Lab 1: Clusters and Classification Boundaries

A Report Submitted in Partial Fulfillment
of the Requirements for SYDE 372

David Kadish, 20176757
Zhao Peng, 20326604
Matt Stewart, 20205320

Faculty of Engineering
Department of Systems Design Engineering

February 9, 2009.
Course Instructor: Professor P. Fieguth

1

Introduction

The purpose of this lab was to apply pattern recognition classification algorithms and concepts to data sets in Matlab. Parameters such as number of data points, means, covariance matrices, and number of clustes were given. Randomized clusters were generated based on the class specifications, forming classification boundaries, and determining the probability of error based on those classification boundaries.

2

Implementation

The implementation for Lab 1 is done using MATLAB's class structures to maximize the reusability and allow for experiementation beyond the requirements of the lab. This is discussed further in Chapter 3.

2.1 Properties and class functions

MATLAB classes were created for parametric and non-parametric classifiers. The classes, `ParametricClass` and `NonParametricClass` are presented in Appendix A.

Properties

The two classes store properties related to the pattern recognition problems they represent. The `ParametricClass` stores for a class A the values of μ_A, Σ_A and $p(A)$. The `NonParametricClass` stores a cluster of n points in a Gaussian distribution with the parameters μ and Σ.

Class functions

Each class provides methods for calculating the various distance measures associated with the type of problem that it represents. The `ParametricClass` has functions for calculating d^2 using both MED and GED as well as a function for calculating the value of $p(A) \cdot P(x|A)$ as a measure of probability for MAP classification. The `NonParametricClass` contains a function for calculating distance to the class using kNN.

Calculations

Distance-squared by MED is calculated in the `ParametricClass` class in the `MED( point )` function. For a `ParametricClass` A and a point p,

$$d_{MED}^2 = (p - \mu_A)^T \cdot (p - \mu_A) \tag{2.1}$$

Distance-squared by GED is calculated in the `ParametricClass` class in the `GED( point )` function. For a `ParametricClass` A and a point p,

$$d_{GED}^2 = (p - \mu_A)^T \cdot \Sigma_A^{-1} \cdot (p - \mu_A) \tag{2.2}$$

The `MAP( point )` function does not really calculate distance at all. The value returned is one side of the Bayes Theorem inequality $\bar{x} \in A \iff p(\bar{x}|A) \cdot P(A)$ where A is the class calling the function. Equation 4.16 of the course notes gives one side of the inequality as

$$\begin{aligned} &P(A) \cdot \frac{1}{(2\pi)^{n/2}} \cdot \exp(-\frac{1}{2}(p - \mu_A)^T \cdot \Sigma_A^{-1} \cdot (p - \mu_A)) \\ =&P(A) \cdot \frac{1}{(2\pi)^{n/2}} \cdot \exp(-\frac{1}{2} \cdot d_{GED}^2) \end{aligned}$$

Since the $\frac{1}{(2\pi)^{n/2}}$ portion of the equation is constant, it can be removed from the comparison, giving the final weighted probability as

$$p_{weighted} = P(A) \cdot e^{-\frac{1}{2} \cdot d_{GED}^2} \tag{2.3}$$

Finally, kNN is calculated in `NonParametricClass` in the `kNN( point, k )` function. The point p is used to generate an $2 \times n$ matrix A where A_{1j} is the x-coordinate and A_{2j} is the y-coordinate of p. An $n \times n$ matrix with the distance-squared from each point in the class C is computed by the function

$$D = (A^T - C) \cdot (A^T - C)^T$$

The diagonal entries of D are converted to a vector, rooted and sorted. The kth element is then returned as the distance.

Plotting functions

The classes also provide helper functions for creating graphical representations of their data. `ParametricClass` has a function for plotting a the unit standard deviation curve and `NonParametricClass` contains a function for plotting the cluster of points the comprise the class.

2.2 Static methods

Classification

The classes include methods for classifying points based on the various distance and probability methods. With the exception of the MAP classifier, their functionality is similar. The logic is defined in Algorithm 2.1.

Algorithm 2.1 Classify a point based on distance to the classes

```
class number = 0
minimum distance = ∞
for i = 1 to n_classes do
  if distance to class i ≤ minimum distance then
    class number = i
    minimum distance = distance to class i
  end if
end for
```

The difference between the function for each classification method is the distance function that is called to determine the distance from the point to the class. In the

MAP class is that the search is for the highest weighted probability instead of the shortest distance. Otherwise the MAP classification algorithm is similar to the rest.

Class boundaries

Another static method included in the classes is a function to find the class boundaries using the different distance and probability methods. The functions classify an $n \times m$ set of points in the x-y plane to generate the class boundaries. The logic for these functions is defined in Algorithm 2.2.

Algorithm 2.2 Populate the matrix that determines class boundaries

```
C = an n × m matrix
for i = 1 to n do
  for j = 1 to m do
    C_ij = class of the point(x_i, y_J)
  end for
end for
```

The function returns an $n \times m$ matrix C with the elements $C_{ij} = \{1, 2 \ldots n_{classes}\}$. The contours of this are plotted on a graph to reveal the boundaries of the classes.

Algorithm 2.3 Calculate the confusion matrix for a given set of classes and test data

$$M_{confusion_{n,n}} = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}$$

```
for i = 1 to n_test_classes do
  for j = 1 to n_points_i do
    class = the evaluated class of point j
    add 1 to M_confusion at the cell (class, i)
  end for
end for
```

Class testing

Finally, the MATLAB classes provide two functions for testing; one for determining the confusion matrix and the other for calculating the probability of error ($P(\varepsilon)$)

given a confusion matrix. The confusion matrix calculators generate an $n \times n$ matrix with n being the number of classes in the space. Using classes C_i and test data T_i with $i \in \{1, 2, \ldots, n\}$, the method is defined in Algorithm 2.3.

Algorithm 2.3 returns the confusion matrix, $M_{confusion}$. The confusion matrix is then used to calculate $P(\varepsilon)$ as defined in Algorithm 2.4.

Algorithm 2.4 Calculate the probability of error from a confusion matrix

correct assignments = diag($M_{confusion}$)
incorrect assignments = $M_{confusion}$ - correct assignments
$P(\varepsilon) = \frac{\sum \text{elements of incorrect assignments}}{\sum \text{elements of } M_{confusion}}$

3

Results and Conclusions

3.1 Cluster Generation

The unit contour represents a collection of equally likely points in space. The elliptical unit standard deviation contours match the rough elliptical shape of the data clusters. The unit standard deviation contour does not enclose all data points. This is expected as the random data points that make up the clusters were generated on a normal distribution with a given mean and covariance; we would not expect all data points to be within one standard deviation of the mean.

3.2 Classification Boundaries

Parametric

Figure 3.2 demonstrates how the GED classifier was better than MED in the 2 class case. The MED classifier does not take the shape of the cluster (ie. the variances) into account and relies solely on the location of the cluster mean. GED, on the other hand, accounts for the variances and therefore generates a rotated classifier. In the 2 class case, the GED and MAP classifiers generate the same boundary because the variances of the two classes are equal.

The 3 class case in Figure 3.2 demonstrates the relative strength of the MAP

Figure 3.1: The clusters, unit standard deviation curves and parametric classification boundaries for the 2 class case.

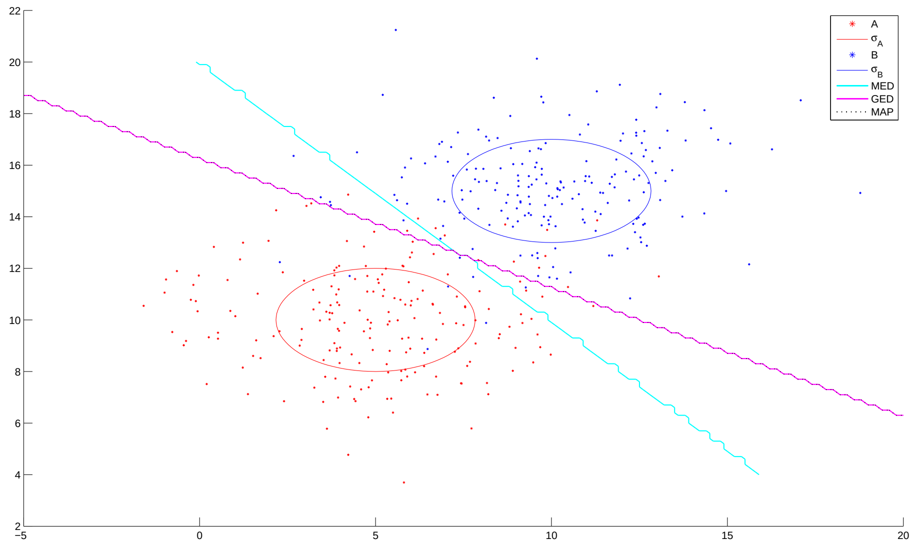


Figure 3.2: The clusters, unit standard deviation curves and parametric classification boundaries for the 3 class case.

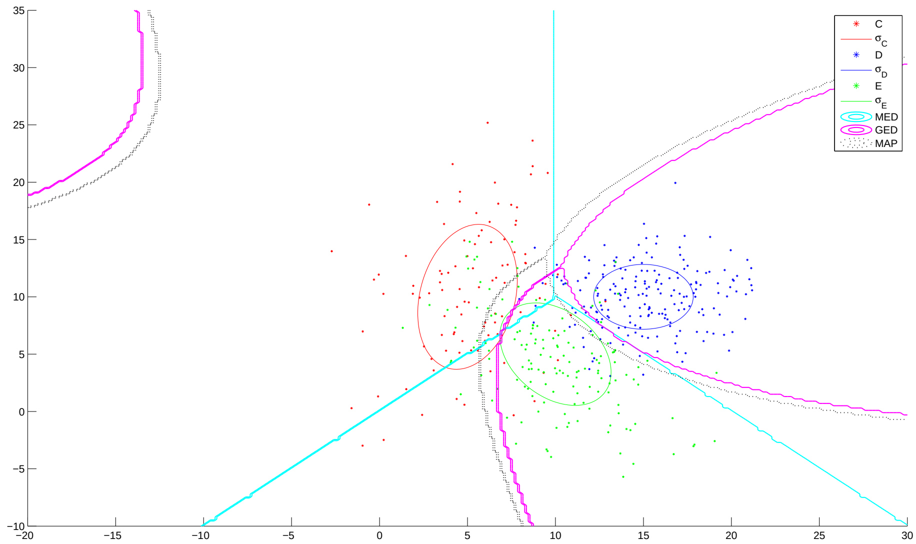
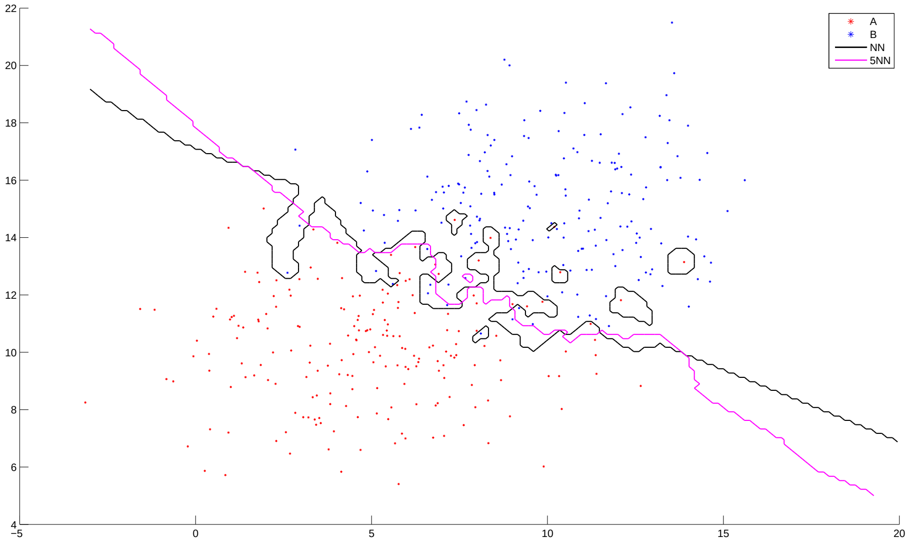


Figure 3.3: The clusters and non-parametric classification boundaries for the 2 class case.



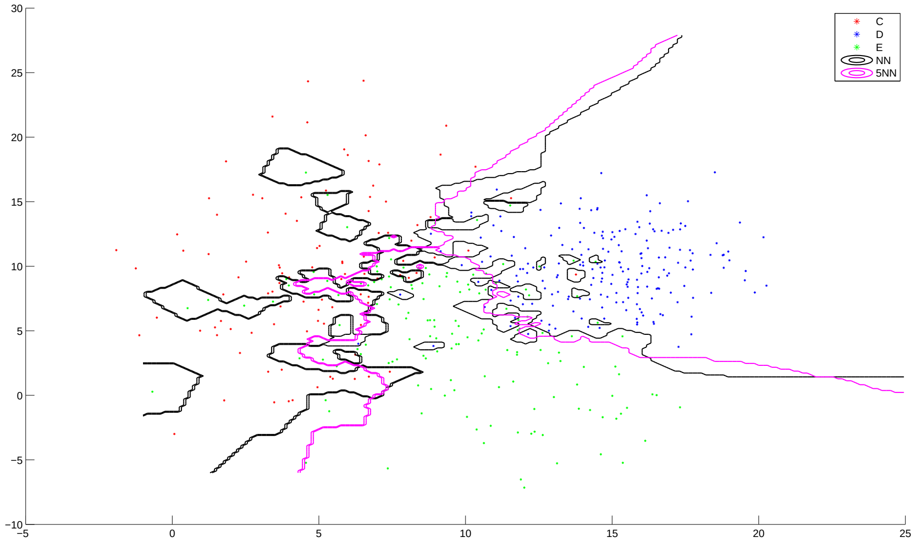
method. The MED boundary is simply an intersecting set of straight lines reflecting the points equidistant from the closest class averages. The GED and MAP boundaries have the same basic shape, but the MAP boundary provides a narrower avenue where it would classify a point as C or E. This reflects the relatively high value of $P(D)$ in the space.

Non-parametric

The 5NN classifier displays a much simpler boundary than the NN classifier. The key distinction is the fact that the 5NN is less sensitive to outliers because it ignores the four nearest points from each class. NN differs from 5NN in two major ways: Firstly, the NN classifier has more individual boundaries. Secondly, these boundaries are much less smooth than 5NN.

It is interesting to note in both the 2- and 3-class cases that the kNN boundary is beginning to look similar to the MAP boundary for the parametric class with the same μ and Σ as k goes from 1 to 5.

Figure 3.4: The clusters and non-parametric classification boundaries for the 3 class case.



3.3 Error Anaylsis

Table 3.1: Confusion matrix and probability of error for the 2 class case

	Test 1		Test 2	
	$M_{confusion}$	$P(\varepsilon)$	$M_{confusion}$	$P(\varepsilon)$
MED	$\begin{bmatrix} 188 & 15 \\ 12 & 185 \end{bmatrix}$	0.0675	$\begin{bmatrix} 182 & 22 \\ 18 & 178 \end{bmatrix}$	0.1000
GED	$\begin{bmatrix} 190 & 17 \\ 10 & 183 \end{bmatrix}$	0.0675	$\begin{bmatrix} 186 & 18 \\ 14 & 182 \end{bmatrix}$	0.0800
MAP	$\begin{bmatrix} 190 & 17 \\ 10 & 183 \end{bmatrix}$	0.0675	$\begin{bmatrix} 186 & 18 \\ 14 & 182 \end{bmatrix}$	0.0800
NN	$\begin{bmatrix} 175 & 22 \\ 25 & 178 \end{bmatrix}$	0.1175	$\begin{bmatrix} 178 & 27 \\ 22 & 173 \end{bmatrix}$	0.1225
5NN	$\begin{bmatrix} 187 & 16 \\ 13 & 184 \end{bmatrix}$	0.0725	$\begin{bmatrix} 187 & 23 \\ 13 & 177 \end{bmatrix}$	0.0900

In the 2 class case, the covariance matrices and probabilities of both classes are identical, making the covariance matrix result for GED and MAP equivalent. In

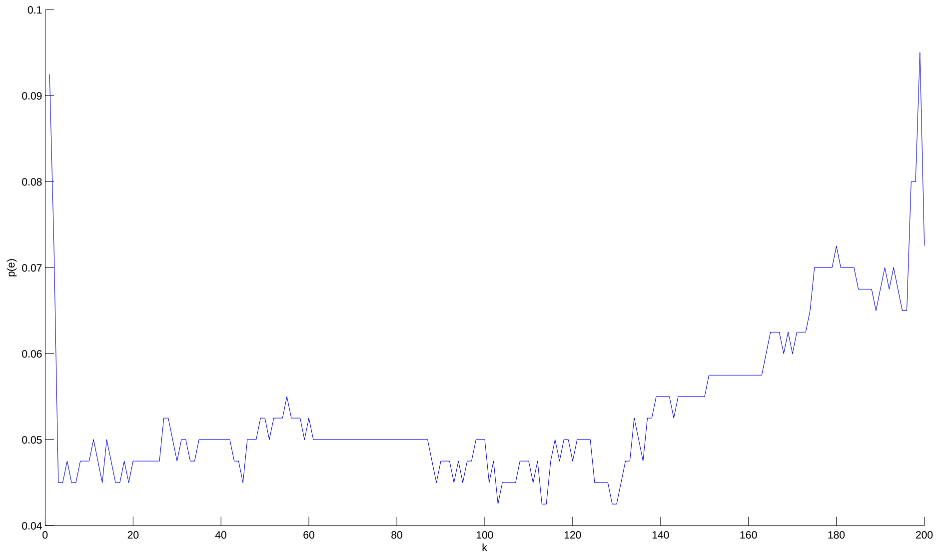
Table 3.2: Confusion matrix and probability of error for the 3 class case

	Test 1		Test 2	
	$M_{confusion}$	$P(\varepsilon)$	$M_{confusion}$	$P(\varepsilon)$
MED	$\begin{bmatrix} 78 & 1 & 29 \\ 4 & 183 & 16 \\ 18 & 16 & 105 \end{bmatrix}$	0.1867	$\begin{bmatrix} 80 & 4 & 22 \\ 2 & 175 & 21 \\ 18 & 21 & 107 \end{bmatrix}$	0.1956
GED	$\begin{bmatrix} 94 & 3 & 36 \\ 2 & 174 & 11 \\ 4 & 23 & 103 \end{bmatrix}$	0.1756	$\begin{bmatrix} 89 & 1 & 28 \\ 0 & 170 & 18 \\ 11 & 29 & 104 \end{bmatrix}$	0.1933
MAP	$\begin{bmatrix} 86 & 0 & 25 \\ 2 & 187 & 28 \\ 12 & 13 & 97 \end{bmatrix}$	0.1778	$\begin{bmatrix} 80 & 0 & 16 \\ 1 & 184 & 26 \\ 19 & 16 & 108 \end{bmatrix}$	0.1733
NN	$\begin{bmatrix} 73 & 2 & 36 \\ 4 & 188 & 26 \\ 23 & 10 & 88 \end{bmatrix}$	0.2244	$\begin{bmatrix} 66 & 1 & 21 \\ 3 & 172 & 28 \\ 31 & 27 & 101 \end{bmatrix}$	0.2467
5NN	$\begin{bmatrix} 76 & 0 & 22 \\ 4 & 187 & 25 \\ 20 & 13 & 103 \end{bmatrix}$	0.1867	$\begin{bmatrix} 75 & 0 & 18 \\ 0 & 174 & 20 \\ 25 & 26 & 112 \end{bmatrix}$	0.1978

the MAP calculation the $\ln(\Theta)$ goes to zero and the $\ln(\Sigma)$ terms cancelling out to zero. This leaves the exact GED formula, thus confirming the observed result. In the three class case, the three classes are not equally likely and therefore MAP generally provides a better classifier. MAP prefers classes that are more compact and have a higher probability density in a given region. From these observations, it can be gathered that, if the two classes have the same covariance and if both means fall on the line drawn by the axes of the other class, then MED, GED and MAP will all be the identical (and they will in fact be right bisectors).

Table 3.1 shows the results of two sets of test data for the two class case. It is interesting to note that in Test 1, the probability of error is the same for all three classifiers. Although GED and MAP generaly outperform MED, it is important to remember that for a given set of testing data this might not be the case. In this case, the test data is distributed so that the same number of erronous classifications was made with all three classifiers. Test 2, however shows that the GED and MAP

Figure 3.5: The probability of error as k increases for the 2 class case.



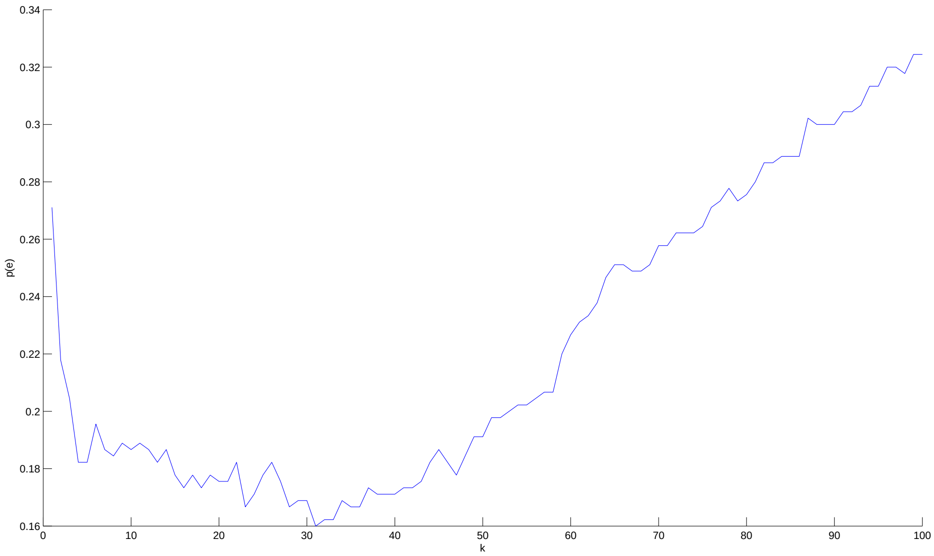
classifiers outperform the MED classifier.

Table 3.2 shows a similar result for the 3 class case. In Test 1, GED outperforms both MED and MAP in terms of error rate - albeit only slightly for MAP. Test 2, however, shows GED performing only slightly better than MED while MAP outperforms both by a significant margin.

For the non-parametric case, kNN has smaller error than NN as shown in Tables 3.1 and 3.2. This is attributed to the fact that the former is less sensitive to outliers in the training data. It is observed from the confusion matrices that the elements in (1, 2) and (2, 1) are much smaller than the rest of the off-diagonal elements. This provides a good indication that Class C and D probably have comparitively very little overlap with each other. Observations such as this can provide some intuition about the location of the classes simply from seeing an error analysis.

To explore the effect of the choice of k on the probability of error, a simple for loop was created to calculate the probability of error for the range of possible k ($0 < k < n_{smallest\_class}$). The graphs of a sample run of this code are depicted in Figures 3.3 and 3.3 The graphs generally show a sharp initial decrease as outliers are

Figure 3.6: The probability of error as k increases for the 3 class case.



ignored. After the initial drop, the graphs tend to behave slightly differently for the 2- and 3-class cases. For the 2-class case, the probability of error remains relatively stable for the majority of the values of k, but begins to increase as k approaches 75% of its maximum value. For the 3-class case, however, it seems that the probability of error begins to climb shortly following the initial drop. This suggests that class structure is eroded more quickly due to the exclusion of good data for the 3-class case. From this result, we might suspect that this trend would hold as the number of classes increases.

Appendix A

Code

A.1 PlotElipse.m

```
function PlotEllipse(x,y,theta,a,b, colour)

if nargin<5, error('Too few arguments to Plot Ellipse.'); end;

np = 100;
ang = [0:np]*2*pi/np;
pts = [x;y]*ones(size(ang)) + [cos(theta) -sin(theta); sin(theta) cos(theta)]*[cos(ang)*a; sin(ang
    )*b];
plot( pts(1,:), pts(2,:), colour);
```

A.2 ParametricClass.m

```
classdef ParametricClass
    %Class containing a Pattern Rec Classification Class

    properties
        Mu
        Sigma
        Probability
    end

    methods
        %% Initialzation
        function PC = ParametricClass(mu, sigma, prob)
            PC.Mu = mu;
            PC.Sigma = sigma;
            PC.Probability = prob;
        end

        %% Plotting
        function t = TestData(PC, n_pts)
            t = NonParametricClass(PC.Mu, PC.Sigma, n_pts);
```

```
        end

        function PlotStdDev(PC, colour)
            x=PC.Mu(1);
            y=PC.Mu(2);

            [V,D]= eig(PC.Sigma);

            rta = sqrt(D(1,1));
            rtc = sqrt(D(2,2));

            theta = atan(V(2,1)/V(1,1));

            PlotEllipse(x,y,theta,rta,rtc, colour)
        end

        %% Distance Calculations
        % MED
        function d = MED(PC, point)
            d = (point - PC.Mu)'*(point - PC.Mu); %dist squared
        end

        % GED
        function d = GED(PC, point)
            d = (point - PC.Mu)'*PC.Sigma^(-1)*(point - PC.Mu); %dist squared
        end

        function p = MAP(PC, point)
           p = PC.Probability * sqrt(2 * pi * det(PC.Sigma))^(-1) * exp(-0.5 * PC.GED(point)); % probability-ish
        end
    end

    %% Static Methods
    methods (Static = true)
        %% Classification Methods
        % Classify based on MED
        % Use: ParametricClass.ClassifyMED( unknown_point, {Class1 Class2})
        % Returns: The index of the selected class.
        function c = ClassifyMED(point, classes)
            c = 0; %The class index
            d = Inf; %The distance
            for i = 1:length(classes)
                if classes{i}.MED(point) <= d
                   c = i;
                   d = classes{i}.MED(point);
                end
            end
        end

        % Classify based on GED
        % Use: ParametricClass.ClassifyGED( unknown_point, {Class1 Class2})
        % Returns: The index of the selected class.
        function c = ClassifyGED(point, classes)
            c = 0; %The class index
            d = Inf; %The distance
            for i = 1:length(classes)
                if classes{i}.GED(point) <= d
                   c = i;
                   d = classes{i}.GED(point);
                end
```

```
            end
        end

        % Classify based on MAP
        % Use: ParametricClass.ClassifyMAP( unknown_point, {Class1 Class2}, {P1 P2})
        % Returns: The index of the selected class.
        function c = ClassifyMAP(point, classes)
            c = 0; %The class index
            p = 0;
            for i = 1:length(classes)
                if classes{i}.MAP(point) >= p
                   c = i;
                   p = classes{i}.MAP(point);
                end
            end
        end

        %% Boundary Plotting Methods
        % Plot boundary based on MED
        function map = BoundMatrixMED(classes, x_pts, y_pts)
           map = zeros(length(x_pts),length(y_pts));
           for i = 1:length(x_pts)
               for j = 1:length(y_pts)
                   map(i,j) = ParametricClass.ClassifyMED([x_pts(i) y_pts(j)]', classes);
               end
           end
        end

        % Plot boundary based on GED
        function map = BoundMatrixGED(classes, x_pts, y_pts)
           map = zeros(length(x_pts),length(y_pts));
           for i = 1:length(x_pts)
               for j = 1:length(y_pts)
                   map(i,j) = ParametricClass.ClassifyGED([x_pts(i) y_pts(j)]', classes);
               end
           end
        end

        % Plot boundary based on MAP
        function map = BoundMatrixMAP(classes, x_pts, y_pts)
           map = zeros(length(x_pts),length(y_pts));
           for i = 1:length(x_pts)
               for j = 1:length(y_pts)
                   map(i,j) = ParametricClass.ClassifyMAP([x_pts(i) y_pts(j)]', classes);
               end
           end
        end

        %% Testing Methods
        % Generate confusion matrix based on MED
        function conf = ConfusionMatrixMED(classes, test_data)
            conf = zeros(length(classes));

            %populate test classes and confusion matrix
            for i=1:length(classes)
                td_size = size(test_data{i}.Cluster);
                for j=1:td_size(1)
                    c = ParametricClass.ClassifyMED(test_data{i}.Cluster(j, :)', classes);
                    conf(c,i) = conf(c,i) + 1;
                end
            end
```

```
        end

        % Generate confusion matrix based on GED
        function conf = ConfusionMatrixGED(classes, test_data)
            conf = zeros(length(classes));

            %populate test classes and confusion matrix
            for i=1:length(classes)
                td_size = size(test_data{i}.Cluster);
                for j=1:td_size(1)
                    c = ParametricClass.ClassifyGED(test_data{i}.Cluster(j, :)', classes);
                    conf(c,i) = conf(c,i) + 1;
                end
            end

        end

        % Generate confusion matrix based on MAP
        function conf = ConfusionMatrixMAP(classes, test_data)
            conf = zeros(length(classes));

            %populate test classes and confusion matrix
            for i=1:length(classes)
                td_size = size(test_data{i}.Cluster);
                for j=1:td_size(1)
                    c = ParametricClass.ClassifyMAP(test_data{i}.Cluster(j, :)', classes);
                    conf(c,i) = conf(c,i) + 1;
                end
            end

        end

        function prob = ErrorProbability(confusion)
            correct = diag(diag(confusion));
            incorrect = confusion - correct;
            prob = sum(sum(incorrect)) / sum(sum(confusion));
        end
    end
end
```

A.3 NonParametricClass.m

```
classdef NonParametricClass
    %NonParametricClass Contains a non-parametric class
    %   Holds a set of points that form a cluster

    properties
        Cluster
    end

    methods
        %% Initialization
        function NPC = NonParametricClass(mu, sigma, n_pts)
           NPC.Cluster = mvnrnd(mu, sigma, n_pts);
        end

```

```
        %% Plotting
        function PlotCluster(NPC, colour)
            Y_1=NPC.Cluster*[1;0];
            Y_2=NPC.Cluster*[0;1];
            scatter(Y_1, Y_2, 5, strcat('*', colour))
        end

        %% Distance
        function d = kNN(NPC, point, k)
            d = inf;
            s_cluster = size(NPC.Cluster);
            p = repmat(point,1,s_cluster(1));
            d_matrix = sort(sqrt(diag((p' - NPC.Cluster)*(p' - NPC.Cluster)')));
            d = d_matrix(k);
        end
    end

    %% Static Methods
    methods (Static = true)
        % Classify based on kNN
        % Use: NonParametricClass.ClassifyKNN( unknown_point, {Class1 Class2}, k)
        % Returns: The index of the selected class.
        function c = ClassifyKNN(point, classes, k)
             c = 0; %The class index
             d = Inf; %The distance
             for i = 1:length(classes)
                 if classes{i}.kNN(point, k) <= d
                    c = i;
                    d = classes{i}.kNN(point, k);
                 end
             end
        end

        %% Boundary Plotting Methods
        % Plot boundary based on KNN
        function map = BoundMatrixKNN(classes, k, x_pts, y_pts)
            map = zeros(length(x_pts),length(y_pts));
            for i = 1:length(x_pts)
                for j = 1:length(y_pts)
                    map(i,j) = NonParametricClass.ClassifyKNN([x_pts(i) y_pts(j)]', classes, k);
                end
            end
        end

        %% Testing Methods
        % Generate confusion matrix based on kNN
        function conf = ConfusionMatrixKNN(classes, test_data, k)
             conf = zeros(length(classes));

            %populate test classes and confusion matrix
             for i=1:length(classes)
                 td_size = size(test_data{i}.Cluster);
                 for j=1:td_size(1)
                     c = NonParametricClass.ClassifyKNN(test_data{i}.Cluster(j, :)', classes, k);
                     conf(c,i) = conf(c,i) + 1;
                 end
             end

        end

        function prob = ErrorProbability(confusion)
```

```
            correct = diag(diag(confusion));
            incorrect = confusion - correct;
            prob = sum(sum(incorrect)) / sum(sum(confusion));
        end
    end
end
```

A.4 Tools.m

```
classdef Tools
    %Tools Summary of this class goes here
    %   Detailed explanation goes here

    properties
    end

    methods (Static = true)
        function ParametricPlot(classes, colours, n_pts, x_range, y_range, contours, names)

            % Plot the clusters and the unit standard deviations
            for i=1:length(classes)
                classes{i}.TestData(classes{i}.Probability * n_pts).PlotCluster(colours{i})
                hold on;
                classes{i}.PlotStdDev(colours{i})
                hold on;
            end

            % Calculate and plot the boundaries
            m = ParametricClass.BoundMatrixMED(classes, x_range, y_range);
            g = ParametricClass.BoundMatrixGED(classes, x_range, y_range);
            p = ParametricClass.BoundMatrixMAP(classes, x_range, y_range);

            bounds = {m g p};
            bound_styles = {'cyan' 'magenta' ':black'};

            for i=1:length(bounds)
                contour(x_range, y_range, bounds{i}', contours, bound_styles{i}, 'LineWidth', 1)
                hold on;
            end

            legend(names)

        end

        function NonParametricPlot(classes, colours, n_pts, x_range, y_range, contours, names)
            % Create the NP Classes and plot the clusters
            np_classes = {};
            for i=1:length(classes)
                np_classes{i} = classes{i}.TestData(classes{i}.Probability * n_pts);

                np_classes{i}.PlotCluster(colours{i})
                hold on;
            end

            % Compute the boundaries
            n = NonParametricClass.BoundMatrixKNN(np_classes, 1, x_range, y_range);
            k = NonParametricClass.BoundMatrixKNN(np_classes, 5, x_range, y_range);
```

```
            bounds = {n k};
            bound_styles = {'black' 'magenta'};

            for i=1:length(bounds)
                contour(x_range, y_range, bounds{i}', contours, bound_styles{i}, 'LineWidth', 1)
                hold on;
            end

            legend(names)
        end

        function Testing(classes, n_pts)
            np_classes = cell(size(classes));
            test_data = cell(size(classes));

            for i=1:length(classes)
                np_classes{i} = classes{i}.TestData(n_pts{i}); %n_pts
                test_data{i} = classes{i}.TestData(n_pts{i});
            end

            conf_MED = ParametricClass.ConfusionMatrixMED(classes, test_data)
            prob_MED = ParametricClass.ErrorProbability(conf_MED)

            conf_GED = ParametricClass.ConfusionMatrixGED(classes, test_data)
            prob_GED = ParametricClass.ErrorProbability(conf_GED)

            conf_MAP = ParametricClass.ConfusionMatrixMAP(classes, test_data)
            prob_MAP = ParametricClass.ErrorProbability(conf_MAP)

            conf_NN = NonParametricClass.ConfusionMatrixKNN(np_classes, test_data, 1)
            prob_NN = NonParametricClass.ErrorProbability(conf_NN)

            conf_kNN = NonParametricClass.ConfusionMatrixKNN(np_classes, test_data, 5)
            prob_kNN = NonParametricClass.ErrorProbability(conf_kNN)
        end
    end
end
```

A.5 lab1.m

```
%% File Info
%SYDE 372 Lab 1 - Clusters and Classification Boundaries
%Feb 5, 2009

clear

%% Set Up Classes
A = ParametricClass([5;10], [8 0; 0 4], 0.5);
B = ParametricClass([10;15], [8 0; 0 4], 0.5);
C = ParametricClass([5;10], [8 4; 4 40], 100/450);
D = ParametricClass([15;10], [8 0; 0 8], 200/450);
E = ParametricClass([10;5], [10 -5; -5 20], 150/450);

%% CASE 1: A,B
% PLOTS
% Plot clusters and standard deviations
```

```
figure;

n_pts = 400;
colours = {'r' 'b'};
classes = {A B};

x_range = -5:0.2:20;
y_range = 4:0.2:20;
contours = 1.5;

Tools.ParametricPlot(classes, colours, n_pts, x_range, y_range, contours, {'A' '\sigma_A' 'B' '\sigma_B' 'MED' 'GED' 'MAP'})

figure;

x_range = -3:0.15:20;
y_range = 5:0.15:23;

Tools.NonParametricPlot(classes, colours, n_pts, x_range, y_range, contours, {'A' 'B' 'NN' '5NN'})

% TESTING
Tools.Testing(classes, {200 200})

%% CASE 2: C,D,E
% PLOTS
% Plot clusters and standard deviations
figure;

n_pts = 450;
colours = {'red' 'blue' 'green'};
classes = {C D E};

% Plot bounds
x_range =  -20:0.2:30;
y_range = -10:0.2:35;
contours = [1.5 2.5];

Tools.ParametricPlot(classes, colours, n_pts, x_range, y_range, contours, {'C' '\sigma_C' 'D' '\sigma_D' 'E' '\sigma_E' 'MED' 'GED' 'MAP'})

figure;

%Plot KNN and NN
x_range =  -1:0.15:25;
y_range = -6:0.15:28;

Tools.NonParametricPlot(classes, colours, n_pts, x_range, y_range, contours, {'C' 'D' 'E' 'NN' '5NN'})

% TESTING
Tools.Testing(classes, {100 200 150})

%% Extra Fun Stuff
% % Investigating effect of choice of k on probability of error
% classes = {A B};
% n_pts = {200 200};
%
% np_classes = cell(size(classes));
% test_data = cell(size(classes));
% conf_kNN_play = cell(200);
%
```

```
% for i=1:length(classes)
%     np_classes{i} = classes{i}.TestData(n_pts{i}); %n_pts
%     test_data{i} = classes{i}.TestData(n_pts{i});
% end
%
% for k = 1:200
%     conf_kNN_play{k} = NonParametricClass.ConfusionMatrixKNN(np_classes, test_data, k);
%     prob_kNN_play(k) = NonParametricClass.ErrorProbability(conf_kNN_play{k});
% end
%
% figure
% line(1:200,prob_kNN_play);
%
% classes = {C D E};
% n_pts = {100 200 150};
%
% np_classes = cell(size(classes));
% test_data = cell(size(classes));
% conf_kNN_play = cell(100);
%
% for i=1:length(classes)
%     np_classes{i} = classes{i}.TestData(n_pts{i}); %n_pts
%     test_data{i} = classes{i}.TestData(n_pts{i});
% end
%
% for k = 1:100
%     conf_kNN_play{k} = NonParametricClass.ConfusionMatrixKNN(np_classes, test_data, k);
%     prob_kNN_play2(k) = NonParametricClass.ErrorProbability(conf_kNN_play{k});
% end
% size(1:100)
% size(prob_kNN_play2)
% figure
% line(1:100,prob_kNN_play2);
```