpropagation algorithm.

Input:

- D, a data set consisting of the training tuples and their associated target values;
- l, the learning rate;
- *network*, a multilayer feed-forward network.

Output: A trained neural network.

Method:

(1) Initialize all weights and biases in *network*;
(2) **while** terminating condition is not satisfied {
(3) for each training tuple $\boldsymbol{X}$ in D {
(4) // Propagate the inputs forward:
(5) for each input layer unit j {
(6) $O_j = I_j$; // output of an input unit is its actual input value
(7) for each hidden or output layer unit j {
(8) $I_j = \sum_i w_{ij} O_i + \theta_j$; //compute the net input of unit j with respect to the previous layer, i
(9) $O_j = \frac{1}{1+e^{-I_j}}$; } // compute the output of each unit j
(10) // Backpropagate the errors:
(11) for each unit j in the output layer
(12) $Err_j = O_j(1 - O_j)(T_j - O_j)$; // compute the error
(13) for each unit j in the hidden layers, from the last to the first hidden layer
(14) $Err_j = O_j(1 - O_j)\sum_k Err_k w_{jk}$; // compute the error with respect to the next higher layer, k
(15) for each weight w_{ij} in *network* {
(16) $\Delta w_{ij} = (l) Err_j O_i$; // weight increment
(17) $w_{ij} = w_{ij} + \Delta w_{ij}$; } // weight update
(18) for each bias θ_j in *network* {
(19) $\Delta\theta_j = (l) Err_j$; // bias increment
(20) $\theta_j = \theta_j + \Delta\theta_j$; } // bias update
(21) } }

Figure 6.16 Backpropagation algorithm.

Initialize the weights: The weights in the network are initialized to small random numbers (e.g., ranging from −1.0 to 1.0, or −0.5 to 0.5). Each unit has a *bias* associated with it, as explained below. The biases are similarly initialized to small random numbers.

Each training tuple, X, is processed by the following steps.

Propagate the inputs forward: First, the training tuple is fed to the input layer of the network. The inputs pass through the input units, unchanged. That is, for an input unit, j, its output, O_j, is equal to its input value, I_j. Next, the net input and output of each unit in the hidden and output layers are computed. The net input to a unit in the hidden or output layers is computed as a linear combination of its inputs. To help illustrate this point, a hidden layer or output layer unit is shown in Figure 6.17. Each such unit has a number of inputs to it that are, in fact, the outputs of the units connected to it in the previous layer. Each connection has a weight. To compute the net input to the unit, each input connected to the unit is multiplied by its corresponding weight, and this is summed. Given a unit j in a hidden or output layer, the net input, I_j, to unit j is

$$I_j = \sum_i w_{ij} O_i + \theta_j, \tag{6.24}$$

where w_{ij} is the weight of the connection from unit i in the previous layer to unit j; O_i is the output of unit i from the previous layer; and θ_j is the bias of the unit. The bias acts as a threshold in that it serves to vary the activity of the unit.

Each unit in the hidden and output layers takes its net input and then applies an **activation** function to it, as illustrated in Figure 6.17. The function symbolizes the activation

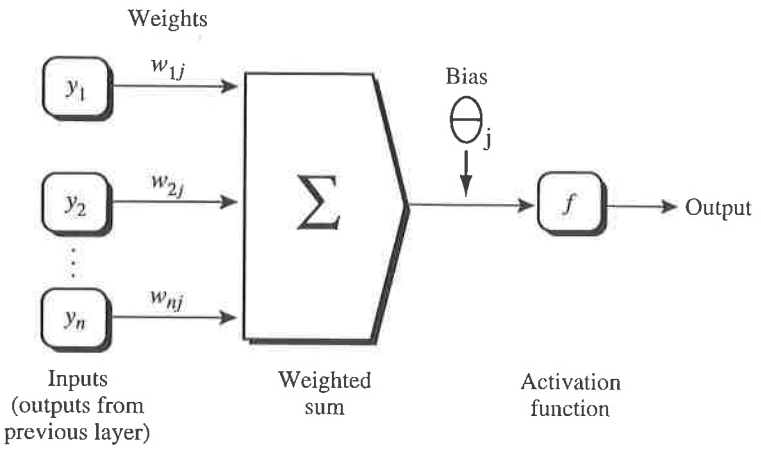


Figure 6.17 A hidden or output layer unit j: The inputs to unit j are outputs from the previous layer. These are multiplied by their corresponding weights in order to form a weighted sum, which is added to the bias associated with unit j. A nonlinear activation function is applied to the net input. (For ease of explanation, the inputs to unit j are labeled $y_1, y_2, \ldots, y_n$. If unit j were in the first hidden layer, then these inputs would correspond to the input tuple $(x_1, x_2, \ldots, x_n)$.)

of the neuron represented by the unit. The **logistic**, or **sigmoid**, function is used. Given the net input I_j to unit j, then O_j, the output of unit j, is computed as

$$O_j = \frac{1}{1+e^{-I_j}}. \tag{6.25}$$

This function is also referred to as a *squashing function*, because it maps a large input domain onto the smaller range of 0 to 1. The logistic function is nonlinear and differentiable, allowing the backpropagation algorithm to model classification problems that are linearly inseparable.

We compute the output values, O_j, for each hidden layer, up to and including the output layer, which gives the network's prediction. In practice, it is a good idea to cache (i.e., save) the intermediate output values at each unit as they are required again later, when backpropagating the error. This trick can substantially reduce the amount of computation required.

Backpropagate the error: The error is propagated backward by updating the weights and biases to reflect the error of the network's prediction. For a unit j in the output layer, the error Err_j is computed by

$$Err_j = O_j(1-O_j)(T_j - O_j), \tag{6.26}$$

where O_j is the actual output of unit j, and T_j is the known target value of the given training tuple. Note that $O_j(1-O_j)$ is the derivative of the logistic function.

To compute the error of a hidden layer unit j, the weighted sum of the errors of the units connected to unit j in the next layer are considered. The error of a hidden layer unit j is

$$Err_j = O_j(1-O_j)\sum_k Err_k w_{jk}, \tag{6.27}$$

where w_{jk} is the weight of the connection from unit j to a unit k in the next higher layer, and Err_k is the error of unit k.

The weights and biases are updated to reflect the propagated errors. Weights are updated by the following equations, where Δw_{ij} is the change in weight w_{ij}:

$$\Delta w_{ij} = (l) Err_j O_i \tag{6.28}$$

$$w_{ij} = w_{ij} + \Delta w_{ij} \tag{6.29}$$

"What is the 'l' in Equation (6.28)?" The variable l is the **learning rate**, a constant typically having a value between 0.0 and 1.0. Backpropagation learns using a method of gradient descent to search for a set of weights that fits the training data so as to minimize the mean squared distance between the network's class prediction and the known target value of the tuples.[8] The learning rate helps avoid getting stuck at a local minimum

[8] A method of gradient descent was also used for training Bayesian belief networks, as described in Section 6.4.4.

in decision space (i.e., where the weights appear to converge, but are not the optimum solution) and encourages finding the global minimum. If the learning rate is too small, then learning will occur at a very slow pace. If the learning rate is too large, then oscillation between inadequate solutions may occur. A rule of thumb is to set the learning rate to $1/t$, where t is the number of iterations through the training set so far.

Biases are updated by the following equations below, where $\Delta\theta_j$ is the change in bias θ_j:

$$\Delta\theta_j = (l)Err_j \tag{6.30}$$

$$\theta_j = \theta_j + \Delta\theta_j \tag{6.31}$$

Note that here we are updating the weights and biases after the presentation of each tuple. This is referred to as **case updating.** Alternatively, the weight and bias increments could be accumulated in variables, so that the weights and biases are updated after all of the tuples in the training set have been presented. This latter strategy is called **epoch updating,** where one iteration through the training set is an **epoch.** In theory, the mathematical derivation of backpropagation employs epoch updating, yet in practice, case updating is more common because it tends to yield more accurate results.

Terminating condition: Training stops when

- All Δw_{ij} in the previous epoch were so small as to be below some specified threshold, or
- The percentage of tuples misclassified in the previous epoch is below some threshold, or
- A prespecified number of epochs has expired.

In practice, several hundreds of thousands of epochs may be required before the weights will converge.

"How efficient is backpropagation?" The computational efficiency depends on the time spent training the network. Given $|D|$ tuples and w weights, each epoch requires $O(|D| \times w)$ time. However, in the worst-case scenario, the number of epochs can be exponential in n, the number of inputs. In practice, the time required for the networks to converge is highly variable. A number of techniques exist that help speed up the training time. For example, a technique known as *simulated annealing* can be used, which also ensures convergence to a global optimum.

Example 6.9 Sample calculations for learning by the backpropagation algorithm. Figure 6.18 shows a multilayer feed-forward neural network. Let the learning rate be 0.9. The initial weight and bias values of the network are given in Table 6.3, along with the first training tuple, $X = (1, 0, 1)$, whose class label is 1.

This example shows the calculations for backpropagation, given the first training tuple, X. The tuple is fed into the network, and the net input and output of each unit are computed. These values are shown in Table 6.4. The error of each unit is computed

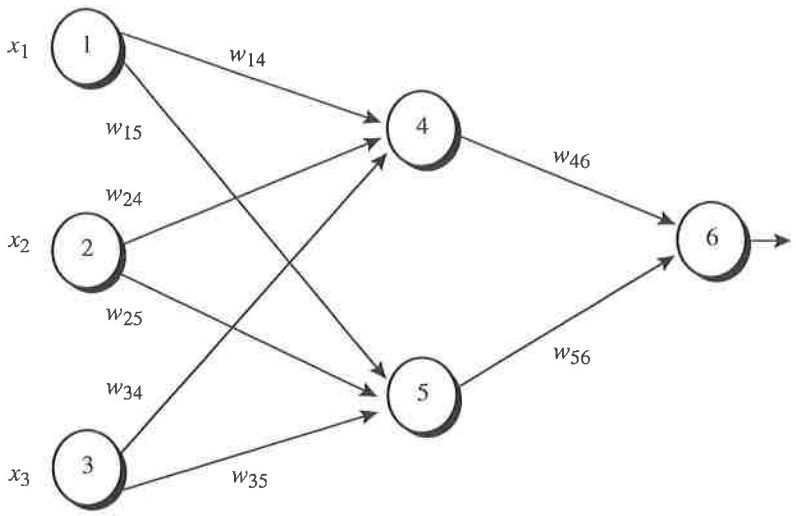


Figure 6.18 An example of a multilayer feed-forward neural network.

Table 6.3 Initial input, weight, and bias values.

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5	θ_6
1	0	1	0.2	−0.3	0.4	0.1	−0.5	0.2	−0.3	−0.2	−0.4	0.2	0.1

Table 6.4 The net input and output calculations.

Unit j	*Net input, I_j*	*Output, O_j*
4	$0.2+0-0.5-0.4=-0.7$	$1/(1+e^{0.7})=0.332$
5	$-0.3+0+0.2+0.2=0.1$	$1/(1+e^{-0.1})=0.525$
6	$(-0.3)(0.332)-(0.2)(0.525)+0.1=-0.105$	$1/(1+e^{0.105})=0.474$

and propagated backward. The error values are shown in Table 6.5. The weight and bias updates are shown in Table 6.6. ■

Several variations and alternatives to the backpropagation algorithm have been proposed for classification in neural networks. These may involve the dynamic adjustment of the network topology and of the learning rate or other parameters, or the use of different error functions.

6.6.4 Inside the Black Box: Backpropagation and Interpretability

"Neural networks are like a black box. How can I 'understand' what the backpropagation network has learned?" A major disadvantage of neural networks lies in their knowledge

Table 6.5 Calculation of the error at each node.

Unit j	Err_j
6	$(0.474)(1-0.474)(1-0.474) = 0.1311$
5	$(0.525)(1-0.525)(0.1311)(-0.2) = -0.0065$
4	$(0.332)(1-0.332)(0.1311)(-0.3) = -0.0087$

Table 6.6 Calculations for weight and bias updating.

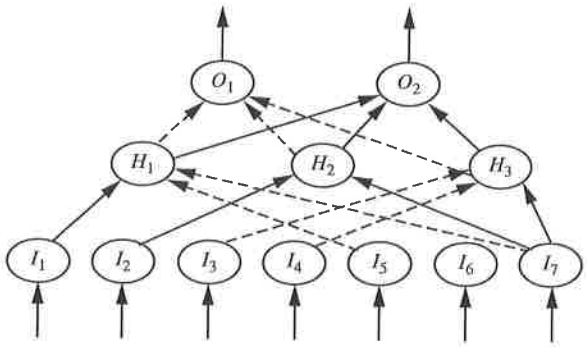
Weight or bias	New value
w_{46}	$-0.3 + (0.9)(0.1311)(0.332) = -0.261$
w_{56}	$-0.2 + (0.9)(0.1311)(0.525) = -0.138$
w_{14}	$0.2 + (0.9)(-0.0087)(1) = 0.192$
w_{15}	$-0.3 + (0.9)(-0.0065)(1) = -0.306$
w_{24}	$0.4 + (0.9)(-0.0087)(0) = 0.4$
w_{25}	$0.1 + (0.9)(-0.0065)(0) = 0.1$
w_{34}	$-0.5 + (0.9)(-0.0087)(1) = -0.508$
w_{35}	$0.2 + (0.9)(-0.0065)(1) = 0.194$
θ_6	$0.1 + (0.9)(0.1311) = 0.218$
θ_5	$0.2 + (0.9)(-0.0065) = 0.194$
θ_4	$-0.4 + (0.9)(-0.0087) = -0.408$

representation. Acquired knowledge in the form of a network of units connected by weighted links is difficult for humans to interpret. This factor has motivated research in extracting the knowledge embedded in trained neural networks and in representing that knowledge symbolically. Methods include extracting rules from networks and sensitivity analysis.

Various algorithms for the extraction of rules have been proposed. The methods typically impose restrictions regarding procedures used in training the given neural network, the network topology, and the discretization of input values.

Fully connected networks are difficult to articulate. Hence, often the first step toward extracting rules from neural networks is **network pruning**. This consists of simplifying the network structure by removing weighted links that have the least effect on the trained network. For example, a weighted link may be deleted if such removal does not result in a decrease in the classification accuracy of the network.

Once the trained network has been pruned, some approaches will then perform link, unit, or activation value clustering. In one method, for example, clustering is used to find the set of common activation values for each hidden unit in a given trained two-layer neural network (Figure 6.19). The combinations of these activation values for each hidden unit are analyzed. Rules are derived relating combinations of activation values with corresponding output unit values. Similarly, the sets of input



Identify sets of common activation values for each hidden node, H_i:
for H_1: (–1,0,1)
for H_2: (0.1)
for H_3: (–1,0.24,1)

Derive rules relating common activation values with output nodes, O_j:
IF ($H_2 = 0$ AND $H_3 = -1$) OR
($H_1 = -1$ AND $H_2 = 1$ AND $H_3 = -1$) OR
($H_1 = -1$ AND $H_2 = 0$ AND $H_3 = 0.24$)
THEN $O_1 = 1$, $O_2 = 0$
ELSE $O_1 = 0$, $O_2 = 1$

Derive rules relating input nodes, I_j, to output nodes, O_j:
IF ($I_2 = 0$ AND $I_7 = 0$) THEN $H_2 = 0$
IF ($I_4 = 1$ AND $I_6 = 1$) THEN $H_3 = -1$
IF ($I_5 = 0$) THEN $H_3 = -1$

Obtain rules relating inputs and output classes:
IF ($I_2 = 0$ AND $I_7 = 0$ AND $I_4 = 1$ AND $I_6 = 1$) THEN class = 1
IF ($I_2 = 0$ AND $I_7 = 0$ AND $I_5 = 0$) THEN class = 1

Figure 6.19 Rules can be extracted from training neural networks. Adapted from [LSL95].

values and activation values are studied to derive rules describing the relationship between the input and hidden unit layers. Finally, the two sets of rules may be combined to form IF-THEN rules. Other algorithms may derive rules of other forms, including M-of-N rules (where M out of a given N conditions in the rule antecedent must be true in order for the rule consequent to be applied), decision trees with M-of-N tests, fuzzy rules, and finite automata.

Sensitivity analysis is used to assess the impact that a given input variable has on a network output. The input to the variable is varied while the remaining input variables are fixed at some value. Meanwhile, changes in the network output are monitored. The knowledge gained from this form of analysis can be represented in rules such as "*IF X decreases 5% THEN Y increases 8%*."

Support Vector Machines

In this section, we study **Support Vector Machines**, a promising new method for the classification of both linear and nonlinear data. In a nutshell, a support vector machine (or SVM) is an algorithm that works as follows. It uses a nonlinear mapping to transform the original training data into a higher dimension. Within this new dimension, it searches for the linear optimal separating hyperplane (that is, a "decision boundary" separating the tuples of one class from another). With an appropriate nonlinear mapping to a sufficiently high dimension, data from two classes can always be separated by a hyperplane. The SVM finds this hyperplane using *support vectors* ("essential" training tuples) and *margins* (defined by the support vectors). We will delve more into these new concepts further below.

"I've heard that SVMs have attracted a great deal of attention lately. Why?" The first paper on support vector machines was presented in 1992 by Vladimir Vapnik and colleagues Bernhard Boser and Isabelle Guyon, although the groundwork for SVMs has been around since the 1960s (including early work by Vapnik and Alexei Chervonenkis on statistical learning theory). Although the training time of even the fastest SVMs can be extremely slow, they are highly accurate, owing to their ability to model complex nonlinear decision boundaries. They are much less prone to overfitting than other methods. The support vectors found also provide a compact description of the learned model. SVMs can be used for prediction as well as classification. They have been applied to a number of areas, including handwritten digit recognition, object recognition, and speaker identification, as well as benchmark time-series prediction tests.

6.7.1 The Case When the Data Are Linearly Separable

To explain the mystery of SVMs, let's first look at the simplest case—a two-class problem where the classes are linearly separable. Let the data set D be given as $(\boldsymbol{X}_1, y_1)$, $(\boldsymbol{X}_2, y_2), \ldots, (\boldsymbol{X}_{|D|}, y_{|D|})$, where $\boldsymbol{X}_i$ is the set of training tuples with associated class labels, y_i. Each y_i can take one of two values, either $+1$ or -1 (i.e., $y_i \in \{+1, -1\}$), corresponding to the classes *buys_computer* = *yes* and *buys_computer* = *no*, respectively. To aid in visualization, let's consider an example based on two input attributes, A_1 and A_2, as shown in Figure 6.20. From the graph, we see that the 2-D data are **linearly separable** (or "linear," for short) because a straight line can be drawn to separate all of the tuples of class $+1$ from all of the tuples of class -1. There are an infinite number of separating lines that could be drawn. We want to find the "best" one, that is, one that (we hope) will have the minimum classification error on previously unseen tuples. How can we find this best line? Note that if our data were 3-D (i.e., with three attributes), we would want to find the best separating *plane*. Generalizing to n dimensions, we want to find the best *hyperplane*. We will use the term "hyperplane" to refer to the decision boundary that we are seeking, regardless of the number of input attributes. So, in other words, how can we find the best hyperplane?

An SVM approaches this problem by searching for the **maximum marginal hyperplane**. Consider Figure 6.21, which shows two possible separating hyperplanes and

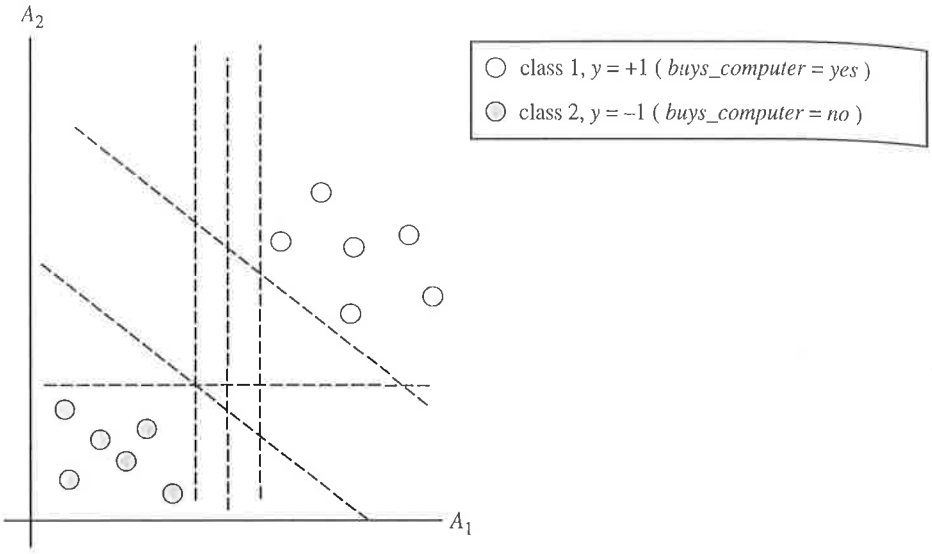


Figure 6.20 The 2-D training data are linearly separable. There are an infinite number of (possible) separating hyperplanes or "decision boundaries." Which one is best?

their associated margins. Before we get into the definition of margins, let's take an intuitive look at this figure. Both hyperplanes can correctly classify all of the given data tuples. Intuitively, however, we expect the hyperplane with the larger margin to be more accurate at classifying future data tuples than the hyperplane with the smaller margin. This is why (during the learning or training phase), the SVM searches for the hyperplane with the largest margin, that is, the *maximum marginal hyperplane* (MMH). The associated margin gives the largest separation between classes. Getting to an informal definition of margin, we can say that the shortest distance from a hyperplane to one side of its margin is equal to the shortest distance from the hyperplane to the other side of its margin, where the "sides" of the margin are parallel to the hyperplane. When dealing with the MMH, this distance is, in fact, the shortest distance from the MMH to the closest training tuple of either class.

A separating hyperplane can be written as

$$\boldsymbol{W} \cdot \boldsymbol{X} + b = 0, \tag{6.32}$$

where $\boldsymbol{W}$ is a weight vector, namely, $\boldsymbol{W} = \{w_1, w_2, \ldots, w_n\}$; n is the number of attributes; and b is a scalar, often referred to as a bias. To aid in visualization, let's consider two input attributes, A_1 and A_2, as in Figure 6.21(b). Training tuples are 2-D, e.g., $\boldsymbol{X} = (x_1, x_2)$, where x_1 and x_2 are the values of attributes A_1 and A_2, respectively, for $\boldsymbol{X}$. If we think of b as an additional weight, w_0, we can rewrite the above separating hyperplane as

$$w_0 + w_1 x_1 + w_2 x_2 = 0. \tag{6.33}$$

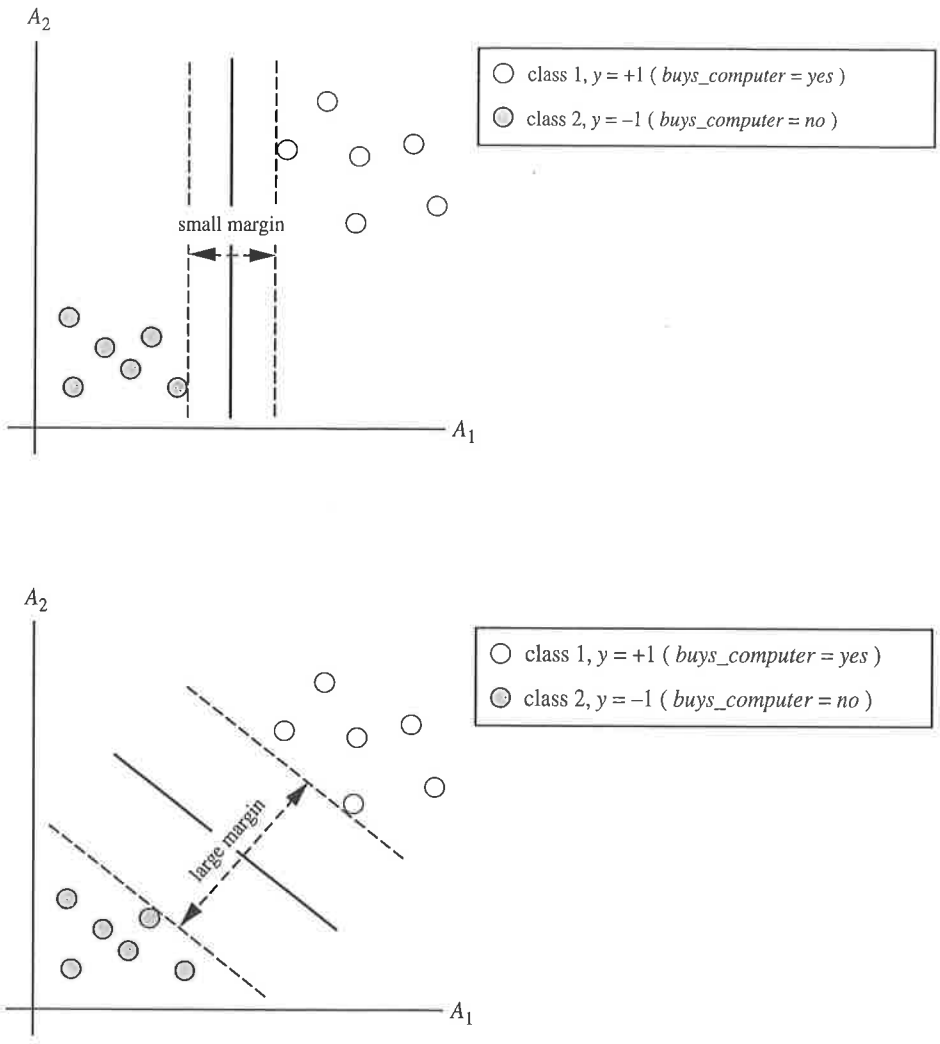


Figure 6.21 Here we see just two possible separating hyperplanes and their associated margins. Which one is better? The one with the larger margin should have greater generalization accuracy.

Thus, any point that lies above the separating hyperplane satisfies

$$w_0 + w_1 x_1 + w_2 x_2 > 0. \tag{6.34}$$

Similarly, any point that lies below the separating hyperplane satisfies

$$w_0 + w_1 x_1 + w_2 x_2 < 0. \tag{6.35}$$

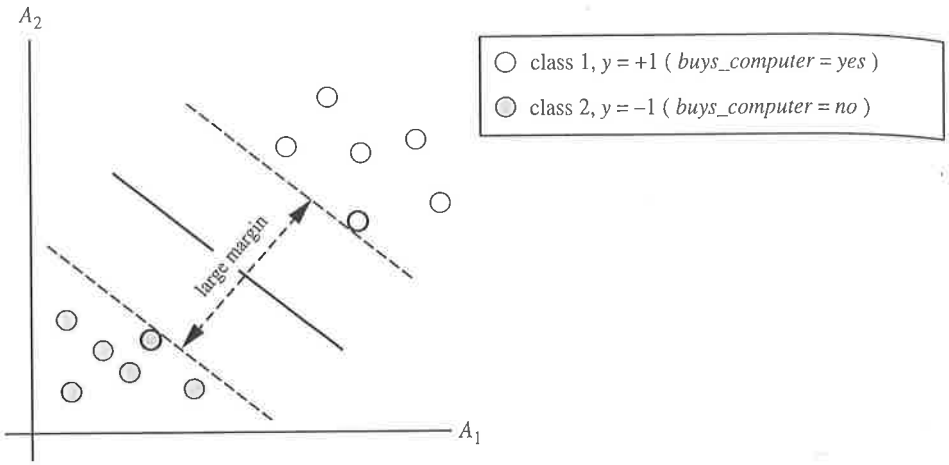


Figure 6.22 Support vectors. The SVM finds the maximum separating hyperplane, that is, the one with maximum distance between the nearest training tuples. The support vectors are shown with a thicker border.

The weights can be adjusted so that the hyperplanes defining the "sides" of the margin can be written as

$$H_1 : w_0 + w_1 x_1 + w_2 x_2 \geq 1 \text{ for } y_i = +1, \text{ and} \tag{6.36}$$
$$H_2 : w_0 + w_1 x_1 + w_2 x_2 \leq -1 \text{ for } y_i = -1. \tag{6.37}$$

That is, any tuple that falls on or above H_1 belongs to class $+1$, and any tuple that falls on or below H_2 belongs to class -1. Combining the two inequalities of Equations (6.36) and (6.37), we get

$$y_i(w_0 + w_1 x_1 + w_2 x_2) \geq 1, \ \forall i. \tag{6.38}$$

Any training tuples that fall on hyperplanes H_1 or H_2 (i.e., the "sides" defining the margin) satisfy Equation (6.38) and are called **support vectors**. That is, they are equally close to the (separating) MMH. In Figure 6.22, the support vectors are shown encircled with a thicker border. Essentially, the support vectors are the most difficult tuples to classify and give the most information regarding classification.

From the above, we can obtain a formulae for the size of the maximal margin. The distance from the separating hyperplane to any point on H_1 is $\frac{1}{||W||}$, where $||W||$ is the Euclidean norm of $\boldsymbol{W}$, that is $\sqrt{\boldsymbol{W} \cdot \boldsymbol{W}}$.[9] By definition, this is equal to the distance from any point on H_2 to the separating hyperplane. Therefore, the maximal margin is $\frac{2}{||W||}$.

"So, how does an SVM find the MMH and the support vectors?" Using some "fancy math tricks," we can rewrite Equation (6.38) so that it becomes what is known as a constrained

[9] If $\boldsymbol{W} = \{w_1, w_2, \ldots, w_n\}$ then $\sqrt{\boldsymbol{W} \cdot \boldsymbol{W}} = \sqrt{w_1^2 + w_2^2 + \cdots + w_n^2}$.

(convex) quadratic optimization problem. Such fancy math tricks are beyond the scope of this book. Advanced readers may be interested to note that the tricks involve rewriting Equation (6.38) using a Lagrangian formulation and then solving for the solution using Karush-Kuhn-Tucker (KKT) conditions. Details can be found in references at the end of this chapter. If the data are small (say, less than 2,000 training tuples), any optimization software package for solving constrained convex quadratic problems can then be used to find the support vectors and MMH. For larger data, special and more efficient algorithms for training SVMs can be used instead, the details of which exceed the scope of this book. Once we've found the support vectors and MMH (note that the support vectors define the MMH!), we have a trained support vector machine. The MMH is a linear class boundary, and so the corresponding SVM can be used to classify linearly separable data. We refer to such a trained SVM as a *linear SVM*.

"Once I've got a trained support vector machine, how do I use it to classify test (i.e., new) tuples?" Based on the Lagrangian formulation mentioned above, the MMH can be rewritten as the decision boundary

$$d(\boldsymbol{X}^T) = \sum_{i=1}^{l} y_i \alpha_i \boldsymbol{X}_i \boldsymbol{X}^T + b_0, \tag{6.39}$$

where y_i is the class label of support vector $\boldsymbol{X}_i$; $\boldsymbol{X}^T$ is a test tuple; α_i and b_0 are numeric parameters that were determined automatically by the optimization or SVM algorithm above; and l is the number of support vectors.

Interested readers may note that the α_i are Lagrangian multipliers. For linearly separable data, the support vectors are a subset of the actual training tuples (although there will be a slight twist regarding this when dealing with nonlinearly separable data, as we shall see below).

Given a test tuple, $\boldsymbol{X}^T$, we plug it into Equation (6.39), and then check to see the sign of the result. This tells us on which side of the hyperplane the test tuple falls. If the sign is positive, then $\boldsymbol{X}^T$ falls on or above the MMH, and so the SVM predicts that $\boldsymbol{X}^T$ belongs to class $+1$ (representing *buys_computer* = *yes*, in our case). If the sign is negative, then $\boldsymbol{X}^T$ falls on or below the MMH and the class prediction is -1 (representing *buys_computer* = *no*).

Notice that the Lagrangian formulation of our problem (Equation (6.39)) contains a dot product between support vector $\boldsymbol{X}_i$ and test tuple $\boldsymbol{X}^T$. This will prove very useful for finding the MMH and support vectors for the case when the given data are nonlinearly separable, as described further below.

Before we move on to the nonlinear case, there are two more important things to note. The complexity of the learned classifier is characterized by the number of support vectors rather than the dimensionality of the data. Hence, SVMs tend to be less prone to overfitting than some other methods. The support vectors are the essential or critical training tuples—they lie closest to the decision boundary (MMH). If all other training tuples were removed and training were repeated, the same separating hyperplane would be found. Furthermore, the number of support vectors found can be used to compute an (upper) bound on the expected error rate of the SVM classifier, which is independent of the data dimensionality. An SVM with a small number of support vectors can have good generalization, even when the dimensionality of the data is high.

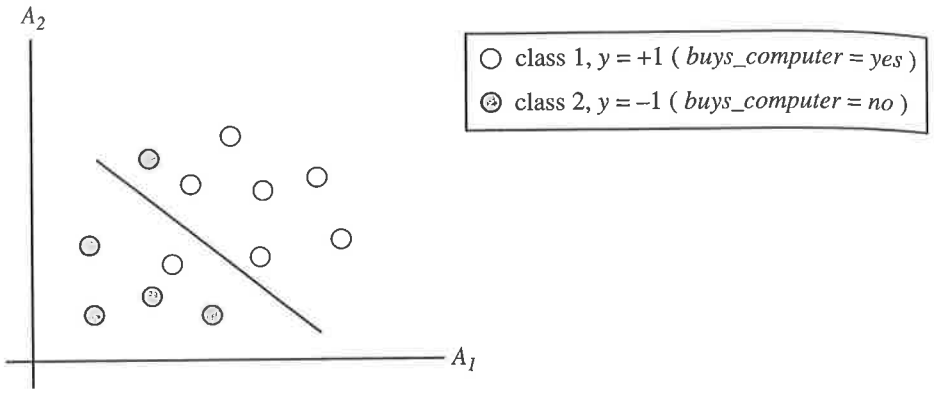


Figure 6.23 A simple 2-D case showing linearly inseparable data. Unlike the linear separable data of Figure 6.20, here it is not possible to draw a straight line to separate the classes. Instead, the decision boundary is nonlinear.

6.7.2 The Case When the Data Are Linearly Inseparable

In Section 6.7.1 we learned about linear SVMs for classifying linearly separable data, but what if the data are not linearly separable, as in Figure 6.23? In such cases, no straight line can be found that would separate the classes. The linear SVMs we studied would not be able to find a feasible solution here. Now what?

The good news is that the approach described for linear SVMs can be extended to create *nonlinear SVMs* for the classification of *linearly inseparable data* (also called *nonlinearly separable data*, or *nonlinear data*, for short). Such SVMs are capable of finding nonlinear decision boundaries (i.e., nonlinear hypersurfaces) in input space.

"*So,*" you may ask, "*how can we extend the linear approach?*" We obtain a nonlinear SVM by extending the approach for linear SVMs as follows. There are two main steps. In the first step, we transform the original input data into a higher dimensional space using a nonlinear mapping. Several common nonlinear mappings can be used in this step, as we will describe further below. Once the data have been transformed into the new higher space, the second step searches for a linear separating hyperplane in the new space. We again end up with a quadratic optimization problem that can be solved using the linear SVM formulation. The maximal marginal hyperplane found in the new space corresponds to a nonlinear separating hypersurface in the original space.

Example 6.10 **Nonlinear transformation of original input data into a higher dimensional space.** Consider the following example. A 3D input vector $X = (x_1, x_2, x_3)$ is mapped into a 6D space, Z, using the mappings $\phi_1(X) = x_1$, $\phi_2(X) = x_2$, $\phi_3(X) = x_3$, $\phi_4(X) = (x_1)^2$, $\phi_5(X) = x_1x_2$, and $\phi_6(X) = x_1x_3$. A decision hyperplane in the new space is $d(Z) = WZ + b$, where W and Z are vectors. This is linear. We solve for W and b and then substitute back

so that the linear decision hyperplane in the new ($\mathbf{Z}$) space corresponds to a nonlinear second-order polynomial in the original 3-D input space,

$$\begin{aligned} d(\mathbf{Z}) &= w_1x_1 + w_2x_2 + w_3x_3 + w_4(x_1)^2 + w_5x_1x_2 + w_6x_1x_3 + b \\ &= w_1z_1 + w_2z_2 + w_3z_3 + w_4z_4 + w_5z_5 + w_6z_6 + b \end{aligned}$$

■

But there are some problems. First, how do we choose the nonlinear mapping to a higher dimensional space? Second, the computation involved will be costly. Refer back to Equation (6.39) for the classification of a test tuple, $\boldsymbol{X^T}$. Given the test tuple, we have to compute its dot product with every one of the support vectors.[10] In training, we have to compute a similar dot product several times in order to find the MMH. This is especially expensive. Hence, the dot product computation required is very heavy and costly. We need another trick!

Luckily, we can use another math trick. It so happens that in solving the quadratic optimization problem of the linear SVM (i.e., when searching for a linear SVM in the new higher dimensional space), the training tuples appear only in the form of dot products, $\phi(\boldsymbol{X_i}) \cdot \phi(\boldsymbol{X_j})$, where $\phi(\boldsymbol{X})$ is simply the nonlinear mapping function applied to transform the training tuples. Instead of computing the dot product on the transformed data tuples, it turns out that it is mathematically equivalent to instead apply a *kernel function*, $K(\boldsymbol{X_i}, \boldsymbol{X_j})$, to the original input data. That is,

$$K(\boldsymbol{X_i}, \boldsymbol{X_j}) = \phi(\boldsymbol{X_i}) \cdot \phi(\boldsymbol{X_j}). \tag{6.40}$$

In other words, everywhere that $\phi(\boldsymbol{X_i}) \cdot \phi(\boldsymbol{X_j})$ appears in the training algorithm, we can replace it with $K(\boldsymbol{X_i}, \boldsymbol{X_j})$. In this way, all calculations are made in the original input space, which is of potentially much lower dimensionality! We can safely avoid the mapping—it turns out that we don't even have to know what the mapping is! We will talk more later about what kinds of functions can be used as kernel functions for this problem.

After applying this trick, we can then proceed to find a maximal separating hyperplane. The procedure is similar to that described in Section 6.7.1, although it involves placing a user-specified upper bound, C, on the Lagrange multipliers, α_i. This upper bound is best determined experimentally.

"*What are some of the kernel functions that could be used?*" Properties of the kinds of kernel functions that could be used to replace the dot product scenario described above have been studied. Three admissible kernel functions include:

$$\text{Polynomial kernel of degree } h: \quad K(\boldsymbol{X_i}, \boldsymbol{X_j}) = (\boldsymbol{X_i} \cdot \boldsymbol{X_j} + 1)^h \tag{6.41}$$

$$\text{Gaussian radial basis function kernel}: \quad K(\boldsymbol{X_i}, \boldsymbol{X_j}) = e^{-\|\boldsymbol{X_i} - \boldsymbol{X_j}\|^2/2\sigma^2} \tag{6.42}$$

$$\text{Sigmoid kernel}: \quad K(\boldsymbol{X_i}, \boldsymbol{X_j}) = \tanh(\kappa \boldsymbol{X_i} \cdot \boldsymbol{X_j} - \delta) \tag{6.43}$$

[10] The dot product of two vectors, $\boldsymbol{X^T} = (x_1^T, x_2^T, \ldots, x_n^T)$ and $\boldsymbol{X_i} = (x_{i1}, x_{i2}, \ldots, x_{in})$ is $x_1^T x_{i1} + x_2^T x_{i2} + \cdots + x_n^T x_{in}$. Note that this involves one multiplication and one addition for each of the n dimensions.

Each of these results in a different nonlinear classifier in (the original) input space. Neural network aficionados will be interested to note that the resulting decision hyperplanes found for nonlinear SVMs are the same type as those found by other well-known neural network classifiers. For instance, an SVM with a Gaussian radial basis function (RBF) gives the same decision hyperplane as a type of neural network known as a radial basis function (RBF) network. An SVM with a sigmoid kernel is equivalent to a simple two-layer neural network known as a multilayer perceptron (with no hidden layers). There are no golden rules for determining which admissible kernel will result in the most accurate SVM. In practice, the kernel chosen does not generally make a large difference in resulting accuracy. SVM training always finds a global solution, unlike neural networks such as backpropagation, where many local minima usually exist (Section 6.6.3).

So far, we have described linear and nonlinear SVMs for binary (i.e., two-class) classification. SVM classifiers can be combined for the multiclass case. A simple and effective approach, given m classes, trains m classifiers, one for each class (where classifier j learns to return a positive value for class j and a negative value for the rest). A test tuple is assigned the class corresponding to the largest positive distance.

Aside from classification, SVMs can also be designed for linear and nonlinear regression. Here, instead of learning to predict discrete class labels (like the $y_i \in \{+1, -1\}$ above), SVMs for regression attempt to learn the input-output relationship between input training tuples, $\boldsymbol{X}_i$, and their corresponding continuous-valued outputs, $y_i \in \mathcal{R}$. An approach similar to SVMs for classification is followed. Additional user-specified parameters are required.

A major research goal regarding SVMs is to improve the speed in training and testing so that SVMs may become a more feasible option for very large data sets (e.g., of millions of support vectors). Other issues include determining the best kernel for a given data set and finding more efficient methods for the multiclass case.

6.8 Associative Classification: Classification by Association Rule Analysis

Frequent patterns and their corresponding association or correlation rules characterize interesting relationships between attribute conditions and class labels, and thus have been recently used for effective classification. Association rules show strong associations between attribute-value pairs (or *items*) that occur frequently in a given data set. Association rules are commonly used to analyze the purchasing patterns of customers in a store. Such analysis is useful in many decision-making processes, such as product placement, catalog design, and cross-marketing. The discovery of association rules is based on *frequent itemset mining*. Many methods for frequent itemset mining and the generation of association rules were described in Chapter 5. In this section, we look at **associative classification**, where association rules are generated and analyzed for use in classification. The general idea is that we can search for strong associations between frequent patterns (conjunctions of attribute-value pairs) and class labels. Because association rules