Lab 2: Model Estimation and Descriminant Functions

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

March 16, 2009.

Course Instructor: Professor P. Fieguth

1

Introduction

The previous report explored the idea of classification; how to categorize data based on pre-defined models. This report investigates the process of creating models based on pre-classified data.

Section 2.1 explores the creation of 1-dimesional models. Section 2.2 extends that work to 2-dimensions. Section 2.3 describes the process of using sets of linear discriminants to create a sequential classifier.

2

Implementation and Results

2.1 1D Model Estimation

Implementation

The 1-dimensional models are run from `lab2p1.m` in Section A.1. The models themselves are represented by the `OneD` class from the `OneD.m` file in Section A.2.

Results

Each data set yielded varying results with the different approximation methods. The Gaussian samples, shown in Figures 2.1, 2.2 and 2.3, the parametric estimation assuming the unknown density is Gaussian is closest to the original. For the exponential samples in Figures 2.4, 2.5 and 2.6, the parametric estimation assuming theunknown density is exponential is closest to the original.

The Parzen window method, depicted in Figures 2.7 and 2.8 does a better estimation than the parametric methods when the model assumption does not match the real distribution. This is due to the fact that Parzen windows do not make assumptions about the distribution, they simply model the points that they find. With a wider window such as in Figure 2.8, the estimated density can be made smoother. However, trade-off is made between the smoothness of the estimated PDF and its sensitivity to sample data.

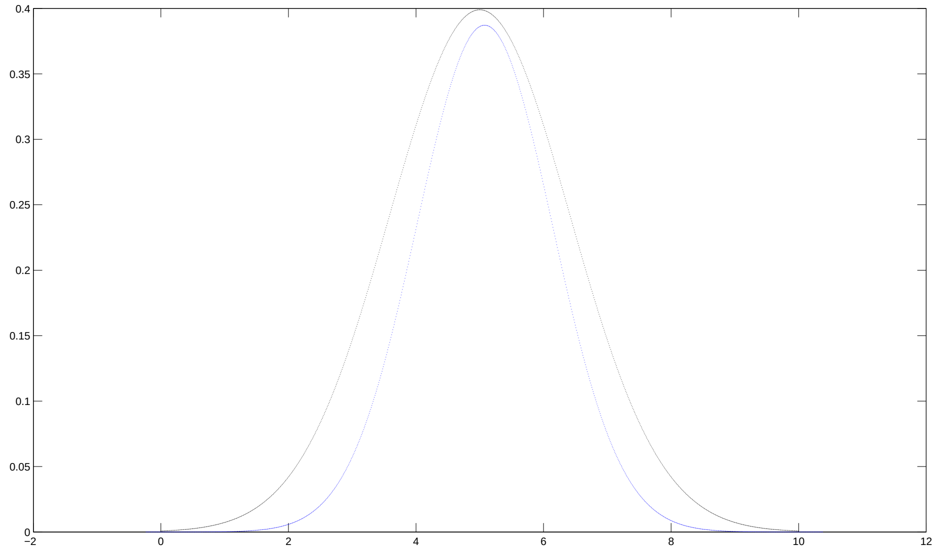

Figure 2.1: Gaussian sample (black) estimated assuming a Gaussian PDF (blue)

It is not always possible to use a parametric approach. Parametric approaches require estimation of the parameters using methods such as ML that requires us to solve some equations. However, sometimes these equations could be extremely difficult to solve if the PDF of the assumed distribution are not in simple form. Besides, it is likely that the sample data do not follow any known distributions, especially when the number of the sample data is very small. Therefore, it is sometimes hard if not impossible to use a pararametric approach.

A parametric method is preferred if the sample data follows the assumed distribution closely and the parameters can be easily computed. In contrast, a non-parametric approach is preferable if the sample data do not follow any particular known distribution or the number of samples are very small.

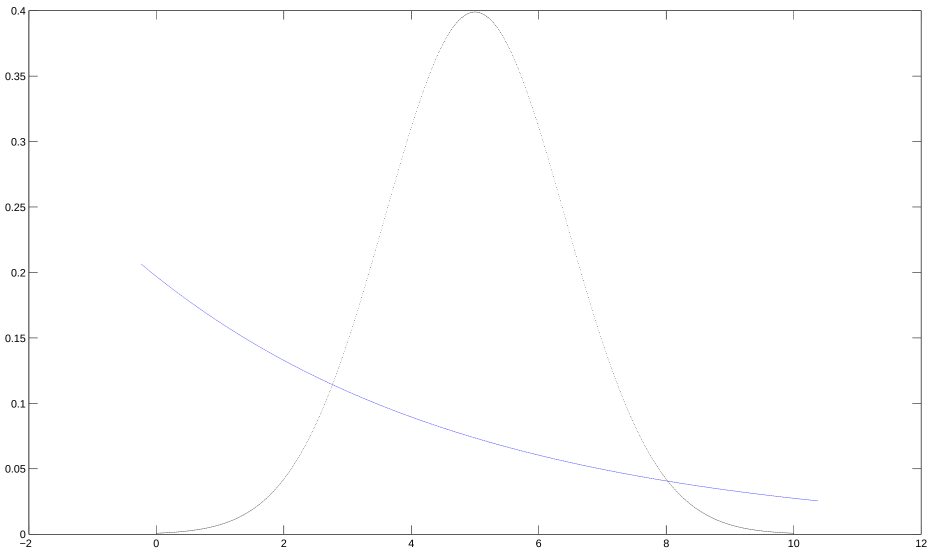

Figure 2.2: Gaussian sample (black) estimated assuming an exponential PDF (blue)

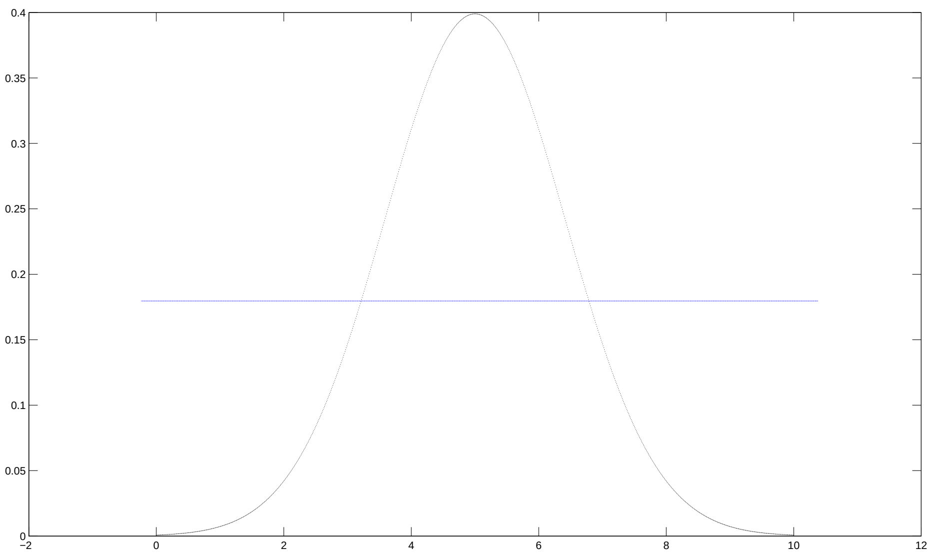

Figure 2.3: Gaussian sample (black) estimated assuming a uniform PDF (blue)

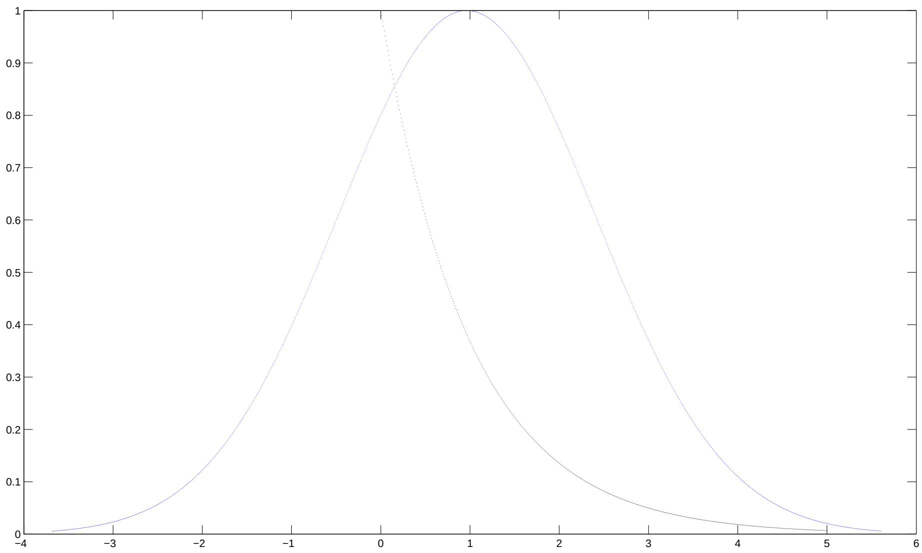

Figure 2.4: Exponential sample (black) estimated assuming a Gaussian PDF (blue)

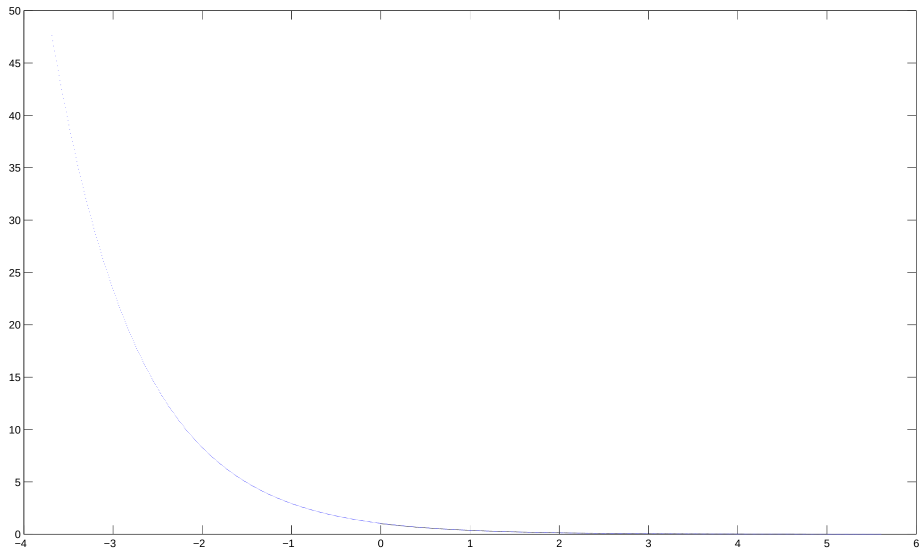

Figure 2.5: Exponential sample (black) estimated assuming an exponential PDF (blue)

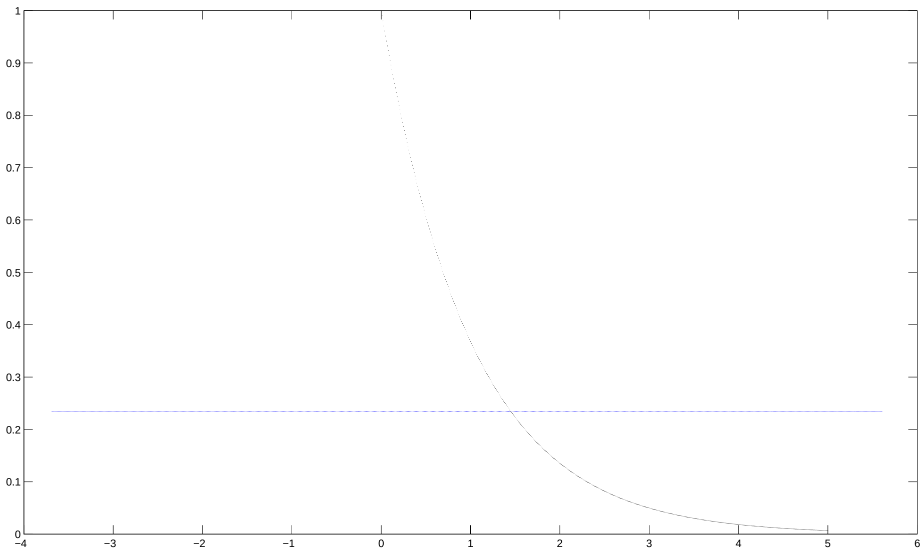

Figure 2.6: Exponential sample (black) estimated assuming a uniform PDF (blue)

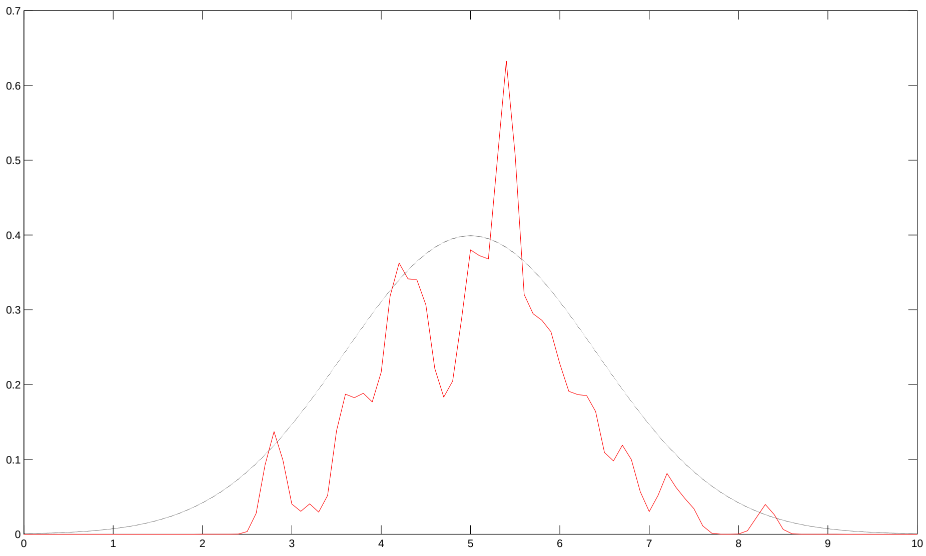

Figure 2.7: Gaussian sample (black) estimated using Gaussian Parzen windows with $\sigma = 0.1$ (red)

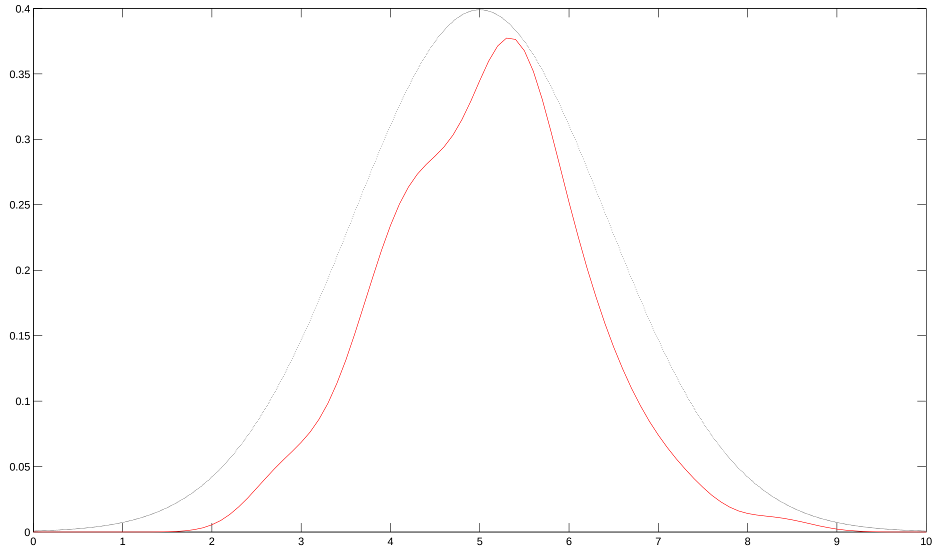

Figure 2.8: Gaussian sample (black) estimated using Gaussian Parzen windows with $\sigma = 0.4$ (red)

2.2 2D Model Estimation

Implementation

The non-parametric estimation was implemented first for the 2 case. A 2D Gaussian matrix was created, using the specified variance of $\sigma^2 = 400$ over a range of x-values from -60 to 60. Similar to the previous lab, class structure was used to streamline the code. Each of the data clusters, al, bl, cl, were passed into the function TwoD, where the mean and covariance were estimated. The provided parzen methods, included as part of the OneD class, was then called to estimate the PDF of each of the clusters, using the previously defined 2D Gaussian matrix and and resolution. The returned parameters p, x, and y were assigned to ap, ax, and ay for Cluster A, bp, bx, and by for Cluster B and so on. These estimated PDFs for each cluster (ap, bp, cp) were then passed into the ML function to form the classification boundaries. The ML function steps through each sqare on the grid defined by the resolution, looking for the cluster with the highest probability each time. A value of 0 is assigned to all elements of the grid ML for which Class A had a higher estimated probability, a value 1 for elements in which Class B was most likely, and a value 2 for elements in which Class C was most likely. The ML boundaries were then plotted using a contour plot.

The parametric estimation was implemented to mimic the inputs and outputs of the parzen function in order to simplify the code. The parametric methods of the TwoD class was then called to estimate the PDF of the cluster. It used the previously defined resolution as well as the estimated mean and covariance parameters from earlier. a matrix, p, of probability densities was the result, along with x and y bounds matching the parzen method. These values, p, x, and y were then passed into the ML method, in the exact same manner as with the non-parametric case and the resulting ML boundaries were plotted using a contour plot.

Results

For the two dimensional case, there are three clusters of data, all with different shapes. Parametric estimation was performed, assuming that each cluster was Gaussian distributed. The mean and variance were learned from the data and the resulting classification boundaries were plotted. These parametric boundaries do a reasonable job of differentiating the classes from each other. For example, the majority of Class A data points will indeed be classified as A. The classification boundaries themselves, however, do not track the boundaries of the classes particularly well. The classification boundary for Class C (represented by red stars), for example, appears to be shifted up and to the left of where we would intuitively like it to be. The boundary has not physically been shifted anywhere of course, but rather this apparent "shift" is due to the fact that the classes are not truly Gaussian in shape. The classification boundaries are based on statistics that match an assumed Gaussian model. The effects of assuming the shape of the PDF can be further seen by examining the classification boundary between Class A and B (red + and blue diamond). Class B is somewhat crescent-shaped, enveloping a large portion of Class A. The ML classification boundary, based on estimated Gaussian PDFs, does quite a poor job of capturing this intrusion of Class A into Class B. The assumed Gaussian shape causes the boundary to cut-off a some of the "arms" of B and misclassifying them as A.

In the nonparametric case, no assumptions are made about shape, and the PDFs are driven directly by the data. The flexibility allows the boundaries to fit the data considerably better. A Gaussian parzen window with $\sigma^2 = 400$ was used to estimate the PDF for each cluster. Applying ML to these estimated PDFs, the classification boundary between Class B and Class C no longer misclassifies any data points. The apparent "shift" observed for the parametric estimation case is gone. This shows the advantage that nonparametric estimation holds over parametric estmation. By making no assumptions about the shape of the PDF, nonparametric estimation is very capable of handling slight variations from a Guassian. Considering the classification boundary between Class A and B in the nonparametric case, we see the true power of nonparametric estimation. Despite the fact that one class intrudes fairly deep into the other class, the nonparametric estimation is able to form a rather good classification boundary to separate the two classes. Again, this performance is due to the flexibility of nonparametric estimation and the fact that no assumptions about the shape of the PDF were made. The boundary here is very smooth and, except for a few outliers, tracks the actual boundary of the clusters very closely.

In general, it is possible to always use a parametric approach to parameter estimation. The parameters required for the chosen model can always be estimated from the sample data. However, they will not necessarily give a good approximation to the true classe shape. Poor choice of the parametric model could lead to very poor classification boundaries. This is evident in Figure 2.9, where the boundary associated with Class B is quite poor due to the fact that is not a Gaussian distributed class.

In summary, it is better to use a parametric method of parameter estimation in the event that the cluster data fits a known model very closely. If this is the case, the estimated parameters will provide a PDF that very closely resembles the actual class statistics. The nonparametric approach is preferred when the cluster data does not fit into one of the standard, known models very well. In these instances, the flexibility of the nonparametric approach allows it to yield a PDF that closely matches the actual cluster data without requiring the data to match some model. It should also be noted that the nonparametric will still perform well when the cluster data does fit a known

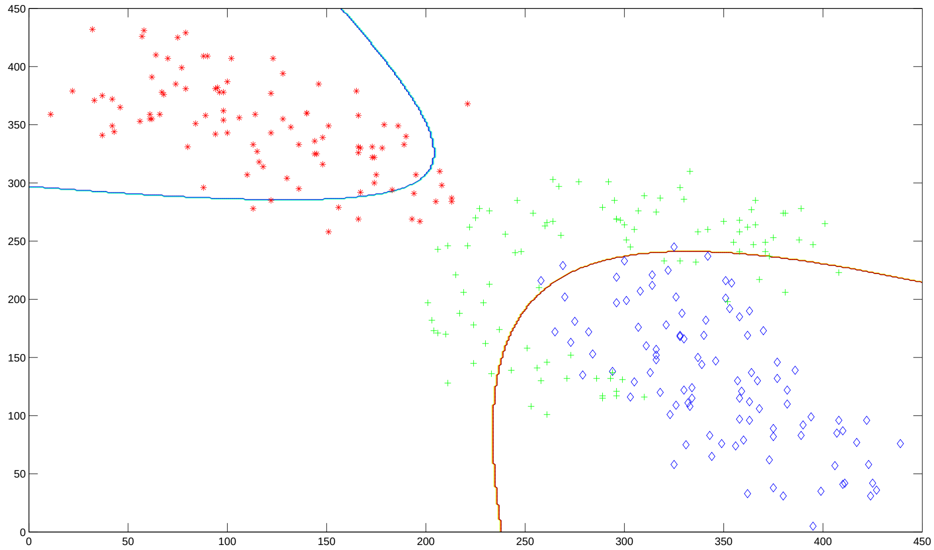

Figure 2.9: 2D Parametric Estimation

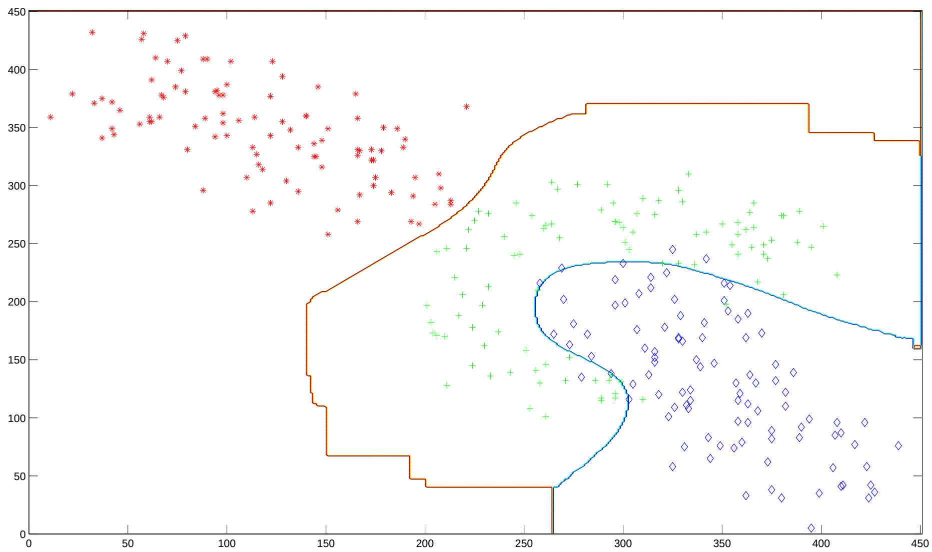

Figure 2.10: 2D Non-Parametric Estimation

model; the parametric approach will just be computationally faster.

Table 2.1: Error rates (%) in classifying the test data

J	min	μ	max	σ
1	22.25	28.1375	35.25	3.3936
2	5.5	11.6375	21	4.3728
3	2.5	7.4625	18.25	4.5337
4	0.25	6.725	38.75	10.1472
5	0	10.1625	37	12.1421

2.3 Sequential Discriminants

Implementation

The sequential discriminant implementation is inplemented in the `SequentialEstimation` class (see Section A.5). The class comprises of 3 main functions to generate the discriminants along with 4 helper functions that classify points based on the discriminants, generate a confusion matrix for the classifier, and plot the discriminants and the class boundaries.

Discriminant calculations in `SequentialEstimation` make use of the `ParametricClass` and `NonParametricClass` classes from Lab 1. Each linear discriminant is represented by a MED boundary between the two reference points. This is used to calculate the confusion matricies to identify acceptable classifiers as well as in plotting and classifying points after the overall sequential classifier is formed.

Results

When testing the classifier against the training data, the probability of error will be zero. As opposed to parametric classifiers where the boundaries are generated from the statistics of the sample, sequential discriminants form their boundaries by ensuring that they capture each data point from the training set inside the classifier. Thus, the training data will fall entirely into the correct classifiers and the probability of error is zero.

This result changes if the number of discriminants allowed is limited. Figure 2.11 shows how error rates generally decrease in inverse proportion to the number of

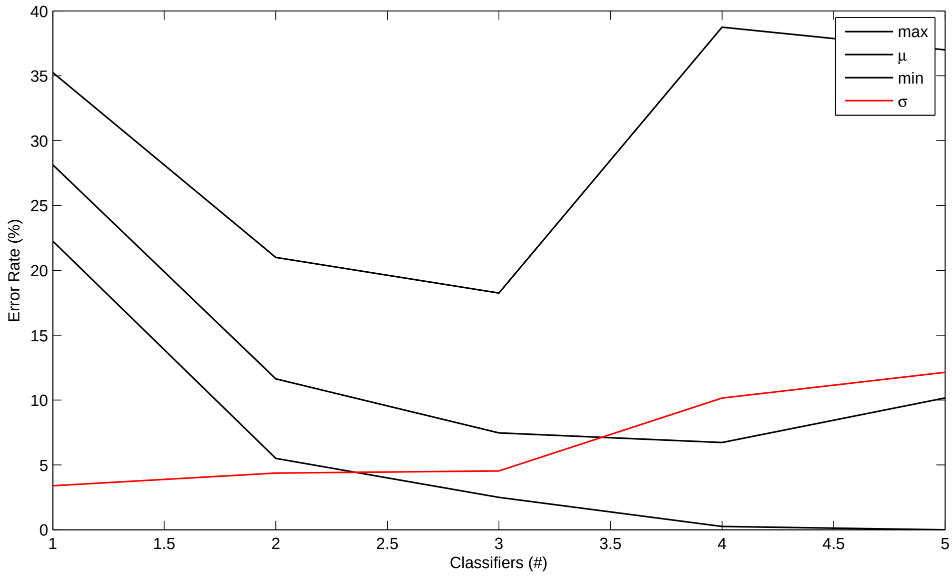

Figure 2.11: Plot of error rates for different numbers of discriminants.

classifiers. It also shows, however, that while the general trend is decreasing error, it is prone to random spikes due to the nature of the classifier selection.

Limiting the number of point pairs to test would also shift the results somewhat. The shift would depend on how the new classifiers were selected. One possible method would be to track how many points from each class fall on the correct side of a potential classifier and select the one with the highest number or proportion of correct points in a class after a specified number of iterations. This method would likely smooth class shapes somewhat as some outlier points may be neglected in a given sectioning.

Another strategy may be to simply stop after a number of iterations if a suitable classifier has not been found. This could produce a rather random set of classifiers, potentially cutting off large sections of a class in the process.

2.4 Conclusions

In the 1D and 2D cases, the nonparametric approach performed much better, in general, than the parametric approach.

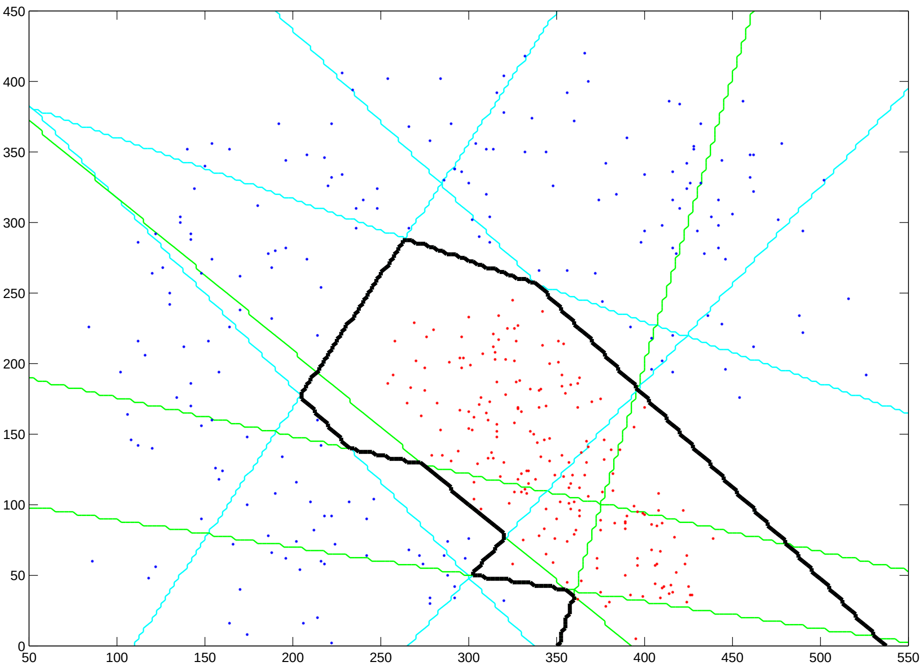

Figure 2.12: Plots the learning data, linear discriminants (cyan and green) and the class boundary (black) using sequential discriminants.

In the 1D case, the parametric approach did produce an incredibly good estimation when the assumed model closely matched the sample distribution. However, when the assumed model was incorrect and not well correlated to the sample data, the resulting estimation was poor.

In the 2D case, similar results were observed. The parametric approach worked well for clusters that were close to guassian in shape, but performance dropped significantly for clusters of unusual shape (ie crescent-shaped). The non-parametric approach is much more flexible because it does not assume a standard PDF but rather estimates it directly from the data. This makes it a much more powerful and robust estimation method in general, and hence it is preferred in most cases.

The sequential discriminant approach attempts to classify the data by sequentially combining linear discriminants that get some part of the class exactly right. The performance of this approach was very good, provided that the classes are separable. Under this condition, it is possible to get the classification completely correct with very little overhead.

Appendix A

Code

A.1 lab2p1.m

```
clear all
close all
load lab2_1

mua = 5;
sigmaa=1;
ParaA=[mua, sigmaa];
np_sigma1=0.1;
np_sigma2=0.4;

A=OneD(a,ParaA);
A.plotGauss()
A.plotTrue()
A.plotExp()
A.plotTrue()
A.plotUni()
A.plotTrue()
A.plotNPE(np_sigma1)
A.plotTrue()
A.plotNPE(np_sigma2)
A.plotTrue()

clear
load lab2_1

lamdab=1;
B=ParametricEstimation(b,lamdab)
B.plotGauss()
B.plotTrue()
B.plotExp()
B.plotTrue()
B.plotUni()
B.plotTrue()
```

A.2 OneD.m

```
classdef OneD

    properties
        Mu
        Sigma
        Lamda
        a
        b
        TrueMu
        TrueSigma
        TrueLamda
        sample
    end

    methods

        function PE = OneD(sample,para)
            PE.sample = sample;
            if(length(para)==2)
                PE.TrueMu=para*[1,0]';
                PE.TrueSigma=para*[0,1]';
                PE.TrueLamda=0;
            else
                PE.TrueLamda=para;
            end
            R=size(sample);
            N=R*[0;1];
            PE.Mu = (sample*ones(N,1))/N;
            PE.Sigma = sqrt((sample-(PE.Mu*ones(1,N)))*(sample-(PE.Mu*ones(1,N)))'/N);
            PE.Lamda = N/(sample*ones(N,1));
            PE.a = min(sample);
            PE.b = max(sample);
        end

        function plotGauss(PE)
            figure;
            for x=PE.Mu-5*PE.Sigma:0.01:PE.Mu+5*PE.Sigma
                p=exp(-(x-PE.Mu)^2/(2*(PE.Sigma)^2))/sqrt(2*pi*PE.Sigma);
                plot(x,p)
                hold on
            end
        end

        function plotExp(PE)
            figure;
            for x=PE.Mu-5*PE.Sigma:0.01:PE.Mu+5*PE.Sigma
                p=PE.Lamda*exp(-PE.Lamda*x);
                plot(x,p)
                hold on
             end
        end

        function plotUni(PE)
            figure;
            for x=PE.Mu-5*PE.Sigma:0.01:PE.Mu+5*PE.Sigma
                p=1/(PE.b-PE.a);
                plot(x,p)
                hold on
             end
```

```
        end

        function plotTrue(PE)
            if (PE.TrueLamda == 0)
                for x=PE.TrueMu-5*PE.TrueSigma:0.01:PE.TrueMu+5*PE.TrueSigma
                    p=exp(-(x-PE.TrueMu)^2/(2*PE.TrueSigma)^2)/sqrt(2*pi*PE.TrueSigma);
                    plot(x,p,'black')
                    hold on
                end
            else
                for x=0:0.01:5*PE.TrueLamda
                    p=PE.TrueLamda*exp(-PE.TrueLamda*x);
                    plot(x,p,'black')
                    hold on
                end
            end
        end

        function plotNPE(PE, sigma)
            figure;
            x=(0:0.1:10);
            p=zeros(1,101);
            N=length(PE.sample);
            for j=1:100
                for i=1:N
                    xi=PE.sample(i);
                    phi=exp(-((x(j)-xi)^2)/(2*(sigma)^2))/(sqrt(2*pi)*sigma);
                    p(j)=p(j)+phi;
                end
                p(j)=p(j)/N;
            end
            plot(x,p,'r')
            hold on
        end

    end
end
```

A.3 lab2p2.m

```
%lab2p2
close all
clear all
load lab2_2


%create a 2-D Gaussian matrix for non-parametric case
increment=1;
max=60;
min=-60;
x=[min:increment:max];
n=length(x);
y=zeros(1,n);
for i=1:n
    y(i)=exp(-x(i)^2/2/400)/sqrt(2*pi)/20;
end
matrix=y'*y;
```

```
%specify range of matrix
res =[1 ,0 ,0 ,450 ,450];

a=TwoD( a1 ) ;
b=TwoD( b1 ) ;
c=TwoD( c1 ) ;

%%nonparametric method( parzen )

[ ap , ax , ay]=a . parzen ( res , matrix ) ;
[ bp , bx , by]=b . parzen ( res , matrix ) ;
[ cp , cx , cy]=c . parzen ( res , matrix ) ;

%Apply ML for classification

ML=ML( ap , bp , cp ) ;

%plot contour
figure ;
contour ( cx , cy ,ML)
hold on
%plot clusters
a . plotCluster ( 'bd ' ) ;
b . plotCluster ( 'g+' ) ;
c . plotCluster ( ' r * ' ) ;

%parametric method
[ ap , ax , ay]=a . parametric ( res ) ;
[ bp , bx , by]=b . parametric ( res ) ;
[ cp , cx , cy]=c . parametric ( res ) ;
ML=ML( ap , bp , cp ) ;

figure ;
contour ( cx , cy ,ML) ;
hold on
a . plotCluster ( 'bd ' ) ;
b . plotCluster ( 'g+' ) ;
c . plotCluster ( ' r * ' ) ;
```

A.4 TwoD.m

```
classdef TwoD

    properties
        data
        mean_estm
        cov_estm
        size
    end

    methods

        function PE= TwoD( sample )
            PE. data=sample ;

```

```
            size=length(sample');

            PE.mean_estm = mean(sample);

            temp=ones(size,2);
            temp=temp*[PE.mean_estm(1,1),0 ; 0,PE.mean_estm(1,2)];

            PE.cov_estm = (PE.data-temp)'*(PE.data-temp)/(size-1);


        end

        function plot=plotCluster(PE,colour)
            scatter([1,0]*PE.data',[0,1]*PE.data',colour);
            hold on
            plot=0;
        end

        function [p,x,y]=parametric(PE,res)
            x = [res(1,2):res(1,1):res(1,4)];
            y =[res(1,3):res(1,1):res(1,5)];
            p = zeros(length(x),length(y));
            for i=1:length(x)
                for j=1: length(y)
                    temp=[x(1,i),y(1,j)]-PE.mean_estm;
                    p(i,j)=exp(-temp*(PE.cov_estm)^(-1)*(temp')/2)/(2*pi)/sqrt(det(PE.cov_estm));
                end
            end
        end


        function [p,x,y] = parzen(PE, res, win )

            if (size(PE.data,2)>size(PE.data,1))
                PE.data = PE.data';
            end
            if (size(PE.data,2)==2)
                PE.data = [PE.data ones(size(PE.data))];
            end

            numpts = sum(PE.data(:,3));

            dl = min(PE.data(:,1:2));
            dh = max(PE.data(:,1:2));
            if length(res)>1
                dl = [res(2) res(3)];
                dh = [res(4) res(5)];
                res = res(1);
            end

            if (nargin == 2)
                win = 10;
            end
            if (max(dh-dl)/res>1000)
                error('Excessive PE.data range relative to resolution.');
            end

            if length(win)==1
                win = ones(1,win);
            end
            if min(size(win))==1
```

```
                win = win(:) * win(:)';
            end
            win = win / (res*res*sum(sum(win)));

            p = zeros(2+(dh(2)-dl(2))/res,2+(dh(1)-dl(1))/res);
            fdl1 = find(PE.data(:,1)>dl(1));
            fdh1 = find(PE.data(fdl1,1)<dh(1));
            fdl2 = find(PE.data(fdl1(fdh1),2)>dl(2));
            fdh2 = find(PE.data(fdl1(fdh1(fdl2)),2)<dh(2));

            for i=fdl1(fdh1(fdl2(fdh2)))'
                j1 = round(1+(PE.data(i,1)-dl(1))/res);
                j2 = round(1+(PE.data(i,2)-dl(2))/res);
                p(j2,j1) = p(j2,j1) + PE.data(i,3);
            end

            p = conv2(p,win,'same')/numpts;
            x = (dl(1):res:(dh(1)+res)); x = x(1:size(p,2));
            y = [dl(2):res:(dh(2)+res)]; y = y(1:size(p,1));

        end
    end
end
```

A.5 SequentialEstimation.m

```
classdef SequentialEstimation < handle
    %SEQUENTIALESTIMATION Summary of this class goes here
    %   Detailed explanation goes here

    properties
        classes = {};
        class_pts = {};
        discriminants = {};
    end

    methods
        function SE = SequentialEstimation(classes)
            SE.classes = classes;
            SE.class_pts = classes;
            SE.discriminants = {};
        end

        function [conf, points] = Confusion(SE)
            % Decede on the random point pair
            pc_1 = ParametricClass(SE.class_pts{1}(randi(size(SE.class_pts{1},1),1),:)',0,0);
            pc_2 = ParametricClass(SE.class_pts{2}(randi(size(SE.class_pts{2},1),1),:)', 0, 0);
            points = {pc_1 pc_2};

            tc_1 = NonParametricClass(SE.class_pts{1});
            tc_2 = NonParametricClass(SE.class_pts{2});
            conf = ParametricClass.ConfusionMatrixMED(points, {tc_1 tc_2});
        end

        function [correct_class_no, points, remaining] = GenerateDiscriminant(SE)
            % Find a good discriminant. Returns a set of ParametricClasses
            % that comprise the discriminant when the MED is used.
```

```
            [c, points] = SE.Confusion();
            while c(1,2) > 0 && c(2,1) > 0
                [c, points] = SE.Confusion();
            end

            if c(1,2) == 0
                %'a' is all within the correct classifier
                %remove 'b's that are classed as 'b'
                correct_class_no = 1;
            else
                %'b' is all within the correct classifier
                %remove 'a's that are classed as 'a'
                correct_class_no = 2;
            end

            % Remove properly classified points from the other class
            r = 0;
            for p=1:size(SE.class_pts{correct_class_no},1)
                class = ParametricClass.ClassifyMED(SE.class_pts{correct_class_no}(p-r, :)', points);
                if class == correct_class_no
                   SE.class_pts{correct_class_no}(p-r,:) = [];
                   r = r+1;
                end
            end

            remaining = SE.class_pts{correct_class_no};
%             Plot Stuff
%             x_range =  50:1:550;
%             y_range = 0:1:450;
%            m = ParametricClass.BoundMatrixMED(points, x_range, y_range);
%             contour(x_range, y_range, m', [1 1], 'LineWidth', 1)
%             hold on;
%             scatter(incomplete_class(:,1), incomplete_class(:,2), 5, strcat('green'))
%             hold on
        end

        function FindDiscriminants(SE, limit)
            SE.discriminants = {};
            SE.class_pts = SE.classes;
            if nargin <= 1
               limit = Inf;
            end
            i = 0;
            while (size(SE.class_pts{1},1) > 0) && (size(SE.class_pts{2},1) > 0) && (i < limit)
                [class_no, pts, remaining] = SE.GenerateDiscriminant();
                SE.discriminants = [SE.discriminants {{class_no, pts}}];
                SE.class_pts{class_no} = remaining;
                i = i+1;
            end
        end

        function class = Classify(SE,point)
            class = 0;
            for i = 1:size(SE.discriminants, 2)
                med_class = ParametricClass.ClassifyMED(point, SE.discriminants{i}{2});
                if med_class == SE.discriminants{i}{1}
                    class = med_class;
                    break
                elseif SE.discriminants{i}{1} == 1
                    class = 2;
```

```
                else
                    class = 1;
                end
            end
        end

        function conf = ConfusionMatrix(SE, limit)
            if nargin <= 1
               limit = Inf;
            end

            SE.FindDiscriminants(limit)

            test_data = {NonParametricClass(SE.classes{1}) NonParametricClass(SE.classes{2})};
            conf = zeros(length(SE.classes));
            %populate test classes and confusion matrix
            for i=1:length(SE.classes)
                for j=1:size(test_data{i}.Cluster,1)
                    c = SE.Classify(test_data{i}.Cluster(j, :)');
                    conf(c,i) = conf(c,i) + 1;
                end
            end
        end

        function PlotDiscriminants(SE)
            x_range =  50:5:550;
            y_range = 0:5:450;

            for i = 1:size(SE.discriminants,2)
                m = ParametricClass.BoundMatrixMED(SE.discriminants{i}{2}, x_range, y_range);

                if SE.discriminants{i}{1} == 1
                    col = 'green';
                else
                    col = 'cyan';
                end

                contour(x_range, y_range, m', [1 1], 'LineWidth', 1, 'Edgecolor', col)
                hold on;
            end

%               scatter(SE.class_pts{class_no}(:,1), SE.class_pts{class_no}(:,2), 5, strcat('green'))
%               hold on
            % Plots
            scatter(SE.classes{1}(:,1), SE.classes{1}(:,2), 5, strcat('*', 'red'))
            hold on
            scatter(SE.classes{2}(:,1), SE.classes{2}(:,2), 5, strcat('*', 'blue'))
            hold on
        end

        function PlotBoundary(SE)
            x_range =  50:2.5:550;
            y_range = 0:2.5:450;
            map = zeros(length(x_range),length(y_range));
            for i = 1:length(x_range)
               for j = 1:length(y_range)
                   map(i,j) = SE.Classify([x_range(i) y_range(j)]');
               end
            end

```

```
            contour(x_range, y_range, map', [1 1], 'LineWidth', 3, 'Edgecolor', 'black')
            hold on;
        end
    end

end
```