Question 1

In the Parametric Method section of the course, we learned how to draw a separation hyperplane between two classes by obtaining w_0, the argmax of the cost function $J(w) = w^T S_B w / w^T S_w w$. The solution was found to be $w_0 = S_w^{-1}(m_1 - m_2)$, where m_1 and m_2 are the sample means of each class, respectively.

Some students raised the question: can one simply use $J(w) = w^T S_B w$ instead (i.e. setting S_w as the identity matrix in the solution w_0? Investigate this question by numerical experimentation.

Background & Method

Fisher's linear discriminant is a classification method that projects high-dimensional data onto a line $y = w^T x$ and performs classification in this one-dimensional space. The goal is to find *w* such that the projection maximizes the distance between the means of the two classes while minimizing the variance within each class. Thus, we can write the problem as

$$argmax_w J(w) = \frac{w^T S_B w}{w^T S_w w} \qquad \rightarrow \qquad w_{opt} \propto S_W^{-1}(m_1 - m_2)$$

where $S_B = (m_1 - m_2)(m_1^T - m_2^T)$ is "between class scatter matrix" and $S_w = \sum_{\vec{y}\ in\ class}(y_i - m_i)(y_i^T - m_i^T)$ is "within class scatter matrix".

In this problem, we want to compare the projection data using $S_w = \Sigma_1 + \Sigma_2$ (Σ_i is the covariance of class *i*) to $S_w = I$.

Experiment & Analysis

We considered two dimensional datasets with two classes. We constructed our experiments based on two different scenarios, where Case 1 contains non-separable data and Case 2 contains separable data.

Case 1:

$$\text{mean } 1 = \begin{bmatrix}0\\0\end{bmatrix}, \text{mean } 2 = \begin{bmatrix}0\\0\end{bmatrix}, \text{standard deviation } 1 = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}, \text{standard deviation } 2 = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$$

150 samples were generated for each class.

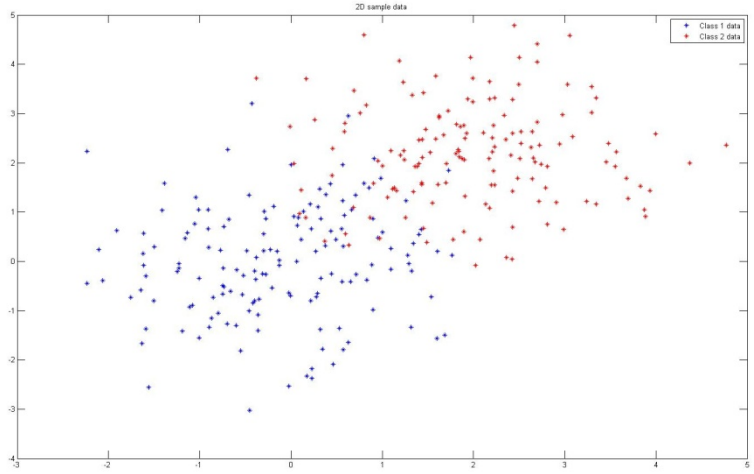

Figure 1a

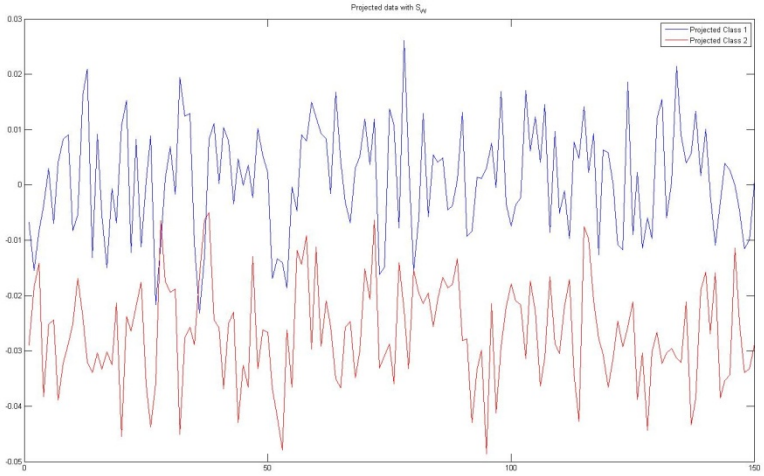

Figure 1b

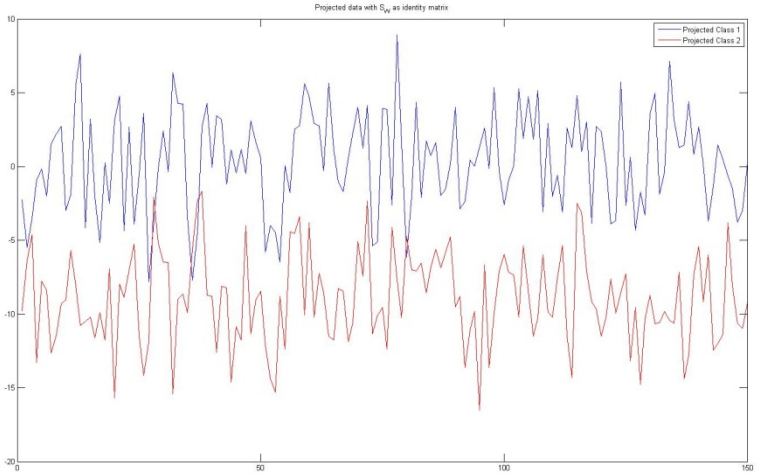

Figure 1c

Figure 1a shows the 2D non-separable data, Figure 1b is the optimal projection given $\boldsymbol{S_w = \Sigma_1 + \Sigma_2}$ and Figure 1c is the results obtained by considering $S_w = I$. We inspected from the results that by setting $\boldsymbol{S_w = \Sigma_1 + \Sigma_2}$, the optimal solution w_{opt} is normalized, whereas when $S_w = I$, the range of w_{opt} became "arbitrary". However in the Fisher's Linear Discriminating analysis, we are only interested in direction of projection, the resulting range of the project is not as important. In the following case we see bigger difference between the two situations both in the between class separation and in the within class separation.

Case 2:

$$\text{mean } 1 = \begin{bmatrix} 0 \\ 3 \end{bmatrix}, \text{mean } 2 = \begin{bmatrix} -5 \\ 0 \end{bmatrix}, \text{standard deviation } 1 = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}, \text{standard deviation } 2 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

150 samples were generated for each class.

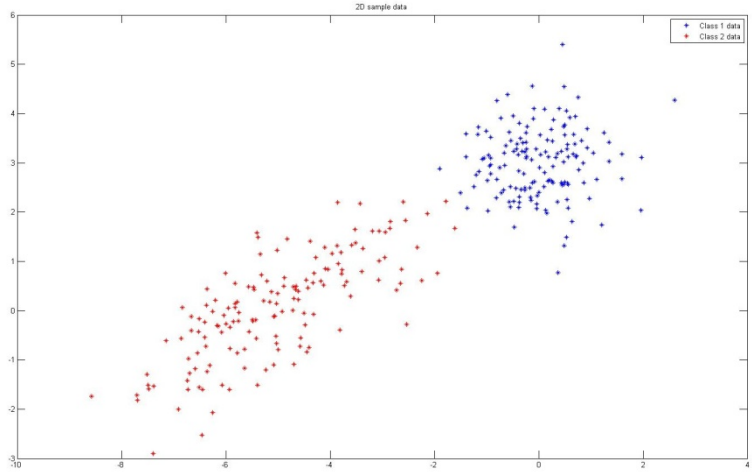

Figure 2a

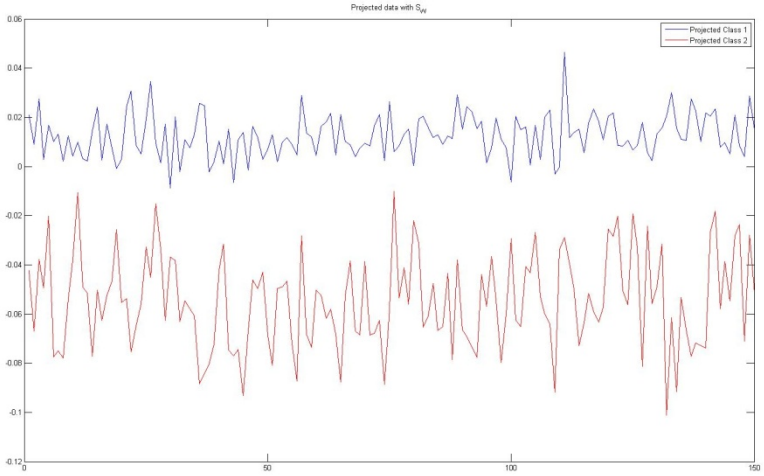

Figure 2b

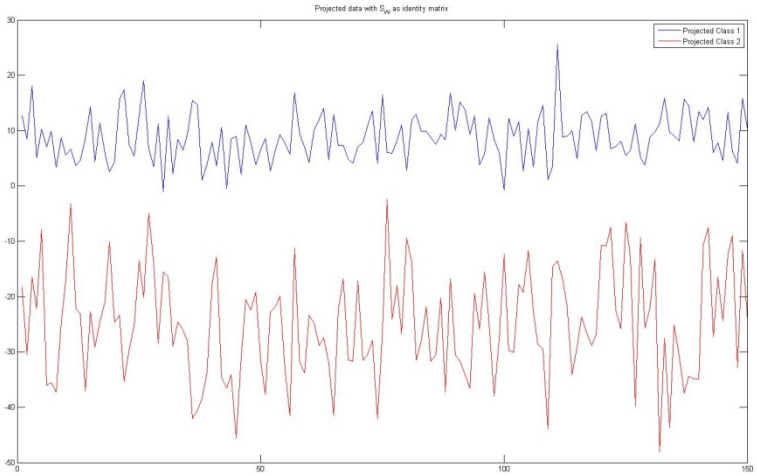

Figure 2c

Figure 2a shows the 2D separable data, Figure 2b is the optimal projection given $S_w = \Sigma_1 + \Sigma_2$ and Figure 2c is the results obtained by considering $S_w = I$. In this second case, when we set $S_w = I$ the algorithm only optimizes the term $w^T S_B w$, and thus the dimension with larger |m_1-m_2| will be chosen (larger between classes). Additionally, in this case the algorithm does not take into consideration the variance within class scatters; this is why we see clearly that in this situation the within projected classes plot is less compacted in comparison with the situation where we set $S_w = \Sigma_1 + \Sigma_2$. Hence the results using the optimal Fisher's solution show that the classes are better separated.

Question 2

Obtain a set of training data. Divide the training data into two sets. Use the first set as training data and the second set as test data.

a) Experiment with designing a classifier using the neural network approach.

b) Experiment with designing a classifier using the support vector machine approach.

c) Compare the two approaches.

a) Neural Network

Background & Method

In this section we implemented an artificial multilayer neural network with one hidden network, in particular, the Back Propagation network, which is widely used and on which many others are based [1]. An illustration is shown in Figure 3.

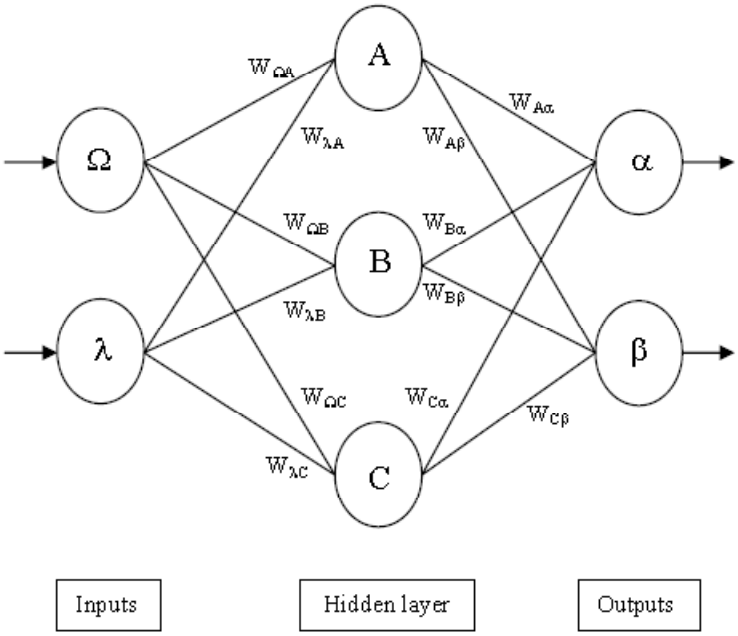

Figure 3 An example of ANN

The Back Propagation algorithm is based on the gradient descent error with the following steps:

- Select a network architecture
- Initialize the weights to small random values
- Compute the corresponding outputs according to the training set

- For each epoch and each training example
    - Input the training example to the network and compute the network outputs
    - For each output unit *k*, we compute its error
        - $\delta_k \leftarrow out_k(1-out_k)(tar_k - out_k)$
    - For each hidden unit *h*, we compute its error
        - $\delta_h \leftarrow out_h(1-out_h)\sum_{k\in outputs} w_{h,k}\delta_k$
    - Update each network weight $w_{i,j}$
        - $w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$
          $\Delta w_{i,j} = \eta\delta_j x_{i,j}$

Through each iteration, the algorithm minimizes the error between the targeted output and the real output. We initialized the weights near to zero for convergence purposes, and set the algorithm to terminate when the change in the criterion function *J(w)* (a function of the error) is smaller than some preset value. We employed the sigmoid activation function as our output function $P(t) = \frac{1}{1+e^{-t}}$.

For this experiment we have taken two different Gaussian classes or patterns, and we have divided them using one part as a training set and the other one as a test set. The goal is to verify that error training decreases as a function of epochs and the error in the test data decreases too, but is higher than the previous one. We have experimented with several network configurations (different number of nodes).

Experiment & Analysis

Case 1:

Mean1=-1. Mean2=1. Standard deviation1= $\sqrt{2}$ .Standard deviation2=$\sqrt{2}$. 200 training samples and 200 testing samples.

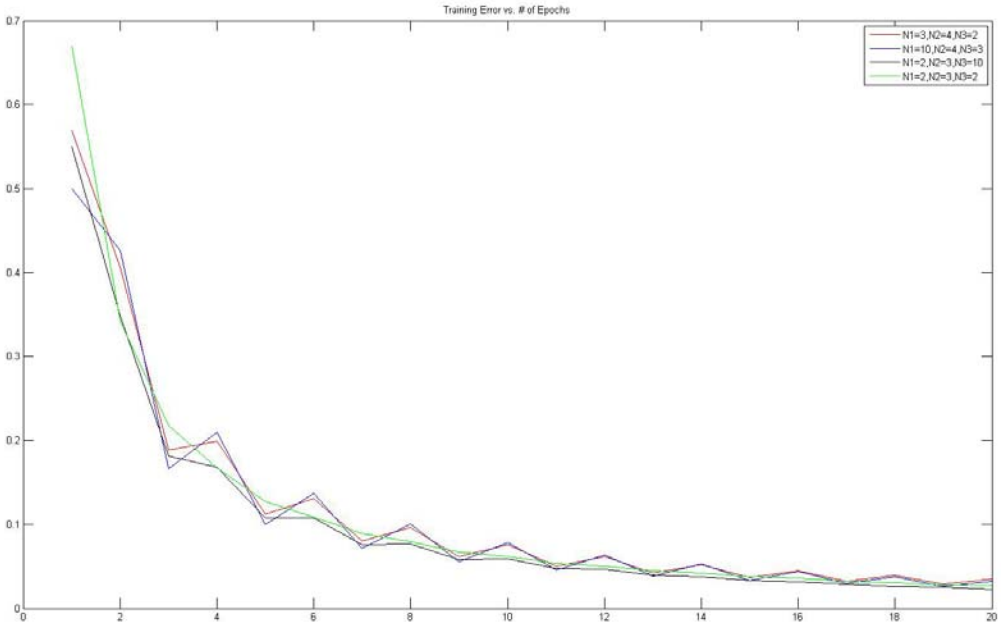

Figure 4 Training error as a function of epochs

Table 1 Sample training error at different epochs

Number of nodes	2 epochs	8 epochs	16 epochs	20 epochs
No. 1	0.3481	0.0817	0.0368	0.0278
No. 2	0.2500	0.0625	0.0313	0.0250
No. 3	0.5000	0.1250	0.0625	0.0500
No. 4	0.3037	0.0714	0.0326	0.0249

Case 2:

Mean1=-3. Mean2=3. Standard deviation1= $\sqrt{3}$ .Standard deviation2= $\sqrt{3}$. 200 training samples and 200 testing samples.

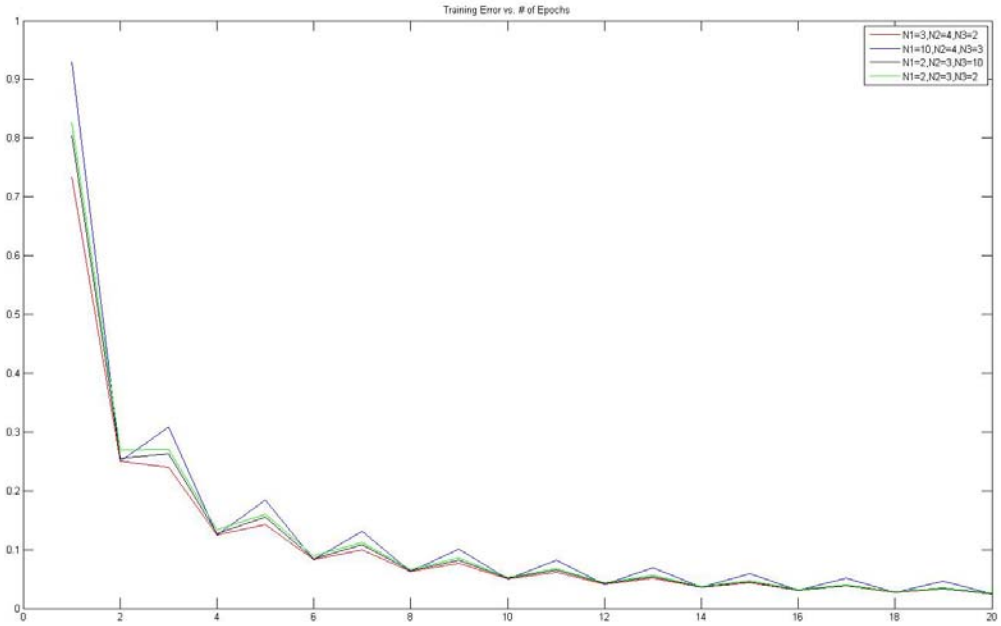

Figure 5 Training error as a function of epochs

Table 2 Sample training error at different epochs

Number of nodes	2 epochs	8 epochs	16 epochs	20 epochs
No. 1	0.2506	0.0626	0.0313	0.0251
No. 2	0.2500	0.0625	0.0313	0.0250
No. 3	0.2556	0.0635	0.0316	0.0252
No. 4	0.2689	0.0663	0.0326	0.0258

It is noted that the correct way to train the network is to apply the training samples of the first class first and change the weights in the network ONCE. Next to apply the training samples of the second class. Once we have all the classes trained once, we return to the first one again and repeat the process until the stop criterion is achieved. Figure 4 and 5 showed plots f training error as functions of epochs, while Table 1 and 2 gave numeric values at sample epochs. The different cases represent different network configurations. We define N1= number of nodes in the first layer, N2 = number of nodes in the hidden layer, N3 = number of nodes in the last layer. As stated in the experiments, in all the simulations we have taken half of samples as training samples and the other half for testing purposes. So we see that for small number of epochs the error training is small when use more nodes, regardless at which layer they are. This is due to the larger number of weights (order of freedom) that it used.

b) Support Vector Machine

Background & Method

Support Vector Machine (SVM) is a supervised learning method commonly used in pattern classification. Viewing the input data as two sets of vectors in an *n*-dimensional space, an SVM will construct a separating hyperplane in that space, one which maximizes the "margin" between the two data sets. To calculate the margin, we construct two parallel hyperplanes, one on each side of the separating one, which are "pushed up against" the two data sets. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the neighboring data points of both classes.

Here, we aimed to learn the mapping: $X \mapsto Y$, where $x \in X$ is a feature (data point) and $y \in Y$ is a class label. The goal is to find a function which minimizes an objective such as: Training Error + Complexity Term. For this experiment we chose the following formulation:

$$\min_{\alpha} D(\alpha) = \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j\Phi(x_i)\Phi(x_j) - \sum_i y_i\alpha_i \qquad \text{where} \qquad \Phi(x_i)\Phi(x_j) = y_i y_j (x_i x_j)$$

Subject to these constraints: $0 \le \alpha_i \le C \quad \forall k$ and $\sum_i \alpha_i y_i = 0$

The cost function is minimized via MATLAB function **quadprog**, which solves quadratic programming problems.

We define: $w = \sum_i \alpha_i y_i x_i$, and $b = y_I(1-\varepsilon_I) - x_I w_I$ where $I = a\max_i\{\alpha_i\}$.

Note that all data points having α_i >0 will be the support vectors. Then the classification rule is defined as:

$$f(x, w, b) = sign(w \cdot x - b)$$

Experiment & Analysis

Similar to previous experiment, we generated a pseudo-random sample points, which generate normally distributed random numbers N(μ,σ) and separated into two different labeled classes. As in the neural network classifier we show the performance of this classifier by showing its error training and error test. In this case we have carried out two experiments: one set with data points more compacted and the other with a more “relaxed” location, and we change different simulation parameter to examine the behavior of our classifier.

Case 1: Mean1=1. Mean2=-1. Standard deviation1= $\sqrt{2}$ .Standard deviation2=$\sqrt{2}$. Using 500 samples to train the classifier, and 500 samples to test it.

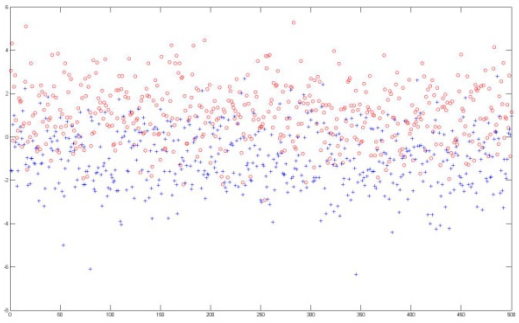

Figure 6a Sample data

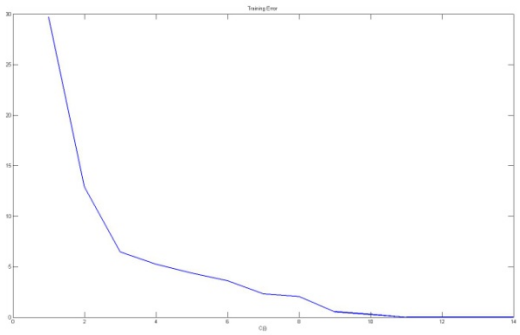

Figure 6b Classification error

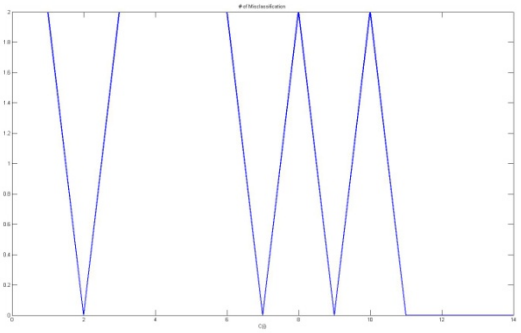

Figure 6c # of misclassification

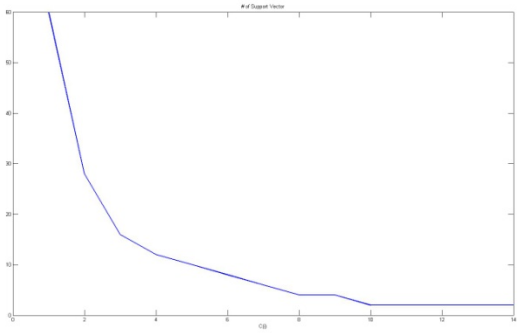

Figure 6d # of support vector

Case 2: Mean1=3. Mean2=-3. Standard deviation1= $\sqrt{3}$ .Standard deviation2=$\sqrt{3}$. Using 500 samples of each class to train the classifier, and 500 samples to test it.

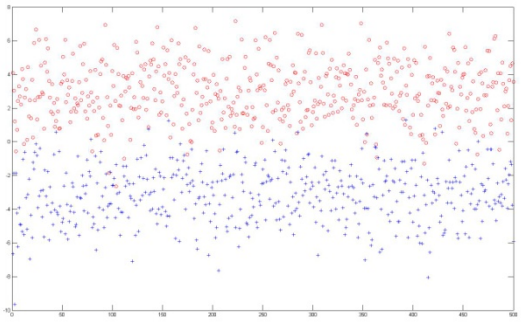

Figure 7a Sample data

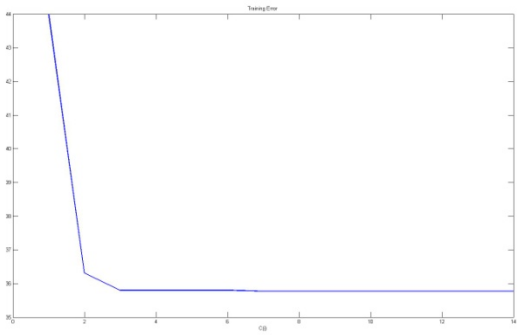

Figure 7b Classification error

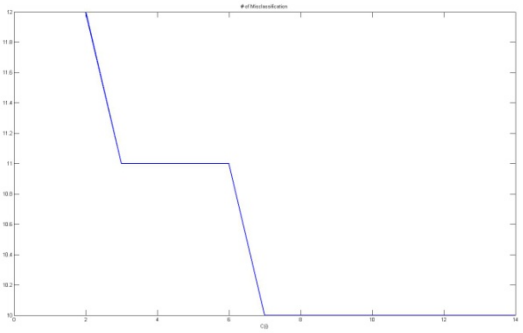

Figure 7c # of misclassification

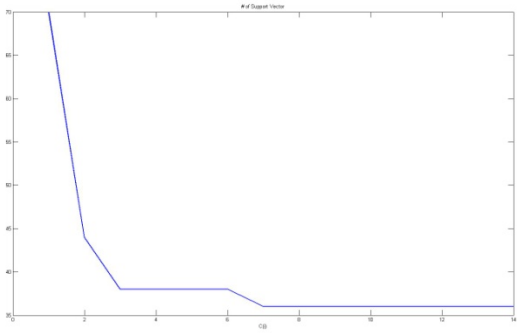

Figure 7d # of support vector

From the results in both cases we can conclude that, for our data points, the parameter C may have the optimal value as it gets larger, so when working with Gaussian data we may not need to know *a priori* the problem under consideration since in both cases the trend as far as C is concerned is the same. Surprisingly, the results in terms of number of misclassification for small C values are quite unexpected. In the first case, where the data is more compacted, we have less number of misclassified points in the second case, and intuitively we would have expected different results (other way around). Moreover, when the number of training samples was examined we observed that the resulting optimization problems are dependent upon the number of training examples. As such, when this data is large other methods for speeding up the algorithm should be addressed. The results in terms of the error are shown in the following section along with the neural network classifier error.

c) Comparison between Neural Network classifier and Support Vector Machine classifier

In this section we briefly discuss the performance of both the neural network classifier and the SVM classifier in terms of the test error performance. Note though, that there is no perfect comparison between these methods, here we have focused on the error performance for different situations.

Table 3. Performance analysis of Neural Network and Support Vector Machine

P(e)	Neural Network			SVM		
Case 1	0.6060 2 epochs	0.0855 8 epochs	0.0319 To inf.	29.7119 C=0.1	0.276 C=10	0 C= inf.
Case 2	0.5763 2 epochs	0.0625 8 epochs	0.0253 To inf.	43.9958 C=0.1	35.7721 C=10	35.7721 C=inf.

For our set-up we obtained the results shown above, where the SVM classifier performed better in one case since it achieves smaller error values for some C's than the convergence error (best case) of the neural network classifier. However, in the other case, the NN performs much better. In both cases, NN is more constant in terms of error performance.

However, in terms of computational time, the neural network classifier performed much faster, and this is something to take into account for large data sets.

Question 3

Using the same data as for question 2 (perhaps projected to one or two dimensions for better visualization),

a) Design a classifier using the Parzen window technique.

b) Design a classifier using the K-nearest neighbor technique

c) Design a classifier using the nearest neighbor technique.

d) Compare the three approaches.

a) Parzen Window

Background & Method

Parzen window approach consists of estimating densities by temporarily assuming that the region R_n is a *d*-dimensional hypercube. If h_n is the length of an edge of that hypercube, then its volume is given by $V_n = h_n^d$.

In the simplest case, if the window function is a unit function. Then, the estimate of the density at x is given as:

$$p_n(x) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{V_n}\phi\left(\frac{x - x_i}{h_n}\right)$$

And by unit function we mean:

$$\phi(v) = \begin{cases} 1, |\, v_j \,| \le 1/2;\ j = 1,...,d \\ 0, otherwise \end{cases}$$

$p_n(x)$ expression suggests a general approach to estimating density functions. For Parzen window method, the choice of the hypercube volume has an important effect on $p_n(x)$. If V_n is too large, the estimate will suffer from too little resolution; if V_n is too small, the estimate will suffer from too much statistical variability. With a limited number of samples, the best we can do is to seek some acceptable compromise. However, with an unlimited number of samples, it is possible to let V_n slowly approach zero as *n* increases and have $p_n(x)$ converge to the unknown density $p(x)$.

Due to the Gaussian nature of our test and training data, the Parzen window is designed using the following function:

$$\phi(v) = \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-v^2}{2}\right)$$

Using $h_n = \frac{h_1}{\sqrt{n}}$, where h_1 is a design parameter we can obtain the estimate of the density expressed as follows:

$$p_n(x) \propto \frac{1}{n} \sum_{i=1}^{n} \frac{1}{\left(h_1 / \sqrt{n}\right)^n} \exp\left(-(x - x_i)^T (x - x_i) / (2h_n^2)\right)$$

Experiment & Analysis

Similar to Question 2, two classes of data were generated, half of which was used as training data and the other half is used as test data. These data were generated using Gaussian distributed random number generator in MATLAB. Note that for an easy visualization of the results we opted for the 1-dimensional case. To see the performance of our classifier we have carried out several simulations using different lengths of data samples and different hypercube sizes. We used the following statistical parameters to generate our sample points both training and test. Mean1=1. Mean2=-1. Standard deviation1= $\sqrt{2}$ .Standard deviation2= $\sqrt{2}$.

We first estimate the density function given the training points. Three sample sizes and hypercube sizes were used. N = sample size (e.g. 1000, 10000, 100000) and h1 = hypercube size (e.g. 0.1, 1, 5).

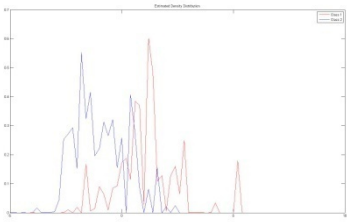

Figure 8a N=1000, h1=0.1

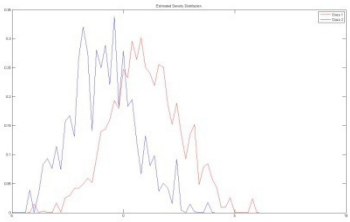

Figure 8b N=1000, h1=1

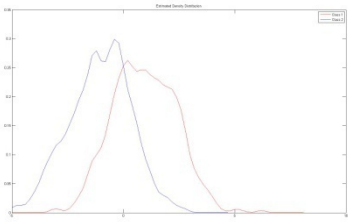

Figure 8c N=1000, h1=5

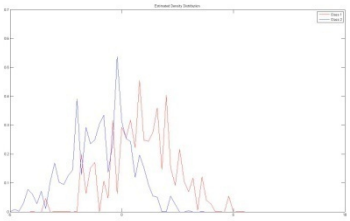

Figure 8d N=10000, h1=0.1

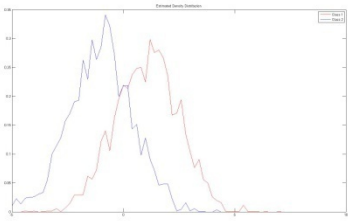

Figure 8e N=10000, h1=1

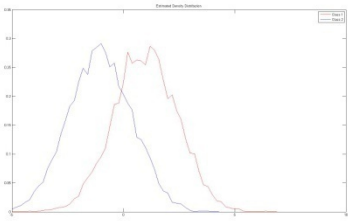

Figure 8f N=10000, h1=5

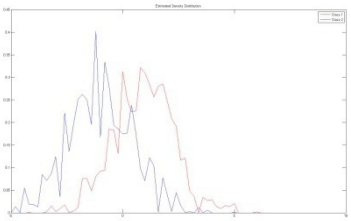

Figure 8g N=100000, h1=0.1

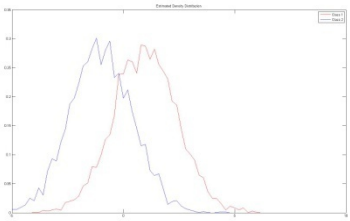

Figure 8h N=100000, h1=1

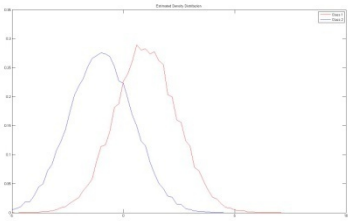

Figure 8i N=100000, h1=5

It can be seen from the above figures that the results depend on both N and h1. As N increases the estimate matches better with the true density function, and as h1 decreases the estimated density function becomes thinner. Therefore, once we have tested several estimates of density functions using the training data we can classify the data test by Parzen window method. To do so, we first need to estimate the *a posteriori* probability of the data test given the training set, i.e., $P(w_i \mid x)$. Thus, using the total joint probability theorem we can denote:

$$P_n(w_i \mid x) = \frac{p_n(x, w_i)}{\sum_{j=1}^{c} p_n(x, w_j)}$$

Where $p_n(x, w_i)$ is the estimate for the joint probability $P(w_i, x)$, that can be thought as if we place a cell of volume V_n around the training set and capture k samples, k_i of which turn out to be labeled w_i. Roughly speaking the estimate of the posteriori probability that w_i is the state of nature is merely the fraction of the samples within the cell that are labeled w_i. Therefore, to get a minimum error expression we select the category most frequently represented within the cell. Note this approach is highly dependent on the V_n. In Table 4, we compared the error probability using the Parzen Window method for different sample size and hypercube size.

Table 4 Comparison of probability of error using Parzen Window

P(e)	N = 1000	N = 10000	N = 100000
h1 = 0.1	0.2800	0.2812	0.2539
h1 = 1	0.2350	0.2473	0.2386
h1 = 5	0.2690	0.2426	0.2362

b) K-nearest Neighbor

Background & Method

The idea of the K-Nearest Neighbor technique consist of estimating p(x) from a set of training samples where a cell is centered around x and it grows until captures k_n samples, where k_n is function of n (*g*enerally $k_n = \sqrt{n\_training}$ is good enough, n_training=n/2). Obviously the key point is to set k_n to go to infinity as n goes to infinity as well, assuring that simple k_n / n is a good estimate of the probability that any given point falls in the cell of volume V_n.

Experiment & Analysis

In order to keep consistency in data analysis, we considered 1-dimensional case and particularly the same sample data as we used in Parzen window method. Therefore, the dataset had the following parameters: Mean1=1, Mean2=-1, Standard deviation1=$\sqrt{2}$, Standard deviation2=$\sqrt{2}$, and $k_n = \sqrt{\text{n\_training}}$, e.g. n_training = 1000, 10000, 100000, 1000000.

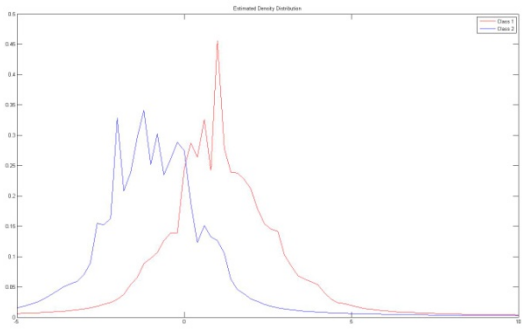

Figure 9a N=1000

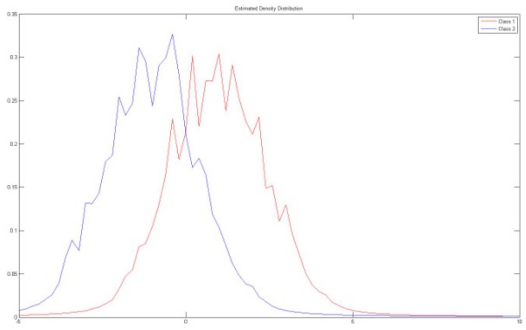

Figure 9b N=10000

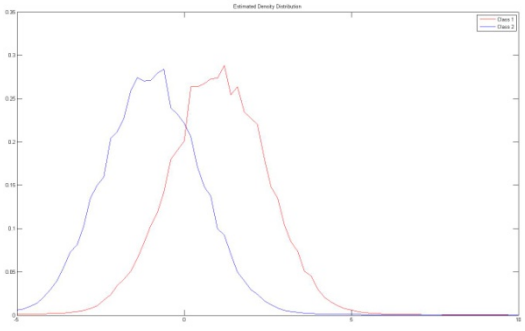

Figure 9c N=100000

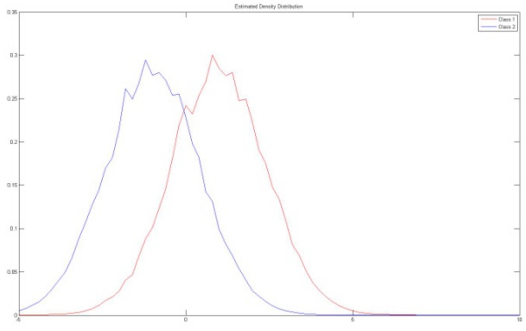

Figure 9d N=1000000

We classify the test data according to the following method:

- We estimate the probability $P(x \mid w_i)$, i=1 and 2, as $p_n(x \mid w_i) = \frac{k_i / n}{V}$
- We compare $P(x \mid w_1)$ and $P(x \mid w_2)$ to choose the larger one as the class. We ignore prior probabilities since we assume them equal.

Table 5 showed the results of the performance of the K-Nearest Neighbor classifier in terms of P(e) as function of the length of our data set.

Table 5. Comparison of probability of error using kNN

P(e)	N = 1000	N = 10000	N = 100000	N = 1000000
	0.2510	0.2400	0.2398	0.2395

c) Nearest Neighbor

Background & Method

This is a particular case of the K-Nearest Neighbor method, where the class is predicted to be the class of the closest training sample, i.e. the algorithm just looks at one nearby neighbor. If the number of samples is not large it makes a good sense to use, instead of the k-nearest neighbor, the single nearest neighbor.

Experiment & Analysis

Again, we employed the following parameter: Mean1=1. Mean2=-1. Standard deviation1= $\sqrt{2}$ .Standard deviation2= $\sqrt{2}$, and $k_n = 1$.

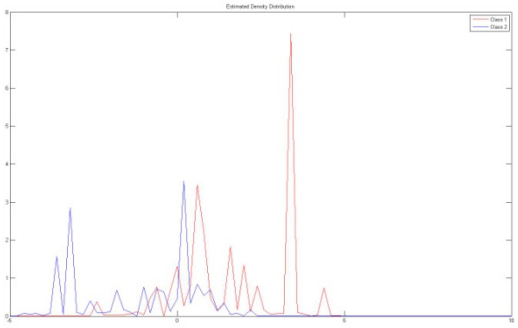

Figure 10a N=1000

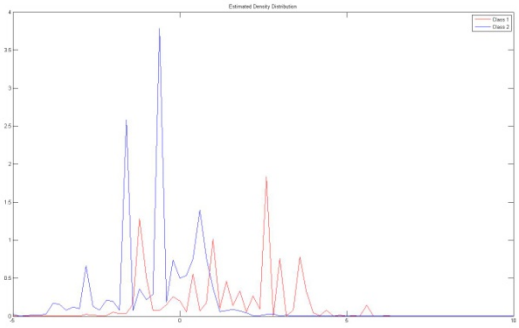

Figure 10b N=10000

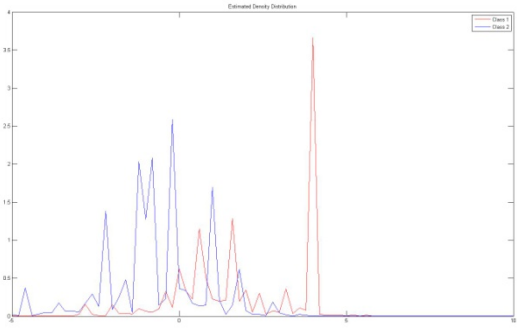

Figure 10c N=100000

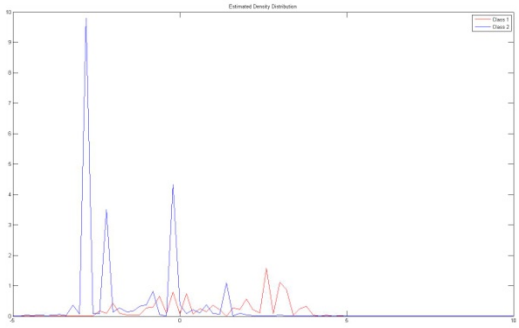

Figure 10d N=1000000

Table 6 showed the results of the performance of the Nearest Neighbor classifier in terms of the probability error as function of the length of our data set.

Table 6. Comparison of probability of error using NN

P(e)	N = 1000	N = 10000	N = 100000	N = 1000000
	0.4050	0.3572	0.3406	0.3397

d) Compare the three approaches.

To analyze the performance the three different methods given the same set of data, we generated 100000 sample points using following parameter: Mean1=1. Mean2=-1. Standard deviation1= $\sqrt{2}$ .Standard deviation2= $\sqrt{2}$, and $k_n = 1$. Figure 10 showed a comparison of the estimated distribution function between the Parzen window (h1=1), K-nearest Neighbor (k=224), and Nearest Neighbor methods.

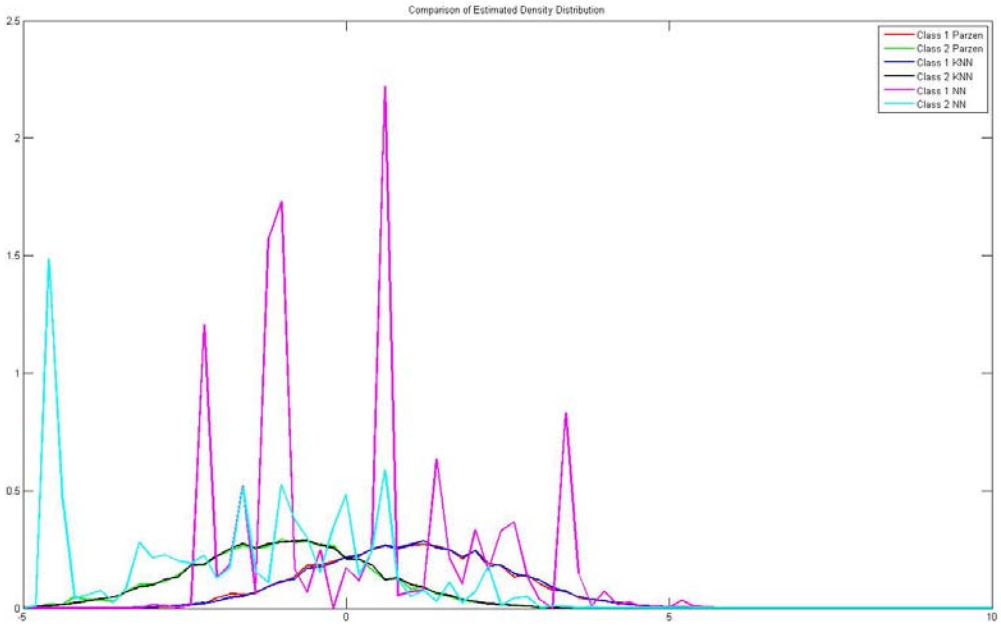

Figure 11 Comparison between Parzon Window, KNN and NN methods

From Figure 10, we noted that the result of the NN is shown in the following figure, since using the same number of data points than the other two methods the estimate has large peaks. The probability also showed the same trend, i.e. P(e)_Parzen = 0.2419, P(e)_KNN = 0.2419, P(e)_NN = 0.4287.

```
% Hw 2 p1 Parametric Method

clear all
close all

% sample points
n1=150;
n2=150;
% % 1-dim
% mean_x1 = 2;
% var_x1 = 4;
% mean_x2 = 0;
% var_x2 = 1;
% x1 = mean_x1 + sqrt(var_x1)*randn(1,n1);
% x2 = mean_x2 + sqrt(var_x2)*randn(1,n2);
% 2-dim
Mean1 = [ 0 3]';
Mean2 = [ -5 0]';
std1 = [0.5 0; 0 0.5];
std2 = [2 1; 1 1];
data_class1 = mvnrnd(Mean1,std1,n1);
data_class2 = mvnrnd(Mean2,std2,n2);
plot(data_class1(:,1),data_class1(:,2),'b*',...
    data_class2(:,1),data_class2(:,2),'r*');
legend('Class 1 data','Class 2 data');
title('2D sample data');
x1=data_class1;
x2=data_class2;
mhu_1=(1/n1)*(sum(x1));
mhu_2=(1/n2)*(sum(x2));

bet_scatter= (mhu_1-mhu_2);

% S_B = eye(f,c);
S_W1 = size(x1,1)*cov(x1);
S_W2 = size(x2,1)*cov(x2);
S_W = S_W1+S_W2;
[f,c]=size(S_W);
S_W_I=eye(f,c);

w_opt=S_W\bet_scatter';
w_opt_I =S_W_I\bet_scatter';

% Projections
y1 = x1*w_opt;
y2 = x2*w_opt;
y1_I = x1*w_opt_I;
y2_I = x2*w_opt_I;

bin = 0.1;
x = -25:bin:25;
xa = 1:length(y1);
xb=1:length(y2);
figure,
plot(xa,y1,'b',xb,y2,'r');
legend('Projected Class 1','Projected Class 2');
title('Projected data with S_W');
```

```
figure,
plot(xa,y1_I,'b',xb,y2_I,'r');
legend('Projected Class 1','Projected Class 2');
title('Projected data with S_W as identity matrix');
```

```
% performs backpropagation algorithm
close all;
clear all;
%rand('state',100);
% the neurons have a sigmoid function activation
% data length
N1 = 2;
N2 = 3;
N3 = 2;
% length training set
% iter = epochs
iter = 20;
iter_test = 20;
Target = zeros(1,N3);

% initialize weights
W_hid_in = rand(1,N1);
W_hid_out = rand(1,N2);
error_epoch = zeros(1,iter);
error_epoch_test = zeros(1,iter_test);
Mean1 = -1;
Mean2 = 1;
std1 = sqrt(2);
std2 = sqrt(2);
data_class1 = Mean1 + std1*randn(1,N1);
data_class2 = Mean2 + std2*randn(1,N1);
for k=1:iter
if (mod(k,2)==0)
    training_data = data_class1;
else
    training_data = data_class2;
    epoch=k,
end
for i=1:N1
    sig_output(i) = training_data(i);
end
% training the neural network step
% outputs
for n=1:N3
    in_last(n)=0;
for j=1:N2
    input_hid(j)=0;
for i=1:N1
    input_hid(j) = input_hid(j)+W_hid_in(i)*sig_output(i);
end

W_old_hidden(:,j) = W_hid_in';
    sig_output_hid(j) = (1)/(1+exp(-input_hid(j)));
    in_last(n) = sig_output_hid(j)*W_hid_out(j)+in_last(n);
end
out(n) = (1)/(1+exp(-in_last(n)));

W_old_output(:,n) = W_hid_out';

end

lear_rate = 0.25;
```

```
% backpropagation step

% calculate errors of output neurons
for i=1:N3
    delta(i) = out(i)*(1-out(i))*(Target(i)-out(i));
end
% Change output layer weights
for i=1:N2
    for j=1:N3
        W_new_output(i,j) = W_old_output(i,j)+lear_rate*delta(j)*sig_output_hid(i);
    end
end
% back-propagate
for i=1:N2
    ssuumm=0;
    for j=1:N3
        ssuumm = delta(j)*W_new_output(i,j)+ssuumm;
    end
    delta_hid(i) = sig_output_hid(i)*(1-sig_output_hid(i))*ssuumm;
end

% change hidden layer weights
for i=1:N1
    for j=1:N2
        W_new_hidden(i,j) = W_old_hidden(i,j)+lear_rate*delta_hid(j)*training_data(i);
    end
end

W_old_output = W_new_output;
W_old_hidden = W_new_hidden;

% forward pass with the new weights
for i=1:N1
    sig_output(i) = training_data(i);
end
% outputs
for n=1:N3
    in_last(n) = 0;
    W_hid_out = W_new_output(:,n)';
for j=1:N2
    input_hid(j) = 0;
    W_hid_in = W_new_hidden(:,j)';

for i=1:N1
    input_hid(j) = input_hid(j)+W_hid_in(i)*sig_output(i);
end
    sig_output_hid(j) = (1)/(1+exp(-input_hid(j)));
    in_last(n) = sig_output_hid(j)*W_hid_out(j)+in_last(n);
end
output(n,k) = (1)/(1+exp(-in_last(n)));
error(k) = abs(Target(n)-output(n,k));
end
error_epoch(k) = (error_epoch(k)+error(k))/k;
end
x=1:iter;
plot(x,error_epoch,'b'); hold on;
```

```
% hold on;
y=zeros(1,iter_test);

%% Testing...
for k=1:iter_test
data_class1 = Mean1 + std1*randn(1,N1);
data_class2 = Mean2 + std2*randn(1,N1);
% Generating the test data
p=randperm(2);
if (p(1)==1)
training_data = data_class1;
else
training_data = data_class1;
end
epoch=k,
for i=1:N1
    sig_output(i) = training_data(i);
end
% outputs
for n=1:N3
    in_last(n) = 0;
for j=1:N2
    input_hid(j) = 0;
for i=1:N1
    input_hid(j) = input_hid(j)+W_hid_in(i)*sig_output(i);
end
    sig_output_hid(j) = (1)/(1+exp(-input_hid(j)));
    in_last(n) = sig_output_hid(j)*W_hid_out(j)+in_last(n);
end
outpu_test(n,k) = (1)/(1+exp(-in_last(n)));
error_test(k) = abs(Target(n)-outpu_test(n,k));
end
error_epoch_test(k) = (error_epoch_test(k)+error_test(k))/k;
y(k)=(y(k)+1)/k
end
x=1:iter_test;
plot(x,error_epoch_test,'r'); hold off;
% W_hid_in
% W_hid_out
```

```
% clear;   % clear variables from memory
% close all;

nsample = 300;

Mean1 = 1;
Mean2 = -1;
std1 = 2;
std2 = 2;
data_class1 = Mean1 + std1*randn(1,nsample/2);
data_class2 = Mean2 + std2*randn(1,nsample/2);
X(1:nsample/2) = data_class1;
X(nsample/2+1:nsample) = data_class2;
X = sort(X);
plot(data_class1,'ro');hold on;
plot(data_class2,'b+');
p = randperm(nsample);
Y(p(1:nsample/2)) = -1;
Y(p(nsample/2+1:nsample)) = 1;

C = [0.1, 1, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 100000];
Margin = [];     % margin; initialized as null
nSV = [];        % number of support vector;
nMis = [];       % number of misclassification;
Err = [];        % training errors;
% X,Y,
for n = 1 : max(size(C)),
    H = zeros(nsample, nsample);
    for i = 1 : nsample,
        for j = 1 : nsample,
            H(i,j) = X(i)*X(j)*Y(i)*Y(j);
        end
    end
    H = H+1e-10*eye(size(H));
    F = -ones(nsample,1);
    A = Y;
    b = zeros(size(Y));


    UB = zeros(nsample,1);
    LB = C(n)*ones(nsample,1);

    % starting point of alpha
    Alpha0 = zeros(nsample, 1);

   % optimizing alpha with quadratic programming


    [Alpha] = quadprog(H, F, [], [], A, b, LB, UB, Alpha0),

%   [Alpha,FVAL] = quadprog(H, F, A, b),
    % tolerance for support vector detection; we will ignore the alphas less than tol
    tol = 0.0001;

    % calculate weight
    w = 0;
    for i = 1 : nsample,
```

```
    w = w + Alpha(i) * Y(i) * X(i);
end

% calculate bias
bias = 0;
b1 = 0;
b2 = 0;
for i = 1 : nsample,
if (Alpha(i) > tol & Alpha(i) < C(n) - tol),
    b1 = b1 + X(i) * w - Y(i);
        b2 = b2 - 1;
    end
end

if b2 ~= 0,
    bias = b1 / b2;
else     % unlikely
    b1 = 0;
    for i = 1 : nsample,
        if Alpha(i) < tol,
        b1 = b1 + X(i) * w - Y(i);
            b2 = b2 - 1;
        end
    end

if b2 ~= 0,
    bias = b1 / b2;
    else     % even unlikelier
    b1 = 0;
        for i = 1 : nsample,
        b1 = b1 + X(i) * w - Y(i);
            b2 = b2 - 1;
        end
    if b2 ~= 0,
        bias = b1 / b2;
        end
    end
end

% margin = 2 / ||w||
Margin = [Margin, 2 / abs(w)];
nSV = [nSV, size(find(Alpha > tol), 1)];


% calculate # of misclassification and training error
m = 0;
e = 0;
for i = 1 : nsample,
predict = w * X(i) + bias;     % Y = w * X + b
    if predict >= 0 & Y(i) < 0,
        m = m + 1;
    end
    if predict < 0 & Y(i) >= 0,
        m = m + 1;
    end
    if Alpha(i) > tol,              % consider support vectors only; why?
    e = e + 1 - predict * Y(i);
```

```
        end
    end
    nMis = [nMis, m],
    Err = [Err, e],
end

Z = zeros(size(C));
for i = 1 : size(C, 2)
     Z(i) = i;
end

figure
plot(Z, Margin);
title('Margin');
xlabel('C(i)');

figure
plot(Z, Err);
title('Training Error');
xlabel('C(i)');

figure
plot(Z, nMis);
title('# of Misclassification');
xlabel('C(i)');

figure
plot(Z, nSV);
title('# of Support Vector');
xlabel('C(i)');
```

```
clear all
close all

n = 100000;
train_data = n/2;
test_data = n/2;

% Data set 1: x1 with distribution N(a,b) (mean=a, var=b)
mean_x1 = 1;
var_x1 = 2;
x1 = mean_x1 + sqrt(var_x1)*randn(1,n);
x1_train = x1(1:train_data);
x1_test = x1(train_data+1:end);

% Data set 2: x2 with distribution N(a,b) (mean=a, var=b)
mean_x2 = -1;
var_x2 = 2;
x2 = mean_x2 + sqrt(var_x2)*randn(1,n);
x2_train = x2(1:train_data);
x2_test = x2(train_data+1:end);

% 1st estimation by Parzen window

d = 1;  % dimention
x = -5:0.2:10;
L_x = length(x);
%setting h1
h1 = 5;
hn = h1/sqrt(train_data);
Vn = hn^d;

Q1 = zeros(1,train_data);
prob1_train = zeros(1,L_x);

Q2 = zeros(1,train_data);
prob2_train = zeros(1,L_x);

for i = 1:L_x
    for j = 1:train_data
        Q1(j) = 1/(sqrt(2*pi))*exp(-(x(i) - x1_train(j))^2/(2*hn^2));
        Q2(j) = 1/(sqrt(2*pi))*exp(-(x(i) - x2_train(j))^2/(2*hn^2));
        prob1_train(i) = prob1_train(i) + 1/train_data*1/Vn*Q1(j);
        prob2_train(i) = prob2_train(i) + 1/train_data*1/Vn*Q2(j);
    end
end



figure
plot(x,prob1_train,'r-', x,prob2_train,'b-');
legend('Class 1','Class 2');
title('Estimated Density Distribution');



% 2nd step classification and errors by Parzen window method
error1 = 0;
error2 = 0;
```

```
for i = 1:test_data
  parzen = find(abs(x-x1_test(i)) <= 0.1);
    if (prob1_train(parzen) < prob2_train(parzen))
        error1 = error1 + 1;
    end
    parzen2 = find(abs(x-x2_test(i)) <= 0.1);
    if(prob2_train(parzen2) < prob1_train(parzen2))
        error2 = error2 + 1;
    end
end

error_total = error1 + error2
error_parzen_prob = error_total/(2*test_data)
```

```
% Nearest Neighbor
clear all
close all

% sample data
n = 1000000;
train_data = n/2;
test_data = n/2;

% Data set 1:
mean_x1 = 1;
var_x1 = 2;
x1 = mean_x1 + sqrt(var_x1)*randn(1,n);
x1_train = x1(1:train_data);
x1_test = x1(train_data+1:end);

% Data set 2:
mean_x2 = -1;
var_x2 = 2;
x2 = mean_x2 + sqrt(var_x2)*randn(1,n);
x2_train = x2(1:train_data);
x2_test = x2(train_data+1:end);



% function of kn (KNN)
kn = ceil(sqrt(train_data));
% function of kn (NN)
%kn = 1;


x = -5:0.2:10;
L_x = length(x);
p1_nn = zeros(1,L_x);
p2_nn = zeros(1,L_x);

for i = 1:L_x
    index_sort1 = sort(abs(x1_train - x(i)));
    V1 = 2 * index_sort1(kn);
    index_sort2 = sort(abs(x2_train - x(i)));
    V2 = 2 * index_sort2(kn);
    if (V1 > 0)
        p1_nn(i) = kn/train_data/V1;
    end
    if(V2 > 0)
        p2_nn(i) = kn/train_data/V2;
    end
    if (p1_nn(i)>10)
        p1_nn(i)=0;
    end
    if (p2_nn(i)>10)
        p2_nn(i)=0;
    end
end

figure
plot(x,p1_nn,'r-',x,p2_nn,'b-');
legend('Class 1','Class 2');
```

```
title('Estimated Density Distribution');


% Classification

error_nn_total = 0;
error1 = 0;
error2 = 0;

for i = 1:test_data
    j1_nn = find(abs(x-x1_test(i)) <=0.1);

    if (p1_nn(j1_nn) < p2_nn(j1_nn))
        error1 = error1 +1;
    end

    j2_nn = find (abs(x-x2_test(i))<=0.1);

    if(p2_nn(j2_nn) < p1_nn(j2_nn))
        error2 = error2 +1;
    end
end
error_nn_total = (error1 + error2)/2/test_data
```

```
% Nearest Neighbor
clear all
close all

% sample data
n = 100000;
train_data = n/2;
test_data = n/2;

% Data set 1:
mean_x1 = 1;
var_x1 = 2;
x1 = mean_x1 + sqrt(var_x1)*randn(1,n);
x1_train = x1(1:train_data);
x1_test = x1(train_data+1:end);

% Data set 2:
mean_x2 = -1;
var_x2 = 2;
x2 = mean_x2 + sqrt(var_x2)*randn(1,n);
x2_train = x2(1:train_data);
x2_test = x2(train_data+1:end);



% function of kn (KNN)
% kn = ceil(sqrt(train_data));
% function of kn (NN)
kn = 1;


x = -5:0.2:10;
L_x = length(x);
p1_nn = zeros(1,L_x);
p2_nn = zeros(1,L_x);

for i = 1:L_x
    index_sort1 = sort(abs(x1_train - x(i)));
    V1 = 2 * index_sort1(kn);
    index_sort2 = sort(abs(x2_train - x(i)));
    V2 = 2 * index_sort2(kn);
    if (V1 > 0)
        p1_nn(i) = kn/train_data/V1;
    end
    if(V2 > 0)
        p2_nn(i) = kn/train_data/V2;
    end
    if (p1_nn(i)>10)
        p1_nn(i)=0;
    end
    if (p2_nn(i)>10)
        p2_nn(i)=0;
    end
end

figure
plot(x,p1_nn,'r-',x,p2_nn,'b-')
legend('Class 1','Class 2');
```

```
title('Estimated Density Distribution');



% Classification

error_nn_total = 0;
error1 = 0;
error2 = 0;

for i = 1:test_data
    j1_nn = find(abs(x-x1_test(i)) <=0.1);

    if (p1_nn(j1_nn) < p2_nn(j1_nn))
        error1 = error1 +1;
    end

    j2_nn = find (abs(x-x2_test(i))<=0.1);

    if(p2_nn(j2_nn) < p1_nn(j2_nn))
        error2 = error2 +1;
    end
end
error_nn_total = (error1 + error2)/2/test_data
```

```
clear all
close all
clc

load x1_train
load x1_test
load x2_train
load x2_test

n = 100000;
train_data = n/2;
test_data = n/2;

%---------- TRAIN ---------------%

% Parzen window
d = 1;  % dimention
x = -5:0.2:10;
L_x = length(x);
%setting h1
h1 = 1;
hn = h1/sqrt(train_data);
Vn = hn^d;

Q1 = zeros(1,train_data);
prob1_train = zeros(1,L_x);

Q2 = zeros(1,train_data);
prob2_train = zeros(1,L_x);

for i = 1:L_x
    for j = 1:train_data
        Q1(j) = 1/(sqrt(2*pi))*exp(-(x(i) - x1_train(j))^2/(2*hn^2));
        Q2(j) = 1/(sqrt(2*pi))*exp(-(x(i) - x2_train(j))^2/(2*hn^2));
        prob1_train(i) = prob1_train(i) + 1/train_data*1/Vn*Q1(j);
        prob2_train(i) = prob2_train(i) + 1/train_data*1/Vn*Q2(j);
    end
end



% KNN
knn = ceil(sqrt(train_data));
p1_knn = zeros(1,L_x);
p2_knn = zeros(1,L_x);

for i = 1:L_x
    index_sort1 = sort(abs(x1_train - x(i)));
    V1 = 2 * index_sort1(knn);
    index_sort2 = sort(abs(x2_train - x(i)));
    V2 = 2 * index_sort2(knn);
    if (V1 > 0)
        p1_knn(i) = knn/train_data/V1;
    end
    if(V2 > 0)
        p2_knn(i) = knn/train_data/V2;
    end
    if (p1_knn(i)>10)
```

```
        p1_knn(i)=0;
    end
    if (p2_knn(i)>10)
        p2_knn(i)=0;
    end
end



% NN
kn = 1;
p1_nn = zeros(1,L_x);
p2_nn = zeros(1,L_x);

for i = 1:L_x
    index_sort1 = sort(abs(x1_train - x(i)));
    V1 = 2 * index_sort1(kn);
    index_sort2 = sort(abs(x2_train - x(i)));
    V2 = 2 * index_sort2(kn);
    if (V1 > 0)
        p1_nn(i) = kn/train_data/V1;
    end
    if(V2 > 0)
        p2_nn(i) = kn/train_data/V2;
    end
    if (p1_nn(i)>10)
        p1_nn(i)=0;
    end
    if (p2_nn(i)>10)
        p2_nn(i)=0;
    end
end

figure
plot(x,prob1_train,'r-',x,prob2_train,'g-',x,p1_knn,'b-',x,p2_knn,'k-',x,p1_nn,'m-',x,p2_nn,
'c-','LineWidth',2);
legend('Class 1 Parzen','Class 2 Parzen','Class 1 KNN','Class 2 KNN','Class 1 NN','Class 2
NN');
title('Comparison of Estimated Density Distribution');

%---------- TEST ----------%

%Parzen window method
error1 = 0;
error2 = 0;

for i = 1:test_data
  parzen = find(abs(x-x1_test(i)) <= 0.1);
    if (prob1_train(parzen) < prob2_train(parzen))
        error1 = error1 + 1;
    end
    parzen2 = find(abs(x-x2_test(i)) <= 0.1);
    if(prob2_train(parzen2) < prob1_train(parzen2))
        error2 = error2 + 1;
    end
end

error_total = error1 + error2
```

```
error_parzen_prob = error_total/(2*test_data)

% KNN
error_knn_total = 0;
error1 = 0;
error2 = 0;

for i = 1:test_data
    j1_knn = find(abs(x-x1_test(i)) <=0.1);

    if (p1_knn(j1_knn) < p2_knn(j1_knn))
        error1 = error1 +1;
    end

    j2_knn = find (abs(x-x2_test(i))<=0.1);

    if(p2_knn(j2_knn) < p1_knn(j2_knn))
        error2 = error2 +1;
    end
end
error_knn_total = (error1 + error2)/2/test_data

% NN
error_nn_total = 0;
error1 = 0;
error2 = 0;

for i = 1:test_data
    j1_nn = find(abs(x-x1_test(i)) <=0.1);

    if (p1_nn(j1_nn) < p2_nn(j1_nn))
        error1 = error1 +1;
    end

    j2_nn = find (abs(x-x2_test(i))<=0.1);

    if(p2_nn(j2_nn) < p1_nn(j2_nn))
        error2 = error2 +1;
    end
end
error_nn_total = (error1 + error2)/2/test_data
```