# Contents

# Chapter 6

# Classification and Prediction

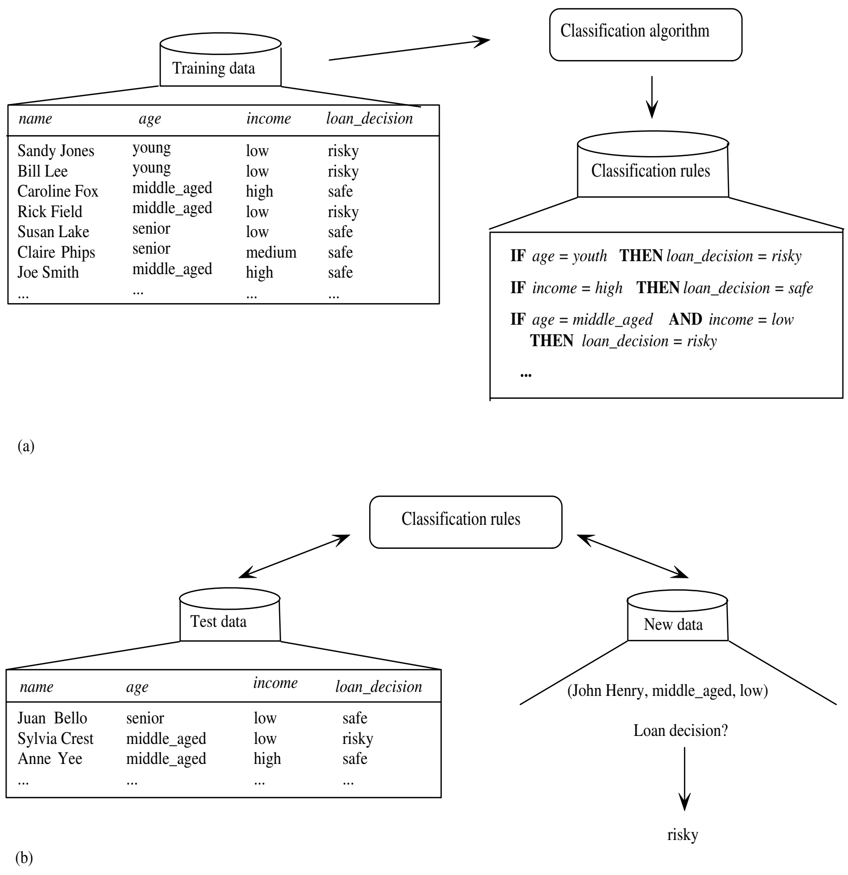
Databases are rich with hidden information that can be used for intelligent decision making. Classification and prediction are two forms of data analysis that can be used to extract models describing important data classes or to predict future data trends. Such analysis can help provide us with a better 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.

## 6.1 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 or not 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, above, where there is no ordering implied among this group of treatment regimes.

Suppose that the marketing manager above 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, since 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*.


Training data
Classification algorithm
Classification rules
**IF** *age = youth* **THEN** *loan_decision = risky*
**IF** *income = high* **THEN** *loan_decision = safe*
**IF** *age = middle_aged* **AND** *income = low* **THEN** *loan_decision = risky*
...
(a)
Classification rules
Test data
New data
(John Henry, middle_aged, low)
Loan decision?
risky
(b)

| *name* | *age* | *income* | *loan_decision* |
|---|---|---|---|
| Sandy Jones | young | low | risky |
| Bill Lee | young | low | risky |
| Caroline Fox | middle_aged | high | safe |
| Rick Field | middle_aged | low | risky |
| Susan Lake | senior | low | safe |
| Claire Phips | senior | medium | safe |
| Joe Smith | middle_aged | high | safe |
| ... | ... | ... | ... |

| *name* | *age* | *income* | *loan_decision* |
|---|---|---|---|
| Juan Bello | senior | low | safe |
| Sylvia Crest | middle_aged | low | risky |
| Anne Yee | middle_aged | high | safe |
| ... | ... | ... | ... |

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.

*"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

[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., $\boldsymbol{X} = (x_1, x_2, x_3)$.

under analysis. In the context of classification, data tuples can be referred to as *samples, examples*, *instances*, *data points*, or *objects.*[2]

Since the class label of each training tuple *is provided*, this step is also known as **supervised learning** (i.e., the learning of the classifier is "supervised" in that it is told to which class each training tuple belongs). It contrasts with **unsupervised learning** (or **clustering**), in which the class label of each training tuple is not known, and the number or set of classes to be learned may not be known in advance. For example, if we did not have the *loan_decision* data available for the training set, we could use clustering to try to determine "groups of like tuples", which may correspond to risk groups within the loan application data. Clustering is the topic of 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 since the classifier tends to **overfit** the data (that is, 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" since 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 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 $\boldsymbol{X}$ 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, $\boldsymbol{X}$. There are various predictor error measures (Section 6.12.2). General methods for error estimation are discussed in Section 6.13.

[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.

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

## 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 in order 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.

  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*. Since 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:

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

- **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. Since 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 flow-chart-like tree structure, where each **internal node** (non-leaf node) denotes a test on an attribute, each **branch** represents an outcome of the test, and each **leaf node** (or *terminal node*) holds a class label. The topmost node in a tree is the **root** node. A typical decision tree is shown in Figure 6.2. It represents the concept *buys_computer*, that is, it predicts whether or not 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), while others can produce non-binary 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.

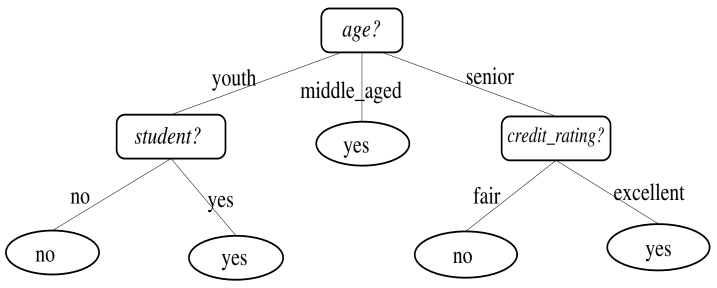


Figure 6.2: A decision tree for the concept *buys_computer*, indicating whether or not a customer at *AllElectronics* is likely to purchase a computer. Each internal (nonleaf) node represents a test on an attribute. Each leaf node represents a class (either *buys_computer* = *yes* or *buys_computer* = *no*).

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., non-backtracking) approach where decision trees are constructed in a top-down recursive divide-and-conquer manner. The vast majority of algorithms for decision tree induction also follow such a top-down approach, which starts with a training set of tuples and their associated class labels. The training set is recursively partitioned into smaller subsets as the tree is being built. A basic decision tree algorithm is summarized in Figure 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 or not 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]

[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

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

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

- If the tuples in $D$ are all of the same class, then node $N$ becomes a leaf and is labeled with that class (steps 2 and 3). Note that steps 4 and 5 are terminating conditions. All 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 10-11). There are three possible scenarios, as illustrated in Figure 6.4. Let $A$ be the splitting attribute. $A$ has $v$ distinct values, $\{a_1, a_2, \cdots, a_v\}$, 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

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.

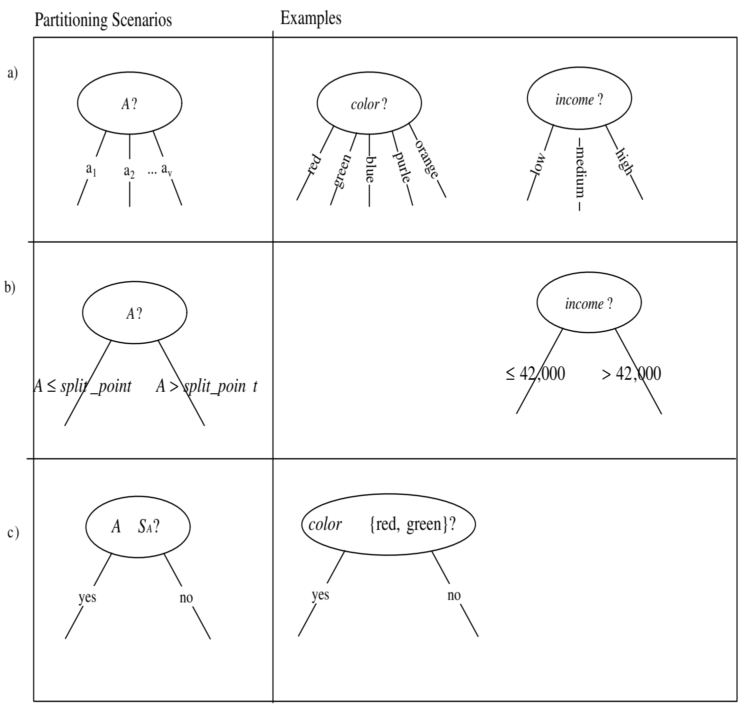


Figure 6.4: Three possibilities for partitioning tuples based on the splitting criterion, shown with examples. Let $A$ be the splitting attribute. (a) If $A$ is discrete-valued then one branch is grown for each known value of $A$. (b) If $A$ is continuous-valued then two branches are grown, corresponding 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 $A$. [TO EDITOR The symbol $\in$ somehow is not appearing in $A \in S_A$. Please fix! Also, two lines from part (a) show up on screen but not in printout. Please check. Thanks!]

values of $A$. A branch is created for each known value, $a_j$, of $A$ and labeled with that value (Figure 6.4a)). Partition $D_j$ is the subset of class-labeled tuples in $D$ having value $a_j$ of $A$. Since 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 8-9).

2. *A is continuous-valued*: In this case, the test at node $N$ has two possible outcomes, corresponding to the conditions $A \leq$ *split_point* and $A >$ *split_point*, respectively, where *split_point* is the split point returned by *Attribute_selection_method* as part of the splitting criterion. (In practice, the split point, $a$, is often taken as the midpoint of two known adjacent values of $A$ and therefore may not actually be a pre-existing value of $A$ from the training data.) Two branches are grown from $N$ and labeled according to the above outcomes (Figure 6.4b)). The tuples are partitioned such that $D_1$ holds the subset of class-labeled tuples in $D$ for which $A \leq$ *split_point*, while $D_2$ holds the rest.

3. *A is discrete-valued* and a *binary tree* must be produced (as dictated by the attribute selection measure or algorithm being used): The test at node $N$ is of the form "$A \in S_A$?". $S_A$ 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 6.4c)). By convention, the left branch out of $N$ is labeled *yes* so that $D_1$ corresponds to the subset of class-labeled tuples in $D$ that satisfy the test. The right branch out of $N$ is labeled *no* so that $D_2$ corresponds to the subset of class-labeled tuples from $D$ that do not satisfy the test.

- The algorithm uses the same process recursively to form a decision tree for the tuples at each resulting partition, $D_j$, of $D$ (step 14).

- The recursive partitioning stops only when any one of the following terminating conditions is true:
  1. All 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 labelling 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, i.e., 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.

**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 (where 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** since 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

[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.

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 since 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, $\{a_1, a_2, \cdots, a_v\}$, 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, $\{D_1, D_2, \cdots, D_v\}$, where $D_j$ contains those tuples in $D$ that have outcome $a_j$ of $A$. These partitions would correspond to the branches grown from node $N$. Ideally, we would like this partitioning to produce an exact classification of the tuples. That is, we would like for each partition to be pure. However, it is quite likely that the partitions will be impure, e.g., where a partition may contain a collection of tuples from different classes rather than from a single class. How much more information would we still need (after the partitioning) 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 $j$th partition. $Info_A(D)$ is the expected information required to classify a tuple from $D$ based on the partitioning by $A$. The smaller the expected information (still) required, the greater the purity of the partitions.

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

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

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 9 tuples of class *yes* and 5 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 $D$:

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

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*

| *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 |

Table 6.1: Class-labeled training tuples from the *AllElectronics* customer database.

category *"youth"*, there are 2 *yes* tuples and 3 *no* tuples. For the category *"middle_aged"*, there are 4 *yes* tuples and 0 *no* tuples. For the category *"senior"*, there are 3 *yes* tuples and 2 *no* tuples. Using Equation (6.2), the expected information needed to classify a tuple in $D$ if the tuples are partitioned according to *age* is

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

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

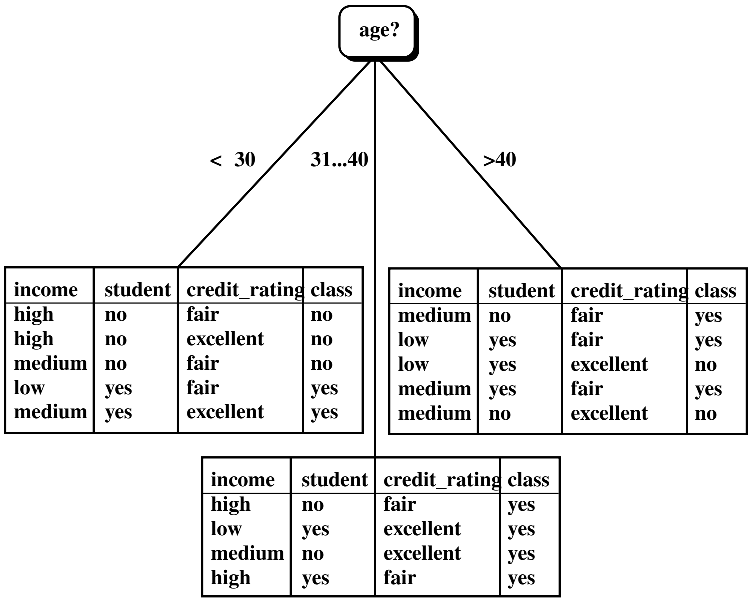
$$Gain(age) = Info(D) - Info_{age}(D) = 0.940 - 0.694 = 0.246 \text{ bits.}$$

Similarly, we can compute $Gain(income)$ = 0.029 bits, $Gain(student)$ = 0.151 bits, and $Gain(credit\_rating)$ = 0.048 bits. Since *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. Since 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. ■

*"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

$$\frac{a_i + a_{i+1}}{2}. \qquad (6.4)$$

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, i.e.,



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

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

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

Figure 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. [TO EDITOR Please replace "< 30" with *youth*, "31...40" with *middle_aged*, and "> 40" with *senior*. Please italicize *age, income, student, credit_rating*, and *class*. If possible please organize the three branches (tables) so that they all line up at the same level. Note, some rows from middle branch may not be showing up on printout. Please verify with actual figure file. ]

$v = 2$ (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 towards 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. Since 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

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

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 4, 6, and 4 tuples, respectively. To compute the gain ratio of *income*, we first use Equation (6.5) to obtain

$$\begin{aligned} SplitInfo_A(D) &= -\frac{4}{14} \times \log_2\left(\frac{4}{14}\right) - \frac{6}{14} \times \log_2\left(\frac{6}{14}\right) - \frac{4}{14} \times \log_2\left(\frac{4}{14}\right) \\ &= 0.926. \end{aligned}$$

From Example 6.1, we have *Gain(income)* = 0.029. Therefore, *GainRatio(income)* = 0.029/0.926 = 0.013. ■

## 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, $\{a_1, a_2, \cdots, a_v\}$, 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 9 tuples belonging to the class *buys_computer = yes* and the remaining 5 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* ∈ {*low, medium*}". The remaining 4 tuples of $D$ would be assigned to partition $D_2$. The gini index value computed based on this partitioning is

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

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* ∈ {*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 (non-binary) 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 towards 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 towards 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 $\chi^2$ test for independence. Other

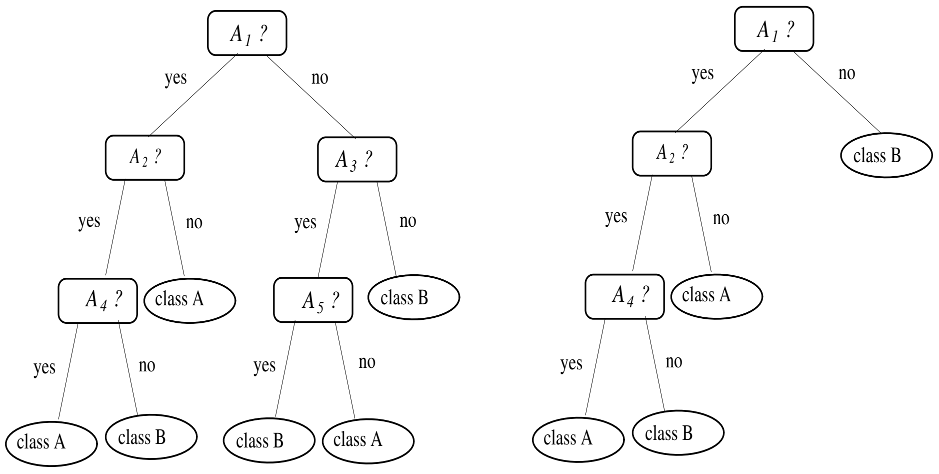


Figure 6.6: An unpruned decision tree, and a pruned version of it.

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 $\chi^2$ distribution).

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

Other attribute selection measures consider **multivariate splits**, i.e., where the partitioning of tuples is based on a *combination* of attributes, rather than on a single attribute. The CART system, for example, can find multivariate splits based on a linear combination of attributes. Multivariate splits are a form of **attribute** (or feature) **construction**, where new attributes are created based on the existing ones. (Attribute construction 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. In spite of 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). 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, while 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. While 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

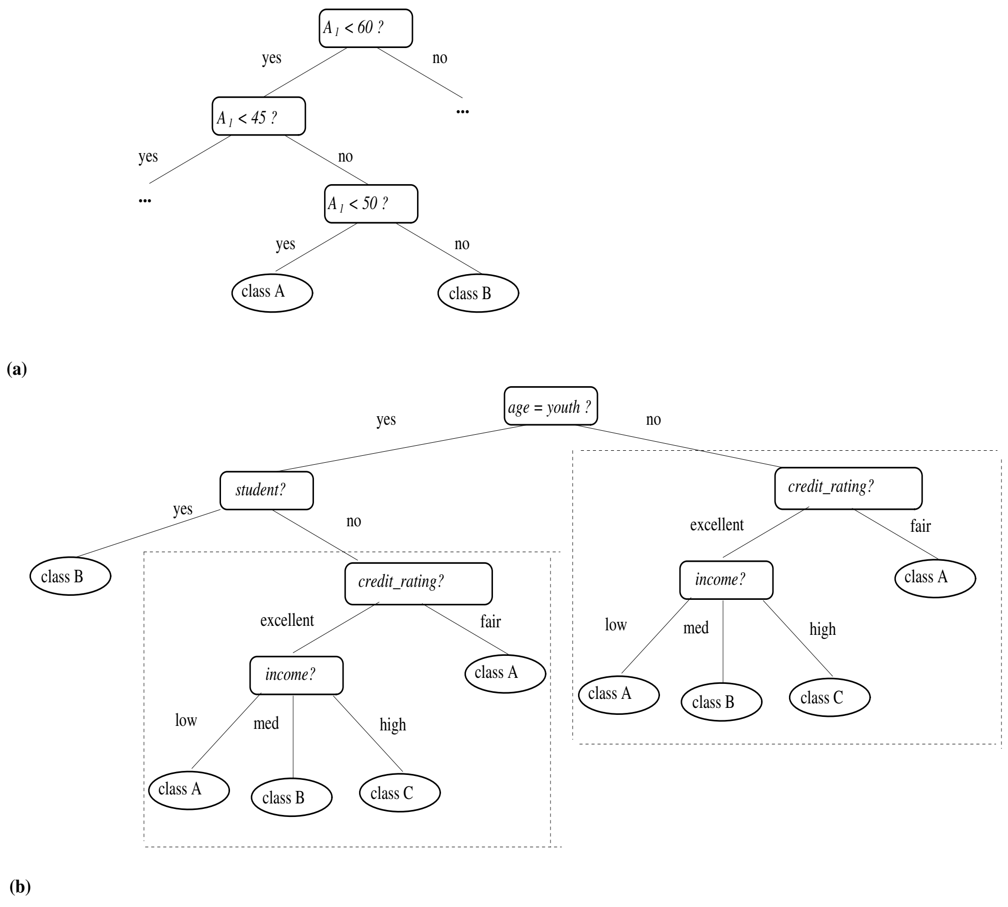


Figure 6.7: An example of subtree (a) **repetition** (where an attribute is repeatedly tested along a given branch of the tree, e.g., *age*) and (b) **replication** (where duplicate subtrees exist within a tree, such as the subtree headed by the node "*credit_rating?*").

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, however, still assume that the training set can fit in memory.

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 in the decision tree. The class list remains in memory since 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

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

Table 6.2: Tuple data for the class *buys_computer*.

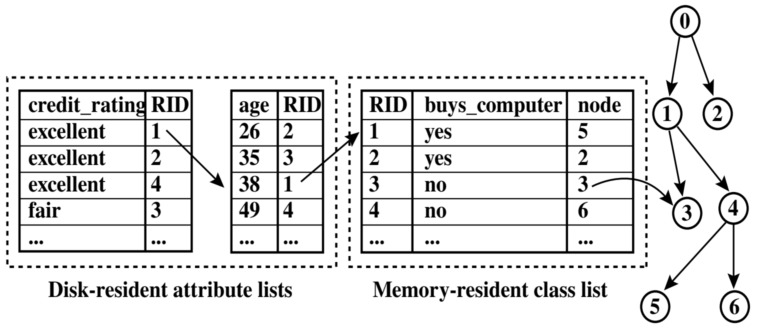


Figure 6.8: Attribute list and class list data structures used in SLIQ for the tuple data of Table 6.2. [TO EDITOR Please italicize *credit_rating, age, RID, buys_computer* and *node.*]

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 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 district 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 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 than the "real" tree, $T$, that would have been generated using the entire data. It refines $T'$ in order to arrive at $T$.

| 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. [TO EDITOR Please italicize *credit_rating, age, RID* and *buys_computer.*]

| age | buys_computer | |
|---|---|---|
| | yes | no |
| youth | 2 | 3 |
| middle_age | 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. [TO EDITOR Please italicize *age, income, student, credit_rating* and *buys_computer.*]

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.

Naive 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 "naive." *Bayesian belief networks* are graphical models, which unlike naive 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 naive 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 $P(H|\mathbf{X})$, the probability that the hypothesis $H$ holds given the "evidence" or observed data tuple $\boldsymbol{X}$. In other words, we are looking for the probability that tuple $\boldsymbol{X}$ belongs to class $C$, given that we know the attribute description of $\boldsymbol{X}$.

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

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

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

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

*"How are these probabilities estimated?"* $P(H)$, and $P(\boldsymbol{X}|H)$, and $P(\boldsymbol{X})$ 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, $P(H|\boldsymbol{X})$, from $P(H)$, $P(\boldsymbol{X}|H)$, and $P(\boldsymbol{X})$. Bayes' theorem is

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

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 Naive Bayesian Classification

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

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

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

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

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

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

3. As $P(\boldsymbol{X})$ is constant for all classes, only $P(\boldsymbol{X}|C_i)P(C_i)$ need be maximized. If the class prior probabilities are not known, then it is commonly assumed that the classes are equally likely, that is, $P(C_1) = P(C_2) = \ldots = P(C_m)$, 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,

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

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 $\mu$ and standard deviation $\sigma$, defined by

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

   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 $\mu_{C_i}$ and $\sigma_{C_i}$, which are the mean (i.e., average) and standard deviation, respectively, of the values of attribute $A_k$ for training tuples of class $C_i$. We then plug these two quantities into Equation (6.13), together with $x_k$, in order to estimate $P(x_k|C_i)$. For example, let $\boldsymbol{X} = (35, \$40K)$, where $A_1$ and $A_2$ are the attributes *age* and *income*, respectively. Let the class label attribute be *buys_computer.* The associated class label for $\boldsymbol{X}$ is *"yes"*, i.e., *buys_computer = yes.* Let's suppose that *age* has not been discretized and therefore exists as a continuous-valued attribute. Suppose that from the training set, we find that customers in $D$ who buy a computer are $38 \pm 12$ years of age. In other words, for attribute *age* and this class, we have $\mu = 38$ years and $\sigma = 12$. We can plug these quantities, along with $x_1 = 35$ for our tuple $\boldsymbol{X}$ into 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 $\boldsymbol{X}$, $P(\boldsymbol{X}|C_i)P(C_i)$ is evaluated for each class $C_i$. The classifier predicts that the class label of tuple $\boldsymbol{X}$ is the class $C_i$ if and only if

$$P(\boldsymbol{X}|C_i)P(C_i) > P(\boldsymbol{X}|C_j)P(C_j) \quad \text{for} \quad 1 \le j \le m, j \ne i. \tag{6.15}$$

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

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

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

**Example 6.4 Predicting a class label using naive Bayesian classification.** We wish to predict the class label of a tuple using naive Bayesian classification, given the same training data as in Example 6.1 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

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

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

$$\begin{aligned} P(buys\_computer = yes) &= 9/14 = 0.643 \\ P(buys\_computer = no) &= 5/14 = 0.357 \end{aligned}$$

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

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

Using the above probabilities, we obtain

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

Similarly,

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

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

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

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

*"What if I encounter probability values of zero?"* Recall that in Equation (6.12), we estimate $P(\boldsymbol{X}|C_i)$ as the product of the probabilities $P(x_1|C_i), P(x_2|C_i), \ldots, P(x_n|C_i)$, based on the assumption of class conditional independence. These probabilities can be estimated from the training tuples (step 4). We need to compute $P(\boldsymbol{X}|C_i)$ for *each* class ($i = 1, 2, \ldots, m$) in order to find the class $C_i$ for which $P(\boldsymbol{X}|C_i)P(C_i)$ is the maximum (step 5). Let's consider this calculation. For each attribute-value pair (i.e., $A_k = x_k$, for $k = 1, 2, \ldots, n$) in tuple $\boldsymbol{X}$, we need to count the number of tuples having that attribute-value pair, per class (i.e., per $C_i$, for $i = 1, \ldots, m$). In Example 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 $\boldsymbol{X}$, say, we need two counts—the number of customers who are students and for which *buys_computer = yes* (which contributes to $P(\boldsymbol{X}|buys\_computer = yes)$), and the number of customers who are students and for which *buys_computer = no* (which contributes to $P(\boldsymbol{X}|buys\_computer = no)$). But what if, say, there are no training tuples representing students for the class *buys_computer = no*, resulting in

$P(student = yes|buys\_computer = no) = 0$? In other words, what happens if we should end up with a probability value of zero for some $P(x_k|C_i)$? Plugging this zero value into Equation (6.12) would return a zero probability for $P(\boldsymbol{X}|C_i)$, even though, without the zero probability, we may have ended up with a high probability, suggesting that $\boldsymbol{X}$ belonged to class $C_i$! A zero probability cancels the effects of all 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 so 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 1749-1827. 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):

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

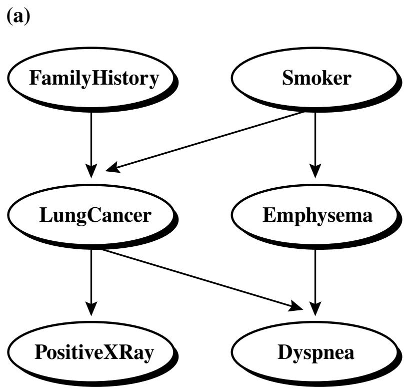
respectively. The "corrected" probability estimates are close to their "uncorrected" counterparts, yet the zero probability value is avoided. ■

### 6.4.3 Bayesian Belief Networks

The naive 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 naive 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 **descendent** of $Y$. *Each variable is conditionally independent of its nondescendents 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 *PostiveXRay* 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 any additional information regarding *PositiveXRay.* The arcs also show



(b)

|  | FH, S | FH, ~S | ~FH, S | ~FH, ~S |
|---|---|---|---|---|
| LC | 0.8 | 0.5 | 0.7 | 0.1 |
| ~LC | 0.2 | 0.5 | 0.3 | 0.9 |

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]. [TO EDITOR Please italicize the text in each node as they are attribute names, along with their corresponding abbreviations (LC, FH, etc.).]

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 | FamilyHistory =yes Smoker = yes") = 0.8*
*P(LungCancer = no | FamilyHistory = no, Smoker = no") = 0.9*

Let $\boldsymbol{X} = (x_1, \ldots, x_n)$ 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 nondescendents 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 $\boldsymbol{X}$, 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 towards 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.

## 6.5 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 extracted, either from a decision tree or directly from the training data.

### 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, $X$, 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 or not 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, $X$. If a rule is satisfied by $X$, 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 $X$ according to *buys_computer*. $X$ satisfies $R1$, which triggers the rule.

If $R1$ is the only rule satisfied, then the rule **fires** by returning the class prediction for $X$. Note that triggering does not always mean firing because they 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 $X$?

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 $X$. 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 the misclassification cost per class. Within each class, the rules are not ordered—they don't have to be since 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 $X$ 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 $X$. How, then, can we determine the class label of $X$? In this case, a fallback or **default rule** can be set up to specify a default class, based on a training set. This may be the class in majority, or the majority class of the tuples that were not covered by any rule. The default rule is evaluated at the end, if and only if no other rule covers $X$. The condition in the default rule is empty. In this way, the rule fires when no other rule is satisfied.

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

### 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. Since 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.*

[FROM MK: **Changes made in this and next paragraph:**] 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 a 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, since 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, since 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, since 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 Extraction from the Training Data

IF-THEN rules can be extracted directly from the training data (that is, 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.

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. Since 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.*

**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.

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 rule for a class, so that different rules many 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 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

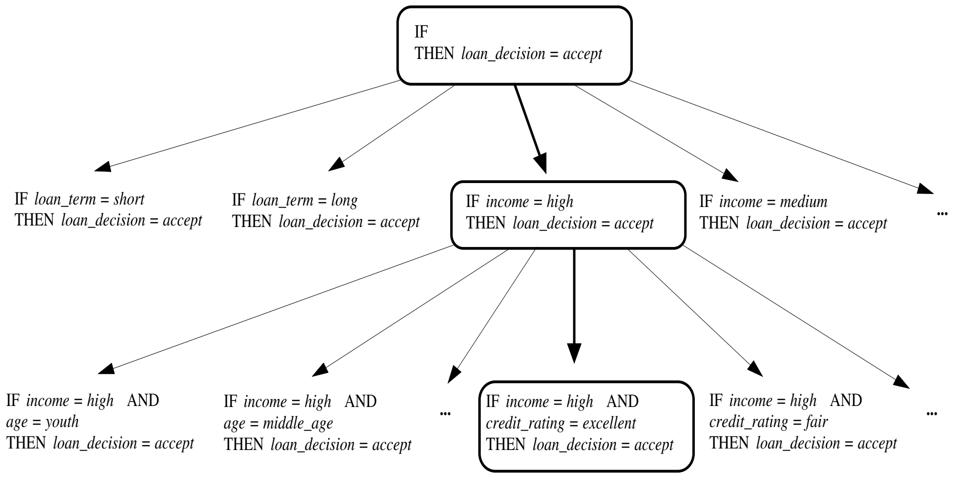

IF
THEN loan_decision = accept
IF loan_term = short THEN loan_decision = accept
IF loan_term = long THEN loan_decision = accept
IF income = high THEN loan_decision = accept
IF income = medium THEN loan_decision = accept
...
IF income = high AND age = youth THEN loan_decision = accept
IF income = high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

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

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 $R1$ correctly classifies 38 of the 40 tuples it covers. Rule $R2$ covers only 2 tuples, which it correctly classifies. Their respective accuracies are 95% and 100%. Thus, $R2$ has greater accuracy than $R1$, however, it is not the better rule because of its small coverage. ∎

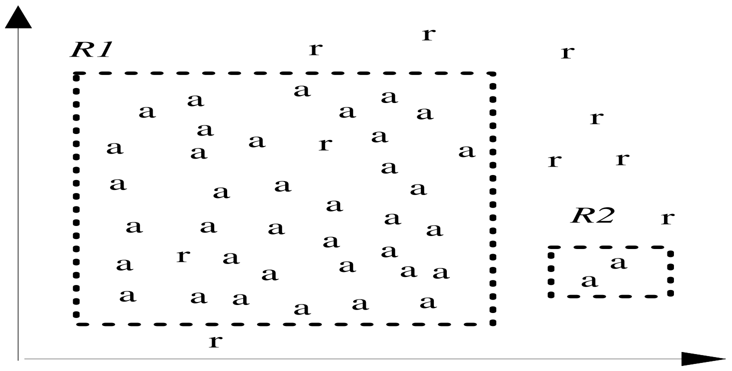


Figure 6.14: Rules rules for the class *loan_decision = accept*, showing *accept* (*a*) and *reject* (*r*) tuples.

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 $R$.

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, 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 (that is, 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

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

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 asses whether any observed differences between these two distributions may be attributed to chance. We can use the **likelihood ratio statistic**,

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

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

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

statistic has a $\chi^2$ 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. Such assessment is optimistic since 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.

## 6.6 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 where 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 to 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, ranging from handwritten character recognition, pathology and laboratory medicine, to 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 towards 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.

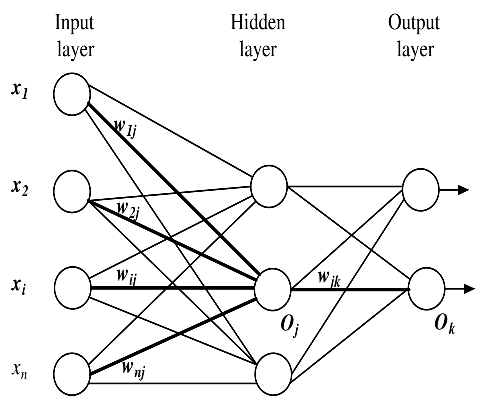


Figure 6.15: A multilayer feed-forward neural network:

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 of output units. Therefore, we say that it is a **two-layer** neural network. (The input layer is not counted since 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 (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.

**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, $\boldsymbol{X}$, 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, 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 $\theta_j$ is the **bias** of the unit. The bias acts as a threshold in that it serves to vary the activity of the unit.

**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 $X$ 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 $\theta_j$ 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.

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 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*, since 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 backwards 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.

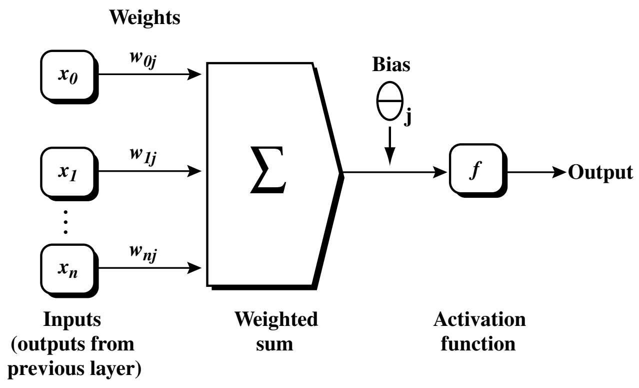


Figure 6.17: A hidden or output layer unit $j$: The inputs to unit $j$ are outputs from the previous layer. These are multiplied by their corresponding weights in order to form a weighted sum, which is added to the bias associated with unit $j$. A nonlinear activation function is applied to the net input. (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 $(x_1, x_2, \ldots, x_n)$.)

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 $\Delta w_{ij}$ 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 to avoid getting stuck at a local minimum 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 $\theta_j$:

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

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

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

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 since it tends to yield more accurate results.

**Terminating condition:** Training stops when

- all $\Delta w_{ij}$ 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, then 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 assures 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, $\boldsymbol{X}$=(1, 0, 1), whose class label is 1.

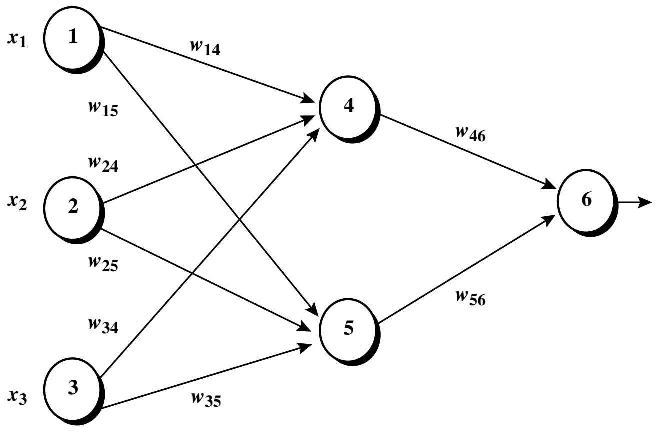


Figure 6.18: An example of a multilayer feed-forward neural network.

Table 6.3: Initial input, weight, and bias values.

| $x_1$ | $x_2$ | $x_3$ | $w_{14}$ | $w_{15}$ | $w_{24}$ | $w_{25}$ | $w_{34}$ | $w_{35}$ | $w_{46}$ | $w_{56}$ | $\theta_4$ | $\theta_5$ | $\theta_6$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

This example shows the calculations for backpropagation, given the first training tuple, $\boldsymbol{X}$. 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 and propagated backwards. The error values are shown in Table 6.5. The weight and bias updates are shown in Table 6.6. ■

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 | $(-0.3)(0.332) - (0.2)(0.525) + 0.1 = -0.105$ | $1/(1 + e^{0.105}) = 0.474$ |

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 | $(0.332)(1 - 0.332)(0.1311)(-0.3) = -0.0087$ |

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 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.

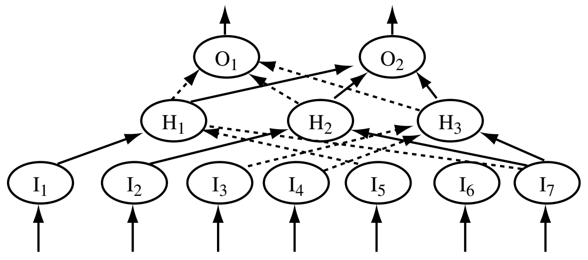
Various algorithms for the extraction of rules have been proposed. The methods typically impose restrictions regarding procedures used in training the given neural network, the network topology, and the discretization of input values.

Fully connected networks are difficult to articulate. Hence, often the first step towards 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.

Table 6.6: Calculations for weight and bias updating.

| Weight or bias | New value |
|---|---|
| $w_{46}$ | $-0.3 + (0.9)(0.1311)(0.332) = -0.261$ |
| $w_{56}$ | $-0.2 + (0.9)(0.1311)(0.525) = -0.138$ |
| $w_{14}$ | $0.2 + (0.9)(-0.0087)(1) = 0.192$ |
| $w_{15}$ | $-0.3 + (0.9)(-0.0065)(1) = -0.306$ |
| $w_{24}$ | $0.4 + (0.9)(-0.0087)(0) = 0.4$ |
| $w_{25}$ | $0.1 + (0.9)(-0.0065)(0) = 0.1$ |
| $w_{34}$ | $-0.5 + (0.9)(-0.0087)(1) = -0.508$ |
| $w_{35}$ | $0.2 + (0.9)(-0.0065)(1) = 0.194$ |
| $\theta_6$ | $0.1 + (0.9)(0.1311) = 0.218$ |
| $\theta_5$ | $0.2 + (0.9)(-0.0065) = 0.194$ |
| $\theta_4$ | $-0.4 + (0.9)(-0.0087) = -0.408$ |

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 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.



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, $I_i$, 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. [TO EDITOR For consistency (for logic terms), please replace all 5 incidences of lower case "and" (i.e., in second box) by upper case"AND". In addition, all $O_1, H_1, I_1$, etc. should be italicized.] Adapted from [LSL95].

**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%".

## 6.7 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 them 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, speaker identification, as well as benchmark time series prediction tests.

### 6.7.1 The Case When the Data are Linearly Separable

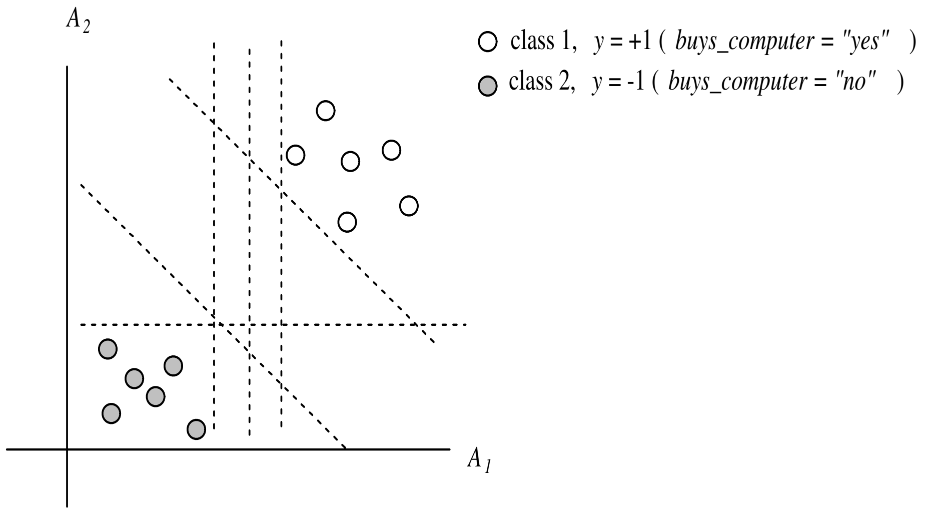
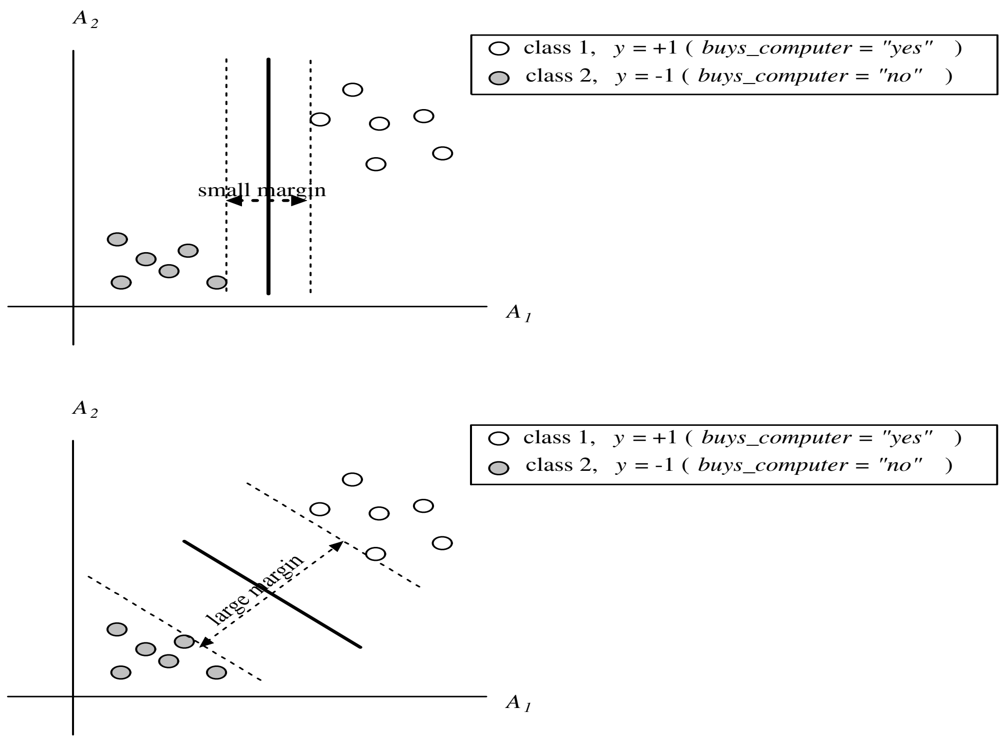

$A_2$
$A_1$
○ class 1, $y = +1$ ( *buys_computer* = "yes" )
● class 2, $y = -1$ ( *buys_computer* = "no" )

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?

To explain the mystery of SVMs, let's first have a look at the simplest case—a two-class problem where the classes are linearly separable. Let the data set $D$ be given as $(\boldsymbol{X_1}, y_1)$, $(\boldsymbol{X_2}, y_2)$, ..., $(\boldsymbol{X_{|D|}}, y_{|D|})$, 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) since 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?



$A_2$
class 1, $y$ = +1 ( *buys_computer* = "yes" )
class 2, $y$ = -1 ( *buys_computer* = "no" )
small margin
$A_1$
$A_2$
class 1, $y$ = +1 ( *buys_computer* = "yes" )
class 2, $y$ = -1 ( *buys_computer* = "no" )
large margin
$A_1$

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.

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

$$w_0 + w_1x_1 + w_2x_2 = 0. \tag{6.33}$$

Thus, any point that lies above the separating hyperplane satisfies

$$w_0 + w_1x_1 + w_2x_2 > 0. \tag{6.34}$$

Similarly, any point that lies below the separating hyperplane satisfies

$$w_0 + w_1x_1 + w_2x_2 < 0. \tag{6.35}$$

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

$$H_1 : w_0 + w_1x_1 + w_2x_2 \geq 1 \text{ for } y_i = +1, \text{ and} \tag{6.36}$$

$$H_2 : w_0 + w_1x_1 + w_2x_2 \leq -1 \text{ for } y_i = -1. \tag{6.37}$$

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

$$y_i(w_0 + w_1x_1 + w_2x_2) \geq 1, \ \forall i. \tag{6.38}$$

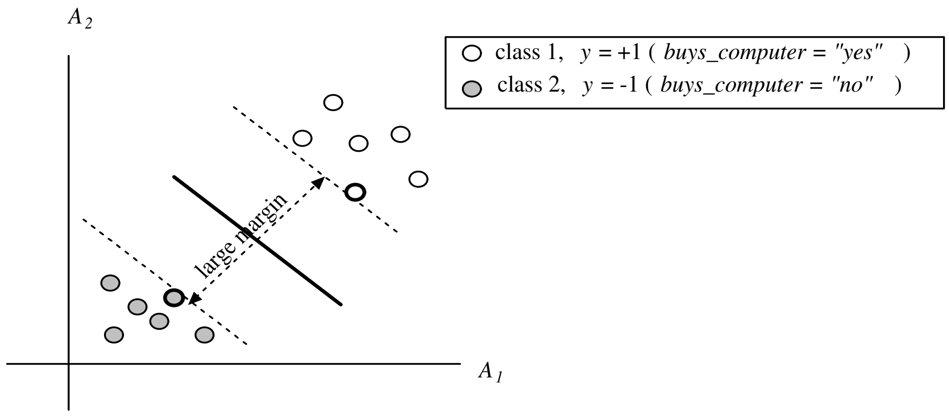

$A_2$
○ class 1, $y = +1$ ( *buys_computer* = "yes" )
● class 2, $y = -1$ ( *buys_computer* = "no" )
large margin
$A_1$

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.

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 $\sqrt{\boldsymbol{W} \cdot \boldsymbol{W}}$. By definition, this is equal to the distance from any point on $H_2$ to the separating hyperplane. Therefore, the maximal margin is $\frac{2}{||\boldsymbol{W}||}$.

*"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 (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 maximum marginal hyperplane 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, $\alpha_i$ 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 $\alpha_i$ 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!).

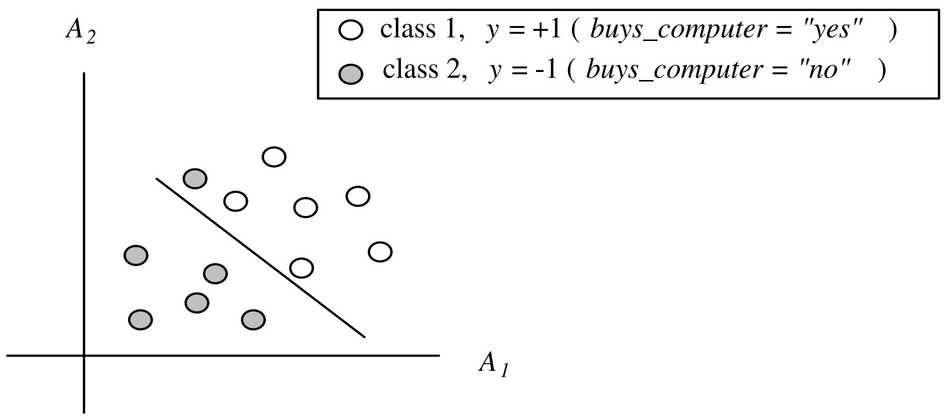
Given a test tuple, $\boldsymbol{X^T}$, we plug it into Equation (6.39), and then check to see the sign of the result. This tells us on which side of the hyperplane the test tuple falls. If the sign is positive, then $\boldsymbol{X^T}$ falls on or above the MMH and so the SVM predicts that $\boldsymbol{X^T}$ 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.

### 6.7.2 The Case When the Data Are Linearly Inseparable

In Section 6.7.1 above 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, there is no straight line that can be found that would



A_2
A_1
○ class 1, y = +1 ( buys_computer = "yes" )
● class 2, y = -1 ( buys_computer = "no" )

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.

separate the classes. The linear SVMs we studied above would not be able to find a feasible solution here. Now what?

The good news is that the approach described above 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 (that is, 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 $\boldsymbol{X} = (x_1, x_2, x_3)$ is mapped into a 6D space, $Z$, using the mappings $\phi_1(\boldsymbol{X}) = x_1, \phi_2(\boldsymbol{X}) = x_2, \phi_3(\boldsymbol{X}) = x_3$, $\phi_4(\boldsymbol{X}) = (x_1)^2, \phi_5(\boldsymbol{X}) = x_1x_2$, and $\phi_6(\boldsymbol{X}) = x_1x_3$. A decision hyperplane in the new space is $d(\boldsymbol{Z})= \boldsymbol{W}\boldsymbol{Z} + b$, where $\boldsymbol{W}$ and $\boldsymbol{Z}$ are vectors. This is linear. We solve for $\boldsymbol{W}$ and $b$ and then substitute back so that the linear decision hyperplane in the new ($\boldsymbol{Z}$) space corresponds to a nonlinear second order polynomial in the original 3-D input space,

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

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 each and every one of the support vectors.[9] In training, we have to compute a similar dot product several times in order to find the maximal marginal hypersurface. This is especially expensive. Hence, the dot product computation required is very heavy and costly. We need another trick!

Luckily, there is another math trick that we can use. 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

[9]The dot product of two vectors, $\boldsymbol{X^T} = (x_1^T, x_2^T, \ldots, x_n^T)$ and $\boldsymbol{X_i} = (x_{i,1}, x_{i,2}, \ldots, x_{i,n})$ is $x_1^T x_{i,1} + x_2^T x_{i,2} + \ldots + x_n^T x_{i,n}$. Note that this involves one multiplication and one addition for each of the $n$ dimensions.

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, $\alpha_i$, above. 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:

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

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

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

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, a 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. A SVM with a sigmoid kernel is equivalent to a simple 2-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 find 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. Since association rules explore highly confident associations among multiple attributes, this approach may overcome some constraints introduced by decision-tree induction, which considers only one attribute at a time. In many studies, associative classification has been found to be more accurate than some traditional classification methods, such as C4.5. In particular, we study three main methods: CBA, CMAR, and CPAR.

Before we begin, let's take a look at association rule mining, in general. Association rules are mined in a two-step process consisting of *frequent itemset mining*, followed by *rule generation.* The first step searches for patterns of attribute-value pairs that occur repeatedly in a data set, where each attribute-value pair is considered an *item.* The resulting attribute-value pairs form *frequent itemsets.* The second step analyzes the frequent itemsets in order to generate association rule. All association rules must satisfy certain criteria regarding their "accuracy" (or *confidence*) and the proportion of the data set that they actually represent (referred to as *support*). For example, the following is an association rule mined from a data set, $D$, shown with its confidence and support.

$$age = youth \wedge credit = OK \Rightarrow buys\_computer = yes \quad [support = 20\%, confidence = 93\%] \tag{6.44}$$

where "$\wedge$" represents a logical "AND". We will say more about confidence and support in a minute.

More formally, let $D$ be a data set of tuples. Each tuple in $D$ is described by $n$ attributes, $A_1, A_2, .., A_n$, and a class label attribute, $A_{class}$. All continuous attributes are discretized and treated as categorical attributes. An **item**, $p$, is an attribute-value pair of the form $(A_i, v)$, where $A_i$ is an attribute taking a value, $v$. A data tuple $\boldsymbol{X} = (x_1, x_2, \ldots, x_n)$ satisfies an item, $p = (A_i, v)$, if and only if $x_i = v$, where $x_i$ is the value of the $i$th attribute of $\boldsymbol{X}$. Association rules can have any number of items in the rule antecedent (left-hand side) and any number of items in the rule consequent (right-hand side). However, when mining association rules for use in classification, we are only interested in association rules of the form $p_1 \wedge p_2 \wedge \ldots p_l \rightarrow A_{class} = C$ where the rule antecedent is a conjunction of items, $p_1, p_2, \ldots, p_l$ $(l \leq n)$, associated with a class label, $C$. For a given rule, $R$, the percentage of tuples in $D$ satisfying the rule antecedent that also have the class label $C$ is called the **confidence** of $R$. For example, a confidence of 93% for Association Rule (6.44) means that 93% of the customers in $D$ who are young and have an OK credit rating belong to the class *buys_computer = yes.* The percentage of tuples in $D$ satisfying the rule antecedent and having class label $C$ is called the **support** of $R$. A support of 20% for Association Rule (6.44) means that 20% of the customers in $D$ are young, have an OK credit rating, and belong to the class *buys_computer = yes.*

Methods of associative classification differ primarily in the approach used for frequent itemset mining and in how the derived rules are analyzed and used for classification. We now look at some of the various methods for associative classification.

One of the earliest and simplest algorithms for associative classification is CBA (Classification-Based Association). CBA uses an iterative approach to frequent itemset mining, similar to that described for Apriori in Section 5.2.1, where multiple passes are made over the data and the derived frequent itemsets are used to generate and test longer itemsets. In general, the number of passes made is equal to the length of the longest rule found. The complete set of rules satisfying minimum confidence and minimum support thresholds are found and then analyzed for inclusion in the classifier. CBA uses a heuristic method to construct the classifier, where the rules are organized according to decreasing precedence based on their confidence and support. If a set of rules have the same antecedent, then the rule with the highest confidence is selected to represent the set. When classifying a new tuple, the first rule satisfying the tuple is used to classify it. The classifier also contains a default rule, having lowest precedence, which specifies a default class for any new tuple that is not satisfied by any other rule in the classifier. In this way, the set of rules making up the classifier form a *decision list.* In general, CBA was empirically found to be more accurate than C4.5 on a good number of data sets.

CMAR (Classification based on Multiple Association Rules) differs from CBA in its strategy for frequent itemset mining and its construction of the classifier. It also employs several rule pruning strategies with the help of a tree-structure for efficient storage and retrieval of rules. CMAR adopts a variant of the *FP-growth* algorithm to find the complete set of rules satisfying the minimum confidence and minimum support thresholds. FP-growth was described in Section 5.2.4. FP-growth uses a tree structure, called an *FP-tree*, to register all of the frequent itemset information contained in the given data set, $D$. This requires only two scans of $D$. The frequent itemsets are then mined from the FP-tree. CMAR uses an enhanced FP-tree that maintains the distribution of class labels among tuples satisfying each frequent itemset. In this way, it is able to combine rule generation together with frequent itemset mining in a single step.

CMAR also employs a different tree structure to store and retrieve rules efficiently, and to prune rules based on confidence, correlation, and database coverage. Rule pruning strategies are triggered whenever a rule is inserted into the tree. For example, given two rules, $R1$ and $R2$, if the antecedent of $R1$ is more general than that of $R2$ and $conf(R1) \geq conf(R2)$, then $R2$ is pruned. The rationale is that highly specialized rules with low confidence can be pruned if a more generalized version with higher confidence exists. CMAR also prunes rules for which the rule antecedent and class are not positively correlated, based on a $\chi^2$ test of statistical significance.

As a classifier, CMAR operates differently than CBA. Suppose that we are given a tuple $X$ to classify and that only one rule satisfies or matches $X$.[10] This case is trivial—we simply assign the class label of the rule. Suppose, instead, that more than one rule satisfies $X$. These rules form a set, $S$. Which rule would we use to determine the class label of $X$? CBA would assign the class label of the most confident rule among the rule set, $S$. CMAR instead considers multiple rules when making its class prediction. It divides the rules into groups according to class labels. All rules within a group share the same class label and each group has a distinct class label. CMAR uses a weighted $\chi^2$ measure to find the "strongest" group of rules, based on the statistical correlation of rules within a group. It then assigns $X$ the class label of the strongest group. In this way it considers multiple rules, rather than a single rule with highest confidence, when predicting the class label of a new tuple. On experiments, CMAR had slightly higher average accuracy in comparison with CBA. Its runtime, scalability, and use of memory was found to be more efficient.

CBA and CMAR adopt methods of frequent itemset mining to generate *candidate* association rules, which include all conjunctions of attribute-value pairs (items) satisfying minimum support. These rules are then examined, and a subset is chosen to represent the classifier. However, such methods generate quite a large number of rules. CPAR takes a different approach to rule generation, based on a rule generation algorithm for classification known as FOIL (First Order Inductive Learner) proposed in Quinlan and Cameron-Jones. FOIL builds rules to distinguish positive tuples (say, having class *buys_computer = yes*) from negative tuples (such as *buys_computer = no*). For multiclass problems, FOIL is applied to each class. That is, for a class, $C$, all tuples of class $C$ are considered positive tuples, while the rest are considered negative tuples. Rules are generated to distinguish $C$ tuples from all others. Each time a rule is generated, the positive samples it satisfies (or *covers*) are removed until all the positive tuples in the data set are covered. CPAR relaxes this step by allowing the covered tuples to remain under consideration, but reducing their 'weight'. The process is repeated for each class. The resulting rules are merged to form the classifier rule set.

During classification, CPAR employs a somewhat different multiple rule strategy than CMAR. If more than one rule satisfies a new tuple, $\boldsymbol{X}$, the rules are divided into groups according to class, similar to CMAR. However, CPAR uses the best $k$ rules of each group to predict the class label of $\boldsymbol{X}$, based on expected accuracy. By considering the best $k$ rules rather than all of the rules of a group, it avoids the influence of lower ranked rules. The accuracy of CPAR on numerous data sets was shown to be close to that of CMAR. However, since CPAR generates far fewer rules than CMAR, it shows much better efficiency with large sets of training data.

In summary, associative classification offers a new alternative to classification schemes by building rules based

[10]If the antecedent of a rule satisfies or matches $X$, then we say that the rule satisfies $X$.

on conjunctions of attribute-value pairs that occur frequently in data.

## 6.9 Lazy Learners (or Learning from Your Neighbors)

The classification methods discussed so far in this chapter—decision tree induction, Bayesian classification, rule-based classification, classification by backpropagation, support vector machines, and classification based on association rule mining—are all examples of *eager learners.* **Eager learners**, when given a set of training tuples, will construct a generalization (i.e., classification) model before receiving new (e.g., test) tuples to classify. We can think of the learned model as being ready and eager to classify previously unseen tuples.

Imagine a contrasting "lazy" approach, in which the learner instead waits until the last minute before doing any model contruction in order to classify a given test tuple. That is, when given a training tuple, a **lazy learner** simply stores it (or does only a little minor processing) and waits until it is given a test tuple. Only when it sees the test tuple does it perform generalization in order to classify the tuple based on its similarity to the stored training tuples. Unlike eager learning methods, lazy learners do less work when a training tuple is presented and more work when making a classification or prediction. Because lazy learners store the training tuples or "instances", they are also referred to as **instance-based learners**, even though all learning is essentially based on instances.

When making a classification or prediction, lazy learners can be quite computationally expensive. They require efficient storage techniques and are well-suited to implementation on parallel hardware. They offer little explanation or insight into the structure of the data. Lazy learners, however, naturally support incremental learning. They are able to model complex decision spaces having hyper-polygonal shapes that may not be as easily describable by other learning algorithms (such as hyper-rectangular shapes modeled by decision trees). In this section, we look at two examples of lazy learners: *k-nearest-neighbor classifiers* and *case-based reasoning classifiers.*

### 6.9.1 *k*-Nearest Neighbor Classifiers

The $k$-nearest neighbor method was first described in the early 1950s. The method is labor intensive when given large training sets, and did not gain popularity until the 1960s when increased computing power became available. It has since been widely used in the area of pattern recognition.

Nearest neighbor classifiers are based on learning by analogy, that is, by comparing a given test tuple with training tuples that are similar to it. The training tuples are described by $n$ attributes. Each tuple represents a point in an $n$-dimensional space. In this way, all of the training tuples are stored in an $n$-dimensional pattern space. When given an unknown tuple, a ***k*-nearest neighbor classifier** searches the pattern space for the $k$ training tuples that are closest to the unknown tuple. These $k$ training tuples are the $k$ "nearest neighbors" of the unknown tuple.

"Closeness" is defined in terms of a distance metric, such as Euclidean distance. The Euclidean distance between two points or tuples, say, $\boldsymbol{X_1}=(x_{11}, x_{12}, \ldots, x_{1n})$ and $\boldsymbol{X_2}=(x_{21}, x_{22}, \ldots, x_{2n})$, is

$$dist(\boldsymbol{X_1}, \boldsymbol{X_2}) = \sqrt{\sum_{i=1}^{n} (x_{1i} - x_{2i})^2}. \tag{6.45}$$

In other words, for each numeric attribute, we take the difference between the corresponding values of that attribute in tuple $\boldsymbol{X_1}$ and in tuple $\boldsymbol{X_2}$, square this difference, and accumulate it to the square of the distance count, $dist(\boldsymbol{X_1}, \boldsymbol{X_2})$. Typically, we normalize the values of each attribute, prior to using Equation (6.45). This helps prevent attributes with initially large ranges (such as *income*) from outweighing attributes with initially smaller ranges (such as binary attributes). Min-max normalization, for example, can be used to transform a value $v$ of a numeric attribute $A$ to $v'$ in the range $[0, 1]$ by computing

$$v' = \frac{v - min_A}{max_A - min_A}, \tag{6.46}$$

where $min_A$ and $max_A$ are the minimum and maximum values of attribute $A$. Chapter 2 describes also other methods for data normalization as a form of data transformation.

For $k$-nearest neighbor classification, the unknown tuple is assigned the most common class among its $k$ nearest neighbors. When $k = 1$, the unknown tuple is assigned the class of the training tuple that is closest to it in pattern space. Nearest neighbor classifiers can also be used for prediction, that is, to return a real-valued prediction for a given unknown tuple. In this case, the classifier returns the average value of the real-valued labels associated with the $k$ nearest neighbors of the unknown tuple.

*"But how can distance be computed for attributes that not numeric, but categorical, such as color?"* The above discussion assumes that the attributes used to describe the tuples are all numeric. For categorical attributes, one simple method is to compare the corresponding value of the attribute in tuple $\boldsymbol{X_1}$ with that in tuple $\boldsymbol{X_2}$. If the two are identical (e.g., tuples $\boldsymbol{X_1}$ and $\boldsymbol{X_2}$ both have the color blue), then the difference between the two is taken as 0. If the two are different (e.g., tuple $\boldsymbol{X_1}$ is blue but tuple $\boldsymbol{X_2}$ is red), then the difference is considered to be 1. Other methods may incorporate more sophisticated schemes for differential grading, e.g., where a larger difference score is assigned, say, for blue and white than for blue and black.

*"What about missing values?"* In general, if the value of a given attribute $A$ is missing in tuple $\boldsymbol{X_1}$ and/or in tuple $\boldsymbol{X_2}$, we assume the maximum possible difference. Suppose that each of the attributes have been mapped to the range $[0, 1]$. For categorical attributes, we take the difference value to be 1 if either one or both of the corresponding values of $A$ are missing. If $A$ is numeric and missing from both tuples $\boldsymbol{X_1}$ and $\boldsymbol{X_2}$, then the difference is also taken to be 1. If only one value is missing and the other (which we'll call $v'$) is present and normalized, then we can take the difference to be either $|1 - v'|$ or $|0 - v'|$ (i.e., $1 - v'$ or $v'$), whichever is greater.

*"How can I determine a good value for k, the number of neighbors?"* This can be determined experimentally. Starting with $k = 1$, we use a test set to estimate the error rate of the classifier. This process can be repeated, each time by incrementing $k$ to allow for one more neighbor. The $k$ giving the minimum error rate may be selected. In general, the larger the number of training tuples is, the larger the value of $k$ will be (so that classification and prediction decisions can be based on a larger portion of the stored tuples). As the number of training tuples approaches infinity and $k = 1$, the error rate can be no worse then twice the Bayes error rate (the latter being the theoretical minimum). If $k$ also approaches infinity, the error rate approaches the Bayes error rate.

Nearest neighbor classifiers use distance-based comparisons that intrinsically assign equal weight to each attribute. They therefore can suffer from poor accuracy when given noisy or irrelevant attributes. The method, however, has been modified to incorporate attribute weighting and the pruning of noisy data tuples. The choice of a distance metric can be critical. The Manhattan (city block) distance (Section 7.2.1) or other distance measurements, may also be used.

Nearest-neighbor classifiers can be extremely slow when classifying test tuples. If $D$ is a training database of $|D|$ tuples and $k = 1$, then $O(|D|)$ comparisons are required in order to classify a given test tuple. By presorting and arranging the stored tuples into search trees, the number of comparisons can be reduced to $O(log(|D|)$. Parallel implementation can reduce the running time to a constant, that is $O(1)$, which is independent of $|D|$. Other techniques to speed up classification time include the use of *partial distance* calculations and *editing* the stored tuples. In the **partial distance** method, we compute the distance based on a subset of the $n$ attributes. If this distance exceeds a threshold then further computation for the given stored tuple is halted, and the process moves on to the next stored tuple. The **editing** method removes training tuples that prove "useless". This method is also referred to as **pruning** or **condensing** since it reduces the total number of tuples stored.

### 6.9.2 Case-Based Reasoning

**Case-based reasoning** (CBR) classifiers use a database of problem solutions to solve new problems. Unlike nearest neighbor classifiers, which store training tuples as points in Euclidean space, CBR stores the tuples or "cases" for problem-solving as complex symbolic descriptions. Business applications of CBR include problem resolution for customer service help desks, where cases describe product-related diagnostic problems. CBR has also been applied to areas such as engineering and law, where cases are either technical designs or legal rulings, respectively. Medical education is another area for CBR, where patient case histories and treatments are used to help diagnose and treat new patients.

When given a new case to classify, a case-based reasoner will first check if an identical training case exists. If one is found, then the accompanying solution to that case is returned. If no identical case is found, then the case-based reasoner will search for training cases having components that are similar to those of the new case. Conceptually, these training cases may be considered as neighbors of the new case. If cases are represented as graphs, this involves searching for subgraphs that are similar to subgraphs within the new case. The case-based reasoner tries to combine the solutions of the neighboring training cases in order to propose a solution for the new case. If incompatibilities arise with the individual solutions, then backtracking to search for other solutions may be necessary. The case-based reasoner may employ background knowledge and problem-solving strategies in order to propose a feasible combined solution.

Challenges in case-based reasoning include finding a good similarity metric (e.g., for matching subgraphs) and suitable methods for combining solutions. Other challenges include the selection of salient features for indexing training cases and the development of efficient indexing techniques. A trade-off between accuracy and efficiency evolves as the number of stored cases becomes very large. As this number increases, the case-based reasoner becomes more intelligent. After a certain point, however, the efficiency of the system will suffer as the time required to search for and process relevant cases increases. As with nearest neighbor classifiers, one solution is to edit the training database. Cases that are redundant or that have not proved useful may be discarded for the sake of improved performance. These decisions, however, are not clear-cut and their automation remains an active area of research.

## 6.10 Other Classification Methods

In this section, we give a brief description of a number of other classification methods. These methods include genetic algorithms, rough set, and fuzzy set approaches. In general, these methods are less commonly used for classification in commercial data mining systems than the methods described earlier in this chapter. However, these methods do show their strength in certain applications, and hence it is worthwhile to include them here.

### 6.10.1 Genetic Algorithms

**Genetic algorithms** attempt to incorporate ideas of natural evolution. In general, genetic learning starts as follows. An initial **population** is created consisting of randomly generated rules. Each rule can be represented by a string of bits. As a simple example, suppose that samples in a given training set are described by two Boolean attributes, $A_1$ and $A_2$, and that there are two classes, $C_1$ and $C_2$. The rule "*IF* $A_1$ *AND NOT* $A_2$ *THEN* $C_2$" can be encoded as the bit string "100", where the two leftmost bits represent attributes $A_1$ and $A_2$, respectively, and the rightmost bit represents the class. Similarly, the rule "*IF NOT* $A_1$ *AND NOT* $A_2$ *THEN* $C_1$" can be encoded as "001". If an attribute has $k$ values, where $k > 2$, then $k$ bits may be used to encode the attribute's values. Classes can be encoded in a similar fashion.

Based on the notion of survival of the fittest, a new population is formed to consist of the *fittest* rules in the current population, as well as *offspring* of these rules. Typically, the **fitness** of a rule is assessed by its classification accuracy on a set of training samples.

Offspring are created by applying genetic operators such as crossover and mutation. In **crossover**, substrings from pairs of rules are swapped to form new pairs of rules. In **mutation**, randomly selected bits in a rule's string are inverted.

The process of generating new populations based on prior populations of rules continues until a population, $P$, "evolves" where each rule in $P$ satisfies a prespecified fitness threshold.

Genetic algorithms are easily parallelizable and have been used for classification as well as other optimization problems. In data mining, they may be used to evaluate the fitness of other algorithms.

### 6.10.2 Rough Set Approach

Rough set theory can be used for classification to discover structural relationships within imprecise or noisy data. It applies to discrete-valued attributes. Continuous-valued attributes must therefore be discretized prior to its use.

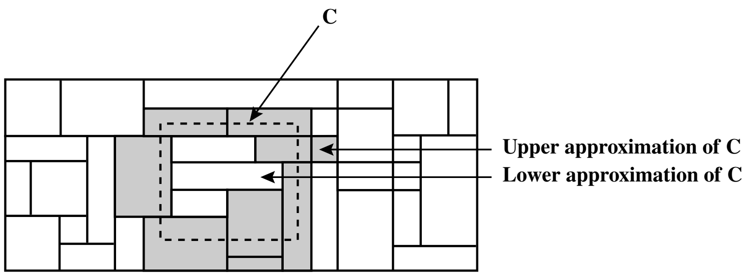


Figure 6.24: A rough set approximation of the set of tuples of the class $C$ using lower and upper approximation sets of $C$. The rectangular regions represent equivalence classes.

Rough set theory is based on the establishment of **equivalence classes** within the given training data. All of the data tuples forming an equivalence class are indiscernible, that is, the samples are identical with respect to the attributes describing the data. Given real-world data, it is common that some classes cannot be distinguished in terms of the available attributes. Rough sets can be used to approximately or "roughly" define such classes. A rough set definition for a given class, $C$, is approximated by two sets—a **lower approximation** of $C$ and an **upper approximation** of $C$. The lower approximation of $C$ consists of all of the data tuples that, based on the knowledge of the attributes, are certain to belong to $C$ without ambiguity. The upper approximation of $C$ consists of all of the tuples that, based on the knowledge of the attributes, cannot be described as not belonging to $C$. The lower and upper approximations for a class $C$ are shown in Figure 6.24, where each rectangular region represents an equivalence class. Decision rules can be generated for each class. Typically, a decision table is used to represent the rules.

Rough sets can also be used for attribute subset selection (or feature reduction, where attributes that do not contribute towards the classification of the given training data can be identified and removed) and relevance analysis (where the contribution or significance of each attribute is assessed with respect to the classification task). The problem of finding the minimal subsets (**reducts**) of attributes that can describe all of the concepts in the given data set is NP-hard. However, algorithms to reduce the computation intensity have been proposed. In one method, for example, a **discernibility matrix** is used that stores the differences between attribute values for each pair of data tuples. Rather than searching on the entire training set, the matrix is instead searched to detect redundant attributes.

### 6.10.3 Fuzzy Set Approaches

Rule-based systems for classification have the disadvantage that they involve sharp cutoffs for continuous attributes. For example, consider the following rule for customer credit application approval. The rule essentially says that applications for customers who have had a job for two or more years and who have a high income (i.e., of at least \$50K) are approved:

$$IF\ (years\_employed \geq 2)\ AND\ (income \geq 50K)\ THEN\ credit = approved. \tag{6.47}$$

By Rule (6.47), a customer who has had a job for at least two years will receive credit if her income is, say, \$50K, but not if it is \$49K. Such harsh thresholding may seem unfair. Instead, we can discretize *income* into categories such as $\{$*low_income, medium_income, high_income*$\}$, and then apply **fuzzy logic** to allow "fuzzy" thresholds or boundaries to be defined for each category (Figure 6.25). Rather than having a precise cutoff between categories, fuzzy logic uses truth values between 0.0 and 1.0 to represent the degree of membership that a certain value has in a given category. Each category then represents a **fuzzy set**. Hence, with fuzzy logic, we can capture the notion that an income of \$49K is, more or less, high, although not as high as an income of \$50K. Fuzzy logic systems typically provide graphical tools to assist users in converting attribute values to fuzzy truth values.

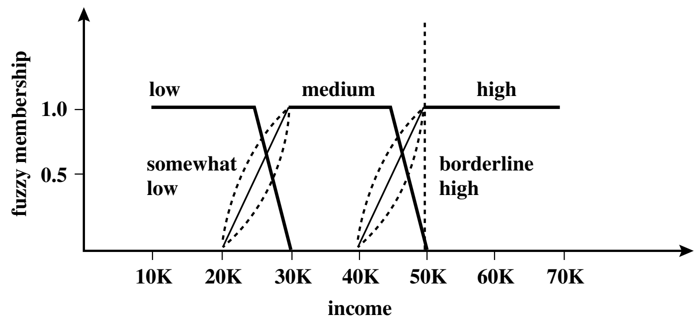


Figure 6.25: Fuzzy truth values for *income*, representing the degree of membership of *income* values with respect to the categories {*low, medium, high*}. Each category represents a fuzzy set. Note that a given income value, $x$, can have membership in more than one fuzzy set. The membership values of $x$ in each fuzzy set do not have to total to 1. [TO EDITOR The figure must be redone: 1) The two upwards sloping lines (i.e., leading the 'medium' and 'high' curves) must be AS THICK as the rest of the lines defining 'medium' and 'high'. 2) Remove the dashed lines (4 curved ones and 1 vertical one). 3) Show ticks and labels for the value 0 on both axes. 4) Remove labels 'somewhat low' and 'borderline high'. 5) Please italicize *income.* ]

Fuzzy set theory is also known as **possibility theory**. It was proposed by Lotfi Zadeh in 1965 as an alternative to traditional two-value logic and probability theory. It lets us work at a high level of abstraction and offers a means for dealing with imprecise measurement of data. Most importantly, fuzzy set theory allows us to deal with vague or inexact facts. For example, being a member of a set of high incomes is inexact (e.g., if \$50K is high, then what about \$49K? Or \$48K?) Unlike the notion of traditional "crisp" sets where an element either belongs to a set $S$ or its complement, in fuzzy set theory, elements can belong to more than one fuzzy set. For example, the income value \$49K belongs to both the *medium* and *high* fuzzy sets, but to differing degrees. Using fuzzy set notation and following Figure 6.25, this can be shown as

$$m_{medium\_income}(\$49K) = 0.15 \text{ and } m_{high\_income}(\$49K) = 0.96,$$

where $m$ denotes the membership function, operating on the fuzzy sets of *medium_income* and *high_income*, respectively. In fuzzy set theory, membership values for a given element, $x$, (e.g., such as for \$49K) do not have to sum to 1. This is unlike traditional probability theory, which is constrained by a summation axiom.

Fuzzy set theory is useful for data mining systems performing rule-based classification. It provides operations for combining fuzzy measurements. Suppose that in addition to the fuzzy sets for *income*, we defined the fuzzy sets *junior_employee* and *senior_employee* for the attribute *years_employed.* Suppose also that we have a rule that, say, tests *high_income* and *senior_employee* in the rule antecedent (IF part) for a given employee, $x$. If these two fuzzy measures are ANDed together, the minimum of their measure is taken as the measure of the rule. In other words,

$$m_{(high\_income\ AND\ senior\_employee)}(x) = min(m_{high\_income}(x), m_{senior\_employee}(x)).$$

This is akin to saying that a chain is as strong as its weakest link. If the two measures are ORed, the maximum of their measure is taken as the measure of the rule. In other words,

$$m_{(high\_income\ OR\ senior\_employee)}(x) = max(m_{high\_income}(x), m_{senior\_employee}(x)).$$

Intuitively, this is like saying that a rope is as strong as its strongest strand.

Given a tuple to classify, more than one fuzzy rule may apply. Each applicable rule contributes a vote for membership in the categories. Typically, the truth values for each predicted category are summed, and these sums are combined. Several procedures exist for translating the resulting fuzzy output into a *defuzzified* or crisp value that is returned by the system.

Fuzzy logic systems have been used in numerous areas for classification, including market research, finance, health care, and environmental engineering.

## 6.11 Prediction

*"What if we would like to predict a continuous value, rather than a categorical label?"* Numeric prediction is the task of predicting continuous (or ordered) values for given input. For example, we may like to predict the salary of college graduates with 10 years of work experience, or the potential sales of a new product given its price. By far, the most widely used approach for numeric prediction (hereafter referred to as prediction) is **regression**, a statistical methodology that was developed by Sir Frances Galton (1822-1911), a mathematician who was also a cousin of Charles Darwin. In fact, many texts use the terms "regression" and "numeric prediction" synonymously. However, as we have seen, some classification techniques (such as backpropagation, support vector machines and $k$-nearest neighbor classifiers) can be adapted for prediction. In this section, we discuss the use of regression techniques for prediction.

Regression analysis can be used to model the relationship between one or more *independent* or **predictor** variables and a *dependent* or **response** variable (which is continuous-valued). In the context of data mining, the predictor variables are the attributes of interest describing the tuple (i.e., making up the attribute vector). In general, the values of the predictor variables are known. (Techniques exist for handling cases where such values may be missing.) The response variable is what we want to predict—it is what we referred to in Section 6.1 as the predicted attribute. Given a tuple described by predictor variables, we want to predict the associated value of the response variable. Regression analysis is a good choice when all of the predictor variables are continuous-valued as well. Many problems can be solved by *linear regression*, and even more can be tackled by applying transformations to the variables so that a nonlinear problem can be converted to a linear one. For reasons of space, we cannot give a fully detailed treatment of regression. Instead, this section provides an intuitive introduction to the topic. Section 6.11.1 discusses straight-line regression analysis (which involves a single predictor variable) and multiple linear regression analysis (which involves two or more predictor variables). Section 6.11.2 provides some pointers on dealing with nonlinear regression. Section 6.11.3 mentions other regression-based methods such as generalized linear models, Poisson regression, log-linear models, and regression trees.

Several software packages exist to solve regression problems. Examples include SAS (*http://www.sas.com*), SPSS (*http://www.spss.com*), and S-Plus (*http://www.insightful.com*). Another useful resource is the book *Numerical Recipes in C* by Press, Flannery, Teukolsky, and Vetterling and its associated source code.

### 6.11.1 Linear Regression

**Straight-line regression analysis** involves a response variable, $y$, and a single predictor variable, $x$. It is the simplest form of regression, and models $y$ as a linear function of $x$. That is,

$$y = b + wx, \tag{6.48}$$

where the variance of $y$ is assumed to be constant, and $b$ and $w$ are **regression coefficients** specifying the Y-intercept and slope of the line, respectively. The regression coefficients, $w$ and $b$, can also be thought of as weights, so that we can equivalently write,

$$y = w_0 + w_1 x. \tag{6.49}$$

These coefficients can be solved for by the **method of least squares**, which estimates the best-fitting straight line as the one that minimizes the error between the actual data and the estimate of the line. Let $D$ be a training set consisting of values of predictor variable, $x$, for some population and their associated values for response variable, $y$.

The training set contains $|D|$ data points of the form $(x_1, y_1)$, $(x_2, y_2), \ldots, (x_{|D|}, y_{|D|})$.[11] The regression coefficients can be estimated using this method with the following equations:

$$w_1 = \frac{\sum_{i=1}^{|D|}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{|D|}(x_i - \bar{x})^2} \tag{6.50}$$

$$w_0 = \bar{y} - w_1\bar{x} \tag{6.51}$$

where $\bar{x}$ is the mean value of $x_1, x_2, \ldots, x_{|D|}$, and $\bar{y}$ is the mean value of $y_1, y_2, \ldots, y_{|D|}$. The coefficients $w_0$ and $w_1$ often provide good approximations to otherwise complicated regression equations.

| $x$ *years experience* | $y$ *salary (in \$1000s)* |
|---|---|
| 3 | 30 |
| 8 | 57 |
| 9 | 64 |
| 13 | 72 |
| 3 | 36 |
| 6 | 43 |
| 11 | 59 |
| 21 | 90 |
| 1 | 20 |
| 16 | 83 |

Table 6.7: Salary data.

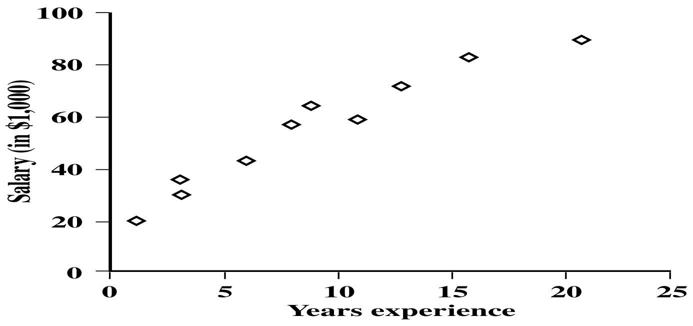


Figure 6.26: Plot of the data in Table 6.7 for Example 6.11. Although the points do not fall on a straight line, the overall pattern suggests a linear relationship between $x$ (*years experience*) and $y$ (*salary*). [TO EDITOR Italicize *years experience* and *salary*. Change vertical axis to read "*salary* (in 1000*s*)", *i.e.*, *add*"*s*"*after*"1000" to match heading in associated table.]

**Example 6.11 Straight-line regression using the method of least squares.** Table 6.7 shows a set of paired data where $x$ is the number of years of work experience of a college graduate and $y$ is the corresponding salary of the graduate. The 2-D data can be graphed on a *scatter plot*, as in Figure 6.26. The plot suggests a

[11]Note that earlier, we had used the notation $(\boldsymbol{X_i},\, y_i)$ to refer to training tuple $i$ having associated class label $y_i$, where $\boldsymbol{X_i}$ was an attribute (or feature) *vector* (that is, $\boldsymbol{X_i}$ was described by more than one attribute). Here, however, we are dealing with just one predictor variable. Since the $\boldsymbol{X_i}$ here are 1-dimensional, we use the notation $x_i$ over $\boldsymbol{X_i}$ in this case.

linear relationship between the two variables, $x$ and $y$. We model the relationship that salary may be related to the number of years of work experience with the equation $y = w_0 + w_1x$.

Given the above data, we compute $\bar{x} = 9.1$ and $\bar{y} = 55.4$. Substituting these values into Equations (6.50) and 6.51), we get

$$w_1 = \frac{(3 - 9.1)(30 - 55.4) + (8 - 9.1)(57 - 55.4) + \cdots + (16 - 9.1)(83 - 55.4)}{(3 - 9.1)^2 + (8 - 9.1)^2 + \cdots + (16 - 9.1)^2} = 3.5$$

$$w_0 = 55.4 - (3.5)(9.1) = 23.6$$

Thus, the equation of the least squares line is estimated by $y = 23.6 + 3.5x$. Using this equation, we can predict that the salary of a college graduate with, say, 10 years of experience is \$58.6K. ■

**Multiple linear regression** is an extension of straight-line regression so as to involve more than one predictor variable. It allows response variable $y$ to be modeled as a linear function of, say, $n$ predictor variables or attributes, $A_1, A_2, \ldots, A_n$, describing a tuple, $\boldsymbol{X}$. (That is, $\boldsymbol{X}=(x_1, x_2, \ldots, x_n)$.) Our training data set, $D$, contains data of the form ($\boldsymbol{X_1}$, $y_1$), ($\boldsymbol{X_2}$, $y_2$), ..., ($\boldsymbol{X}_{|\boldsymbol{D}|}$, $y_{|D|}$), where the $\boldsymbol{X_i}$ are the $n$-dimensional training tuples with associated class labels, $y_i$. An example of a multiple linear regression model based on two predictor attributes or variables, $A_1$ and $A_2$, is

$$y = w_0 + w_1x_1 + w_2x_2 \tag{6.52}$$

where $x_1$ and $x_2$ are the values of attributes $A_1$ and $A_2$, respectively, in $\boldsymbol{X}$. The method of least squares shown above can be extended to solve for $w_0$, $w_1$, and $w_2$. The equations, however, become long and are tedious to solve by hand. Multiple regression problems are instead commonly solved with the use of statistical software packages, such as SAS, SPSS, and S-Plus (see references above.)

### 6.11.2 Nonlinear Regression

*"How can we model data that does not show a linear dependence? For example, what if a given response variable and predictor variable have a relationship that may be modeled by a polynomial function?"* Think back to the straight-line linear regression case above where dependent response variable, $y$, is modeled as a linear function of a single independent predictor variable, $x$. What if we can get a more accurate model using a nonlinear model, such as a parabola or some other higher order polynomial? **Polynomial regression** is often of interest when there is just one predictor variable. It can be modeled by adding polynomial terms to the basic linear model. By applying transformations to the variables, we can convert the nonlinear model into a linear one that can then be solved by the method of least squares.

**Example 6.12 Transformation of a polynomial regression model to a linear regression model.** Consider a cubic polynomial relationship given by

$$y = w_0 + w_1x + w_2x^2 + w_3x^3. \tag{6.53}$$

To convert this equation to linear form, we define new variables:

$$x_1 = x \qquad x_2 = x^2 \qquad x_3 = x^3 \tag{6.54}$$

Equation (6.53) can then be converted to linear form by applying the above assignments, resulting in the equation $y = w_0 + w_1x_1 + w_2x_2 + w_3x_3$, which is easily solved by the method of least squares using software for regression analysis. Note that polynomial regression is a special case of multiple regression. That is, the addition of high-order terms like $x^2$, $x^3$, and so on, which are simple functions of the single variable, $x$, can be considered equivalent to adding new independent variables. ■

In Exercise 15, you are asked to find the transformations required to convert a nonlinear model involving a power function into a linear regression model.

Some models are intractably nonlinear (such as the sum of exponential terms, for example) and cannot be converted to a linear model. For such cases, it may be possible to obtain least square estimates through extensive calculations on more complex formulae.

Various statistical measures exist for determining how well the proposed model can predict $y$. These are described in Section 6.12.2. Obviously, the greater the number of predictor attributes is, the slower the performance is. Prior to applying regression analysis, it is common to perform attribute subset selection (Section 2.5.2) to eliminate attributes that are unlikely to be good predictors for $y$. In general, regression analysis is quite accurate for prediction, except when the data contain outliers. Outliers are data points that are highly inconsistent with the remaining data (e.g., they may be way out of the expected value range). Outlier detection is discussed in Chapter 7. Such techniques must be used with caution, however, so as not to remove data points that are valid although they may vary greatly from the mean.

### 6.11.3 Other Regression-Based Methods

Linear regression is used to model continuous-valued functions. It is widely used, owing largely to its simplicity. *"Can it also be used to predict categorical labels?"* **Generalized linear models** represent the theoretical foundation on which linear regression can be applied to the modeling of categorical response variables. In generalized linear models, the variance of the response variable, $y$, is a function of the mean value of $y$, unlike in linear regression, where the variance of $y$ is constant. Common types of generalized linear models include **logistic regression** and **Poisson regression**. Logistic regression models the probability of some event occurring as a linear function of a set of predictor variables. Count data frequently exhibit a Poisson distribution and are commonly modeled using Poisson regression.

**Log-linear models** approximate *discrete* multidimensional probability distributions. They may be used to estimate the probability value associated with data cube cells. For example, suppose we are given data for the attributes *city, item, year*, and *sales*. In the log-linear method, all attributes must be categorical; hence continuous-valued attributes (like *sales*) must first be discretized. The method can then be used to estimate the probability of each cell in the 4-D base cuboid for the given attributes, based on the 2-D cuboids for *city* and *item*, *city* and *year*, *city* and *sales*, and the 3-D cuboid for *item, year*, and *sales*. In this way, an iterative technique can be used to build higher-order data cubes from lower-order ones. The technique scales up well to allow for many dimensions. Aside from prediction, the log-linear model is useful for data compression (since the smaller-order cuboids together typically occupy less space than the base cuboid) and data smoothing (since cell estimates in the smaller-order cuboids are less subject to sampling variations than cell estimates in the base cuboid).

Decision tree induction can be adapted so as to predict continuous (ordered) values, rather than class labels. There are two main types of trees for prediction—*regression trees* and *model trees*. **Regression trees** were proposed as a component of the CART learning system. (Recall that the acronym CART stands for *Classification and Regression Trees*). Each regression tree leaf stores a continuous-valued prediction, which is actually the average value of the predicted attribute for the training tuples that reach the leaf. Since the terms "regression" and "numeric prediction" are used synonymously in statistics, the resulting trees were called "regression trees", even though they did not use any regression equations. By contrast, in **model trees**, each leaf holds a regression model—a multivariate linear equation for the predicted attribute. Regression and model trees tend to be more accurate than linear regression when the data are not represented well by a simple linear model.

## 6.12 Accuracy and Error Measures

Now that you may have trained a classifier or predictor, there may be many questions going through your mind. For example, suppose you used data from previous sales to train a classifier to predict customer purchasing behavior. You would like an estimate of how accurately the classifier can predict the purchasing behavior of future customers, that is, future customer data on which the classifier has not been trained. You may even have tried different methods to build more than one classifier (or predictor) and now wish to compare their accuracy. But what is accuracy? How can we estimate it? Are their strategies for increasing the accuracy of a learned model? These questions are addressed in the next few sections. Section 6.12.1 describes measures for computing classifier accuracy. Predictor error measures are given in Section 6.12.2. We can use these measures in techniques for accuracy estimation, such as the *holdout, random subsampling*, *k-fold cross-validation*, and *bootstrap* methods (Section 6.13). In Section 6.14, we'll learn some "tricks" for increasing model accuracy, such as *bagging* and *boosting.* Finally, Section 6.15 discusses **model selection** (i.e., choosing one classifier or predictor over another).

### 6.12.1 Classifier Accuracy Measures

Using training data to derive a classifier or predictor and then to estimate the accuracy of the resulting learned model can result in misleading overoptimistic estimates due to overspecialization of the learning algorithm to the data. (We'll say more on this in a moment!) Instead, accuracy is better measured on a test set, consisting of class-labeled tuples that were not used to train the model. The **accuracy** of a classifier on a given test set is the percentage of test set tuples that are correctly classified by the classifier. In the pattern recognition literature, this is also referred to as the overall **recognition rate** of the classifier, that is, it reflects how well the classifier recognizes tuples of the various classes.

We can also speak of the **error rate** or **misclassification rate** of a classifier, $M$, which is simply $1 - Acc(M)$, where $Acc(M)$ is the accuracy of $M$. If we were to use the training set to estimate the error rate of a model, this quantity is known as the **resubstitution error**. This error estimate is optimistic of the true error rate (and similarly, the corresponding accuracy estimate is optimistic) because the model is not tested on any samples that it has not already seen.

| *Classes* | *buys_computer = yes* | *buys_computer = no* | *Total* | *Recognition (%)* |
|---|---|---|---|---|
| *buys_computer = yes* | 6,954 | 46 | 7,000 | 99.34 |
| *buys_computer = no* | 412 | 2,588 | 3,000 | 86.27 |
| *Total* | 7,366 | 2,634 | 10,000 | 95.52 |

Figure 6.27: A confusion matrix for the classes *buys_computer = yes* and *buys_computer = no* where an entry is row $i$ and column $j$ shows the number of tuples of class $i$ that were labeled by the classifier as class $j$. Ideally, the non-diagonal entries should be zero or close to zero.

The *confusion matrix* is a useful tool for analyzing how well your classifier can recognize tuples of different classes. A confusion matrix for two classes is shown in Figure 6.27. Given $m$ classes, a **confusion matrix** is a table of at least size $m$ by $m$. An entry, $CM_{i,j}$ in the first $m$ rows and $m$ columns indicates the number of tuples of class $i$ that were labeled by the classifier as class $j$. For a classifier to have good accuracy, ideally most of the tuples would be represented along the diagonal of the confusion matrix, from entry $CM_{1,1}$ to entry $CM_{m,m}$, with the rest of the entries being close to zero. The table may have additional rows or columns to provide totals or recognition rates per class.

| | $C_1$ | $C_2$ |
|---|---|---|
| $C_1$ | true positives | false negatives |
| $C_2$ | false positives | true negatives |

Figure 6.28: A confusion matrix for positive and negative tuples. [TO EDITOR Please display 'actual class' in bold to left of $C_1$ and $C_2$ of first column (these are the rows). Please display 'predicted class' in bold above the $C_1$ and $C_2$ of columns 2 and 3 (that is, the columns are the predicted class).]

Given two classes, we can talk in terms of **positive tuples** (tuples of the main class of interest, e.g., *buys_computer*

= *yes*) versus **negative tuples** (e.g., *buys_computer* = *no*)[12]. **True positives** refer to the positive tuples that were correctly labeled by the classifier, while **true negatives** are the negative tuples that were correctly labeled by the classifier. **False positives** are the negative tuples that were incorrectly labeled (e.g., tuples of class *buys_computer* = *no* for which the classifier predicted *buys_computer* = *yes*). Similarly, **false negatives** are the positive tuples that were incorrectly labeled (e.g., tuples of class *buys_computer* = *yes* for which the classifier predicted *buys_computer* = *no*). These terms are useful when analyzing a classifier's ability and are summarized in Figure 6.28.

*"Are there alternatives to the accuracy measure?"* Suppose that you have trained a classifier to classify medical data tuples as either *"cancer"* or *"not_cancer"*. An accuracy rate of, say, 90% may make the classifier seem quite accurate, but what if only, say, 3–4% of the training tuples are actually *"cancer"*? Clearly, an accuracy rate of 90% may not be acceptable—the classifier could be correctly labelling only the *"not_cancer"* tuples, for instance. Instead, we would like to be able to access how well the classifier can recognize *"cancer"* tuples (the positive tuples) and how well it can recognize *"not_cancer"* tuples (the negative tuples). The **sensitivity** and **specificity** measures can be used, respectively, for this purpose. Sensitivity is also referred to as the *true positive (recognition) rate* (that is, the proportion of positive tuples that are correctly identified), while specificity is the *true negative rate* (that is, the proportion of negative tuples that are correctly identified). In addition, we may use **precision** to access the percentage of tuples labeled as *"cancer"* that actually are *"cancer"* tuples. These measures are defined as

$$sensitivity = \frac{t\_pos}{pos} \tag{6.55}$$

$$specificity = \frac{t\_neg}{neg} \tag{6.56}$$

$$precision = \frac{t\_pos}{(t\_pos + f\_pos)} \tag{6.57}$$

where *t_pos* is the number of true positives (*"cancer"* tuples that were correctly classified as such), *pos* is the number of positive (*"cancer"*) tuples, *t_neg* is the number of true negatives (*"not_cancer"* tuples that were correctly classified as such), *neg* is the number of negative (*"not_cancer"*) tuples, and *f_pos* is the number of false positives (*"not_cancer"* tuples that were incorrectly labeled as *"cancer"*). It can be shown that accuracy is a function of sensitivity and specificity:

$$accuracy = sensitivity\frac{pos}{(pos + neg)} + specificity\frac{neg}{(pos + neg)}. \tag{6.58}$$

The true positives, true negatives, false positives, and false negatives are also useful in assessing the costs and benefits (or risks and gains) associated with a classification model. The cost associated with a false negative (such as, incorrectly predicting that a cancerous patient is not cancerous) is far greater than that of a false positive (incorrectly yet conservatively labeling a noncancerous patient as cancerous). In such cases, we can outweigh one type of error over another by assigning a different cost to each. These cost may consider the danger to the patient, financial costs of resulting therapies, and other hospital costs. Similarly, the benefits associated with a true positive decision may be different than that of a true negative. Up to now, to compute classifier accuracy, we have assumed equal costs and essentially divide the sum of true positives and true negatives by the total number of test tuples. Alternatively, we can incorporate costs and benefits by instead computing the average cost (or benefit) per decision. Other applications involving cost-benefit analysis include loan application decisions and target marketing mailouts. For example, the cost of loaning to a defaulter greatly exceeds that of the lost business incurred by denying a load to a nondefaulter. Similarly, in an application that tries to identify households that are likely to respond to mailouts of certain promotional material, the cost of mailouts to numerous households that do not respond may outweigh the cost of lost business from not mailing to households that would have responded. Other costs to consider in the overall analysis include the costs to collect the data and to develop the classification tool.

*"Are there other cases where accuracy may not be appropriate?"* In classification problems, it is commonly assumed that all tuples are uniquely classifiable, that is, that each training tuple can belong to only one class.

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

Yet, owing to the wide diversity of data in large databases, it is not always reasonable to assume that all tuples are uniquely classifiable. Rather, it is more probable to assume that each tuple may belong to more than one class. How then can the accuracy of classifiers on large databases be measured? The accuracy measure is not appropriate, since it does not take into account the possibility of tuples belonging to more than one class.

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

### 6.12.2 Predictor Error Measures

*"How can we measure predictor accuracy?"* Let $D^T$ be a test set of the form ($\boldsymbol{X_1}$, $y_1$), ($\boldsymbol{X_2}$, $y_2$), ..., ($\boldsymbol{X_d}$, $y_d$), where the $\boldsymbol{X_i}$ are the $n$-dimensional test tuples with associated known values, $y_i$, for a response variable, $y$, and $d$ is the number of tuples in $D^T$. Since predictors return a continuous value rather than a categorical label, it is difficult to say *exactly* whether the predicted value, $y'_i$, for $\boldsymbol{X_i}$ is correct. Instead of focussing on whether $y'_i$ is an "exact" match with $y_i$, we instead look at how far off the predicted value is from the actual known value. **Loss functions** measure the error between $y_i$ and the predicted value, $y'_i$. The most common loss functions are:

$$\textbf{Absolute error}: \quad |y_i - y'_i| \tag{6.59}$$

$$\textbf{Squared error}: \quad (y_i - y'_i)^2 \tag{6.60}$$

Based on the above, the **test error (rate)**, or **generalization error**, is the average loss over the test set. Thus, we get the following error rates.

$$\textbf{Mean absolute error}: \quad \frac{\sum_{i=1}^{d} |y_i - y'_i|}{d} \tag{6.61}$$

$$\textbf{Mean squared-error}: \quad \frac{\sum_{i=1}^{d} (y_i - y'_i)^2}{d} \tag{6.62}$$

The mean squared-error exaggerates the presence of outliers, while the mean absolute error does not. If we were to take the square root of the mean squared error, the resulting error measure is called the **root mean-squared error**. This is useful in that it allows the error measured to be of the same magnitude as the quantity being predicted.

Sometimes, we may want the error to be relative to what it would have been if we had just predicted $\bar{y}$, the mean value for $y$ from the training data, $D$. That is, we can normalize the total loss by dividing by the total loss incurred from always predicting the mean. Relative measures of error include:

$$\textbf{Relative absolute error} \quad \frac{\sum_{i=1}^{d}|y_i - y_i'|}{\sum_{i=1}^{d}|y_i - \bar{y}|} \tag{6.63}$$

$$\textbf{Relative squared error}: \quad \frac{\sum_{i=1}^{d}(y_i - y_i')^2}{\sum_{i=1}^{d}(y_i - \bar{y})^2} \tag{6.64}$$

where $\bar{y}$ is the mean value of the $y_i$'s of the training data, that is $\bar{y} = \frac{\sum_{i=1}^{t} y_i}{d}$. We can take the root of the relative squared error to obtain the **root relative squared-error** so that the resulting error is of the same magnitude as the quantity predicted.

In practice, the choice of error measure does not greatly affect prediction model selection.

## 6.13 Evaluating the Accuracy of a Classifier or Predictor

How can we use the above measures to obtain a reliable estimate of classifier accuracy (or predictor accuracy in terms of error)? Holdout, random subsampling, cross-validation, and the bootstrap are common techniques for assessing accuracy based on randomly sampled partitions of the given data. The use of such techniques to estimate accuracy increases the overall computation time, yet is useful for model selection.

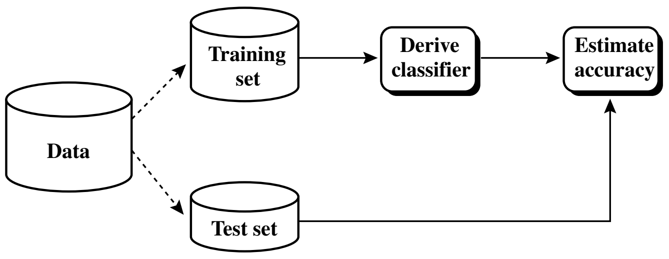


Figure 6.29: Estimating accuracy with the holdout method. [TO EDITOR PLEASE CHANGE "Derive classifier" TO "Derive model".]

### 6.13.1 Holdout Method and Random Subsampling

The **holdout** method is what we have to alluded to so far in our discussions about accuracy. In this method, the given data are randomly partitioned into two independent sets, a *training set* and a *test set.* Typically, two thirds of the data are allocated to the training set, and the remaining one third is allocated to the test set. The training set is used to derive the model, whose accuracy is estimated with the test set (Figure 6.29). The estimate is pessimistic since only a portion of the initial data is used to derive the model.

**Random subsampling** is a variation of the holdout method in which the holdout method is repeated $k$ times. The overall accuracy estimate is taken as the average of the accuracies obtained from each iteration. (For prediction, we can take the average of the predictor error rates.)

### 6.13.2 Cross-validation

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

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

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

### 6.13.3 Bootstrap

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

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

*"Where does the figure, 63.2%, come from?"* Each tuple has a probability of $1/d$ of being selected, so the probability of not being chosen is $(1 - 1/d)$. We have to select $d$ times, so the probability that a tuple will not be chosen during this whole time is $(1 - 1/d)^d$. If $d$ is large, the probability approaches $e^{-1} = 0.368$.[13] Thus, 36.8% of tuples will not be selected for training and thereby end up in the test set, and the remaining 63.2% will form the training set.

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

$$Acc(M) = \sum_{i=1}^{k}(0.632 \times Acc(M_i)_{test\_set} + 0.368 \times Acc(M_i)_{train\_set}), \tag{6.65}$$

where $Acc(M_i)_{test\_set}$ is the accuracy of the model obtained with bootstrap sample $i$ when it is applied to test set $i$. $Acc(M_i)_{train\_set}$ is the accuracy of the model obtained with bootstrap sample $i$ when it is applied to the original set of data tuples. The bootstrap method works well with small data sets.

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

## 6.14 Ensemble Methods—Increasing the Accuracy

In Section 6.3.3, we saw how pruning can be applied to decision tree induction to help improve the accuracy of the resulting decision trees. Are there *general* strategies for improving classifier and predictor accuracy?

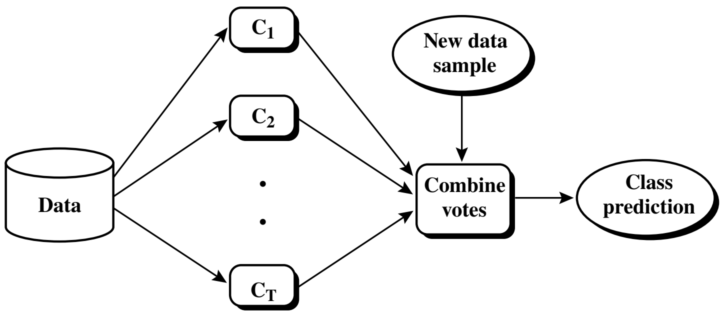


Figure 6.30: Increasing model accuracy: Bagging and boosting each generate a set of classification or prediction models, $M_1, M_2, \ldots, M_k$. Voting strategies are used to combine the predictions for a given unknown tuple. [TO EDITOR PLEASE CHANGE $C_1, C_2, \ldots, C_T$ to $M_1, M_2, \ldots, M_k$. ALSO CHANGE "Class prediction" TO "Prediction".]

The answer is yes. *Bagging* and *boosting* are two such techniques (Figure 6.30). They are examples of **ensemble methods**, or methods that use a *combination* of models. Each combines a series of $k$ learned models (classifiers or predictors), $M_1, M_2, \ldots, M_k$, with the aim of creating an improved composite model, $M*$. Both bagging and boosting can be used for classification as well as prediction.

### 6.14.1 Bagging

We first take an intuitive look at how bagging works as a method of increasing accuracy. For ease of explanation, we will assume at first that our model is a classifier. Suppose that you are a patient and would like to have a diagnosis made based on your symptoms. Instead of asking one doctor, you may choose to ask several. If a certain diagnosis occurs more than any of the others, you may choose this as the final or best diagnosis. That is, the final diagnosis is made based on a majority vote, where each doctor gets an equal vote. Now replace each doctor by a classifier, and you have the basic idea behind bagging. Intuitively, a majority vote made by a large group of doctors may be more reliable than a majority vote made by a small group.

Given a set, $D$, of $d$ tuples, bagging works as follows. For iteration $i$ ($i = 1, 2, \ldots, k$), a training set, $D_i$, of $d$ tuples is sampled with replacement from the original set of tuples, $D$. Note that the term bagging stands for *bootstrap aggregation.* Each training set is a bootstrap sample, as described in Section 6.13.3. Since sampling with replacement is used, some of the original tuples of $D$ may not be included in $D_i$, while others may occur more than once. A classifier model, $M_i$, is learned for each training set, $D_i$. To classify an unknown tuple, $\boldsymbol{X}$, each classifier, $M_i$, returns its class prediction, which counts as one vote. The bagged classifier, $M*$, counts the votes and assigns the class with the most votes to $\boldsymbol{X}$. Bagging can be applied to the prediction of continuous values by taking the average value of each prediction for a given test tuple. The algorithm is summarized in Figure 6.31.

The bagged classifier often has significantly greater accuracy that a single classifier derived from $D$, the original training data. It will not be considerably worse and is more robust to the effects of noisy data. The increased accuracy occurs because the composite model reduces the variance of the individual classifiers. For prediction, it was theoretically proven that a bagged predictor will *always* have improved accuracy over a single predictor derived from $D$.

**Algorithm: Bagging.** The bagging algorithm—create an ensemble of models (classifiers or predictors) for a learning scheme where each model gives an equally-weighted prediction.

**Input:**

- $D$, a set of $d$ training tuples;
- $k$, the number of models in the ensemble;
- a learning scheme (e.g., decision tree algorithm, backpropagation, etc.)

**Output:** A composite model, $M*$.

**Method:**

```
(1)  for $i = 1$ to $k$ do // create $k$ models:
(2)      create bootstrap sample, $D_i$, by sampling $D$ with replacement;
(3)      use $D_i$ to derive a model, $M_i$;
(4)  endfor
```

**To use the composite model on a tuple, $X$:**

```
(1)  if classification then
(2)      let each of the $k$ models classify $X$ and return the majority vote;
(3)  if prediction then
(4)      let each of the $k$ models predict a value for $X$ and return the average predicted value;
```

Figure 6.31: Bagging.

### 6.14.2 Boosting

We now look at the ensemble method of boosting. As in the previous section, suppose that as a patient, you have certain symptoms. Instead of consulting one doctor, you choose to consult several. Suppose you assign weights to the "value" or worth of each doctor's diagnosis, based on the accuracies of previous diagnoses they have made. The final diagnosis is then a combination of the weighted diagnoses. This is the essence behind boosting.

In **boosting**, weights are assigned to each training tuple. A series of $k$ classifiers is iteratively learned. After a classifier $M_i$ is learned, the weights are updated to allow the subsequent classifier, $M_{i+1}$, to "pay more attention" to the training tuples that were misclassified by $M_i$. The final boosted classifier, $M*$, combines the votes of each individual classifier, where the weight of each classifier's vote is a function of its accuracy. The boosting algorithm can be extended for the prediction of continuous values.

Adaboost is a popular boosting algorithm. Suppose we would like to boost the accuracy of some learning method. We are given $D$, a data set of $d$ class-labeled tuples, $(\boldsymbol{X_1}, y_1)$, $(\boldsymbol{X_2}, y_2)$, ..., $(\boldsymbol{X_d}, y_d)$, where $y_i$ is the class label of tuple $\boldsymbol{X_i}$. Initially, Adaboost assigns each training tuple an equal weight of $1/d$. Generating $k$ classifiers for the ensemble requires $k$ rounds through the rest of the algorithm. In round $i$, the tuples from $D$ are sampled to form a training set, $D_i$, of the same size. Each tuple's chance of being selected is based on its weight. Sampling with replacement is used—the same tuple may be selected more than once. A classifier model, $M_i$, is derived from the training tuples of $D_i$. Its error is then calculated using $D_i$ as a test set. The weights of the training tuples are then adjusted according to how they were classified. If a tuple was incorrectly classified, its weight is increased. If a tuple was correctly classified, its weight is decreased. A tuple's weight reflects how hard it is to classify—the higher the weight, the more often it has been misclassified. These weights will be used to generate the training samples for the classifier of the next round. The basic idea is that when we build a classifier, we want it to focus more on the misclassified tuples of the previous round. Some classifiers may be better at classifying some "hard" tuples than others. In this way, we build a series of classifiers that complement each other. The algorithm is summarized in Figure 6.32.

**Algorithm: Adaboost.** A boosting algorithm—create an ensemble of classifiers. Each one gives a weighted vote.

**Input:**

- $D$, a set of $d$ class-labeled training tuples;
- $k$, the number of rounds (one classifier is generated per round);
- a classification learning scheme.

**Output:** A composite model.

**Method:**

```
(1)  initialize the weight of each tuple in $D$ to $1/d$;
(2)  for $i = 1$ to $k$ do // for each round:
(3)      sample $D$ with replacement according to the tuple weights to obtain $D_i$;
(4)      use training set $D_i$ to derive a model, $M_i$;
(5)      compute $error(M_i)$, the error rate of $M_i$ (Equation 6.66)
(6)      if $error(M_i) > 0.5$ then
(7)          reinitialize the weights to $1/d$
(8)          go back to step 3 and try again;
(9)      endif
(10)     for each tuple in $D_i$ that was correctly classified do
(11)         multiply the weight of the tuple by $error(M_i)/(1 - error(M_i))$; // update weights
(12)     normalize the weight of each tuple;
(13) endfor
```

**To use the composite model to classify tuple, $X$:**

```
(1)  initialize weight of each class to 0;
(2)  for $i = 1$ to $k$ do // for each classifier:
(3)      $w_i = log\frac{1-error(M_i)}{error(M_i)}$; // weight of the classifier's vote
(4)      $c = M_i(X)$; // get class prediction for $X$ from $M_i$
(5)      add $w_i$ to weight for class $c$
(6)  endfor
(7)  return the class with the largest weight;
```

Figure 6.32: Adaboost, a boosting algorithm.

Now, let's look at some of the math that's involved in the algorithm. To compute the error rate of model $M_i$, we sum the weights of each of the tuples in $D_i$ that $M_i$ misclassified. That is,

$$error(M_i) = \sum_{j}^{d} w_j \times err(\boldsymbol{X_j}), \tag{6.66}$$

where $err(\boldsymbol{X_j})$ is the misclassification error of tuple $\boldsymbol{X_j}$: If the tuple was misclassified, then $err(\boldsymbol{X_j})$ is 1. Otherwise, it is 0. If the performance of classifier $M_i$ is so poor that its error exceeds 0.5, then we abandon it. Instead, we try again by generating a new $D_i$ training set, from which we derive a new $M_i$.

The error rate of $M_i$ affects how the weights of the training tuples are updated. If a tuple in round $i$ was correctly classified, its weight is multiplied by $error(M_i)/(1 - error(M_i))$. Once the weights of all of the correctly classified tuples are updated, the weights for all tuples (including the misclassified ones) are normalized so that their sum remains the same as it was before. To normalize a weight, we multiply it by the sum of the old weights, divided by the sum of the new weights. As a result, the weights of misclassified tuples are increased and the weights of correctly classified tuples are decreased, as described above.

*"Once boosting is complete, how is the ensemble of classifiers used to predict the class label of a tuple, $\boldsymbol{X}$?"*

Unlike bagging, where each classifier was assigned an equal vote, boosting assigns a weight to each classifier's vote, based on how well the classifier performed. The lower a classifier's error rate, the more accurate it is, and therefore, the higher its weight for voting should be. The weight of classifier $M_i$'s vote is

$$log \frac{1 - error(M_i)}{error(M_i)} \tag{6.67}$$

For each class, $c$, we sum the weights of each classifier that assigned class $c$ to $\boldsymbol{X}$. The class with the highest sum is the "winner" and is returned as the class prediction for tuple $\boldsymbol{X}$.

*"How does boosting compare with bagging?"* Because of the way boosting focusses on the misclassified tuples, it risks overfitting the resulting composite model to such data. Therefore, sometimes the resulting "boosted" model may be less accurate than a single model derived from the same data. Bagging is less susceptible to model overfitting. While both can significantly improve accuracy in comparison to a single model, boosting tends to to achieve greater accuracy.

## 6.15 Model Selection

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

### 6.15.1 Estimating Confidence Intervals

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

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

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

$$t = \frac{\overline{err}(M_1) - \overline{err}(M_2)}{\sqrt{var(M_1 - M_2)/k}}, \tag{6.68}$$

where

$$var(M_1 - M_2) = \frac{1}{k}\sum_{i=1}^{k}[err(M_1)_i - err(M_2)_i - (\overline{err}(M_1) - \overline{err}(M_2))]^2. \tag{6.69}$$

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

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

$$var(M_1 - M_2) = \sqrt{\frac{var(M_1)}{k_1} + \frac{var(M_2)}{k_2}}, \tag{6.70}$$

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

### 6.15.2 ROC Curves

**ROC curves** are a useful visual tool for comparing two classification models. The name ROC stands for *Receiver Operating Characteristic.* ROC curves come from signal detection theory that was developed during WWII for the analysis of radar images. An ROC curve shows the trade-off between the true positive rate or sensitivity (proportion of positive tuples that are correctly identified) and the false positive rate (proportion of negative tuples that are incorrectly identified as positive) for a given model. That is, given a two class problem, it allows us to visualize the trade-off between the rate at which the model can accurately recognize 'yes' cases versus the rate at which it mistakenly identifies 'no' cases as 'yes' for different "portions" of the test set. Any increase in the true positive rate occurs at the cost of an increase in the false positive rate. The area under the ROC curve is a measure of the accuracy of the model.

In order to plot an ROC curve for a given classification model, $M$, the model must be able to return a probability or ranking for the predicted class of each test tuple. That is, we need to rank the test tuples in decreasing order, where the one the classifier thinks is most likely to belong to the positive or 'yes' class appears at the top of the list. Naive Bayesian and backpropagation classifiers are appropriate, while others, such as decision tree classifiers, can easily be modified so as to return a class probability distribution for each prediction. The vertical axis of an ROC curve represents the true positive rate. The horizontal axis represents to the false positive rate. An ROC curve for $M$ is plotted as follows. Starting at the bottom left hand corner (where the true positive rate and false positive rate are both 0), we check the actual class label of the tuple at the top of the list. If we have a true positive (that is, a positive tuple that was correctly classified), then on the ROC curve, we move up and plot a point. If, instead, the tuple really belongs to the 'no' class, we have a false positive. On the ROC curve, we move right and plot a point. This process is repeated for each of the test tuples, each time moving up on the curve for a true positive or towards the right for a false positive.

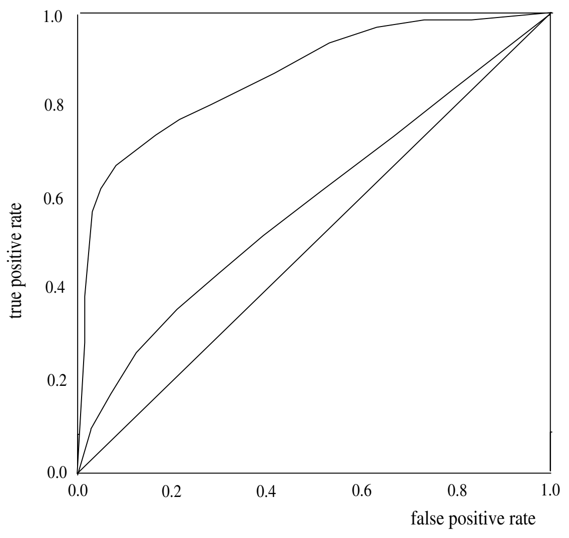


Figure 6.33: The ROC curves of two classification models.

Figure 6.33 shows the ROC curves of two classification models. The plot also shows a diagonal line. The closer the ROC curve of a model is to the diagonal line, the less accurate is the model. For every true positive of such a model, we are just as likely to encounter a false positive. If the model is really good, initially we are more likely to encounter true positives as we move down the ranked list. Thus, the curve would move steeply up from zero. Later, as we start to encounter fewer and fewer true positives, and more and more false positives, the curve eases off and becomes more horizontal.

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

## 6.16 Summary

- 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. While **classification** predicts categorical labels (classes), **prediction** models continuous-valued functions.

- Preprocessing of the data in preparation for classification and prediction can involve **data cleaning** to reduce noise or handle missing values, **relevance analysis** to remove irrelevant or redundant attributes, and **data transformation**, such as generalizing the data to higher-level concepts or normalizing the data.

- Predictive accuracy, computational speed, robustness, scalability, and interpretability are five **criteria** for the evaluation of classification and prediction methods.

- **ID3**, **C4.5**, and **CART** are greedy algorithms for the induction of **decision trees**. Each algorithm uses an attribute selection measure to select the attribute tested for each nonleaf node in the tree. **Pruning** algorithms attempt to improve accuracy by removing tree branches reflecting noise in the data. Early decision tree algorithms typically assume that the data are memory resident—a limitation to data mining on large databases. Several scalable algorithms, such as **SLIQ**, **SPRINT**, and **RainForest**, have been proposed to address this issue.

- **Naive Bayesian classification** and **Bayesian belief networks** are based on Bayes theorem of posterior probability. Unlike naive Bayesian classification (which assumes class conditional independence), Bayesian belief networks allow class conditional independencies to be defined between subsets of variables.

- A **rule-based classifier** uses a set of IF-THEN rules for classification. Rules can be extracted from a decision tree or directly from the training data.

- **Backpropagation** is a neural network algorithm for classification that employs a method of gradient descent. It searches for a set of weights that can model the data so as to minimize the mean squared distance between the network's class prediction and the actual class label of data tuples. Rules may be extracted from trained neural networks in order to help improve the interpretability of the learned network.

- A **Support Vector Machine (SVM)** is an algorithm for the classification of both linear and nonlinear data. It transforms the original data in a higher dimension, from where it can find a hyperplane for separation of the data using essential training tuples called **support vectors**.

- **Association mining** techniques, which search for frequently occurring patterns in large databases, can be adapted for classification.

- Decision tree classifiers, Bayesian classifiers, classification by backpropagation, support vector machines, and classification based on association are all examples of **eager learners** in they use training tuples to construct a generalization model and in this way are ready for classifying new tuples. This contrasts with **lazy learners** or **instance-based** methods of classification, such as nearest neighbor classifiers and case-based reasoning classifiers, which store all of the training tuples in pattern space and wait until presented with a test tuple before performing generalization. Hence, lazy learners require efficient indexing techniques.

- In **genetic algorithms**, populations of rules "evolve" via operations of crossover and mutation until all rules within a population satisfy a specified threshold. **Rough set theory** can be used to approximately define classes that are not distinguishable based on the available attributes. **Fuzzy set** approaches replace "brittle" threshold cutoffs for continuous-valued attributes with degree of membership functions.

- Linear, nonlinear, and generalized linear models of **regression** can be used for prediction. Many nonlinear problems can be converted to linear problems by performing transformations on the predictor variables. Unlike decision trees, regression trees and model trees are used for prediction. In regression trees, each leaf stores a continuous-valued prediction. In model trees, each leaf holds a regression model.

- **Stratified *k*-fold cross-validation** is a recommended method for accuracy estimation. **Bagging** and **boosting** methods can be used to increase overall accuracy by learning and combining a series of individual models. For classifiers, **sensitivity**, **specificity**, and **precision** are useful alternatives to the accuracy measure, particularly when the main class of interest is in the minority. There are many measures of predictor error, such as the **mean squared-error**, the **mean absolute error**, the **relative squared-error**, and the **relative absolute error**. **Significance tests** and **ROC curves** are useful for model selection.

- There have been numerous comparisons of the different classification and prediction methods, and the matter remains a research topic. No single method has been found to be superior over all others for all data sets. Issues such as accuracy, training time, robustness, interpretability, and scalability must be considered and can involve trade-offs, further complicating the quest for an overall superior method. Empirical studies show that the accuracies of many algorithms are sufficiently similar that their differences are statistically insignificant, while training times may differ substantially. For classification, most neural network and statistical methods involving splines tend to be more computationally intensive than most decision tree methods.

## 6.17 Exercises

1. Briefly outline the major steps of *decision tree classification.*
2. Why is *tree pruning* useful in decision tree induction? What is a drawback of using a separate set of tuples to evaluate pruning?
3. Given a decision tree, you have the option of (a) converting the decision tree to rules and then pruning the resulting rules, or (b) pruning the decision tree and then converting the pruned tree to rules. What advantage does (a) have over (b)?
4. It is important to calculate the worst-case computational complexity of the decision tree algorithm. Given data set $D$, the number of attributes $n$, and the number of training tuples $|D|$, show that the computational cost of growing a tree is at most $n \times |D| \times log(|D|)$.
5. Why is *naive Bayesian classification* called "naive"? Briefly outline the major ideas of naive Bayesian classification.
6. Given a 5 GB data set with 50 attributes (each containing 100 distinct values) and 512 MB of main memory in your laptop, outline an efficient method that constructs decision trees in such large data sets. Justify your answer by rough calculation of your main memory usage.
7. Rainforest is an interesting scalable algorithm for decision-tree induction. Develop a scalable naive Bayesian classification algorithm that requires just a single scan of the entire data set for most databases. Discuss whether such an algorithm can be refined to incorporate *boosting* to further enhance its classification accuracy.
8. Compare the advantages and disadvantages of *eager* classification (e.g., decision tree, Bayesian, neural network) versus *lazy* classification (e.g., $k$-nearest neighbor, case-based reasoning).
9. Design an efficient method that performs effective naive Bayesian classification over an *infinite* data stream (i.e., you can scan the data stream only once). If we wanted to discover the *evolution* of such classification schemes (e.g., comparing the classification scheme at this moment with earlier schemes, such as one from a week ago), what modified design would you suggest?
10. What is *association-based classification*? Why is association-based classification able to achieve higher classification accuracy than a classical decision-tree method? Explain how association-based classification can be used for text document classification.
11. The following table consists of training data from an employee database. The data have been generalized. For example, "31 . . . 35" for *age* represents the age range of 31 to 35. For a given row entry, *count* represents the number of data tuples having the values for *department, status, age*, and *salary* given in that row.

| *department* | *status* | *age* | *salary* | *count* |
|---|---|---|---|---|
| sales | senior | 31. . . 35 | 46K. . . 50K | 30 |
| sales | junior | 26. . . 30 | 26K. . . 30K | 40 |
| sales | junior | 31. . . 35 | 31K. . . 35K | 40 |
| systems | junior | 21. . . 25 | 46K. . . 50K | 20 |
| systems | senior | 31. . . 35 | 66K. . . 70K | 5 |
| systems | junior | 26. . . 30 | 46K. . . 50K | 3 |
| systems | senior | 41. . . 45 | 66K. . . 70K | 3 |
| marketing | senior | 36. . . 40 | 46K. . . 50K | 10 |
| marketing | junior | 31. . . 35 | 41K. . . 45K | 4 |
| secretary | senior | 46. . . 50 | 36K. . . 40K | 4 |
| secretary | junior | 26. . . 30 | 26K. . . 30K | 6 |

Let *status* be the class label attribute.

(a) How would you modify the ID3 algorithm to take into consideration the *count* of each generalized data tuple (i.e., of each row entry)?

(b) Use your modified version of ID3 to construct a decision tree from the given data.

(c) Given a data tuple having the values *"systems"*, *"26. . . 30"*, and *"46–50K"* for the attributes *department, age*, and *salary*, respectively, what would a naive Bayesian classification of the *status* for the tuple be?

(d) Design a multilayer feed-forward neural network for the given data. Label the nodes in the input and output layers.

(e) Using the multilayer feed-forward neural network obtained above, show the weight values after one iteration of the backpropagation algorithm given the training instance *"(sales, senior, 31. . . 35, 46K. . . 50K)"*. Indicate your initial weight values and biases, and the learning rate used.

12. The *support vector machine (SVM)* is a highly accurate classification method. However, SVM classifiers suffer from slow processing when training with a large set of data tuples. Discuss how to overcome this difficulty and develop a scalable SVM algorithm for efficient SVM classification in large datasets.

13. Write an algorithm for *k-nearest neighbor classification* given $k$ and $n$, the number of attributes describing each tuple.

14. The following table shows the midterm and final exam grades obtained for students in a database course.

| $x$ *Midterm exam* | $y$ *Final exam* |
|---|---|
| 72 | 84 |
| 50 | 63 |
| 81 | 77 |
| 74 | 78 |
| 94 | 90 |
| 86 | 75 |
| 59 | 49 |
| 83 | 79 |
| 65 | 77 |
| 33 | 52 |
| 88 | 74 |
| 81 | 90 |

(a) Plot the data. Do $x$ and $y$ seem to have a linear relationship?

(b) Use the *method of least squares* to find an equation for the prediction of a student's final exam grade based on the student's midterm grade in the course.

(c) Predict the final exam grade of a student who received an 86 on the midterm exam.

15. Some *nonlinear regression* models can be converted to linear models by applying transformations to the predictor variables. Show how the nonlinear regression equation $y = \alpha X^{\beta}$ can be converted to a linear regression equation solvable by the method of least squares.

16. What is *boosting*? State why it may improve the accuracy of decision tree induction.

17. Show that accuracy is a function of *sensitivity* and *specificity*, that is, prove Equation (6.58).

18. Suppose that we would like to select between two prediction models, $M_1$ and $M_2$. We have performed 10 rounds of 10-fold cross validation on each model, where the same data partitioning in round $i$ is used for both $M_1$ and $M_2$. The error rates obtained for $M_1$ are 30.5, 32.2, 20.7, 20.6, 31.0, 41.0, 27.7, 26.0, 21.5, 26.0. The error rates for $M_2$ are 22.4, 14.5, 22.4, 19.6, 20.7, 20.4, 22.1, 19.4, 16.2, 35.0. Comment on whether one model is significantly better than the other considering a significance level of 1%.

19. It is difficult to assess classification *accuracy* when individual data objects may belong to more than one class at a time. In such cases, comment on what criteria you would use to compare different classifiers modeled after the same data.

## 6.18 Bibliographic Notes

Classification from machine learning, statistics, and pattern recognition perspectives has been described in many books, such as Weiss and Kulikowski [WK91], Michie, Spiegelhalter, and Taylor [MST94], Russel and Norvig [RN95], Langley [Lan96], Mitchell [Mit97], Hastie, Tibshirani, and Friedman [HTF01], Duda, Hart, and Stork [DHS01], Alpaydin [Alp04], Tan, Steinbach, and Kumar [TSK05], and Witten and Frank [WF05]. Many of these books describe each of the basic methods of classification discussed in this chapter, as well as practical techniques for the evaluation of classifier performance. Edited collections containing seminal articles on machine learning can be found in Michalski, Carbonell, and Mitchell [MCM83, MCM86], Kodratoff and Michalski [KM90], Shavlik and Dietterich [SD90], and Michalski and Tecuci [MT94]. For a presentation of machine learning with respect to data mining applications, see Michalski, Bratko, and Kubat [MBK98].

The C4.5 algorithm is described in a book by Quinlan [Qui93]. The CART system is detailed in *Classification and Regression Trees* by Breiman, Friedman, Olshen, and Stone [BFOS84]. Both books give an excellent presentation of many of the issues regarding decision tree induction. C4.5 has a commercial successor, known as C5.0, which can be found at *www.rulequest.com*. ID3, a predecessor of C4.5, is detailed in [Qui86]. It expands on pioneering work on concept learning systems, described by Hunt, Marin, and Stone [HMS66]. Other algorithms for decision tree induction include FACT (Loh and Vanichsetakul [LV88]), QUEST (Loh and Shih [LS97]), PUBLIC (Rastogi and Shim [RS98]), and CHAID (Kass [Kas80] and Magidson [Mag94]). INFERULE (Uthurusamy, Fayyad, and Spangler [UFS91]) learns decision trees from inconclusive data, where probabilistic rather than categorical classification rules are obtained. KATE (Manago and Kodratoff [MK91]) learns decision trees from complex structured data. Incremental versions of ID3 include ID4 (Schlimmer and Fisher [SF86]) and ID5 (Utgoff [Utg88]), the latter of which is extended in Utgoff, Berkman, and Clouse [UBC97]. An incremental version of CART is described in Crawford [Cra89]. BOAT (Gehrke, Ganti, Ramakrishnan, and Loh [GGRL99]), a decision tree algorithm that addresses the scalabilty issue in data mining, is also incremental. Other decision tree algorithms that address scalability include SLIQ (Mehta, Agrawal, and Rissanen [MAR96]), SPRINT (Shafer, Agrawal, and Mehta [SAM96]), RainForest (Gehrke, Ramakrishnan, and Ganti [GRG98]), and earlier approaches, such as [Cat91, CS93a, CS93b]. The integration of attribution-oriented induction with decision tree induction is proposed in Kamber, Winstone, Gong, et al. [KWG$^+$97]. For a comprehensive survey of many salient issues relating to decision tree induction, such as attribute selection and pruning, see Murthy [Mur98].

For a detailed discussion on attribute selection measures, see Kononenko and Hong [KH97]. Information gain was proposed by Quinlan [Qui86] and is based on pioneering work on information theory by Shannon and Weaver [SW49]. The gain ratio, proposed as an extension to information gain, is described as part of C4.5 [Qui93]. The gini index was proposed for CART [BFOS84]. The G-statistic, based on information theory, is given in Sokal and Rohlf [SR81]. Comparisons of attribute selection measures include Buntine and Niblett [BN92], Fayyad and Irani [FI92], Kononenko [Kon95], Loh and Shih [LS97], and Shih [Shi00]. Fayyad and Irani [FI92] show limitations of impurity-based measures such as information gain and gini index. They propose a class of attribute selection measures called C-SEP (Class SEParation), which outperform impurity-based measures in certain cases. Kononenko [Kon95] notes that attribute selection measures based on the minimum description length principle have the least bias towards multivalued attributes. Martin and Hirschberg [MH95] proved that the time complexity of decision tree induction increases exponentially with respect to tree height in the worst case, and under fairly general conditions, in the average case. Fayad and Irani [FI90] found that shallow decision trees tend to have many leaves and higher error rates for a large variety of domains. Attribute (or feature) construction is described in Liu and Motoda [LM98, Le98]. Examples of systems with attribute construction include BACON by Langley, Simon, Bradshaw, Zytkow [LSBZ87], Stagger by Schlimmer [Sch86], FRINGE by Pagallo [Pag89], and AQ17-DCI by Bloedorn and Michalski [BM98].

There are numerous algorithms for decision tree pruning, including cost complexity pruning (Breiman, Friedman, Olshen, and Stone [BFOS84]), reduced error pruning (Quinlan [Qui87]), and pessimistic pruning (Quinlan [Qui86]). PUBLIC (Rastogi and Shim [RS98]) integrates decision tree construction with tree pruning. MDL-based pruning methods can be found in Quinlan and Rivest [QR89], Mehta, Agrawal, and Rissanen [MRA95], and Rastogi and Shim [RS98]. Other methods include Niblett and Bratko [NB86], and Hosking, Pednault, and Sudan [HPS97]. For an empirical comparison of pruning methods, see Mingers [Min89] and Malerba, Floriana, and Semeraro [MFS95]. For a survey on simplifying decision trees, see Breslow and Aha [BA97].

There are several examples of rule-based classifiers. These include AQ15 (Hong, Mozetic, and Michalski [HMM86]), CN2 (Clark and Niblett [CN89]), ITRULE (Smyth and Goodman [SG92]), [NEW RISE (Domingos [Dom94]), IREP (Furnkranz and Widmer [FW94]), RIPPER (Cohen [Coh95]),] FOIL (Quinlan [NEW and Cameron-Jones] [Qui90, QCJ93]), and Swap-1 (Weiss and Indurkhya [WI98]). For the extraction of rules from decision trees, see Quinlan [Qui87, Qui93]. Rule refinement strategies that identify the most interesting rules among a given rule set can be found in Major and Mangano [MM95].

Thorough presentations of Bayesian classification can be found in Duda, Hart, and Stork [DHS01], Weiss and Kulikowski [WK91], and Mitchell [Mit97]. For an analysis of the predictive power of naive Bayesian classifiers when the class conditional independence assumption is violated, see Domingos and Pazzani [DP96]. Experiments with kernel density estimation for continuous-valued attributes, rather than Gaussian estimation, have been reported for naive Bayesian classifiers in John [Joh97]. For an introduction to Bayesian belief networks, see Heckerman [Hec96]. For a thorough presentation of probabilistic networks, see Pearl [Pea88]. Solutions for learning the belief network structure from training data given observable variables are proposed in [CH92, Bun94, HGC95]. Algorithms for inference on belief networks can be found in Russell and Norvig [RN95] and Jensen [Jen96]. The method of gradient descent, described in Section 6.4.4 for training Bayesian belief networks, is given in Russell, Binder, Koller, and Kanazawa [RBKK95]. The example given in Figure 6.11 is adapted from Russell et al. [RBKK95]. Alternative strategies for learning belief networks with hidden variables include application of Dempster, Laird, and Rubin's [DLR77] EM (Expectation Maximization) algorithm (Lauritzen [Lau95]) and methods based on the minimum description length principle (Lam [Lam98]). Cooper [Coo90] showed that the general problem of inference in unconstrained belief networks is NP-hard. Limitations of belief networks, such as their large computational complexity (Laskey and Mahoney [LM97]), have prompted the exploration of hierarchical and composable Bayesian models (Pfeffer, Koller, Milch, and Takusagawa [PKMT99] and Xiang, Olesen, and Jensen [XOJ00]). These follow an object-oriented approach to knowledge representation.

The perceptron is a simple neural network, proposed in 1958 by Rosenblatt [Ros58], which became a landmark in early machine learning history. Its input units are randomly connected to a single layer of output linear threshold units. In 1969, Minsky and Papert [MP69] showed that perceptrons are incapable of learning concepts that are linearly inseparable. This limitation, as well as limitations on hardware at the time, dampened enthusiasm for research in computational neuronal modeling for nearly twenty years. Renewed interest was sparked following presentation of the backpropagation algorithm in 1986 by Rumelhart, Hinton, and Williams [RHW86], as this algorithm can learn concepts that are linearly inseparable. Since then, many variations for backpropagation have been proposed, involving, for example, alternative error functions (Hanson and Burr [HB88]), dynamic adjustment of the network topology (Mézard and Nadal [MN89], Fahlman and Lebiere [FL90], Le Cun, Denker, and Solla [LDS90], and Harp, Samad, and Guha [HSG90]), and dynamic adjustment of the learning rate and momentum parameters (Jacobs [Jac88]). Other variations are discussed in Chauvin and Rumelhart [CR95]. Books on neural networks include [RM86, HN90, HKP91, Fu94, CR95, Bis95, Rip96, Hay99]. Many books on machine learning, such as [Mit97, RN95], also contain good explanations of the backpropagation algorithm. There are several techniques for extracting rules from neural networks, such as [SN88, Gal93, TS93, Fu94, Avn95, LSL95, CS96, LGT97]. The method of rule extraction described in Section 6.6.4 is based on Lu, Setiono, and Liu [LSL95]. Critiques of techniques for rule extraction from neural networks can be found in Craven and Shavlik [CS97]. Roy [Roy00] proposes that the theoretical foundations of neural networks are flawed with respect to assumptions made regarding how connectionist learning models the brain. An extensive survey of applications of neural networks in industry, business, and science is provided in Widrow, Rumelhart, and Lehr [WRL94].

Support Vector Machines (SVMs) grew out of early work by Vapnik and Chervonenkis on statistical learning theory [VC71]. The first paper on SVMs was presented by Boser, Guyon and Vapnik [BGV92]. More detailed accounts can be found in books by Vapnik [Vap95, Vap98]. Good starting points include the tutorial on SVMs by Burges [Bur98] and textbook coverage by Kecman [Kec01]. For methods for solving optimization problems, see Fletcher [Fle87] and Nocedal and Wright [NW99]. These references give additional details alluded to as "fancy math tricks" in our text, such as transformation of the problem to a Lagrangian formulation and subsequent solving using Karush-Kuhn-Tucker (KKT) conditions. For the application of SVMs to regression, see Schlkopf, Bartlett, Smola, and Williamson [SBSW99], and Drucker, Burges, Kaufman, Smola, and Vapnik [DBK$^+$97]. Approaches to SVM for large data include the sequential minimal optimization algorithm by Platt [Pla98], decomposition approaches such as in Osuna, Freund, and Girosi [OFG97], and CB-SVM, a microclustering based SVM algorithm for large data sets, by Yu, Yang and Han [YYH03].

Many algorithms have been proposed that adapt association rule mining to the task of classification. The CBA algorithm for associative classification was proposed by Liu, Hsu, and Ma [LHM98]. A classifier, using emerging patterns, was proposed by Dong and Li [DL99] and Li, Dong, and Ramamohanarao [LDR00]. CMAR (Classification based on Multiple Association Rules) was presented in Li, Han, and Pei [LHP01]. CPAR (Classification based on Predictive Association Rules) was proposed in Yin and Han [YH03]. Cong, Tan, Tung, and Xu proposed a method for mining top-$k$ covering rule groups for classifying gene expression data with high accuracy [CTTX05]. Lent, Swami, and Widom [LSW97] proposed the ARCS system, which was described in Section 5.3 on mining multidimensional association rules. It combines ideas from association rule mining, clustering, and image processing, and applies them to classification. Meretakis and Wüthrich [MW99] proposed to construct a naive Bayesian classifier by mining long itemsets.

Nearest-neighbor classifiers were introduced in 1951 by Fix and Hodges [FH51]. A comprehensive collection of articles on nearest neighbor classification can be found in Dasarathy [Das91]. Additional references can be found in many texts on classification, such as Duda, Hart, and Stork [DHS01] and James [Jam85], as well as articles by Cover and Hart [CH67] and Fukunaga and Hummels [FH87]. Their integration with attribute-weighting and the pruning of noisy instances is described in Aha [Aha92]. The use of search trees to improve nearest neighbor classification time is detailed in Friedman, Bentley, and Finkel[FBF77]. The partial distance method was proposed by researchers in vector quantization and compression. It is outlined in Gersho and Gray [GG92]. The editing method for removing "useless" training tuples was first proposed by Hart [Har68]. The computational complexity of nearest neighbor classifiers is described in Preparata and Shamos [PS85]. References on case-based reasoning (CBR) include the texts [RS89, Kol93, Lea96], as well as [AP94]. For a list of business applications, see [All94]. Examples in medicine include CASEY by Koton [Kot88] and PROTOS by Bareiss, Porter, and Weir [BPW88], while Rissland and Ashley [RA87] is an example of CBR for law. CBR is available in several commerical software products. For texts on genetic algorithms, see [Gol89, Mic92, Mit96]. Rough sets were introduced in Pawlak [Paw91]. Concise summaries of rough set theory in data mining include Ziarko [Zia91], and Cios, Pedrycz, and Swiniarski [CPS98]. Rough sets have been used for feature reduction and expert system design in many applications, including [Zia91, LP97, Swi98]. Algorithms to reduce the computation intensity in finding reducts have been proposed in [SR92]. Fuzzy set theory was proposed by Lofti Zadeh in [Zad65, Zad83]. Additional descriptions can be found in [YZ94, Kec01].

There are many good textbooks that cover the techniques of regression. Examples include [Jam85, Dob90, JW92, Dev95, HC95, NKNW96, Agr96]. The book by Press, Teukolsky, Vetterling, and Flannery [PTVF96] and accompanying source code contain many statistical procedures, such as the method of least squares for both linear and multiple regression. Recent nonlinear regression models include projection pursuit and MARS (Friedman [Fri91]). Log-linear models are also known in the computer science literature as *multiplicative models*. For log-linear models from a computer science perspective, see Pearl [Pea88]. Regression trees (Breiman, Friedman, Olshen, and Stone [BFOS84]) are often comparable in performance with other regression methods, particularly when there exist many higher-order dependencies among the predictor variables. For model trees, see Quinlan [Qui92].

Methods for data cleaning and data transformation are discussed in Kennedy, Lee, Van Roy, et al. [KLV$^+$98], Weiss and Indurkhya [WI98], Pyle [Pyl99], and Chapter 2 of this book. Issues involved in estimating classifier accuracy are described in Weiss and Kulikowski [WK91] [NEW and Witten and Frank] [WF00]. The use of stratified 10-fold cross-validation for estimating classifier accuracy is recommended over the holdout, cross-validation, leave-one-out (Stone [Sto74]) and bootstrapping (Efron and Tibshirani [ET93]) methods, based on a theoretical and empirical study by Kohavi [Koh95]. Bagging is proposed in Breiman [Bre96]. The boosting technique of Freund and Schapire [FS97] has been applied to several different classifiers, including decision tree induction (Quinlan [Qui96]) and naive Bayesian classification (Elkan [Elk97]). Sensitivity, specificity, and precision are discussed in Frakes and Baeza-Yates [FBY92]. For ROC analysis, see Egan [Ega75] and Swets [Swe88].

The University of California at Irvine (UCI) maintains a Machine Learning Repository of data sets, for the development and testing of classification algorithms, and a Knowledge Discovery in Databases (KDD) Archive, an online repository of large data sets that encompasses a wide variety of data types, analysis tasks, and application areas. For information on these two repositories, see *http://www.ics.uci.edu/˜mlearn/MLRepository.html* and *http://kdd.ics.uci.edu.*

No classification method is superior over all others for all data types and domains. Empirical comparisons of classification methods include [Qui88, SMT91, BCP93, CM94, MST94, BU95], and [LLS00].

# Bibliography

[Agr96] A. Agresti. *An Introduction to Categorical Data Analysis.* John Wiley & Sons, 1996.

[Aha92] D. Aha. Tolerating noisy, irrelevant, and novel attributes in instance-based learning algorithms. *International Journal of Man-Machine Studies*, 36:267–287, 1992.

[All94] B. P. Allen. Case-based reasoning: Business applications. *Comm. ACM*, 37:40–42, 1994.

[Alp04] E. Alpaydin. *Introduction to Machine Learning (Adaptive Computation and Machine Learning.* MIT Press, 2004.

[AP94] A. Aamodt and E. Plazas. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI Comm.*, 7:39–52, 1994.

[Avn95] S. Avner. Discovery of comprehensible symbolic rules in a neural network. In *Proc. 1995 Int. Symp. Intelligence in Neural and Biological Systems*, pages 64–67, 1995.

[BA97] L. A. Breslow and D. W. Aha. Simplifying decision trees: A survey. *Knowledge Engineering Review*, 12:1–40, 1997.

[BCP93] D. E. Brown, V. Corruble, and C. L. Pittard. A comparison of decision tree classifiers with back-propagation neural networks for multimodal classification problems. *Pattern Recognition*, 26:953–961, 1993.

[BFOS84] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees.* Wadsworth International Group, 1984.

[BGV92] B. Boser, I. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *Proc. Fifth Annual Workshop on Computational Learning Theory*, pages 144–152, ACM Press: San Mateo, CA, 1992.

[Bis95] C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford, UK: Oxford University Press, 1995.

[BM98] E. Bloedorn and R. S. Michalski. Data-driven constructive induction: A methodology and its applications. In H. Liu H. Motoda, editor, *Feature Selection for Knowledge Discovery and Data Mining.* Kluwer Academic Publishers, 1998.

[BN92] W. L. Buntine and T. Niblett. A further comparison of splitting rules for decision-tree induction. *Machine Learning*, 8:75–85, 1992.

[BPW88] E. R. Bareiss, B. W. Porter, and C. C. Weir. Protos: An exemplar-based learning apprentice. *Intl. J. of Man-Machine Studies*, 29:549–561, 1988.

[Bre96] L. Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996.

[BU95] C. E. Brodley and P. E. Utgoff. Multivariate decision trees. *Machine Learning*, 19:45–77, 1995.

[Bun94] W. L. Buntine. Operations for learning with graphical models. *Journal of Artificial Intelligence Research*, 2:159–225, 1994.

[Bur98] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–168, 1998.

[Cat91] J. Catlett. *Megainduction: Machine Learning on Very large Databases.* Ph.D. Thesis, University of Sydney, 1991.

[CH67] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Trans. Information Theory*, 13:21–27, 1967.

[CH92] G. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9:309–347, 1992.

[CM94] S. P. Curram and J. Mingers. Neural networks, decision tree induction and discriminant analysis: An empirical comparison. *J. Operational Research Society*, 45:440–450, 1994.

[CN89] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[Coh95] W. Cohen. Fast effective rule induction. In *Proc. 1995 Int. Conf. Machine Learning (ICML'95)*, pages 115–123, Tahoe City, CA, July 1995.

[Coo90] G. F. Cooper. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence*, 42:393–405, 1990.

[CPS98] K. Cios, W. Pedrycz, and R. Swiniarski. *Data Mining Methods for Knowledge Discovery.* Kluwer Academic Publishers, 1998.

[CR95] Y. Chauvin and D. Rumelhart. *Backpropagation: Theory, Architectures, and Applications.* Hillsdale, NJ: Lawrence Erlbaum Assoc., 1995.

[Cra89] S. L. Crawford. Extensions to the cart algorithm. *Intl. J. of Man-Machine Studies*, 31:197–217, Aug. 1989.

[CS93a] P. K. Chan and S. J. Stolfo. Experiments on multistrategy learning by metalearning. In *Proc. 2nd. Int. Conf. Information and Knowledge Management*, pages 314–323, 1993.

[CS93b] P. K. Chan and S. J. Stolfo. Metalearning for multistrategy and parallel learning. In *Proc. 1993 Int. Workshop Multistrategy Learning*, pages 150–165, 1993.

[CS96] M. W. Craven and J. W. Shavlik. Extracting tree-structured representations of trained networks. In D. Touretzky and M. Mozer M. Hasselmo, editors, *Advances in Neural Information Processing Systems.* Cambridge, MA: MIT Press, 1996.

[CS97] M. W. Craven and J. W. Shavlik. Using neural networks in data mining. *Future Generation Computer Systems*, 13:211–229, 1997.

[CTTX05] G. Cong, K.-Lee Tan, A. K. H. Tung, and X. Xu. Mining top-k covering rule groups for gene expression data. In *Proc. 2005 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'05)*, Baltimore, MD, June 2005.

[Das91] B. V. Dasarathy. *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques.* IEEE Computer Society Press, Las Alamitos, CA, 1991.

[DBK$^+$97] H. Drucker, C. J. C. Burges, L. Kaufman, A. Smola, and V. N. Vapnik. Support vector regression machines. In M. Mozer, M. Jordan, , and T. Petsche, editors, *Advances in Neural Information Processing Systems 9*, pages 155–161. MIT Press: Cambridge, MA, 1997.

[Dev95] J. L. Devore. *Probability and Statistics for Engineering and the Science, 4th ed.* Duxbury Press, 1995.

[DHS01] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification, 2ed.* John Wiley and Sons, Inc.: New York, 2001.

[DL99] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *Proc. 1999 Int. Conf. Knowledge Discovery and Data Mining (KDD'99)*, pages 43–52, San Diego, CA, Aug. 1999.

[DLR77] A. Dempster, N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. of the Royal Statistical Society*, 39:1–38, 1977.

[Dob90] A. J. Dobson. *An Introduction to Generalized Linear Models.* Chapman and Hall, 1990.

[Dom94] P. Domingos. The RISE system: Conquering without separating. In *Proc. 1994 IEEE Int. Conf. Tools with Artificial Intelligence (TAI'94)*, pages 704–707, New Orleans, LA, 1994.

[DP96] P. Domingos and M. Pazzani. Beyond independence: Conditions for the optimality of the simple Bayesian classifier. In *Proc. 1996 Int. Conf. Machine Learning (ML'96)*, pages 105–112, 1996.

[Ega75] J. P. Egan. *Signal detection theory and ROC analysis.* Academic Press: New York, 1975.

[Elk97] C. Elkan. Boosting and naive Bayesian learning. In *Technical Report CS97-557*, Dept. Computer Science and Engineering, Univ. Calif. at San Diego, Sept. 1997.

[ET93] B. Efron and R. Tibshirani. *An Introduction to the Bootstrap.* New York: Chapman & Hall, 1993.

[FBF77] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Math Software*, 3:209–226, 1977.

[FBY92] W. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms.* Prentice Hall, 1992.

[FH51] E. Fix and J. L. Hodges Jr. Discriminatory analysis, non-parametric discrimination: consistency properties. In *Technical Report 21-49-004(4)*, USAF School of Aviation Medicine, Randolph Field, Texas, 1951.

[FH87] K. Fukunaga and D. Hummels. Bayes error estimation using parzen and k-nn procedure. In *IEEE Trans. Pattern Analysis and Machine Learning*, pages 634–643, 1987.

[FI90] U. M. Fayyad and K. B. Irani. What should be minimized in a decision tree? In *Proc. 1990 Nat. Conf. Artificial Intelligence (AAAI'90)*, pages 749–754, AAAI/MIT Press, 1990.

[FI92] U. M. Fayyad and K. B. Irani. The attribute selection problem in decision tree generation. In *Proc. 1992 Nat. Conf. Artificial Intelligence (AAAI'92)*, pages 104–110, AAAI/MIT Press, 1992.

[FL90] S. Fahlman and C. Lebiere. The cascade-correlation learning algorithm. In *Technical Report CMU-CS-90-100*, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1990.

[Fle87] R. Fletcher. *Practical Methods of Optimization.* John Wiley and Sons, New York, 1987.

[Fri91] J. H. Friedman. Multivariate adaptive regression. *Annalsof Statistics*, 19:1–141, 1991.

[FS97] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55:119–139, 1997.

[Fu94] L. Fu. *Neural Networks in Computer Intelligence.* McGraw-Hill, 1994.

[FW94] J. Furnkranz and G. Widmer. Incremental reduced error pruning. In *Proc. 1994 Int. Conf. Machine Learning (ICML'94)*, pages 70–77, New Brunswick, NJ, 1994.

[Gal93] S. I. Gallant. *Neural Network Learning and Expert Systems.* Cambridge, MA: MIT Press, 1993.

[GG92] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression.* Kluwer, Boston, MA, 1992.

[GGRL99] J. Gehrke, V. Ganti, R. Ramakrishnan, and W.-Y. Loh. BOAT-optimistic decision tree construction. In *Proc. 1999 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'99)*, pages 169–180, Philadelphia, PA, June 1999.

[Gol89] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley: Reading, MA, 1989.

[GRG98] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest: A framework for fast decision tree construction of large datasets. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 416–427, New York, NY, Aug. 1998.

[Har68] P. E. Hart. The condensed nearest neighbor rule. *IEEE Transactions on Information Theory*, 14:515–516, 1968.

[Hay99] S. S. Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall, 1999.

[HB88] S. J. Hanson and D. J. Burr. Minkowski back-propagation: Learning in connectionist models with non-euclidean error signals. In *Neural Information Processing Systems*, American Institute of Physics, 1988.

[HC95] R. V. Hogg and A. T. Craig. *Introduction to Mathematical Statistics, 5th ed.* Prentice Hall, 1995.

[Hec96] D. Heckerman. Bayesian networks for knowledge discovery. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 273–305. Cambridge, MA: MIT Press, 1996.

[HGC95] D. Heckerman, D. Geiger, and D. M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20:197–243, 1995.

[HKP91] J. Hertz, A. Krogh, and R. G. Palmer. *Introduction to the Theory of Neural Computation.* Addison Wesley: Reading, MA., 1991.

[HMM86] J. Hong, I. Mozetic, and R. S. Michalski. AQ15: Incremental learning of attribute-based descriptions from examples, the method and user's guide. In *Report ISG 85-5, UIUCDCS-F-86-949,*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1986.

[HMS66] E. B. Hunt, J. Marin, and P. T. Stone. *Experiments in Induction.* New York: Academic Press, 1966.

[HN90] R. Hecht-Nielsen. *Neurocomputing.* Reading, MA: Addison Wesley, 1990.

[HPS97] J. Hosking, E. Pednault, and M. Sudan. A statistical perspective on data mining. *Future Generation Computer Systems*, 13:117–134, 1997.

[HSG90] S. A. Harp, T. Samad, and A. Guha. Designing application-specific neural networks using the genetic algorithm. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems II*, pages 447–454. Morgan Kaufmann: San Mateo, CA, 1990.

[HTF01] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer-Verlag, New York, 2001.

[Jac88] R. Jacobs. Increased rates of convergence through learning rate adaptation. *Neural Networks*, 1:295–307, 1988.

[Jam85] M. James. *Classification Algorithms.* John Wiley & Sons, 1985.

[Jen96] F. V. Jensen. *An Introduction to Bayesian Networks.* Springer Verlag, 1996.

[Joh97] G. H. John. *Enhancements to the Data Mining Process.* Ph.D. Thesis, Computer Science Dept., Stanford University, 1997.

[JW92] R. A. Johnson and D. A. Wichern. *Applied Multivariate Statistical Analysis, 3rd ed.* Prentice Hall, 1992.

[Kas80] G. V. Kass. An exploratory technique for investigating large quantities of categorical data. *Applied Statistics*, 29:119–127, 1980.

[Kec01] V. Kecman. *Learning and Soft Computing.* MIT Press, Cambridge, MA, 2001.

[KH97] I. Kononenko and S. J. Hong. Attribute selection for modeling. *Future Generation Computer Systems*, 13:181–195, 1997.

[KLV$^+$98] R. L Kennedy, Y. Lee, B. Van Roy, C. D. Reed, and R. P. Lippman. *Solving Data Mining Problems Through Pattern Recognition.* Upper Saddle River, NJ: Prentice Hall, 1998.

[KM90] Y. Kodratoff and R. S. Michalski. *Machine Learning, An Artificial Intelligence Approach, Vol. 3.* Morgan Kaufmann, 1990.

[Koh95] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proc. 14th Joint Int. Conf. Artificial Intelligence (IJCAI'95)*, volume 2, pages 1137–1143, Montreal, Canada, Aug. 1995.

[Kol93] J. L. Kolodner. *Case-Based Reasoning.* Morgan Kaufmann, 1993.

[Kon95] I. Kononenko. On biases in estimating multi-valued attributes. In *Proc. 14th Joint Intl. Conf. Artificial Intelligence (IJCAI'95)*, volume 2, pages 1034–1040, Montreal, Canada, Aug. 1995.

[Kot88] P. Koton. Reasoning about evidence in causal explanation. In *Proc. 7th Nat. Conf. Artificial Intelligence (AAAI'88)*, pages 256–263, Aug. 1988.

[KWG$^+$97] M. Kamber, L. Winstone, W. Gong, S. Cheng, and J. Han. Generalization and decision tree induction: Efficient classification in data mining. In *Proc. 1997 Int. Workshop Research Issues on Data Engineering (RIDE'97)*, pages 111–120, Birmingham, England, April 1997.

[Lam98] W. Lam. Bayesian network refinement via machine learning approach. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 20:240–252, 1998.

[Lan96] P. Langley. *Elements of Machine Learning.* Morgan Kaufmann, 1996.

[Lau95] S. L. Lauritzen. The EM algorithm for graphical association models with missing data. *Computational Statistics and Data Analysis*, 19:191–201, 1995.

[LDR00] J. Li, G. Dong, and K. Ramamohanrarao. Making use of the most expressive jumping emerging patterns for classification. In *Proc. 2000 Pacific-Asia Conf. Knowledge Discovery and Data Mining (PAKDD'00)*, pages 220–232, Kyoto, Japan, April 2000.

[LDS90] Y. Le Cun, J. S. Denker, and S. A. Solla. Optimal brain damage. In D. Touretzky, editor, *Advances in neural Information Processing Systems, 2.* Morgan Kaufmann, 1990.

[Le98] H. Liu and H. Motoda (eds.). *Feature Extraction, Construction, and Selection: A Data Mining Perspective.* Kluwer Academic Publishers, 1998.

[Lea96] D. B. Leake. CBR in context: The present and future. In D. B. Leake, editor, *Cased-Based Reasoning: Experiences, Lessons, and Future Directions*, pages 3–30. AAAI Press, 1996.

[LGT97] S. Lawrence, C. L Giles, and A. C. Tsoi. Symbolic conversion, grammatical inference and rule extraction for foreign exchange rate prediction. In Y. Abu-Mostafa and A. S. Weigend P. N Refenes, editors, *Neural Networks in the Captial Markets.* World Scietific, 1997.

[LHM98] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *Proc. 1998 Int. Conf. Knowledge Discovery and Data Mining (KDD'98)*, pages 80–86, New York, NY, Aug. 1998.

[LHP01] W. Li, J. Han, and J. Pei. CMAR: Accurate and efficient classification based on multiple class-association rules. In *Proc. 2001 Int. Conf. Data Mining (ICDM'01)*, pages 369–376, San Jose, CA, Nov. 2001.

[LLS00] T.-S. Lim, W.-Y. Loh, and Y.-S. Shih. A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. *Machine Learning*, 39, 2000.

[LM97] K. Laskey and S. Mahoney. Network fragments: Representing knowledge for constructing probabilistic models. In *Proc. 13th Annual Conf. on Uncertainty in Artificial Intelligence*, pages 334–341, Morgan Kaufmann: San Francisco, CA, Aug. 1997.

[LM98] H. Liu and H. Motoda. *Feature Selection for Knowledge Discovery and Data Mining.* Kluwer Academic Publishers, 1998.

[LP97] A. Lenarcik and Z. Piasta. Probabilistic rough classifiers with mixture of discrete and continuous variables. In T. Y. Lin N. Cercone, editor, *Rough Sets and Data Mining: Analysis for Imprecise Data*, pages 373–383. Kluwer Academic Publishers, 1997.

[LS97] W. Y. Loh and Y. S. Shih. Split selection methods for classification trees. *Statistica Sinica*, 7:815–840, 1997.

[LSBZ87] P. Langley, H. A. Simon, G. L. Bradshaw, and J. M. Zytkow. *Scientific Discovery: Computational Explorations of the Creative Processes.* MIT Press: Cambridge, Massachusetts, 1987.

[LSL95] H. Lu, R. Setiono, and H. Liu. Neurorule: A connectionist approach to data mining. In *Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95)*, pages 478–489, Zurich, Switzerland, Sept. 1995.

[LSW97] B. Lent, A. Swami, and J. Widom. Clustering association rules. In *Proc. 1997 Int. Conf. Data Engineering (ICDE'97)*, pages 220–231, Birmingham, England, April 1997.

[LV88] W. Y. Loh and N. Vanichsetakul. Tree-structured classificaiton via generalized discriminant analysis. *Journal of the American Statistical Association*, 83:715–728, 1988.

[Mag94] J. Magidson. The CHAID approach to segmentation modeling: CHI-squared automatic interaction detection. In R. P. Bagozzi, editor, *Advanced Methods of Marketing Research*, pages 118–159. Blackwell Business, Cambridge Massachusetts, 1994.

[MAR96] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proc. 1996 Int. Conf. Extending Database Technology (EDBT'96)*, Avignon, France, Mar. 1996.

[MBK98] R. S. Michalski, I. Brakto, and M. Kubat. *Machine Learning and Data Mining: Methods and Applications.* John Wiley & Sons, 1998.

[MCM83] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell. *Machine Learning, An Artificial Intelligence Approach, Vol. 1.* Morgan Kaufmann, 1983.

[MCM86] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell. *Machine Learning, An Artificial Intelligence Approach, Vol. 2.* Morgan Kaufmann, 1986.

[MFS95] D. Malerba, E. Floriana, and G. Semeraro. A further comparison of simplification methods for decision tree induction. In D. Fisher H. Lenz, editor, *Learning from Data: AI and Statistics.* Springer-Verlag, 1995.

[MH95] J. K. Martin and D. S. Hirschberg. The time complexity of decision tree induction. In *Technical Report ICS-TR 95-27*, Dept. Information and Computer Science, Univ.alifornia, Irvine, Aug. 1995.

[Mic92] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer Verlag, 1992.

[Min89] J. Mingers. An empirical comparison of pruning methods for decision-tree induction. *Machine Learning*, 4:227–243, 1989.

[Mit96] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[Mit97] T. M. Mitchell. *Machine Learning.* McGraw Hill, 1997.

[MK91] M. Manago and Y. Kodratoff. Induction of decision trees from complex structured data. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 289–306. AAAI/MIT Press, 1991.

[MM95] J. Major and J. Mangano. Selecting among rules induced from a hurricane database. *Journal of Intelligent Information Systems*, 4:39–52, 1995.

[MN89] M. Mézard and J.-P. Nadal. Learning in feedforward layered networks: The tiling algorithm. *Journal of Physics*, 22:2191–2204, 1989.

[MP69] M. L. Minsky and S. Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, MA, 1969.

[MRA95] M. Metha, J. Rissanen, and R. Agrawal. MDL-based decision tree pruning. In *Proc. 1995 Int. Conf. Knowledge Discovery and Data Mining (KDD'95)*, pages 216–221, Montreal, Canada, Aug. 1995.

[MST94] D. Michie, D. J. Spiegelhalter, and C. C. Taylor. *Machine Learning, Neural and Statistical Classification.* Ellis Horwood, 1994.

[MT94] R. S. Michalski and G. Tecuci. *Machine Learning, A Multistrategy Approach, Vol. 4.* Morgan Kaufmann, 1994.

[Mur98] S. K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data Mining and Knowledge Discovery*, 2:345–389, 1998.

[MW99] D. Meretakis and B. Wüthrich. Extending naive Bayes classifiers using long itemsets. In *Proc. 1999 Int. Conf. Knowledge Discovery and Data Mining (KDD'99)*, pages 165–174, San Diego, CA, Aug. 1999.

[NB86] T. Niblett and I. Bratko. Learning decision rules in noisy domains. In M. A. Bramer, editor, *Expert Systems '86: Research and Development in Expert Systems III*, pages 25–34. British Computer Society Specialist Group on Expert Systems, Dec. 1986.

[NKNW96] J. Neter, M. H. Kutner, C. J. Nachtsheim, and L. Wasserman. *Applied Linear Statistical Models, 4th ed.* Irwin: Chicago, 1996.

[NW99] J. Nocedal and S. J. Wright. *Numerical Optimization.* Springer Verlag, New York, 1999.

[OFG97] E. Osuna, R. Freund, and F. Girosi. An improved training algorithm for support vector machines. In *Proc. of the 1997 IEEE Workshop on Neural Networks for Signal Processing (NNSP'97)*, pages 276–285, Amelia Island, FL, Sept. 1997.

[Pag89] G. Pagallo. Learning DNF by decision trees. In *Proc. 1989 Int. Joint Conf. Artificial Intelligence (IJCAI'89)*, pages 639–644, Morgan Kaufmann Publishers, 1989.

[Paw91] Z. Pawlak. *Rough Sets, Theoretical Aspects of Reasoning about Data.* Kluwer Academic Publishers, 1991.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems.* Palo Alto, CA: Morgan Kauffman, 1988.

[PKMT99] A. Pfeffer, D. Koller, B. Milch, and K. Takusagawa. SPOOK: A system for probabilistic object-oriented knowledge representation. In *Proc. 15th Annual Conf. on Uncertainty in Artificial Intelligence*, pages 541–550, Morgan Kaufmann: San Francisco, CA, 1999.

[Pla98] J. C. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schotolkopf, C. J. C. Burges, , and A. Smola, editors, *Advances in Kernel Methods – Support Vector Learning*, pages 185–208. MIT Press: Cambridge, MA, 1998.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[PTVF96] W. H. Press, S. A. Teukolosky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1996.

[Pyl99] D. Pyle. *Data Preparation for Data Mining.* Morgan Kaufmann, 1999.

[QCJ93] J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In *Proc. 1993 European Conf. Machine Learning*, pages 3–20, Vienna, Austria, 1993.

[QR89] J. R. Quinlan and R. L. Rivest. Inferring decision trees using the minimum description length principle. *Information and Computation*, 80:227–248, Mar. 1989.

[Qui86] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Qui87] J. R. Quinlan. Simplifying decision trees. *Internation Journal of Man-Machine Studies*, 27:221–234, 1987.

[Qui88] J. R. Quinlan. An empirical comparison of genetic and decision-tree classifiers. In *Proc. 1988 Int. Conf. Machine Learning (ML'88)*, pages 135–141, San Mateo, CA: Morgan Kaufmann, 1988.

[Qui90] J. R. Quinlan. Learning logic definitions from relations. *Machine Learning*, 5:139–166, 1990.

[Qui92] J. R. Quinlan. Learning with continuous classes. In *Proc. 1992 Australian Joint Conf. on Artificial Intelligence*, pages 343–348, Hobart, Tasmania, 1992.

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[Qui96] J. R. Quinlan. Bagging, boosting, and C4.5. In *Proc. 1996 Nat. Conf. Artificial Intelligence (AAAI'96)*, volume 1, pages 725–730, Portland, OR, Aug. 1996.

[RA87] E. L. Rissland and K. Ashley. HYPO: A case-based system for trade secret law. In *Proc. 1st Intl. Conf. on Artificial Intelligence and Law*, pages 60–66, May 1987.

[RBKK95] S. Russell, J. Binder, D. Koller, and K. Kanazawa. Local learning in probabilistic networks with hidden variables. In *Proc. 1995 Joint Int. Conf. Artificial Intelligence (IJCAI'95)*, pages 1146–1152, Montreal, Canada, Aug. 1995.

[RHW86] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart J. L. McClelland, editor, *Parallel Distributed Processing.* Cambridge, MA: MIT Press, 1986.

[Rip96] B. D. Ripley. *Pattern Recognition and Neural Networks.* Cambridge: Cambridge University Press, 1996.

[RM86] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Processing.* Cambridge, MA: MIT Press, 1986.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, 1995.

[Ros58] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–498, 1958.

[RS89] C. Riesbeck and R. Schank. *Inside Case-Based Reasoning.* Hillsdale, NJ: Lawrence Erlbaum, 1989.

[RS98] R. Rastogi and K. Shim. Public: A decision tree classifer that integrates building and pruning. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 404–415, New York, NY, Aug. 1998.

[SAM96] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. 1996 Int. Conf. Very Large Data Bases (VLDB'96)*, pages 544–555, Bombay, India, Sept. 1996.

[SBSW99] B. Schlkopf, P. L. Bartlett, A. Smola, and R. Williamson. Shrinking the tube: a new support vector regression algorithm. In M. S. Kearns, S. A. Solla, , and D. A. Cohn, editors, *Advances in Neural Information Processing Systems 11,*, pages 330–336. MIT Press: Cambridge, MA, 1999.

[Sch86] J. C. Schlimmer. Learning and representation change. In *Proc. 1986 Nat. Conf. Artificial Intelligence (AAAI'86)*, pages 511–515, Phildelphia, PA, 1986.

[SD90] J. W. Shavlik and T. G. Dietterich. *Readings in Machine Learning.* Morgan Kaufmann, 1990.

[SF86] J. C. Schlimmer and D. Fisher. A case study of incremental concept induction. In *Proc. 1986 Nat. Conf. Artificial Intelligence (AAAI'86)*, pages 496–501, Phildelphia, PA, 1986.

[SG92] P. Smyth and R. M. Goodman. An information theoretic approach to rule induction. *IEEE Trans. Knowledge and Data Engineering*, 4:301–316, 1992.

[Shi00] Y.-S. Shih. Families of splitting criteria for classification trees. In *Statistics and Computing (to appear)*, 2000.

[SMT91] J. W. Shavlik, R. J. Mooney, and G. G. Towell. Symbolic and neural learning algorithms: An experimental comparison. *Machine Learning*, 6:111–144, 1991.

[SN88] K. Saito and R. Nakano. Medical diagnostic expert system based on PDP model. In *Proc. 1988 IEEE International Conf. Neural Networks*, pages 225–262, San Mateo, CA, 1988.

[SR81] R. Sokal and F. Rohlf. *Biometry.* Freeman: San Francisco, CA, 1981.

[SR92] A. Skowron and C. Rauszer. The discernibility matrices and functions in information systems. In R. Slowinski, editor, *Intelligent Decision Support, Handbook of Applications and Advances of the Rough Set Theory*, pages 331–362. Kluwer Academic Publishers, 1992.

[Sto74] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society*, 36:111–147, 1974.

[SW49] C. E. Shannon and W. Weaver. *The mathematical theory of communication.* University of Illinois Press, Urbana, IL, 1949.

[Swe88] J. Swets. Measuring the accuracy of diagnostic systems. *Science*, 240:1285–1293, 1988.

[Swi98] R. Swiniarski. Rough sets and principal component analysis and their applications in feature extraction and selection, data model building and classification. In S. Pal A. Skowron, editor, *Fuzzy Sets, Rough Sets and Decision Making Processes.* New York: Springer-Verlag, 1998.

[TS93] G. G. Towell and J. W. Shavlik. Extracting refined rules from knowledge-based neural networks. *Machine Learning*, 13:71–101, Oct. 1993.

[TSK05] P. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining.* Addison Wesley, 2005.

[UBC97] P. E. Utgoff, N. C. Berkman, and J. A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29:5–44, 1997.

[UFS91] R. Uthurusamy, U. M. Fayyad, and S. Spangler. Learning useful rules from inconclusive data. In G. Piatetsky-Shapiro and W. J. Frawley, editors, *Knowledge Discovery in Databases*, pages 141–157. AAAI/MIT Press, 1991.

[Utg88] P. E. Utgoff. An incremental ID3. In *Proc. Fifth Int. Conf. Machine Learning*, pages 107–120, San Mateo, CA, 1988.

[Vap95] V. N. Vapnik. *The Nature of Statistical Learning Theory.* Springer-Verlag, New York, 1995.

[Vap98] V. N. Vapnik. *Statistical Learning Theory.* Wiley, New York, 1998.

[VC71] V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16:264–280, 1971.

[WF00] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 2000.

[WF05] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques, 2ed.* Morgan Kaufmann, 2005.

[WI98] S. M. Weiss and N. Indurkhya. *Predictive Data Mining.* Morgan Kaufmann, 1998.

[WK91] S. M. Weiss and C. A. Kulikowski. *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems.* Morgan Kaufman, 1991.

[WRL94] B. Widrow, D. E. Rumelhart, and M. A. Lehr. Neural networks: Applications in industry, business and science. *Comm. ACM*, 37:93–105, 1994.

[XOJ00] Y. Xiang, K. G. Olesen, and F. V. Jensen. Practical issues in modeling large diagnostic systems with multiply sectioned bayesian networks. *Intl. J. of Pattern Recognition and Artificial Intelligence (IJPRAI)*, 14:59–71, 2000.

[YH03] X. Yin and J. Han. CPAR: Classification based on predictive association rules. In *Proc. 2003 SIAM Int. Conf. Data Mining (SDM'03)*, pages 331–335, San Fransisco, CA, May 2003.

[YYH03] H. Yu, J. Yang, and J. Han. Classifying large data sets using SVM with hierarchical clusters. In *Proc. 2003 ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD'03)*, Washington, D.C., Aug. 2003.

[YZ94] R. R. Yager and L. A. Zadeh. *Fuzzy Sets, Neural Networks and Soft Computing.* Van Nostrand Reinhold, New York, 1994.

[Zad65] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[Zad83] L. Zadeh. Commonsense knowledge representation based on fuzzy logic. *Computer*, 16:61–65, 1983.

[Zia91] W. Ziarko. The discovery, analysis, and representation of data dependencies in databases. In G. Piatetsky-Shapiro W. J. Frawley, editor, *Knowledge Discovery in Databases*, pages 195–209. AAAI Press, 1991.