Introduction

In many situations, a decision should be made regarding, for instance, what specific signal out of several standard ones is received.

Such problems are commonly referred to as classification problems.

Block diagram of a typical classification problem:

The classifier receives a number of variables usually derived from signal or image processing algorithms and, based on the analysis of these combined parameters, estimates the state of the system. The inputs can take on any value but the output is a discrete variable that specifies the class. For instance, if only two standard signals (classes) are possible, the output values may be 0 and 1 or -1 and +1 to identify these classes.
Introduction

The earliest classifiers were based on the Bayesian analysis, an approach called maximum likelihood. This technique essentially finds the optimum separation boundary between the probability distributions of the various classes. While this technique guarantees optimal classification, it requires the a-priori knowledge of the distributions of classes – the condition that is never satisfied in practice. It also requires that the distributions should be Gaussian – another condition that does not usually apply to real-world problems. Because of these serious limitations, many other methods have been developed. Classification is usually one component of a more general analysis that tries to associate a pattern of input variables with either a specific class or another variable. If the output is a variable, the analysis is referred to as regression, while if the output is a discrete number identifying a specific class, the analysis is referred to as a classification. We only discuss the classification algorithms that produce discrete outputs. Since classifiers establish a relationship between the pattern of inputs and a discrete output, they are, in the general sense, mathematical functions. Therefore, the development of a classifier can be thought of as function approximation. Other terms for classification are identification, estimation, and pattern recognition.

In principle, a single measurement variable is sufficient for classification. An example of such classification is a heart rate monitor setting off an alarm when the heart rate becomes too low or too high. Here, a classifier is based on a simple thresholding – two thresholds classify heart rate into three classes: “too low”, “normal”, and “too high”. Most classifiers do contain threshold elements, but when two or more variables are involved, a wide range of algorithms exist that attempt making the best classification from the pattern represented by all the variables. The role of the classifier is to determine the most likely class associated with a given input pattern. As the “most likely” suggests, the classifier may use statistical apparatus in its functions. Also, such classifier may not always be correct in its decision.

A large numbers of algorithms have been developed to classify input patterns of two or more variables. We only consider some of the most common methods of classification selected for their effectiveness in many practical situations and, therefore, being popular. The goal of this discussion is to understand the basic problem, the most common approaches that are used for solution, and their strengths and limitations.
Introduction

Classification can be performed using two basic strategies: supervised and unsupervised learning that both are considered as machine learning.

In un supervised learning, the classifier attempts to find patterns within the input data themselves: the classifier has no a priori knowledge of the data patterns and, sometimes, not even the number of classes that exist.

In supervised learning, the classifier is first trained using data, for which the correct class is known. The training data are known as the training set and include the correct classifications.

During the training period, the free parameters associated with the classifier are adjusted adaptively to minimize classification errors. Classifier performance can be evaluated during training using a validation set, for which the correct classifications are also known but are not used to modify classifier parameters. After training is complete, the classifier is applied to a test set, where it performs its designed function to determine the most likely associations based on a given data pattern. The testing error occurring during the testing phase is important. Therefore, a good classifier should be able to perform with minimum error on data that was never available to it.

Introduction

A common failure of classifiers is that they perform well on the training set (after training) but then do poorly on the test data, their real-world application. Such classifiers are said to generalize poorly or to be over-trained.

A geometrical approach is used to study classifiers and the classification problem. Although classifiers often deals with a number of input variables, it is easiest to visualize the data set and the classification operation when only two input variables are involved. Therefore, most examples we consider will only use two input variables for illustration purposes. Classification methods can be easily modified to operate on more input variables.

Using the geometrical approach, a scatter plot (or scattergram) is formed by plotting one variable as a function of another one.
Introduction: a scattergram

Ex. 10.1: Two classes (represented by squares and circles) can be easily separated by a straight line. This line is known as a decision boundary. The classifier checks whether the test data point is below and to the left of a decision boundary (class 0) or not (class 1). Such data are called linearly separable.

Ex. 10.2: Two classes cannot be separated by a straight line but can be separated by a curved line. Such data are called non-linearly separable.
Introduction: a scattergram

Ex. 10.3: Two classes can be separated by a very complicated boundary. Nevertheless, the separation boundary still can be found.

Introduction

So, it may appear that the best way to classify data is by drawing lines between them and constructing an algorithm based on those lines. The problem with this approach is that it is hard to do when more than two parameters are involved, and it may not generalize well. For instance, consider the example 10.4. below:

The previously seen example 10.3

A set with the same statistical properties (Gaussian) as in 10.3. The same boundary results in three errors.
Introduction

Surprisingly, a well-placed straight line does a better job in separating classes with only two errors. Therefore, a straight line generalizes better than the complicated curve for these data. In other words, the complicated curve was a result of overtraining on the training set.

In fact, it can be shown that a simple straight line provides optimal separation for Gaussianly distributed data.

On Classifier design

Many classification algorithms are readily available, for instance, as MATLAB code. Yet most of the important decisions must still be made. In addition to deciding which approach to use, the most important consideration is how complicated to make the decision boundaries. As seen in Ex. 10.4, a complicated boundary is not always the best even if it fits the training data better than a simple boundary.

Classification is a form of machine learning, and the complexity of a classification algorithm is often referred to as machine capacity. Increasing machine capacity leads to a more complex decision boundary, so machine capacity is a major design factor in classifier design. Machine capacity should match in some way the requirements of the data. If the classifier has more capacity than appropriate for the data, it will overtrain on the data, performing well on the training set but not generalizing well to test set data. A machine with too little capacity will show excessive errors in training as well as poor performance in classifying the test data. Machine capacity is closely related to the complexity of the classification algorithm, specifically the number of free parameters.
Linear discriminators

As the name implied, linear classifiers use decision boundaries that are linear: straight lines for two variables, planes for three variables, and hyperplanes for three and more variables. These classifiers only use a single boundary, so they can separate only two classes at a time. However, they can be applied to subsets of the data to identify more than two classes, as will be shown later.

The class predicted by a linear discriminator is determined by the output of a linear equation:

\[ y = \sum_{i=1}^{M} x_i w_i + b \]  

(10.13.1)

where \( M \) is the number of input variables (i.e., the number of different measurements) and \( x_i \) are input variables (measurements themselves). The classifier's free parameters include the weights \( w_i \) and the bias (offset) \( b \). The output \( y \) indicates the class depending on whether it is greater or less than 0.5. If \( y \) is greater than 0.5, the classifier predicts that the input data belongs to class 1, and if \( y \) is less than or equal to 0.5, the classifier predicts that the input data belongs to class 0, where "class 1" and "class 0" are arbitrary names for the two classes.

Linear discriminators

Since most of the examples we consider involve only two variables \( (M = 2) \), \( x_i \) consists of \( x_1 \) and \( x_2 \). However, all of the classifiers described can be extended to any number of input variables.

In addition to the input data \( x_i \), a training set must also include the answers: the correct classification to each input pattern. These correct answers are specified by a vector \( d \) that has the length equal to the number of input patterns. The class may be described as

\[ d = \begin{cases} 
0 \text{ or } -1 & \text{Class 0} \\
1 & \text{Class 1} 
\end{cases} \]  

(10.14.1)

For more than two classes, \( d \) may be a vector of 0s and 1s, where the position of 1 indicates the class:

\[ d = \begin{cases} 
1000 & \text{Class 0} \\
0100 & \text{Class 1} \\
0010 & \text{Class 2} \\
0001 & \text{Class 3} 
\end{cases} \]  

(10.14.2)
Linear discriminators

Alternatively, consecutive numbers, 0, 1, 2, 3,… N) may identify the class.

The output of the linear classifier to any set of input variables is defined by (10.13.1) and is completely determined by the weights and bias, which are the free parameters of the specific classification method. To construct a linear classifier, it is only necessary to find values for $w_i$ and $b$ that best separate the data. (10.13.1) can be modified (simplified) by lumping $b$ with the weights $w_i$. This can be incorporated in linear (or other) classifiers to simplify the calculations. In this case, the bias is taken as the last weight variable, $w_{M+1}$, and the constant 1 is added to the input data set to make it:

$$x_i = [x_1, x_2, ..., x_M, 1]$$  \hspace{1cm} (10.15.1)

In this case, (10.13.1) is replaced by

$$y = \sum_{i=1}^{M+1} x_i w_i = Xw$$  \hspace{1cm} (10.15.2)

where $w$ is the combined weight-bias vector, and $X$ is the matrix of input variables that includes the column of 1s.

Linear discriminators

In some situations, linear classifiers are optimal: for instance, a linear boundary is the best way to separate two Gaussian distributions. Linear classifiers can be trained very quickly and are easy to implement. One popular method for setting the weights is to apply the least squares approach to the learning data. In this approach, the weights are selected to minimize the sum of the squared error between the output of the classifier, $y$, and the actual known class specified as

$$\varepsilon^2 (w) = \sum_{i=1}^{M+1} (d_i - x_i^T w)$$  \hspace{1cm} (10.16.1)

or in matrix notation

$$\varepsilon^2 (w) = (d - Xw)^T (d - Xw)$$  \hspace{1cm} (10.16.2)

To minimize the error $\varepsilon^2 (w)$, it is differentiated with respect to $w$, set to 0, and the resulting equation is solved for $w$. If the matrix $X^TX$ is nonsingular, the unique solution is obtained as

$$w = (X^TX)^{-1} X^T d$$  \hspace{1cm} (10.16.3)
Ex. 10.4. Using MATLAB, generate a test set consisting of two Gaussian distributions with centers being 3 standard deviations apart. The test set should include the correct classification in vector \( d \). Apply the least squares method to classify those two data sets and plot the results indicating classes as squares and circles and any misclassified data points in black. Also, plot the boundary produced.

Solution: the routine “gen_data2” (© J. Semmlow) is used to generate the data sets.

```matlab
function [X d] = gen_data2(distance, angle, type_data, class, npts)
% Function to generate two-variable classification data sets.
% The data are from two classes with Gaussian distribution.
% The means of each class are spaced distance apart and relative positions specified by angle. Both classes have the same
% standard deviation (1).
% Outputs:
% X is a 100 by 2 array containing two-variable
% d is the associated class.
% Xv (optional) is a validation set
% dv (optional) is the associated validation class
% Inputs:
% distance is the Euclidian distance between the class means.
% angle specifies the relative positions in degrees.
% type: ’linear’ Two Gaussian distributions (default)
% ’quad’ Multiple Gaussian in quadratic arrangement
% ’d’(iagonal) Two classes across diagonal
% ’o’(verlap) Both sets have same means but different distributions
% ’c’(onvex) Gaussian data surrounded on three sides
% ’s’(urrond) Gaussian data surrounded on all sides
% class: number for output class (usually either -1,1 or 0,+1 default)
% npts: number of points (default = 100)
% clear R X d;
% if nargin < 5
% npts = 100; % Default npts
% end
% if nargin < 4 | isempty(class)
% class = [0 1]; % Default
% end
% if nargin < 3
% type_data = ’l’; % Default
% end
% nu_clusters = ceil(npts/10); % Number of cluster for nonlinear distribution
% angle = angle*2*pi/360; % Convert to radians
% x1 = distance*cos(angle) + 1;
% y1 = distance*sin(angle) + 1;
% if type_data(1) == ’l’
% R = rand(npts,2);
% for i = 1:npts;
% X(i,1) = R(i,1) + 1; %Generate first class centered at 1.0
% d(i) = class(1);
% X(i,2) = R(i,2) + x1;
% X(i+1,1) = R(i+1,1) + x1;
% X(i+1,2) = R(i+1,2) + y1;
% d(i+1) = class(2);
% end
% end
```
Linear discriminators: Example

```
elseif type_data(1) == 'q'
    index = 1;
    npts1 = fix(npts/nu_clusters);
    for i = 1:nu_clusters
        R = randn(npts1,2)/4;
        for j = i:2:npts1
            X(j,1) = R(j,1) + 1;
            X(j,2) = R(j,2) + 1;
        end
        index = index + 10;
    end

elseif type_data(1) == 'd'  %Diagonal
    R = randn(npts,2);
    for i = 1:4:npts
        X(i,:) = R(i,:) + 1;  %Generate first class centered at 1.0
        d(i) = class(1);
        X(i+1,1) = R(i+1,1) + 1 + distance;
        X(i+1,2) = R(i+1,2) + 1;
        d(i+1) = class(2);
        X(i+2,1) = R(i+1,1) + 1 + distance;
        X(i+2,2) = R(i+1,2) + 1 + distance;
        d(i+2) = class(1);
        X(i+3,1) = R(i+1,1) + 1;  %Generate first class centered at 2.0
        X(i+3,2) = R(i+1,2) + 1 + distance;
        d(i+3) = class(1);
    end

elseif type_data(1) == 's'
    R = randn(npts,2);
    for i = 1:4:npts
        X(i,:) = R(i,:) + distance;  %Generate first class centered at 2.0
        d(i) = class(1);
        X(i+1,1) = R(i+1,1)*3 + distance;
        X(i+1,2) = R(i+1,2) + 2* distance;
        d(i+1) = class(1);
        X(i+2,1) = R(i+1,1)*1.5 + 2* distance;
        X(i+2,2) = R(i+1,2)*3 + .5* distance;
        d(i+2) = class(1);
    end
```

```
Linear discriminators: Example

The routine was used as follows:

```matlab
[X,d] = gen_data2(dist,angle)
```

where “dist” is the distance between the means of the two distributions measured in the standard deviation of the data sets. After generating the training set data, the weights are determined using (10.16.3). These weights, the original data X, and correct classifications d are passed to the routine “linear_eval” that plots the data identifying misclassifications and decision boundary. The following code was made:

distance = 6; %distance between distribution centers
angle = 30; %angle between distribution centers
[X,d] = gen_data2(distance,angle); %generate data
[r,c] = size(x);
X = [X,ones(r,1)]; % to account for bias
w = inv(X'*X)'(X'*d); % train the classifier
linear_eval(X,d,w); % evaluate the classifier
Linear discriminators: Example

The routine "linear_eval" uses the weights established during training to classify the data set using the matrix from (10.13.1): \( y = Xw \). Then the data are plotted, where it is checked whether \( d \) and \( y \) match with 0.5 - the decision threshold between two classes: the data are correctly classified if \( d = 1 \) and \( y > 0.5 \); or \( d = 0 \) and \( y \leq 0.5 \). When the input pattern is incorrectly classified, the circle or square is filled with black. The misclassifications are counted and used to evaluate classification measures: sensitivity and specificity.

The decision boundary is at 0.5: input patterns resulting in classifier output exceeding 0.5 are classified as belonging to class 1, otherwise, they are assigned to class 0. Therefore:

\[
y = \sum_{i=1}^{M+1} x_i w_i = Xw = 0.5
\]  
(10.23.1)

For two inputs, this equation becomes:

\[
x_1 w_1 + x_2 w_2 + w_3 = 0.5
\]  
(10.23.2)

Solving for \( x_2 \) in terms of \( x_1 \):

\[
x_2 = -\frac{w_1}{w_2} x_1 + \frac{0.5 - w_3}{w_2}
\]  
(10.24.1)

These equations are implemented in \texttt{linear_eval}

\begin{verbatim}
function [sensitivity, specificity] = linear_eval(X,d,w,threshold)
  % [sensitivity, specificity] = linear_eval(X,d,w,test)
  % Evaluates performance of a linear discriminator and plots results
  % including the decision boundary
  % Plots only two-variable input data, but has been expanded for
  % three-variables in routine 'linear_eval3D'
  % Uses the criteria of > 0.5 to classify
  %
  % Inputs
  % X inputs (assumed a matrix where the number or rows are different)
  % d correct outputs: i.e., targets (assumed a vector)
  % w linear weights
  % test optional, specifies boundary threshold (default 0.5)
  % Output
  % sensitivity Percent true positives
  % specificity Percent true negative
  % mse mean squared error
end
\end{verbatim}
Linear discriminators: Example

if nargin < 4
    threshold = .5;
end
tp = 0;                     % True positive count
fp = 0;                     % False positive count
tn = 0;                     % True negative count
fn = 0;                     % False negative count
[r,c] = size(X);
% Determine linear response to X
y = X*w;
% Evaluate the output

% Plot the results
clf; hold on;
% Assumes Class 0 is 0 and Class 1 is 1
% Evaluates each point for all four possibilities
for i = 1:r
    if d(i) > threshold & y(i) > threshold
        plot(X(i,1),X(i,2),'sqk','MarkerFaceColor', [.8 .8 .8], 'LineWidth', 1);
        tn = tn + 1;        %True negative
    elseif d(i) > threshold & y(i) <= threshold
        plot(X(i,1),X(i,2),'sqk','MarkerFaceColor','k');
        fp = fp + 1;       % False positive
    elseif d(i) <= threshold & y(i) <= threshold
        plot(X(i,1),X(i,2),'ok','MarkerFaceColor','c');
        tp = tp + 1;       % True positive
    elseif d(i) <= threshold & y(i) > threshold
        plot(X(i,1),X(i,2),'ok','MarkerFaceColor','k');
        fn = fn + 1;        % False negative
    end
end

V = axis;                            % Get current axis
% Plot decision boundary W*x = .5
x1 = [min(X(:,1)),max(X(:,1))];                              % Construct x1 over data range
x2 = -w(1)*x1/w(2) + (-w(3)+threshold)/w(2);       % Calculate x2 using Eq. (10.24.1)
plot(x1,x2,'k','LineWidth',2);                                 % Plot boundary line
axis(V);                                                               % Restore axis
xlabel('x1','FontSize',14);
ylabel('x2','FontSize',14);
% Evaluate performance
specificity = (tn/(tn+fp))*100;
sensitivity = (tp/(tp+fn))*100;

The least squares algorithm produces the boundary illustrated:

The boundary separates two classes with no errors; however, we may argue that other boundaries can be constructed that would perfectly separate the clusters.
Linear discriminators: Example

Few examples of such boundaries are illustrated by dashed lines:

The boundary determined by the least squares linear classifier is based on all data points in both clusters. If the two data clusters are normally distributed and the centers of each are adequately represented by the training set data, the boundary found by this method will be optimal.

However, even if the normality assumption is true, the training set usually does not provide sufficient information about cluster centers, particularly, if the training set is small. The poor generalization of the least squares method, while small training data sets are used, is illustrated in the next example.

Ex. 10.5. Train the least squares classifier in the previous example on two training sets with the same centers and spread. One training set should be large (\(N = 200\)) and the other small (\(N = 20\)). Evaluate the results of each training condition with a test set of 500 points also having the same centers and distributions.

Solution: We modify Ex. 10.4 as follows:

```matlab
% Linear classification using least squares. Trained on 2 data sets: 20 points and 200 points and evaluated on a test set of 500 points
clear all; close all;
distance = 6;                % Distance between distribution centers
angle = 30;                   % Angle between distribution centers
training_set = 20;          % Training set size
[Xt,dt] = gen_data2(distance,angle,'l',[],training_set);       % Generate training data
[r,c] = size(Xt);
Xt = [Xt,ones(r,1)];                                          % To account for bias
w = inv(Xt'*Xt)*(Xt'*dt');                                  % Train on small training set

[X,d] = gen_data2(distance,angle,'l',[],500);  % Generate test data
[r,c] = size(X);
X = [X,ones(r,1)];                                           % To account for bias
linear_eval(X,d,w);                                        % Evaluate boundary

% Repeat for a training set of 200
figure;  training_set = 200;                            % Training set size
[Xt,dt] = gen_data2(distance,angle,'l',[],training_set);  % Generate training data
[r,c] = size(Xt);
Xt = [Xt,ones(r,1)];                                        % To account for bias
w = inv(Xt'*Xt)*(Xt'*dt');                                % Train on larger training set
linear_eval(X,d,w);                                       % Evaluate boundary
```
Linear discriminators: Example

The least squares method of selecting the weights works well if the training set is large and closely reflects the data set. However, if the training set is small (as is often the case) this rule does not generalize as well.

The decision boundary found from the small training set, performs perfectly on the training set:

However, it does poorly on the large set. One class 0 point (circle) and 11 class 1 points (squares) were misclassified in the test set:

If the least squares method is trained on the large training set, only one circle and two squares are misclassified in the test set. This demonstrates the inability of least squares method to generalize well when small training sets are used.
Evaluating classifier performance

Even if the classifier is optimal, classification errors can occur if classes overlap. Perfect classification never occur in most practical cases. Several methods can be used to quantify classification errors. The two most popular methods are the confusion matrix and measurements of sensitivity and specificity.

The confusion matrix is a table of correct and incorrect classification, usually specified in percentages. The confusion matrix for a three-class problem:

<table>
<thead>
<tr>
<th>True Class</th>
<th>Predicted Class</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 0</td>
<td>Class 1</td>
<td>Class 2</td>
<td></td>
</tr>
<tr>
<td>% correct</td>
<td>% error</td>
<td>% error</td>
<td></td>
</tr>
<tr>
<td>% error</td>
<td>% correct</td>
<td>% error</td>
<td></td>
</tr>
<tr>
<td>% error</td>
<td>% error</td>
<td>% correct</td>
<td></td>
</tr>
</tbody>
</table>

The diagonal elements are percentages of correctly classified data for each class; the off-diagonal elements are misclassifications. This approach is useful, since some misclassifications may have more serious consequences than others. Confusion matrices may be extended to cover as many classes as needed.

Evaluating classifier performance

The sensitivity and specificity measures are used when only two classes are involved. If more than two classes exist, these measures still can be applied to pairs of classes. In applying sensitivity and specificity, one class is assumed as the unusual class and the other as the nominal class. In medicine, for instance, the unusual class may represent disease or abnormality, while the nominal class may represent the disease-free or normal condition.

In such situations, the classification problem becomes a detection problem with the unusual class called positives and the normal class negatives. The correct detection (classification) of an abnormal condition is called a true positive; correct classification of a normal condition is called a true negative. Classification errors are either false positives (incorrect classification of normal as abnormal) or false negatives (abnormal is classified as normal). Using these definitions:

\[
\text{Sensitivity} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \cdot 100\% = \frac{\text{True Positives}}{\text{Total Abnormal}} \cdot 100\% \quad (10.32.1)
\]

\[
\text{Specificity} = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}} \cdot 100\% = \frac{\text{True Negatives}}{\text{Total Normal}} \cdot 100\% \quad (10.32.2)
\]
Evaluating classifier performance

Often, it is possible to vary the decision boundary to increase or decrease the detection of abnormals. This produces an adjustable tradeoff, where increasing the detection of true positives increases the number of false positives (which lowers the number of true negatives). The tradeoff is between sensitivity and specificity, where increasing sensitivity will decrease specificity.

A curve showing this tradeoff between sensitivity and specificity can be used to determine where to set the detection threshold. The plot of sensitivity vs. specificity is called the ROC (Receiver Operator Characteristic) curve.

The ideal curve ascends along the vertical axis to pass through the 100% sensitivity and 100% specificity. The diagonal line illustrates a classifier predicting classes absolutely randomly. Curves falling below the diagonal indicate that detection criterion should be reversed.

Evaluating classifier performance

An ROC curve can be obtained by varying the threshold that is used to generate the decision boundary and evaluating sensitivity and specificity for each level, as shown in the following example.

Ex. 10.6. Construct an ROC curve for a least square classifier applied to a data set where the two classes show significant overlap. Set the distance between centers at 2 standard deviations.

Solution: We use the last input argument of `linear_eval` to vary the decision threshold...

```matlab
% Construction of an ROC curve using the least squares linear classifier.
clear all; close all;
distance = 2; angle = 30; % Distance and Angle between distribution centers
[X,d] = gen_data2(distance,angle); % Generate data
[r,c] = size(X); X = [X,ones(r,1)]; % To account for bias
w = inv(X'*X)*(X'*d'); % Train on small training set
for i = 1:18
    threshold(i) = .05*i; % Set threshold between .05 and .9
    figure; [sensitivity(i), specificity(i)] = linear_eval(X,d,w,threshold(i)); title(['Threshold: ',num2str(threshold(i),2)]);
end
figure; plot(100-specificity, sensitivity,'k','LineWidth',2); % Plot ROC curve for i = 1:2:18 % Put threshold value on ROC curve
    text(100-specificity(i),sensitivity(i),num2str(threshold(i),2));
end
xlabel('100% - Specificity','FontSize',14); ylabel('Sensitivity','FontSize',14); axis([ 0 100 0 100]);
```
Evaluating classifier performance

Data plots at four different thresholds:
Since two data clusters were placed quite close to each other, many data points overlap making classification a challenge; however, this situation is very common in practice.
Assuming that circles represent normals \((N = 50)\) and squares abnormals \((N = 50)\), at low threshold of 0.3, all abnormals are correctly classified, while 19 normals misclassified.

As the threshold is increased to 0.4, the number of falsely classified normals reduces to 10, although two abnormals are misclassified (black squares). Increasing the threshold to 0.5, decreases the false positives to 8 and increases the false negatives to 8. Finally, at the threshold of 0.9, all normals are correctly classified but 37 abnormals are missed...

The ROC curve generated from 18 different thresholds:
The value of threshold that should be selected depending on the nature of the problem. One candidate threshold is 0.65 leading to sensitivity of 95% and specificity of 73%.
Higher dimensions: Kernel Machines

Linear classifiers are limited to decision boundaries that are straight lines. Many classification problems involve data that are separable but not by a straight line (or plane).

Two classes (circles and squares), each of which consists of two clusters diagonally across from one another; these classes are separable but not by a single straight line. However, it is still possible to separate the classes using a linear boundary if the data are transformed into a higher dimensional space.

The input data, which are in input space, are mapped into a higher-dimensional feature space. If the number of dimensions is high enough, one can always find a linear boundary (hyperplane) that will separate the data without errors (Cover’s theorem).

Higher dimensions: Kernel Machines

The major drawback to this use of higher dimensions is that it does not generalize well unless a large number of data points is used in the training set. When the training set is limited (as is often the case), the higher dimensional space becomes very sparse, and the optimal boundary becomes hard to define. Additionally, training time increases considerably. These problems are referred to as “the curse of dimensionality”. On the other hand, in such a mapping the number of free parameters (the weights and bias) are no longer bound by the number of input variables (the dimension of the input space) but can be extended to any number of parameters. This is known as “decoupling” machine capacity from the input space dimensions.

Higher dimensional spaces are created using a kernel function \( k(x_i) \) that performs some nonlinear transformation of the original data \( x_i \) to create a set of new variables. Usually, this new set includes the original variables plus other, generated by the function. Popular kernel functions include polynomials, Gaussians, and trigonometric polynomials.

For instance, the quadratic kernel evaluates the square of all the original variables and also includes their cross-products. Therefore, for a two-variable data set consisting of \( x_1 \) and \( x_2 \), the new data set would consist of \( x_1, x_2, x_1 x_2, x_1^2, \) and \( x_2^2 \). The original 2D input space is transformed into a 5D feature space.
Higher dimensions: Kernel Machines

Ex. 10.7. Classify the data in Figure on slide 37 using the least squares linear classifier operating in a higher dimensional space.

Solution: The quadratic kernel can be used to separate data such as shown in that figure. Since the data are widely spread, only the cross-product term $x_1 x_2$ is needed to obtain perfect separation. This allows us to implement the classifier in three dimensions. The evaluation program `linear_eval3D` is a modified (3D) version of `linear_eval` seen before.

```matlab
% Program to use of higher dimensions to separate the data set shown in Figure on slide 37
close all; clear all;
distance = 6;
[X d] = gen_data2(distance, [], 'd');
[r c] = size(X);
X = [X, X(:,1).*X(:,2), ones(r,1)]; % Add the cross product term
w = inv(X'*X)*(X'*d'); % Train the classifier
[sensitivity, specificity, mse] = linear_eval3D(X,d,w);
```

Note that the implementation of the equation to train the classifier (produce weights) has the same MATLAB code in 3D as in 2D. The cross-products are formed by adding a column of $X(:,1) \cdot X(:,2)$.

After the training, the transformed data can be perfectly separated by a plane.
Support vector machines

The linear methods described so far work fairly well if the training sets are large and closely reflect the characteristics of the test data. However, small training sets may lead to errors...

The problem with the least squares method and other similar methods is that they base the decision boundary on all data points, which may put too much emphasis on the data points that are not critical.

The points that are closest to the other class are the most important in separating two classes.

An approach that maximizes the distance between these critical data points would likely produce better separation of the test set data. The points closest to boundary are called the support vectors.

A support vector classifier determines the boundary that maximizes the distance ("margin") between the critical support vectors.

Since support vector classifiers maximize the margin, they are also called maximum margin classifiers. Support vector machine (SVM) classifiers have become quite popular since they may produce excellent results in practical problems.

One key to the SVM classifier is finding the boundaries that maximize the margins. This step involves a classic optimization process. When the data are linearly separable, the goal is to find the hyperplane that maximizes the margin $M$, subject to the constraint that all the data points are on the appropriate side of the boundary (optimal canonical separating hyperplane, OCSH). In SVM analyses, the classes are identified as $\pm 1$ to simplify math. Therefore, the decision boundary is at $y = 0$ and using (10.13.1):

$$y = \sum_{i=1}^{M} x_i w_i + b = 0$$  \hspace{1cm} (10.42.1)

where $x_i$ are the input patterns, $w$ is the weight vector, and $b$ is the offset (bias). Since the two classes are defined by $y = \pm 1$ at the closest points (at the support vectors).
Support vector machines

The two lines passing through the support vectors produce the boundary for $y \geq \pm 1$

Therefore, the equations for the lines that go through these support vectors must be

$$x_i w + b \geq 1 \quad \Rightarrow \quad y = +1$$

$$x_i w + b \leq 1 \quad \Rightarrow \quad y = -1$$

(10.44.1)

Or, combined into a single equation:

$$y_i (x_i w + b) \geq 1$$

(10.44.2)

(10.43.2) states that $w$ and $b$ should be such that the two classes fall on the appropriate side of the support vector lines.

To determine the equation for the margin $M$, note that the distance of any hyperplane, $x w + b = 0$, to the origin is $-b/||w||$, where $||w||$ is the norm of $w$:

$$||w|| = \sqrt{w_1^2 + w_2^2 + \ldots + w_m^2} = \sqrt{w^T w}$$

(10.44.3)

If the hyperplane is equal to $\pm 1$, the distance to the origin is $\frac{\pm 1 - b}{||w||}$
Support vector machines

For the line that separates class 1 (i.e., $y_i \geq 1$), the distance to the origin is

$$d_0 = \frac{1 - b}{\|w\|} \tag{10.45.1}$$

For the line that separates class -1 (i.e., $y_i \leq -1$), the distance to the origin is

$$d_0 = \frac{-1 - b}{\|w\|} \tag{10.45.2}$$

The distance between these two lines is

$$M = \frac{1 - b}{\|w\|} - \frac{-1 - b}{\|w\|} = \frac{2}{\|w\|} \tag{10.45.3}$$

Therefore, the maximum margin is obtained by minimizing $\|w\|$.

Support vector machines

Alternatively, the margins can be maximized by minimizing $\|w\|^2$, which is equivalent to $w^Tw$ and is somewhat easier to perform. The minimization is achieved with the constraint (10.44.2) that insures that the boundaries are on the correct side. This type of minimization problem is called a quadratic programming (QP) optimization problem. A number of routines are available to solve QP problems.

If the data are not linearly separable and the points overlap, the optimization process still maximizes $M$, but the constraint is relaxed, so that some points can be on the wrong side of the boundary. This approach still generally leads to a better boundary than found with other linear classifiers.

The techniques described so far can only produce linear boundaries and are called linear support vector machines (LSVM). They are effective if the data are linearly separable. If not, two alternatives exist: either use linear boundary and accept some error, or transform the data into higher dimensions as previously described. The SVM approach can be applied to the transformed data to produce a hyperplane boundary. In the more general SVMs, the support vector classifier is combined with the use of higher dimensions, so that complex, nonlinear boundaries are obtained. Various kernels are used to transform the input space into a feature space that is symmetric – greatly simplifies the math.
Machine capacity: overfitting or “Less is more”

Several MATLAB-based classifiers implementing various classification methods are readily available on the web...

The problem for the user is to determine the classifier complexity required for a specific classification problem. In many cases, a linear classifier will perform better on a test set than a more complicated classifier. The problem is to fit the machine capacity (that is related to the complexity of the boundary) to the complexity of the data set.

An example of overfitting:
An SVM classifier with the 6th – order polynomial... where a 4th – order boundary would be sufficient. Higher machine capacity results in a more complicated decision boundary. The complexity was used to over-train on a few points in the training set that were not representative to the overall data set. As a result, the performance on the training set increased, the SVM performs poorly on the test set.

Cluster analysis

Cluster analysis is particularly popular for unsupervised classification, especially, when the number of classes is unknown. However, only two supervised versions of cluster analysis are discussed here.

1. The $k$-Nearest Neighbor (k-nn) classifier

The k-nn classifier is very simple but very successful in many practical problems instance-based learning algorithm. The classifier considers each test set point and determines the distances to the $k$ nearest points from the training set. Here, $k$ is a constant. Then, the classifier assigns the test point to the same class, where the majority of the $k$ neighbors belongs.

In other words, the average class value of the $k$ nearest points is rounded to an integer to determine the class value of the test point. For instance, if $k = 5$ and the training points that were closest to the test point had class values 0, 1, 1, 0, 0, the average would be $2/5 = 0.4$ and the class “0” would be assigned to the test point.

This approach can be used for any number of classes and any number of input variables.
Cluster analysis

The distance between points can be evaluated by a variety of metrics, although the most common and intuitive is the Euclidean distance:

\[ D = \sqrt{x_2 - x_1} = \left\| x_2 - x_1 \right\| \]  

(10.49.1)

Here, \( x_1 \) and \( x_2 \) are vectors of coordinates; usually, a second norm is used. It is also common to normalize each variable, so that it would have a variance of \( \pm 1 \). This way, the distance measurements are not skewed by a variable that extends over a much wider range than the other variables.

Ex. 10.8. Use the k-nn approach to classify a two-class test set of 500 points using a training set of 100 points. Use \( k \) of 5 and 15. Both sets are supplied in the MATLAB data file and are nonlinearly separable with some overlap. Both data sets have been normalized by setting their variance to \( \pm 1 \). Calculate the confusion matrix and plot the boundary.

Solution: for each test set input pattern, we will calculate the distance to all training points using the ML “norm” function; store these distances with their associated class values in a matrix; sort the matrix for increasing distance values with the ML “sortrows”; evaluate the mean of the first \( k \) class values and round to an integer that will be the class assigned to the test data point.

clear all; close all;  
K = 5;                     % Number of training points to test  
nu_classes = 2;             % Number of classes  
% Generate data  
% [Xt,dt] = gen_data2(4,1,'d');  %Generate training data  
% [X,d] = gen_data2(4,45,'d',0 1,400);  % Generate test data  
load Ex14_9.mat;       % Load data  
[r,c] = size(X);           % Test set size  
[rt,c] = size(Xt);          % Training set size  
% Find nearest neighbors  
class = zeros(1,1);         % Set up class assignment vector  
for j = 1:r  
    for i = 1:rt  
        Distance(1,i) = norm(X(j,:) - Xt(i,:));  
    end  
    Distance(2,i) = dt(i);  
    Distance = sortrows(Distance,1);  
    majority = mean(Distance(1:K,2));  
    class(j) = round(majority);  
    end  
end
Cluster analysis

% Evaluate errors and plot
hold on;
confusion = zeros(nu_classes);
for i = 1:r               % Count correct and incorrect, build confusion matrix and plot
    if class(i) == d(i)
        plot(X(i,1),X(i,2),marker_type(d(i)+1,:),’MarkerFaceColor’,marker_color(d(i)+1,:));
        confusion(d(i)+1,d(i)+1) = confusion(d(i)+1,d(i)+1) + 1;
    else
        plot(X(i,1),X(i,2),marker_type(d(i)+1,:),’MarkerFaceColor’,’k’);
        confusion(d(i)+1,class(i)+1) = confusion(d(i)+1,class(i)+1) + 1;
    end
end
k_nearest_boundaries(X,K,Xt,dt); % Plot boundaries
xlabel(’x1’,’FontSize’,14); ylabel(’x2’,’FontSize’,14);
axis([-1 1 -1 1]);
disp(confusion)

The boundaries are plotted with the “k_nearest_boundaries” function:

function k_nearest_boundaries(X,K,Xt,dt);
% Determines boundaries over a grid and plots
[r,c] = size(X);         % Size of test set
[r,t,c] = size(Xt);      % Size of training set
resolution = 100;        % Number of evaluated net output points per dimension
hold on;
marker_type = [’sk’; ’ok’; ’dk’; ’vk’; ’^k’];
if max(max(dt)) > 1
    marker_color = [’g’; ’y’; ’b’; ’r’; ’c’];  % More colorful makers
else
    marker_color = [’m’; ’c’; ’b’; ’r’; ’y’];
end
% Determine range for boundary plot
x1_max = max(max(X(:,1)));% x1_max = max(max(X(:,1)));, x1_min = min(min(X(:,1)));
x2_max = max(max(X(:,2)));% x2_min = min(min(X(:,2)));
x1_incr = (x1_max - x1_min)/resolution;
x2_incr = (x2_max - x2_min)/resolution;
i= 0;
% evaluate and plot the boundaries using an exhaustive search
for x1 = x1_min:x1_incr:x1_max
    i = i + 1;
    j = 0;
    x1_axis(i) = x1;
    for x2 = x2_min:x2_incr:x2_max
        j = j + 1;
        x2_axis(j) = x2;
        for r1 = 1:4
            Distance(1,r1) = norm([x1,x2] - Xt(r1,:));% Distance(1,r1) = norm([x1,x2] - Xt(r1,:));
            Distance(1,2) = d(r1);% Distance(1,2) = d(r1);
        end
        Distance = sortrows(Distance);% Distance = sortrows(Distance);
        majority = mean(Distance(1,K));% majority = mean(Distance(1,K));
y1(i,j) = round(majority);% y1(i,j) = round(majority);
    end
end
contour(x1_axis,x2_axis,y1’);
map = [0 1 0; 0 0 1; 1 0 0];
colormap(map);
%xlabel(’x1’,’FontSize’,14); ylabel(’x2’,’FontSize’,14);
Cluster analysis

The results. Boundaries:

The boundaries obtained using k-nn classifier applied to a non-linear overlapping data set.

Cluster analysis

The results. Confusion matrix:

<table>
<thead>
<tr>
<th></th>
<th>True Class</th>
<th>Predicted Class</th>
<th></th>
<th>True Class</th>
<th>Predicted Class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class 0</td>
<td>Class 1</td>
<td></td>
<td>Class 0</td>
<td>Class 1</td>
</tr>
<tr>
<td>k = 5</td>
<td>190</td>
<td>10</td>
<td>k = 15</td>
<td>196</td>
<td>4</td>
</tr>
<tr>
<td>Class 1</td>
<td>12</td>
<td>188</td>
<td>Class 1</td>
<td>11</td>
<td>189</td>
</tr>
</tbody>
</table>

The performance is slightly better with $k = 15$. The larger values of $k$ mean that more of the data are being considered, which improves generality. However, if $k$ is too large, boundary points can be misclassified if they are close to a large group of the other class. On the other hand, smaller values of $k$ can lead to misclassification due to a small number of outliers in the other class.
2. The $k$-Means Clustering classifier

The $k$-means clustering classifier is somewhat similar to k-nn classification method that represents the training data with a number of data centers called prototypes. In this approach, $k$ indicates the number of prototype centers. Once such prototype centers are established, the test data are assigned to the class of the closest prototype.

Therefore, the position of the prototypes determines the boundary; and the number of prototypes selected to represent each class determines the complexity of the boundary: the larger $k$, the more complicated is the boundary. The value of $k$ is directly related to machine capacity. The number of prototypes is selected by the user, and the prototype centers are positioned during a training period.

Several methods are known to training the prototypes (to find the best location to represent the data). We will consider the learning vector quantization (LVQ) method that is reasonably straightforward and fast.

In the LVQ method, initial prototypes are placed randomly within each class. During training, these prototypes are moved to better locations.

A random training data point is selected, and the closest prototype is found. If that prototype is of the same class as the training point, the prototype is moved toward the training point. If not, the prototype is moved away from the training point. The amount of movement is proportional to the distance, and the proportionality constant is called the learning rate constant. Once all training points have been used, the procedure begins again with a smaller learning rate constant. The procedure continues until the learning rate constant is zero.

Ex. 10.9. Use the $k$-means clustering approach to classify the test set used in Ex. 10.8. Use eight prototype centers to represent each of two classes ($k = 8$). After training, apply the classifier to the test set to evaluate the confusion matrix and plot the boundaries.

Solution: since the position of the data points are random, the initial clusters are positioned at the same location as the first 16 ($nu\_classes\_k$) training data points. Since gen\_data2 alternates the classes, $k$ points will be selected in each class.
### Cluster analysis

% Two nonlinearly separable classes using 5 cluster per class
clear all; close all;
k = 8;                                           % Number of clusters per class
nu_classes = 2;                                  % Number of classes
alpha = .25;                                     % Learning constant
nu_training = 1000;                              % Number of training cycles
marker_type = ['sk'; 'ok'; 'dk'; 'vk'; '^k'];    % Marker type for different classes
marker_color = ['m'; 'c'; 'b'; 'r'; 'y'];        % Marker color for different classes
load Ex14_9.mat;
[r,c] = size(Xt);

% Initialize cluster centers. Select centers randomly from the data set.
for i = 1:nu_classes*k
    Proto(i,:) = [Xt(i,:),dt(i)];
end

% Train clusters by moving prototype centers
for i = 1:nu_training
    [Xt,dt] = mix_data(Xt,dt);           % Randomize input sequence
    Proto = cluster_learn(Xt,dt,Proto,alpha);  % Train clusters
    alpha = alpha*(1 - i/nu_training);   % Reduce learning constant
end
[mse, correct, incorrect] = cluster_eval(Xt,dt,Proto);
disp([mse, correct, incorrect])
cluster_boundaries(Xt,dt,Proto);       % Draw cluster boundaries
figure; [mse, correct, incorrect, confusion] = cluster_eval(X,d,Proto);
cluster_boundaries(X,d,Proto);       % Draw cluster boundaries
axis([-1 1 -1 1]);
disp([mse, correct, incorrect])
disp(confusion)

---

Cluster analysis

The routine *cluster_learn* uses the previously discussed strategy to move the prototypes to better positions:

```matlab
function Proto = cluster_learn(X,d,Proto,alpha)
% Function to use Learning Vector Quantization to find best cluster centers
% Outputs:   Proto - prototype cluster center (matrix)
% Inputs:    X     - Inputs signals (matrix)
%            d     - desired response
%            Proto - initial cluster centers (matrix)
%            alpha - learning constant
%            k     - Number of clusters per class
%            nu_classes - Number of classes
%            alpha - Learning constant
%            nu_training - Number of training cycles
%            marker_type - Marker type for different classes
%            marker_color - Marker color for different classes

[r,c] = size(X);       % Determine number of input patterns
[r1,c1] = size(Proto);  % Determine number of prototype centers

% Train centers. Pick data training sequentially. Train over all observations
for i = 1:r
    for j = 1:r1        % Find distances to all prototypes centers
        distance(j) = norm(Proto(j,1:2)-X(i,:));
    end
    [dum,i_close] = min(distance);  % Find closest prototype center
    if d(i) == Proto(i_close,3)     % Closest center is same class
        Proto(i_close,1:2) = Proto(i_close,1:2) + alpha*(X(i,:) - Proto(i_close,1:2));
    else
        Proto(i_close,1:2) = Proto(i_close,1:2) - alpha*(X(i,:) - Proto(i_close,1:2));
    end
end
```

---

Cluster analysis

The routine *cluster_learn* uses the previously discussed strategy to move the prototypes to better positions:
Cluster analysis

The routine `cluster_eval` is used to evaluate the confusion matrix and plot the data:

```matlab
function [mse, correct, incorrect, confusion] = cluster_eval(X,d,Proto);
% Evaluates clusters defined by Proto
[r,c] = size(X); % Determine number of input patterns
[r1,c1] = size(Proto); nu_classes = max(d) + 1;
correct = 0; incorrect = 0; % Zero counters
for i = 1:r;
    for i1 = 1:r1; % Find closest prototype center
        distance(i1) = norm(Proto(i1,1:2)-X(i,:));
    end
    [dum,i_close] = min(distance);
    y(i) = Proto(i_close,3); % Predicted class
end
mse = mean((y-d).^2);
% Plots for results
clf; hold on; marker_type = ['sk'; 'ok'; 'dk'; 'vk'; '^k']; marker_color = ['m'; 'c'; 'b'; 'r'; 'y']; confusion = zeros(nu_classes);
for i = 1:r % Count correct and incorrect, build confusion and plot
    if y(i) == d(i)
        plot(X(i,1),X(i,2),marker_type(d(i)+1,:),'MarkerFaceColor',marker_color(d(i)+1,:));
        confusion(d(i)+1,d(i)+1) = confusion(d(i)+1,d(i)+1) + 1; correct = correct + 1;
    else
        plot(X(i,1),X(i,2), marker_type(d(i)+1,:),'MarkerFaceColor','k');
        confusion(d(i)+1,y(i)+1) = confusion(d(i)+1,y(i)+1) + 1; incorrect = incorrect + 1;
    end
end
% Draw cluster centers
for i = 1:r1
    plot(Proto(i,1),Proto(i,2),'ok','MarkerFaceColor',marker_color(Proto(i,3)+1),'MarkerSize',10);
    plot(Proto(i,1),Proto(i,2),'xk','MarkerSize',10,'LineWidth',2);
end
```

Cluster analysis

The boundaries are plotted by `cluster_boundaries`:

```matlab
function cluster_boundaries(X,d,Proto);
% Evaluates clusters defined by Proto
[r,c] = size(X); % Determine number of input patterns
[r1,c1] = size(Proto);
resolution = 100; %Number of evaluated net output points per dimension
correct = 0; incorrect = 0; % Initialize counters
figure; clf; hold on;
marker_type = ['sk'; 'ok'; 'dr'; 'vk'; '^k'];
if max(max(d)) > 1
    marker_color = ['g'; 'y'; 'b'; 'r'; 'c']; % More colorful markers
else
    marker_color = ['m'; 'c'; 'b'; 'r'; 'y'];
end
for i = 1:r
    for i1 = 1:r1
        distance(i1) = norm(Proto(i1,1:2)-X(i,:));
    end
    [dum,i_close] = min(distance);
    y(i) = Proto(i_close,3); % Predicted class
end
% Plots results
for i = 1:r
    if y(i) == d(i)
        plot(X(i,1),X(i,2),marker_type(d(i)+1,:),'MarkerFaceColor',marker_color(d(i)+1,:));
        correct = correct + 1;
    else
        plot(X(i,1),X(i,2), marker_type(d(i)+1,:),'MarkerFaceColor','k');
        incorrect = incorrect + 1;
    end
end
```
Cluster analysis

% Determine range for boundary plot
x1_max = max([max(X(:,1)), max(Proto(:,1))]);
x1_min = min([min(X(:,1)), min(Proto(:,1))]);
x2_max = max([max(X(:,2)), max(Proto(:,2))]);
x2_min = min([min(X(:,2)), min(Proto(:,2))]);
x1_incr = (x1_max - x1_min)/resolution;
x2_incr = (x2_max - x2_min)/resolution;

% Evaluate and plot the boundaries using an exhaustive search
for x1 = x1_min:x1_incr:x1_max
  for x2 = x2_min:x2_incr:x2_max
    distance(i) = sqrt((Proto(i,1)-x1)^2 + (Proto(i,2)-x2)^2);
  end
  [dum,i_close] = min(distance);
  y1(i) = Proto(i_close,3);
end

contour(x1_axis,x2_axis,y1');
map = [0 1 0; .5 .5 .5; 0 0 1; 1 0 0];
colormap(map);
xlabel('x1','FontSize',14); ylabel('x2','FontSize',14);

% Draw cluster centers
for i = 1:r1
  plot(Proto(i,1),Proto(i,2),marker_type(Proto(i,3)+1),'MarkerFaceColor',marker_color(Proto(i,3)+1),'MarkerSize',12,'LineWidth',2);
  plot(Proto(i,1),Proto(i,2),'xk','MarkerSize',12,'LineWidth',2);
end

Cluster analysis

The training points are selected in sequence, however, they are randomized before each pass by the routine mix_data that randomly rearranges the training matrix while keeping the correct associated classifications.

function [Xout,yout] = mix_data(X,y);
% Function to randomly mixes the order of the test set. Also used to generate a random sequence of numbers between 1 and r if X is a scalar and y is absent
[r,c] = size(X);
available = ones(r,1);
Xout = X; %Initialize outputs for better speed
yout = y;
for i = 1:r
  index = ceil(r*rand); %Random number between 1 and r
  while available(index) == 0; %Check if already used
    index = index + 1; %Increment index if used
    if index > r
      index = 1; %Wrap around increment
    end
  end
  Xout(i,:) = X(index,:); %Transfer to output
  if nargin == 2
    yout(i) = y(index);
  else
    yout(i) = 0;
  end
  available(index) = 0; %Make this index unavaillable
end
Cluster analysis

The evaluation routine `cluster_eval` classifies the data by searching for the closest prototype to each of the test data points. The test data are then classified as belonging to the same class as the prototype. This routine also plots the test data set, while flagging the errors (filled markers). Finally, the routine plots the final location of the prototypes as larger symbols marked with an “x”.

The boundaries determined by the k-means clustering classifier with $k = 8$ and for a two-class problem.

Cluster analysis

The confusion matrix:

<table>
<thead>
<tr>
<th>True Class</th>
<th>Predicted Class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class 0</td>
</tr>
<tr>
<td>Class 0</td>
<td>188</td>
</tr>
<tr>
<td>Class 1</td>
<td>9</td>
</tr>
</tbody>
</table>

Judging by the confusion matrix, this approach is slightly less accurate than one of the knn classifiers considered before.

Both cluster algorithms can be used for multiclass data with no modifications. Only the “nu_classes” parameter needs to be changed. K-mean classifier, 4-class data set:
Cluster analysis

The boundaries determined by the k-means clustering classifier with $k = 8$ and for a four-class problem. The test data set contained 400 points and was normalized. The corresponding confusion matrix is show in the previous slide.