Chapter 9
Statistical Pattern Recognition

9.1 Introduction

Statistical pattern recognition is an application in computational statistics that uses many of the concepts we have covered so far, such as probability density estimation and cross-validation. Examples where statistical pattern recognition techniques can be used are numerous and arise in disciplines such as medicine, computer vision, robotics, military systems, manufacturing, finance and many others. Some of these include the following:

- A doctor diagnoses a patient’s illness based on the symptoms and test results.
- A radiologist locates areas where there is non-healthy tissue in x-rays.
- A military analyst classifies regions of an image as natural or man-made for use in targeting systems.
- A geologist determines whether a seismic signal represents an impending earthquake.
- A loan manager at a bank must decide whether a customer is a good credit risk based on their income, past credit history and other variables.
- A manufacturer must classify the quality of materials before using them in their products.

In all of these applications, the human is often assisted by statistical pattern recognition techniques.

Statistical methods for pattern recognition are covered in this chapter. In this section, we first provide a brief introduction to the goals of pattern recognition and a broad overview of the main steps of building classifiers. In Section 9.2 we present a discussion of Bayes classifiers and pattern recognition in an hypothesis testing framework. Section 9.3 contains techniques for
evaluating the classifier. In Section 9.4, we illustrate how to construct classification trees. Section 9.5 contains methods for unsupervised classification or clustering, including agglomerative methods and $k$-means clustering.

We first describe the process of statistical pattern recognition in a supervised learning setting. With supervised learning, we have cases or observations where we know which class each case belongs to. Figure 9.1 illustrates the major steps of statistical pattern recognition.

The first step in pattern recognition is to select features that will be used to distinguish between the classes. As the reader might suspect, the choice of features is perhaps the most important part of the process. Building accurate classifiers is much easier with features that allow one to readily distinguish between classes.

Once features are selected, we obtain a sample of these features for the different classes. This means that we find objects that belong to the classes of interest and then measure the features. Each observed set of feature measurements (sometimes also called a case or pattern) has a class label attached to it. Now that we have data that are known to belong to the different classes, we can use this information to create the methodology that will take as input a set of feature measurements and output the class that it belongs to. How these classifiers are created will be the topic of this chapter.

One of the main examples we use to illustrate these ideas is one that we encountered in Chapter 5. In the iris data set, we have three species of iris: Iris setosa, Iris versicolor and Iris virginica. The data were used by Fisher [1936] to develop a classifier that would take measurements from a new iris and determine its species based on the features [Hand, et al., 1994]. The four features that are used to distinguish the species of iris are sepal length, sepal width, petal length and petal width. The next step in the pattern recognition process is to find many flowers from each species and measure the corresponding sepal length, sepal width, petal length, and petal width. For each set of measured features, we attach a class label that indicates which species

© 2002 by Chapman & Hall/CRC
it belongs to. We build a classifier using these data and (possibly) one of the techniques that are described in this chapter. To use the classifier, we measure the four features for an iris of unknown species and use the classifier to assign the species membership.

Sometimes we are in a situation where we do not know the class membership for our observations. Perhaps we are unable or unwilling to assume how many groups are represented by the data. In this case, we are in the unsupervised learning mode. To illustrate this, say we have data that comprise measurements of a type of insect called *Chaetocnema* [Lindsey, Herzberg, and Watts, 1987; Hand, et al., 1994]. These variables measure the width of the first joint of the first tarsus, the width of the first joint of the second tarsus, and the maximal width of the aedegus. All measurements are in microns. We suspect that there are three species represented by these data. To explore this hypothesis further, we could use one of the unsupervised learning or clustering techniques that will be covered in Section 9.5.

### 9.2 Bayes Decision Theory

The Bayes approach to pattern classification is a fundamental technique, and we recommend it as the starting point for most pattern recognition applications. If this method is not adequate, then more complicated techniques may be used (e.g., neural networks, classification trees). Bayes decision theory poses the classification problem in terms of probabilities; therefore, all of the probabilities must be known or estimated from the data. We will see that this is an excellent application of the probability density estimation methods from Chapter 8.

We have already seen an application of Bayes decision theory in Chapter 2. There we wanted to know the probability that a piston ring came from a particular manufacturer given that it failed. It makes sense to make the decision that the part came from the manufacturer that has the highest posterior probability. To put this in the pattern recognition context, we could think of the part failing as the feature. The resulting classification would be the manufacturer ($M_A$ or $M_B$) that sold us the part. In the following, we will see that Bayes decision theory is an application of Bayes’ Theorem, where we will classify observations using the posterior probabilities.

We start off by fixing some notation. Let the class membership be represented by $\omega_j$, $j = 1, ..., J$ for a total of $J$ classes. For example, with the iris data, we have $J = 3$ classes:

\[
\begin{align*}
\omega_1 &= Iris\ setosa \\
\omega_2 &= Iris\ versicolor \\
\omega_3 &= Iris\ virginica.
\end{align*}
\]
The features we are using for classification are denoted by the \( d \)-dimensional vector \( \mathbf{x} \). With the \textit{iris} data, we have four measurements, so \( d = 4 \). In the supervised learning situation, each of the observed feature vectors will also have a class label attached to it.

Our goal is to use the data to create a decision rule or classifier that will take a feature vector \( \mathbf{x} \) whose class membership is unknown and return the class it most likely belongs to. A logical way to achieve this is to assign the class label to this feature vector using the class corresponding to the highest posterior probability. This probability is given by

\[
P(\omega_j | \mathbf{x}); \quad j = 1, \ldots, J. \tag{9.1}
\]

Equation 9.1 represents the probability that the case belongs to the \( j \)-th class given the observed feature vector \( \mathbf{x} \). To use this rule, we would evaluate all of the \( J \) posterior probabilities, and the one with the highest probability would be the class we choose. We can find the posterior probabilities using Bayes’ Theorem:

\[
P(\omega_j | \mathbf{x}) = \frac{P(\omega_j)P(\mathbf{x}|\omega_j)}{P(\mathbf{x})}, \tag{9.2}
\]

where

\[
P(\mathbf{x}) = \sum_{j=1}^{J} P(\omega_j)P(\mathbf{x}|\omega_j). \tag{9.3}
\]

We see from Equation 9.2 that we must know the prior probability that it would be in class \( j \) given by

\[
P(\omega_j); \quad j = 1, \ldots, J, \tag{9.4}
\]

and the class-conditional probability (sometimes called the state-conditional probability)

\[
P(\mathbf{x}|\omega_j); \quad j = 1, \ldots, J. \tag{9.5}
\]

The class-conditional probability in Equation 9.5 represents the probability distribution of the features for each class. The prior probability in Equation 9.4 represents our initial degree of belief that an observed set of features is a case from the \( j \)-th class. The process of estimating these probabilities is how we build the classifier.

We start our explanation with the prior probabilities. These can either be inferred from prior knowledge of the application, estimated from the data or...
assumed to be equal. In the piston ring example, we know how many parts we buy from each manufacturer. So, the prior probability that the part came from a certain manufacturer would be based on the percentage of parts obtained from that manufacturer. In other applications, we might know the prevalence of some class in our population. This might be the case in medical diagnosis, where we have some idea of the percentage of the population who are likely to have a certain disease or medical condition. In the case of the iris data, we could estimate the prior probabilities using the proportion of each class in our sample. We had 150 observed feature vectors, with 50 coming from each class. Therefore, our estimated prior probabilities would be

\[
\hat{P}(\omega_j) = \frac{n_j}{N} = \frac{50}{150} = 0.33; \quad j = 1, 2, 3.
\]

Finally, we might use equal priors when we believe each class is equally likely.

Now that we have our prior probabilities, \( \hat{P}(\omega_j) \), we turn our attention to the class-conditional probabilities \( P(x|\omega_j) \). We can use the density estimation techniques covered in Chapter 8 to obtain these probabilities. In essence, we take all of the observed feature vectors that are known to come from class \( \omega_j \) and estimate the density using only those cases. We will cover two approaches: parametric and nonparametric.

**Estimating Class-Conditional Probabilities: Parametric Method**

In parametric density estimation, we assume a distribution for the class-conditional probability densities and estimate them by estimating the corresponding distribution parameters. For example, we might assume the features come from a multivariate normal distribution. To estimate the density, we have to estimate \( \hat{\mu}_j \) and \( \hat{\Sigma}_j \) for each class. This procedure is illustrated in Example 9.1 for the iris data.

**Example 9.1**

In this example, we estimate our class-conditional probabilities using the iris data. We assume that the required probabilities are multivariate normal for each class. The following MATLAB code shows how to get the class-conditional probabilities for each species of iris.

```matlab
load iris
% This loads up three matrices:
% setosa, virginica and versicolor
% We will assume each class is multivariate normal.
% To get the class-conditional probabilities, we
% get estimates for the parameters for each class.
muset = mean(setosa);
```

© 2002 by Chapman & Hall/CRC
Estimating Class-Conditional Probabilities: Nonparametric

If it is not appropriate to assume the features for a class follow a known distribution, then we can use the nonparametric density estimation techniques from Chapter 8. These include the averaged shifted histogram, the frequency polygon, kernel densities, finite mixtures and adaptive mixtures. To obtain the class-conditional probabilities, we take the set of measured features from each class and estimate the density using one of these methods. This is illustrated in Example 9.2, where we use the product kernel to estimate the probability densities for the iris data.

Example 9.2

We estimate the class-conditional probability densities for the iris data using the product kernel, where the univariate normal kernel is used for each dimension. We illustrate the use of two functions for estimating the product kernel. One is called cskern2d that can only be used for bivariate data. The output arguments from this function are matrices for use in the MATLAB plotting functions surf and mesh. The cskern2d function should be used when the analyst wants to plot the resulting probability density. We use it on the first two dimensions of the iris data and plot the surface for Iris virginica in Figure 9.2.

```matlab
load iris
% This loads up three matrices:
% setosa, virginica and versicolor
% We will use the product kernel to estimate densities.
% To try this, get the kernel estimate for the first
% two features and plot.
% The arguments of 0.1 indicate the grid size in
% each dimension. This creates the domain over
% which we will estimate the density.
[xset,yset,pset]=cskern2d(setosa(:,1:2),0.1,0.1);
xvir,yvir,pvir=cskern2d(virginica(:,1:2),0.1,0.1);
xver,yver,pver=cskern2d(versicolor(:,1:2),0.1,0.1);
mesh(xvir,yvir,pvir)
colormap(gray(256))
```
A more useful function for statistical pattern recognition is \texttt{cskernmd}, which returns the value of the probability density $f(x)$ for a given $d$-dimensional vector $x$.

% If one needs the value of the probability curve, % then use this.  
\begin{verbatim}
ps = cskernmd(setosa(1,1:2),setosa(:,1:2));
pver = cskernmd(setosa(1,1:2),versicolor(:,1:2));
pvir = cskernmd(setosa(1,1:2),virginica(:,1:2));
\end{verbatim}

\section*{Bayes Decision Rule}

Now that we know how to get the prior probabilities and the class-conditional probabilities, we can use Bayes' Theorem to obtain the posterior probabilities. Bayes Decision Rule is based on these posterior probabilities.
**BAYES DECISION RULE:**

*Given a feature vector* \( x \), *assign it to class* \( \omega_i \) *if*

\[
P(\omega_i | x) > P(\omega_j | x); \quad i = 1, ..., J; \quad i \neq j. \tag{9.6}
\]

This states that we will classify an observation \( x \) as belonging to the class that has the highest posterior probability. It is known [Duda and Hart, 1973] that the decision rule given by Equation 9.6 yields a classifier with the minimum probability of error.

We can use an equivalent rule by recognizing that the denominator of the posterior probability (see Equation 9.2) is simply a normalization factor and is the same for all classes. So, we can use the following alternative decision rule:

\[
P(x | \omega_i)P(\omega_i) > P(x | \omega_j)P(\omega_j); \quad i = 1, ..., J; \quad i \neq j. \tag{9.7}
\]

Equation 9.7 is Bayes Decision Rule in terms of the class-conditional and prior probabilities. If we have equal priors for each class, then our decision is based only on the class-conditional probabilities. In this case, the decision rule partitions the feature space into \( J \) decision regions \( \Omega_1, \Omega_2, ..., \Omega_J \). If \( x \) is in region \( \Omega_j \), then we will say it belongs to class \( \omega_j \).

We now turn our attention to the error we have in our classifier when we use Bayes Decision Rule. An error is made when we classify an observation as class \( \omega_i \) when it is really in the \( j \)-th class. We denote the complement of region \( \Omega_j \) as \( \Omega'_j \), which represents every region except \( \Omega_j \). To get the probability of error, we calculate the following integral over all values of \( x \) [Duda and Hart, 1973; Webb, 1999]

\[
P(error) = \sum_{i=1}^{J} \int_{\Omega'_i} P(x | \omega_i)P(\omega_i)dx. \tag{9.8}
\]

Thus, to find the probability of making an error (i.e., assigning the wrong class to an observation), we find the probability of error for each class and add the probabilities together. In the following example, we make this clearer by looking at a two class case and calculating the probability of error.

**Example 9.3**

We will look at a univariate classification problem with equal priors and two classes. The class-conditionals are given by the normal distributions as follows:
\[ P(x|\omega_1) = \phi(x; -1, 1) \]
\[ P(x|\omega_2) = \phi(x; 1, 1). \]

The priors are

\[ P(\omega_1) = 0.6 \]
\[ P(\omega_2) = 0.4. \]

The following MATLAB code creates the required curves for the decision rule of Equation 9.7.

```matlab
% This illustrates the 1-D case for two classes.
% We will shade in the area where there can be
% misclassified observations.
% Get the domain for the densities.
dom = -6:.1:8;
dom = dom';
% Note: could use csnormp or normpdf.
pxg1 = csevalnorm(dom,-1,1);
pxg2 = csevalnorm(dom,1,1);
plot(dom,pxg1,dom,pxg2)
% Find decision regions - multiply by priors
ppxg1 = pxg1*0.6;
ppxg2 = pxg2*0.4;
plot(dom,ppxg1,'k',dom,ppxg2,'k')
xlabel('x')
```

The resulting plot is given in Figure 9.3, where we see that the decision regions given by Equation 9.7 are obtained by finding where the two curves intersect. If we observe a value of a feature given by \( x = -2 \), then we would classify that object as belonging to class \( \omega_1 \). If we observe \( x = 4 \), then we would classify that object as belonging to class \( \omega_2 \). Let’s see what happens when \( x = -0.75 \). We can find the probabilities using

\[ x = -0.75; \]
\% Evaluate each un-normalized posterior.
\[ p1 = csevalnorm(x,-1,1)*0.6; \]
\[ p2 = csevalnorm(x,1,1)*0.4; \]

\[ P(-0.75|\omega_1)P(\omega_1) = 0.23 \]
\[ P(-0.75|\omega_2)P(\omega_2) = 0.04. \]

These are shown in Figure 9.4. Note that there is non-zero probability that the case corresponding to \( x = -0.75 \) could belong to class 2. We now turn our attention to how we can estimate this error.
% To get estimates of the error, we can
% estimate the integral as follows
% Note that 0.1 is the step size and we
% are approximating the integral using a sum.
% The decision boundary is where the two curves meet.
ind1 = find(ppxg1 >= ppxg2);
% Now find the other part.
ind2 = find(ppxg1<ppxg2);
pmis1 = sum(ppxg1(ind2))*0.1;
pmis2 = sum(ppxg2(ind1))*0.1;
errorhat = pmis1 + pmis2;

From this, we estimate the probability of error as 0.15. To get this probability, we find the shaded area under the curves as shown in Figure 9.5.

We would like to note several points regarding Bayes Decision Rule and the classification error. First, as we already saw in Example 9.3, the boundaries

© 2002 by Chapman & Hall/CRC
for the decision regions are found as the $x$ such that the following equation is satisfied:

$$P(x|\omega_i)P(\omega_i) = P(x|\omega_j)P(\omega_j); \quad i \neq j.$$  

Secondly, we can change this decision region as we will see shortly when we discuss the likelihood ratio approach to classification. If we change the decision boundary, then the error will be greater, illustrating that Bayes Decision Rule is one that minimizes the probability of misclassification [Duda and Hart, 1973].

**Example 9.4**

We continue Example 9.3, where we show what happens when we change the decision boundary to $x = -0.5$. This means that if a feature has a value of $x < -0.5$, then we classify it as belonging to class 1. Otherwise, we say it belongs to class 2. The areas under the curves that we need to calculate are shown in Figure 9.6. As we see from the following MATLAB code, where we estimate the error, that the probability of error increases.

```matlab
% Change the decision boundary.
```

© 2002 by Chapman & Hall/CRC
BOUND = -0.5;
ind1 = find(dom <= bound);
ind2 = find(dom > bound);
pmis1 = sum(ppxg1(ind2))*.1;
pmis2 = sum(ppxg2(ind1))*.1;
errorhat = pmis1 + pmis2;

This yields an estimated error of 0.20.

Bayes decision theory can address more general situations where there might be a variable cost or risk associated with classifying something incorrectly or allowing actions in addition to classifying the observation. For example, we might want to penalize the error of classifying some section of tissue in an image as cancerous when it is not, or we might want to include the action of not making a classification if our uncertainty is too great. We will provide references at the end of the chapter for those readers who require the more general treatment of statistical pattern recognition.
Chapter 9: Statistical Pattern Recognition

The likelihood ratio technique addresses the issue of variable misclassification costs in a hypothesis testing framework. This methodology does not assign an explicit cost to making an error as in the Bayes approach, but it enables us to set the amount of error we will tolerate for misclassifying one of the classes.

Recall from Chapter 6 that in hypothesis testing we have two types of errors. One type of error is when we wrongly reject the null hypothesis when it is really true. This is the Type I error. The other way we can make a wrong decision is to not reject the null hypothesis when we should. Typically, we try to control the probability of Type I error by setting a desired significance level $\alpha$, and we use this level to determine our decision boundary. We can fit our pattern recognition process into the same framework.

In the rest of this section, we consider only two classes, $\omega_1$ and $\omega_2$. First, we have to determine what class corresponds to the null hypothesis and call this the non-target class. The other class is denoted as the target class. In this book, we use $\omega_1$ to represent the target class and $\omega_2$ to represent the non-target class. The following examples should clarify these concepts.
• We are building a classifier for a military command and control system that will take features from images of objects and classify them as targets or non-targets. If an object is classified as a target, then we will destroy it. Target objects might be tanks or military trucks. Non-target objects are such things as school buses or automobiles. We would want to make sure that when we build a classifier we do not classify an object as a tank when it is really a school bus. So, we will control the amount of acceptable error in wrongly saying it (a school bus or automobile) is in the target class. This is the same as our Type I error, if we write our hypotheses as

\[ H_0 \quad \text{Object is a school bus, automobile, etc.} \]
\[ H_1 \quad \text{Object is a tank, military vehicle, etc.} \]

• Another example, where this situation arises is in medical diagnosis. Say that the doctor needs to determine whether a patient has cancer by looking at radiographic images. The doctor does not want to classify a region in the image as cancer when it is not. So, we might want to control the probability of wrongly deciding that there is cancer when there is none. However, failing to identify a cancer when it is really there is more important to control. Therefore, in this situation, the hypotheses are

\[ H_0 \quad \text{X-ray shows cancerous tissue} \]
\[ H_1 \quad \text{X-ray shows only healthy tissue} \]

The terminology that is sometimes used for the Type I error in pattern recognition is false alarms or false positives. A false alarm is wrongly classifying something as a target \((\omega_1)\), when it should be classified as non-target \((\omega_2)\). The probability of making a false alarm (or the probability of making a Type I error) is denoted as

\[ P(FA) = \alpha. \]

This probability is represented as the shaded area in Figure 9.7.

Recall that Bayes Decision Rule gives a rule that yields the minimum probability of incorrectly classifying observed patterns. We can change this boundary to obtain the desired probability of false alarm \(\alpha\). Of course, if we do this, then we must accept a higher probability of misclassification as shown in Example 9.4.

In the two class case, we can put our Bayes Decision Rule in a different form. Starting from Equation 9.7, we have our decision as

\[ P(x|\omega_1)P(\omega_1) > P(x|\omega_2)P(\omega_2) \Rightarrow x \text{ is in } \omega_1, \quad (9.9) \]
or else we classify \( x \) as belonging to \( \omega_2 \). Rearranging this inequality yields the following decision rule

\[
L_R(x) = \frac{P(x|\omega_1)P(\omega_1)}{P(x|\omega_2)P(\omega_2)} > \tau \Rightarrow x \text{ is in } \omega_1.
\]  

(9.10)

The ratio on the left of Equation 9.10 is called the likelihood ratio, and the quantity on the right is the threshold. If \( L_R > \tau \), then we decide that the case belongs to class \( \omega_1 \). If \( L_R < \tau \), then we group the observation with class \( \omega_2 \).

If we have equal priors, then the threshold is one (\( \tau = 1 \)). Thus, when \( L_R > 1 \), we assign the observation or pattern to \( \omega_1 \), and if \( L_R < 1 \), then we classify the observation as belonging to \( \omega_2 \). We can also adjust this threshold to obtain a desired probability of false alarm, as we show in Example 9.5.

**Example 9.5**

We use the class-conditional and prior probabilities of Example 9.3 to show how we can adjust the decision boundary to achieve the desired probability of false alarm. Looking at Figure 9.7, we see that

![Figure 9.7](image-url)

The shaded region shows the probability of false alarm or the probability of wrongly classifying as target (class \( \omega_1 \)) when it really belongs to class \( \omega_2 \).
where \( C \) represents the value of \( x \) that corresponds to the decision boundary. We can factor out the prior, so

\[
P(FA) = \int_{-\infty}^{C} P(x|\omega)P(\omega)dx,
\]

We then have to find the value for \( C \) such that

\[
\int_{-\infty}^{C} P(x|\omega_2)dx = \frac{P(FA)}{P(\omega_2)}.
\]

From Chapter 3, we recognize that \( C \) is a quantile. Using the probabilities in Example 9.3, we know that \( P(\omega_2) = 0.4 \) and \( P(x|\omega_2) \) is normal with mean 1 and variance of 1. If our desired \( P(FA) = 0.05 \), then

\[
\int_{-\infty}^{C} P(x|\omega_2)dx = \frac{0.05}{0.40} = 0.125.
\]

We can find the value for \( C \) using the inverse cumulative distribution function for the normal distribution. In MATLAB, this is

\[
c = \text{norminv}(0.05/0.4,1,1);
\]

This yields a decision boundary of \( x = -0.15 \).

\[
\square
\]

9.3 Evaluating the Classifier

Once we have our classifier, we need to evaluate its usefulness by measuring the percentage of observations that we correctly classify. This yields an estimate of the probability of correctly classifying cases. It is also important to report the probability of false alarms, when the application requires it (e.g., when there is a target class). We will discuss two methods for estimating the probability of correctly classifying cases and the probability of false alarm: the use of an independent test sample and cross-validation.
If our sample is large, we can divide it into a training set and a testing set. We use the training set to build our classifier and then we classify observations in the test set using our classification rule. The proportion of correctly classified observations is the estimated classification rate. Note that the classifier has not seen the patterns in the test set, so the classification rate estimated in this way is not biased. Of course, we could collect more data to be used as the independent test set, but that is often impossible or impractical.

By biased we mean that the estimated probability of correctly classifying a pattern is not overly optimistic. A common mistake that some researchers make is to build a classifier using their sample and then use the same sample to determine the proportion of observations that are correctly classified. That procedure typically yields much higher classification success rates, because the classifier has already seen the patterns. It does not provide an accurate idea of how the classifier recognizes patterns it has not seen before. However, for a thorough discussion on these issues, see Ripley [1996]. The steps for evaluating the classifier using an independent test set are outlined below.

**PROBABILITY OF CORRECT CLASSIFICATION - INDEPENDENT TEST SAMPLE**

1. Randomly separate the sample into two sets of size \( n_{\text{TEST}} \) and \( n_{\text{TRAIN}} \), where \( n_{\text{TRAIN}} + n_{\text{TEST}} = n \). One is for building the classifier (the training set), and one is used for testing the classifier (the testing set).
2. Build the classifier (e.g., Bayes Decision Rule, classification tree, etc.) using the training set.
3. Present each pattern from the test set to the classifier and obtain a class label for it. Since we know the correct class for these observations, we can count the number we have successfully classified. Denote this quantity as \( N_{\text{CC}} \).
4. The rate at which we correctly classified observations is

\[
P(\text{CC}) = \frac{N_{\text{CC}}}{n_{\text{TEST}}}.\]

The higher this proportion, the better the classifier. We illustrate this procedure in Example 9.6.

**Example 9.6**

We first load the data and then divide the data into two sets, one for building the classifier and one for testing it. We use the two species of *iris* that are hard to separate: *Iris versicolor* and *Iris virginica*. 

© 2002 by Chapman & Hall/CRC
load iris
% This loads up three matrices:
% setosa, versicolor and virginica.
% We will use the versicolor and virginica.
% To make it interesting, we will use only the
% first two features.
% Get the data for the training and testing set. We
% will just pick every other one for the testing set.
indtrain = 1:2:50;
indtest = 2:2:50;
versitest = versicolor(indtest,1:2);
versitrain = versicolor(indtrain,1:2);
virgitest = virginica(indtest,1:2);
virgitrain = virginica(indtrain,1:2);

We now build the classifier by estimating the class-conditional probabilities. We use the parametric approach, making the assumption that the class-conditional densities are multivariate normal. In this case, the estimated priors are equal.

% Get the classifier. We will assume a multivariate
% normal model for these data.
muver = mean(versitrain);
covver = cov(versitrain);
muvir = mean(virgitrain);
covvir = cov(virgitrain);

Note that the classifier is obtained using the training set only. We use the testing set to estimate the probability of correctly classifying observations.

% Present each test case to the classifier. Note that
% we are using equal priors, so the decision is based
% only on the class-conditional probabilities.
% Put all of the test data into one matrix.
X = [versitest;virgitest];
% These are the probability of x given versicolor.
pxgver = csevalnorm(X,muver,covver);
% These are the probability of x given virginica.
pxgvir = csevalnorm(X,muvir,covvir);
% Check which are correctly classified.
% In the first 25, pxgver > pxgvir are correct.
ind = find(pxgver(1:25)>pxgvir(1:25));
ncc = length(ind);
% In the last 25, pxgvir > pxgver are correct.
ind = find(pxgvir(26:50) > pxgver(26:50));
ncc = ncc + length(ind);
pcc = ncc/50;
Using this type of classifier and this partition of the learning sample, we estimate the probability of correct classification to be 0.74.

Cross-Validation

The cross-validation procedure is discussed in detail in Chapter 7. Recall that with cross-validation, we systematically partition the data into testing sets of size $k$. The $n - k$ observations are used to build the classifier, and the remaining $k$ patterns are used to test it. We continue in this way through the entire data set. When the sample is too small to partition it into a single testing and training set, then cross-validation is the recommended approach. The following is the procedure for calculating the probability of correct classification using cross-validation with $k = 1$.

**PROBABILITY OF CORRECT CLASSIFICATION - CROSS-VALIDATION**

1. Set the number of correctly classified patterns to 0, $N_{CC} = 0$.
2. Keep out one observation, call it $x_i$.
3. Build the classifier using the remaining $n - 1$ observations.
4. Present the observation $x_i$ to the classifier and obtain a class label using the classifier from the previous step.
5. If the class label is correct, then increment the number correctly classified using

\[
N_{CC} = N_{CC} + 1.
\]

6. Repeat steps 2 through 5 for each pattern in the sample.
7. The probability of correctly classifying an observation is given by

\[
P(CC) = \frac{N_{CC}}{n}.
\]

**Example 9.7**

We return to the `iris` data of Example 9.6, and we estimate the probability of correct classification using cross-validation with $k = 1$. We first set up some preliminary variables and load the data.

```matlab
load iris
% This loads up three matrices:
% setosa, versicolor and virginica.
% We will use the versicolor and virginica.
% Note that the priors are equal, so the decision is
```
% based on the class-conditional probabilities.
ncc = 0;
% We will use only the first two features of
% the iris data for our classification.
% This should make it more difficult to
% separate the classes.
% Delete 3rd and 4th features.
vr = virginica(:,1:2);
v = versicolor(:,1:2);

[nvr,d] = size(v);
[nv,d] = size(vr);
n = nv+nv;

First, we will loop through all of the versicolor observations. We build a
classifier, leaving out one pattern at a time for testing purposes. Throughout
this loop, the class-conditional probability for virginica remains the same,
so we find that first.

% Loop first through all of the patterns corresponding
to versicolor. Here correct classification
% is obtained if pxgver > pxgvir;
muv = mean(virginica);
covv = cov(virginica);

for i = 1:nver
  % Get the test point and the training set
  versitrain = versicolor;
  % This is the testing point.
  x = versitrain(i,:);
  % Delete from training set.
  % The result is the training set.
  versitrain(i,:)=[];
  muv = mean(versitrain);
covv = cov(versitrain);
  pxgver = csevalnorm(x,muv,covv);
  pxgvir = csevalnorm(x,muv,covv);
  if pxgver > pxgvir
    % then we correctly classified it
    ncc = ncc+1;
  end
end

We repeat the same procedure leaving out each virginica observation as the
test pattern.

% Loop through all of the patterns of virginica notes.
% Here correct classification is obtained when
% pxgvir > pxgver
muver = mean(versicolor);
covver = cov(versicolor);
% Those remain the same for the following.
for i = 1:nvir
    % Get the test point and training set.
    virtrain = virginica;
    x = virtrain(i,:);
    virtrain(i,:)=[];
    muvir = mean(virtrain);
    covvir = cov(virtrain);
    pxgver = csevalnorm(x,muver,covver);
    pxgvir = csevalnorm(x,muvir,covvir);
    if pxgvir > pxgver
        % then we correctly classified it
        ncc = ncc+1;
    end
end

Finally, the probability of correct classification is estimated using

\[ pcc = \frac{ncc}{n}; \]

The estimated probability of correct classification for the iris data using cross-validation is 0.68.

Receiver Operating Characteristic (ROC) Curve

We now turn our attention to how we can use cross-validation to evaluate a classifier that uses the likelihood approach with varying decision thresholds \( \tau_c \). It would be useful to understand how the classifier performs for various thresholds (corresponding to the probability of false alarm) of the likelihood ratio. This will tell us what performance degradation we have (in terms of correctly classifying the target class) if we limit the probability of false alarm to some level.

We start by dividing the sample into two sets: one with all of the target observations and one with the non-target patterns. Denote the observations as follows

\[ x^{(1)}_i \Rightarrow \text{Target pattern (\( \omega_1 \))} \]
\[ x^{(2)}_i \Rightarrow \text{Non-target pattern (\( \omega_2 \))}. \]

Let \( n_1 \) represent the number of target observations (class \( \omega_1 \)) and \( n_2 \) denote the number of non-target (class \( \omega_2 \)) patterns. We work first with the non-target observations to determine the threshold we need to get a desired proba-
bility of false alarm. Once we have the threshold, we can determine the probability of correctly classifying the observations belonging to the target class.

Before we go on to describe the receiver operating characteristic (ROC) curve, we first describe some terminology. For any boundary we might set for the decision regions, we are likely to make mistakes in classifying cases. There will be some target patterns that we correctly classify as targets and some we misclassify as non-targets. Similarly, there will be non-target patterns that are correctly classified as non-targets and some that are misclassified as targets. This is summarized as follows:

- **True Positives - TP**: This is the fraction of patterns correctly classified as target cases.
- **False Positives - FP**: This is the fraction of non-target patterns incorrectly classified as target cases.
- **True Negatives - TN**: This is the fraction of non-target cases correctly classified as non-target.
- **False Negatives - FN**: This is the fraction of target cases incorrectly classified as non-target.

In our previous terminology, the false positives (FP) correspond to the false alarms. Figure 9.8 shows these areas for a given decision boundary.

A **ROC curve** is a plot of the true positive rate against the false positive rate. ROC curves are used primarily in signal detection and medical diagnosis [Egan, 1975; Lusted, 1971; McNeil, et. al., 1975; Hanley and McNeil, 1983; Hanley and Hajian-Tilaki, 1997]. In their terminology, the true positive rate is also called the sensitivity. **Sensitivity** is the probability that a classifier will classify a pattern as a target when it really is a target. **Specificity** is the probability that a classifier will correctly classify the true non-target cases. Therefore, we see that a ROC curve is also a plot of sensitivity against 1 minus specificity.

One of the purposes of a ROC curve is to measure the discriminating power of the classifier. It is used in the medical community to evaluate the diagnostic power of tests for diseases. By looking at a ROC curve, we can understand the following about a classifier:

- It shows the trade-off between the probability of correctly classifying the target class (sensitivity) and the false alarm rate (1 – specificity).
- The area under the ROC curve can be used to compare the performance of classifiers.

We now show in more detail how to construct a ROC curve. Recall that the likelihood ratio is given by
We start off by forming the likelihood ratios using the non-target observations and cross-validation to get the distribution of the likelihood ratios when the class membership is truly $\omega_2$. We use these likelihood ratios to set the threshold that will give us a specific probability of false alarm.

Once we have the thresholds, the next step is to determine the rate at which we correctly classify the target cases. We first form the likelihood ratio for each target observation using cross-validation, yielding a distribution of likelihood ratios for the target class. For each given threshold, we can determine the number of target observations that would be correctly classified by counting the number of $L_R$ that are greater than that threshold. These steps are described in detail in the following procedure.

**CROSS-VALIDATION FOR SPECIFIED FALSE ALARM RATE**

1. Given observations with class labels $\omega_1$ (target) and $\omega_2$ (non-target), set desired probabilities of false alarm and a value for $k$. 

\[
L_R(x) = \frac{P(x|\omega_1)}{P(x|\omega_2)}. 
\]
2. Leave \( k \) points out of the non-target class to form a set of test cases denoted by \( TEST \). We denote cases belonging to class \( \omega_2 \) as \( x^{(2)}_i \).

3. Estimate the class-conditional probabilities using the remaining \( n_2 - k \) non-target cases and the \( n_1 \) target cases.

4. For each of those \( k \) observations, form the likelihood ratios

\[
L_R(x^{(2)}_i) = \frac{P(x^{(2)}_i | \omega_1)}{P(x^{(2)}_i | \omega_2)}; \quad x^{(2)}_i \text{ in } TEST.
\]

5. Repeat steps 2 through 4 using all of the non-target cases.

6. Order the likelihood ratios for the non-target class.

7. For each probability of false alarm, find the threshold that yields that value. For example, if the \( P(FA) = 0.1 \), then the threshold is given by the quantile \( q_{0.1} \) of the likelihood ratios. Note that higher values of the likelihood ratios indicate the target class. We now have an array of thresholds corresponding to each probability of false alarm.

8. Leave \( k \) points out of the target class to form a set of test cases denoted by \( TEST \). We denote cases belonging to \( \omega_1 \) by \( x^{(1)}_i \).

9. Estimate the class-conditional probabilities using the remaining \( n_1 - k \) target cases and the \( n_2 \) non-target cases.

10. For each of those \( k \) observations, form the likelihood ratios

\[
L_R(x^{(1)}_i) = \frac{P(x^{(1)}_i | \omega_1)}{P(x^{(1)}_i | \omega_2)}; \quad x^{(1)}_i \text{ in } TEST.
\]

11. Repeat steps 8 through 10 using all of the target cases.

12. Order the likelihood ratios for the target class.

13. For each threshold and probability of false alarm, find the proportion of target cases that are correctly classified to obtain the \( P(\text{CC}_{\text{Target}}) \). If the likelihood ratios \( L_R(x^{(1)}_i) \) are sorted, then this would be the number of cases that are greater than the threshold.

This procedure yields the rate at which the target class is correctly classified for a given probability of false alarm. We show in Example 9.8 how to implement this procedure in MATLAB and plot the results in a ROC curve.

**Example 9.8**

In this example, we illustrate the cross-validation procedure and ROC curve using the univariate model of Example 9.3. We first use MATLAB to generate some data.
% Generate some data, use the model in Example 9.3.
% \( p(x|w1) \sim N(-1,1), p(w1) = 0.6 \)
% \( p(x|w2) \sim N(1,1), p(w2) = 0.4; \)
% Generate the random variables.
\[
\begin{align*}
n & = 1000; \\
u & = \text{rand}(1,n); \% \text{find out what class they are from} \\
n1 & = \text{length}(\text{find}(u \leq 0.6)); \% \# \text{in target class} \\
n2 & = n-n1; \\
x1 & = \text{randn}(1,n1) - 1; \\
x2 & = \text{randn}(1,n2) + 1;
\end{align*}
\]

We set up some arrays to store the likelihood ratios and estimated probabilities. We also specify the values for the \( P(FA) \). For each \( P(FA) \), we will be estimating the probability of correctly classifying objects from the target class.

\[
\begin{align*}
% \text{Set up some arrays to store things.} \\
\text{lr1} & = \text{zeros}(1,n1); \\
\text{lr2} & = \text{zeros}(1,n2); \\
\text{pfa} & = 0.01:.01:0.99; \\
\text{pcc} & = \text{zeros(size(pfa))};
\end{align*}
\]

We now implement steps 2 through 7 of the cross-validation procedure. This is the part where we find the thresholds that provide the desired probability of false alarm.

\[
\begin{align*}
% \text{First find the threshold corresponding} \\
% \text{to each false alarm rate.} \\
% \text{Build classifier using target data.} \\
\text{mu1} & = \text{mean}(x1); \\
\text{var1} & = \text{cov}(x1); \\
% \text{Do cross-validation on non-target class.} \\
\text{for} \ i = 1:n2 \\
\text{train} & = x2; \\
\text{test} & = x2(i); \\
\text{train}(i) & = []; \\
\text{mu2} & = \text{mean(train)}; \\
\text{var2} & = \text{cov(train)}; \\
\text{lr2}(i) & = \text{csevalnorm(test,mu1,var1)}/... \\
& \quad \text{csevalnorm(test,mu2,var2);}
\text{end}
\end{align*}
\]

\[
\begin{align*}
% \text{sort the likelihood ratios for the non-target class} \\
\text{lr2} & = \text{sort(lr2)}; \\
% \text{Get the thresholds.} \\
\text{thresh} & = \text{zeros(size(pfa))}; \\
\text{for} \ i = 1:length(pfa) \\
\text{thresh}(i) & = \text{cquantiles(lr2,1-pfa(i))};
\text{end}
\end{align*}
\]
For the given thresholds, we now find the probability of correctly classifying the target cases. This corresponds to steps 8 through 13.

% Now find the probability of correctly classifying targets.
mu2 = mean(x2);
var2 = cov(x2);
% Do cross-validation on target class.
for i = 1:n1
    train = x1;
    test = x1(i);
    train(i) = [];
    mu1 = mean(train);
    var1 = cov(train);
    lr1(i) = csevalnorm(test,mu1,var1)./...
          csevalnorm(test,mu2,var2);
end
% Find the actual pcc.
for i = 1:length(pfa)
    pcc(i) = length(find(lr1 >= thresh(i)));
end
pcc = pcc/n1;

The ROC curve is given in Figure 9.9. We estimate the area under the curve as 0.91, using

area = sum(pcc)*.01;

9.4 Classification Trees

In this section, we present another technique for pattern recognition called classification trees. Our treatment of classification trees follows that in the book called Classification and Regression Trees by Breiman, Friedman, Olshen and Stone [1984]. For ease of exposition, we do not include the MATLAB code for the classification tree in the main body of the text, but we do include it in Appendix D. There are several main functions that we provide to work with trees, and these are summarized in Table 9.1. We will be using these functions in the text when we discuss the classification tree methodology.

While Bayes decision theory yields a classification rule that is intuitively appealing, it does not provide insights about the structure or the nature of the classification rule or help us determine what features are important. Classification trees can yield complex decision boundaries, and they are appropriate for ordered data, categorical data or a mixture of the two types. In this book,
we will be concerned only with the case where all features are continuous random variables. The interested reader is referred to Breiman, et al. [1984], Webb [1999], and Duda, Hart and Stork [2001] for more information on the other cases.
A decision or classification tree represents a multi-stage decision process, where a binary decision is made at each stage. The tree is made up of nodes and branches, with nodes being designated as an internal or a terminal node. Internal nodes are ones that split into two children, while terminal nodes do not have any children. A terminal node has a class label associated with it, such that observations that fall into the particular terminal node are assigned to that class.

To use a classification tree, a feature vector is presented to the tree. If the value for a feature is less than some number, then the decision is to move to the left child. If the answer to that question is no, then we move to the right child. We continue in that manner until we reach one of the terminal nodes, and the class label that corresponds to the terminal node is the one that is assigned to the pattern. We illustrate this with a simple example.

![Classification Tree Diagram]

**Figure 9.10**
This simple classification tree for two classes is used in Example 9.9. Here we make decisions based on two features, $x_1$ and $x_2$. 

© 2002 by Chapman & Hall/CRC
Example 9.9
We show a simple classification tree in Figure 9.10, where we are concerned with only two features. Note that all internal nodes have two children and a splitting rule. The split can occur on either variable, with observations that are less than that value being assigned to the left child and the rest going to the right child. Thus, at node 1, any observation where the first feature is less than 5 would go to the left child. When an observation stops at one of the terminal nodes, it is assigned to the corresponding class for that node. We illustrate these concepts with several cases. Say that we have a feature vector given by \( \mathbf{x} = (4, 6) \), then passing this down the tree, we get

\[
\text{node 1 } \rightarrow \text{ node 2 } \Rightarrow \omega_1 .
\]

If our feature vector is \( \mathbf{x} = (6, 6) \), then we travel the tree as follows:

\[
\text{node 1 } \rightarrow \text{ node 3 } \rightarrow \text{ node 4 } \rightarrow \text{ node 6 } \Rightarrow \omega_2 .
\]

For a feature vector given by \( \mathbf{x} = (10, 12) \), we have

\[
\text{node 1 } \rightarrow \text{ node 3 } \rightarrow \text{ node 5 } \Rightarrow \omega_2 .
\]

We give a brief overview of the steps needed to create a tree classifier and then explain each one in detail. To start the process, we must grow an overly large tree using a criterion that will give us optimal splits for the tree. It turns out that these large trees fit the training data set very well. However, they do not generalize, so the rate at which we correctly classify new patterns is low. The proposed solution [Breiman, et al., 1984] to this problem is to continually prune the large tree using a minimal cost complexity criterion to get a sequence of sub-trees. The final step is to choose a tree that is the ‘right size’ using cross-validation or an independent test sample. These three main procedures are described in the remainder of this section. However, to make things easier for the reader, we first provide the notation that will be used to describe classification trees.

**CLASSIFICATION TREES - NOTATION**

- \( \mathbf{L} \) denotes a learning set made up of observed feature vectors and their class label.
- \( J \) denotes the number of classes.
- \( T \) is a classification tree.
- \( t \) represents a node in the tree.
$t_L$ and $t_R$ are the left and right child nodes.

$\{t_1\}$ is the tree containing only the root node.

$T_1$ is a branch of tree $T$ starting at node $t$.

$\hat{T}$ is the set of terminal nodes in the tree.

$|\hat{T}|$ is the number of terminal nodes in tree $T$.

$t_k^*$ is the node that is the weakest link in tree $T_k$.

$n$ is the total number of observations in the learning set.

$n_j$ is the number of observations in the learning set that belong to the $j$-th class $\omega_j$, $j = 1, ..., J$.

$n(t)$ is the number of observations that fall into node $t$.

$n_j(t)$ is the number of observations at node $t$ that belong to class $\omega_j$.

$\pi_j$ is the prior probability that an observation belongs to class $\omega_j$. This can be estimated from the data as

$$\hat{\pi}_j = \frac{n_j}{n}.$$  \hfill (9.11)

$p(\omega_j, t)$ represents the joint probability that an observation will be in node $t$ and it will belong to class $\omega_j$. It is calculated using

$$p(\omega_j, t) = \frac{\pi_j n_j(t)}{n_j}. \hfill (9.12)$$

$p(t)$ is the probability that an observation falls into node $t$ and is given by

$$p(t) = \sum_{j=1}^{J} p(\omega_j, t). \hfill (9.13)$$

$p(\omega_j | t)$ denotes the probability that an observation is in class $\omega_j$ given it is in node $t$. This is calculated from

$$p(\omega_j | t) = \frac{p(\omega_j, t)}{p(t)}. \hfill (9.14)$$

$r(t)$ represents the resubstitution estimate of the probability of misclassification for node $t$ and a given classification into class $\omega_j$. This
is found by subtracting the maximum conditional probability \( p(\omega_j | t) \) for the node from 1:

\[
r(t) = 1 - \max_j \{ p(\omega_j | t) \} .
\]  

(9.15)

\( R(t) \) is the resubstitution estimate of risk for node \( t \). This is

\[
R(t) = r(t)p(t) .
\]  

(9.16)

\( R(T) \) denotes a resubstitution estimate of the overall misclassification rate for a tree \( T \). This can be calculated using every terminal node in the tree as follows

\[
R(T) = \sum_{i \in \hat{t}} r(t)p(t) = \sum_{i \in \hat{t}} R(t) .
\]  

(9.17)

\( \alpha \) is the complexity parameter.

\( i(t) \) denotes a measure of impurity at node \( t \).

\( \Delta i(s, t) \) represents the decrease in impurity and indicates the goodness of the split \( s \) at node \( t \). This is given by

\[
\Delta i(s, t) = i(t) - p_R i(t_R) - p_L i(t_L) .
\]  

(9.18)

\( p_L \) and \( p_R \) are the proportion of data that are sent to the left and right child nodes by the split \( s \).

**Growing the Tree**

The idea behind binary classification trees is to split the \( d \)-dimensional space into smaller and smaller partitions, such that the partitions become purer in terms of the class membership. In other words, we are seeking partitions where the majority of the members belong to one class. To illustrate these ideas, we use a simple example where we have patterns from two classes, each one containing two features, \( x_1 \) and \( x_2 \). How we obtain these data are discussed in the following example.

**Example 9.10**

We use synthetic data to illustrate the concepts of classification trees. There are two classes, and we generate 50 points from each class. From Figure 9.11, we see that each class is a two term mixture of bivariate uniform random variables.
% This shows how to generate the data that will be used
% to illustrate classification trees.
deln = 25;
data(1:deln,:) = rand(deln,2)+.5;
so=deln+1; sf = 2*deln;
data(so:sf,:) = rand(deln,2)-.5;
so=sf+1; sf = 3*deln;
data(so:sf,1) = rand(deln,1)-.5;
data(so:sf,2) = rand(deln,1)+.5;
so=sf+1; sf = 4*deln;
data(so:sf,1) = rand(deln,1)+.5;
data(so:sf,2) = rand(deln,1)-.5;

A scatterplot of these data is given in Figure 9.11. One class is depicted by the
"*" and the other is represented by the ‘o’. These data are available in the file
called cartdata, so the user can load them and reproduce the next several
examples.

FIGURE 9.11
This shows a scatterplot of the data that will be used in our classification tree examples.
Data that belong to class 1 are shown by the ‘*’, and those that belong to class 2 are denoted
by an ‘o’.
To grow a tree, we need to have some criterion to help us decide how to split the nodes. We also need a rule that will tell us when to stop splitting the nodes, at which point we are finished growing the tree. The stopping rule can be quite simple, since we first grow an overly large tree. One possible choice is to continue splitting terminal nodes until each one contains observations from the same class, in which case some nodes might have only one observation in the node. Another option is to continue splitting nodes until there is some maximum number of observations left in a node or the terminal node is pure (all observations belong to one class). Recommended values for the maximum number of observations left in a terminal node are between 1 and 5.

We now discuss the splitting rule in more detail. When we split a node, our goal is to find a split that reduces the impurity in some manner. So, we need a measure of impurity \( i(t) \) for a node \( t \). Breiman, et al. [1984] discuss several possibilities, one of which is called the Gini diversity index. This is the one we will use in our implementation of classification trees. The Gini index is given by

\[
i(t) = \sum_{i \neq j} p(\omega_i|t)p(\omega_j|t),
\]

which can also be written as

\[
i(t) = 1 - \sum_{j=1}^{J} p^2(\omega_j|t).
\]

Equation 9.20 is the one we code in the MATLAB function `csgrowc` for growing classification trees.

Before continuing with our description of the splitting process, we first note that our use of the term ‘best’ does not necessarily mean that the split we find is the optimal one out of all the infinite possible splits. To grow a tree at a given node, we search for the best split (in terms of decreasing the node impurity) by first searching through each variable or feature. We have \( d \) possible best splits for a node (one for each feature), and we choose the best one out of these \( d \) splits. The problem now is to search through the infinite number of possible splits. We can limit our search by using the following convention. For all feature vectors in our learning sample, we search for the best split at the \( k \)-th feature by proposing splits that are halfway between consecutive values for that feature. For each proposed split, we evaluate the impurity criterion and choose the split that yields the largest decrease in impurity.

Once we have finished growing our tree, we must assign class labels to the terminal nodes and determine the corresponding misclassification rate. It makes sense to assign the class label to a node according to the likelihood that it is in class \( \omega_j \) given that it fell into node \( t \). This is the posterior probability

\[
p(\omega_j|t) = \frac{\sum_{i=1}^{n} I(y_i = j)}{n}.
\]
given by Equation 9.14. So, using Bayes decision theory, we would classify an observation at node $t$ with the class $\omega_j$ that has the highest posterior probability. The error in our classification is then given by Equation 9.15. We summarize the steps for growing a classification tree in the following procedure. In the learning set, each observation will be a row in the matrix $X$, so this matrix has dimensionality $n \times (d + 1)$, representing $d$ features and a class label. The measured value of the $k$-th feature for the $i$-th observation is denoted by $x_{ik}$.

**PROCEDURE - GROWING A TREE**

1. Determine the maximum number of observations $n_{\text{max}}$ that will be allowed in a terminal node.
2. Determine the prior probabilities of class membership $\pi_j$. These can be estimated from the data (Equation 9.11), or they can be based on prior knowledge of the application.
3. If a terminal node in the current tree contains more than the maximum allowed observations and contains observations from several classes, then search for the best split. For each feature $k$,
   a. Put the $x_{ik}$ in ascending order to give the ordered values $x_{(i)k}$.
   b. Determine all splits $s_{(i)k}$ in the $k$-th feature using
      $$s_{(i)k} = x_{(i)k} + (x_{(i)k} - x_{(i+1)k})/2$$
   c. For each proposed split, evaluate the impurity function $i(t)$ and the goodness of the split using Equations 9.20 and 9.18.
   d. Pick the best, which is the one that yields the largest decrease in impurity.
4. Out of the $k$ best splits in step 3, split the node on the variable that yields the best overall split.
5. For that split found in step 4, determine the observations that go to the left child and those that go to the right child.
6. Repeat steps 3 through 5 until each terminal node satisfies the stopping rule (has observations from only one class or has the maximum allowed cases in the node).

**Example 9.11**

In this example, we grow the initial large tree on the data set given in the previous example. We stop growing the tree when each terminal node has a maximum of 5 observations or the node is pure. We first load the data that we generated in the previous example. This file contains the data matrix, the inputs to the function `csgrowc`, and the resulting tree.
load cartdata
% Loads up data.
% Inputs to function - csgrowc.
maxn = 5; % maximum number in terminal nodes
clas = [1 2]; % class labels
pies = [0.5 0.5]; % optional prior probabilities
Nk = [50, 50]; % number in each class

The following MATLAB commands grow the initial tree and plot the results in Figure 9.12.

```
    tree = csgrowc(X,maxn,clas,Nk,pies);
csporetrec(tree)
```

We see from Figure 9.12, that the tree has partitioned the feature space into eight decision regions or eight terminal nodes.

![Classification Tree Diagram]

**FIGURE 9.12**
This is the classification tree for the data shown in Figure 9.11. This tree partitions the feature space into 8 decision regions.
Pruning the Tree

Recall that the classification error for a node is given by Equation 9.15. If we grow a tree until each terminal node contains observations from only one class, then the error rate will be zero. Therefore, if we use the classification error as a stopping criterion or as a measure of when we have a good tree, then we would grow the tree until there are pure nodes. However, as we mentioned before, this procedure over fits the data and the classification tree will not generalize well to new patterns. The suggestion made in Breiman, et al. [1984] is to grow an overly large tree, denoted by \( T_{\text{max}} \), and then to find a nested sequence of subtrees by successively pruning branches of the tree. The best tree from this sequence is chosen based on the misclassification rate estimated by cross-validation or an independent test sample. We describe the two approaches after we discuss how to prune the tree.

The pruning procedure uses the misclassification rates along with a cost for the complexity of the tree. The complexity of the tree is based on the number of terminal nodes in a subtree or branch. The cost complexity measure is defined as

\[
R_\alpha(T) = R(T) + \alpha \left\lvert \tilde{T} \right\rvert; \quad \alpha \geq 0. \tag{9.21}
\]

We look for a tree that minimizes the cost complexity given by Equation 9.21. The \( \alpha \) is a parameter that represents the complexity cost per terminal node. If we have a large tree where every terminal node contains observations from only one class, then \( R(T) \) will be zero. However, there will be a penalty paid because of the complexity, and the cost complexity measure becomes

\[
R_\alpha(T) = \alpha \left\lvert \tilde{T} \right\rvert.
\]

If \( \alpha \) is small, then the penalty for having a complex tree is small, and the resulting tree is large. The tree that minimizes \( R_\alpha(T) \) will tend to have few nodes and large \( \alpha \).

Before we go further with our explanation of the pruning procedure, we need to define what we mean by the branches of a tree. A branch \( T_t \) of a tree \( T \) consists of the node \( t \) and all its descendent nodes. When we prune or delete this branch, then we remove all descendent nodes of \( t \), leaving the branch root node \( t \). For example, using the tree in Figure 9.10, the branch corresponding to node 3 contains nodes 3, 4, 5, 6, and 7, as shown in Figure 9.13. If we delete that branch, then the remaining nodes are 1, 2, and 3.

Minimal complexity pruning searches for the branches that have the weakest link, which we then delete from the tree. The pruning process produces a sequence of subtrees with fewer terminal nodes and decreasing complexity.

We start with our overly large tree and denote this tree as \( T_{\text{max}} \). We are searching for a finite sequence of subtrees such that
Note that the starting point for this sequence is the tree $T_1$. Tree $T_1$ is found in a way that is different from the other subtrees in the sequence. We start off with $T_{max}$, and we look at the misclassification rate for the terminal node pairs (both sibling nodes are terminal nodes) in the tree. It is shown in Breiman, et al. [1984] that

$$T_{max} > T_1 > T_2 > \ldots > T_K = \{ t_1 \}.$$  

Note that the starting point for this sequence is the tree $T_1$. Tree $T_1$ is found in a way that is different from the other subtrees in the sequence. We start off with $T_{max}$, and we look at the misclassification rate for the terminal node pairs (both sibling nodes are terminal nodes) in the tree. It is shown in Breiman, et al. [1984] that

$$R(t) \geq R(t_1) + R(t_k). \quad (9.22)$$

Equation 9.22 indicates that the misclassification error in the parent node is greater than or equal to the sum of the error in the children. We search through the terminal node pairs in $T_{max}$ looking for nodes that satisfy

$$R(t) = R(t_1) + R(t_k), \quad (9.23)$$

and we prune off those nodes. These splits are ones that do not improve the overall misclassification rate for the descendants of node $t$. Once we have completed this step, the resulting tree is $T_1$. 

© 2002 by Chapman & Hall/CRC
There is a continuum of values for the complexity parameter \( \alpha \), but if a tree \( T(\alpha) \) is a tree that minimizes \( R_\alpha(T) \) for a given \( \alpha \), then it will continue to minimize it until a jump point for \( \alpha \) is reached. Thus, we will be looking for a sequence of complexity values \( \alpha \) and the trees that minimize the cost complexity measure for each level. Once we have our tree \( T_1 \), we start pruning off the branches that have the weakest link. To find the weakest link, we first define a function on a tree as follows

\[
g_k(t) = \frac{R(t) - R(T_{kl})}{|T_{kl}| - 1} \quad \text{for an internal node } t \text{,} \tag{9.24}
\]

where \( T_{kl} \) is the branch \( T \), corresponding to the internal node \( t \) of subtree \( T_k \). From Equation 9.24, for every internal node in tree \( T_k \), we determine the value for \( g_k(t) \). We define the weakest link \( t_k^* \) in tree \( T_k \) as the internal node \( t \) that minimizes Equation 9.24,

\[
g_k(t_k^*) = \min_t \{g_k(t)\} \tag{9.25}
\]

Once we have the weakest link, we prune the branch defined by that node. The new tree in the sequence is obtained by

\[
T_{k+1} = T_k - T_{k,t_k^*} \tag{9.26}
\]

where the subtraction in Equation 9.26 indicates the pruning process. We set the value of the complexity parameter to

\[
\alpha_{k+1} = g_k(t_k^*) \tag{9.27}
\]

The result of this pruning process will be a decreasing sequence of trees,

\[T_{max} > T_1 > T_2 > \ldots > T_K = \{t_1\}, \]

along with an increasing sequence of values for the complexity parameter

\[0 = \alpha_1 < \ldots < \alpha_k < \alpha_{k+1} < \ldots < \alpha_K.\]

We need the following key fact when we describe the procedure for choosing the best tree from the sequence of subtrees:
For \( k \geq 1 \), the tree \( T_k \) is the minimal cost complexity tree for the interval \( \alpha_k \leq \alpha < \alpha_{k+1} \), and

\[
T(\alpha) = T(\alpha_k) = T_k.
\]

**PROCEDURE - PRUNING THE TREE**

1. Start with a large tree \( T_{max} \).
2. Find the first tree in the sequence \( T_1 \) by searching through all terminal node pairs. For each of these pairs, if \( R(t) = R(t_L) + R(t_R) \), then delete nodes \( t_L \) and \( t_R \).
3. For all internal nodes in the current tree, calculate \( g_\alpha(t) \) as given in Equation 9.24.
4. The weakest link is the node that has the smallest value for \( g_\alpha(t) \).
5. Prune off the branch that has the weakest link.
6. Repeat steps 3 through 5 until only the root node is left.

**Example 9.12**

We continue with the same data set from the previous examples. We apply the pruning procedure to the large tree obtained in Example 9.11. The pruning function for classification trees is called `csprunec`. The input argument is a tree, and the output argument is a cell array of subtrees, where the first tree corresponds to tree \( T_1 \) and the last tree corresponds to the root node.

```matlab
treeseq = csprunec(tree);
K = length(treeseq);
alpha = zeros(1,K);
% Find the sequence of alphas.
% Note that the root node corresponds to K,
% the last one in the sequence.
for i = 1:K
    alpha(i) = treeseq{i}.alpha;
end
```

The resulting sequence for \( \alpha \) is

\[
\alpha = 0, 0.01, 0.03, 0.07, 0.08, 0.10.
\]

We see that as \( k \) increases (or, equivalently, the complexity of the tree decreases), the complexity parameter increases. We plot two of the subtrees in Figures 9.14 and 9.15. Note that tree \( T_1 \) with \( \alpha = 0.08 \) has fewer terminal nodes than tree \( T_1 \) with \( \alpha = 0.03 \).
In the previous section, we discussed the importance of using independent test data to evaluate the performance of our classifier. We now use the same procedures to help us choose the right size tree. It makes sense to choose a tree that yields the smallest true misclassification cost, but we need a way to estimate this.

The values for misclassification rates that we get when constructing a tree are really estimates using the learning sample. We would like to get less biased estimates of the true misclassification costs, so we can use these values to choose the tree that has the smallest estimated misclassification rate. We can get these estimates using either an independent test sample or cross-validation. In this text, we cover the situation where there is a unit cost for misclassification and the priors are estimated from the data. For a general treatment of the procedure, the reader is referred to Breiman, et al. [1984].
Using an Independent Test Sample

We first describe the independent test sample case, because it is easier to understand. The notation that we use is summarized below.

**NOTATION - INDEPENDENT TEST SAMPLE METHOD**

- \( L_1 \) is the subset of the learning sample \( L \) that will be used for building the tree.
- \( L_2 \) is the subset of the learning sample \( L \) that will be used for testing the tree and choosing the best subtree.
- \( n^{(2)} \) is the number of cases in \( L_2 \).
- \( n^{(2)}_j \) is the number of observations in \( L_2 \) that belong to class \( \omega_j \).

**Selecting the Best Tree Using an Independent Test Sample**

We first describe the independent test sample case, because it is easier to understand. The notation that we use is summarized below.

**FIGURE 9.15**

Here is the subtree corresponding to \( k = 3 \) from Example 9.12. For this tree, \( \alpha = 0.03 \).
\( n_{ij}^{(2)} \) is the number of observations in \( L_2 \) that belong to class \( \omega_i \) that were classified as belonging to class \( \omega_j \).

\( \hat{Q}^{TS} (\omega_i|\omega_j) \) represents the estimate of the probability that a case belonging to class \( \omega_i \) is classified as belonging to class \( \omega_j \), using the independent test sample method.

\( \hat{R}^{TS} (\omega_i) \) is an estimate of the expected cost of misclassifying patterns in class \( \omega_i \), using the independent test sample.

\( \hat{R}^{TS} (T_k) \) is the estimate of the expected misclassification cost for the tree represented by \( T_k \) using the independent test sample method.

If our learning sample is large enough, we can divide it into two sets, one for building the tree and one for estimating the misclassification costs. We use the set \( L_1 \) to build the tree \( T_{\text{max}} \) and to obtain the sequence of pruned subtrees. This means that the trees have never seen any of the cases in the second sample \( L_2 \). So, we present all observations in \( L_2 \) to each of the trees to obtain an honest estimate of the true misclassification rate of each tree.

Since we have unit cost and estimated priors given by Equation 9.11, we can write \( \hat{Q}^{TS} (\omega_i|\omega_j) \) as

\[
\hat{Q}^{TS} (\omega_i|\omega_j) = \frac{n_{ij}^{(2)}}{n_{ij}^{(2)}},
\]

(9.28)

Note that if it happens that the number of cases belonging to class \( \omega_j \) is zero (i.e., \( n_{ij}^{(2)} = 0 \)), then we set \( \hat{Q}^{TS} (\omega_i|\omega_j) = 0 \). We can see from Equation 9.28 that this estimate is given by the proportion of cases that belong to class \( \omega_j \) that are classified as belonging to class \( \omega_i \).

The total proportion of observations belonging to class \( \omega_j \) that are misclassified is given by

\[
\hat{R}^{TS} (\omega_j) = \sum_i \hat{Q}^{TS} (\omega_i|\omega_j).
\]

(9.29)

This is our estimate of the expected misclassification cost for class \( \omega_j \). Finally, we use the total proportion of test cases misclassified by tree \( T \) as our estimate of the misclassification cost for the tree classifier. This can be calculated using

\[
\hat{R}^{TS} (T_k) = \frac{1}{n^{(2)}} \sum_{i,j} n_{ij}^{(2)}.
\]

(9.30)

Equation 9.30 is easily calculated by simply counting the number of misclassified observations from \( L_2 \) and dividing by the total number of cases in the test sample.
The rule for picking the best subtree requires one more quantity. This is the standard error of our estimate of the misclassification cost for the trees. In our case, the prior probabilities are estimated from the data, and we have unit cost for misclassification. Thus, the standard error is estimated by

\[
\hat{SE}(\hat{R}^{TS}(T_k)) = \left\{ \hat{R}^{TS}(T_k)(1 - \hat{R}^{TS}(T_k)) / n^{(2)} \right\}^{1/2},
\]  

(9.31)

where \( n^{(2)} \) is the number of cases in the independent test sample.

To choose the right size subtree, Breiman, et al. [1984] recommend the following. First find the tree that gives the smallest value for the estimated misclassification error. Then we add the standard error given by Equation 9.31 to that misclassification error. Find the smallest tree (the tree with the largest subscript \( k \)) such that its misclassification cost is less than the minimum misclassification plus its standard error. In essence, we are choosing the least complex tree whose accuracy is comparable to the tree yielding the minimum misclassification rate.

**PROCEDURE - CHOOSING THE BEST SUBTREE - TEST SAMPLE METHOD**

1. Randomly partition the learning set into two parts, \( L_1 \) and \( L_2 \) or obtain an independent test set by randomly sampling from the population.
2. Using \( L_1 \), grow a large tree \( T_{max} \).
3. Prune \( T_{max} \) to get the sequence of subtrees \( T_k \).
4. For each tree in the sequence, take the cases in \( L_2 \) and present them to the tree.
5. Count the number of cases that are misclassified.
6. Calculate the estimate for \( \hat{R}^{TS}(T_k) \) using Equation 9.30.
7. Repeat steps 4 through 6 for each tree in the sequence.
8. Find the minimum error

\[
\hat{R}_{min}^{TS} = \min_k \{ \hat{R}^{TS}(T_k) \}.
\]

9. Calculate the standard error in the estimate of \( \hat{R}_{min}^{TS} \) using Equation 9.31.
10. Add the standard error to \( \hat{R}_{min}^{TS} \) to get

\[
\hat{R}_{min}^{TS} + \hat{SE}(\hat{R}_{min}^{TS}).
\]
11. Find the tree with the fewest number of nodes (or equivalently, the largest $k$) such that its misclassification error is less than the amount found in step 10.

**Example 9.13**

We implement this procedure using the sequence of trees found in Example 9.12. Since our sample was small, only 100 points, we will not divide this into a testing and training set. Instead, we will simply generate another set of random variables from the same distribution. The testing set we use in this example is contained in the file `cartdata`. First we generate the data that belong to class 1.

```matlab
% Priors are 0.5 for both classes.
% Generate 200 data points for testing.
% Find the number in each class.
n = 200;
u = rand(1,n);
% Find the number in class 1.
N1 = length(find(u<=0.5));
N2 = n - N1;
% Generate the ones for class 1
% Half are upper right corner, half are lower left
data1 = zeros(N1,2);
u = rand(1,N1);
N11 = length(find(u<=0.5));
N12 = N1 - N11;
data1(1:N11,:) = rand(N11,2)+.5;
data1(N11+1:N1,:) = rand(N12,2)-.5;
```

Next we generate the data points for class 2.

```matlab
% Generate the ones for class 2.
% Half are in lower right corner, half are upper left.
data2 = rand(N2,2);
u = rand(1,N2);
N21 = length(find(u<=0.5));
N22 = N2 - N21;
data2(1:N21,1) = rand(N21,1)-.5;
data2(1:N21,2) = rand(N21,1)+.5;
data2(N21+1:N2,1) = rand(N22,1)+.5;
data2(N21+1:N2,2) = rand(N22,1)-.5;
```

Now we determine the misclassification rate for each tree in the sequence using the independent test cases. The function `cstreec` returns the class label for a given feature vector.

```matlab
% Now check the trees using independent test cases in data1 and data2.
```
% Keep track of the ones misclassified.
K = length(treeseq);
Rk = zeros(1,K-1); % we do not check the root
for k = 1:K-1
    nmis = 0;
treek = treeseq{k};
    % loop through the cases from class 1
    for i = 1:n1
        [clas,pclass,node]=cstreec(data1(i,:),treek);
        if clas ~= 1
            nmis = nmis+1; % misclassified
        end
    end
    % Loop through class 2 cases
    for i = 1:n2
        [clas,pclass,node] = cstreec(data2(i,:),treek);
        if clas ~= 2
            nmis = nmis+1; % misclassified
        end
    end
    Rk(k) = nmis/n;
end
The estimated misclassification errors are:
    Rk = 0.01, 0.035, 0.050, 0.19, 0.32.
We see that the minimum estimated misclassification error is the tree $T_1$. We show below how to use Equation 9.31 to get the estimated standard error.

% Find the minimum Rk.
[mrk,ind] = min(Rk);
% The tree $T_1$ corresponds to the minimum Rk.
% Now find the se for that one.
semrk = sqrt(mrk*(1-mrk)/n);
% The SE is 0.0070. We add that to min(Rk).
Rk2 = mrk+semrk;
When we add the estimated standard error of 0.007 to the minimum estimated misclassification error, we get 0.017. None of the other trees in the sequence has an error less than this, so tree $T_1$ is the one we would select as the best tree.

**Selecting the Best Tree Using Cross-Validation**

We now turn our attention to the case where we use cross-validation to estimate our misclassification error for the trees. In cross-validation, we divide
our learning sample into several training and testing sets. We use the training sets to build sequences of trees and then use the test sets to estimate the misclassification error.

In previous examples of cross-validation, our testing sets contained only one observation. In other words, the learning sample was sequentially partitioned into $n$ test sets. As we discuss shortly, it is recommended that far fewer than $n$ partitions be used when estimating the misclassification error for trees using cross-validation. We first provide the notation that will be used in describing the cross-validation method for choosing the right size tree.

**NOTATION - CROSS-VALIDATION METHOD**

$L_v$ denotes a partition of the learning sample $L$, such that

$$L^{(v)} = L - L_v,$$

$v = 1, ..., V$.

$T^{(v)}_k$ is a tree grown using the partition $L^{(v)}$.

$\alpha^{(v)}_k$ denotes the complexity parameter for a tree grown using the partition $L^{(v)}$.

$\hat{R}^{CV}(T)$ represents the estimate of the expected misclassification cost for the tree using cross-validation.

We start the procedure by dividing the learning sample $L$ into $V$ partitions $L_v$. Breiman, et al. [1984] recommend a value of $V = 10$ and show that cross-validation using finer partitions does not significantly improve the results. For better results, it is also recommended that systematic random sampling be used to ensure a fixed fraction of each class will be in $L_v$ and $L^{(v)}$. These partitions $L_v$ are set aside and used to test our classification tree and to estimate the misclassification error. We use the remainder of the learning set $L^{(v)}$ to get a sequence of trees

$$T^{(v)}_{max} > T^{(v)}_1 > ... > T^{(v)}_k > T^{(v)}_{k+1} > ... > T^{(v)}_K = \{t_1\},$$

for each training partition. Keep in mind that we have our original sequence of trees that were created using the entire learning sample $L$, and that we are going to use these sequences of trees $T^{(v)}_k$ to evaluate the classification performance of each tree in the original sequence $T_k$. Each one of these sequences will also have an associated sequence of complexity parameters

$$0 = \alpha^{(v)}_{1} < ... < \alpha^{(v)}_{k} < \alpha^{(v)}_{k+1} < ... < \alpha^{(v)}_{K}.$$ 

At this point, we have $V + 1$ sequences of subtrees and complexity parameters.

© 2002 by Chapman & Hall/CRC
We use the test samples $\mathbf{L}_v$ along with the trees $T_k^{(v)}$ to determine the classification error of the subtrees $T_k$. To accomplish this, we have to find trees that have equivalent complexity to $T_k$ in the sequence of trees $T_k^{(v)}$.

Recall that a tree $T_k$ is the minimal cost complexity tree over the range $\alpha_k \leq \alpha < \alpha_{k+1}$. We define a representative complexity parameter for that interval using the geometric mean

$$\alpha_k' = \sqrt{\alpha_k \alpha_{k+1}}. \quad (9.32)$$

The complexity for a tree $T_k$ is given by this quantity. We then estimate the misclassification error using

$$\hat{R}^{CV}(T_k) = \hat{R}^{CV}(T(\alpha_k')),$$  \quad (9.33)

where the right hand side of Equation 9.33 is the proportion of test cases that are misclassified, using the trees $T_k^{(v)}$ that correspond to the complexity parameter $\alpha_k'$.

To choose the best subtree, we need an expression for the standard error of the misclassification error $\hat{R}^{CV}(T_k)$. When we present our test cases from the partition $\mathbf{L}_v$, we record a zero or a one, denoting a correct classification and an incorrect classification, respectively. We see then that the estimate in Equation 9.33 is the mean of the ones and zeros. We estimate the standard error of this from

$$\hat{SE}(\hat{R}^{CV}(T_k)) = \sqrt{\frac{s^2}{n}}, \quad (9.34)$$

where $s^2$ is $(n - 1)/n$ times the sample variance of the ones and zeros.

The cross-validation procedure for estimating the misclassification error when we have unit cost and the priors are estimated from the data is outlined below.

**PROCEDURE - CHOOSING THE BEST SUBTREE (CROSS-VALIDATION)**

1. Obtain a sequence of subtrees $T_k$ that are grown using the learning sample $\mathbf{L}$.
2. Determine the cost complexity parameter $\alpha_k'$ for each $T_k$ using Equation 9.32.
3. Partition the learning sample into $V$ partitions, $\mathbf{L}_v$. These will be used to test the trees.
4. For each $\mathbf{L}_v$, build the sequence of subtrees using $\mathbf{L}_v^{(v)}$. We should now have $V + 1$ sequences of trees.
5. Now find the estimated misclassification error $\hat{R}_{CV}(T_k)$. For $\alpha_k'$ corresponding to $T_k$, find all equivalent trees $T_k^{(v)}$, $v = 1, \ldots, V$. We do this by choosing the tree $T_k$ such that

$$\alpha_k' \in [\alpha_k^{(v)}, \alpha_k^{(v+1)}].$$

6. Take the test cases in each $L_v$ and present them to the tree $T_k^{(v)}$ found in step 5. Record a one if the test case is misclassified and a zero if it is classified correctly. These are the classification costs.

7. Calculate $\hat{R}_{CV}(T_k)$ as the proportion of test cases that are misclassified (or the mean of the array of ones and zeros found in step 6).

8. Calculate the standard error as given by Equation 9.34.

9. Continue steps 5 through 8 to find the misclassification cost for each subtree $T_k$.

10. Find the minimum error

$$\hat{R}_{min}^{CV} = \min_k \{\hat{R}_{CV}(T_k)\}.$$

11. Add the estimated standard error to it to get

$$\hat{R}_{min}^{CV} + SE(\hat{R}_{min}^{CV}).$$

12. Find the tree with the largest $k$ or fewest number of nodes such that its misclassification error is less than the amount found in step 11.

**Example 9.14**

For this example, we return to the *iris* data, described at the beginning of this chapter. We implement the cross-validation approach using $V = 5$. We start by loading the data and setting up the indices that correspond to each partition. The fraction of cases belonging to each class is the same in all testing sets.

```matlab
load iris
% Attach class labels to each group.
setosa(:,5)=1;
versicolor(:,5)=2;
virginica(:,5)=3;
X = [setosa;versicolor;virginica];
n = 150;% total number of data points
% These indices indicate the five partitions
% for cross-validation.
ind1 = 1:5:50;
```

© 2002 by Chapman & Hall/CRC
ind2 = 2:5:50;
ind3 = 3:5:50;
ind4 = 4:5:50;
ind5 = 5:5:50;

Next we set up all of the testing and training sets. We use the MATLAB eval function to do this in a loop.

```matlab
% Get the testing sets: test1, test2, ...
for i = 1:5
    eval(['test' int2str(i) '= [setosa(ind' int2str(i) 
       ',:); versicolor(ind' int2str(i) ...
       ',:); virginica(ind' int2str(i) ',:)];'])
end

for i = 1:5
    tmp1 = setosa;
    tmp2 = versicolor;
    tmp3 = virginica;
    % Remove points that are in the test set.
    eval(['tmp1(ind' int2str(i) ',:) = [];'])
    eval(['tmp2(ind' int2str(i) ',:) = [];'])
    eval(['tmp3(ind' int2str(i) ',:) = [];'])
    eval(['train' int2str(i) '= [tmp1;tmp2;tmp3];'])
end

Now we grow the trees using all of the data and each training set.

```matlab
% Grow all of the trees.
pies = ones(1,3)/3;
maxn = 2;% get large trees
clas = 1:3;
Nk = [50,50,50];
tree = csgrowc(X,maxn,clas,Nk,pies);
Nk1 = [40 40 40];
for i = 1:5
    eval(['tree' int2str(i) '= ... 
       csgrowc(train',...
       int2str(i) ',maxn,clas,Nk1,pies);'])
end
```

The following MATLAB code gets all of the sequences of pruned subtrees:

```matlab
% Now prune each sequence.
treeseq = csprunec(tree);
for i = 1:5
    eval(['treeseq' int2str(i) '= ... 
       csprunec(tree' int2str(i) ');'])
end
```
The complexity parameters must be extracted from each sequence of sub-trees. We show how to get this for the main tree and for the sequences of sub-trees grown on the first partition. This must be changed appropriately for each of the remaining sequences of subtrees.

\[
K = \text{length}(\text{treeseq});
\]

\[
\text{alpha} = \text{zeros}(1,K);
\]

% Find the sequence of alphas.

\[
\text{for } i = 1:K
\]

\[
\alpha(i) = \text{treeseq}(i).\text{alpha};
\]

end

% For the other subtree sequences, change the
% 1 to 2, 3, 4, 5 and re-run.

\[
K1 = \text{length}(\text{treeseq}1);
\]

\[
\text{for } i = 1:K1
\]

\[
\alpha1(i) = \text{treeseq}1(i).\text{alpha};
\]

end

We need to obtain the equivalent complexity parameters for the main sequence of trees using Equation 9.32. We do this in MATLAB as follows:

\[
\% \text{Get the } \text{akprime } \text{equivalent values for the main tree.}
\]

\[
\text{for } i = 1:K-1
\]

\[
\text{akprime}(i) = \sqrt{\alpha(i) \times \alpha(i+1)};
\]

end

We must now loop through all of the subtrees in the main sequence, find the equivalent subtrees in each partition and use those trees to classify the cases in the corresponding test set. We show a portion of the MATLAB code here to illustrate how we find the equivalent subtrees. The complete steps are contained in the M-file called ex9_14.m (downloadable with the Computational Statistics Toolbox). In addition, there is an alternative way to implement cross-validation using cell arrays (courtesy of Tom Lane, The MathWorks). The complete procedure can be found in ex9_14alt.m.

\[
n = 150;
\]

\[
k = \text{length}(\text{akprime});
\]

\[
\text{misclass} = \text{zeros}(1,n);
\]

% For the first tree, find the
% equivalent tree from the first partition

\[
\text{ind} = \text{find}(\alpha1 <= \text{akprime}(1)) ;
\]

% Should be the last one.

% Get the tree that corresponds to that one.

\[
\text{tk} = \text{treeseq}1(\text{ind}(\text{end}));
\]

% Get the misclassified points in the test set.

\[
\text{for } j = 1:30 \quad \% \text{loop through the points in test 1}
\]

\[
[\text{c},\text{pclass},\text{node}] = \text{cstreec}(\text{test}1(j,1:4),\text{tk});
\]

\[
\text{if } \text{c} ~= \text{test}1(j,5)
\]
misclass(j) = 1;
end
end

We continue in this manner using all of the subtrees. The estimated misclassification error using cross-validation is

\[ R_k = 0.047, 0.047, 0.047, 0.067, 0.21, 0.41, \]

and the estimated standard error for \( R_k^{CV} \) is 0.017. When we add this to the minimum of the estimated errors, we get 0.064. We see that the tree with the minimum complexity that has error less than this is tree \( T_3 \). All of the data and variables that are generated in this example can be loaded from irisexamp.mat.

9.5 Clustering

Clustering methodology is used to explore a data set where the goal is to separate the sample into groups or to provide understanding about the underlying structure or nature of the data. The results from clustering methods can be used to prototype supervised classifiers or to generate hypotheses. Clustering is called unsupervised classification because we typically do not know what groups there are in the data or the group membership of an individual observation. In this section, we discuss two main methods for clustering. The first is hierarchical clustering, and the second method is called \( k \)-means clustering. First, however, we cover some preliminary concepts.

Measures of Distance

The goal of clustering is to partition our data into groups such that the observations that are in one group are dissimilar to those in other groups. We need to have some way of measuring that dissimilarity, and there are several measures that fit our purpose.

The first measure of dissimilarity is the \textit{Euclidean distance} given by

\[ d_{rs} = \sqrt{(x_r - x_s)^T (x_r - x_s)}, \quad (9.35) \]

where \( x_r \) is a column vector representing one observation. We could also use the \textit{Mahalanobis distance} defined as

\[ d_{rs} = \sqrt{(x_r - x_s)^T \Sigma^{-1} (x_r - x_s)}, \quad (9.36) \]

© 2002 by Chapman & Hall/CRC
where $\Sigma^{-1}$ denotes the inverse covariance matrix. The \textit{city block distance} is found using absolute values rather than squared distances, and it is calculated using

$$d_{rs} = \sum_{j=1}^{d} |x_{rj} - x_{sj}|.$$  \hspace{1cm} (9.37)

In Equation 9.37, we take the absolute value of the difference between the observations $x_r$ and $x_s$ componentwise and then add up the values. The final distance that we present covers the more general case of the Euclidean distance or the city block distance. This is called the \textit{Minkowski distance}, and it is found using

$$d_{rs} = \left\{ \sum_{j=1}^{d} |x_{rj} - x_{sj}|^p \right\}^{1/p}.$$  \hspace{1cm} (9.38)

If $p = 1$, then we have the city block distance, and if $p = 2$ we have the Euclidean distance.

The researcher should be aware that distances might be affected by differing scales or magnitude among the variables. For example, suppose our data measured two variables: age and annual income in dollars. Because of its magnitude, the income variable could influence the distances between observations, and we would end up clustering mostly on the incomes. In some situations, we might want to standardize the observations. The MATLAB Statistics Toolbox contains a function called \texttt{zscore} that will perform this standardization.

The MATLAB Statistics Toolbox also has a function that calculates distances. It is called \texttt{pdist} and takes as its argument a matrix $X$ that is dimension $n \times d$. Each row represents an observation in our data set. The \texttt{pdist} function returns a vector containing the distance information. The default distance is Euclidean, but the user can specify other distances as discussed above. We illustrate the use of this function in the following example.

\textbf{Example 9.15}

We use a small data set to illustrate the various distances available in the MATLAB Statistics Toolbox. We have only five data points. The following commands set up the matrix of values and plots the points in Figure 9.16.

```matlab
% Let's make up a data set - 2-D.
x = [1 1; 1.2; 2 1; -1 -1; -1 -2];
plot(x(:,1),x(:,2),'kx') % plots the points.
axis([-3 3 -3 3])
text(x(:,1)+.1,x(:,2)+.1,'1|2|3|4|5');
```

© 2002 by Chapman & Hall/CRC
We first find the Euclidean distance between the points using the `pdist` function. We also illustrate the use of the function `squareform` that puts the distances in a more familiar matrix form, where the $ij$-th element corresponds to the distance between the $i$-th and $j$-th observation.

```matlab
% Find the Euclidean distance using pdist.
% Convert to matrix form for easier reading.
ye = pdist(x,'euclid');
ye_mat = squareform(ye);
```

The matrix we get from this is

```
0   1.0000   1.0000  2.8284  3.6056
1.0000         0   1.4142  3.6056  4.4721
1.0000   1.4142         0  3.6056  4.2426
2.8284  3.6056   3.6056         0   1.0000
3.6056  4.4721  4.2426   1.0000         0
```

We contrast this with the city block distance.

```matlab
% Contrast with city block metric.
ycb = pdist(x,'cityblock');
ycb_mat = squareform(ycb);
```

The result we get from this is

```
ycb_mat =
0   1    1    4    5
1    0    2    5    6
1    2    0    5    6
4    5    5    0    1
5    6    6    1    0
```

Hierarchical Clustering

There are two types of hierarchical clustering methods: agglomerative and divisive. *Divisive* methods start with one large group and successively split the groups until there are $n$ groups with one observation per group. In general, methods for this type of hierarchical clustering are computationally inefficient [Webb, 1999], so we do not discuss them further. *Agglomerative* methods are just the opposite; we start with $n$ groups (one observation per group) and successively merge the two most similar groups until we are left with only one group.

There are five commonly used methods for merging clusters in agglomerative clustering. These are single linkage, complete linkage, average linkage,
centroid linkage and Ward’s method. The MATLAB Statistics Toolbox provides a function called \textit{linkage} that will perform agglomerative clustering using any of these methods. Its use is illustrated in the next example, but first we briefly describe each of the methods [Hair, et al., 1995].

The \textit{single linkage} method uses minimum distance, where the distance between clusters is defined as the distance between the closest pair of observations. Pairs consisting of one case from each group are used in the calculation. The first cluster is formed by merging the two groups with the shortest distance. Then the next smallest distance is found between all of the clusters (keep in mind that an observation is also a cluster). The two clusters corresponding to the smallest distance are then merged. The process continues in this manner until there is one group. In some cases, single linkage can lead to chaining of the observations, where those on the ends of the chain might be very dissimilar.

The \textit{complete linkage} method is similar to single linkage, but the clustering criterion is different. The distance between groups is defined as the most distant pair of observations, with one coming from each group. The logic behind using this type of similarity criterion is that the maximum distance between observations in each cluster represents the smallest sphere that can enclose all of the objects in both clusters. Thus, the closest of these cluster pairs should be grouped together. The complete linkage method does not have the chaining problem that single linkage has.
The average linkage method for clustering starts out the same way as single and complete linkage. In this case, the cluster criterion is the average distance between all pairs, where one member of the pair comes from each cluster. Thus, we find all pairwise distances between observations in each cluster and take the average. This linkage method tends to combine clusters with small variances and to produce clusters with approximately equal variance.

Centroid linkage calculates the distance between two clusters as the distance between the centroids. The centroid of a cluster is defined as the \(d\)-dimensional sample mean for those observations that belong to the cluster. Whenever we merge clusters together or add an observation to a cluster, the centroid is recalculated.

The distance between two clusters using Ward’s linkage method is defined as the incremental sum of the squares between two clusters. To merge clusters, the within-group sum-of-squares is minimized over all possible partitions obtained by combining two clusters. The within-group sum-of-squares is defined as the sum of the squared distances between all observations in a cluster and its centroid. This method tends to produce clusters with approximately the same number of observations in each one.

Example 9.16
We illustrate the linkage function using the data and distances from the previous example. We look only at single linkage and complete linkage using the Euclidean distances. We show the results of the clustering in dendrograms given in Figures 9.17 and 9.18.

```matlab
% Get the cluster output from the linkage function.
zsingle = linkage(ye,'single');
zcomplete = linkage(ye,'complete');
% Get the dendrogram.
dendrogram(zsingle)
title('Clustering - Single Linkage')
dendrogram(zcomplete)
title('Clustering - Complete Linkage')
```

A dendrogram shows the links between objects as inverted U-shaped lines, where the height of the U represents the distance between the objects. The cases are listed along the horizontal axis. Cutting the tree at various \(y\) values of the dendrogram yields different clusters. For example, cutting the complete linkage tree at \(y = 1.2\) would yield 3 clusters. As expected, if we choose to create two clusters, then the two linkage methods give the same cluster definitions.

Now that we have our cases clustered, we would like to measure the validity of the clustering. One way to do this would be to compare the distances between all observations with the links in the dendrogram. If the clustering
This is the dendrogram using Euclidean distances and single linkage.

This is the dendrogram using Euclidean distances and complete linkage.

FIGURE 9.17
This is the dendrogram using Euclidean distances and single linkage.

FIGURE 9.18
This is the dendrogram using Euclidean distances and complete linkage.
is a valid one, then there should be a strong correlation between them. We can measure this using the \textit{cophenetic correlation coefficient}. A cophenetic matrix is defined using the results of the linkage procedure. The \(ij\)-th entry of the cophenetic matrix is the fusion level at which the \(i\)-th and \(j\)-th objects appear together in the same cluster for the first time. The correlation coefficient between the distances and the corresponding cophenetic entries is the cophenetic correlation coefficient. Large values indicate that the linkage provides a reasonable clustering of the data. The MATLAB Statistics Toolbox provides a function that will calculate the cophenetic correlation coefficient. Its use is illustrated in the following example.

\textbf{Example 9.17}

In this example, we show how to obtain the cophenetic correlation coefficient in MATLAB. We use the same small data set from before and calculate the cophenetic correlation coefficient when we have clusters based on different distances and linkages. First, we get the clusters using the following commands.

\begin{verbatim}
  x = [1 1; 1 2; 2 1; -1 -1; -1 -2];
  ye = pdist(x,'euclid');
  ycb = pdist(x,'cityblock');
  zsineu = linkage(ye,'single');
  zcompeu = linkage(ye,'complete');
  zsincb = linkage(ycb,'single');
  zcomcb = linkage(ycb,'complete');
\end{verbatim}

We now have four different cluster hierarchies. Their cophenetic correlation coefficients can be found from the following:

\begin{verbatim}
  ccompeu = cophenet(zcompeu,ye);
  csineu = cophenet(zsineu,ye);
  csincb = cophenet(zsincb,ycb);
  ccomcb = cophenet(zcomcb,ycb);
\end{verbatim}

As expected, all of the resulting cophenetic correlation coefficients are large (approximately 0.95), with the largest corresponding to the complete linkage clustering based on the city block distance.

\textbf{K-Means Clustering}

The goal of \(k\)-means clustering is to partition the data into \(k\) groups such that the within-group sum-of-squares is minimized. One way this technique differs from hierarchical clustering is that we must specify the number of groups or clusters that we are looking for. We briefly describe two algorithms for obtaining clusters via \(k\)-means.
One of the basic algorithms for $k$-means clustering is a two step procedure. First, we assign observations to its closest group, usually using the Euclidean distance between the observation and the cluster centroid. The second step of the procedure is to calculate the new cluster centroid using the assigned objects. These steps are alternated until there are no changes in cluster membership or until the centroids do not change. This algorithm is sometimes referred to as HMEANS [Spath, 1980] or the basic ISODATA method.

**PROCEDURE - HMEANS ALGORITHM**

1. Specify the number of clusters $k$.
2. Determine initial cluster centroids. These can be randomly chosen or the user can specify them.
3. Calculate the distance between each observation and each cluster centroid.
4. Assign every observation to the closest cluster.
5. Calculate the centroid (i.e., the $d$-dimensional mean) of every cluster using the observations that were just grouped there.
6. Repeat steps 3 through 5 until no more changes are made.

There are two problems with the HMEANS algorithm. The first one is that this method could lead to empty clusters, so users should be aware of this possibility. As the centroid is recalculated and observations are reassigned to groups, some clusters could become empty. The second issue concerns the optimality of the partitions. With $k$-means, we are searching for partitions where the within-group sum-of-squares is minimum. It can be shown [Webb, 1999] that in some cases, the final $k$-means cluster assignment is not optimal, in the sense that moving a single point from one cluster to another may reduce the sum of squared errors. The following procedure helps address the second problem.

**PROCEDURE - K-MEANS**

1. Obtain a partition of $k$ groups, possibly from the HMEANS algorithm.
2. Take each data point $x_i$ and calculate the Euclidean distance between it and every cluster centroid.
3. Here $x_i$ is in the $r$-th cluster, $n_r$ is the number of points in the $r$-th cluster, and $d_{ir}^2$ is the Euclidean distance between $x_i$ and the centroid of cluster $r$. If there is a group $s$ such that

$$\frac{n_r}{n_r - 1} d_{ir}^2 > \frac{n_s}{n_s + 1} d_{is}^2$$

© 2002 by Chapman & Hall/CRC
then move $x_i$ to cluster $s$.

4. If there are several clusters that satisfy the above inequality, then move the $x_i$ to the group that has the smallest value for

$$\frac{n_s}{n_s + 1}d_{is}^2.$$ 

5. Repeat steps 2 through 4 until no more changes are made.

We note that there are many algorithms for $k$-means clustering described in the literature. We provide some references to these in the last section of this chapter.

**Example 9.18**

We show how to implement HMEANS in MATLAB, using the iris data. Normally, clustering methods would be used on data where we do not know what groups are there, unlike the iris data. However, since we do know the true groups represented by the data, these will give us a way to verify that the clusters make sense. We first obtain the cluster centers by randomly picking observations from the data set. Note that initial cluster centers do not have to be actual observations.

```matlab
load iris
k = 3;
% Put all of the data together.
x = [setosa; versicolor; virginica];
[n,d] = size(x);

% Pick some observations to be the cluster centers.
ind = randperm(n);
ind = ind(1:k);
nc = x(ind,:);

% Set up storage.
% Integers 1,...,k indicating cluster membership
 cid = zeros(1,n);
% Make this different to get the loop started.
oldcid = ones(1,n);
% The number in each cluster.
nr = zeros(1,k);
% Set up maximum number of iterations.
maxiter = 100;
iter = 1;

while ~isequal(cid,oldcid)& iter < maxiter
    oldcid = cid;
```
% Implement the hmeans algorithm.
% For each point, find the distance
% to all cluster centers.
for i = 1:n
    dist = sum((repmat(x(i,:),k,1)-nc).^2,2);
    % assign it to this cluster
    [m,ind] = min(dist);
    cid(i) = ind;
end
% Find the new cluster centers.
for i = 1:k
    % Find all points in this cluster.
    ind = find(cid==i);
    % Find the centroid.
    nc(i,:) = mean(x(ind,:));
    % Find the number in each cluster.
    nr(i) = length(ind);
end
iter = iter + 1
end

To check these results, we show a scatterplot of the first two features of the iris data in Figure 9.19, where the three classes are represented by different plotting symbols. The clusters we obtain from this implementation of k-means clustering (using the HMEANS procedure) are shown in Figure 9.20. The algorithm finds the one group, corresponding to Iris setosa, but has trouble separating the other two species. However, the results are certainly reasonable.

9.6 MATLAB Code

We provide a function called cshmeans that implements the HMEANS algorithm given above. We also have a function called cskmeans that checks to see if moving individual observations changes the sum-square error. With both of these functions, the user can specify the initial centers as an input argument. However, if that argument is omitted, then the function will randomly pick the initial cluster centers.

As we stated in the body of the text, there are many MATLAB functions available that the analyst can use to develop classifiers using Bayes decision theory. These are any of the functions in the Statistics Toolbox that estimates a probability density function using the parametric approach: normfit, expfit, gamfit, unifit, betafit, and weibfit. These functions return
the appropriate distribution parameters estimated from the sample. For the nonparametric approach, one can use any of the techniques from Chapter 8: histograms, frequency polygons, kernel methods, finite mixtures or adaptive mixtures. Also, there is a function in the Statistics Toolbox called classify. This performs linear discriminant analysis [Duda, Hart, and Stork, 2001] using Mahalanobis distances. Class labels are assigned based on the distance between the observation and the cases in the training set.

A set of M-files implementing many of the methods described in Ripley [1996] are available for download at


There are functions for k-means, Bayesian classifiers and logistic discriminant analysis.

The MATLAB Statistics Toolbox has several functions for clustering. In Examples 9.15 through 9.17, we illustrated the use of pdist, squareform, linkage, and cophenet. There are other clustering functions that the data analyst might find useful. One is called cluster, which is used to divide the
output of linkage into clusters. It does this in one of two ways: 1) by finding the natural divisions, or 2) by the user specifying arbitrary clusters. The function `inconsistent` helps the user find natural divisions in the data set by comparing the length of the links in a cluster tree with the lengths of neighboring links. If the link is approximately the same as its neighbors, then it exhibits a high level of consistency. If not, then they are considered to be inconsistent. Inconsistent links might indicate a division of the data. The reader is asked to explore this further in the exercises. Finally, the function `clusterdata` combines the three functions, `pdist`, `linkage`, and `cluster` into one. However, `clusterdata` uses Euclidean distance and single linkage clustering. So, if another cluster methodology is needed, the three separate functions must be used.

**FIGURE 9.20**
This shows the first two features of the clusters found using k-means, where all four features were used in the clustering algorithm. As expected, the cluster in the upper left corner is found. The other two clusters do not show the same separation, but the results are reasonable when compared to the true groups shown in Figure 9.19.
9.7 Further Reading

There are many excellent books on statistical pattern recognition that can be used by students at the graduate level or researchers with a basic knowledge of calculus and linear algebra. The text by Duda and Hart [1973] is a classic book on pattern recognition and includes the foundational theory behind Bayes decision theory, classification and discriminant analysis. It has recently been revised and updated [Duda, Hart, and Stork, 2001]. This second edition contains many new topics, examples, and pseudo-code. Fukunaga [1990] is at the same level and includes similar subjects; however, it goes into more detail on the feature extraction aspects of pattern recognition. Devroye, Gyorfi, and Lugosi [1996] CONTAINS an extensive treatment of the probabilistic theory behind pattern recognition. Ripley [1996] covers pattern recognition from a neural network perspective. This book is recommended for both students and researchers as a standard reference. An excellent book that discusses all aspects of statistical pattern recognition is the text by Webb [1999]. This is suitable for advanced undergraduate students and professionals. The author explains the techniques in a way that is understandable, and he provides enough theory to explain the methodology, but does not overwhelm the reader with it.
The definitive book on classification trees is the one by Breiman, et al. [1984]. This text provides algorithms for building classification trees using ordered or categorical data, mixtures of data types, and splitting nodes using more than one variable. They also provide the methodology for using trees in regression. A paper by Safavian and Landgrebe [1991] provides a review of methodologies for building and using classification trees. A description of classification trees can also be found in Webb [1999] and Duda, Hart, and Stork [2001].

Many books are available that describe clustering techniques, and we mention a few of them here. The books by Hartigan [1975], Spath [1980], Anderberg [1973], Kaufman and Rousseeuw [1990], and Jain and Dubes [1988] provide treatments of the subject at the graduate level. Most of the texts mentioned above on statistical pattern recognition discuss clustering also. For example, see Duda and Hart [1973], Duda, Hart and Stork [2001], Ripley [1996], or Webb [1999]. For two books that are appropriate at the undergraduate level, we refer the reader to Everitt [1993] and Gordon [1999].

We conclude this chapter with a brief discussion of a technique that combines agglomerative clustering and finite mixtures. This method is called model-based clustering [Fraley, 1998; Fraley and Raftery, 1998]. First, agglomerative clustering is performed, where clusters are merged based on the finite mixture model, rather than the distances. The partitions obtained from the model-based agglomerative clustering provide an initialization (number of components, means, variances and weights) to the finite mixtures EM algorithm (with normal components). An approximation to Bayes factors is used to pick the best model.

© 2002 by Chapman & Hall/CRC
Exercises

9.1. Load the insect data [Hand, et al., 1994; Lindsey, et al., 1987]. These are three variables measured on each of ten insects from three species. Using the parametric approach and assuming that these data are multivariate normal with different covariances, construct a Bayes classifier. Use the classifier to classify the following vectors as species I, II, or III:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>190</td>
<td>143</td>
<td>52</td>
</tr>
<tr>
<td>174</td>
<td>131</td>
<td>50</td>
</tr>
<tr>
<td>218</td>
<td>126</td>
<td>49</td>
</tr>
<tr>
<td>130</td>
<td>131</td>
<td>51</td>
</tr>
<tr>
<td>138</td>
<td>127</td>
<td>52</td>
</tr>
<tr>
<td>211</td>
<td>129</td>
<td>49</td>
</tr>
</tbody>
</table>

9.2. The household [Hand, et al., 1994; Aitchison, 1986] data set contains the expenditures for housing, food, other goods, and services (four expenditures) for households comprised of single people. Apply the clustering methods of Section 9.5 to see if there are two groups in the data, one for single women and one for single men. To check your results, the first 20 cases correspond to single men, and the last 20 cases are for single women.

9.3. Grow a classification tree for the household data, using the class labels as given in problem 9.2. Is the tree consistent with the results from the clustering?

9.4. The measure [Hand, et. al., 1994] data contain 20 measurements of chest, waist and hip data. Half of the measured individuals are women and half are men. Use cluster analysis to see if there is evidence of two groups.

9.5. Use the online help to find out more about the MATLAB Statistics Toolbox functions cluster and inconsistent. Use these with the data and clusters of Examples 9.15 through 9.17 to extract the clusters.

9.6. Apply the cross-validation procedure and ROC curve analysis of Example 9.8 to the tibetan data. Designate Type A skulls as the target class and Type B skulls as the non-target class.

9.7. Use the bank data along with the independent test sample approach to estimate the probability of correctly classifying patterns (see Example 9.7). The file contains two matrices, one corresponding to features
taken from 100 forged Swiss bank notes and the other comprising features from 100 genuine Swiss bank notes [Flury and Riedwyl, 1988]. There are six features: length of the bill, left width of the bill, right width of the bill, width of the bottom margin, width of the top margin and length of the image diagonal. Compare classifiers obtained from: 1) the parametric approach, assuming the class-conditional are multivariate normal with different covariances, and 2) the nonparametric approach, estimating the class-conditional probabilities using the product kernel. Which classifier performs better based on the estimated probability of correct classification?

9.8. Apply the cross-validation procedure and ROC curve analysis of Example 9.8 to the bank data. The target class corresponds to the forged bills. Obtain ROC curves for a classifier built using: 1) the parametric approach, assuming the class-conditional are multivariate normal with different covariances, and 2) the nonparametric approach, estimating the class-conditional probabilities using the product kernel. Which classifier performs better, based on the ROC curve analysis?

9.9. For the bank data, obtain a classification tree. Use the independent test sample approach to pick a final pruned tree.

9.10. Apply k-means clustering to the complete bank data, without class labels. Apply the hierarchical clustering methods to the data. Is there significant evidence of two groups?

9.11. Do a Monte Carlo study of the probability of misclassification. Generate $n$ random variables using the class-conditional probabilities and the priors from Example 9.3. Estimate the probability of misclassification based on the data. Note that you will have to do some probability density estimation here. Record the probability of error for this trial. Repeat for $M$ Monte Carlo trials. Plot a histogram of the errors. What can you say about the probability of error based on this Monte Carlo experiment?

9.12. The flea data set [Hand, et al., 1994; Lubischew, 1962] contains measurements on three species of flea beetle: Chaetocnema concinna, Chaetocnema heikertingeri, and Chaetocnema heptapotamica. The features for classification are the maximal width of aedeagus in the forepart (microns) and the front angle of the aedeagus (units are 7.5 degrees). Build a classifier for these data using a Bayes classifier. For the Bayes classifier, experiment with different methods of estimating the class-conditional probability densities. Construct ROC curves and use them to compare the classifiers.

9.13. Build a classification tree using the flea data. Based on a three-term multivariate normal finite mixture model for these data, obtain an estimate of the model. Using the estimated model, generate an independent test sample to pick the best tree in the sequence of subtrees.
9.14. The \textit{k-nearest neighbor rule} assigns patterns \( x \) to the class that is the most common amongst its \( k \) nearest neighbors. To fix the notation, let \( k_m \) represent the number of cases belonging to class \( \omega_m \) out of the \( k \) nearest neighbors to \( x \). We classify \( x \) as belonging to class \( \omega_m \), if \( k_m \geq k_i \), for \( i = 1, \ldots, J \). Write a MATLAB function that implements this classifier.

9.15. Repeat Example 9.7 using all of the features for \texttt{versicolor} and \texttt{virginica}. What is your estimated probability of correct classification?

9.16. Apply the method of Example 9.7 to the \texttt{virginica} and \texttt{setosa} classes.