CHAPTER 6

Classification: basic concepts and methods

Classification is a form of data analysis that extracts models describing important data classes. Such models, called classifiers, predict categorical (discrete, unordered) class labels. For example, we can build a classification model to categorize bank loan applications as either safe or risky, or identify the early sign of cognitive impairment based on a patient's functional magnetic resonance imaging (fMRI) scan, or help a self-driving car automatically recognize various road signs. Such analysis can help provide us with a better understanding of the data at large. Many classification methods have been proposed by researchers in machine learning, pattern recognition, and statistics. Traditional classification algorithms typically assume a small or medium data size. Modern classification techniques have built on such work, developing scalable classification and prediction techniques capable of handling very large amounts of data. Classification belongs to supervised learning and is closely connected to many other data mining tasks. Classification has numerous applications, including fraud detection, target marketing, performance prediction, manufacturing, medical diagnosis, and many more.

We start off by introducing the main ideas of classification in Section 6.1. In the rest of this chapter, you will learn the basic techniques for data classification such as how to build decision tree classifiers (Section 6.2), Bayes classifiers (Section 6.3), lazy learners (Section 6.4), and linear classifiers (Section 6.5). Section 6.6 discusses how to evaluate and compare different classifiers. Various measures of accuracy are given, as well as techniques for obtaining reliable accuracy estimates. Methods for improving classifier accuracy are presented in Section 6.7, including ensemble methods and class-imbalanced data (i.e., where the main class of interest is rare).

6.1 Basic concepts

We introduce the concept of classification in Section 6.1.1. Section 6.1.2 describes the general approach to classification as a two-step process. In the first step, we build a classification model based on previous data. In the second step, we determine if the model's accuracy is acceptable, and if so, we use the model to classify new data.

6.1.1 What is classification?

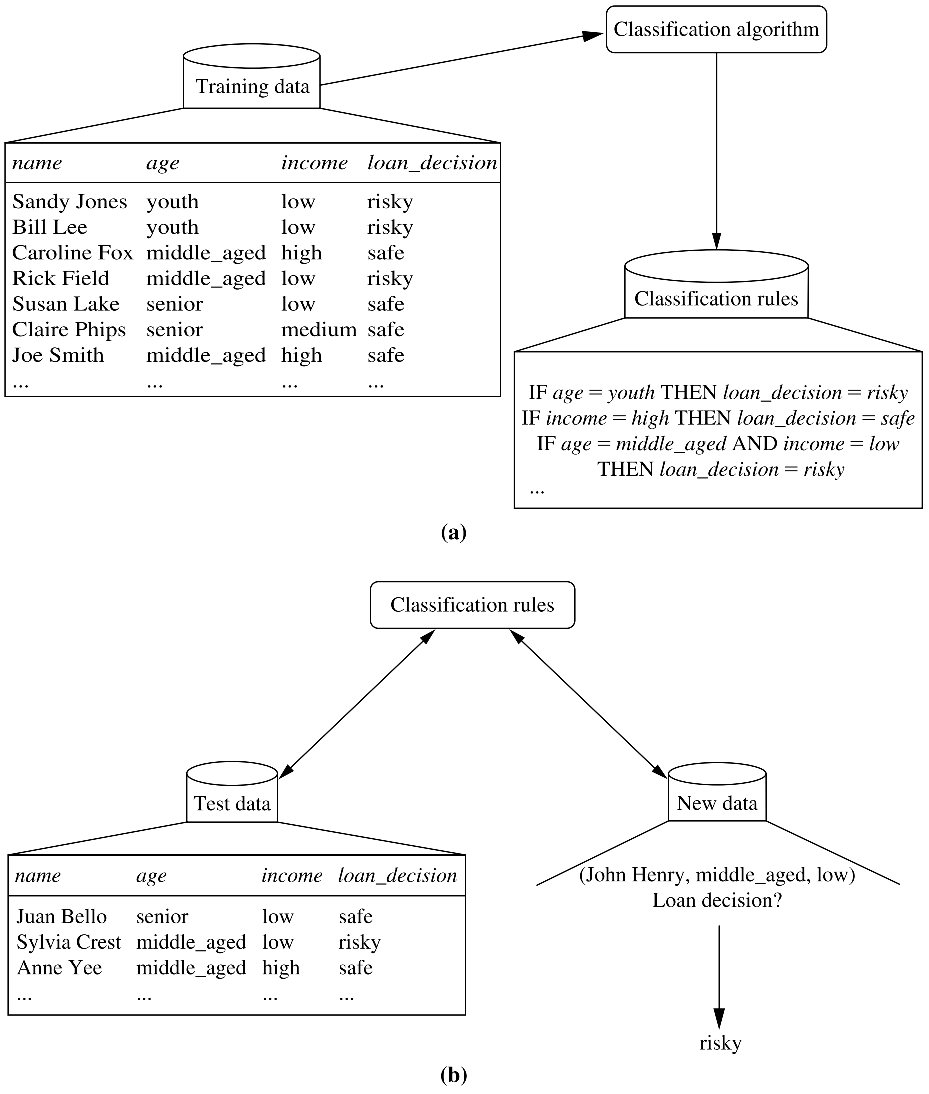
A bank loans officer needs analysis of her data to learn which loan applicants are "safe" and which are "risky" for the bank, and her colleague from the risk management department wishes to detect fraudulent transactions. A marketing manager at an electronics store needs data analysis to help guess whether a customer with a given profile will buy a new computer, or understand the *sentiment* of social media posts regarding a newly released product, or detect *fake reviews* about a new product from an online review site, or identify a subscribed customer who is likely to switch to a competitive electronics store (i.e., churn prediction). An IT security analyst wants to know if the network system is under attack (intrusion detection) or if a given application is contaminated with malware (malware detection). A teacher wishes to know if a student enrolled in an online course will drop out before she completes the course. A talent recruiter wants to know if an individual is looking for the next career move. A medical researcher wants to analyze breast cancer data to predict which one of three specific treatments a patient should receive, a cardiologist wants to identify the patient who is likely to have a congestive heart failure based on her chronic medical history, a neuroscientist wants to identify the early sign of cognitive impairment (which could lead to, say Alzheimer's disease) based on a patient's functional magnetic resonance imaging (fMRI) scan. An intelligent question-answering system needs to understand what type of question the user is asking (question classification), as the first step to automatically provide a high-quality answer. A self-driving car needs to automatically recognize various road signs (e.g., 'stop,' 'detour,' etc.). A physicist needs to identify *high energy event* from massive experiment data, which might lead to new discoveries. Law enforcement wishes to predict the crime hot spot so that the precaution measures can be taken proactively.

In each of these examples, the data analysis task is **classification**, where a model or **classifier** is constructed to predict *class (categorical) labels*, such as "safe" or "risky" for the loan application data; or "positive" or "negative" for sentiment classification; or "yes" or "no" for the marketing data; or "dropout" or "stay" for online course enrollment, or "treatment A," "treatment B," or "treatment C" for the medical data; or various question types for a question-answering system. 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 wants to predict how much a given customer will spend during a sale; or a realtor might be interested in knowing the average house pricing of the next year in different residential areas; or a career planner wants to forecast the average yearly income of students immediately after graduating from the college in different majors. This kind of 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 class label. **Regression analysis** is a statistical methodology that is most often used for numeric prediction; hence the two terms tend to be used synonymously, although other methods for numeric prediction exist. **Ranking** is another type of numerical prediction where the model predicts the ordered values (i.e., ranks), for example, a web search engine (e.g., Google) ranks the relevant webpages with respect to a given query, with the higher-ranked webpages being more relevant to the query. Classification and numeric prediction are the two major types of **prediction problems**. This chapter primarily focuses on classification. It is worth pointing out that classification and numerical prediction (e.g., regression) are closely related to each other. Many classification techniques can be modified for the purpose of regression. We will see some examples, including regression trees (Section 6.2), lazy learners (Section 6.4.1), linear regression (Section 6.5), and gradient tree boosting (Section 6.7.1).

6.1.2 General approach to classification

"How does classification work?" **Data classification** is a two-step process, consisting of a *learning step* (where a classification model is constructed) and a *classification step* (where the model is used



name	*age*	*income*	*loan_decision*
Sandy Jones	youth	low	risky
Bill Lee	youth	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
...	...	...	...

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*
...

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 acceptable, the rules can be applied to the classification of new data tuples.

to predict class labels for given data). The process is shown for the loan application data in Fig. 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** (also known as the 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* (or nominal) 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 randomly sampled 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 belongs to **supervised learning** (i.e., the learning of the classifier is "supervised" in that it is told to which class each training tuple belongs). The scope of supervised learning is larger than classification, and it broadly encompasses learning methods for training a numerical prediction model (e.g., regression, ranking) if the true target values of training tuples are known during the learning step. Supervised learning contrasts with **unsupervised learning** (e.g., **clustering**), in which the true target value (e.g., 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. Likewise, we could use clustering techniques to find social media posts sharing similar topics without knowing their actual class labels. Clustering is the topic of Chapters 8 and 9. The landscape of the prediction problem (e.g., classification, regression, ranking) has gone beyond supervised vs. unsupervised learning. To name a few, in **semisupervised classification**, it builds a classifier based on a limited number of labeled training tuples (whose true class labels are given during training) and a large number of unlabeled training tuples (whose class labels are unknown during training); in **zero-shot learning**, some class label might appear *after* the classification model has been built. In other words, during the training phase, there are no (i.e., zero) labeled training tuples for such a class label. Both semisupervised learning and zero-shot learning belong to **weakly supervised learning** in that the supervision information for training the model is weaker than the standard supervised learning. For the classification task, this means that the supervision (i.e., the true class labels of training tuples) is known only for a small fraction of the entire training set in semisupervised learning; or is absent for certain class label(s) in zero-shot learning. *Classification with weak supervision* will be introduced in Chapter 7.

The 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 map-

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

[2] In the machine learning literature, training tuples are commonly referred to as *training samples*. Throughout this text, we prefer to use the term *tuples* instead of *samples*.

ping is represented as classification rules that identify loan applications as being either safe or risky (Fig. 6.1(a)). The rules can be used to categorize future data tuples, as well as provide deeper insight into the data contents. They also provide a compressed data representation.

"What about classification accuracy?" In the second step (Fig. 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 classifier's accuracy, this estimate would likely be too optimistic, because the classifier tends to **overfit** the data (i.e., during learning it may incorporate some particular anomalies of the training data that do not represent the general data set). Therefore a **test set** is used, made up of **test tuples** and their associated class labels. They are independent of the training tuples, meaning that they were not used to construct the classifier.

The **accuracy** of a classifier on a given test set is the percentage of test 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 6.6 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 Fig. 6.1(a) from the analysis of data from previous loan applications can be used to approve or reject new or future loan applicants.

6.2 Decision tree induction

Decision tree induction is the learning of decision trees from class-labeled training tuples. A **decision tree** is a flowchart-like tree structure, where each **internal node** (nonleaf node) denotes a test on an attribute, each **branch** represents an outcome of the test, and each **leaf node** (or *terminal node*) holds a class label. The topmost node in a tree is the **root** node. A typical decision tree is shown in Fig. 6.2. It represents the concept *buys_computer*; that is, it predicts whether a customer at an electronics store is

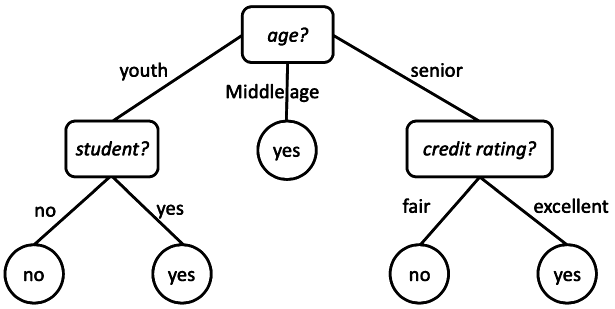


FIGURE 6.2

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

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

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

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

In Section 6.2.1, we describe a basic algorithm for learning decision trees. During tree construction, *attribute selection measures* are used to select the attribute that best partitions the tuples into distinct classes. Popular measures of attribute selection are given in Section 6.2.2. When decision trees are built, many of the branches may reflect noise or outliers in the training data. *Tree pruning* attempts to identify and remove such branches, with the goal of improving classification accuracy on unseen data. Tree pruning is described in Section 6.2.3.

6.2.1 Decision tree induction

During the late 1970s and early 1980s, J. Ross Quinlan, a researcher in machine learning, developed a decision tree algorithm known as **ID3** (Iterative Dichotomizer). 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 independent of one another at around the same time, yet follow a similar approach for learning decision trees from training tuples. These two cornerstone algorithms spawned a flurry of work on decision tree induction.

ID3, C4.5, and CART adopt a greedy (i.e., nonbacktracking) approach in which decision trees are constructed in a top-down recursive divide-and-conquer manner. Most algorithms for decision tree induction also follow a top-down approach, which starts with a training set of tuples and their associated class labels. The training set is recursively partitioned into smaller subsets as the tree is being built. A basic decision tree algorithm is summarized in Fig. 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*. D is 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

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

Input:

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

Output: A decision tree.
Method:

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

FIGURE 6.3

Basic algorithm for inducing a decision tree from training tuples.

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

- The tree starts as a single node, N, representing the training tuples in D (step 1).[3]
- If the tuples in D are all of the same class, then node N becomes a leaf and is labeled with that class (steps 2 and 3). Note that steps 4 and 5 are terminating conditions. All terminating conditions are explained at the end of the algorithm.
- Otherwise, the algorithm calls *Attribute_selection_method* to determine the **splitting criterion**. The splitting criterion tells us which attribute to test at node N by determining the "best" way to separate

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

or partition the tuples in D into individual classes (step 6). The splitting criterion also tells us which branches to grow from node N with respect to the outcomes of the chosen test. More specifically, the splitting criterion indicates the **splitting attribute** and may also indicate either a **split-point** or a **splitting subset**. The splitting criterion is determined so that, ideally, the resulting partitions at each branch are as "pure" as possible. A partition is **pure** if all the tuples in it belong to the same class. In other words, if we split up the tuples in D according to the mutually exclusive outcomes of the splitting criterion, we hope for the resulting partitions to be as pure as possible.

- The node N is labeled with the splitting criterion, which serves as a test at the node (step 7). A branch is grown from node N for each of the outcomes of the splitting criterion. The tuples in D are partitioned accordingly (steps 10–11). There are three possible scenarios, as illustrated in Fig. 6.4. Let A be the splitting attribute. A has v distinct values, $\{a_1, a_2, \ldots, a_v\}$, based on the training data.

1. *A is discrete-valued:* In this case, the outcomes of the test at node N directly correspond to the known values of A. A branch is created for each known value, a_j, of A and labeled with that value (Fig. 6.4(a)). Partition D_j is the subset of class-labeled tuples in D having value a_j of A. Because all the tuples in a given partition have the same value for A, A does not need to be

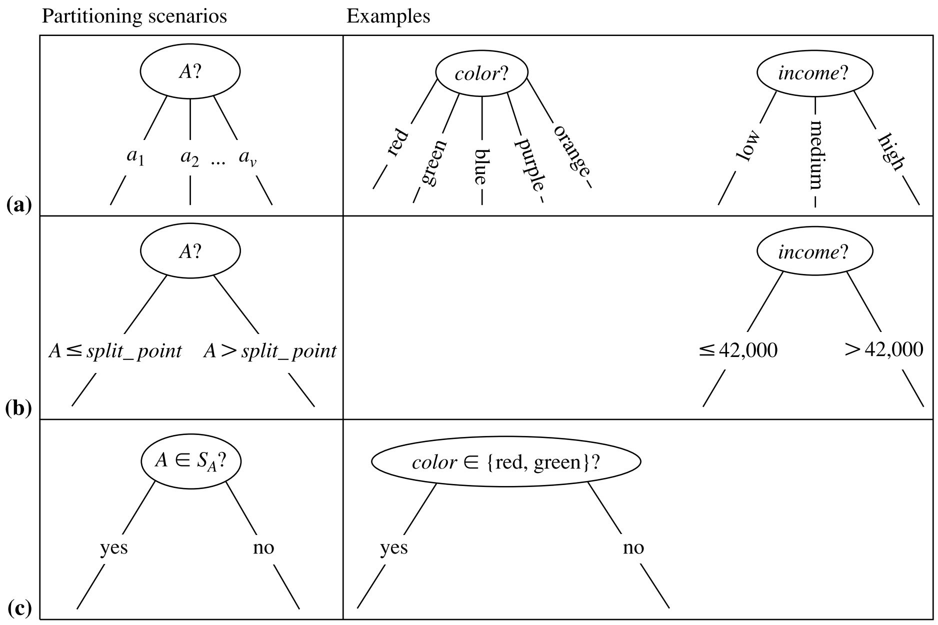


FIGURE 6.4

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

considered in any future partitioning of the tuples. Therefore it is removed from *attribute_list* (steps 8 and 9).

2. *A is continuous-valued:* In this case, the test at node N has two possible outcomes, corresponding to the conditions $A \leq split\_point$ and $A > split\_point$, respectively, where *split_point* is the split-point returned by *Attribute_selection_method* as part of the splitting criterion. (In practice, the split-point, a, is often taken as the midpoint of two known adjacent values of A and therefore may not actually be a preexisting value of A from the training data.) Two branches are grown from N and labeled according to the previous outcomes (Fig. 6.4(b)). The tuples are partitioned such that D_1 holds the subset of class-labeled tuples in D for which $A \leq split\_point$, while D_2 holds the rest.
3. *A is discrete-valued* and a *binary tree* must be produced (as dictated by the attribute selection measure or algorithm being used): The test at node N is of the form "$A \in S_A$?," where S_A is the splitting subset for A, returned by *Attribute_selection_method* as part of the splitting criterion. It is a subset of the known values of A. If a given tuple has value a_j of A, and if $a_j \in S_A$, then the test at node N is satisfied. Two branches are grown from N (Fig. 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 the tuples in partition D (represented at node N) belong to the same class (steps 2 and 3).
2. There are no remaining attributes on which the tuples may be further partitioned (step 4). In this case, **majority voting** is employed (step 5). This involves converting node N into a leaf and labeling it with the most common class in D. Alternatively, the class distribution of the node tuples may be stored.
3. There are no tuples for a given branch, that is, a partition D_j is empty (step 12). In this case, a leaf is created with the majority class in D (step 13).

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

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

Incremental versions of decision tree induction have also been proposed. When given new training data, it restructures 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 6.2.2) and the mechanisms used for pruning (Section 6.2.3).

Decision tree is closely related to another type of tree, called **regression tree**, which is used to predict the continuous output value. A regression tree is very similar to a decision tree in that it also partitions the entire attribute space into multiple subregions, each corresponding to a leaf node. The main difference is as follows. In a regression tree, a leaf node holds a *continuous value* instead of a

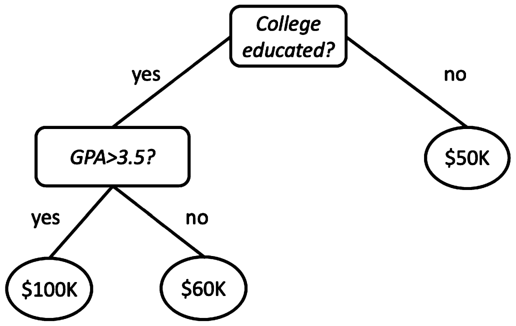


FIGURE 6.5

A regression tree for predicting the average yearly income based on an individual's education. The values of the three leaf nodes are calculated as follows. $50K is the average yearly income of all training individuals who do not have a college degree; $60K is the average yearly income of all training individuals who have a college degree with a GPA less than or equal to 3.5; and $100K is the average yearly income of all training individuals who have a college degree with a GPA higher than 3.5. The leaf node values ($50K, $60K, and $100K) are used to predict the yearly income of any test individual who falls into the corresponding leaf nodes.

categorical value (i.e., class label) in a decision tree. The continuous value of a leaf node is learned during the training phase, which is set as the average output value of all training tuples fallen in the corresponding subregions. CART uses **residual sum of squares** (RSS) as the objective function, which is the sum of the squared difference between the actual and predicted output values of training tuples

$$\text{RSS} = \sum_i (y_i - \hat{y}_i)^2, \tag{6.1}$$

where y_i is the actual output value of the ith training tuple, and $\hat{y}_i$ is the predicted output by the regression tree. Choosing the average output of all training tuples in the corresponding subregion is optimal in that it minimizes the RSS in Eq. (6.1). Each leaf node value is then used to predict the output of a test tuple which falls into it. Fig. 6.5 presents an example of a regression tree for predicting the average yearly income based on an individual's education (e.g., whether or not the individual attended the college, the average GPA at college, etc.).

6.2.2 Attribute selection measures

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

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

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

Information gain

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

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

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2(p_i), \tag{6.2}$$

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

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

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

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

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Info(D_j). \tag{6.3}$$

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

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,

$$Gain(A) = Info(D) - Info_A(D). \tag{6.4}$$

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 customer database of an electronics store. (The data are adapted from Quinlan [Qui86]. In this example, each attribute is discrete-valued. Continuous-valued attributes have been generalized.) The class label attribute, *buys_computer*, has two distinct

Table 6.1 Class-labeled training tuples from the customer database of an electronics store.

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

values (namely, {*yes, no*}); therefore, there are two distinct classes (i.e., $m = 2$). Let class C_1 correspond to *yes* and class C_2 correspond to *no*. There are nine tuples of class *yes* and five tuples of class *no*. A (root) node N is created for the tuples in D. To find the splitting criterion for these tuples, we must compute the information gain of each attribute. We first use Eq. (6.2) to compute the expected information needed to classify a tuple in D:

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

Next, we need to compute the expected information requirement for each attribute. Let's start with the attribute *age*. We need to look at the distribution of *yes* and *no* tuples for each category of *age*. For the *age* category "youth" there are two *yes* tuples and three *no* tuples. For the category "middle_aged," there are four *yes* tuples and zero *no* tuples. For the category "senior," there are three *yes* tuples and two *no* tuples. Using Eq. (6.3), the expected information needed to classify a tuple in D if the tuples are partitioned according to *age* is

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

Hence, the gain in information from such partitioning would be

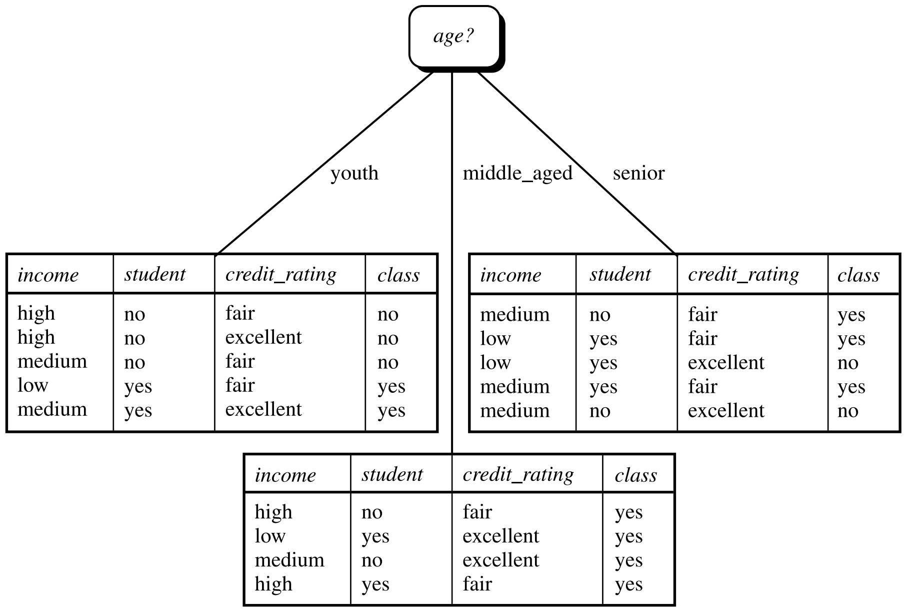
$$Gain(age) = Info(D) - Info_{age}(D) = 0.940 - 0.694 = 0.246 \text{ bits.}$$

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 Fig. 6.6. Notice that the tuples falling into the partition for *age* = *middle_aged* all belong to the same class. Because they all belong to class *"yes,"* a leaf should therefore be created at the end of this branch and labeled *"yes."* The final decision tree returned by the algorithm was shown earlier in Fig. 6.2. □

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

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

$$\frac{a_i + a_{i+1}}{2}. \tag{6.5}$$



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 6.6

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.

If the values of A are sorted in advance, then determining the best split for A requires only one pass through the values. For each possible split-point for A, we evaluate $Info_A(D)$, where the number of partitions is two, that is, $v = 2$ (or $j = 1, 2$) in Eq. (6.3). The point with the minimum expected information requirement for A is selected as the *split_point* for A. D_1 is the set of tuples in D satisfying $A \leq split\_point$, and D_2 is the set of tuples in D satisfying $A > split\_point$.

Gain ratio

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

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

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2\left(\frac{|D_j|}{|D|}\right). \qquad (6.6)$$

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

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo_A(D)}. \tag{6.7}$$

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 Eq. (6.6) to obtain

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

From Example 6.1, we have $Gain(income) = 0.029$. Therefore $GainRatio(income) = 0.029/1.557 = 0.019$. □

Gini impurity

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

$$Gini(D) = 1 - \sum_{i=1}^{m} p_i^2, \tag{6.8}$$

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

$$Gini_A(D) = \frac{|D_1|}{|D|}Gini(D_1) + \frac{|D_2|}{|D|}Gini(D_2). \tag{6.9}$$

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

For continuous-valued attributes, each possible split-point must be considered. The strategy is similar to that described earlier for information gain, where the midpoint between each pair of (sorted) adjacent values is taken as a possible split-point. The point giving the minimum Gini impurity 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

$$\Delta Gini(A) = Gini(D) - Gini_A(D). \tag{6.10}$$

The attribute that maximizes the reduction in impurity (or, equivalently, has the minimum Gini impurity) 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 the Gini impurity. Let D be the training data shown earlier in Table 6.1, where there are nine tuples belonging to the class $buys\_computer = yes$ and the remaining five tuples belong to the class $buys\_computer = no$. A (root) node N is created for the tuples in D. We first use Eq. (6.8) for the Gini impurity 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 impurity for each attribute. Let's start with the attribute *income* and consider each of the possible splitting subsets. Consider the subset $\{low, medium\}$. This would result in 10 tuples in partition D_1 satisfying the condition "$income \in \{low, medium\}$." The remaining four tuples of D would be assigned to partition D_2. The Gini impurity value computed based on this partitioning is

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

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

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

"So, what is the relationship between Gini impurity and information gain?" Intuitively, both measures aim to quantify to what extent the impurity will be reduced if we split the current node based on the given attribute. Information gain, rooted in information theory, measures the impurity based on (the change of) the average amount of information needed to identify the class label of a tuple. Gini impurity is related to *mis-classification* in the following way. Based on the class label distribution in the current node, it tells how likely a randomly chosen tuple will be mis-classified if it is assigned to a random class label. Gini impurity is always used for binary split, whereas information gain allows multiway split. In terms of computation, Gini impurity is slightly more efficient than information gain, since the latter involves the logarithm computation. In practice, however, both measures often lead to very similar decision trees.

Other attribute selection measures

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

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

For regression tree, it is natural to use RSS (Eq. (6.1)) as the splitting criteria. That is, the best split point for a given attribute is the one that leads the smallest RSS. We choose the attribute with the minimum RSS to split the tree node into two nodes, including left leaf node and right leaf node.

Example 6.4. Let us look at an example in Table 6.2 on how to use RSS to find the best split point. Suppose there are five training tuples at a regression tree node, and each training tuple has a true output value y_i and a continuous attribute $x_i (i = 1, ..., 5)$. We want to find the best split point for attribute x_i to split the tree node into two leaf nodes. More specifically, all the tuples whose x_i is less than or equal to the split point will go to the left leaf node, and the remaining training tuples will go to the right leaf node.

Table 6.2 Training data for regression.

attribute x_i	1	2	3	4	5
output y_i	10	12	8	20	22

Given five training tuples at a regression tree node, each with a true output value y_i and a continuous attribute $x_i (i = 1, ..., 5)$. We want to find the best split point for attribute x_i to split the tree node into two nodes (left node and right node).

Table 6.3 Using RSS to choose the best split point for data tuples in Table 6.2.

candidate split point x_i	1.5	2.5	3.5	4.5
predicted value of left leaf node y_l	10	11	10	12.5
predicted value of right leaf node y_r	15.5	16.7	21	22
RSS	131	116.67	10	83

Since x_i is a continuous attribute with five possible values, there are four candidate split points, including $x_i = 1.5$, $x_i = 2.5$, $x_i = 3.5$ and $x_i = 4.5$. For each candidate split point, we partition the current tree node into two leaf nodes. The average output value y_l of the training tuples in the left leaf node is used to predict the output of all tuples residing in the left leaf node. Likewise, the average output value y_r of the training tuples in the right leaf node is used to predict the output of all tuples residing in the right leaf node. For example, if the split point $x_i = 1.5$, only the first training tuple goes to the left leaf node, and we have that $y_l = y_1 = 10$; and $y_r = (y_2 + y_3 + y_4 + y_5)/4 = (12 + 8 + 20 + 22)/4 = 15.5$. Using the predicted output values for all five training tuples (y_l or y_r), we can use Eq. (6.1) to calculate RSS. Again, if the split point $x_i = 1.5$, we have that RSS $= \sum_{i=1}^{5}(y_i - \hat{y}_i)^2 = (y_1 - y_l)^2 + (y_2 - y_r)^2 + (y_3 - y_r)^2 + (y_4 - y_r)^2 + (y_5 - y_r)^2 = 122.25$. The computation results for all four possible split points are summarized in Table 6.3. Since $x_i = 3.5$ has the smallest RSS, it is chosen as the split point. □

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 solution is preferred. The philosophy underlying the MLD principle is **Occam's razor**, also known as *law of parsimony*. In data mining and machine learning, Occam's razor is often translated into a design principle that one should favor a model with a shorter description (hence minimum description length) for the data over a lengthier model, provided that everything else is equal (e.g., both shorter and lengthier models share the same training set errors).

Other attribute selection measures consider **multivariate splits** (i.e., where the partitioning of tuples is based on a *combination* of attributes, rather than on a single attribute). The CART system, for example, can find multivariate splits based on a linear combination of attributes. Multivariate splits are a form of **attribute** (or feature) **construction**, where new attributes are created based on the existing ones. (Attribute construction was also discussed in Chapter 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 (Section 6.10).

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

6.2.3 Tree pruning

When a decision tree is built, many of the branches will reflect anomalies in the training data due to noise or outliers. Tree pruning methods address this problem of *overfitting* the data. Such methods typically use statistical measures to remove the least-reliable branches. An unpruned tree and a pruned version of it are shown in Fig. 6.7. 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 label among the subset tuples or the probability distribution of the class labels of those tuples.

When constructing a tree, measures such as statistical significance, information gain, Gini impurity, 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.

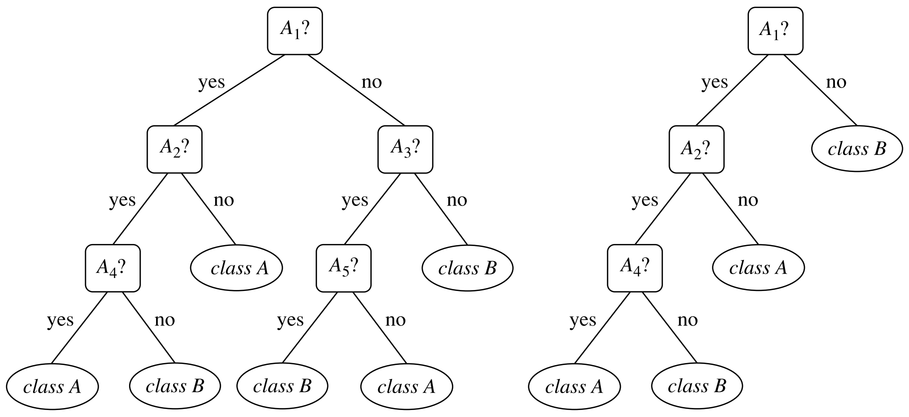


FIGURE 6.7

An unpruned decision tree (left) and a pruned version of it (right).

There are difficulties, however, in choosing an appropriate threshold. High thresholds could result in oversimplified trees, whereas low thresholds could result in very little simplification.

The second and more common approach is **postpruning**, which removes subtrees from a "fully grown" tree. A subtree at a given node is pruned by removing its branches and replacing it with a leaf. The leaf is labeled with the most frequent class label among the subtree being replaced. For example, notice the subtree at node "A_3?" in the unpruned tree of Fig. 6.7. 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.

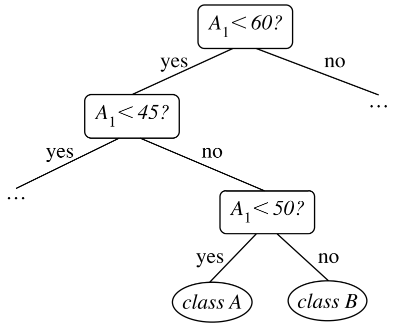
A **pruning set** of class-labeled tuples is used to estimate the cost complexity. This set is independent (1) of the training set used to build the unpruned tree and (2) of any test set used for accuracy estimation. The algorithm generates a set of progressively pruned trees. In general, the smallest 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 pruning set. Instead, it uses the training set to estimate error rates. Recall that an estimate of accuracy or error based on the training set is overly optimistic and therefore strongly biased. The pessimistic pruning method, therefore, adjusts the error rates obtained from the training set by adding a penalty, so as to counter the bias incurred.

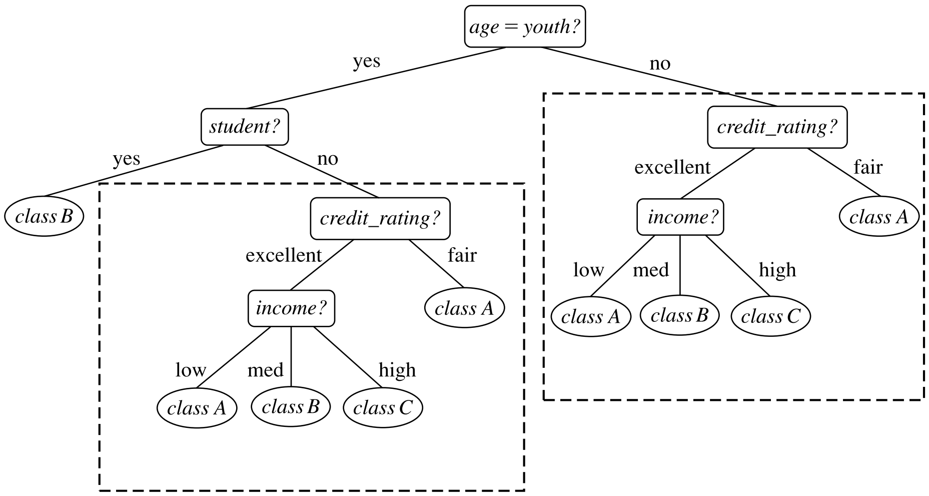
Rather than pruning trees based on estimated error rates, we can prune trees based on the number of bits required to encode them. The "best" pruned tree is the one that minimizes the number of encoding bits. This method adopts the MDL principle, which was briefly introduced in Section 6.2.2. The basic idea is that the simplest solution is preferred. Unlike cost complexity pruning, it does not require an independent set of tuples (i.e., the pruning set).

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

Although pruned trees tend to be more compact than their unpruned counterparts, they may still be rather large and complex. Decision trees can suffer from *repetition* and *replication* (Fig. 6.8), making them overwhelming to interpret. **Repetition** occurs when an attribute is repeatedly tested along a given branch of the tree (e.g., *"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 Chapter 7, which shows how a *rule-based classifier* can be constructed by extracting IF-THEN rules from a decision tree.



(a)



(b)

FIGURE 6.8

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

6.3 Bayes classification methods

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

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

6.4 Lazy learners (or learning from your neighbors)

The classification methods discussed so far in this book—decision tree induction and Bayesian classification—are both 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 construction 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 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 numeric 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 numeric prediction, lazy learners can be computationally expensive. They require efficient storage techniques and are well suited to implementation on parallel hardware. They offer little explanation or insight into the data's structure. Lazy learners, however, naturally support incremental learning. They are able to model complex decision spaces having hyperpolygonal shapes that may not be as easily describable by other learning algorithms (such as hyperrectangular shapes modeled by decision trees). In this section, we look at two examples of lazy learners: *k-nearest-neighbor classifiers* (Section 6.4.1) and *case-based reasoning classifiers* (Section 6.4.2).

6.4.1 *k*-nearest-neighbor classifiers

The k-nearest-neighbor method was first described in the early 1950s. The method is labor-intensive when given a large training set, 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.

Suppose you want to make a decision on whether or not you should buy a computer. What would you do? One possible way to make such a decision is to find out your friends' decision on this (whether or not to buy a computer). If most of your close friends buy a computer, maybe you will decide to buy a computer as well. Nearest-neighbor classifiers follow a very similar idea of 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 the training tuples are stored in an n-dimensional attribute space. When given an unknown tuple, a ***k*-nearest-neighbor classifier** searches the attribute space for the k training tuples that are closest to the unknown tuple (i.e., to find your close friends in the above example). These k training tuples are the k "nearest neighbors" of the unknown tuple. Then *k-nearest-neighbor classifier* chooses the most common class label among the k nearest neighbors as the predicted class label of the unknown tuple (i.e., to follow the majority decision of your friends in the above example).

"Closeness" is defined in terms of a distance metric, such as Euclidean distance. The Euclidean distance between two points or tuples, say, $\boldsymbol{X}_1 = (x_{11}, x_{12}, \dots, x_{1n})$ and $\boldsymbol{X}_2 = (x_{21}, x_{22}, \dots, x_{2n})$, is

$$dist(\boldsymbol{X}_1, \boldsymbol{X}_2) = \sqrt{\sum_{i=1}^{n} (x_{1i} - x_{2i})^2}. \tag{6.17}$$

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. The square root is taken of the total accumulated distance count. Typically, we normalize the values of each attribute before using Eq. (6.17). This helps prevent attributes with initially large ranges (e.g., *income*) from outweighing attributes with initially smaller ranges (e.g., 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

$$v' = \frac{v - min_A}{max_A - min_A}, \tag{6.18}$$

where min_A and max_A are the minimum and maximum values of attribute A. Chapter 2 describes 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 label 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 the attribute space. When $k > 1$, we can take a (weighted) majority voting on the class labels among its k-nearest neighbors. Nearest-neighbor classifiers can also be used for numeric prediction, that is, to return a real-valued prediction for a given unknown tuple. In this case, the classifier returns the (weighted) 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 are not numeric, but nominal (or categorical) such as color?*" The previous discussion assumes that the attributes used to describe the tuples are all numeric. For nominal attributes, a 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 $\boldsymbol{X}_1$ or in tuple $\boldsymbol{X}_2$, we assume the maximum possible difference. Suppose that each of the attributes has been mapped to the range [0, 1]. For nominal 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 will 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 value that gives the minimum error rate may be selected. In general, the larger the number of training tuples, the larger the value of k will be (so that classification and numeric 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 than twice the Bayes error rate (the latter being the theoretical minimum). In

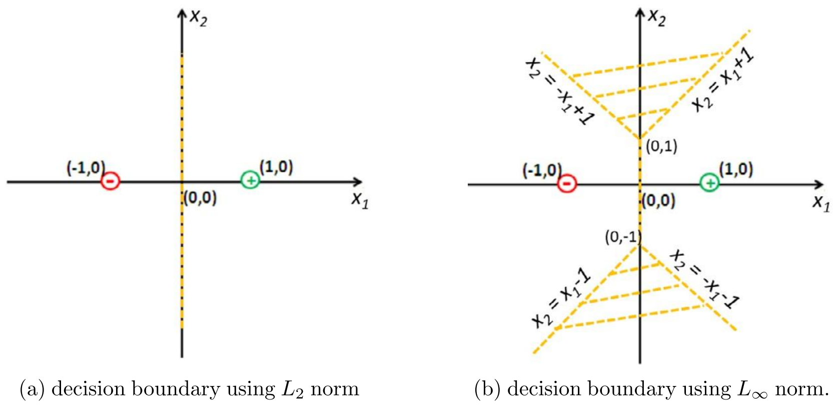


FIGURE 6.9

The impact of distance metrics on 1-nearest-neighbor classifier. Given two training examples, including a positive example at $(1, 0)$ and a negative example at $(-1, 0)$. The decision boundaries of 1-nearest-neighbor classifier using different distance metrics are quite different from each other. Using L_2 norm (on the left), the decision boundary is a vertical line at $x_2 = 0$. Using L_∞ norm (on the right), the decision boundary includes a line segment between $(0, -1)$ and $(0, 1)$ and two shaded areas.

other words, 1-nearest-neighbor classifier is asymptotically near-optimal. If k 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 2.3), or other distance measurements, may also be used. Fig. 6.9 presents an illustrative example in terms of the impact of distance metrics on the decision boundary of k-nearest-neighbor classifier.

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 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 are proven useless. This method is also referred to as **pruning** or **condensing** because it reduces the total number of tuples stored. Another technique to speed up nearest-neighbor search is via **locality-sensitive-hashing** (LSH). The key idea is to hash the similar tuples into the same bucket with a high probability via *locality-preserving hash functions*. Then, given a test tuple, we first identify which bucket it belongs to, and then we only search the training tuples in the same bucket to identify its nearest neighbors.

6.4.2 Case-based reasoning

Case-based reasoning (CBR) classifiers use a database of problem solutions to solve new problems. Unlike k-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 in the common law system, 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 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 to propose a feasible combined solution.

Key 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 system's efficiency 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 those 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.

6.5 Linear classifiers

So far, we have learned a few classifiers which are capable of generating complex decision boundaries. For example, a decision tree classifier might output a hyperrectangular-shaped decision boundary (Fig. 6.10(a)), and a k-nearest-neighbor classifier might output a hyperpolygonal-shaped decision boundary (Fig. 6.10(b)). However, what about a simple, linear decision boundary? For the example in Fig. 6.10, intuitively, a linear decision boundary (the straight line in Fig. 6.10(c)) is (almost) as good as decision tree classifiers and k-nearest-neighbor classifier in separating the positive training tuples from the negative training ones. Yet, such a linear decision boundary might offer additional advantages, such as efficient computation for training the classifier, better generalization performance, and better interpretability.

In this section, we introduce basic techniques to learn such linear classifiers. We will start with **linear regression**, which forms the basis for linear classifiers. Then, we will introduce two linear classi-

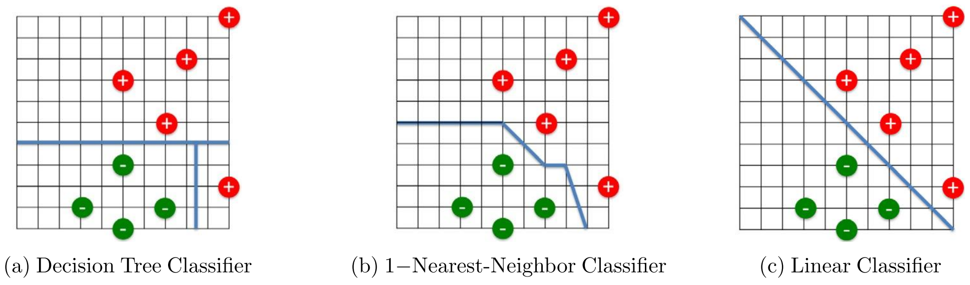

(a) Decision Tree Classifier
(b) 1−Nearest-Neighbor Classifier
(c) Linear Classifier

FIGURE 6.10

Decision boundaries by different classifiers. Note that this example is *linearly separable*, meaning that a linear classifier (c) can perfectly separate all the positive training tuples from all the negative training tuples. If the training set is *linearly inseparable*, we could still use a linear classifier, at the expense that some training tuples are on the 'wrong' side of the decision boundary. In Chapter 7, we will introduce techniques (e.g., support vector machines) to handle linearly inseparable case.

fiers, including (1) **perception**, which is one of the earliest linear classifiers, and (2) **logistic regression** which is one of the most widely used linear classifiers. Additional linear classifiers will be introduced in Chapter 7, such as linear support vector machines.

6.5.1 Linear regression

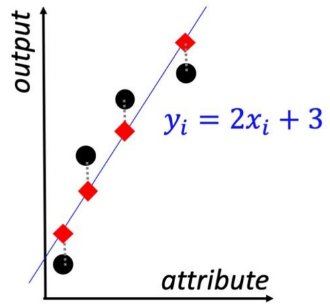
Linear regression is a statistical technique that predicts a continuous value based on one or more independent attributes. For example, we might want to predict the housing price based on the living area or to predict the future income of a student based on which college she attended, in which major and the overall GPA, etc. Since linear regression aims to predict a *continuous* value, it cannot be directly applied to the classification task, where the output is a categorical variable. Nonetheless, the core techniques in linear regression form the basis of linear classifiers. Therefore, let us first briefly introduce linear regression.

Suppose we have n tuples, each of which is represented by p attributes $x_i = (x_{i,1}, ..., x_{i,p})^T$ and a continuous output value y_i $(i = 1, ..., n)$. In linear regression, we want to learn a linear function that maps the p input attributes x_i to the output variable y_i, that is, $\hat{y}_i = w^T x_i + b = \sum_{j=1}^{p} w_j x_{i,j} + b$, where $\hat{y}_i$ is the predicted output value for the ith tuple, $w = (w_1, ..., w_p)^T$ is a p-dimensional weight vector and b is the bias scalar. In other words, linear regression assumes that the output value is a linear weighted summation of the p input attribute values, offset by the bias scalar b. The entries in the weight vector w_j $(j = 1, ..., p)$ tell how important the corresponding attribute $x_{i,j}$ is in predicting the output variable $\hat{y}_i$. In the aforementioned examples, a linear regression model would assume that the housing prices are linearly correlated with the living area; the future income of a student can be predicted by a linear weighted combination of the college she attended, the major, and the overall GPA (plus a bias scalar b). If we know the weight vector w and the bias scalar b, we can make a prediction of the output value based on its p input attribute values.

"So, how can we determine the weight vector w and the bias scalar b?" Intuitively, we want to learn the "best" weight vector w and the "best" bias scalar b from the training data, so that the lin-

Index (i)	1	2	3	4
Attribute (x_i)	1	3	5	7
Output (y_i)	4	10	14	16

(a) Training tuples



(b) Least square regression

FIGURE 6.11

An example of least square regression. (a) Four training tuples. (b) Scatter-plot of the training tuples (black dots) and least square regression model (the blue line). Red diamonds are the predicted output $\hat{y}_i$ ($i = 1, 2, 3, 4$) and dashed lines indicate the prediction errors ($|y_i - \hat{y}_i|$) of the corresponding training tuples.

ear regression model can make the "best" prediction. That is, the predicted value $\hat{y}_i = w^T x_i + b$ is as close as possible to the actual observed value y_i ($i = 1, ..., n$). One of the most common linear regression methods is called **least square regression**, which aims to minimize the following loss function $L(w, b) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - (w^T x_i + b))^2$. Therefore the best weight vector w and the bias scalar b are the ones that minimize the loss function $L(w, b)$, which measures the sum of the squared difference between the predicted output value $\hat{y}_i$ and the actual observed value y_i. For example, if there is only one input attribute (i.e., $p = 1$), the optimal weight $w = \frac{\sum_{i=1}^{n} x_i (y_i - \bar{y})}{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}(\sum_{i=1}^{n} x_i)^2}$ and the optimal bias scalar $b = \frac{1}{n} \sum_{i=1}^{n} (y_i - w x_i)$, where $\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i$ is the average observed output value among all n training tuples.

Example 6.7. Let us look at an example of least square regression in Fig. 6.11. There are four training tuples, each represented by a single-dimensional attribute x_i and an output variable y_i ($i = 1, 2, 3, 4$). We want to find least square regression model $y = wx + b$ that predicts the output y based on the input attribute x. We use the two equations mentioned above to find the optimal weight w and the optimal bias scalar b. We first find the optimal weight w as follows. The average output of four training tuples is $\bar{y} = (y_1 + y_2 + y_3 + y_4)/4 = (4 + 10 + 14 + 16)/4 = 11$. Therefore we have that $\sum_{i=1}^{4} x_i (y_i - \bar{y}) = 1(4 - 11) + 3(10 - 11) + 5(14 - 11) + 7(16 - 11) = 40$. In the meanwhile, we have that $\sum_{i=1}^{4} x_i^2 = 1^2 + 3^2 + 5^2 + 7^2 = 84$ and $1/4(\sum_{i=1}^{4} x_i)^2 = (1 + 3 + 5 + 7)^2/4 = 64$. Therefore the optimal weight $w = \frac{\sum_{i=1}^{n} x_i (y_i - \bar{y})}{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}(\sum_{i=1}^{n} x_i)^2} = \frac{40}{84-64} = 2$. Based on the optimal weight w, the optimal bias scalar $b = \frac{\sum_{i=1}^{4}(y_i - w x_i)}{4} = \frac{(4-2\times1)+(10-2\times3)+(14-2\times5)+(16-2\times7)}{4} = 3$. □

"But, what if there are multiple p ($p > 1$) attributes?" In this case (which is called **multilinear regression**), let us first change our notation a little bit. We assume there is an additional "dummy" attribute which always takes the value of 1 for any tuple. Let the weight for this dummy attribute be w_0. Then the overall weight vector $w = (w_0, w_1, ..., w_p)$ and the new input attribute vector $x_i = (1, x_{i,1}, ..., x_{i,p})$ are both $(p+1)$-dimensional vectors. The multilinear regression model can be re-written as $\hat{y}_i = w^T x_i = w_0 + w_1 x_{i,1} + ... + w_p x_{i,p}$. We use the same loss function as before, that is, $L(w) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} (y_i - (w^T x_i))^2$. It turns out the optimal weight vector w can be computed as $w = (XX^T)^{-1} Xy$, where $X = [x_1, x_2, ..., x_n]$ is a $(p+1) \times n$ matrix, and $y = [y_1, ..., y_n]^T$ is an $n \times 1$ vector. (How to derive the closed form solutions for single linear regression as well as multilinear regression are left as exercises.)

In least square regression, we measure the "goodness" of the learned regression model by the sum of the squared difference between predicted and actual output values. The squared loss might be sensitive to the outliers in the training set. In *robust regression*, it uses alternative loss functions that are less sensitive to such outliers. For example, the Huber method in robust regression uses the following loss: $L(w) = \sum_{i=1}^{n} l_H(y_i - \hat{y}_i)$, where $l_H(y_i - \hat{y}_i) = (y_i - \hat{y}_i)^2$ if $|y_i - \hat{y}_i| < \theta$, $l_H(y_i - \hat{y}_i) = 2\theta|y_i - \hat{y}_i| - \theta^2$ otherwise, and $\theta > 0$ is a user-specified parameter. Notice that the optimal weight vector w for multilinear regression involves a matrix inverse (i.e., $(XX^T)^{-1}$). In case $p > n$ (i.e., the number of attributes is more than the number of training tuples), such a matrix inverse does not exist. An effective way to address this issue is to introduce a *regularization term* regarding the norm of the weight vector w. For example, if we use l_2 norm of the weight vector w, the corresponding regression model is called Ridge regression; if we use l_1 norm of the weight vector w instead, the corresponding regression model is called Lasso regression which often learns a *sparse* weight vector. This means that some entries of the learned weight vector w are zeros, which indicates that those attributes are not used in the regression model. In Section 7.1, we will use Lasso regression for feature selection.

6.5.2 Perceptron: turning linear regression to classification

"How can we modify a linear regression model to perform classification task?" Suppose we have a binary classification task.[10] The output value y_i for a given tuple is a binary variable: $y_1 = +1$ indicates the ith tuple is a positive tuple (e.g., *buy computer*) and $y_i = 0$ indicates the ith tuple is a negative one (e.g., *not buy computer*). One way to modify the linear regression model for such a binary classification task is to use the *sign* of the output of the linear regression model as the predicted class label, that is, $\hat{y}_i = \text{sign}(w^T x_i)$, where $\hat{y}_i$ is the predicted class label for ith tuple, $\text{sign}(z) = 1$ if $z > 0$ and $\text{sign}(z) = 0$ otherwise. Notice that we use the same notation as multilinear regression where we have introduced a "dummy" attribute which always takes the value of 1 for any tuple. Therefore if we know the weight vector w, we can use it to predict the class label of a given tuple as follows. We compute a linear combination of the attribute values of the given tuple, weighted by the corresponding entries of the weight vector w. If the resulting value of such a linear combination is positive, we predict that the given tuple is a positive tuple. Otherwise, we predict that it is a negative one.

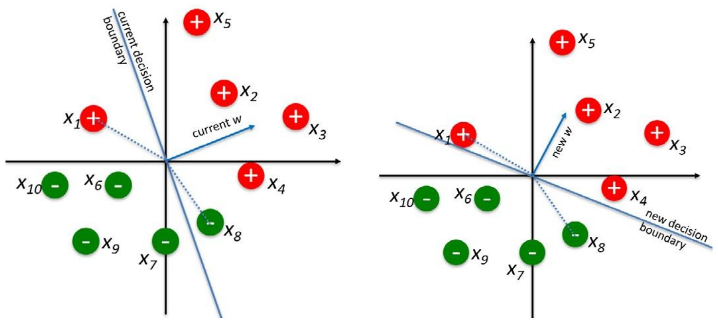
"How can we find the optimal weight vector w from a set of training tuples?" The classic learning algorithm to train a perceptron is as follows. We start with an initial guess of the weight vector w (e.g.,

[10] For both perceptron and logistic regression classifiers that we will introduce next, we focus on binary classification task. However, the techniques we introduce can be generalized to handle multiclass classification task for both classifiers.

we can simply set $w = 0$). Then, the learning algorithm will iterate until it converges, or the maximum iteration number or some other preset stopping criteria are met. In each iteration, we do the following for each training tuple x_i. We try to predict the class label of x_i using the current weight vector w, that is, $\hat{y}_i = \text{sign}(w^T x_i)$. If the prediction is correct (i.e., $\hat{y}_i = y_i$), we do nothing about the weight vector. However, if the prediction is incorrect (i.e., $\hat{y}_i \neq y_i$), we update the current weight vector in one of the following two ways. If $y_i = +1$ (i.e., the ith tuple is a positive tuple, but the current classifier predicts it is a negative tuple), we update weight vector as $w \leftarrow w + \eta x_i$. If $y_i = 0$ (i.e., the ith tuple is a negative tuple, which is wrongly predicted by the current classifier as a positive tuple), we update weight vector as $w \leftarrow w - \eta x_i$, where $\eta > 0$ is the user-specified learning rate. So, the intuition is that in each iteration of the training process, the algorithm will focus on those wrongly predicted training tuples by the current weight vector w. If the wrongly predicted training tuple x_i is a positive tuple, we update the weight vector w by moving it *towards* the attribute vector x_i of this training tuple (i.e., $w \leftarrow w + \eta x_i$). On the other hand, if the wrongly predicted training tuple x_i is a negative tuple, we update the weight vector w by moving it *away from* the attribute vector x_i of this training tuple (i.e., $w \leftarrow w - \eta x_i$).

Example 6.8. Let us look at an example in Fig. 6.12 for training a perceptron classifier. In Fig. 6.12, we assume the bias $w_0 = 0$ for illustration clarity. Fig. 6.12(a) (left) shows the current decision boundary and the weight vector w, where two training tuples are wrongly classified, including a positive tuple x_1 and a negative tuple x_8. Therefore only these two tuples are used to update the weight vector in the current iteration, that is, $w \leftarrow w + \eta x_1 - \eta x_8$. The updated weight vector w and the corresponding decision boundary are shown in Fig. 6.12(b) (right), where all training tuples are correctly classified. □

"How effective is the perceptron learning algorithm?" If the training tuples are linearly separable (e.g., the example in Fig. 6.12), the perceptron algorithm is guaranteed to find a weight vector (i.e., a hyperplane decision boundary) that perfectly separates all the positive training tuples from all the negative training tuples. However, if the training tuples are not linearly separable, this algorithm will fail to converge.



(a) current weight vector w (b) updated weight vector w

FIGURE 6.12

Training a perceptron classifier.

Perceptron, one of the earliest linear classifiers, was first invented back in 1958. It can also be used as a building block (called a "neuron") in deep neural networks that will be introduced in Chapter 10.

6.5.3 Logistic regression

Perceptron that we have just introduced in the previous section is capable of predicting the binary class label of a given tuple. However, can we also tell how confident such a prediction is? Again, let us consider a binary classification task, and we assume that there are two possible class labels, that is, $y = 1$ for a positive tuple and $y = 0$ for a negative tuple. Recall that in (naïve) Bayes classifier, we can estimate the posterior probability $P(y_i = 1|x_i)$, which can be directly used to indicate how confident the predicted classification result is. For example, if $P(y_i = 1|x_i)$ is close to 1, the classifier is highly confident that the tuple x_i is a positive example.

How can we make a linear classifier not only predict which class label a tuple has, but also tell how confident it is in making such a prediction? An effective way to this end is via **logistic regression** classifier. Let us first introduce an important function called **sigmoid** function, which is defined as $\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z}$. From Fig. 6.13, we can see that the sigmoid function maps a real number in $(-\infty, +\infty)$ (i.e., the x-axis of Fig. 6.13) to an output value in the range of $(0, 1)$ (i.e., the y-axis of Fig. 6.13). Therefore if we leverage the sigmoid function to map the output of a linear regression model to a number between 0 and 1, we can interpret the mapping result as the posterior probability of observing a positive class label. This is exactly what logistic regression classifier tries to do!

Formally, we have $P(\hat{y}_i = 1|x_i, w) = \sigma(w^T x_i) = \frac{1}{1+e^{-w^T x_i}}$, where $\hat{y}_i$ is the predicted class label for the tuple with attributes x_i, and w is the weight vector. Notice that we have absorbed the bias term b into the weight vector w by introducing a dummy attribute to simplify the notation, as we did in the multilinear regression model and in perceptron. Naturally, if $P(\hat{y}_i = 1|x_i, w) > 0.5$, the classifier predicts that the tuple x_i is a positive tuple (i.e., $\hat{y}_i = 1$), otherwise, it predicts a negative tuple (i.e., $\hat{y}_i = 0$). This (details are left as an exercise) is equivalent to the following linear classifier: predict $\hat{y}_i = 1$ (i.e., positive tuple) if $w^T x_i > 0$, and predict $\hat{y}_i = 0$ (i.e., negative tuple) if $w^T x_i < 0$. Therefore if we know the weight vector w, the classification task for a given tuple is quite simple. That is, we

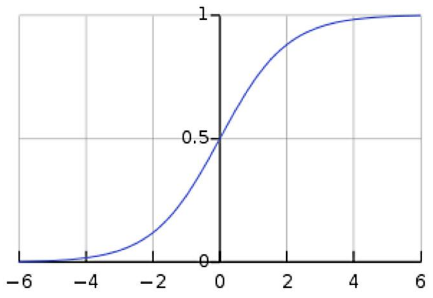


FIGURE 6.13

Illustration of sigmoid function. The sigmoid function "squashes" an input from a larger range $(-\infty, +\infty)$ to a smaller range $(0, 1)$. For this reason, sigmoid function is also called *squash function*. In Chapter 10, we will see other types of squash functions, which are called activation functions in the deep learning terminology.

only need to multiply the attribute vector x_i of the given tuple with the weight vector w, and then make a prediction based on the sign of $w^T x_i$. If $w^T x_i$ is a positive number, we predict that the given tuple is a positive tuple. Otherwise, we predict that it is a negative tuple.

"How can we determine the optimal weight vector w from a set of training tuples?" The classic method to train a logistic regression classifier (i.e., to determine the best weight vector w from the training set) is via *maximum likelihood estimation* (MLE). Again, let us assume there are n training tuples (x_i, y_i) $(i = 1, ..., n)$. Since we have a binary classification task, we can view the predicted class label $\hat{y}_i$ as a Bernoulli random variable, which can only take two possible values, including $P(\hat{y}_i = 1|x_i, w) = p_i$ and $P(\hat{y}_i = 0|x_i, w) = 1 - p_i$, where $p_i = \sigma(w^T x_i) = \frac{1}{1+e^{-w^T x_i}}$ is determined by the sigmoid function and it describes the probability of observing a positive outcome for the predicted class label (i.e., $\hat{y}_i = 1$). Notice that the true class label y_i for the ith tuple is a binary variable. Therefore we have that $P(\hat{y}_i = y_i) = p_i^{y_i}(1 - p_i)^{1-y_i}$. The maximum likelihood estimation method aims to solve the following optimization problem, which says that we should choose the best weight vector w that maximizes the likelihood of the training set. The intuition is that we want to find the optimal model parameter (i.e., the weight vector w) so that there is the highest "chance" (i.e., the likelihood or the probability) of observing the entire training set.

$$w^* = \text{argmax}_w \; L(w) = \Pi_{i=1}^n p_i^{y_i}(1 - p_i)^{1-y_i} = \Pi_{i=1}^n (\frac{e^{w^T x_i}}{1 + e^{w^T x_i}})^{y_i} (\frac{1}{1 + e^{w^T x_i}})^{1-y_i} \tag{6.19}$$

"But, how can we develop an algorithm to solve this optimization problem to find the optimal weight vector w?" First, we notice that the likelihood function $L(w)$ has many nonnegative terms that are multiplied with each other. In practice, it is often more convenient to work with the logarithm of such a complicated function. Thus we have the following equivalent optimization problem, where $l(w)$ is called the log likelihood

$$w^* = \text{argmax}_w \; l(w) = \sum_{i=1}^{n} y_i x_i^T w - \log(1 + e^{w^T x_i}). \tag{6.20}$$

From the optimization perspective, the good news is that the log likelihood function in Eq. (6.20) is a strictly concave function, and therefore its maximum (the optimal solution) uniquely exists. However, the bad news is that the closed-form solution for the above optimization problem does not exist. In this case, a common strategy is to find the optimal solution w^* iteratively as follows. In each iteration, we try to improve the current weight vector w so that the objective function we wish to maximize (the log likelihood function $l(w)$) is improved most. In order to increase the current objective function $l(w)$ most, it turns out the best direction to update the current estimation of the weight vector w is to follow its *gradient*. This leads to the following algorithm to learn the optimal weight vector w^* from the training set. We start with an initial guess of the weight vector w (e.g., we can simply set $w = 0$). Then, the learning algorithm will iterate until it converges, or the maximum iteration number or some other preset stopping criteria are met. In each iteration, it updates the weight vector w as follows $w \leftarrow w + \eta \sum_{i=1}^n (y_i - P(\hat{y}_i = 1|x_i, w))x_i$, where $\eta > 0$ is the user-specified learning rate.

"So, what is the intuition of the above algorithm?" Let us analyze the impact of each training tuple (x_i, y_i) on updating the estimation of the weight vector w. We consider two situations depending on whether it is a positive tuple (i.e., $y_i = 1$) or a negative tuple (i.e., $y_i = 0$). For the former, the

impact of the given tuple on updating the weight vector w can be calculated as $w \leftarrow w + \eta(1 - P(\hat{y}_i = 1|x_i, w))x_i$. The intuition is that we want update the current weight vector w *towards* the direction of the attribute vector x_i of this positive tuple. For the latter case (i.e., $y_i = 0$), the impact of the given tuple on updating the weight vector w can be calculated as $w \leftarrow w - \eta P(\hat{y}_i = 1|x_i, w)x_i$. The intuition is that we want update the current weight vector w *away from* the direction of the attribute vector x_i of this negative tuple. From this perspective, the learning algorithm for training a logistic regression classifier bears some similarities to the perceptron algorithm. That is, both algorithms try to update the current weight vector w so that is (1) more aligned with the attribute vectors of positive tuples and (2) more mis-aligned with (i.e., towards the opposite direction of) the attribute vectors of negative tuples.

However, the two algorithms (perceptron vs. logistic regression) differ regarding to what extent the algorithms update the weight vector w. In perceptron, it uses a *fixed* learning rate η for all wrongly predicted tuples by the current weight vector w. On the other hand, in logistic regression, it depends on the learning rate η as well as $P(\hat{y}_i = 1|x_i, w)$ (i.e., the probability that the given tuple belongs to the positive class based on the current weight vector w). This makes the logistic regression algorithm *adaptive* in the following sense. For example, if $P(\hat{y}_i = 1|x_i, w)$ is high for a positive tuple, it means that the prediction by the current weight vector w for this positive tuple is not only correct (i.e., $P(\hat{y}_i = 1|x_i, w) > 0.5$), but also quite confident (i.e., $P(\hat{y}_i = 1|x_i, w)$ is close to 1). Then, the impact of this positive tuple (i.e., $\eta(1 - P(\hat{y}_i = 1|x_i, w)))$ on updating the weight vector is relatively small. On the other hand, if $P(\hat{y}_i = 1|x_i, w)$ is high for a negative tuple, it means that the prediction by the current weight vector w for this negative tuple is either wrong (i.e., $P(\hat{y}_i = 1|x_i, w) > 0.5$), or correct but with low confidence (i.e., $P(\hat{y}_i = 1|x_i, w)$ is barely below 0.5). Then, the impact of this negative tuple (i.e., $\eta P(\hat{y}_i = 1|x_i, w))$ on updating the weight vector will be relatively large. In other words, the logistic regression learning algorithm pays more attention to those "hard" training tuples, which are either wrongly predicted or correctly predicted with a low confidence by the current weight vector w. Recall that for the example in Fig. 6.12(a), perceptron only uses x_1 and x_8 to update the current weight vector w since these two tuples are wrongly classified by the current w. In contrast, logistic regression uses *all* training tuples to update the weight vector w. Among them, x_1 and x_8 have the highest impact on updating w since they are both wrongly classified by the current classifier; x_2, x_3, x_5, x_9 and x_{10} have the least impact since they are all correctly classified by the current weight vector w with a high confidence; x_4, x_6 and x_7 have the moderate impact since they are correctly classified but with a relatively low confidence.

"How good is the logistic regression algorithm? What are the potential limitations and how to mitigate?" Since the log likelihood function $l(w)$ is a concave function, the algorithm for training a logistic regression classifier described above is guaranteed to converge to its optimal solution. However, if the training set is linearly separable, the algorithm might converge to a weight vector w with an infinitely large norm. (See an illustrative example in Fig. 6.14.) A "large" weight vector w could make the trained classifier prone to the noise of certain attributes of a given tuple. This will, in turn, lead to a poor generalization performance of the learned logistic regression classifier. In other words, the learned logistic regression classifier *overfits* the training set. An effective way to mitigate the overfitting is to introduce a regularization term $\|w\|_2^2$ into the objective function $l(w)$ to prevent the learned weight vector from

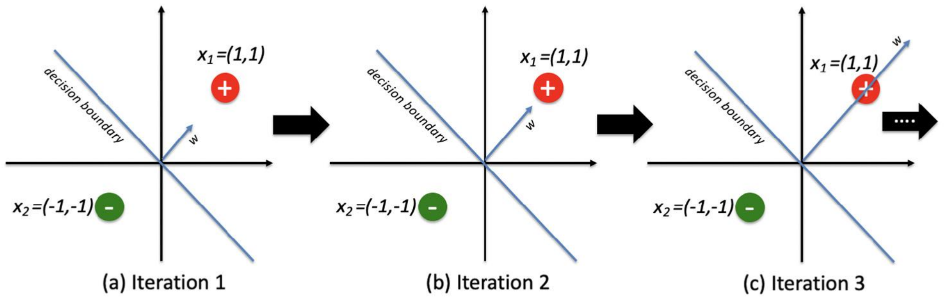


FIGURE 6.14

Illustration of the infinitely large weight vector of logistic regression in linearly separable case. There are two training tuples in 2d space, with one positive training tuple $x_1 = (1, 1)$ and one negative training tuple $x_2 = (-1, -1)$. For simplicity, we let the bias scalar $b = 0$ and the learning rate $\eta = 1$. Suppose that at iteration 1, the weight vector $w = (1, 1)$. Then, logistic regression algorithm introduced above will update the weight vector as $w_{\text{new}} = w_{\text{old}} + (1 - P(\hat{y}_1 = 1|x_1, w_{\text{old}}))x_1 - P(\hat{y}_2 = 1|x_2, w_{\text{old}})x_2 = (0.5, 0.5) + a(1, 1) = (1 + 2a)w_{\text{old}}$, where $a = 1 - P(\hat{y}_1 = 1|x_1, w_{\text{old}}) + P(\hat{y}_2 = 1|x_2, w_{\text{old}}) > 0$. As such, the new weight vector w_{new} shares the same direction as the old w_{old}. Therefore, the decision boundary remains the same, but new weight vector w_{new} grows in the magnitude by a factor of $(1 + 2a)$. This trend will continue as the logistic regression algorithm progresses, leading to a weight vector with an infinitely large magnitude.

becoming "too large."[11] The second potential limitation is the *independence* assumption behind logistic regression. Recall that when we calculate the likelihood $L(w)$ of the training set, we simply multiply the likelihood of each training tuple together (Eq. (6.19)). This means that we have implicitly assumed that different training tuples are independent of each other. However, this assumption might be violated in some applications (e.g., users on a social network are interconnected with each other). The graph-based classification might provide a natural remedy for this issue. The third potential limitation lies in the computational challenge. Notice that in the updating rule $w \leftarrow w + \eta \sum_{i=1}^{n} (y_i - P(\hat{y}_i = 1|x_i, w))x_i$ described above, we need to calculate the gradients $(y_i - P(\hat{y}_i = 1|x_i, w))$ for *all* training tuples and then sum them up to update the weight vector w. If there are millions of training tuples, it is computationally very expensive to perform such computation. An efficient way to address this issue is to use *stochastic gradient descent* method to train a logistic regression classifier. That is, at each iteration, we will randomly sample a small subset of training tuples (this is often referred to as a *minibatch*) and *only* use the sampled tuples (instead of all training tuples) to update the weight vector. It is worth pointing out that the stochastic gradient descent is extensively used in many other data mining algorithms, such as deep learning methods, which will be introduced in Chapter 10.

[11] From the statistical parameter estimation perspective, we are switching from the maximum likelihood estimation (MLE) to maximum a posterior estimation (MAP). Adding a regularization term $\|w\|_2^2$ into $l(w)$ is equivalent to imposing a Gaussian prior with the mean vector at the origin for the weight vector w.

6.6 Model evaluation and selection

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

Section 6.6.1 describes various evaluation metrics for the predictive accuracy of a classifier. Based on randomly sampled partitions of the given data, holdout and random subsampling (Section 6.6.2), cross-validation (Section 6.6.3), and bootstrap methods (Section 6.6.4) are common techniques for assessing accuracy. What if we have more than one classifier and want to choose the "best" one? This is referred to as **model selection** (i.e., choosing one classifier over another). The last two sections address this issue. Section 6.6.5 discusses how to use tests of statistical significance to assess whether the difference in accuracy between two classifiers is due to chance. Section 6.6.6 presents how to compare classifiers based on cost–benefit and receiver operating characteristic (ROC) curves.

6.6.1 Metrics for evaluating classifier performance

This section presents measures for assessing how good or how "accurate" your classifier is at predicting the class label of tuples. We will consider the case where the class tuples are more or less evenly distributed, as well as the case where classes are unbalanced (e.g., where an important class of interest is rare such as in medical tests). The classifier evaluation measures presented in this section are summarized in Fig. 6.15. They include accuracy (also known as recognition rate), sensitivity (or recall), specificity, precision, F_1, and F_β. Note that although accuracy is a specific measure, the word "accuracy" is also used as a general term to refer to a classifier's predictive abilities.

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

Before we discuss the various measures, we need to become comfortable with some terminology. Recall that we can talk in terms of **positive tuples** (tuples of the main class of interest) and **negative tuples** (all other tuples).[12] Given two classes, for example, the positive tuples may be *buys_computer* $=$ *yes* while the negative tuples are *buys_computer* $=$ *no*. Suppose we use our classifier on a test set of labeled tuples. P is the number of positive tuples, and N is the number of negative tuples. For each tuple, we compare the classifier's class label prediction with the tuple's known class label.

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

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

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

FIGURE 6.15

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

		Predicted class		
		yes	*no*	Total
Actual class	*yes*	TP	FN	P
	no	FP	TN	N
	Total	P'	N'	$P+N$

FIGURE 6.16

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

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

These terms are summarized in the **confusion matrix** of Fig. 6.16.

A confusion matrix is a useful tool for analyzing how well your classifier can recognize tuples of different classes. TP and TN tell us when the classifier is getting things right, whereas FP and FN tell us when the classifier is getting things wrong (i.e., mislabeling). Given m classes (where $m \geq 2$), a **confusion matrix** is a table of at least size m by m. An entry, $CM_{i,j}$ at the ith row and the jth column indicates the number of tuples of class i that were labeled by the classifier as class j. For a classifier to have good accuracy, ideally most of the tuples would be represented along the diagonal of the confusion matrix, from entry $CM_{1,1}$ to entry $CM_{m,m}$, with the rest of the entries being zero or close to zero. That is, ideally, FP and FN are around zero.

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

FIGURE 6.17

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

The table may have additional rows or columns to provide totals. For example, in the confusion matrix of Fig. 6.16, P and N are shown. In addition, P' is the number of tuples that were labeled as positive ($TP + FP$), and N' is the number of tuples that were labeled as negative ($TN + FN$). The total number of tuples is $TP + TN + FP + TN$, or $P + N$, or $P' + N'$. Note that although the confusion matrix shown is for a binary classification problem, confusion matrices can be easily drawn for multiple classes in a similar manner.

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

$$accuracy = \frac{TP + TN}{P + N}. \tag{6.21}$$

In the pattern recognition literature, this is also referred to as the overall **recognition rate** of the classifier; that is, it reflects how well the classifier recognizes tuples of the various classes. An example of a confusion matrix for the two classes *buys_computer = yes* (positive) and *buys_computer = no* (negative) is given in Fig. 6.17. Totals are shown, as well as the recognition rates per class and overall. By glancing at a confusion matrix, it is easy to see if the corresponding classifier is confusing two classes.

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

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

$$error\ rate = \frac{FP + FN}{P + N}. \tag{6.22}$$

If we were to use the training set (instead of a test set) to estimate the error rate of a model, this quantity is known as the **resubstitution error**.[13] This error estimate is optimistic of the true error rate (and similarly, the corresponding accuracy estimate is optimistic) because the model is not tested on any samples that it has not already seen.

We now consider the **class imbalance problem**, where the main class of interest is rare. That is, the data set distribution reflects a significant majority of the negative class and a minority positive class. For example, in fraud detection applications, the class of interest (or positive class) is *"fraud"* which occurs much less frequently than the negative *"nonfraudulant"* class. In medical data, there may

[13] In machine learning literature, it is often referred to as the training error.

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

FIGURE 6.18

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

be a rare class, such as *"cancer."* Suppose that you have trained a classifier to classify medical data tuples, where the class label attribute is *"cancer"* and the possible class values are *"yes"* and *"no."* An accuracy rate of, say, 97% may make the classifier seem quite accurate, but what if only, say, 3% of the training tuples are actually cancer? Clearly, an accuracy rate of 97% may not be acceptable—the classifier could be correctly labeling only the noncancer tuples, for instance, and misclassifying all the cancer tuples. Instead, we need other measures, which assess how well the classifier can recognize the positive tuples (*cancer* = *yes*) and how well it can recognize the negative tuples (*cancer* = *no*).

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

$$sensitivity = \frac{TP}{P} \tag{6.23}$$

$$specificity = \frac{TN}{N}. \tag{6.24}$$

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

$$accuracy = sensitivity\frac{P}{(P+N)} + specificity\frac{N}{(P+N)}. \tag{6.25}$$

Example 6.9. Sensitivity and specificity. Fig. 6.18 shows a confusion matrix for medical data where the class values are *yes* and *no* for a class label attribute, *cancer*. The sensitivity of the classifier is $\frac{90}{300} = 30.00\%$. The specificity is $\frac{9560}{9700} = 98.56\%$. The classifier's overall accuracy is $\frac{9650}{10{,}000} = 96.50\%$. Thus we note that although the classifier has a high accuracy, it's ability to correctly label the positive (rare) class is poor given its low sensitivity. It has high specificity, meaning that it can accurately recognize negative tuples. Techniques for handling class-imbalanced data are given in Section 6.7.5. □

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

$$precision = \frac{TP}{TP+FP} \tag{6.26}$$

$$recall = \frac{TP}{TP + FN} = \frac{TP}{P}. \tag{6.27}$$

Example 6.10. Precision and recall. The precision of the classifier in Fig. 6.18 for the *yes* class is $\frac{90}{230} = 39.13\%$. The recall is $\frac{90}{300} = 30.00\%$, which is the same calculation for sensitivity in Example 6.9.
□

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

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

$$F = \frac{2 \times precision \times recall}{precision + recall} \tag{6.28}$$

$$F_\beta = \frac{(1+\beta^2) \times precision \times recall}{\beta^2 \times precision + recall}, \tag{6.29}$$

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

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

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

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

- **Speed:** This refers to the computational cost involved in generating and using the given classifier.

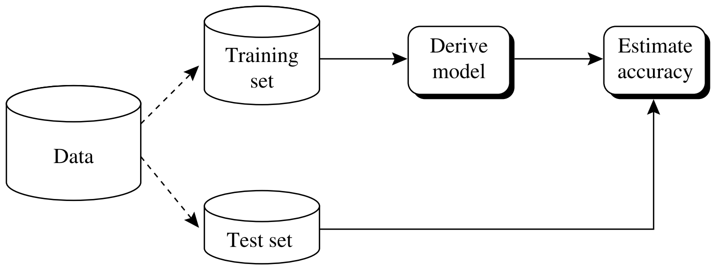


FIGURE 6.19

Estimating accuracy with the holdout method.

- **Robustness:** This is the ability of the classifier to make correct predictions given noisy data or data with missing values. Robustness is typically assessed with a series of synthetic data sets representing increasing degrees of noise and missing values.
- **Scalability:** This refers to the ability to construct the classifier efficiently given large amounts of data. Scalability is typically assessed with a series of data sets of increasing size.
- **Interpretability:** This refers to the level of understanding and insight that is provided by the classifier or predictor. Interpretability could be subjective and therefore more difficult to assess. Decision trees and classification rules can be easy to interpret, yet their interpretability may diminish the more they become complex. We will introduce some basic techniques to improve the interpretability of classification models in Chapter 7.

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

6.6.2 Holdout method and random subsampling

The **holdout** method is what we have alluded to so far in our discussions about accuracy. In this method, the given data are randomly partitioned into two independent sets, a *training set* and a *test set*. Typically, two-thirds of the data are allocated to the training set, and the remaining one-third is allocated to the test set. The training set is used to derive the model. The model's accuracy is then estimated with the test set (Fig. 6.19). The estimate is pessimistic because only a portion of the initial data is used to derive the model.

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

6.6.3 Cross-validation

In ***k*-fold cross-validation**, the initial data are randomly partitioned into k mutually exclusive subsets or "folds" $D_1, D_2, \ldots, D_k$, each of approximately equal size. Training and testing are performed k

times. In iteration i, partition D_i is reserved as the test set, and the remaining partitions are collectively used to train the model. That is, in the first iteration, subsets $D_2, \ldots, D_k$ collectively serve as the training set to obtain the first model, which is tested on D_1; the second iteration is trained on subsets $D_1, D_3, \ldots, D_k$ and tested on D_2; and so on. Unlike the holdout and random subsampling methods, here each sample is used the same number of times for training and once for testing. For classification, the accuracy estimate is the overall number of correct classifications from the k iterations, divided by the total number of tuples in the initial data.

Leave-one-out-cross-validation is a special case of k-fold cross-validation where k is set to the number of initial tuples. That is, only one sample is "left out" at a time for the test set. Leave-one-out-cross-validation is often used when the initial data set is small. 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 practice, 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.

6.6.4 Bootstrap

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

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

"Where does the figure, 63.2%, come from?" Each tuple has a probability of $1/d$ of being selected, so the probability of not being chosen is $(1 - 1/d)$. We have to select d times, so the probability that a tuple will not be chosen during this whole time is $(1 - 1/d)^d$. If d is large, the probability approaches $e^{-1} = 0.368$.[14] 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, wherein each iteration, we use the current test set to obtain an estimated accuracy of the model obtained from the current bootstrap sample. The overall accuracy of the model, M, is then estimated as

$$Acc(M) = \frac{1}{k}\sum_{i=1}^{k}(0.632 \times Acc(M_i)_{test\_set} + 0.368 \times Acc(M_i)_{train\_set}), \qquad (6.30)$$

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

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

6.6.5 Model selection using statistical tests of significance

Suppose that we have generated two classification models, M_1 and M_2, from our data. We have performed 10-fold cross-validation to obtain a mean error rate[15] 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 the error on the true population of future data cases. There can be considerable variance between error rates within any given 10-fold cross-validation experiment. Although the mean error rates obtained for M_1 and M_2 may appear different, that difference may not be statistically significant. What if any difference between the two may just be attributed to chance? This section addresses these questions.

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

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

In data mining practice, we may often employ a single test set, that is, the same test set can be used for both M_1 and M_2. In such cases, we do a **pairwise comparison** of the two models *for each* 10-fold cross-validation round. That is, for the ith round of 10-fold cross-validation, the same cross-validation partitioning is used to obtain an error rate for M_1 and 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

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

from ten 10-fold cross-validations for each model. The t-statistic for pairwise comparison is computed as follows:

$$t = \frac{\overline{err}(M_1) - \overline{err}(M_2)}{\sqrt{var(M_1 - M_2)/k}}, \tag{6.31}$$

where

$$var(M_1 - M_2) = \frac{1}{k}\sum_{i=1}^{k}\left[err(M_1)_i - err(M_2)_i - (\overline{err}(M_1) - \overline{err}(M_2))\right]^2. \tag{6.32}$$

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

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

$$var(M_1 - M_2) = \frac{var(M_1)}{k_1} + \frac{var(M_2)}{k_2}, \tag{6.33}$$

and k_1 and k_2 are the number of cross-validation samples (in our case, 10-fold cross-validation rounds) used for M_1 and M_2, respectively. This is also known as the **two sample *t*-test**. 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.

6.6.6 Comparing classifiers based on cost–benefit and ROC curves

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

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

Receiver operating characteristic curves are a useful visual tool for comparing two classification models. ROC curves come from signal detection theory that was developed during World War II for the analysis of radar images. A ROC curve for a given model shows the trade-off between the *true positive rate* (TPR) and the *false positive rate* (FPR).[16] Given a test set and a model, TPR is the proportion of positive (or "yes") tuples that are correctly labeled by the model; FPR is the proportion of negative (or "no") tuples that are mislabeled as positive. Recall that TP, FP, P, and N are the number of true positive, false positive, positive, and negative tuples, respectively. From Section 6.6.1, we know that $TPR = \frac{TP}{P}$, which is sensitivity. Furthermore, $FPR = \frac{FP}{N}$, which is $1 -$ specificity.

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

To plot a ROC curve for a given classification model, M, the model must be able to return a probability of the predicted class for each test tuple. With this information, we rank and sort the tuples so that the tuple that is most likely to belong to the positive or "yes" class appears at the top of the list, and the tuple that is least likely to belong to the positive class lands at the bottom of the list. Naïve Bayesian (Section 6.3) and logistic regression (Section 6.5) classifiers return a class probability distribution for each prediction and, therefore, are appropriate, although other classifiers, such as decision tree classifiers (Section 6.2), can easily be modified to return class probability predictions. Let the value that a probabilistic classifier returns for a given tuple $\boldsymbol{X}$ be $f(\boldsymbol{X}) \to [0, 1]$. For a binary problem, a threshold t is typically selected so that tuples where $f(\boldsymbol{X}) \geq t$ are considered positive and all the other tuples are considered negative. Note that the number of true positives and the number of false positives are both functions of t, so that we could write $TP(t)$ and $FP(t)$. Both are monotonic nonincreasing functions.

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

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

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

FIGURE 6.20

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

Example 6.11. Plotting a ROC curve. Fig. 6.20 shows the probability value (column 3) returned by a probabilistic classifier for each of the 10 tuples in a test set, sorted in the decreasing probability order. Column 1 is merely a tuple identification number, which aids in our explanation. Column 2 is the actual class label of the tuple. There are five positive tuples and five negative tuples; thus $P = 5$ and $N = 5$. As we examine the known class label of each tuple, we can determine the values of the remaining columns, TP, FP, TN, FN, TPR, and FPR. We start with tuple 1, which has the highest probability score, and take that score as our threshold, that is, $t = 0.9$. Thus the classifier considers tuple 1 to be positive, and all the other tuples are considered negative. Since the actual class label of tuple 1 is positive, we have a true positive, hence $TP = 1$ and $FP = 0$. Among the remaining nine tuples, which are all classified as negative, five actually are negative (thus, $TN = 5$). The remaining four are all actually positive; thus, $FN = 4$. We can therefore compute $TPR = \frac{TP}{P} = \frac{1}{5} = 0.2$, whereas $FPR = 0$. Thus we have the point (0.2, 0) for the ROC curve.

Next, threshold t is set to 0.8, the probability value for tuple 2, so this tuple is now also considered positive, whereas tuples 3 through 10 are considered negative. The actual class label of tuple 2 is positive, thus now $TP = 2$. The rest of the row can easily be computed, resulting in the point (0.4, 0). Next, we examine the class label of tuple 3 and let t be 0.7, the probability value returned by the classifier for that tuple. Thus tuple 3 is considered positive, yet its actual label is negative, and so it is a false positive. Thus TP stays the same and FP increments so that $FP = 1$. The rest of the values in the row can also be easily computed, yielding the point (0.4, 0.2). The resulting ROC graph, from examining each tuple, is the jagged line shown in Fig. 6.21.

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

Fig. 6.22 shows the ROC curves of two classification models. The diagonal line representing random guessing is also shown. Thus the closer the ROC curve of a model is to the diagonal line, the less accurate the model. If the model is really good, initially we are more likely to encounter true positives as we move down the ranked list. Thus the curve moves steeply up from zero. Later, as we start to encounter fewer and fewer true positives, and more and more false positives, the curve eases off and becomes more horizontal.

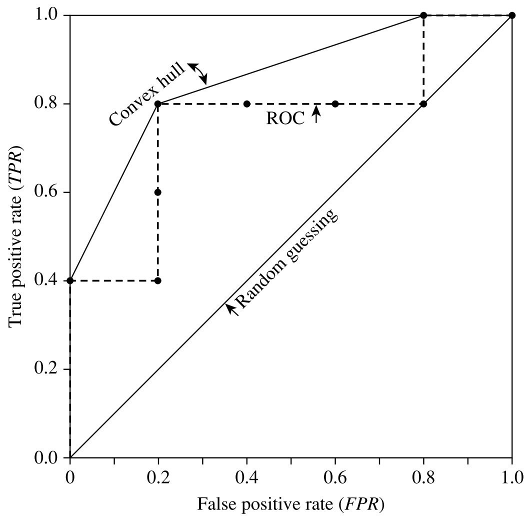


FIGURE 6.21

ROC curve for the data in Figure 6.20.

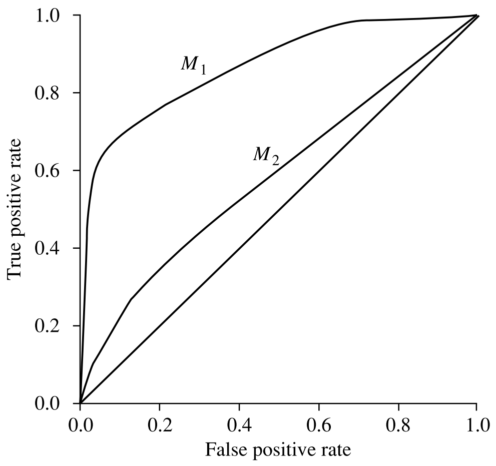


FIGURE 6.22

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

To assess the accuracy of a model, we can measure the area under the curve (AUC). 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 AUC of 1.0.