



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. | What Is Classification? What Is Prediction?

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

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

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

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

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

¹Each attribute represents a “feature” of 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., $\mathbf{X} = (x_1, x_2, x_3)$.

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

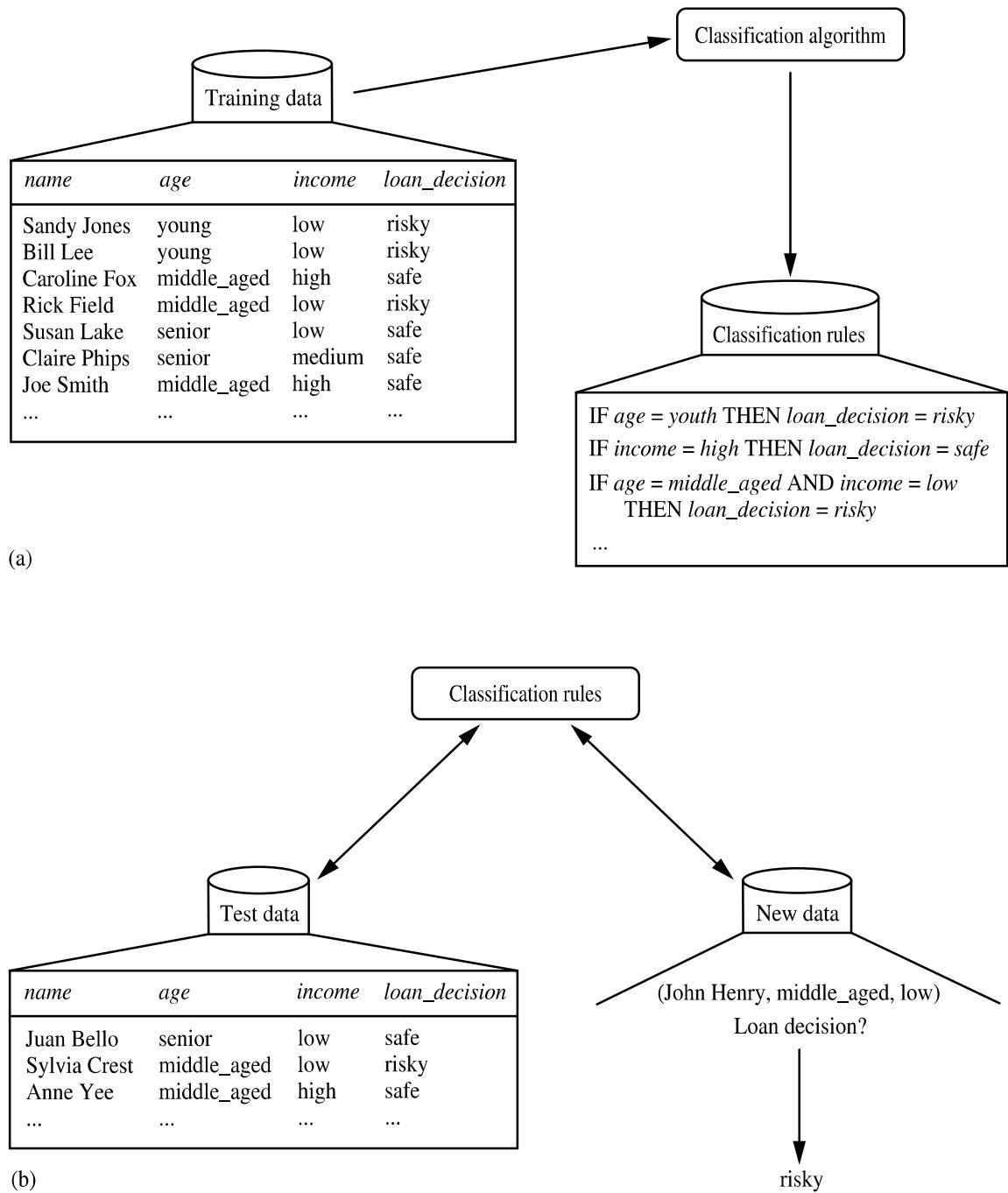


Figure 6.1 The data classification process: (a) *Learning*: Training data are analyzed by a classification algorithm. Here, the class label attribute is *loan_decision*, and the learned model or classifier is represented in the form of classification rules. (b) *Classification*: Test data are used to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

to which class each training tuple belongs). It contrasts with **unsupervised learning** (or **clustering**), in which the class label of each training tuple is not known, and the number or set of classes to be learned may not be known in advance. For example, if we did not have the *loan_decision* data available for the training set, we could use clustering to try to

determine “groups of like tuples,” which may correspond to risk groups within the loan application data. Clustering is the topic of Chapter 7.

This first step of the classification process can also be viewed as the learning of a mapping or function, $y = f(X)$, that can predict the associated class label y of a given tuple X . In this view, we wish to learn a mapping or function that separates the data classes. Typically, this mapping is represented in the form of classification rules, decision trees, or mathematical formulae. In our example, the mapping is represented as classification rules that identify loan applications as being either safe or risky (Figure 6.1(a)). The rules can be used to categorize future data tuples, as well as provide deeper insight into the database contents. They also provide a compressed representation of the data.

“*What about classification accuracy?*” In the second step (Figure 6.1(b)), the model is used for classification. First, the predictive accuracy of the classifier is estimated. If we were to use the training set to measure the accuracy of the classifier, this estimate would likely be optimistic, because the classifier tends to **overfit** the data (i.e., during learning it may incorporate some particular anomalies of the training data that are not present in the general data set overall). Therefore, a **test set** is used, made up of **test tuples** and their associated class labels. These tuples are randomly selected from the general data set. They are independent of the training tuples, meaning that they are not used to construct the classifier.

The **accuracy** of a classifier on a given test set is the percentage of test set tuples that are correctly classified by the classifier. The associated class label of each test tuple is compared with the learned classifier’s class prediction for that tuple. Section 6.13 describes several methods for estimating classifier accuracy. If the accuracy of the classifier is considered acceptable, the classifier can be used to classify future data tuples for which the class label is not known. (Such data are also referred to in the machine learning literature as “*unknown*” or “*previously unseen*” data.) For example, the classification rules learned in Figure 6.1(a) from the analysis of data from previous loan applications can be used to approve or reject new or future loan applicants.

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

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

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

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

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

6.2 Issues Regarding Classification and Prediction

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

6.2.1 Preparing the Data for Classification and Prediction

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

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

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

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

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

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

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

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

6.2.2 Comparing Classification and Prediction Methods

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

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

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

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

6.3 Classification by Decision Tree Induction

Decision tree induction is the learning of decision trees from class-labeled training tuples. A **decision tree** is a flowchart-like tree structure, where each **internal node** (nonleaf node) denotes a test on an attribute, each **branch** represents an outcome of the test, and each **leaf node** (or *terminal node*) holds a class label. The topmost node in a tree is the **root node**.

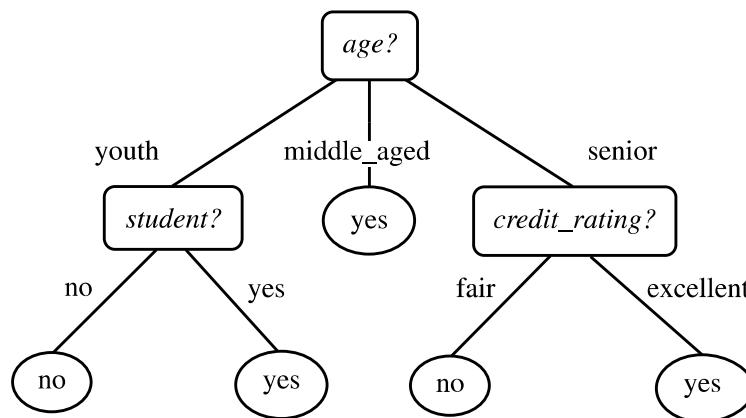


Figure 6.2 A decision tree for the concept *buys_computer*, indicating whether a customer at *AllElectronics* is likely to purchase a computer. Each internal (nonleaf) node represents a test on an attribute. Each leaf node represents a class (either *buys_computer* = yes or *buys_computer* = no).

A typical decision tree is shown in Figure 6.2. It represents the concept *buys_computer*, that is, it predicts whether a customer at *AllElectronics* is likely to purchase a computer. Internal nodes are denoted by rectangles, and leaf nodes are denoted by ovals. Some decision tree algorithms produce only *binary* trees (where each internal node branches to exactly two other nodes), whereas others can produce nonbinary trees.

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

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

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

6.3.1 Decision Tree Induction

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

ID3, C4.5, and CART adopt a greedy (i.e., nonbacktracking) approach in which decision trees are constructed in a top-down recursive divide-and-conquer manner. Most algorithms for decision tree induction also follow such a top-down approach, which

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

Input:

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

Output: A decision tree.

Method:

- (1) create a node N ;
- (2) if tuples in D are all of the same class, C then
 - (3) return N as a leaf node labeled with the class C ;
 - (4) if $attribute_list$ is empty then
 - (5) return N as a leaf node labeled with the majority class in D ; // majority voting
 - (6) apply $Attribute_selection_method(D, attribute_list)$ to find the “best” $splitting_criterion$;
 - (7) label node N with $splitting_criterion$;
 - (8) if $splitting_attribute$ is discrete-valued and
 - multiway splits allowed then // not restricted to binary trees
 - (9) $attribute_list \leftarrow attribute_list - splitting_attribute$; // 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 ;
 - (15) endfor
- (15) return N ;

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

starts with a training set of tuples and their associated class labels. The training set is recursively partitioned into smaller subsets as the tree is being built. A basic decision tree algorithm is summarized in Figure 6.3. At first glance, the algorithm may appear long, but fear not! It is quite straightforward. The strategy is as follows.

- The algorithm is called with three parameters: D , $attribute_list$, and $Attribute_selection_method$. We refer to D as a data partition. Initially, it is the complete set of training tuples and their associated class labels. The parameter $attribute_list$ is a list of attributes describing the tuples. $Attribute_selection_method$ specifies a heuristic procedure for selecting the attribute that “best” discriminates the given tuples according

to class. This procedure employs an attribute selection measure, such as information gain or the gini index. Whether the tree is strictly binary is generally driven by the attribute selection measure. Some attribute selection measures, such as the gini index, enforce the resulting tree to be binary. Others, like information gain, do not, therein allowing multiway splits (i.e., two or more branches to be grown from a node).

- The tree starts as a single node, N , representing the training tuples in D (step 1).⁵
- If the tuples in D are all of the same class, then node N becomes a leaf and is labeled with that class (steps 2 and 3). Note that steps 4 and 5 are terminating conditions. All of the terminating conditions are explained at the end of the algorithm.
- Otherwise, the algorithm calls *Attribute_selection_method* to determine the **splitting criterion**. The splitting criterion tells us which attribute to test at node N by determining the “best” way to separate or partition the tuples in D into individual classes (step 6). The splitting criterion also tells us which branches to grow from node N with respect to the outcomes of the chosen test. More specifically, the splitting criterion indicates the **splitting attribute** and may also indicate either a **split-point** or a **splitting subset**. The splitting criterion is determined so that, ideally, the resulting partitions at each branch are as “pure” as possible. A partition is **pure** if all of the tuples in it belong to the same class. In other words, if we were to split up the tuples in D according to the mutually exclusive outcomes of the splitting criterion, we hope for the resulting partitions to be as pure as possible.
- The node N is labeled with the splitting criterion, which serves as a test at the node (step 7). A branch is grown from node N for each of the outcomes of the splitting criterion. The tuples in D are partitioned accordingly (steps 10 to 11). There are three possible scenarios, as illustrated in Figure 6.4. Let A be the splitting attribute. A has v distinct values, $\{a_1, a_2, \dots, a_v\}$, based on the training data.
 1. *A is discrete-valued:* In this case, the outcomes of the test at node N correspond directly to the known values of A . A branch is created for each known value, a_j , of A and labeled with that value (Figure 6.4(a)). Partition D_j is the subset of class-labeled tuples in D having value a_j of A . Because all of the tuples in a given partition have the same value for A , then A need not be considered in any future partitioning of the tuples. Therefore, it is removed from *attribute_list* (steps 8 to 9).
 2. *A is continuous-valued:* In this case, the test at node N has two possible outcomes, corresponding to the conditions $A \leq \text{split_point}$ and $A > \text{split_point}$, respectively,

⁵The partition of class-labeled training tuples at node N is the set of tuples that follow a path from the root of the tree to node N when being processed by the tree. This set is sometimes referred to in the literature as the *family* of tuples at node N . We have referred to this set as the “tuples represented at node N ,” “the tuples that reach node N ,” or simply “the tuples at node N .” Rather than storing the actual tuples at a node, most implementations store pointers to these tuples.

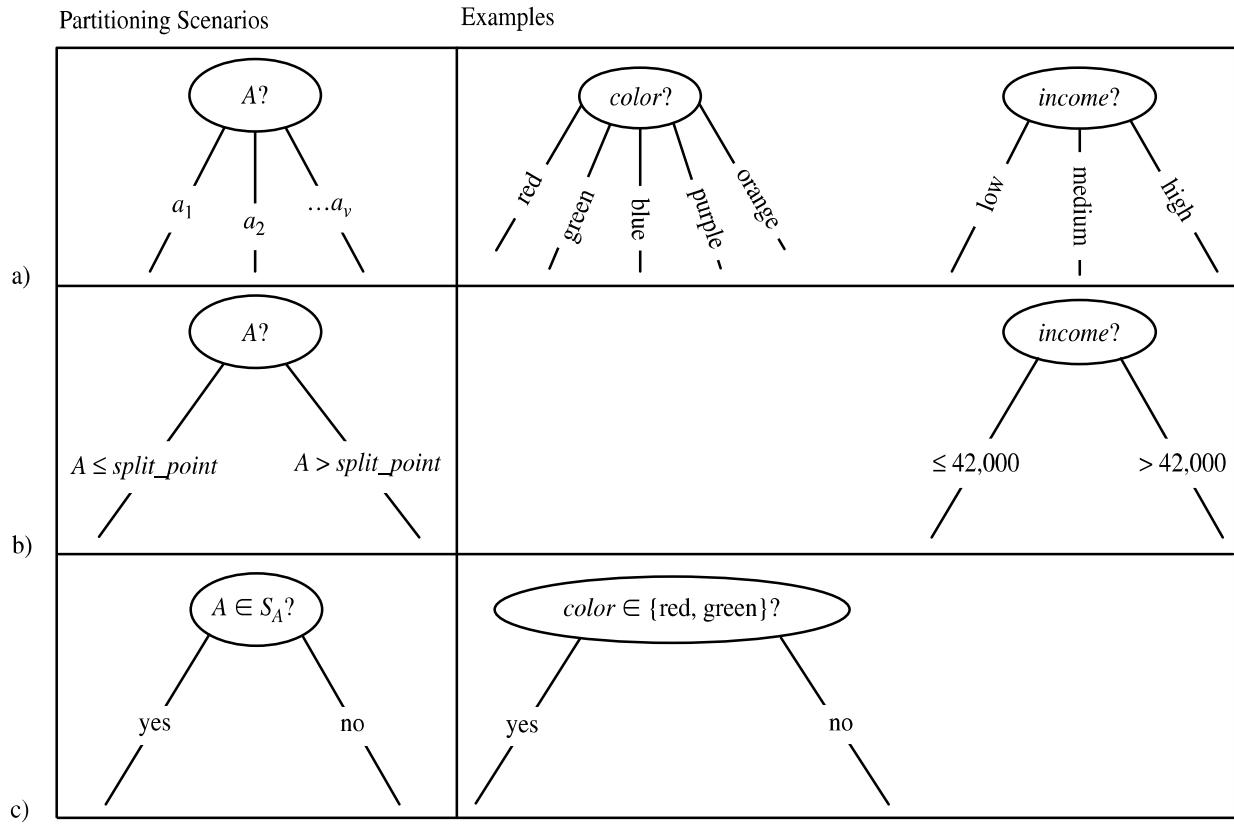


Figure 6.4 Three possibilities for partitioning tuples based on the splitting criterion, shown with examples. Let A be the splitting attribute. (a) If A is discrete-valued, then one branch is grown for each known value of A . (b) If A is continuous-valued, then two branches are grown, corresponding to $A \leq \text{split_point}$ and $A > \text{split_point}$. (c) If A is discrete-valued and a binary tree must be produced, then the test is of the form $A \in S_A$, where S_A is the splitting subset for A .

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

3. A is discrete-valued and a *binary tree* must be produced (as dictated by the attribute selection measure or algorithm being used): The test at node N is of the form “ $A \in S_A?$ ”. S_A is the splitting subset for A , returned by *Attribute_selection_method* as part of the splitting criterion. It is a subset of the known values of A . If a given tuple has value a_j of A and if $a_j \in S_A$, then the test at node N is satisfied. Two branches are grown from N (Figure 6.4(c)). By convention, the left branch out of N is labeled *yes* so that D_1 corresponds to the subset of class-labeled tuples in D

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

- The algorithm uses the same process recursively to form a decision tree for the tuples at each resulting partition, D_j , of D (step 14).
- The recursive partitioning stops only when any one of the following terminating conditions is true:
 1. All of the tuples in partition D (represented at node N) belong to the same class (steps 2 and 3), or
 2. There are no remaining attributes on which the tuples may be further partitioned (step 4). In this case, **majority voting** is employed (step 5). This involves converting node N into a leaf and labeling it with the most common class in D . Alternatively, the class distribution of the node tuples may be stored.
 3. There are no tuples for a given branch, that is, a partition D_j is empty (step 12). In this case, a leaf is created with the majority class in D (step 13).
- The resulting decision tree is returned (step 15).

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

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

Differences in decision tree algorithms include how the attributes are selected in creating the tree (Section 6.3.2) and the mechanisms used for pruning (Section 6.3.3). The basic algorithm described above requires one pass over the training tuples in D for each level of the tree. This can lead to long training times and lack of available memory when dealing with large databases. Improvements regarding the scalability of decision tree induction are discussed in Section 6.3.4. A discussion of strategies for extracting rules from decision trees is given in Section 6.5.2 regarding rule-based classification.

6.3.2 Attribute Selection Measures

An **attribute selection measure** is a heuristic for selecting the splitting criterion that “best” separates a given data partition, D , of class-labeled training tuples into individual classes. If we were to split D into smaller partitions according to the outcomes of the splitting criterion, ideally each partition would be pure (i.e., all of the tuples that fall into a given partition would belong to the same class). Conceptually, the “best” splitting criterion is the one that most closely results in such a scenario. Attribute selection

measures are also known as **splitting rules** because they determine how the tuples at a given node are to be split. The attribute selection measure provides a ranking for each attribute describing the given training tuples. The attribute having the best score for the measure⁶ 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 $i = 1, \dots, m$). Let $C_{i,D}$ be the set of tuples of class C_i in D . Let $|D|$ and $|C_{i,D}|$ denote the number of tuples in D and $C_{i,D}$, respectively.

Information gain

ID3 uses **information gain** as its attribute selection measure. This measure is based on pioneering work by Claude Shannon on information theory, which studied the value or “information content” of messages. Let node N represent or hold the tuples of partition D . The attribute with the highest information gain is chosen as the splitting attribute for node N . This attribute minimizes the information needed to classify the tuples in the resulting partitions and reflects the least randomness or “impurity” in these partitions. Such an approach minimizes the expected number of tests needed to classify a given tuple and guarantees that a simple (but not necessarily the simplest) tree is found.

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

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

where p_i is the probability that an arbitrary tuple in D belongs to class C_i and is estimated by $|C_{i,D}|/|D|$. A log function to the base 2 is used, because the information is encoded in bits. $\text{Info}(D)$ 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. $\text{Info}(D)$ 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, $\{a_1, a_2, \dots, a_v\}$, 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, $\{D_1, D_2, \dots, D_v\}$, 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

⁶Depending on the measure, either the highest or lowest score is chosen as the best (i.e., some measures strive to maximize while others strive to minimize).

of the tuples. That is, we would like for each partition to be pure. However, it is quite likely that the partitions will be impure (e.g., where a partition may contain a collection of tuples from different classes rather than from a single class). How much more information would we still need (after the partitioning) in order to arrive at an exact classification? This amount is measured by

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

The term $\frac{|D_j|}{|D|}$ acts as the weight of the j th partition. $\text{Info}_A(D)$ 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,

$$\text{Gain}(A) = \text{Info}(D) - \text{Info}_A(D). \quad (6.3)$$

In other words, $\text{Gain}(A)$ 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, ($\text{Gain}(A)$), 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 $\text{Info}_A(D)$).

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, $\{\text{yes}, \text{no}\}$); therefore, there are two distinct classes (that is, $m = 2$). Let class C_1 correspond to *yes* and class C_2 correspond to *no*. There are nine tuples of class *yes* and five tuples of class *no*. A (root) node N is created for the tuples in D . To find the splitting criterion for these tuples, we must compute the information gain of each attribute. We first use Equation (6.1) to compute the expected information needed to classify a tuple in D :

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

Next, we need to compute the expected information requirement for each attribute. Let’s start with the attribute *age*. We need to look at the distribution of *yes* and *no* tuples for each category of *age*. For the *age* category *youth*, there are two *yes* tuples and three *no* tuples. For the category *middle_aged*, there are four *yes* tuples and zero *no* tuples. For the category *senior*, there are three *yes* tuples and two *no* tuples. Using Equation (6.2),

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

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

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

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

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

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

Similarly, we can compute $Gain(income) = 0.029$ bits, $Gain(student) = 0.151$ bits, and $Gain(credit_rating) = 0.048$ bits. Because age has the highest information gain among the attributes, it is selected as the splitting attribute. Node N is labeled with age , and branches are grown for each of the attribute's values. The tuples are then partitioned accordingly, as shown in Figure 6.5. Notice that the tuples falling into the partition for $age = middle_aged$ all belong to the same class. Because they all belong to class "yes," a leaf should therefore be created at the end of this branch and labeled with "yes." The final decision tree returned by the algorithm is shown in Figure 6.2. ■

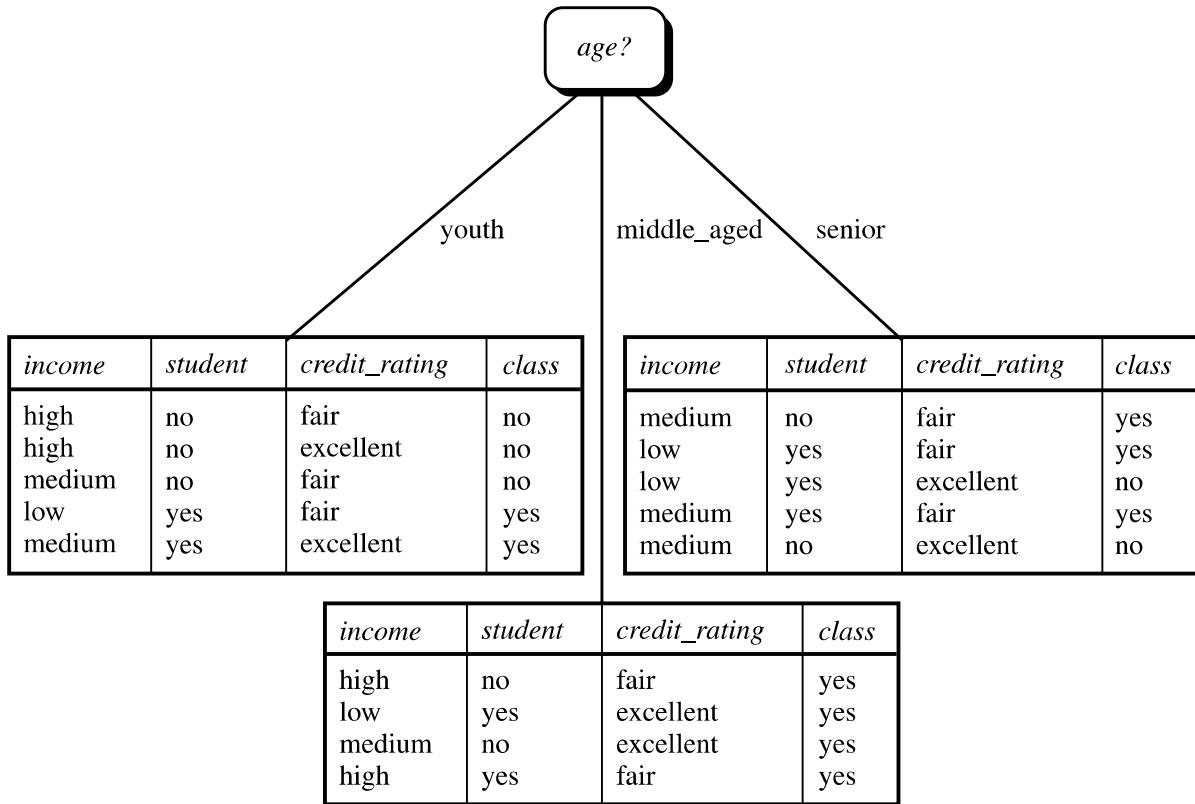


Figure 6.5 The attribute *age* has the highest information gain and therefore becomes the splitting attribute at the root node of the decision tree. Branches are grown for each outcome of *age*. The tuples are shown partitioned accordingly.

“But how can we compute the information gain of an attribute that is continuous-valued, unlike above?” Suppose, instead, that we have an attribute *A* that is continuous-valued, rather than discrete-valued. (For example, suppose that instead of the discretized version of *age* above, we instead have the raw values for this attribute.) For such a scenario, we must determine the “best” split-point for *A*, where the split-point is a threshold on *A*. We first sort the values of *A* in increasing order. Typically, the midpoint between each pair of adjacent values is considered as a possible split-point. Therefore, given *v* values of *A*, then *v* – 1 possible splits are evaluated. For example, the midpoint between the values *a_i* and *a_{i+1}* of *A* is

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

If the values of *A* are sorted in advance, then determining the best split for *A* requires only one pass through the values. For each possible split-point for *A*, we evaluate *Info_A(D)*, where the number of partitions is two, that is *v* = 2 (or *j* = 1, 2) in Equation (6.2). The point with the minimum expected information requirement for *A* is selected as the *split_point* for *A*. *D₁* is the set of tuples in *D* satisfying *A* ≤ *split_point*, and *D₂* is the set of tuples in *D* satisfying *A* > *split_point*.

Gain ratio

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

$$\text{SplitInfo}_A(D) = - \sum_{j=1}^v \frac{|D_j|}{|D|} \times \log_2 \left(\frac{|D_j|}{|D|} \right). \quad (6.5)$$

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

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

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

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

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

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

Gini index

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

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

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, $\{a_1, a_2, \dots, a_v\}$, 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

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

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 $A \leq \text{split_point}$, and D_2 is the set of tuples in D satisfying $A > \text{split_point}$.

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

$$\Delta Gini(A) = Gini(D) - Gini_A(D). \quad (6.9)$$

The attribute that maximizes the reduction in impurity (or, equivalently, has the minimum Gini index) is selected as the splitting attribute. This attribute and either its

splitting subset (for a discrete-valued splitting attribute) or split-point (for a continuous-valued splitting attribute) together form the splitting criterion.

Example 6.3 Induction of a decision tree using gini index. Let D be the training data of Table 6.1 where there are nine tuples belonging to the class $\text{buys_computer} = \text{yes}$ and the remaining five tuples belong to the class $\text{buys_computer} = \text{no}$. 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 :

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

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 $\{\text{low}, \text{medium}\}$. This would result in 10 tuples in partition D_1 satisfying the condition " $\text{income} \in \{\text{low}, \text{medium}\}$ ". The remaining four tuples of D would be assigned to partition D_2 . The Gini index value computed based on this partitioning is

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

Similarly, the Gini index values for splits on the remaining subsets are: 0.315 (for the subsets $\{\text{low}, \text{high}\}$ and $\{\text{medium}\}$) and 0.300 (for the subsets $\{\text{medium}, \text{high}\}$ and $\{\text{low}\}$). Therefore, the best binary split for attribute income is on $\{\text{medium}, \text{high}\}$ (or $\{\text{low}\}$) because it minimizes the gini index. Evaluating the attribute, we obtain $\{\text{youth}, \text{senior}\}$ (or $\{\text{middle_aged}\}$) as the best split for age with a Gini index of 0.375; the attributes $\{\text{student}\}$ and $\{\text{credit_rating}\}$ are both binary, with Gini index values of 0.367 and 0.429, respectively.

The attribute income and splitting subset $\{\text{medium}, \text{high}\}$ therefore give the minimum gini index overall, with a reduction in impurity of $0.459 - 0.300 = 0.159$. The binary split " $\text{income} \in \{\text{medium}, \text{high}\}$ " results in the maximum reduction in impurity of the tuples in D and is returned as the splitting criterion. Node N is labeled with the criterion, two branches are grown from it, and the tuples are partitioned accordingly. Hence, the Gini index has selected income instead of age at the root node, unlike the (nonbinary) tree created by information gain (Example 6.1). ■

This section on attribute selection measures was not intended to be exhaustive. We have shown three measures that are commonly used for building decision trees. These measures are not without their biases. Information gain, as we saw, is biased toward multivalued attributes. Although the gain ratio adjusts for this bias, it tends to prefer unbalanced splits in which one partition is much smaller than the others. The Gini index is