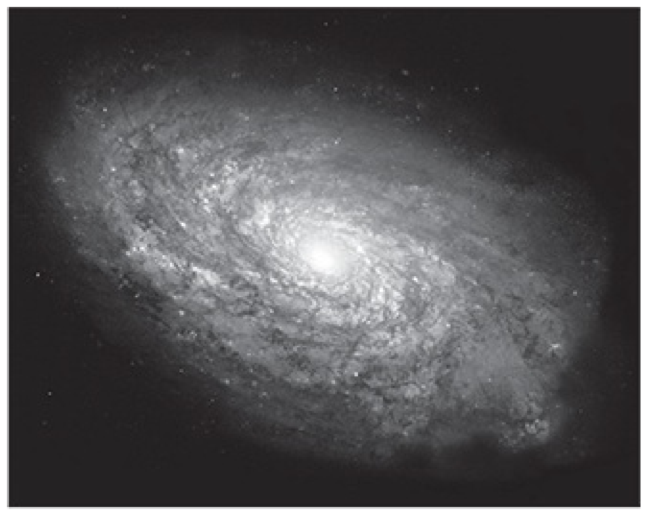
3 Classification: Basic Concepts and Techniques

Humans have an innate ability to classify things into categories, e.g., mundane tasks such as filtering spam email messages or more specialized tasks such as recognizing celestial objects in telescope images (see ***Figure 3.1****). While manual classification often suffices for small and simple data sets with only a few attributes, larger and more complex data sets require an automated solution.*



(a) A spiral galaxy.

(b) An elliptical galaxy.

Figure 3.1.

Classification of galaxies from telescope images taken from the NASA website.

*This chapter introduces the basic concepts of classification and describes some of its key issues such as model overfitting, model selection, and model evaluation. While these topics are illustrated using a classification technique known as decision tree induction, most of the discussion in this chapter is also applicable to other classification techniques, many of which are covered in **Chapter 4**.*

3.1 Basic Concepts

Figure 3.2 illustrates the general idea behind classification. The data for a classification task consists of a collection of instances (records). Each such instance is characterized by the tuple ($\mathbf{x}$, *y*), where $\mathbf{x}$ is the set of attribute values that describe the instance and *y* is the class label of the instance. The attribute set $\mathbf{x}$ can contain attributes of any type, while the class label *y* must be categorical.

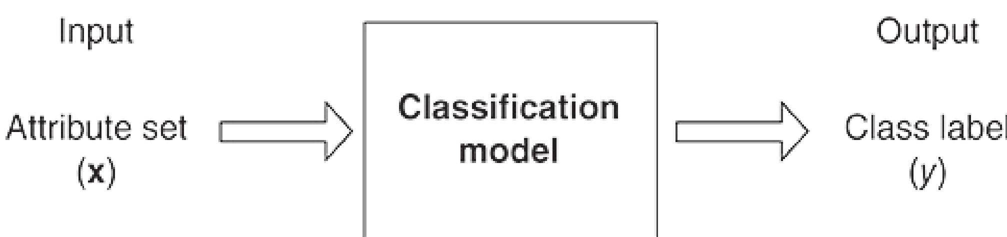


Figure 3.2.
A schematic illustration of a classification task.

A **classification model** is an abstract representation of the relationship between the attribute set and the class label. As will be seen in the next two chapters, the model can be represented in many ways, e.g., as a tree, a probability table, or simply, a vector of real-valued parameters. More formally, we can express it mathematically as a target function *f* that takes as input the attribute set $\mathbf{x}$ and produces an output corresponding to the predicted class label. The model is said to classify an instance ($\mathbf{x}$, *y*) correctly if f(x)=y.

Table 3.1 shows examples of attribute sets and class labels for various classification tasks. Spam filtering and tumor identification are examples of binary classification problems, in which each data instance can be categorized into one of two classes. If the number of classes is larger than 2, as in the

galaxy classification example, then it is called a multiclass classification problem.

Table 3.1. Examples of classification tasks.

Task	Attribute set	Class label
Spam filtering	Features extracted from email message header and content	spam or non-spam
Tumor identification	Features extracted from magnetic resonance imaging (MRI) scans	malignant or benign
Galaxy classification	Features extracted from telescope images	elliptical, spiral, or irregular-shaped

We illustrate the basic concepts of classification in this chapter with the following two examples.

3.1. Example Vertebrate Classification

Table 3.2 shows a sample data set for classifying vertebrates into mammals, reptiles, birds, fishes, and amphibians. The attribute set includes characteristics of the vertebrate such as its body temperature, skin cover, and ability to fly. The data set can also be used for a binary classification task such as mammal classification, by grouping the reptiles, birds, fishes, and amphibians into a single category called non-mammals.

Table 3.2. A sample data for the vertebrate classification problem.

Vertebrate Name	Body Temperature	Skin Cover	Gives Birth	Aquatic Creature	Aerial Creature	Has Legs	Hibernates	Class Label
human	warm-blooded	hair	yes	no	no	yes	no	mammal

	blooded							
python	cold-blooded	scales	no	no	no	no	yes	reptile
salmon	cold-blooded	scales	no	yes	no	no	no	fish
whale	warm-blooded	hair	yes	yes	no	no	no	mammal
frog	cold-blooded	none	no	semi	no	yes	yes	amphibian
komodo	cold-blooded	scales	no	no	no	yes	no	reptile
dragon								
bat	warm-blooded	hair	yes	no	yes	yes	yes	mammal
pigeon	warm-blooded	feathers	no	no	yes	yes	no	bird
cat	warm-blooded	fur	yes	no	no	yes	no	mammal
leopard	cold-blooded	scales	yes	yes	no	no	no	fish
shark								
turtle	cold-blooded	scales	no	semi	no	yes	no	reptile
penguin	warm-blooded	feathers	no	semi	no	yes	no	bird
porcupine	warm-blooded	quills	yes	no	no	yes	yes	mammal
eel	cold-blooded	scales	no	yes	no	no	no	fish
salamander	cold-blooded	none	no	semi	no	yes	yes	amphibian

3.2. Example Loan Borrower Classification

Consider the problem of predicting whether a loan borrower will repay the loan or default on the loan payments. The data set used to build the

classification model is shown in **Table 3.3**. The attribute set includes personal information of the borrower such as marital status and annual income, while the class label indicates whether the borrower had defaulted on the loan payments.

Table 3.3. A sample data for the loan borrower classification problem.

ID	Home Owner	Marital Status	Annual Income	Defaulted?
1	Yes	Single	125000	No
2	No	Married	100000	No
3	No	Single	70000	No
4	Yes	Married	120000	No
5	No	Divorced	95000	Yes
6	No	Single	60000	No
7	Yes	Divorced	220000	No
8	No	Single	85000	Yes
9	No	Married	75000	No
10	No	Single	90000	Yes

A classification model serves two important roles in data mining. First, it is used as a **predictive model** to classify previously unlabeled instances. A good classification model must provide accurate predictions with a fast response time. Second, it serves as a **descriptive model** to identify the characteristics that distinguish instances from different classes. This is particularly useful for critical applications, such as medical diagnosis, where it

is insufficient to have a model that makes a prediction without justifying how it reaches such a decision.

For example, a classification model induced from the vertebrate data set shown in **Table 3.2** can be used to predict the class label of the following vertebrate:

Vertebrate Name	Body Temperature	Skin Cover	Gives Birth	Aquatic Creature	Aerial Creature	Has Legs	Hibernates	Class Label
gila monster	cold-blooded	scales	no	no	no	yes	yes	?

In addition, it can be used as a descriptive model to help determine characteristics that define a vertebrate as a mammal, a reptile, a bird, a fish, or an amphibian. For example, the model may identify mammals as warm-blooded vertebrates that give birth to their young.

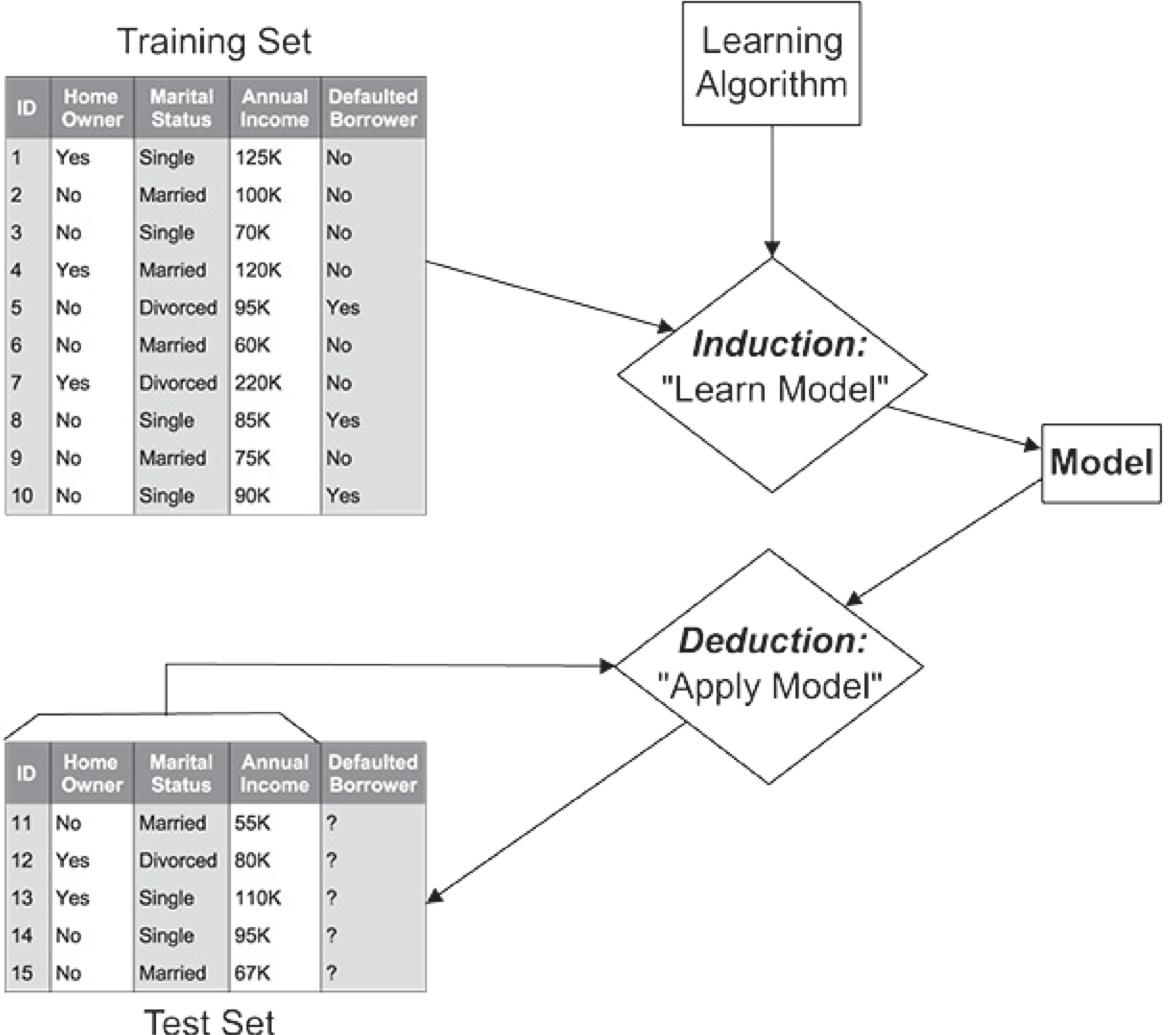
There are several points worth noting regarding the previous example. First, although all the attributes shown in **Table 3.2** are qualitative, there are no restrictions on the type of attributes that can be used as predictor variables. The class label, on the other hand, must be of nominal type. This distinguishes classification from other predictive modeling tasks such as regression, where the predicted value is often quantitative. More information about regression can be found in Appendix D.

Another point worth noting is that not all attributes may be relevant to the classification task. For example, the average length or weight of a vertebrate may not be useful for classifying mammals, as these attributes can show same value for both mammals and non-mammals. Such an attribute is typically discarded during preprocessing. The remaining attributes might not be able to distinguish the classes by themselves, and thus, must be used in

concert with other attributes. For instance, the Body Temperature attribute is insufficient to distinguish mammals from other vertebrates. When it is used together with Gives Birth, the classification of mammals improves significantly. However, when additional attributes, such as Skin Cover are included, the model becomes overly specific and no longer covers all mammals. Finding the optimal combination of attributes that best discriminates instances from different classes is the key challenge in building classification models.

3.2 General Framework for Classification

Classification is the task of assigning labels to unlabeled data instances and a **classifier** is used to perform such a task. A classifier is typically described in terms of a model as illustrated in the previous section. The model is created using a given a set of instances, known as the **training set**, which contains attribute values as well as class labels for each instance. The systematic approach for learning a classification model given a training set is known as a **learning algorithm**. The process of using a learning algorithm to build a classification model from the training data is known as **induction**. This process is also often described as "learning a model" or "building a model." This process of applying a classification model on unseen test instances to predict their class labels is known as **deduction**. Thus, the process of classification involves two steps: applying a learning algorithm to training data to learn a model, and then applying the model to assign labels to unlabeled instances. Figure 3.3 illustrates the general framework for classification.



ID	Home Owner	Marital Status	Annual Income	Defaulted Borrower
1	Yes	Single	125K	No
2	No	Married	100K	No
3	No	Single	70K	No
4	Yes	Married	120K	No
5	No	Divorced	95K	Yes
6	No	Married	60K	No
7	Yes	Divorced	220K	No
8	No	Single	85K	Yes
9	No	Married	75K	No
10	No	Single	90K	Yes

ID	Home Owner	Marital Status	Annual Income	Defaulted Borrower
11	No	Married	55K	?
12	Yes	Divorced	80K	?
13	Yes	Single	110K	?
14	No	Single	95K	?
15	No	Married	67K	?

Figure 3.3.

General framework for building a classification model.

A **classification technique** refers to a general approach to classification, e.g., the decision tree technique that we will study in this chapter. This classification technique like most others, consists of a family of related models and a number of algorithms for learning these models. In **Chapter 4**, we will study additional classification techniques, including neural networks and support vector machines.

A couple notes on terminology. First, the terms “classifier” and “model” are often taken to be synonymous. If a classification technique builds a single,

global model, then this is fine. However, while every model defines a classifier, not every classifier is defined by a single model. Some classifiers, such as *k*-nearest neighbor classifiers, do not build an explicit model (**Section 4.3**), while other classifiers, such as ensemble classifiers, combine the output of a collection of models (**Section 4.10**). Second, the term “classifier” is often used in a more general sense to refer to a classification technique. Thus, for example, “decision tree classifier” can refer to the decision tree classification technique or a specific classifier built using that technique. Fortunately, the meaning of “classifier” is usually clear from the context.

In the general framework shown in **Figure 3.3**, the induction and deduction steps should be performed separately. In fact, as will be discussed later in **Section 3.6**, the training and test sets should be independent of each other to ensure that the induced model can accurately predict the class labels of instances it has never encountered before. Models that deliver such predictive insights are said to have good **generalization performance**. The performance of a model (classifier) can be evaluated by comparing the predicted labels against the true labels of instances. This information can be summarized in a table called a **confusion matrix**. **Table 3.4** depicts the confusion matrix for a binary classification problem. Each entry fij denotes the number of instances from class *i* predicted to be of class *j*. For example, f01 is the number of instances from class 0 incorrectly predicted as class 1. The number of correct predictions made by the model is (f11+f00) and the number of incorrect predictions is (f10+f01).

Table 3.4. Confusion matrix for a binary classification problem.

		Predicted Class	
		Class=1	Class=0
Actual Class	Class=1	f11	f10

	Class=0	f01	f00

Although a confusion matrix provides the information needed to determine how well a classification model performs, summarizing this information into a single number makes it more convenient to compare the relative performance of different models. This can be done using an **evaluation metric** such as **accuracy**, which is computed in the following way:

Accuracy =

Accuracy=Number of correct predictionsTotal number of predictions. (3.1)

For binary classification problems, the accuracy of a model is given by

Accuracy=f11+f00f11+f10+f01+f00. (3.2)

Error rate is another related metric, which is defined as follows for binary classification problems:

Error rate=Number of wrong predictionsTotal number of predictions=f10+f01 (3.3)

The learning algorithms of most classification techniques are designed to learn models that attain the highest accuracy, or equivalently, the lowest error rate when applied to the test set. We will revisit the topic of model evaluation in **Section 3.6**.

3.3 Decision Tree Classifier

This section introduces a simple classification technique known as the **decision tree** classifier. To illustrate how a decision tree works, consider the classification problem of distinguishing mammals from non-mammals using the vertebrate data set shown in **Table 3.2**. Suppose a new species is discovered by scientists. How can we tell whether it is a mammal or a non-mammal? One approach is to pose a series of questions about the characteristics of the species. The first question we may ask is whether the species is cold- or warm-blooded. If it is cold-blooded, then it is definitely not a mammal. Otherwise, it is either a bird or a mammal. In the latter case, we need to ask a follow-up question: Do the females of the species give birth to their young? Those that do give birth are definitely mammals, while those that do not are likely to be non-mammals (with the exception of egg-laying mammals such as the platypus and spiny anteater).

The previous example illustrates how we can solve a classification problem by asking a series of carefully crafted questions about the attributes of the test instance. Each time we receive an answer, we could ask a follow-up question until we can conclusively decide on its class label. The series of questions and their possible answers can be organized into a hierarchical structure called a decision tree. **Figure 3.4** shows an example of the decision tree for the mammal classification problem. The tree has three types of nodes:

- A **root node**, with no incoming links and zero or more outgoing links.
- **Internal nodes**, each of which has exactly one incoming link and two or more outgoing links.
- **Leaf** or **terminal** nodes, each of which has exactly one incoming link and no outgoing links.

Every leaf node in the decision tree is associated with a class label. The **non-terminal** nodes, which include the root and internal nodes, contain **attribute test conditions** that are typically defined using a single attribute. Each possible outcome of the attribute test condition is associated with exactly one child of this node. For example, the root node of the tree shown in **Figure 3.4** uses the attribute `Body Temperature` to define an attribute test condition that has two outcomes, warm and cold, resulting in two child nodes.

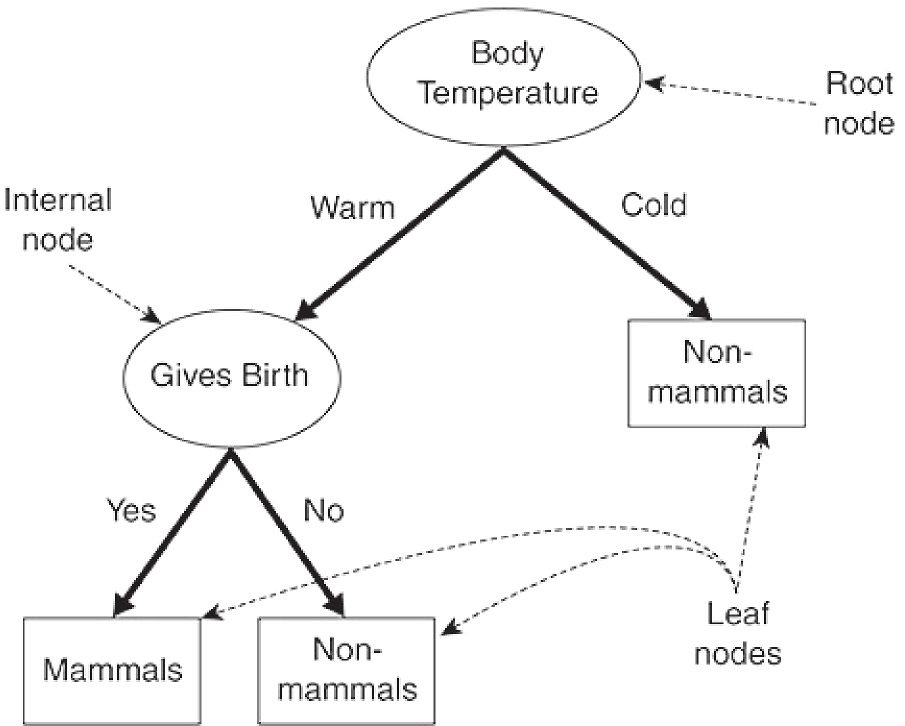


Figure 3.4.
A decision tree for the mammal classification problem.

Given a decision tree, classifying a test instance is straightforward. Starting from the root node, we apply its attribute test condition and follow the appropriate branch based on the outcome of the test. This will lead us either to another internal node, for which a new attribute test condition is applied, or to a leaf node. Once a leaf node is reached, we assign the class label associated with the node to the test instance. As an illustration, **Figure 3.5**

traces the path used to predict the class label of a flamingo. The path terminates at a leaf node labeled as `Non-mammals`.

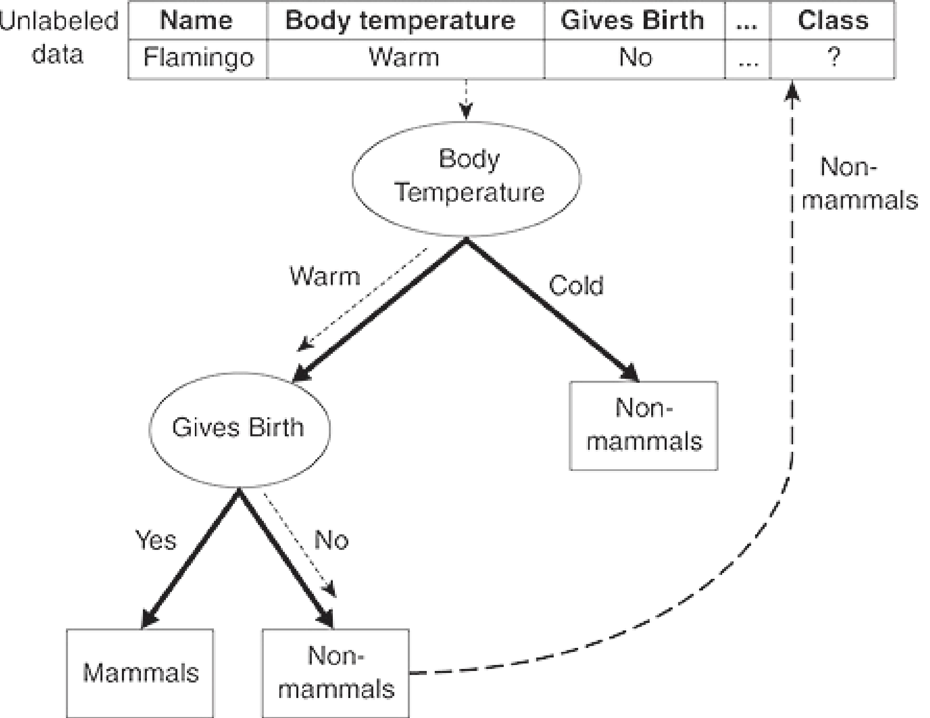


Figure 3.5.

Classifying an unlabeled vertebrate. The dashed lines represent the outcomes of applying various attribute test conditions on the unlabeled vertebrate. The vertebrate is eventually assigned to the `Non-mammals` class.

3.3.1 A Basic Algorithm to Build a Decision Tree

Many possible decision trees that can be constructed from a particular data set. While some trees are better than others, finding an optimal one is computationally expensive due to the exponential size of the search space. Efficient algorithms have been developed to induce a reasonably accurate,

albeit suboptimal, decision tree in a reasonable amount of time. These algorithms usually employ a greedy strategy to grow the decision tree in a top-down fashion by making a series of locally optimal decisions about which attribute to use when partitioning the training data. One of the earliest method is **Hunt's algorithm**, which is the basis for many current implementations of decision tree classifiers, including ID3, C4.5, and CART. This subsection presents Hunt's algorithm and describes some of the design issues that must be considered when building a decision tree.

Hunt's Algorithm

In Hunt's algorithm, a decision tree is grown in a recursive fashion. The tree initially contains a single root node that is associated with all the training instances. If a node is associated with instances from more than one class, it is expanded using an attribute test condition that is determined using a **splitting criterion**. A child leaf node is created for each outcome of the attribute test condition and the instances associated with the parent node are distributed to the children based on the test outcomes. This node expansion step can then be recursively applied to each child node, as long as it has labels of more than one class. If all the instances associated with a leaf node have identical class labels, then the node is not expanded any further. Each leaf node is assigned a class label that occurs most frequently in the training instances associated with the node.

To illustrate how the algorithm works, consider the training set shown in **Table 3.3** for the loan borrower classification problem. Suppose we apply Hunt's algorithm to fit the training data. The tree initially contains only a single leaf node as shown in **Figure 3.6(a)**. This node is labeled as Defaulted = No, since the majority of the borrowers did not default on their loan payments. The training error of this tree is 30% as three out of the ten training instances have

the class label Defaulted = Yes. The leaf node can therefore be further expanded because it contains training instances from more than one class.

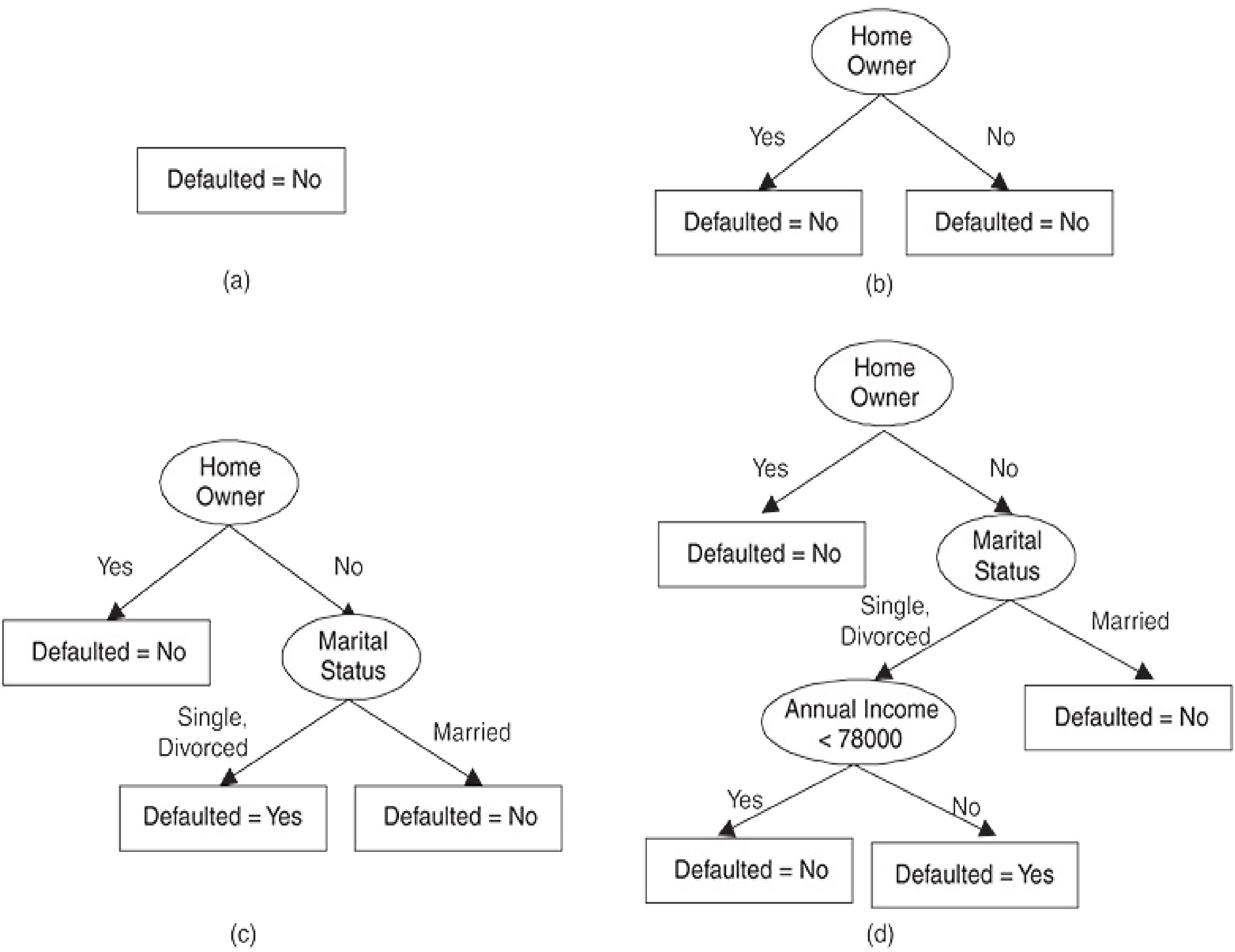


Figure 3.6.

Hunt's algorithm for building decision trees.

Let Home Owner be the attribute chosen to split the training instances. The justification for choosing this attribute as the attribute test condition will be discussed later. The resulting binary split on the Home Owner attribute is shown in **Figure 3.6(b)**. All the training instances for which Home Owner = Yes are propagated to the left child of the root node and the rest are propagated to the right child. Hunt's algorithm is then recursively applied to each child. The left child becomes a leaf node labeled Defaulted = No, since

all instances associated with this node have identical class label Defaulted = No. The right child has instances from each class label. Hence, we split it further. The resulting subtrees after recursively expanding the right child are shown in **Figures 3.6(c)** and **(d)**.

Hunt's algorithm, as described above, makes some simplifying assumptions that are often not true in practice. In the following, we describe these assumptions and briefly discuss some of the possible ways for handling them.

1. Some of the child nodes created in Hunt's algorithm can be empty if none of the training instances have the particular attribute values. One way to handle this is by declaring each of them as a leaf node with a class label that occurs most frequently among the training instances associated with their parent nodes.
2. If all training instances associated with a node have identical attribute values but different class labels, it is not possible to expand this node any further. One way to handle this case is to declare it a leaf node and assign it the class label that occurs most frequently in the training instances associated with this node.

Design Issues of Decision Tree Induction

Hunt's algorithm is a generic procedure for growing decision trees in a greedy fashion. To implement the algorithm, there are two key design issues that must be addressed.

1. **What is the splitting criterion?** At each recursive step, an attribute must be selected to partition the training instances associated with a node into smaller subsets associated with its child nodes. The splitting criterion determines which attribute is chosen as the test condition and

how the training instances should be distributed to the child nodes. This will be discussed in **Sections 3.3.2** and **3.3.3**.
2. **What is the stopping criterion?** The basic algorithm stops expanding a node only when all the training instances associated with the node have the same class labels or have identical attribute values. Although these conditions are sufficient, there are reasons to stop expanding a node much earlier even if the leaf node contains training instances from more than one class. This process is called early termination and the condition used to determine when a node should be stopped from expanding is called a stopping criterion. The advantages of early termination are discussed in **Section 3.4**.

3.3.2 Methods for Expressing Attribute Test Conditions

Decision tree induction algorithms must provide a method for expressing an attribute test condition and its corresponding outcomes for different attribute types.

Binary Attributes

The test condition for a binary attribute generates two potential outcomes, as shown in **Figure 3.7**.

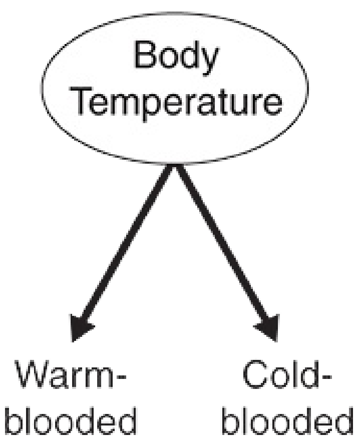


Figure 3.7.

Attribute test condition for a binary attribute.

Nominal Attributes

Since a nominal attribute can have many values, its attribute test condition can be expressed in two ways, as a multiway split or a binary split as shown in **Figure 3.8**. For a multiway split (**Figure 3.8(a)**), the number of outcomes depends on the number of distinct values for the corresponding attribute. For example, if an attribute such as marital status has three distinct values—single, married, or divorced—its test condition will produce a three-way split. It is also possible to create a binary split by partitioning all values taken by the nominal attribute into two groups. For example, some decision tree algorithms, such as CART, produce only binary splits by considering all 2k −1−1 ways of creating a binary partition of *k* attribute values. **Figure 3.8(b)** illustrates three different ways of grouping the attribute values for marital status into two subsets.

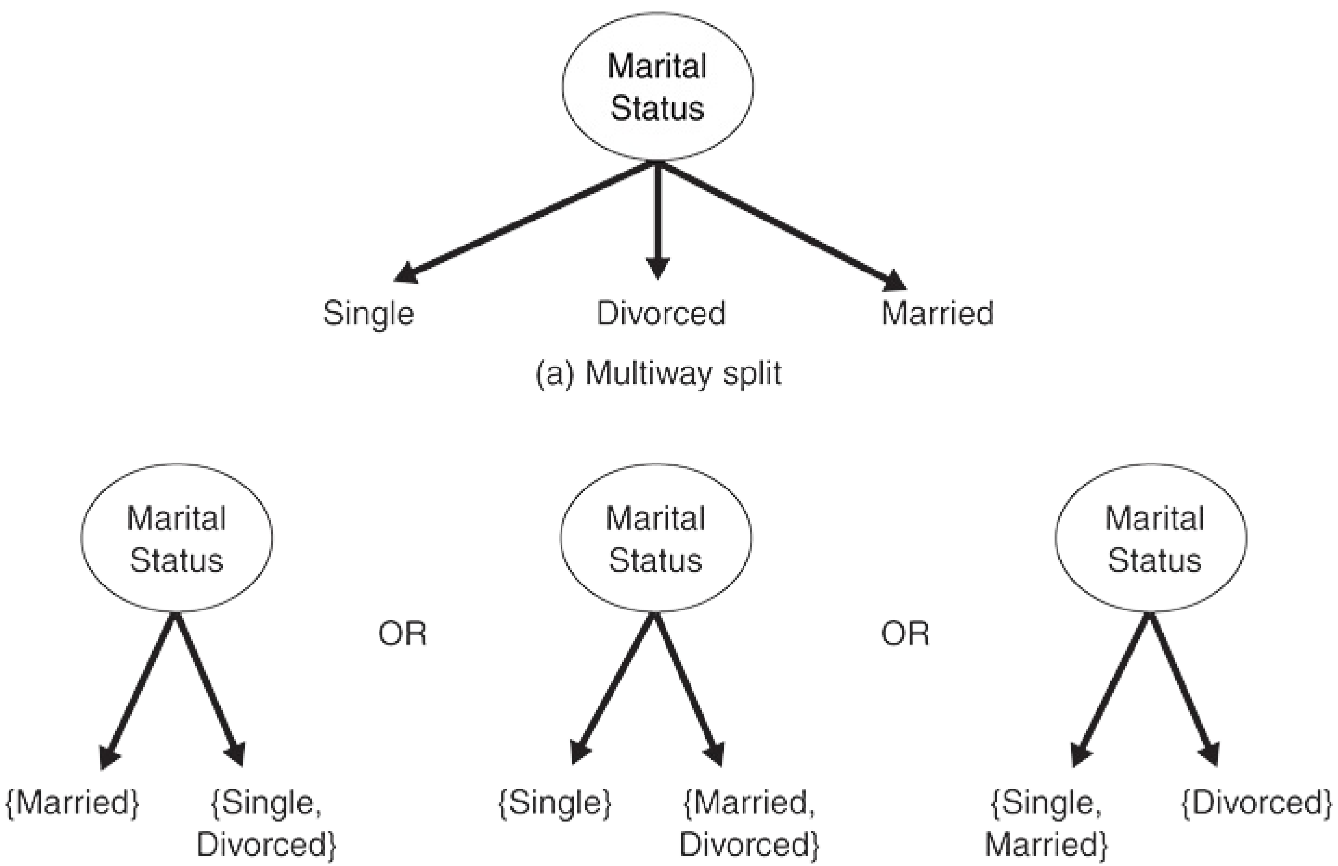


Figure 3.8.
Attribute test conditions for nominal attributes.

Ordinal Attributes

Ordinal attributes can also produce binary or multi-way splits. Ordinal attribute values can be grouped as long as the grouping does not violate the order property of the attribute values. **Figure 3.9** illustrates various ways of splitting training records based on the Shirt Size attribute. The groupings shown in **Figures 3.9(a)** and **(b)** preserve the order among the attribute values, whereas the grouping shown in **Figure 3.9(c)** violates this property because it combines the attribute values Small and Large into the same partition while Medium and Extra Large are combined into another partition.

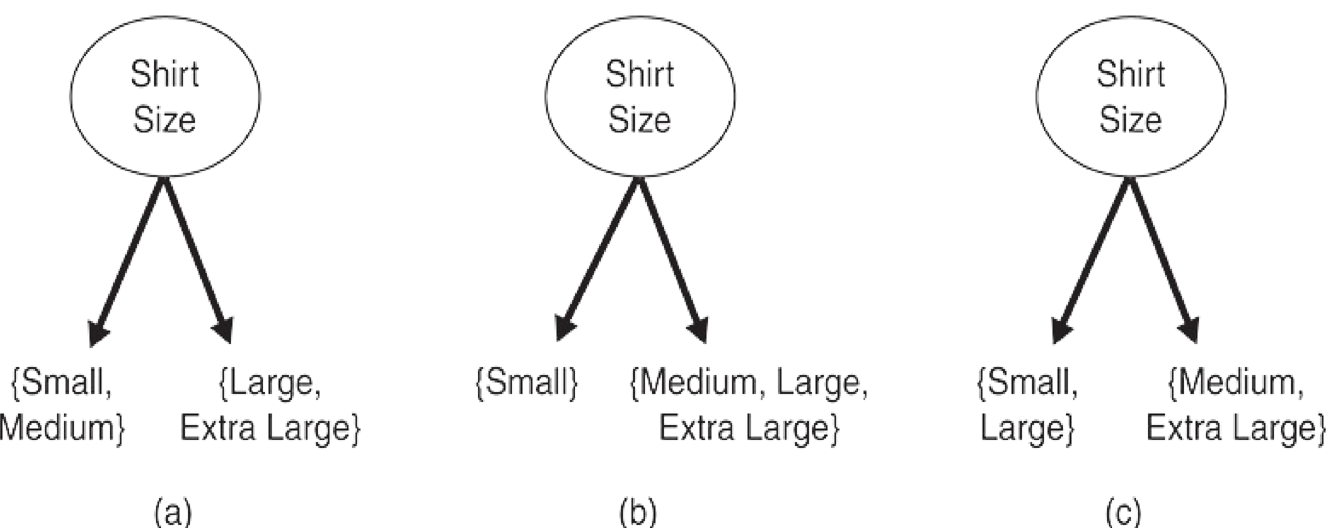


Figure 3.9.

Different ways of grouping ordinal attribute values.

Continuous Attributes

For continuous attributes, the attribute test condition can be expressed as a comparison test (e.g., $A<v$) producing a binary split, or as a range query of the form $v_i \le A < v_{i+1}$, for $i=1, \ldots, k$, producing a multiway split. The difference between these approaches is shown in **Figure 3.10**. For the binary split, any possible value *v* between the minimum and maximum attribute values in the training data can be used for constructing the comparison test $A<v$. However, it is sufficient to only consider distinct attribute values in the training set as candidate split positions. For the multiway split, any possible collection of attribute value ranges can be used, as long as they are mutually exclusive and cover the entire range of attribute values between the minimum and maximum values observed in the training set. One approach for constructing multiway splits is to apply the discretization strategies described in **Section 2.3.6** on **page 63**. After discretization, a new ordinal value is assigned to each discretized interval, and the attribute test condition is then defined using this newly constructed ordinal attribute.

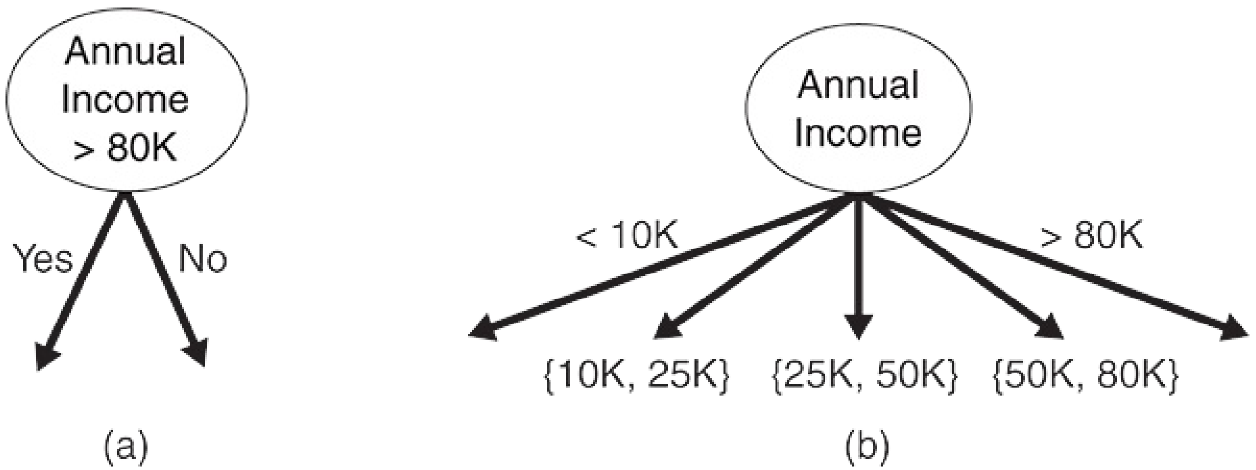


Figure 3.10.

Test condition for continuous attributes.

3.3.3 Measures for Selecting an Attribute Test Condition

There are many measures that can be used to determine the goodness of an attribute test condition. These measures try to give preference to attribute test conditions that partition the training instances into *purer* subsets in the child nodes, which mostly have the same class labels. Having purer nodes is useful since a node that has all of its training instances from the same class does not need to be expanded further. In contrast, an impure node containing training instances from multiple classes is likely to require several levels of node expansions, thereby increasing the depth of the tree considerably. Larger trees are less desirable as they are more susceptible to model overfitting, a condition that may degrade the classification performance on unseen instances, as will be discussed in **Section 3.4**. They are also difficult to interpret and incur more training and test time as compared to smaller trees.

In the following, we present different ways of measuring the impurity of a node and the collective impurity of its child nodes, both of which will be used to identify the best attribute test condition for a node.

Impurity Measure for a Single Node

The impurity of a node measures how dissimilar the class labels are for the data instances belonging to a common node. Following are examples of measures that can be used to evaluate the impurity of a node *t*:

$$\text{Entropy} = -\sum_{i=0}^{c-1} p_i(t) \log_2 p_i(t), \tag{3.4}$$

$$\text{Gini index} = 1 - \sum_{i=0}^{c-1} p_i(t)^2, \tag{3.5}$$

$$\text{Classification error} = 1 - \max_i [p_i(t)], \tag{3.6}$$

where $p_i(t)$ is the relative frequency of training instances that belong to class i at node t, c is the total number of classes, and $0 \log_2 0 = 0$ in entropy calculations. All three measures give a zero impurity value if a node contains instances from a single class and maximum impurity if the node has equal proportion of instances from multiple classes.

Figure 3.11 compares the relative magnitude of the impurity measures when applied to binary classification problems. Since there are only two classes, $p_0(t)+p_1(t)=1$. The horizontal axis p refers to the fraction of instances that belong to one of the two classes. Observe that all three measures attain their maximum value when the class distribution is uniform (i.e., $p_0(t)+p_1(t)=0.5$) and minimum value when all the instances belong to a single class (i.e., either $p_0(t)$ or $p_1(t)$ equals to 1). The following examples illustrate how the values of the impurity measures vary as we alter the class distribution.

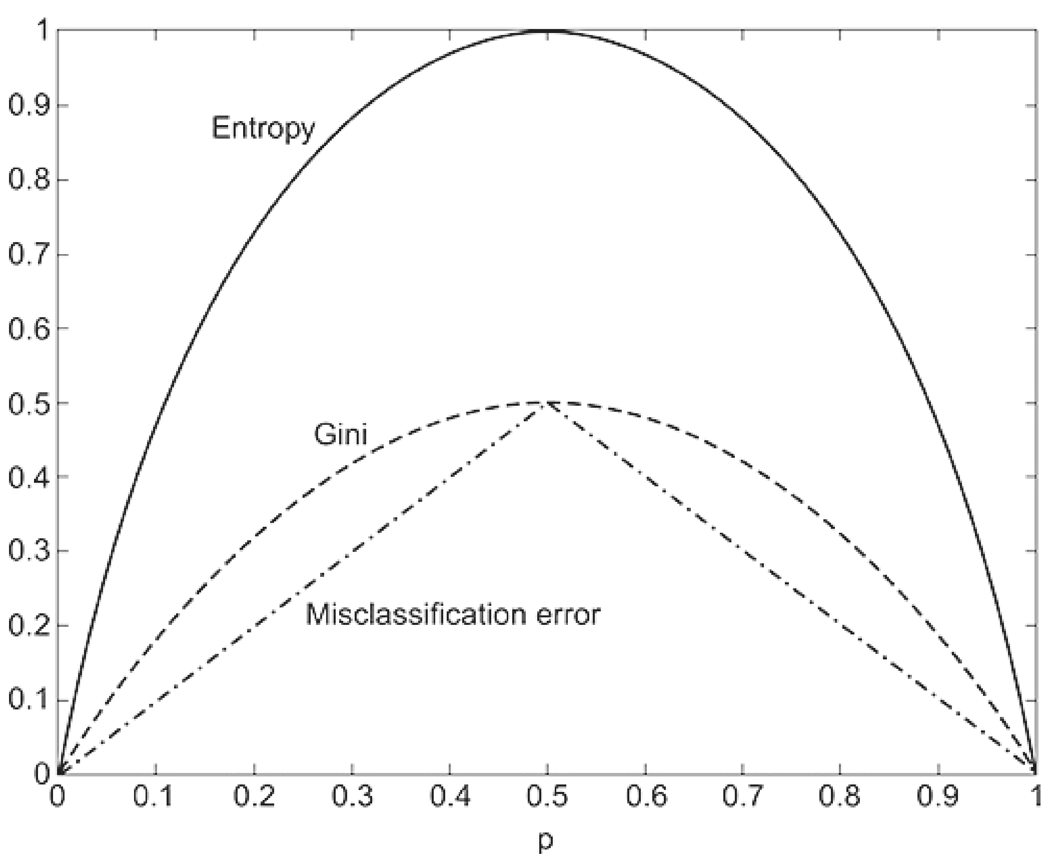


Figure 3.11.

Comparison among the impurity measures for binary classification problems.

Node N1	Count	Gini=1−(0/6)2−(6/6)2=0
Class=0	0	Entropy=−(0/6) log2(0/6)−(6/6) log2(6/6)=0
Class=1	6	Error=1−max[0/6, 6/6]=0
Node N2	Count	Gini=1−(1/6)2−(5/6)2=0.278
Class=0	1	Entropy=−(1/6) log2(1/6)−(5/6) log2(5/6)=0.650
Class=1	5	Error=1−max[1/6, 5/6]=0.167
Node N3	Count	Gini=1−(3/6)2−(3/6)2=0.5
Class=0	3	Entropy=−(3/6) log2(3/6)−(3/6) log2(3/6)=1

Class=1	3	Error=1−max[6/6, 3/6]=0.5

Based on these calculations, node N1 has the lowest impurity value, followed by N2 and N3. This example, along with **Figure 3.11**, shows the consistency among the impurity measures, i.e., if a node N1 has lower entropy than node N2, then the Gini index and error rate of N1 will also be lower than that of N2. Despite their agreement, the attribute chosen as splitting criterion by the impurity measures can still be different (see Exercise 6 on page 187).

Collective Impurity of Child Nodes

Consider an attribute test condition that splits a node containing *N* training instances into *k* children, $\{v_1, v_2, \cdots, v_k\}$, where every child node represents a partition of the data resulting from one of the *k* outcomes of the attribute test condition. Let $N(v_j)$ be the number of training instances associated with a child node v_j, whose impurity value is $I(v_j)$. Since a training instance in the parent node reaches node v_j for a fraction of $N(v_j)/N$ times, the collective impurity of the child nodes can be computed by taking a weighted sum of the impurities of the child nodes, as follows:

$$I(\text{children}) = \sum_{j=1}^{k} \frac{N(v_j)}{N} I(v_j), \qquad (3.7)$$

3.3. Example Weighted Entropy

Consider the candidate attribute test condition shown in **Figures 3.12(a)** and **(b)** for the loan borrower classification problem. Splitting on the Home Owner attribute will generate two child nodes

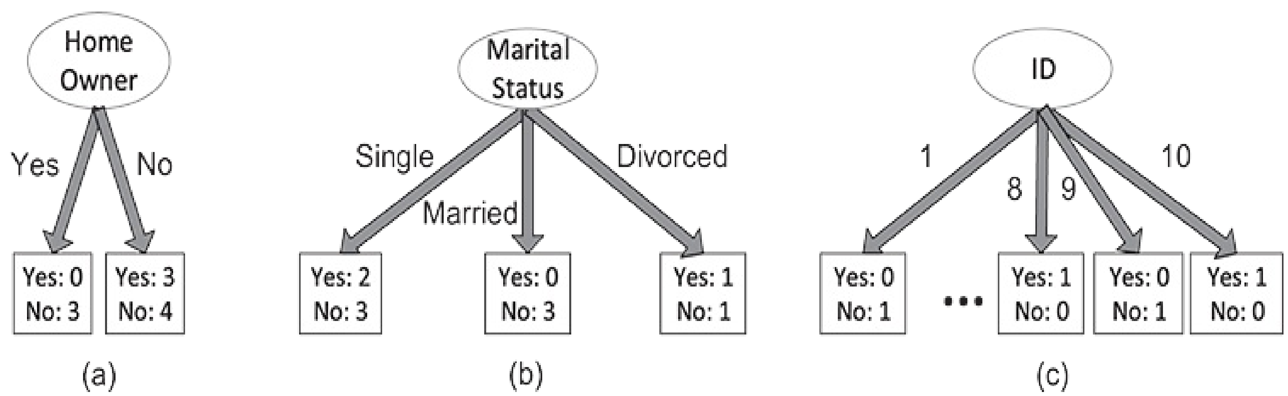


Figure 3.12.
Examples of candidate attribute test conditions.

whose weighted entropy can be calculated as follows:

$$I(\text{Home Owner}=\text{yes}) = \frac{0}{3}\log_2\frac{0}{3} - \frac{3}{3}\log_2\frac{3}{3} = 0$$
$$I(\text{Home Owner}=\text{no}) = -\frac{3}{7}\log_2\frac{3}{7} - \frac{4}{7}\log_2\frac{4}{7} = 0.985$$
$$I(\text{Home Owner} = \frac{3}{10}\times 0 + \frac{7}{10}\times 0.985 = 0.690$$

Splitting on Marital Status, on the other hand, leads to three child nodes with a weighted entropy given by

$$I(\text{Marital Status}=\text{Single}) = -\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5} = 0.971$$
$$I(\text{Marital Status}=\text{Married}) = -\frac{0}{3}\log_2\frac{0}{3} - \frac{3}{3}\log_2\frac{3}{3} = 0$$
$$I(\text{Marital Status}=\text{Divorced}) = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1.000$$
$$I(\text{Marital Status}) = \frac{5}{10}\times 0.971 + \frac{3}{10}\times 0 + \frac{2}{10}\times 1 = 0.$$

Thus, Marital Status has a lower weighted entropy than Home Owner.

Identifying the best attribute test condition

To determine the goodness of an attribute test condition, we need to compare the degree of impurity of the parent node (before splitting) with the weighted degree of impurity of the child nodes (after splitting). The larger their

difference, the better the test condition. This difference, Δ, also termed as the **gain** in purity of an attribute test condition, can be defined as follows:

$$\Delta = I(\text{parent}) - I(\text{children}), \quad (3.8)$$

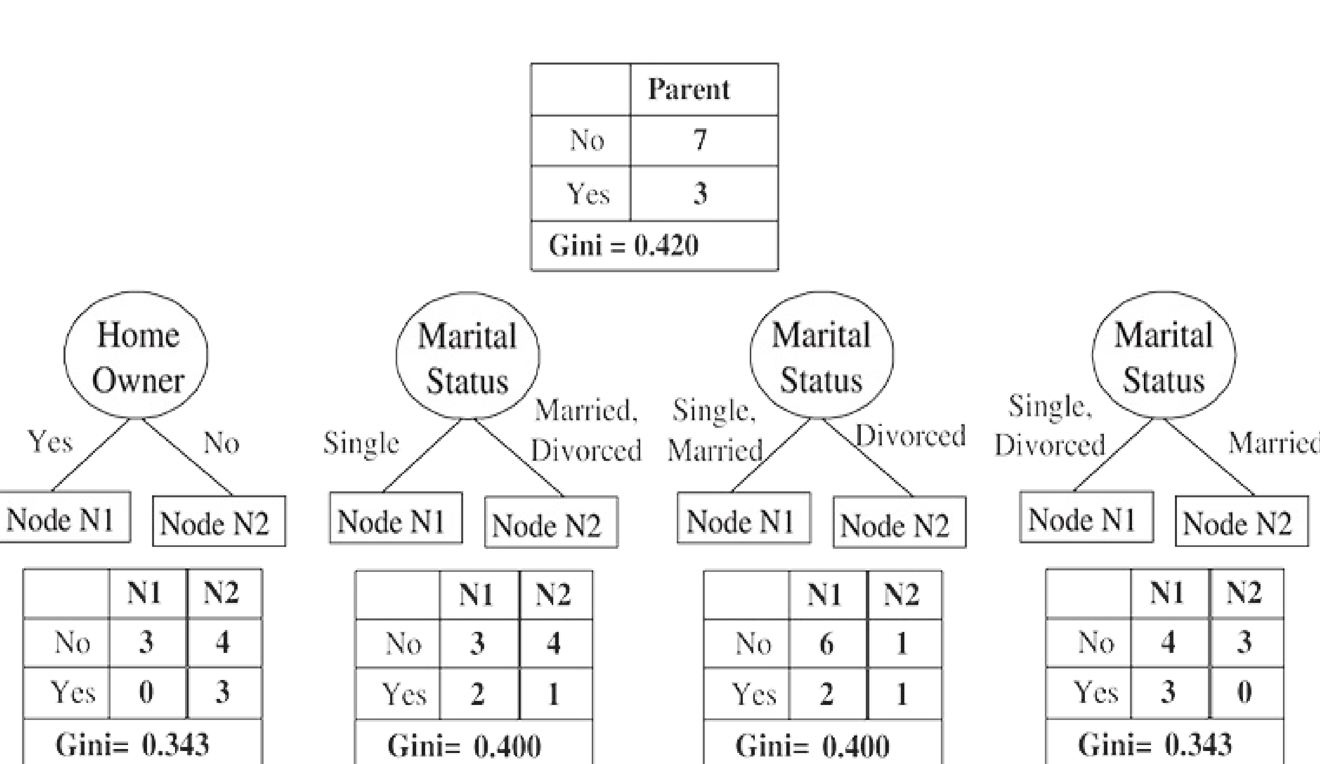


Figure 3.13.
Splitting criteria for the loan borrower classification problem using Gini index.

where *I*(parent) is the impurity of a node before splitting and *I*(children) is the weighted impurity measure after splitting. It can be shown that the gain is non-negative since $I(\text{parent}) \geq I(\text{children})$ for any reasonable measure such as those presented above. The higher the gain, the purer are the classes in the child nodes relative to the parent node. The splitting criterion in the decision tree learning algorithm selects the attribute test condition that shows the maximum gain. Note that maximizing the gain at a given node is equivalent to minimizing the weighted impurity measure of its children since *I*(parent) is the same for all candidate attribute test conditions. Finally, when entropy is used

as the impurity measure, the difference in entropy is commonly known as **information gain**, Δinfo.

In the following, we present illustrative approaches for identifying the best attribute test condition given qualitative or quantitative attributes.

Splitting of Qualitative Attributes

Consider the first two candidate splits shown in **Figure 3.12** involving qualitative attributes `Home Owner` and `Marital Status`. The initial class distribution at the parent node is (0.3, 0.7), since there are 3 instances of class `Yes` and 7 instances of class `No` in the training data. Thus,

$$I(\text{parent}) = -\frac{3}{10}\log_2\frac{3}{10} - \frac{7}{10}\log_2\frac{7}{10} = 0.881$$

The information gains for Home Owner and Marital Status are each given by

$$\Delta\text{info}(\text{Home Owner}) = 0.881 - 0.690 = 0.191 \quad \Delta\text{info}(\text{Marital Status}) = 0.881 - 0.686 = 0.$$

The information gain for Marital Status is thus higher due to its lower weighted entropy, which will thus be considered for splitting.

Binary Splitting of Qualitative Attributes

Consider building a decision tree using only binary splits and the Gini index as the impurity measure. **Figure 3.13** shows examples of four candidate splitting criteria for the `Home Owner` and `Marital Status` attributes. Since there are 3 borrowers in the training set who defaulted and 7 others who repaid their loan (see Table in **Figure 3.13**), the Gini index of the parent node before splitting is

$1-(310)2-(710)2=0.420.$

If `Home Owner` is chosen as the splitting attribute, the Gini index for the child nodes N1 and N2 are 0 and 0.490, respectively. The weighted average Gini index for the children is

$(3/10)\times 0+(7/10)\times 0.490=0.343,$

where the weights represent the proportion of training instances assigned to each child. The gain using `Home Owner` as splitting attribute is $0.420-0.343=0.077$. Similarly, we can apply a binary split on the `Marital Status` attribute. However, since `Marital Status` is a nominal attribute with three outcomes, there are three possible ways to group the attribute values into a binary split. The weighted average Gini index of the children for each candidate binary split is shown in **Figure 3.13**. Based on these results, `Home Owner` and the last binary split using `Marital Status` are clearly the best candidates, since they both produce the lowest weighted average Gini index. Binary splits can also be used for ordinal attributes, if the binary partitioning of the attribute values does not violate the ordering property of the values.

Binary Splitting of Quantitative Attributes

Consider the problem of identifying the best binary split $\text{Annual Income} \le \tau$ for the preceding loan approval classification problem. As discussed previously, even though τ can take any value between the minimum and maximum values of annual income in the training set, it is sufficient to only consider the annual income values observed in the training set as candidate split positions. For each candidate τ, the training set is scanned once to count the number of borrowers with annual income less than or greater than τ along with their class proportions. We can then compute the Gini index at each candidate split

position and choose the τ that produces the lowest value. Computing the Gini index at each candidate split position requires $O(N)$ operations, where N is the number of training instances. Since there are at most N possible candidates, the overall complexity of this brute-force method is O(N2). It is possible to reduce the complexity of this problem to $O(N \log N)$ by using a method described as follows (see illustration in **Figure 3.14**). In this method, we first sort the training instances based on their annual income, a one-time cost that requires $O(N \log N)$ operations. The candidate split positions are given by the midpoints between every two adjacent sorted values: $55,000, $65,000, $72,500, and so on. For the first candidate, since none of the instances has an annual income less than or equal to $55,000, the Gini index for the child node with Annual Income< $55,000 is equal to zero. In contrast, there are 3 training instances of class `Yes` and `7` instances of class No with annual income greater than $55,000. The Gini index for this node is 0.420. The weighted average Gini index for the first candidate split position, τ=$55,000, is equal to 0×0+1×0.420=0.420.

Class	No	No	No	Yes	Yes	Yes	No	No	No	No
	Annual Income (in '000s)									
Sorted Values	60	70	75	85	90	95	100	120	125	220

Split Positions	55		65		72.5		80		87.5		92.5		97.5		110		122.5		172.5		230	
	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>	<=	>
Yes	0	3	0	3	0	3	0	3	1	2	2	1	3	0	3	0	3	0	3	0	3	0
No	0	7	1	6	2	5	3	4	3	4	3	4	3	4	4	3	5	2	6	1	7	0
Gini	0.420		0.400		0.375		0.343		0.417		0.400		*0.300*		0.343		0.375		0.400		0.420	

Figure 3.14.
Splitting continuous attributes.

For the next candidate, τ=$65,000, the class distribution of its child nodes can be obtained with a simple update of the distribution for the previous candidate. This is because, as τ increases from $55,000 to $65,000, there is only one

training instance affected by the change. By examining the class label of the affected training instance, the new class distribution is obtained. For example, as τ increases to \$65,000, there is only one borrower in the training set, with an annual income of \$60,000, affected by this change. Since the class label for the borrower is `No`, the count for class `No` increases from 0 to 1 (for Annual Income≤\$65,000) and decreases from 7 to 6 (for Annual Income>\$65,000), as shown in **Figure 3.14**. The distribution for the `Yes` class remains unaffected. The updated Gini index for this candidate split position is 0.400.

This procedure is repeated until the Gini index for all candidates are found. The best split position corresponds to the one that produces the lowest Gini index, which occurs at τ=\$97,500. Since the Gini index at each candidate split position can be computed in $O(1)$ time, the complexity of finding the best split position is $O(N)$ once all the values are kept sorted, a one-time operation that takes $O(N \log N)$ time. The overall complexity of this method is thus $O(N \log N)$, which is much smaller than the O(N2) time taken by the brute-force method. The amount of computation can be further reduced by considering only candidate split positions located between two adjacent sorted instances with different class labels. For example, we do not need to consider candidate split positions located between \$60,000 and \$75,000 because all three instances with annual income in this range (\$60,000, \$70,000, and \$75,000) have the same class labels. Choosing a split position within this range only increases the degree of impurity, compared to a split position located outside this range. Therefore, the candidate split positions at τ=\$65,000 and τ=\$72,500 can be ignored. Similarly, we do not need to consider the candidate split positions at \$87,500, \$92,500, \$110,000, \$122,500, and \$172,500 because they are located between two adjacent instances with the same labels. This strategy reduces the number of candidate split positions to consider from 9 to 2 (excluding the two boundary cases τ=\$55,000 and τ=\$230,000).

Gain Ratio

One potential limitation of impurity measures such as entropy and Gini index is that they tend to favor qualitative attributes with large number of distinct values. **Figure 3.12** shows three candidate attributes for partitioning the data set given in **Table 3.3**. As previously mentioned, the attribute `Marital Status` is a better choice than the attribute `Home Owner`, because it provides a larger information gain. However, if we compare them against `Customer ID`, the latter produces the purest partitions with the maximum information gain, since the weighted entropy and Gini index is equal to zero for its children. Yet, `Customer ID` is not a good attribute for splitting because it has a unique value for each instance. Even though a test condition involving `Customer ID` will accurately classify every instance in the training data, we cannot use such a test condition on new test instances with `Customer ID` values that haven't been seen before during training. This example suggests having a low impurity value alone is insufficient to find a good attribute test condition for a node. As we will see later in **Section 3.4**, having more number of child nodes can make a decision tree more complex and consequently more susceptible to overfitting. Hence, the number of children produced by the splitting attribute should also be taken into consideration while deciding the best attribute test condition.

There are two ways to overcome this problem. One way is to generate only binary decision trees, thus avoiding the difficulty of handling attributes with varying number of partitions. This strategy is employed by decision tree classifiers such as CART. Another way is to modify the splitting criterion to take into account the number of partitions produced by the attribute. For example, in the C4.5 decision tree algorithm, a measure known as **gain ratio** is used to compensate for attributes that produce a large number of child nodes. This measure is computed as follows:

$$\text{Gain ratio} = \frac{\Delta_{\text{info}}}{\text{Split Info}} = \frac{\text{Entropy(Parent)} - \sum_{i=1}^{k} \frac{N(v_i)}{N}\text{Entropy}(v_i)}{-\sum_{i=1}^{k} \frac{N(v_i)}{N} \log_2 \frac{N(v_i)}{N}} \qquad (3.9)$$

where $N(v_i)$ is the number of instances assigned to node v_i and k is the total number of splits. The split information measures the entropy of splitting a node into its child nodes and evaluates if the split results in a larger number of equally-sized child nodes or not. For example, if every partition has the same number of instances, then $\forall i: N(v_i)/N = 1/k$ and the split information would be equal to $\log_2 k$. Thus, if an attribute produces a large number of splits, its split information is also large, which in turn, reduces the gain ratio.

3.4. Example Gain Ratio

Consider the data set given in Exercise 2 on page 185. We want to select the best attribute test condition among the following three attributes: `Gender`, `Car Type`, and `Customer ID`. The entropy before splitting is

$$\text{Entropy(parent)} = -\frac{10}{20}\log_2\frac{10}{20} - \frac{10}{20}\log_2\frac{10}{20} = 1.$$

If `Gender` is used as attribute test condition:

$$\text{Entropy(children)} = \frac{10}{20}\left[-\frac{6}{10}\log_2\frac{6}{10} - \frac{4}{10}\log_2\frac{4}{10}\right] \times 2 = 0.971$$
$$\text{Gain Ratio} = \frac{1 - 0.971}{-\frac{10}{20}\log_2\frac{10}{20} - \frac{10}{20}\log_2\frac{10}{20}} = \frac{0.029}{1} = 0.02$$

If `Car Type` is used as attribute test condition:

$$\text{Entropy(children)} = \frac{4}{20}\left[-\frac{1}{4}\log_2\frac{1}{4} - \frac{3}{4}\log_2\frac{3}{4}\right] + \frac{8}{20}\times 0 + \frac{8}{20}\left[-\frac{1}{8}\log_2\frac{1}{8} - \frac{7}{8}\log_2\frac{7}{8}\right] = 0.380$$
$$\text{Gain Ratio} = \frac{1 - 0.380}{-\frac{4}{20}\log_2\frac{4}{20} - \frac{8}{20}\log_2\frac{8}{20} - \frac{8}{20}\log_2\frac{8}{20}} = \frac{0.620}{1}$$

Finally, if `Customer ID` is used as attribute test condition:

Entropy(children)=120[−11log211−01log201
]×20=0Gain Ratio=1−0−120log2120×20=14.32=0.23

Thus, even though `Customer ID` has the highest information gain, its gain ratio is lower than `Car Type` since it produces a larger number of splits.

3.3.4 Algorithm for Decision Tree Induction

Algorithm 3.1 presents a pseudocode for decision tree induction algorithm. The input to this algorithm is a set of training instances *E* along with the attribute set *F* . The algorithm works by recursively selecting the best attribute to split the data (Step 7) and expanding the nodes of the tree (Steps 11 and 12) until the stopping criterion is met (Step 1). The details of this algorithm are explained below.

1. The `createNode()` function extends the decision tree by creating a new node. A node in the decision tree either has a test condition, denoted as *node.test cond*, or a class label, denoted as *node.label*.
2. The `find best split()` function determines the attribute test condition for partitioning the training instances associated with a node. The splitting attribute chosen depends on the impurity measure used. The popular measures include entropy and the Gini index.
3. The `Classify()` function determines the class label to be assigned to a leaf node. For each leaf node *t*, let p(i|t) denote the fraction of training instances from class *i* associated with the node *t*. The label assigned to

the leaf node is typically the one that occurs most frequently in the training instances that are associated with this node.

Algorithm 3.1 A skeleton decision tree induction algorithm.

```
TreeGrowth (E, F)
  1: if stopping cond(E,F ) = true then
  2:    leaf = createNode().
  3:    leaf.label = Classify(E).
  4:    return leaf.
  5: else
  6:    root = createNode().
  7:    root.test cond = find best split(E, F ).
  8:    let V = {v|v is a possible outcome of root.test cond }.
  9:    for each v ∈ V do
 10:     E_v = {e | root.test cond(e) = v and e ∈ E}.
 11:     child = TreeGrowth(Ev, F ).
 12:     add child as descendent of root and label the edge
(root → child) as v.
 13:    end for
 14: end if
 15: return root.
```

leaf.label=argmaxi p(i|t), (3.10)

where the argmax operator returns the class *i* that maximizes p(i|t). Besides providing the information needed to determine the class label

of a leaf node, p(i|t) can also be used as a rough estimate of the probability that an instance assigned to the leaf node *t* belongs to class *i*. **Sections 4.11.2** and **4.11.4** in the next chapter describe how such probability estimates can be used to determine the performance of a decision tree under different cost functions.

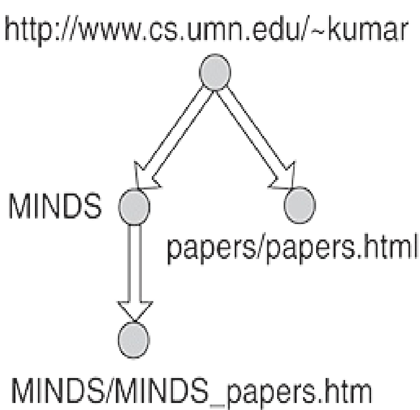
4. The `stopping cond()` function is used to terminate the tree-growing process by checking whether all the instances have identical class label or attribute values. Since decision tree classifiers employ a top-down, recursive partitioning approach for building a model, the number of training instances associated with a node decreases as the depth of the tree increases. As a result, a leaf node may contain too few training instances to make a statistically significant decision about its class label. This is known as the **data fragmentation** problem. One way to avoid this problem is to disallow splitting of a node when the number of instances associated with the node fall below a certain threshold. A more systematic way to control the size of a decision tree (number of leaf nodes) will be discussed in **Section 3.5.4**.

3.3.5 Example Application: Web Robot Detection

Consider the task of distinguishing the access patterns of web robots from those generated by human users. A web robot (also known as a web crawler) is a software program that automatically retrieves files from one or more websites by following the hyperlinks extracted from an initial set of seed URLs. These programs have been deployed for various purposes, from gathering web pages on behalf of search engines to more malicious activities such as spamming and committing click frauds in online advertisements.

Session	IP Address	Timestamp	Request Method	Requested Web Page	Protocol	Status	Number of Bytes	Referrer	User Agent
1	160.11.11.11	08/Aug/2004 10:15:21	GET	http://www.cs.umn.edu/~kumar	HTTP/1.1	200	6424		Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
1	160.11.11.11	08/Aug/2004 10:15:34	GET	http://www.cs.umn.edu/~kumar/MINDS	HTTP/1.1	200	41378	http://www.cs.umn.edu/~kumar	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
1	160.11.11.11	08/Aug/2004 10:15:41	GET	http://www.cs.umn.edu/~kumar/MINDS/MINDS_papers.htm	HTTP/1.1	200	1018516	http://www.cs.umn.edu/~kumar/MINDS	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
1	160.11.11.11	08/Aug/2004 10:16:11	GET	http://www.cs.umn.edu/~kumar/papers/papers.html	HTTP/1.1	200	7463	http://www.cs.umn.edu/~kumar	Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
2	35.9.2.2	08/Aug/2004 10:16:15	GET	http://www.cs.umn.edu/~steinbac	HTTP/1.0	200	3149		Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7) Gecko/20040616

(a) Example of a Web server log.



(b) Graph of a Web session.

Attribute Name	Description
totalPages	Total number of pages retrieved in a Web session
ImagePages	Total number of image pages retrieved in a Web session
TotalTime	Total amount of time spent by Web site visitor
RepeatedAccess	The same page requested more than once in a Web session
ErrorRequest	Errors in requesting for Web pages
GET	Percentage of requests made using GET method
POST	Percentage of requests made using POST method
HEAD	Percentage of requests made using HEAD method
Breadth	Breadth of Web traversal
Depth	Depth of Web traversal
MultiIP	Session with multiple IP addresses
MultiAgent	Session with multiple user agents

(c) Derived attributes for Web robot detection.

Figure 3.15.

Input data for web robot detection.

The web robot detection problem can be cast as a binary classification task. The input data for the classification task is a web server log, a sample of which is shown in **Figure 3.15(a)**. Each line in the log file corresponds to a request made by a client (i.e., a human user or a web robot) to the web server. The fields recorded in the web log include the client's IP address, timestamp of the request, URL of the requested file, size of the file, and **user agent**, which is a field that contains identifying information about the client.

For human users, the user agent field specifies the type of web browser or mobile device used to fetch the files, whereas for web robots, it should technically contain the name of the crawler program. However, web robots may conceal their true identities by declaring their user agent fields to be identical to known browsers. Therefore, user agent is not a reliable field to detect web robots.

The first step toward building a classification model is to precisely define a data instance and associated attributes. A simple approach is to consider each log entry as a data instance and use the appropriate fields in the log file as its attribute set. This approach, however, is inadequate for several reasons. First, many of the attributes are nominal-valued and have a wide range of domain values. For example, the number of unique client IP addresses, URLs, and referrers in a log file can be very large. These attributes are undesirable for building a decision tree because their split information is extremely high (see **Equation (3.9)**). In addition, it might not be possible to classify test instances containing IP addresses, URLs, or referrers that are not present in the training data. Finally, by considering each log entry as a separate data instance, we disregard the sequence of web pages retrieved by the client—a critical piece of information that can help distinguish web robot accesses from those of a human user.

A better alternative is to consider each web session as a data instance. A web session is a sequence of requests made by a client during a given visit to the website. Each web session can be modeled as a directed graph, in which the nodes correspond to web pages and the edges correspond to hyperlinks connecting one web page to another. **Figure 3.15(b)** shows a graphical representation of the first web session given in the log file. Every web session can be characterized using some meaningful attributes about the graph that contain discriminatory information. **Figure 3.15(c)** shows some of the attributes extracted from the graph, including the depth and breadth of its

corresponding tree rooted at the entry point to the website. For example, the depth and breadth of the tree shown in **Figure 3.15(b)** are both equal to two.

The derived attributes shown in **Figure 3.15(c)** are more informative than the original attributes given in the log file because they characterize the behavior of the client at the website. Using this approach, a data set containing 2916 instances was created, with equal numbers of sessions due to web robots (class 1) and human users (class 0). 10% of the data were reserved for training while the remaining 90% were used for testing. The induced decision tree is shown in **Figure 3.16**, which has an error rate equal to 3.8% on the training set and 5.3% on the test set. In addition to its low error rate, the tree also reveals some interesting properties that can help discriminate web robots from human users:

1. Accesses by web robots tend to be broad but shallow, whereas accesses by human users tend to be more focused (narrow but deep).
2. Web robots seldom retrieve the image pages associated with a web page.
3. Sessions due to web robots tend to be long and contain a large number of requested pages.
4. Web robots are more likely to make repeated requests for the same web page than human users since the web pages retrieved by human users are often cached by the browser.

3.3.6 Characteristics of Decision Tree Classifiers

The following is a summary of the important characteristics of decision tree induction algorithms.

1. **Applicability:** Decision trees are a nonparametric approach for building classification models. This approach does not require any prior assumption about the probability distribution governing the class and attributes of the data, and thus, is applicable to a wide variety of data sets. It is also applicable to both categorical and continuous data without requiring the attributes to be transformed into a common representation via binarization, normalization, or standardization. Unlike some binary classifiers described in **Chapter 4**, it can also deal with multiclass problems without the need to decompose them into multiple binary classification tasks. Another appealing feature of decision tree classifiers is that the induced trees, especially the shorter ones, are relatively easy to interpret. The accuracies of the trees are also quite comparable to other classification techniques for many simple data sets.
2. **Expressiveness:** A decision tree provides a universal representation for discrete-valued functions. In other words, it can encode any function of discrete-valued attributes. This is because every discrete-valued function can be represented as an assignment table, where every unique combination of discrete attributes is assigned a class label. Since every combination of attributes can be represented as a leaf in the decision tree, we can always find a decision tree whose label assignments at the leaf nodes matches with the assignment table of the original function. Decision trees can also help in providing compact representations of functions when some of the unique combinations of attributes can be represented by the same leaf node. For example, **Figure 3.17** shows the assignment table of the Boolean function $(A\wedge B)\vee(C\wedge D)$ involving four binary attributes, resulting in a total of 24=16 possible assignments. The tree shown in **Figure 3.17** shows

a compressed encoding of this assignment table. Instead of requiring a fully-grown tree with 16 leaf nodes, it is possible to encode the function using a simpler tree with only 7 leaf nodes. Nevertheless, not all decision trees for discrete-valued attributes can be simplified. One notable example is the parity function, whose value is 1 when there is an even number of true values among its Boolean attributes, and 0 otherwise. Accurate modeling of such a function requires a full decision tree with 2d nodes, where *d* is the number of Boolean attributes (see Exercise 1 on page 185).

Decision Tree:

```
depth = 1:
| breadth> 7 :   class 1
| breadth<= 7:
| | breadth <= 3:
| | | ImagePages> 0.375:   class 0
| | | ImagePages<= 0.375:
| | | | totalPages<= 6:   class 1
| | | | totalPages> 6:
| | | | | breadth <= 1:   class 1
| | | | | breadth > 1:   class 0
| | width > 3:
| | | MultiIP = 0:
| | | | ImagePages<= 0.1333:   class 1
| | | | ImagePages> 0.1333:
| | | | breadth <= 6:   class 0
| | | | breadth > 6:   class 1
| | | MultiIP = 1:
| | | | TotalTime <= 361:   class 0
| | | | TotalTime > 361:   class 1
depth> 1:
| MultiAgent = 0:
| | depth > 2:   class 0
| | depth < 2:
| | | MultiIP = 1:   class 0
| | | MultiIP = 0:
| | | | breadth <= 6:   class 0
| | | | breadth > 6:
| | | | | RepeatedAccess <= 0.322:   class 0
| | | | | RepeatedAccess > 0.322:   class 1
| MultiAgent = 1:
| | totalPages <= 81:   class 0
| | totalPages > 81:   class 1
```

Figure 3.16.

Decision tree model for web robot detection.

A	B	C	D	class
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

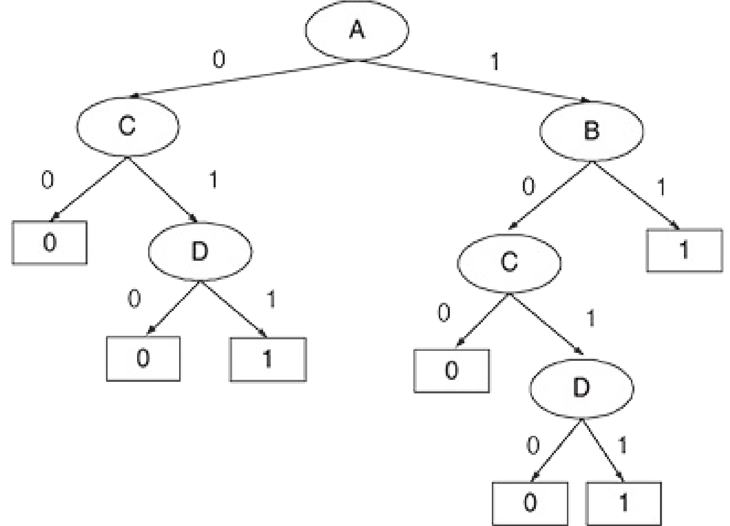


Figure 3.17.

Decision tree for the Boolean function (A∧B)∨(C∧D).

3. **Computational Efficiency:** Since the number of possible decision trees can be very large, many decision tree algorithms employ a heuristic-based approach to guide their search in the vast hypothesis space. For example, the algorithm presented in **Section 3.3.4** uses a greedy, top-down, recursive partitioning strategy for growing a decision tree. For many data sets, such techniques quickly construct a reasonably good decision tree even when the training set size is very large. Furthermore, once a decision tree has been built, classifying a test record is extremely fast, with a worst-case complexity of $O(w)$, where w is the maximum depth of the tree.
4. **Handling Missing Values:** A decision tree classifier can handle missing attribute values in a number of ways, both in the training and the test sets. When there are missing values in the test set, the classifier must decide which branch to follow if the value of a splitting

node attribute is missing for a given test instance. One approach, known as the **probabilistic split method**, which is employed by the C4.5 decision tree classifier, distributes the data instance to every child of the splitting node according to the probability that the missing attribute has a particular value. In contrast, the CART algorithm uses the **surrogate split method**, where the instance whose splitting attribute value is missing is assigned to one of the child nodes based on the value of another non-missing surrogate attribute whose splits most resemble the partitions made by the missing attribute. Another approach, known as the **separate class method** is used by the CHAID algorithm, where the missing value is treated as a separate categorical value distinct from other values of the splitting attribute. Figure 3.18 shows an example of the three different ways for handling missing values in a decision tree classifier. Other strategies for dealing with missing values are based on data preprocessing, where the instance with missing value is either imputed with the mode (for categorical attribute) or mean (for continuous attribute) value or discarded before the classifier is trained.

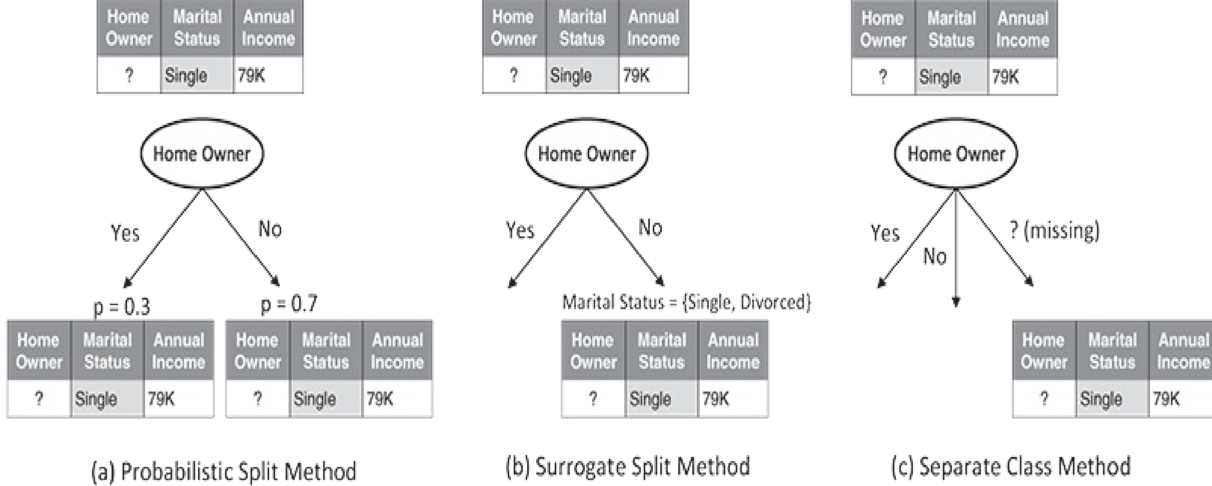


Figure 3.18.

Methods for handling missing attribute values in decision tree classifier.

During training, if an attribute *v* has missing values in some of the training instances associated with a node, we need a way to measure the gain in purity if *v* is used for splitting. One simple way is to exclude instances with missing values of *v* in the counting of instances associated with every child node, generated for every possible outcome of *v*.Further, if *v* is chosen as the attribute test condition at a node, training instances with missing values of *v* can be propagated to the child nodes using any of the methods described above for handling missing values in test instances.

5. **Handling Interactions among Attributes:** Attributes are considered interacting if they are able to distinguish between classes when used together, but individually they provide little or no information. Due to the greedy nature of the splitting criteria in decision trees, such attributes could be passed over in favor of other attributes that are not as useful. This could result in more complex decision trees than necessary. Hence, decision trees can perform poorly when there are interactions among attributes.
   To illustrate this point, consider the three-dimensional data shown in **Figure 3.19(a)**, which contains 2000 data points from one of two classes, denoted as + and ∘ in the diagram. **Figure 3.19(b)** shows the distribution of the two classes in the two-dimensional space involving attributes *X* and *Y* , which is a noisy version of the XOR Boolean function. We can see that even though the two classes are well-separated in this two-dimensional space, neither of the two attributes contain sufficient information to distinguish between the two classes when used alone. For example, the entropies of the following attribute test conditions: $X \le 10$ and $Y \le 10$, are close to 1, indicating that neither *X* nor *Y* provide any reduction in the impurity measure when used individually. *X* and *Y* thus represent a case of interaction among attributes. The data set also contains a third attribute, *Z*, in which both classes are distributed uniformly, as shown in **Figures 3.19(c)** and

3.19(d), and hence, the entropy of any split involving Z is close to 1. As a result, Z is as likely to be chosen for splitting as the interacting but useful attributes, X and Y. For further illustration of this issue, readers are referred to **Example 3.7** in **Section 3.4.1** and Exercise 7 at the end of this chapter.

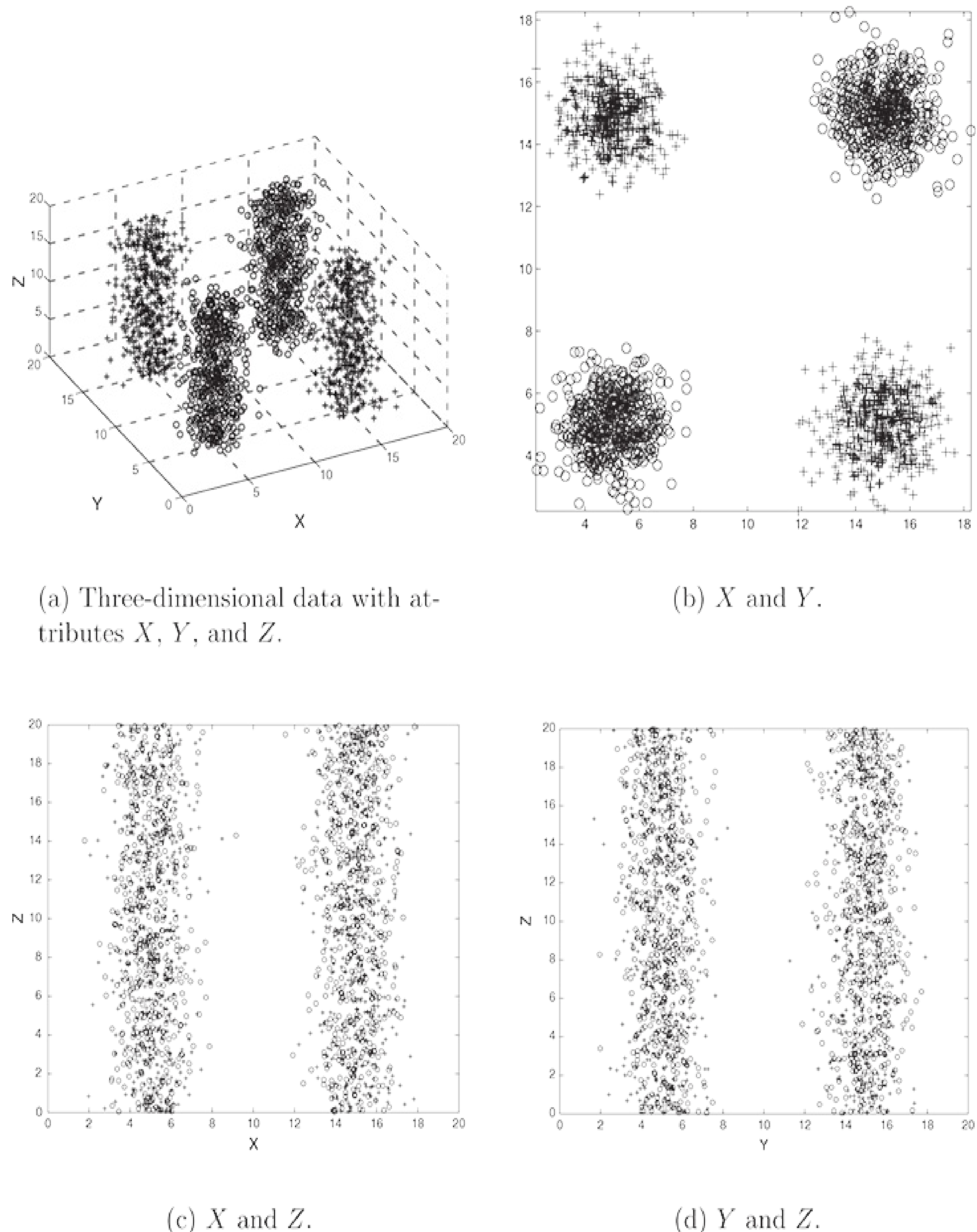


Figure 3.19.

Example of a XOR data involving X and Y, along with an irrelevant attribute Z.

6. **Handling Irrelevant Attributes:** An attribute is irrelevant if it is not useful for the classification task. Since irrelevant attributes are poorly associated with the target class labels, they will provide little or no gain in purity and thus will be passed over by other more relevant features. Hence, the presence of a small number of irrelevant attributes will not impact the decision tree construction process. However, not all attributes that provide little to no gain are irrelevant (see **Figure 3.19**). Hence, if the classification problem is complex (e.g., involving interactions among attributes) and there are a large number of irrelevant attributes, then some of these attributes may be accidentally chosen during the tree-growing process, since they may provide a better gain than a relevant attribute just by random chance. Feature selection techniques can help to improve the accuracy of decision trees by eliminating the irrelevant attributes during preprocessing. We will investigate the issue of too many irrelevant attributes in **Section 3.4.1**.
7. **Handling Redundant Attributes:** An attribute is redundant if it is strongly correlated with another attribute in the data. Since redundant attributes show similar gains in purity if they are selected for splitting, only one of them will be selected as an attribute test condition in the decision tree algorithm. Decision trees can thus handle the presence of redundant attributes.
8. **Using Rectilinear Splits:** The test conditions described so far in this chapter involve using only a single attribute at a time. As a consequence, the tree-growing procedure can be viewed as the process of partitioning the attribute space into disjoint regions until each region contains records of the same class. The border between two neighboring regions of different classes is known as a **decision boundary**. **Figure 3.20** shows the decision tree as well as the decision boundary for a binary classification problem. Since the test condition involves only a single attribute, the decision boundaries are

rectilinear; i.e., parallel to the coordinate axes. This limits the expressiveness of decision trees in representing decision boundaries of data sets with continuous attributes. **Figure 3.21** shows a two-dimensional data set involving binary classes that cannot be perfectly classified by a decision tree whose attribute test conditions are defined based on single attributes. The binary classes in the data set are generated from two skewed Gaussian distributions, centered at (8,8) and (12,12), respectively. The true decision boundary is represented by the diagonal dashed line, whereas the rectilinear decision boundary produced by the decision tree classifier is shown by the thick solid line. In contrast, an **oblique decision tree** may overcome this limitation by allowing the test condition to be specified using more than one attribute. For example, the binary classification data shown in **Figure 3.21** can be easily represented by an oblique decision tree with a single root node with test condition

$x+y<20.$

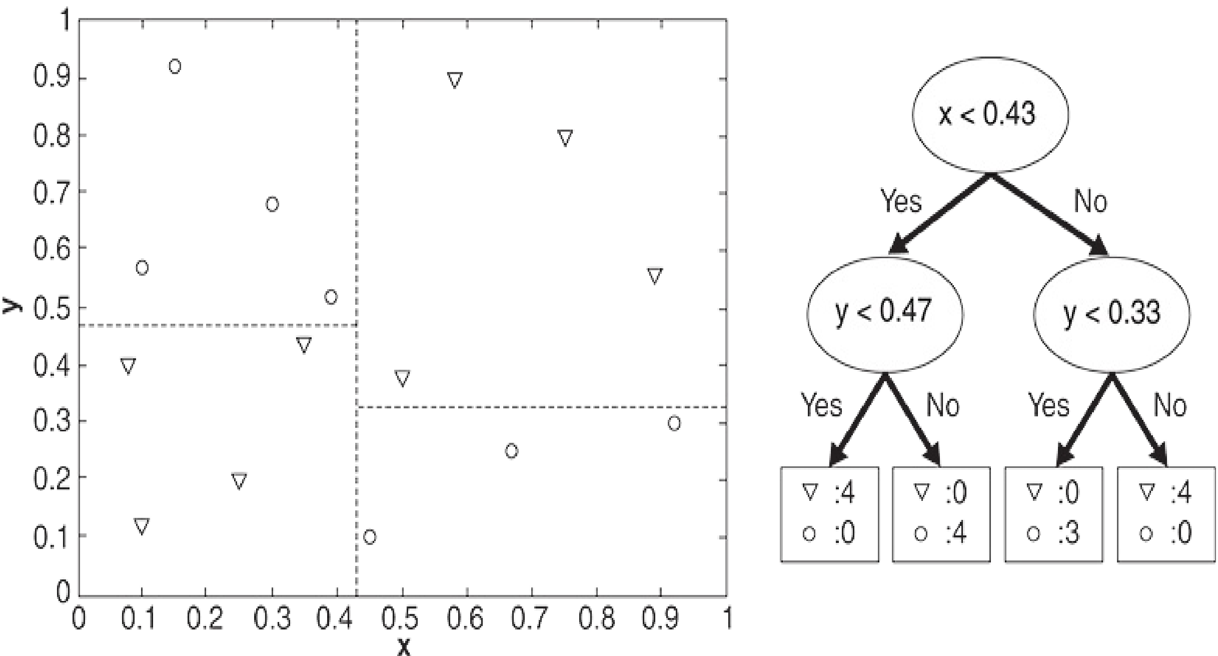


Figure 3.20.

Example of a decision tree and its decision boundaries for a two-dimensional data set.

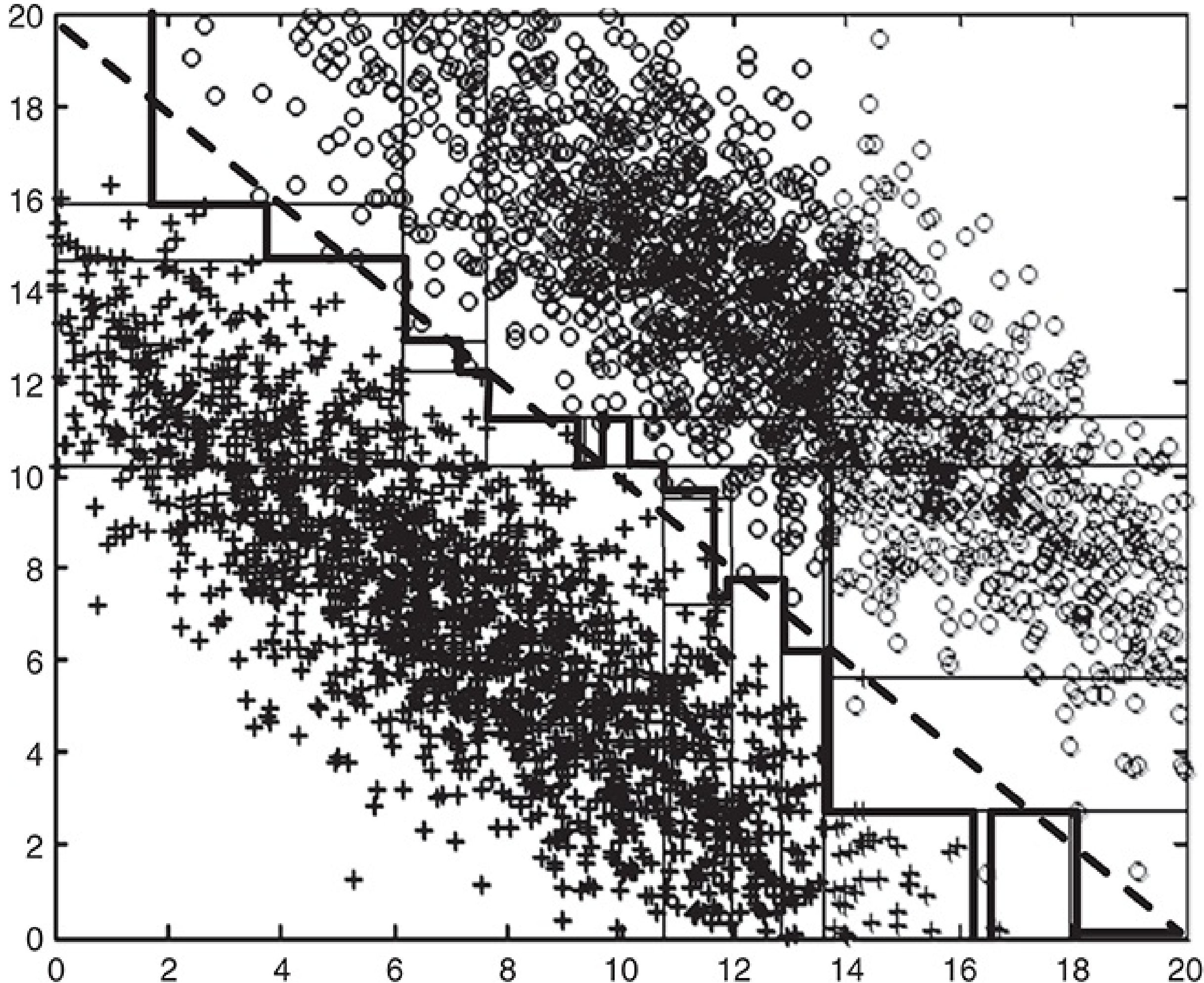


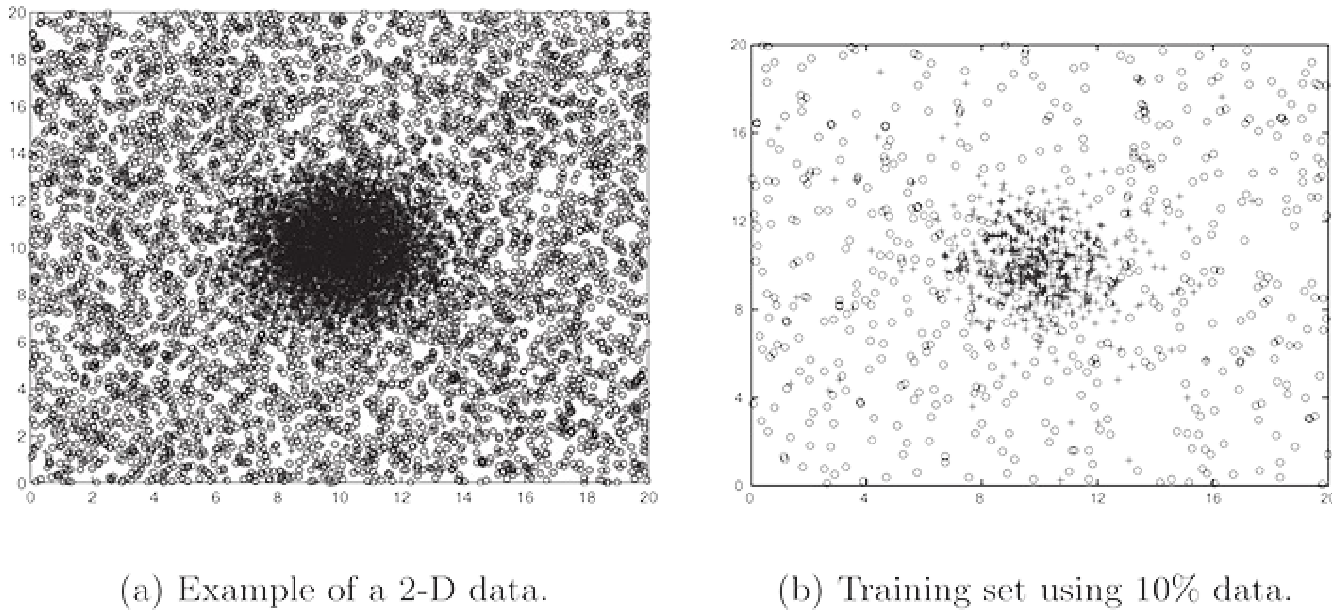
Figure 3.21.
Example of data set that cannot be partitioned optimally using a decision tree with single attribute test conditions. The true decision boundary is shown by the dashed line.

Although an oblique decision tree is more expressive and can produce more compact trees, finding the optimal test condition is computationally more expensive.

9. **Choice of Impurity Measure:** It should be noted that the choice of impurity measure often has little effect on the performance of decision tree classifiers since many of the impurity measures are quite consistent with each other, as shown in **Figure 3.11** on **page 129**. Instead, the strategy used to prune the tree has a greater impact on the final tree than the choice of impurity measure.

3.4 Model Overfitting

Methods presented so far try to learn classification models that show the lowest error on the training set. However, as we will show in the following example, even if a model fits well over the training data, it can still show poor generalization performance, a phenomenon known as model overfitting.

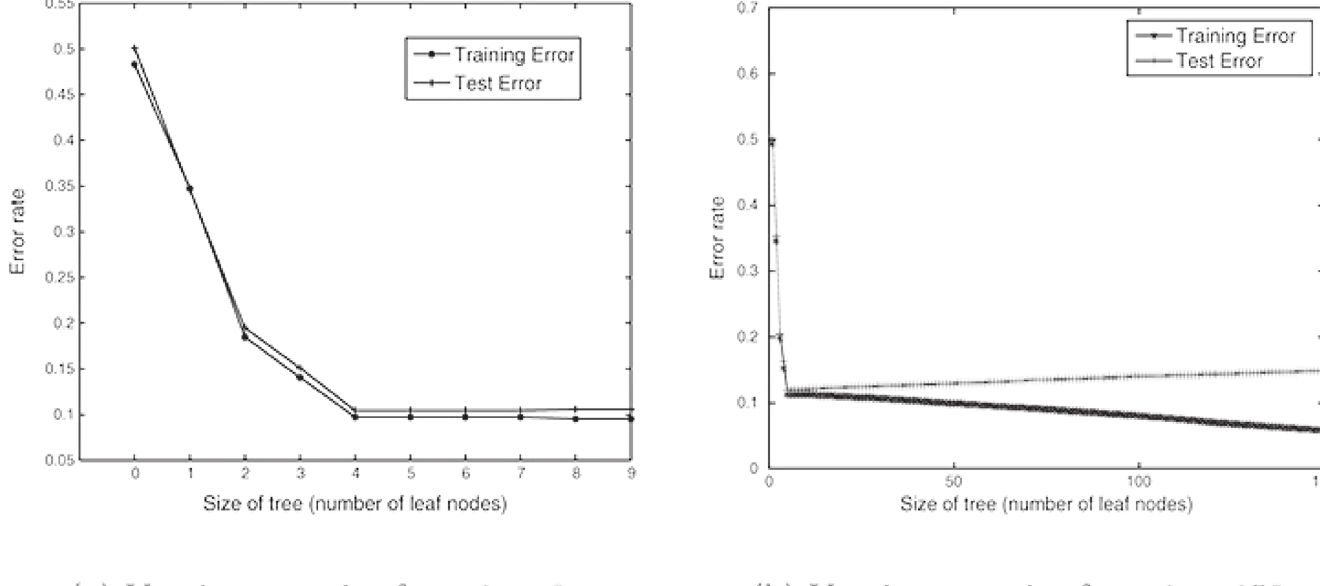


(a) Example of a 2-D data.

(b) Training set using 10% data.

Figure 3.22.

Examples of training and test sets of a two-dimensional classification problem.



(a) Varying tree size from 1 to 8. (b) Varying tree size from 1 to 150.

Figure 3.23.

Effect of varying tree size (number of leaf nodes) on training and test errors.

3.5. Example Overfitting and Underfitting of Decision Trees

Consider the two-dimensional data set shown in **Figure 3.22(a)**. The data set contains instances that belong to two separate classes, represented as + and ∘, respectively, where each class has 5400 instances. All instances belonging to the ∘ class were generated from a uniform distribution. For the + class, 5000 instances were generated from a Gaussian distribution centered at (10,10) with unit variance, while the remaining 400 instances were sampled from the same uniform distribution as the ∘ class. We can see from **Figure 3.22(a)** that the + class can be largely distinguished from the ∘ class by drawing a circle of appropriate size centered at (10,10). To learn a classifier using this two-dimensional data set, we randomly sampled 10% of the data for training and used the remaining 90% for testing. The training set, shown in **Figure 3.22(b)**, looks quite representative of the overall data. We used Gini index as the

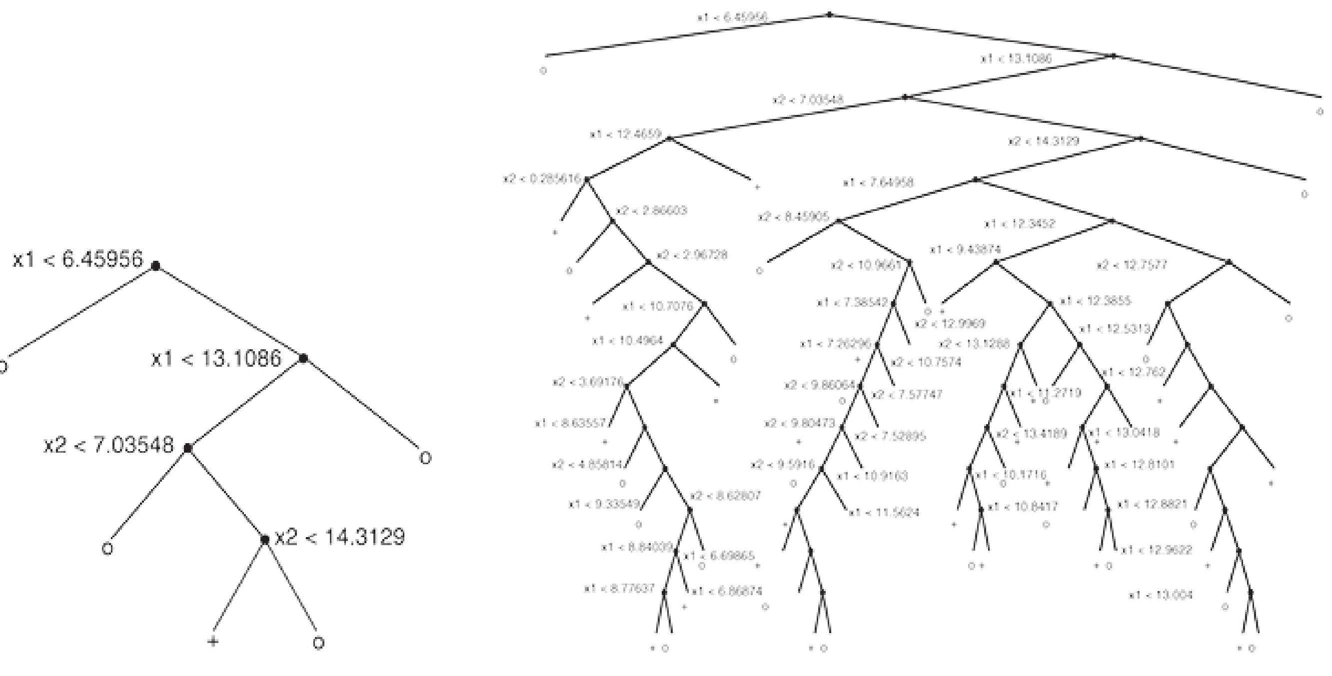
impurity measure to construct decision trees of increasing sizes (number of leaf nodes), by recursively expanding a node into child nodes till every leaf node was pure, as described in **Section 3.3.4**.

Figure 3.23(a) shows changes in the training and test error rates as the size of the tree varies from 1 to 8. Both error rates are initially large when the tree has only one or two leaf nodes. This situation is known as **model underfitting**. Underfitting occurs when the learned decision tree is too simplistic, and thus, incapable of fully representing the true relationship between the attributes and the class labels. As we increase the tree size from 1 to 8, we can observe two effects. First, both the error rates decrease since larger trees are able to represent more complex decision boundaries. Second, the training and test error rates are quite close to each other, which indicates that the performance on the training set is fairly representative of the generalization performance. As we further increase the size of the tree from 8 to 150, the training error continues to steadily decrease till it eventually reaches zero, as shown in **Figure 3.23(b)**. However, in a striking contrast, the test error rate ceases to decrease any further beyond a certain tree size, and then it begins to increase. The training error rate thus grossly under-estimates the test error rate once the tree becomes too large. Further, the gap between the training and test error rates keeps on widening as we increase the tree size. This behavior, which may seem counter-intuitive at first, can be attributed to the phenomena of **model overfitting**.

3.4.1 Reasons for Model Overfitting

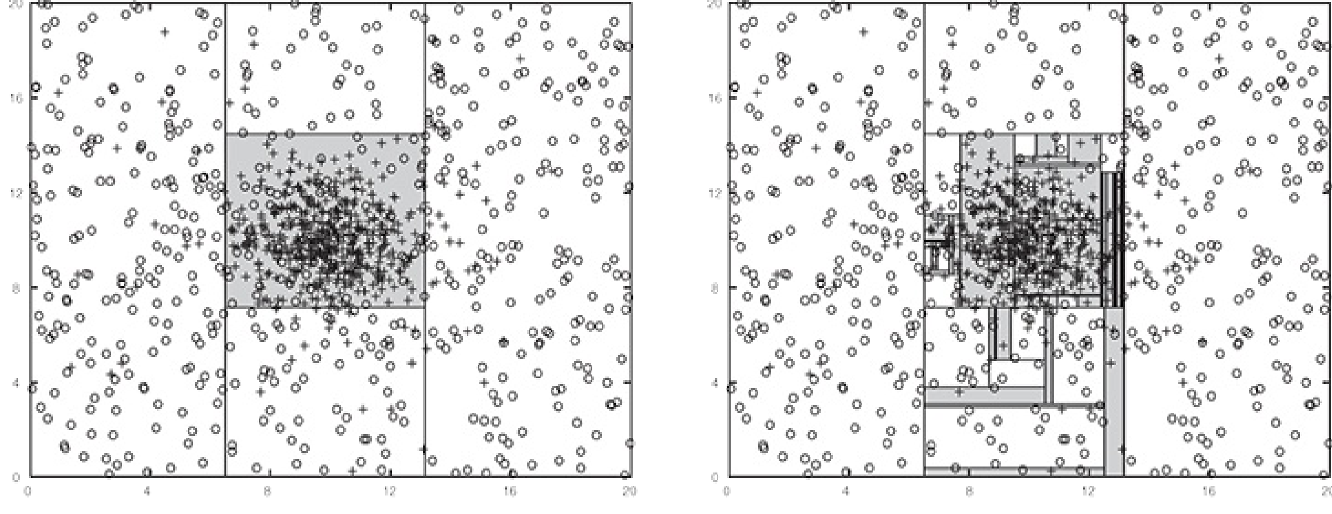
Model overfitting is the phenomena where, in the pursuit of minimizing the training error rate, an overly complex model is selected that captures specific

patterns in the training data but fails to learn the *true* nature of relationships between attributes and class labels in the overall data. To illustrate this, **Figure 3.24** shows decision trees and their corresponding decision boundaries (shaded rectangles represent regions assigned to the + class) for two trees of sizes 5 and 50. We can see that the decision tree of size 5 appears quite simple and its decision boundaries provide a reasonable approximation to the ideal decision boundary, which in this case corresponds to a circle centered around the Gaussian distribution at (10, 10). Although its training and test error rates are non-zero, they are very close to each other, which indicates that the patterns learned in the training set should generalize well over the test set. On the other hand, the decision tree of size 50 appears much more complex than the tree of size 5, with complicated decision boundaries. For example, some of its shaded rectangles (assigned the + class) attempt to cover narrow regions in the input space that contain only one or two + training instances. Note that the prevalence of + instances in such regions is highly specific to the training set, as these regions are mostly dominated by - instances in the overall data. Hence, in an attempt to perfectly fit the training data, the decision tree of size 50 starts fine tuning itself to specific patterns in the training data, leading to poor performance on an independently chosen test set.



(a) Decision tree with 5 leaf nodes.

(b) Decision tree with 50 leaf nodes.

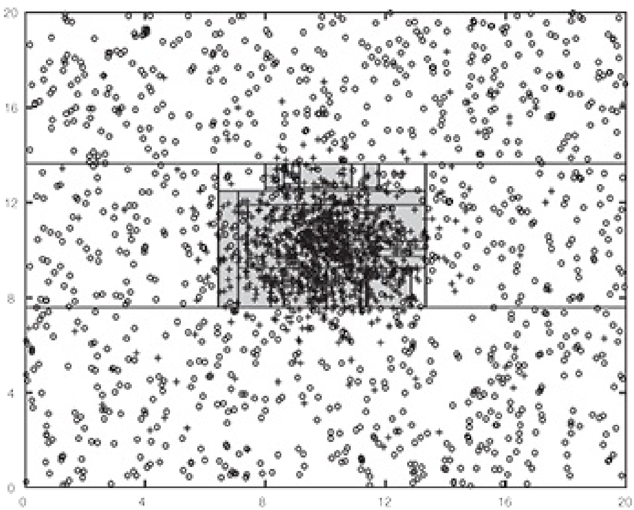


(c) Decision boundary for tree with 5 leaf nodes.

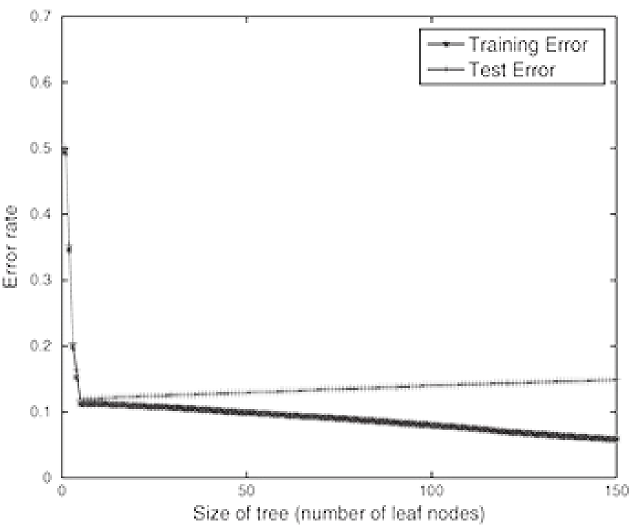
(d) Decision boundary for tree with 50 leaf nodes.

Figure 3.24.

Decision trees with different model complexities.



(a) Decision boundary for tree with 50 leaf nodes using 20% data for training.



(b) Training and test error rates using 20% data for training.

Figure 3.25.

Performance of decision trees using 20% data for training (twice the original training size).

There are a number of factors that influence model overfitting. In the following, we provide brief descriptions of two of the major factors: limited training size and high model complexity. Though they are not exhaustive, the interplay between them can help explain most of the common model overfitting phenomena in real-world applications.

Limited Training Size

Note that a training set consisting of a finite number of instances can only provide a limited representation of the overall data. Hence, it is possible that the patterns learned from a training set do not fully represent the true patterns in the overall data, leading to model overfitting. In general, as we increase the size of a training set (number of training instances), the patterns learned from the training set start resembling the true patterns in the overall data. Hence,

the effect of overfitting can be reduced by increasing the training size, as illustrated in the following example.

3.6 Example Effect of Training Size

Suppose that we use twice the number of training instances than what we had used in the experiments conducted in **Example 3.5**. Specifically, we use 20% data for training and use the remainder for testing. **Figure 3.25(b)** shows the training and test error rates as the size of the tree is varied from 1 to 150. There are two major differences in the trends shown in this figure and those shown in **Figure 3.23(b)** (using only 10% of the data for training). First, even though the training error rate decreases with increasing tree size in both figures, its rate of decrease is much smaller when we use twice the training size. Second, for a given tree size, the gap between the training and test error rates is much smaller when we use twice the training size. These differences suggest that the patterns learned using 20% of data for training are more generalizable than those learned using 10% of data for training.

Figure 3.25(a) shows the decision boundaries for the tree of size 50, learned using 20% of data for training. In contrast to the tree of the same size learned using 10% data for training (see **Figure 3.24(d)**), we can see that the decision tree is not capturing specific patterns of noisy + instances in the training set. Instead, the high model complexity of 50 leaf nodes is being effectively used to learn the boundaries of the + instances centered at (10, 10).

High Model Complexity

Generally, a more complex model has a better ability to represent complex patterns in the data. For example, decision trees with larger number of leaf

nodes can represent more complex decision boundaries than decision trees with fewer leaf nodes. However, an overly complex model also has a tendency to learn specific patterns in the training set that do not generalize well over unseen instances. Models with high complexity should thus be judiciously used to avoid overfitting.

One measure of model complexity is the number of “parameters” that need to be inferred from the training set. For example, in the case of decision tree induction, the attribute test conditions at internal nodes correspond to the parameters of the model that need to be inferred from the training set. A decision tree with larger number of attribute test conditions (and consequently more leaf nodes) thus involves more “parameters” and hence is more complex.

Given a class of models with a certain number of parameters, a learning algorithm attempts to select the best combination of parameter values that maximizes an evaluation metric (e.g., accuracy) over the training set. If the number of parameter value combinations (and hence the complexity) is large, the learning algorithm has to select the best combination from a large number of possibilities, using a limited training set. In such cases, there is a high chance for the learning algorithm to pick a *spurious* combination of parameters that maximizes the evaluation metric just by random chance. This is similar to the **multiple comparisons problem** (also referred as multiple testing problem) in statistics.

As an illustration of the multiple comparisons problem, consider the task of predicting whether the stock market will rise or fall in the next ten trading days. If a stock analyst simply makes random guesses, the probability that her prediction is correct on any trading day is 0.5. However, the probability that she will predict correctly at least nine out of ten times is

$$\frac{\binom{10}{9}+\binom{10}{10}}{2^{10}}=0.0107,$$

which is extremely low.

Suppose we are interested in choosing an investment advisor from a pool of 200 stock analysts. Our strategy is to select the analyst who makes the most number of correct predictions in the next ten trading days. The flaw in this strategy is that even if all the analysts make their predictions in a random fashion, the probability that at least one of them makes at least nine correct predictions is

$$1-(1-0.0107)^{200}=0.8847,$$

which is very high. Although each analyst has a low probability of predicting at least nine times correctly, considered together, we have a high probability of finding at least one analyst who can do so. However, there is no guarantee in the future that such an analyst will continue to make accurate predictions by random guessing.

How does the multiple comparisons problem relate to model overfitting? In the context of learning a classification model, each combination of parameter values corresponds to an analyst, while the number of training instances corresponds to the number of days. Analogous to the task of selecting the best analyst who makes the most accurate predictions on consecutive days, the task of a learning algorithm is to select the best combination of parameters that results in the highest accuracy on the training set. If the number of parameter combinations is large but the training size is small, it is highly likely for the learning algorithm to choose a spurious parameter combination that provides high training accuracy just by random chance. In the following example, we illustrate the phenomena of overfitting due to multiple comparisons in the context of decision tree induction.

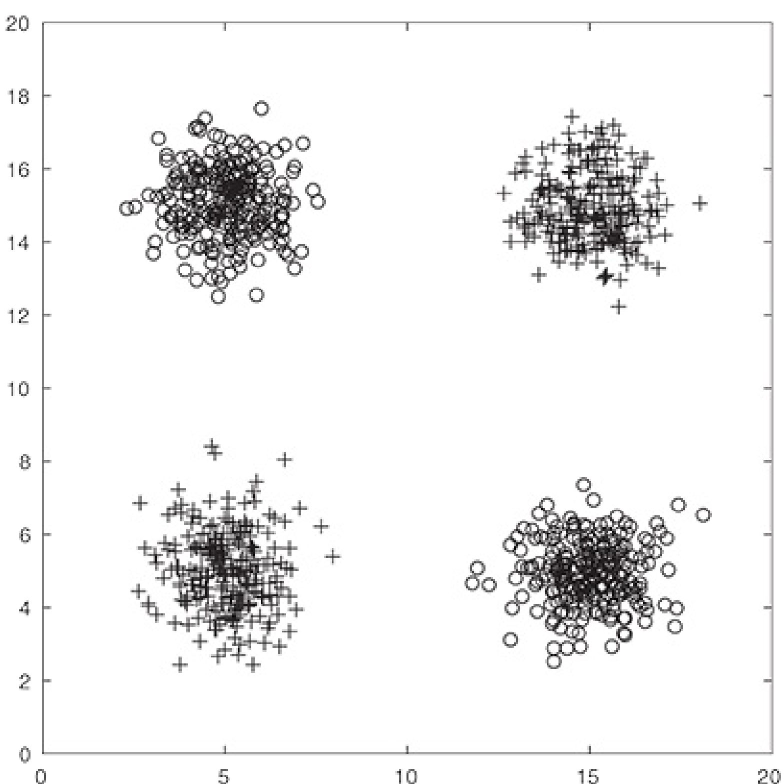
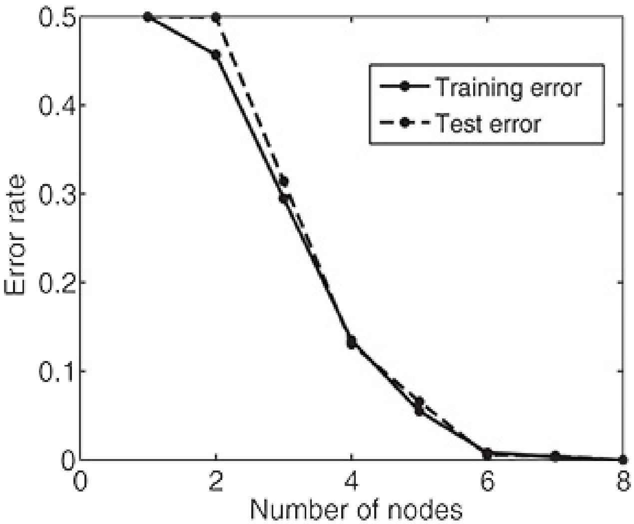
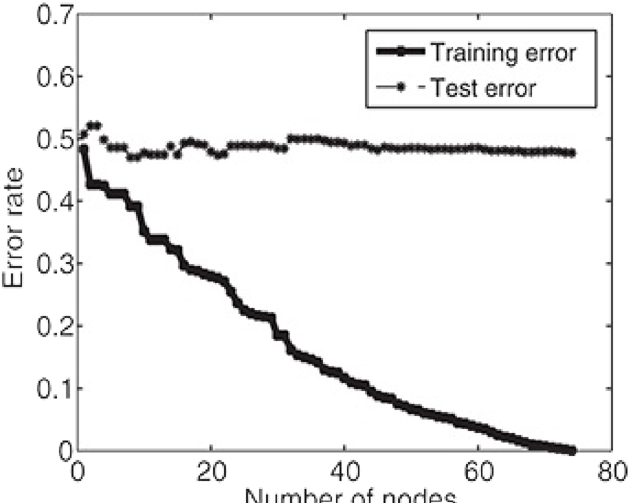


Figure 3.26.

Example of a two-dimensional (X-Y) data set.



(a) Using X and Y attributes only.



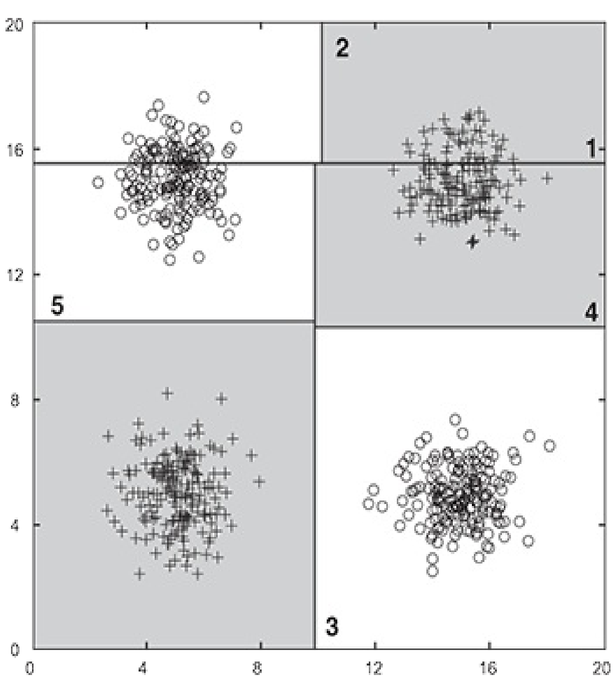
(b) After adding 100 irrelevant attributes.

Figure 3.27.

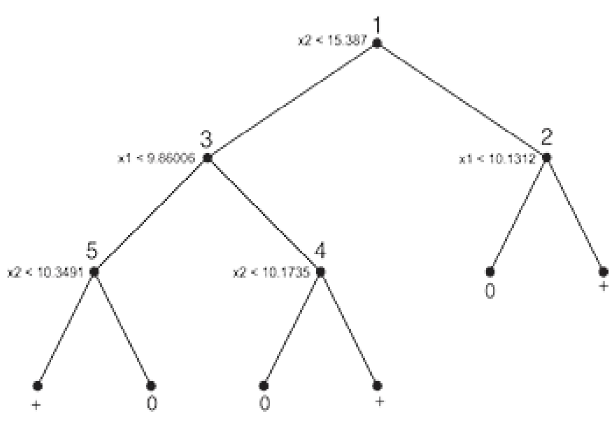
Training and test error rates illustrating the effect of multiple comparisons problem on model overfitting.

3.7. Example Multiple Comparisons and Overfitting

Consider the two-dimensional data set shown in **Figure 3.26** containing 500 + and 500 ∘ instances, which is similar to the data shown in **Figure 3.19**. In this data set, the distributions of both classes are well-separated in the two-dimensional (*XY*) attribute space, but none of the two attributes (*X* or *Y*) are sufficiently informative to be used alone for separating the two classes. Hence, splitting the data set based on any value of an X or Y attribute will provide close to zero reduction in an impurity measure. However, if X and Y attributes are used together in the splitting criterion (e.g., splitting X at 10 *and* Y at 10), the two classes can be effectively separated.



(a) Decision boundary for tree with 6 leaf nodes.



(b) Decision tree with 6 leaf nodes.

Figure 3.28.
Decision tree with 6 leaf nodes using X and Y as attributes. Splits have been numbered from 1 to 5 in order of other occurrence in the tree.

Figure 3.27(a) shows the training and test error rates for learning decision trees of varying sizes, when 30% of the data is used for training and the remainder of the data for testing. We can see that the two classes can be separated using a small number of leaf nodes. Figure 3.28 shows the decision boundaries for the tree with six leaf nodes, where the splits have been numbered according to their order of appearance in the tree. Note that the even though splits 1 and 3 provide trivial gains, their consequent splits (2, 4, and 5) provide large gains, resulting in effective discrimination of the two classes.

Assume we add 100 irrelevant attributes to the two-dimensional X-Y data. Learning a decision tree from this resultant data will be challenging because the number of candidate attributes to choose for splitting at every internal node will increase from two to 102. With such a large number of candidate attribute test conditions to choose from, it is quite likely that spurious attribute test conditions will be selected at internal nodes because of the multiple comparisons problem. Figure 3.27(b) shows the training and test error rates after adding 100 irrelevant attributes to the training set. We can see that the test error rate remains close to 0.5 even after using 50 leaf nodes, while the training error rate keeps on declining and eventually becomes 0.

3.5 Model Selection

There are many possible classification models with varying levels of model complexity that can be used to capture patterns in the training data. Among these possibilities, we want to select the model that shows lowest generalization error rate. The process of selecting a model with the right level of complexity, which is expected to generalize well over unseen test instances, is known as **model selection**. As described in the previous section, the training error rate cannot be reliably used as the sole criterion for model selection. In the following, we present three generic approaches to estimate the generalization performance of a model that can be used for model selection. We conclude this section by presenting specific strategies for using these approaches in the context of decision tree induction.

3.5.1 Using a Validation Set

Note that we can always estimate the generalization error rate of a model by using “out-of-sample” estimates, i.e. by evaluating the model on a separate **validation set** that is not used for training the model. The error rate on the validation set, termed as the validation error rate, is a better indicator of generalization performance than the training error rate, since the validation set has not been used for training the model. The validation error rate can be used for model selection as follows.

Given a training set *D.train*, we can partition *D.train* into two smaller subsets, *D.tr* and *D.val*, such that *D.tr* is used for training while *D.val* is used as the validation set. For example, two-thirds of *D.train* can be reserved as *D.tr* for

training, while the remaining one-third is used as *D.val* for computing validation error rate. For any choice of classification model *m* that is trained on *D.tr*, we can estimate its validation error rate on *D.val*, errval(m). The model that shows the lowest value of errval(m) can then be selected as the preferred choice of model.

The use of validation set provides a generic approach for model selection. However, one limitation of this approach is that it is sensitive to the sizes of *D.tr* and *D.val*, obtained by partitioning *D.train*. If the size of *D.tr* is too small, it may result in the learning of a poor classification model with sub-standard performance, since a smaller training set will be less representative of the overall data. On the other hand, if the size of *D.val* is too small, the validation error rate might not be reliable for selecting models, as it would be computed over a small number of instances.

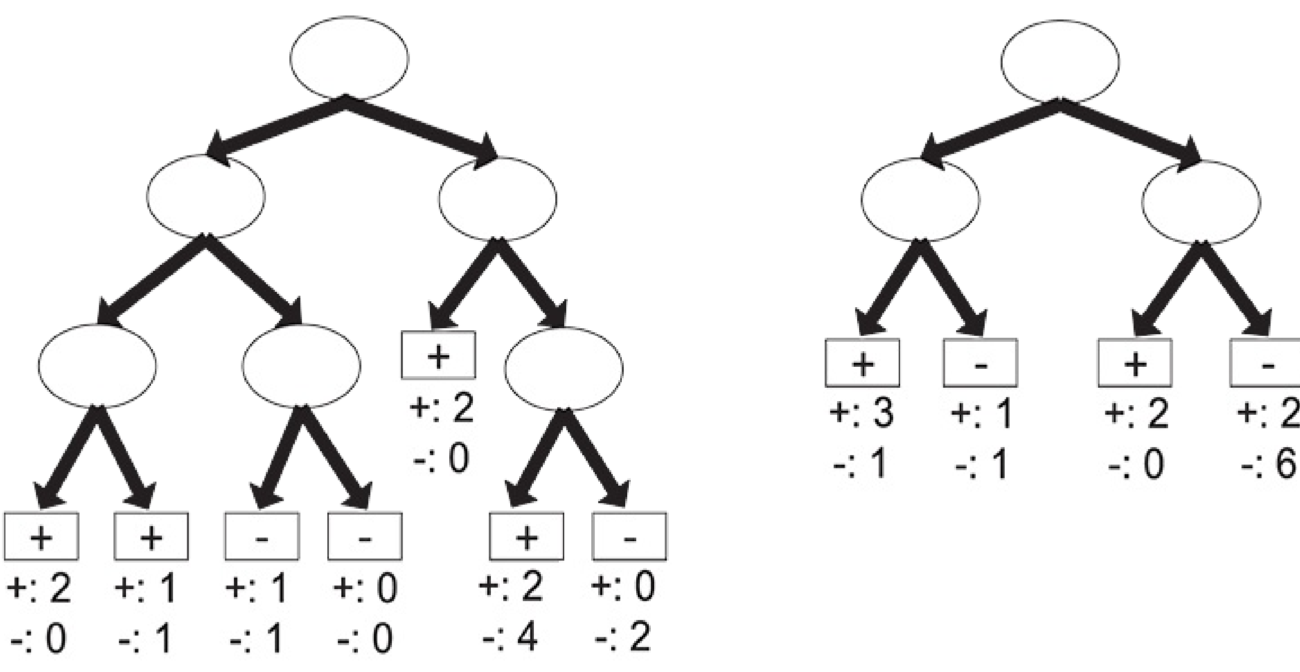


Figure 3.29.

Class distribution of validation data for the two decision trees shown in **Figure 3.30**.

3.8. Example Validation Error

In the following example, we illustrate one possible approach for using a validation set in decision tree induction. **Figure 3.29** shows the predicted labels at the leaf nodes of the decision trees generated in **Figure 3.30**. The counts given beneath the leaf nodes represent the proportion of data instances in the validation set that reach each of the nodes. Based on the predicted labels of the nodes, the validation error rate for the left tree is errval(TL)=6/16=0.375, while the validation error rate for the right tree is errval(TR)=4/16=0.25. Based on their validation error rates, the right tree is preferred over the left one.

3.5.2 Incorporating Model Complexity

Since the chance for model overfitting increases as the model becomes more complex, a model selection approach should not only consider the training error rate but also the model complexity. This strategy is inspired by a well-known principle known as **Occam's razor** or the **principle of parsimony**, which suggests that given two models with the same errors, the simpler model is preferred over the more complex model. A generic approach to account for model complexity while estimating generalization performance is formally described as follows.

Given a training set *D.train*, let us consider learning a classification model *m* that belongs to a certain class of models, M. For example, if M represents the set of all possible decision trees, then *m* can correspond to a specific decision

tree learned from the training set. We are interested in estimating the generalization error rate of *m*, *gen.error*(*m*). As discussed previously, the training error rate of *m*, *train.error*(*m, D.train*), can under-estimate *gen.error*(*m*) when the model complexity is high. Hence, we represent *gen.error*(*m*) as a function of not just the training error rate but also the model complexity of M, complexity(M), as follows:

$$\text{gen.error}(m) = \text{train.error}(m, \text{D.train}) + \alpha \times \text{complexity}(M), \tag{3.11}$$

where α is a hyper-parameter that strikes a balance between minimizing training error and reducing model complexity. A higher value of α gives more emphasis to the model complexity in the estimation of generalization performance. To choose the right value of α, we can make use of the validation set in a similar way as described in **3.5.1**. For example, we can iterate through a range of values of α and for every possible value, we can learn a model on a subset of the training set, *D.tr*, and compute its validation error rate on a separate subset, *D.val*. We can then select the value of α that provides the lowest validation error rate.

Equation 3.11 provides one possible approach for incorporating model complexity into the estimate of generalization performance. This approach is at the heart of a number of techniques for estimating generalization performance, such as the structural risk minimization principle, the Akaike's Information Criterion (AIC), and the Bayesian Information Criterion (BIC). The structural risk minimization principle serves as the building block for learning support vector machines, which will be discussed later in **Chapter 4**. For more details on AIC and BIC, see the Bibliographic Notes.

In the following, we present two different approaches for estimating the complexity of a model, complexity(M). While the former is specific to decision trees, the latter is more generic and can be used with any class of models.

Estimating the Complexity of Decision Trees

In the context of decision trees, the complexity of a decision tree can be estimated as the ratio of the number of leaf nodes to the number of training instances. Let *k* be the number of leaf nodes and Ntrain be the number of training instances. The complexity of a decision tree can then be described as k/Ntrain. This reflects the intuition that for a larger training size, we can learn a decision tree with larger number of leaf nodes without it becoming overly complex. The generalization error rate of a decision tree *T* can then be computed using **Equation 3.11** as follows:

$$errgen(T)=err(T)+\Omega\times kNtrain,$$

where *err*(*T*) is the training error of the decision tree and Ω is a hyper-parameter that makes a trade-off between reducing training error and minimizing model complexity, similar to the use of α in **Equation 3.11**. Ω can be viewed as the relative cost of adding a leaf node relative to incurring a training error. In the literature on decision tree induction, the above approach for estimating generalization error rate is also termed as the **pessimistic error estimate**. It is called pessimistic as it assumes the generalization error rate to be worse than the training error rate (by adding a penalty term for model complexity). On the other hand, simply using the training error rate as an estimate of the generalization error rate is called the **optimistic error estimate** or the **resubstitution estimate**.

3.9. Example Generalization Error Estimates

Consider the two binary decision trees, TL and TR, shown in **Figure 3.30**. Both trees are generated from the same training data and TL is generated by expanding three leaf nodes of TR. The counts shown in the leaf nodes of the trees represent the class distribution of the training

instances. If each leaf node is labeled according to the majority class of training instances that reach the node, the training error rate for the left tree will be err(TL)=4/24=0.167, while the training error rate for the right tree will be err(TR)=6/24=0.25. Based on their training error rates alone, TL would preferred over TR, even though TL is more complex (contains larger number of leaf nodes) than TR.

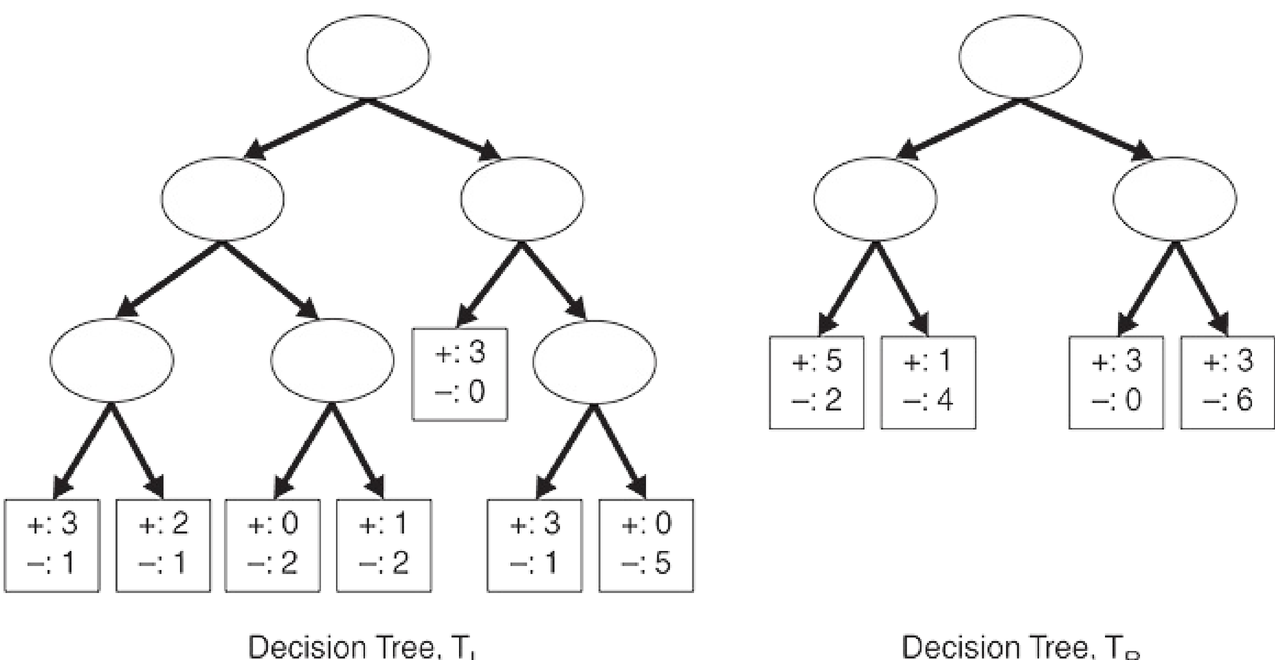


Figure 3.30.
Example of two decision trees generated from the same training data.

Now, assume that the cost associated with each leaf node is Ω=0.5. Then, the generalization error estimate for TL will be

errgen(TL)=424+0.5×724=7.524=0.3125

and the generalization error estimate for TR will be

errgen(TR) =624+0.5×424=824=0.3333.

Since TL has a lower generalization error rate, it will still be preferred over TR. Note that Ω=0.5 implies that a node should always be expanded into its two child nodes if it improves the prediction of at least one training instance, since expanding a node is less costly than misclassifying a training instance. On the other hand, if Ω=1, then the generalization error rate for TL is errgen(TL)=11/24=0.458 and for TR is errgen(TR)=10/24=0.417. In this case, TR will be preferred over TL because it has a lower generalization error rate. This example illustrates that different choices of Ω can change our preference of decision trees based on their generalization error estimates. However, for a given choice of Ω, the pessimistic error estimate provides an approach for modeling the generalization performance on unseen test instances. The value of Ω can be selected with the help of a validation set.

Minimum Description Length Principle

Another way to incorporate model complexity is based on an information-theoretic approach known as the minimum description length or MDL principle. To illustrate this approach, consider the example shown in **Figure 3.31**. In this example, both person `A` and person `B` are given a set of instances with known attribute values `x`. Assume person A knows the class label *y* for every instance, while person `B` has no such information. `A` would like to share the class information with `B` by sending a message containing the labels. The message would contain Θ(N) bits of information, where *N* is the number of instances.

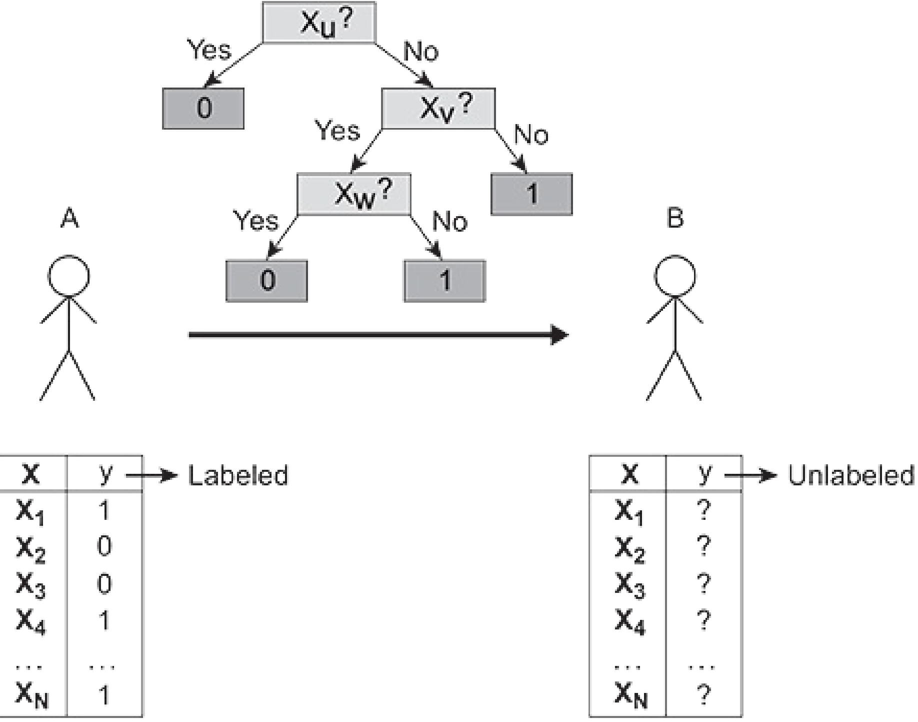


Figure 3.31.

An illustration of the minimum description length principle.

Alternatively, instead of sending the class labels explicitly, `A` can build a classification model from the instances and transmit it to `B`. `B` can then apply the model to determine the class labels of the instances. If the model is 100% accurate, then the cost for transmission is equal to the number of bits required to encode the model. Otherwise, `A` must also transmit information about which instances are misclassified by the model so that `B` can reproduce the same class labels. Thus, the overall transmission cost, which is equal to the total description length of the message, is

$$\text{Cost}(\text{model}, \text{data}) = \text{Cost}(\text{data}|\text{model}) + \alpha \times \text{Cost}(\text{model}), \tag{3.12}$$

where the first term on the right-hand side is the number of bits needed to encode the misclassified instances, while the second term is the number of bits required to encode the model. There is also a hyper-parameter α that trades-off the relative costs of the misclassified instances and the model.

Notice the familiarity between this equation and the generic equation for generalization error rate presented in **Equation 3.11**. A good model must have a total description length less than the number of bits required to encode the entire sequence of class labels. Furthermore, given two competing models, the model with lower total description length is preferred. An example showing how to compute the total description length of a decision tree is given in Exercise 10 on page 189.

3.5.3 Estimating Statistical Bounds

Instead of using **Equation 3.11** to estimate the generalization error rate of a model, an alternative way is to apply a statistical correction to the training error rate of the model that is indicative of its model complexity. This can be done if the probability distribution of training error is available or can be assumed. For example, the number of errors committed by a leaf node in a decision tree can be assumed to follow a binomial distribution. We can thus compute an upper bound limit to the observed training error rate that can be used for model selection, as illustrated in the following example.

3.10. Example Statistical Bounds on Training Error

Consider the left-most branch of the binary decision trees shown in **Figure 3.30**. Observe that the left-most leaf node of TR has been expanded into two child nodes in TL. Before splitting, the training error rate of the node is 2/7=0.286. By approximating a binomial distribution with a normal distribution, the following upper bound of the training error rate e can be derived:

$$e_{upper}(N, e, \alpha) = \frac{e + \frac{z_{\alpha/2}^2}{2N} + z_{\alpha/2}\sqrt{\frac{e(1-e)}{N} + \frac{z_{\alpha/2}^2}{4N^2}}}{1 + \frac{z_{\alpha/2}^2}{N}}, \qquad (3.13)$$

where α is the confidence level, $z_{\alpha/2}$ is the standardized value from a standard normal distribution, and N is the total number of training instances used to compute e. By replacing $\alpha=25\%, N=7$, and $e=2/7$, the upper bound for the error rate is $e_{upper}(7, 2/7, 0.25)=0.503$, which corresponds to $7\times0.503=3.521$ errors. If we expand the node into its child nodes as shown in T_L, the training error rates for the child nodes are $1/4=0.250$ and $1/3=0.333$, respectively. Using **Equation (3.13)**, the upper bounds of these error rates are $e_{upper}(4, 1/4, 0.25)=0.537$ and $e_{upper}(3, 1/3, 0.25)=0.650$, respectively. The overall training error of the child nodes is $4\times0.537+3\times0.650=4.098$, which is larger than the estimated error for the corresponding node in T_R, suggesting that it should not be split.

3.5.4 Model Selection for Decision Trees

Building on the generic approaches presented above, we present two commonly used model selection strategies for decision tree induction.

Prepruning (Early Stopping Rule)

In this approach, the tree-growing algorithm is halted before generating a fully grown tree that perfectly fits the entire training data. To do this, a more restrictive stopping condition must be used; e.g., stop expanding a leaf node when the observed gain in the generalization error estimate falls below a certain threshold. This estimate of the generalization error rate can be

computed using any of the approaches presented in the preceding three subsections, e.g., by using pessimistic error estimates, by using validation error estimates, or by using statistical bounds. The advantage of prepruning is that it avoids the computations associated with generating overly complex subtrees that overfit the training data. However, one major drawback of this method is that, even if no significant gain is obtained using one of the existing splitting criterion, subsequent splitting may result in better subtrees. Such subtrees would not be reached if prepruning is used because of the greedy nature of decision tree induction.

Post-pruning

In this approach, the decision tree is initially grown to its maximum size. This is followed by a tree-pruning step, which proceeds to trim the fully grown tree in a bottom-up fashion. Trimming can be done by replacing a subtree with (1) a new leaf node whose class label is determined from the majority class of instances affiliated with the subtree (approach known as **subtree replacement**), or (2) the most frequently used branch of the subtree (approach known as **subtree raising**). The tree-pruning step terminates when no further improvement in the generalization error estimate is observed beyond a certain threshold. Again, the estimates of generalization error rate can be computed using any of the approaches presented in the previous three subsections. Post-pruning tends to give better results than prepruning because it makes pruning decisions based on a fully grown tree, unlike prepruning, which can suffer from premature termination of the tree-growing process. However, for post-pruning, the additional computations needed to grow the full tree may be wasted when the subtree is pruned.

Figure 3.32 illustrates the simplified decision tree model for the web robot detection example given in **Section 3.3.5**. Notice that the subtree rooted at depth=1 has been replaced by one of its branches corresponding to

breadth<=7, width>3, and MultiP=1, using subtree raising. On the other hand, the subtree corresponding to depth>1 and MultiAgent=0 has been replaced by a leaf node assigned to class 0, using subtree replacement. The subtree for depth>1 and MultiAgent=1 remains intact.

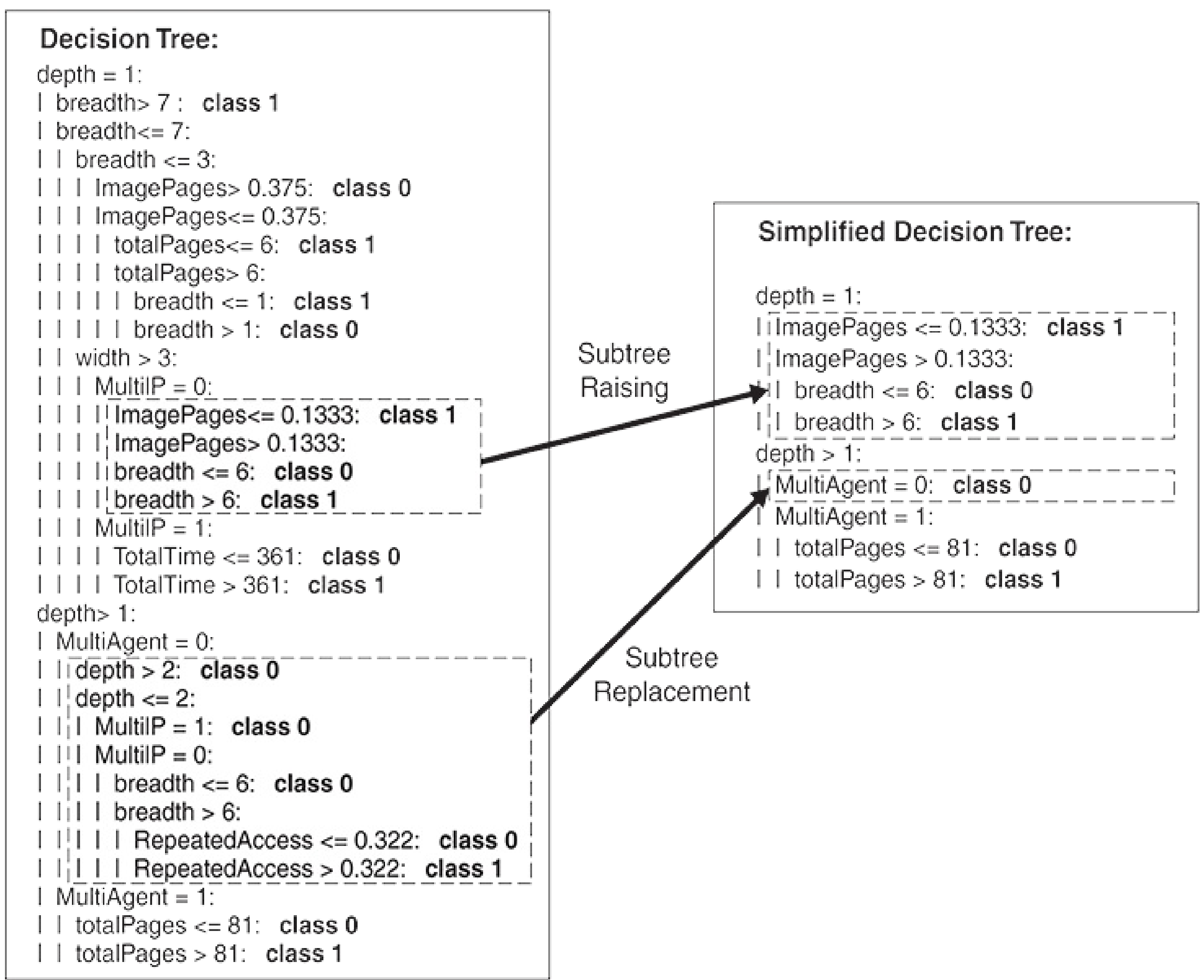


Figure 3.32.

Post-pruning of the decision tree for web robot detection.

3.6 Model Evaluation

The previous section discussed several approaches for model selection that can be used to learn a classification model from a training set *D.train*. Here we discuss methods for estimating its generalization performance, i.e. its performance on unseen instances outside of *D.train*. This process is known as **model evaluation**.

Note that model selection approaches discussed in **Section 3.5** also compute an estimate of the generalization performance using the training set *D.train*. However, these estimates are *biased* indicators of the performance on unseen instances, since they were used to guide the selection of classification model. For example, if we use the validation error rate for model selection (as described in **Section 3.5.1**), the resulting model would be deliberately chosen to minimize the errors on the validation set. The validation error rate may thus under-estimate the true generalization error rate, and hence cannot be reliably used for model evaluation.

A correct approach for model evaluation would be to assess the performance of a learned model on a labeled test set has not been used at any stage of model selection. This can be achieved by partitioning the entire set of labeled instances *D*, into two disjoint subsets, *D.train*, which is used for model selection and *D.test*, which is used for computing the test error rate, errtest. In the following, we present two different approaches for partitioning *D* into *D.train* and *D.test*, and computing the test error rate, errtest.

3.6.1 Holdout Method

The most basic technique for partitioning a labeled data set is the holdout method, where the labeled set *D* is randomly partitioned into two disjoint sets, called the training set *D.train* and the test set *D.test*. A classification model is then induced from *D.train* using the model selection approaches presented in **Section 3.5**, and its error rate on *D.test*, errtest, is used as an estimate of the generalization error rate. The proportion of data reserved for training and for testing is typically at the discretion of the analysts, e.g., two-thirds for training and one-third for testing.

Similar to the trade-off faced while partitioning *D.train* into *D.tr* and *D.val* in **Section 3.5.1**, choosing the right fraction of labeled data to be used for training and testing is non-trivial. If the size of *D.train* is small, the learned classification model may be improperly learned using an insufficient number of training instances, resulting in a biased estimation of generalization performance. On the other hand, if the size of *D.test* is small, errtest may be less reliable as it would be computed over a small number of test instances. Moreover, errtest can have a high variance as we change the random partitioning of *D* into *D.train* and *D.test*.

The holdout method can be repeated several times to obtain a distribution of the test error rates, an approach known as **random subsampling** or repeated holdout method. This method produces a distribution of the error rates that can be used to understand the variance of errtest.

3.6.2 Cross-Validation

Cross-validation is a widely-used model evaluation method that aims to make effective use of all labeled instances in *D* for both training and testing. To illustrate this method, suppose that we are given a labeled set that we have

randomly partitioned into three equal-sized subsets, S1, S2, and S3, as shown in **Figure 3.33**. For the first run, we train a model using subsets S2 and S_3 (shown as empty blocks) and test the model on subset S1. The test error rate on S1, denoted as err(S1), is thus computed in the first run. Similarly, for the second run, we use S1 and S3 as the training set and S2 as the test set, to compute the test error rate, err(S2), on S2. Finally, we use S1 and S3 for training in the third run, while S3 is used for testing, thus resulting in the test error rate err(S3) for S3. The overall test error rate is obtained by summing up the number of errors committed in each test subset across all runs and dividing it by the total number of instances. This approach is called three-fold cross-validation.

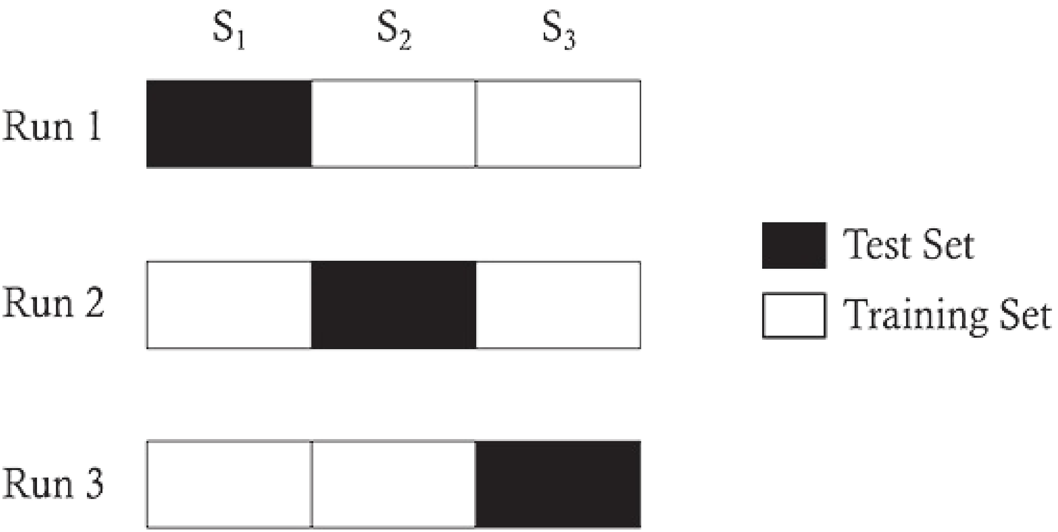


Figure 3.33.
Example demonstrating the technique of 3-fold cross-validation.

The *k*-fold cross-validation method generalizes this approach by segmenting the labeled data *D* (of size *N*) into *k* equal-sized partitions (or folds). During the ith run, one of the partitions of *D* is chosen as *D.test*(*i*) for testing, while the rest of the partitions are used as *D.train*(*i*) for training. A model *m*(*i*) is learned using *D.train*(*i*) and applied on *D.test*(*i*) to obtain the sum of test errors,

errsum(i). This procedure is repeated *k* times. The total test error rate, errtest, is then computed as

$$errtest=\frac{\sum_{i=1}^{k} errsum(i)}{N}. \quad (3.14)$$

Every instance in the data is thus used for testing exactly once and for training exactly $(k-1)$ times. Also, every run uses $(k-1)/k$ fraction of the data for training and 1/*k* fraction for testing.

The right choice of *k* in *k*-fold cross-validation depends on a number of characteristics of the problem. A small value of *k* will result in a smaller training set at every run, which will result in a larger estimate of generalization error rate than what is expected of a model trained over the entire labeled set. On the other hand, a high value of *k* results in a larger training set at every run, which reduces the bias in the estimate of generalization error rate. In the extreme case, when $k=N$, every run uses exactly one data instance for testing and the remainder of the data for testing. This special case of *k*-fold cross-validation is called the **leave-one-out** approach. This approach has the advantage of utilizing as much data as possible for training. However, leave-one-out can produce quite misleading results in some special scenarios, as illustrated in Exercise 11. Furthermore, leave-one-out can be computationally expensive for large data sets as the cross-validation procedure needs to be repeated *N* times. For most practical applications, the choice of *k* between 5 and 10 provides a reasonable approach for estimating the generalization error rate, because each fold is able to make use of 80% to 90% of the labeled data for training.

The *k*-fold cross-validation method, as described above, produces a single estimate of the generalization error rate, without providing any information about the variance of the estimate. To obtain this information, we can run *k*-fold cross-validation for every possible partitioning of the data into *k* partitions,

and obtain a distribution of test error rates computed for every such partitioning. The average test error rate across all possible partitionings serves as a more robust estimate of generalization error rate. This approach of estimating the generalization error rate and its variance is known as the **complete cross-validation** approach. Even though such an estimate is quite robust, it is usually too expensive to consider all possible partitionings of a large data set into *k* partitions. A more practical solution is to repeat the cross-validation approach multiple times, using a different random partitioning of the data into *k* partitions at every time, and use the average test error rate as the estimate of generalization error rate. Note that since there is only one possible partitioning for the leave-one-out approach, it is not possible to estimate the variance of generalization error rate, which is another limitation of this method.

The *k*-fold cross-validation does not guarantee that the fraction of positive and negative instances in every partition of the data is equal to the fraction observed in the overall data. A simple solution to this problem is to perform a stratified sampling of the positive and negative instances into *k* partitions, an approach called **stratified cross-validation**.

In *k*-fold cross-validation, a different model is learned at every run and the performance of every one of the *k* models on their respective test folds is then aggregated to compute the overall test error rate, errtest. Hence, errtest does not reflect the generalization error rate of any of the *k* models. Instead, it reflects the *expected* generalization error rate of the *model selection approach*, when applied on a training set of the same size as one of the training folds (N(k−1)/k). This is different than the errtest computed in the holdout method, which exactly corresponds to the specific model learned over *D.train*. Hence, although effectively utilizing every data instance in *D* for training and testing, the errtest computed in the cross-validation method does not represent the performance of a single model learned over a specific *D.train*.

Nonetheless, in practice, errtest is typically used as an estimate of the generalization error of a model built on *D*. One motivation for this is that when the size of the training folds is closer to the size of the overall data (when *k* is large), then errtest resembles the expected performance of a model learned over a data set of the same size as *D*. For example, when *k* is 10, every training fold is 90% of the overall data. The errtest then should approach the expected performance of a model learned over 90% of the overall data, which will be close to the expected performance of a model learned over *D*.

3.7 Presence of Hyper-parameters

Hyper-parameters are parameters of learning algorithms that need to be determined before learning the classification model. For instance, consider the hyper-parameter α that appeared in **Equation 3.11**, which is repeated here for convenience. This equation was used for estimating the generalization error for a model selection approach that used an explicit representations of model complexity. (See **Section 3.5.2**.)

$$gen.error(m)=train.error(m, D.train)+\alpha\times complexity(M)$$

For other examples of hyper-parameters, see **Chapter 4**.

Unlike regular model parameters, such as the test conditions in the internal nodes of a decision tree, hyper-parameters such as α do not appear in the final classification model that is used to classify unlabeled instances. However, the values of hyper-parameters need to be determined during model selection—a process known as **hyper-parameter selection**—and must be taken into account during model evaluation. Fortunately, both tasks can be effectively accomplished via slight modifications of the cross-validation approach described in the previous section.

3.7.1 Hyper-parameter Selection

In **Section 3.5.2**, a validation set was used to select α and this approach is generally applicable for hyper-parameter section. Let *p* be the hyper-parameter that needs to be selected from a finite range of values, P=

{p1, p2, … pn }. Partition *D.train* into *D.tr* and *D.val*. For every choice of hyper-parameter value pi, we can learn a model mi on *D.tr*, and apply this model on *D.val* to obtain the validation error rate errval(pi). Let p* be the hyper-parameter value that provides the lowest validation error rate. We can then use the model m* corresponding to p* as the final choice of classification model.

The above approach, although useful, uses only a subset of the data, *D.train*, for training and a subset, *D.val*, for validation. The framework of cross-validation, presented in **Section 3.6.2**, addresses both of those issues, albeit in the context of model evaluation. Here we indicate how to use a cross-validation approach for hyper-parameter selection. To illustrate this approach, let us partition *D.train* into three folds as shown in **Figure 3.34**. At every run, one of the folds is used as *D.val* for validation, and the remaining two folds are used as *D.tr* for learning a model, for every choice of hyper-parameter value pi. The overall validation error rate corresponding to each pi is computed by summing the errors across all the three folds. We then select the hyper-parameter value p* that provides the lowest validation error rate, and use it to learn a model m* on the entire training set *D.train*.

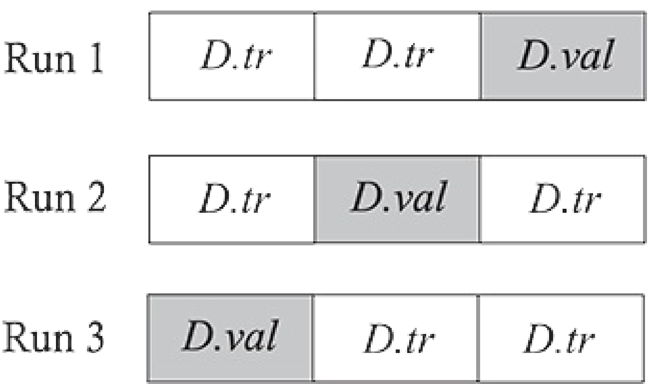


Figure 3.34.
Example demonstrating the 3-fold cross-validation framework for hyper-parameter selection using *D.train*.

Algorithm 3.2 generalizes the above approach using a *k*-fold cross-validation framework for hyper-parameter selection. At the ith run of cross-validation, the data in the ith fold is used as *D.val*(*i*) for validation (Step 4), while the remainder of the data in *D.train* is used as *D.tr*(*i*) for training (Step 5). Then for every choice of hyper-parameter value pi, a model is learned on *D.tr*(*i*) (Step 7), which is applied on *D.val*(*i*) to compute its validation error (Step 8). This is used to compute the validation error rate corresponding to models learning using pi over all the folds (Step 11). The hyper-parameter value p* that provides the lowest validation error rate (Step 12) is now used to learn the final model m* on the entire training set *D.train* (Step 13). Hence, at the end of this algorithm, we obtain the best choice of the hyper-parameter value as well as the final classification model (Step 14), both of which are obtained by making an effective use of every data instance in *D.train*.

Algorithm 3.2 Procedure model-select(k, P, D.train)

```
 1: N_train = |D.train|    {Size of D.train.}
 2: Divide D.train into k partitions, D.train_1 to D.train_k.
 3: for each run i = 1 to k do
 4:   D.val(i) = D.train_i. {Partition used for validation.}
 5:   D.tr(i) = D.train \ D.train_i. {Partitions used for training.}
 6:   for each parameter p ∈ P do
 7:     m = model-train(p, D.tr(i)). {Train model}
 8:     err_sum(p, i) = model-test(m, D.val(i)). {Sum of validation
errors.}
 9:   end for
10: end for
```

```
11: errval(p)=∑ik errsum(p, i)/Ntrain. {Compute validation error rate.}
12: p* = argmin_p err_val(p). {Select best hyper-parameter value.}
13: m* = model-train(p*, D.train). {Learn final model on D.train}
14: return (p*, m*).
```

3.7.2 Nested Cross-Validation

The approach of the previous section provides a way to effectively use all the instances in *D.train* to learn a classification model when hyper-parameter selection is required. This approach can be applied over the entire data set *D* to learn the final classification model. However, applying **Algorithm 3.2** on *D* would only return the final classification model m* but not an estimate of its generalization performance, errtest. Recall that the validation error rates used in **Algorithm 3.2** cannot be used as estimates of generalization performance, since they are used to guide the selection of the final model m*. However, to compute errtest, we can again use a cross-validation framework for evaluating the performance on the entire data set *D*, as described originally in **Section 3.6.2**. In this approach, *D* is partitioned into *D.train* (for training) and *D.test* (for testing) at every run of cross-validation. When hyper-parameters are involved, we can use **Algorithm 3.2** to train a model using *D.train* at every run, thus "internally" using cross-validation for model selection. This approach is called **nested cross-validation** or double cross-validation. Algorithm 3.3 describes the complete approach for estimating errtest using nested cross-validation in the presence of hyper-parameters.

As an illustration of this approach, see **Figure 3.35** where the labeled set *D* is partitioned into *D.train* and *D.test*, using a 3-fold cross-validation method.

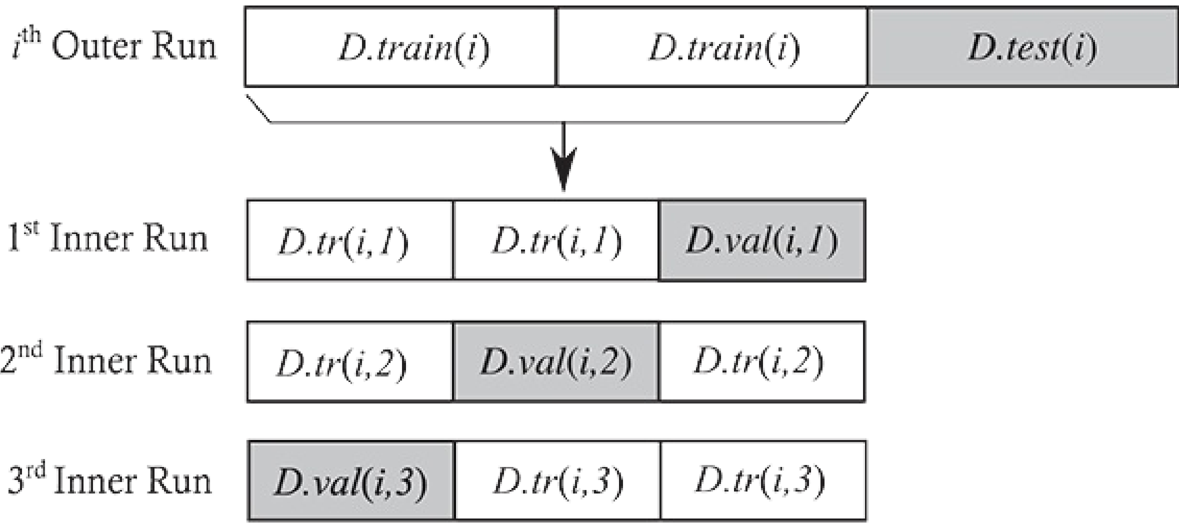


Figure 3.35.

Example demonstrating 3-fold nested cross-validation for computing errtest.

At the ith run of this method, one of the folds is used as the test set, *D.test*(*i*), while the remaining two folds are used as the training set, *D.train*(*i*). This is represented in **Figure 3.35** as the ith "outer" run. In order to select a model using *D.train*(*i*), we again use an "inner" 3-fold cross-validation framework that partitions *D.train*(*i*) into *D.tr* and *D.val* at every one of the three inner runs (iterations). As described in **Section 3.7**, we can use the inner cross-validation framework to select the best hyper-parameter value p*(i) as well as its resulting classification model m*(i) learned over *D.train*(*i*). We can then apply m*(i) on *D.test*(*i*) to obtain the test error at the ith outer run. By repeating this process for every outer run, we can compute the average test error rate, errtest, over the entire labeled set *D*. Note that in the above approach, the inner cross-validation framework is being used for model selection while the outer cross-validation framework is being used for model evaluation.

Algorithm 3.3 The nested cross-validation approach for computing errtest.

```
 1: Divide D into k partitions, D_1 to D_k.
 2: for each outer run i = 1 to k do
 3:    D.test(i) = D_i. {Partition used for testing.}
 4:    D.train(i) = D \ D_i. {Partitions used for model selection.}
 5:    (p*, m*(i)) = model-select(k, P, D.train(i)). {Inner cross-validation.}
 6:    err_sum(i) = model-test(m*(i), D.test(i)). {Sum of test errors.}
 7: end for
 8: errtest=∑ikerrtest(i)/N. {Compute test error rate.}
```

3.8 Pitfalls of Model Selection and Evaluation

Model selection and evaluation, when used effectively, serve as excellent tools for learning classification models and assessing their generalization performance. However, when using them effectively in practical settings, there are several pitfalls that can result in improper and often misleading conclusions. Some of these pitfalls are simple to understand and easy to avoid, while others are quite subtle in nature and difficult to catch. In the following, we present two of these pitfalls and discuss best practices to avoid them.

3.8.1 Overlap between Training and Test Sets

One of the basic requirements of a *clean* model selection and evaluation setup is that the data used for model selection (*D.train*) must be kept separate from the data used for model evaluation (*D.test*). If there is any overlap between the two, the test error rate errtest computed over *D.test* cannot be considered representative of the performance on *unseen* instances. Comparing the effectiveness of classification models using errtest can then be quite misleading, as an overly complex model can show an inaccurately low value of errtest due to model overfitting (see Exercise 12 at the end of this chapter).

To illustrate the importance of ensuring no overlap between *D.train* and *D.test*, consider a labeled data set where all the attributes are irrelevant, i.e. they have no relationship with the class labels. Using such attributes, we should expect no classification model to perform better than random guessing. However, if the test set involves even a small number of data instances that were used for training, there is a possibility for an overly complex model to show better performance than random, even though the attributes are completely irrelevant. As we will see later in **Chapter 10**, this scenario can actually be used as a criterion to detect overfitting due to improper setup of experiment. If a model shows better performance than a random classifier even when the attributes are irrelevant, it is an indication of a potential feedback between the training and test sets.

3.8.2 Use of Validation Error as Generalization Error

The validation error rate errval serves an important role during model selection, as it provides "out-of-sample" error estimates of models on *D.val*, which is not used for training the models. Hence, errval serves as a better metric than the training error rate for selecting models and hyper-parameter values, as described in **Sections 3.5.1** and **3.7**, respectively. However, once the validation set has been used for selecting a classification model m*, errval no longer reflects the performance of m* on *unseen* instances.

To realize the pitfall in using validation error rate as an estimate of generalization performance, consider the problem of selecting a hyper-parameter value *p* from a range of values P, using a validation set *D.val*. If the number of possible values in P is quite large and the size of *D.val* is small, it is

possible to select a hyper-parameter value p* that shows favorable performance on *D.val* just by random chance. Notice the similarity of this problem with the multiple comparisons problem discussed in **Section 3.4.1**. Even though the classification model m* learned using p* would show a low validation error rate, it would lack generalizability on unseen test instances.

The correct approach for estimating the generalization error rate of a model m* is to use an independently chosen test set *D.test* that hasn't been used in any way to influence the selection of m*. As a rule of thumb, the test set should never be examined during model selection, to ensure the absence of any form of overfitting. If the insights gained from any portion of a labeled data set help in improving the classification model even in an indirect way, then that portion of data must be discarded during testing.

3.9 Model Comparison*

One difficulty when comparing the performance of different classification models is whether the observed difference in their performance is statistically significant. For example, consider a pair of classification models, MA and MB. Suppose MA achieves 85% accuracy when evaluated on a test set containing 30 instances, while MB achieves 75% accuracy on a different test set containing 5000 instances. Based on this information, is MA a better model than MB? This example raises two key questions regarding the statistical significance of a performance metric:

1. Although MA has a higher accuracy than MB, it was tested on a smaller test set. How much confidence do we have that the accuracy for MA is actually 85%?
2. Is it possible to explain the difference in accuracies between MA and MB as a result of variations in the composition of their test sets?

The first question relates to the issue of estimating the confidence interval of model accuracy. The second question relates to the issue of testing the statistical significance of the observed deviation. These issues are investigated in the remainder of this section.

3.9.1 Estimating the Confidence Interval for Accuracy

To determine its confidence interval, we need to establish the probability distribution for sample accuracy. This section describes an approach for deriving the confidence interval by modeling the classification task as a binomial random experiment. The following describes characteristics of such an experiment:

1. The random experiment consists of *N* independent trials, where each trial has two possible outcomes: success or failure.
2. The probability of success, *p*, in each trial is constant.

An example of a binomial experiment is counting the number of heads that turn up when a coin is flipped *N* times. If *X* is the number of successes observed in *N* trials, then the probability that *X* takes a particular value is given by a binomial distribution with mean Np and variance $Np(1-p)$:

$$P(X=\upsilon)=\binom{N}{\upsilon}p^{\upsilon}(1-p)^{N-\upsilon}.$$

For example, if the coin is fair ($p=0.5$) and is flipped fifty times, then the probability that the head shows up 20 times is

$$P(X=20)=\binom{50}{20}0.5^{20}(1-0.5)^{30}=0.0419.$$

If the experiment is repeated many times, then the average number of heads expected to show up is $50\times0.5=25$, while its variance is $50\times0.5\times0.5=12.5$.

The task of predicting the class labels of test instances can also be considered as a binomial experiment. Given a test set that contains *N* instances, let *X* be the number of instances correctly predicted by a model and *p* be the true accuracy of the model. If the prediction task is modeled as a binomial experiment, then *X* has a binomial distribution with mean Np and variance $Np(1-p)$. It can be shown that the empirical accuracy, $acc=X/N$, also

has a binomial distribution with mean *p* and variance $p(1-p)/N$ (see Exercise 14). The binomial distribution can be approximated by a normal distribution when *N* is sufficiently large. Based on the normal distribution, the confidence interval for *acc* can be derived as follows:

$$P\left(-Z_{\alpha/2} \le \frac{acc-p}{\sqrt{p(1-p)/N}} \le Z_{1-\alpha/2}\right)=1-\alpha, \tag{3.15}$$

where $Z_{\alpha/2}$ and $Z_{1-\alpha/2}$ are the upper and lower bounds obtained from a standard normal distribution at confidence level $(1-\alpha)$. Since a standard normal distribution is symmetric around $Z=0$, it follows that $Z_{\alpha/2}=Z_{1-\alpha/2}$. Rearranging this inequality leads to the following confidence interval for *p*:

$$\frac{2\times N\times acc\times Z_{\alpha/2}^2 \pm Z_{\alpha/2}\sqrt{Z_{\alpha/2}^2+4N\,acc-4N\,acc^2}}{2(N+Z_{\alpha/2}^2)}. \tag{3.16}$$

The following table shows the values of $Z_{\alpha/2}$ at different confidence levels:

$1-\alpha$	0.99	0.98	0.95	0.9	0.8	0.7	0.5
$Z_{\alpha/2}$	2.58	2.33	1.96	1.65	1.28	1.04	0.67

3.11. Example Confidence Interval for Accuracy

Consider a model that has an accuracy of 80% when evaluated on 100 test instances. What is the confidence interval for its true accuracy at a 95% confidence level? The confidence level of 95% corresponds to $Z_{a/2}=1.96$ according to the table given above. Inserting this term into **Equation 3.16** yields a confidence interval between 71.1% and 86.7%. The following table shows the confidence interval when the number of instances, *N*, increases:

N	20	50	100	500	1000	5000

Confidence	0.584	0.670	0.711	0.763	0.774	0.789
Interval	−0.919	−0.888	−0.867	−0.833	−0.824	−0.811

Note that the confidence interval becomes tighter when *N* increases.

3.9.2 Comparing the Performance of Two Models

Consider a pair of models, M_1 and M_2, which are evaluated on two independent test sets, D_1 and D_2. Let n_1 denote the number of instances in D_1 and n_2 denote the number of instances in D_2. In addition, suppose the error rate for M_1 on D_1 is e_1 and the error rate for M_2 on D_2 is e_2. Our goal is to test whether the observed difference between e_1 and e_2 is statistically significant.

Assuming that n_1 and n_2 are sufficiently large, the error rates e_1 and e_2 can be approximated using normal distributions. If the observed difference in the error rate is denoted as $d=e_1-e_2$, then *d* is also normally distributed with mean d_t, its true difference, and variance, σ_d^2. The variance of *d* can be computed as follows:

$$\sigma_d^2 \simeq \hat{\sigma}_d^2 = \frac{e_1(1-e_1)}{n_1}+\frac{e_2(1-e_2)}{n_2}, \tag{3.17}$$

where $e_1(1-e_1)/n_1$ and $e_2(1-e_1)/n_2$ are the variances of the error rates. Finally, at the $(1-\alpha)\%$ confidence level, it can be shown that the confidence interval for the true difference *dt* is given by the following equation:

$d_t = d \pm z_{\alpha/2} \hat{\sigma}_d.$ (3.18)

3.12. Example Significance Testing

Consider the problem described at the beginning of this section. Model M_A has an error rate of $e_1=0.15$ when applied to $N_1=30$ test instances, while model M_B has an error rate of $e_2=0.25$ when applied to $N_2=5000$ test instances. The observed difference in their error rates is $d=|0.15-0.25|=0.1$. In this example, we are performing a two-sided test to check whether $d_t=0$ or $d_t \neq 0$. The estimated variance of the observed difference in error rates can be computed as follows:

$$\hat{\sigma}_d^2 = \frac{0.15(1-0.15)}{30} + \frac{0.25(1-0.25)}{5000} = 0.0043$$

or $\hat{\sigma}_d = 0.0655$. Inserting this value into **Equation 3.18**, we obtain the following confidence interval for d_t at 95% confidence level:

$$d_t = 0.1 \pm 1.96 \times 0.0655 = 0.1 \pm 0.128.$$

As the interval spans the value zero, we can conclude that the observed difference is not statistically significant at a 95% confidence level.

At what confidence level can we reject the hypothesis that $d_t=0$? To do this, we need to determine the value of $Z_{\alpha/2}$ such that the confidence interval for d_t does not span the value zero. We can reverse the preceding computation and look for the value $Z_{\alpha/2}$ such that $d > Z_{\sigma/2} \hat{\sigma}_d$. Replacing the values of *d* and $\hat{\sigma}_d$ gives $Z_{\sigma/2} < 1.527$. This value first occurs when $(1-\alpha) \lesssim 0.936$ (for a two-sided test). The result suggests that the null hypothesis can be rejected at confidence level of 93.6% or lower.