3 April 2008

Homework 2

EE662 – Pattern Recognition

Deen King-Smith
dkingsmi@purdue.edu

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 $J(w)=w^TS_Bw / w^TS_ww$. 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 $J(w)= w^TS_Bw$ instead (i.e. setting S_w as the identity matrix in the solution w_o? Investigate this question by numerical experimentation.

Using matlab, two sets of data of random data were generated based on were generated to compare the use of both cost functions.

The data used is plotted below:

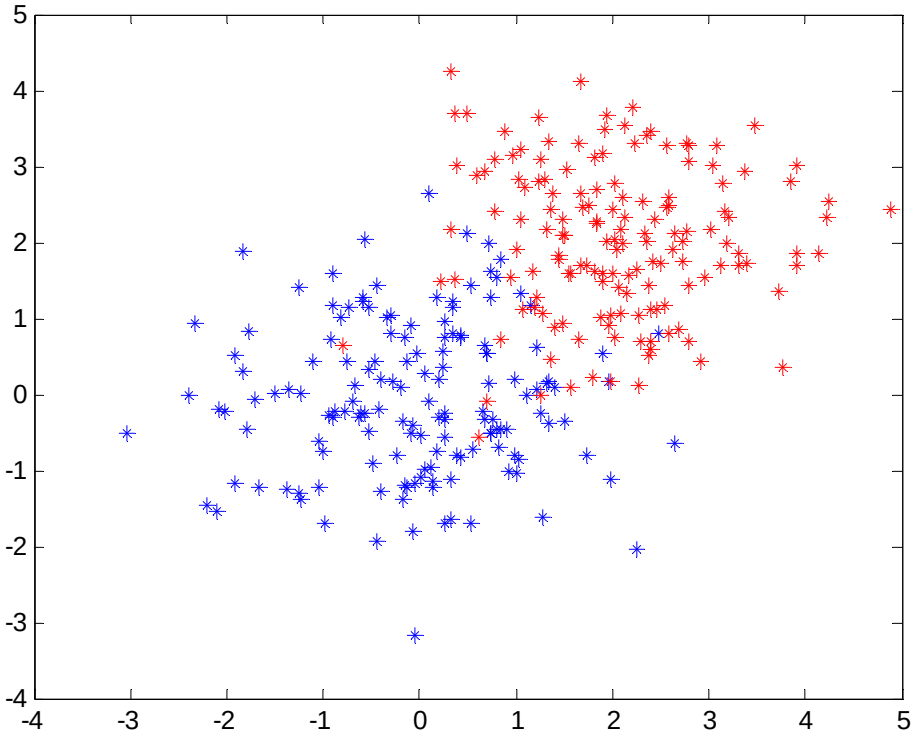

Figure 1

Above, data set 1 is in blue and data set 2 is in red.

The solution to the cost function $J(w)=w^TS_Bw / w^TS_ww$, is $w_0= S_w^{-1}(m_1-m_2)$. To compare the two we take both values of w_0 and project them onto the data sets above.

Using S_w calculated normally:

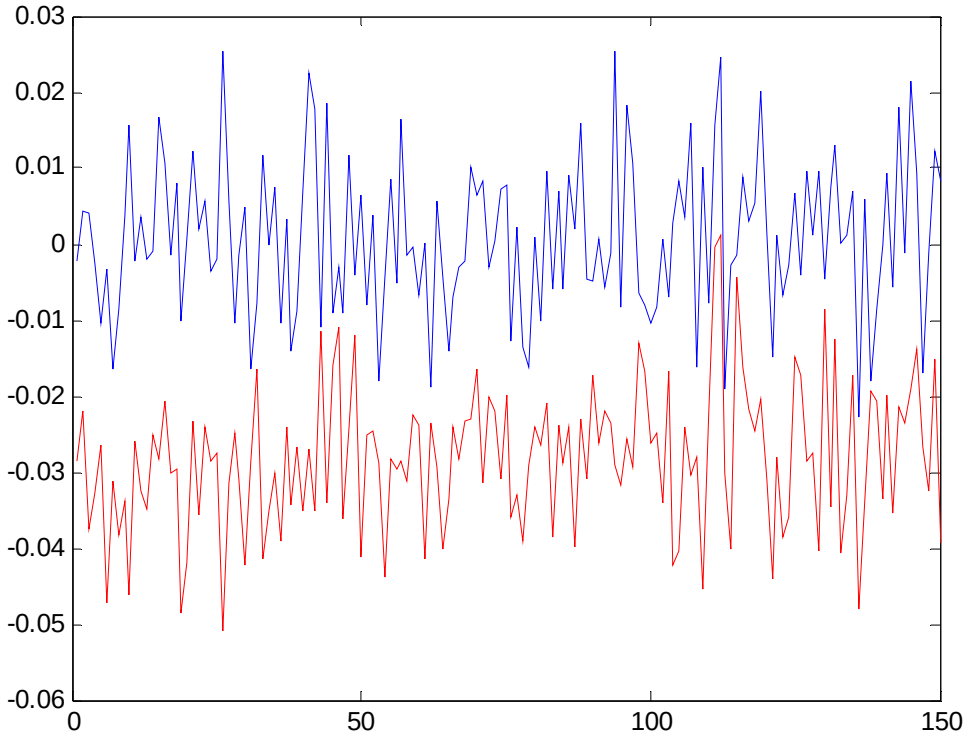

Figure 2

Using the identity matrix for S_w:

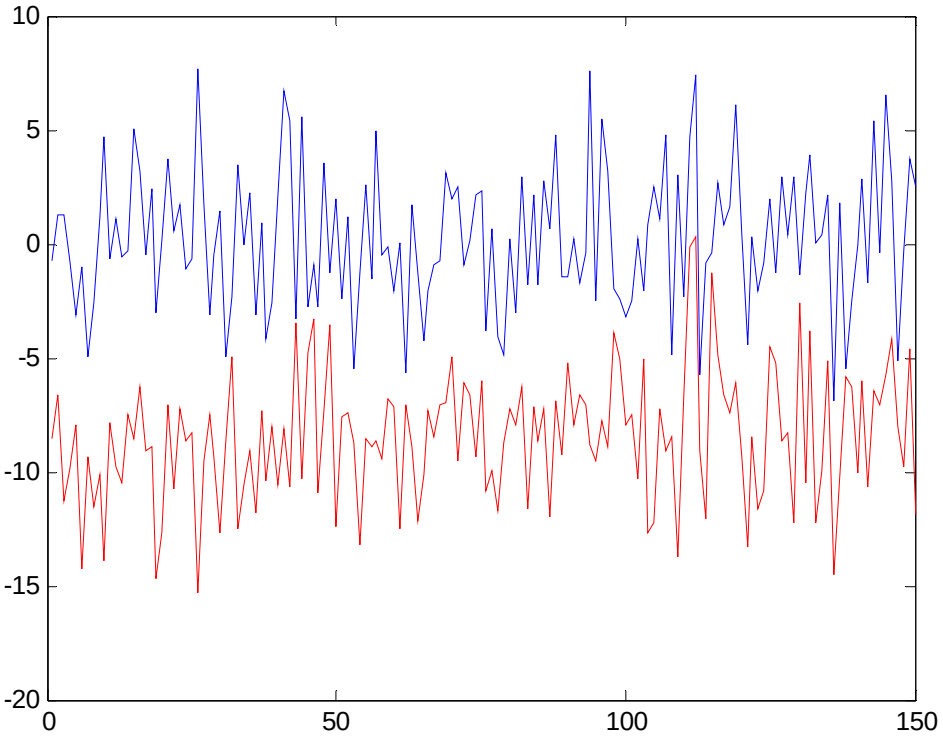

Figure 3

The waveforms generated by both projections look nearly identical, the difference being in the scale of the projection. The original method provides the equivalent normalized of w_0 where as the version proposed in class is not.

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.

Note: you may use code downloaded from the web, but if you do so, please be sure to explain what the code does in your report and give the reference.

Neural Networks:

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected processing elements working in unison to solve specific problems. ANNs, like people,

learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the elements. This is true of ANNs as well. In most cases an ANN is an adaptive system that changes its structure based on external or internal information that flows through the network during the learning phase. The elements that receive input from outside the network is called the input layer. The output layer consists of elements that contain prediction and/or classification information. All other elements are contained in the hidden layer.

In our design the input elements are scaled between 0 - 1 or -1 - 1. When the values leave the input layer they move through the hidden layer. When each value traverses a between element layers, weights are assigned to each interconnecting line and is multiplied by the values. The values are summed, modified by the transfer function, and generally passed directly to all elements in the next layer with a weight assigned to each value. Values of the interconnecting weights predetermine the neural network's computation reaction to any arbitrary input pattern. As information is passed forward from the inputs toward the outputs, a back-propagation algorithm adjusts interconnecting weights during the learning phase so that known outputs will best match predicted outputs.

In order to adjust the weights based on the training patterns and matching the known outputs to the predicted outputs, each epoch changes the weight by an amount proportional to the different between the desired output and actual output according to $\Delta w = \eta(out_k - tar_k)x_i$

Where out is the target output, out is the actual output, and η is the learning rate.

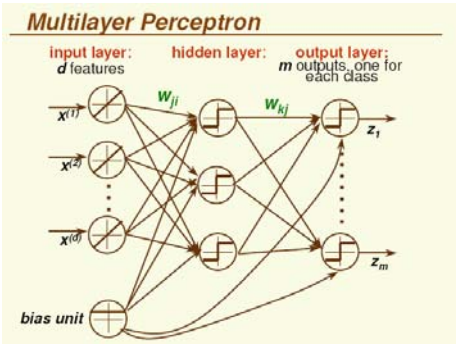

Figure 4

In our design we 1D samples with N(2,2) and N(-2,2) distributions for our experiments. The samples generated are divided into two with half used for testing and the other training. The training error is used to show the performance of the classifier. The input parameters are N1, N2, and N3 which are the number of nodes in the input, hidden, and output layer respectively.

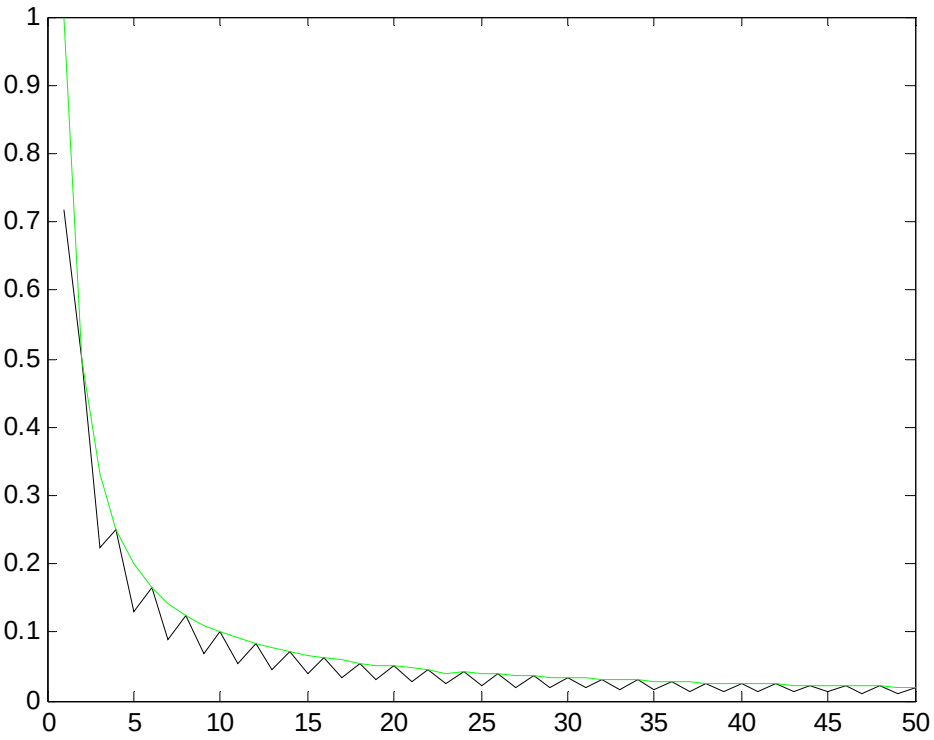

Figure 5 - N1=6, N2=12, N3=3

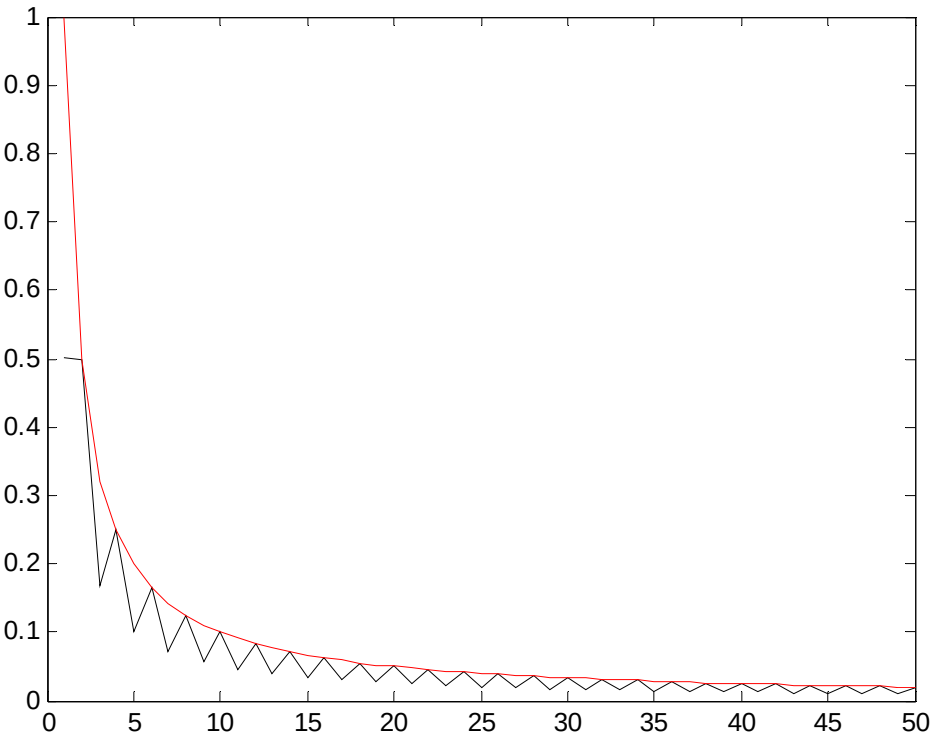

Figure 6 – N1=6, N2=18, N3=3

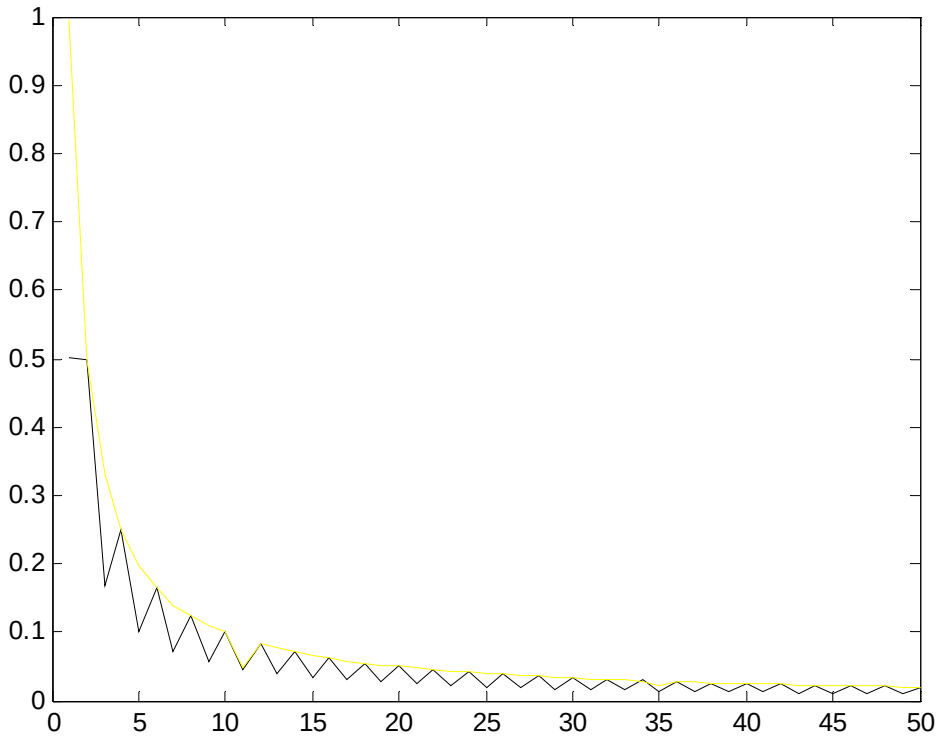

Figure 7 – N1=3, N2=12, N3=6

In each of the above figures the Y-axis is the error from the training data and X-axis is the number of epochs.

Support Vector Machines

Using the SVM approach, we view the input data as a two sets of vectors in an N-dimensional space, and then leverage the SVM to construct a hyperplane that maximizes the margin between the sets of data. The Hyper-plane based on this margin is known as the Maximum-Margin Hyperplane. To accomplish this we map the binary set of training data to a feature space with a high dimension. Mathematically this can be expressed as the following:

- We want to maximize

$$L(\alpha) = \sum_{k=1}^{n} \alpha_k - \frac{1}{2} \sum_{k=1}^{n} \sum_{j=1}^{n} \alpha_k \alpha_j z_k z_j k(y_j, y_k)$$

Within the constraints

$$\sum_{k=1}^{n} \alpha_k z_k = 0 \qquad \alpha_k \geq 0, \quad k = 1, \ldots\ldots, n$$

- From the above we are able to define the kernel function as

$$k(y_j, y_k) = y_j^t \cdot y_k = \phi(x_j)^t \cdot \phi(x_k)$$

To implement this in code we use Quadratic programming. This special type of mathematical optimization allows us to compute a global minimize if there exist at least one vector satisfying the above constraints and that L(a) is bounded on the feasible region.

For out design we use data similar to Neural Networks except we use the N(1,2) and N(-1,2) for each respective classes.

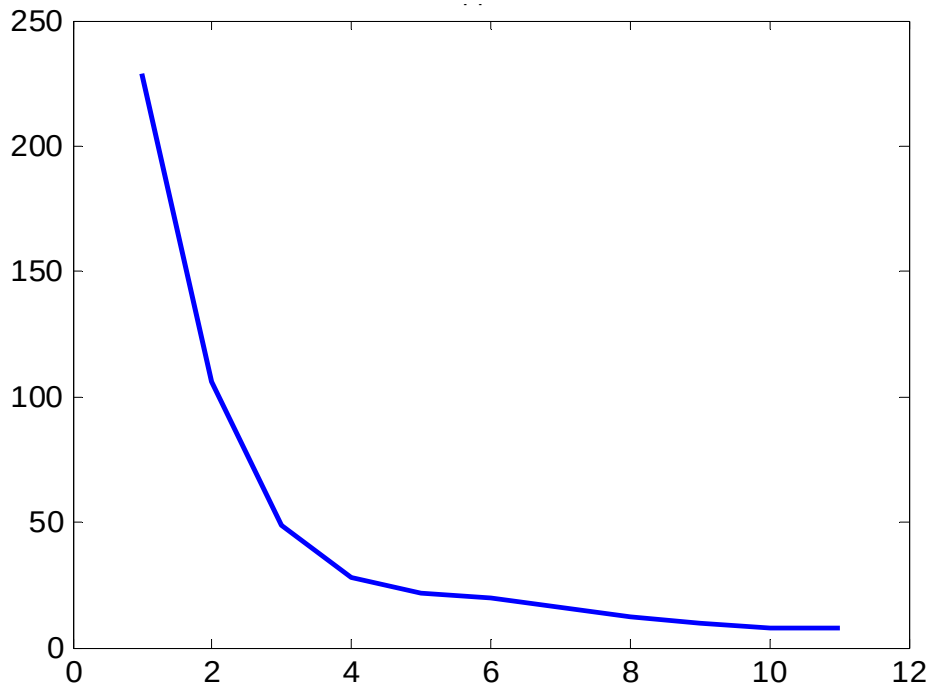

Figure 8 - # of Support Vectors

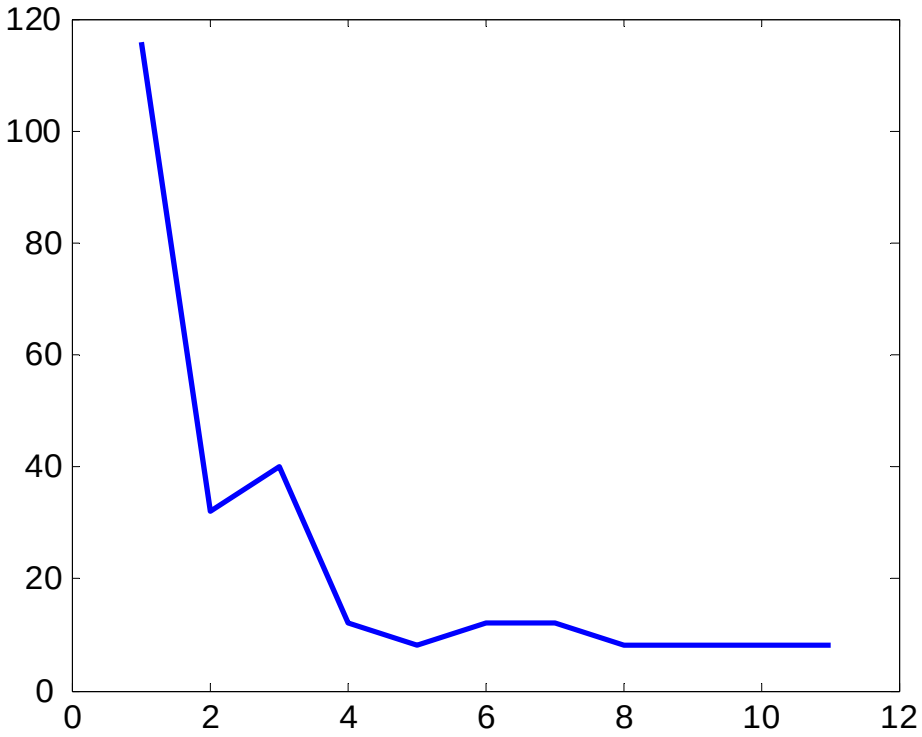

Figure 9 - # Of Misclassifications

The above figures show the number of support vector and respective misclassifications generated by the designed SVM classifier.

Comparing the two methods we see that SVM are closely related to Neural Networks. A significant advantage of SVMs is that Neural Networks can suffer from multiple local minima, where the solution to an SVM is global and unique. Two more advantages of SVMs are that they have a simple geometric interpretation and give a sparse solution. Unlike Neural Networks, the computational complexity of SVMs does not depend on the dimensionality of the input space. Neural network use empirical risk minimization, where SVMs use structural risk minimization. The reason that SVMs often outperform Neural Networks in practice due to the fact that they deal with overfitting which is the biggest problem with Neural Networks.

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

To Illustrate the difference between the these techniques we start with the Nearest and K-nearest neighbor techniques

K-Nearest Neighbor, and Nearest Neighbor

The Nearest Neighbor techniques are methods for classifying data based on the closest training examples in the feature space. The purpose of this algorithm is to classify a new object based on attributes and training samples. The classifier does not use any model to fit and only based on memory. Given a query point, we find K number of objects or training points closest to the query point. The classification is using majority vote among the classification of the K objects. Any ties can be broken at random. K Nearest neighbor algorithm uses neighborhood classification as the prediction value of the new query instance.

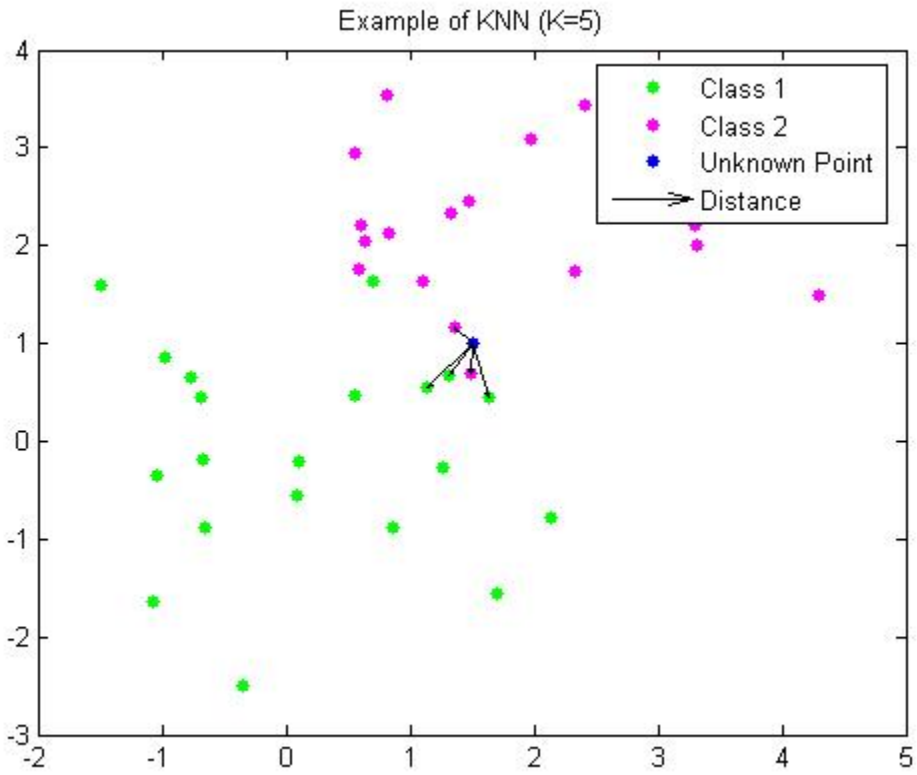

Figure 10 – An example of KNN

In the figure above the unknown data point is classified into class 1 or class 2, using KNN with K = 5. In this case the 5 nearest neighbors are used for voting. Euclidean distance is used for the metric and due to the existence of more samples from Class 1 than Class 2, the point is classified as Class 1. During kNN, the volume is centered on the unknown point and grows until it captures k samples for classifications.

Nearest Neighbor is a special case of K-Nearest Neighbor. In NN k is set to 1 and unknown data will be classified in the class that it's nearest neighbor is in.

In our design we 1D samples with N(2,2) and N(-2,2) distributions for our experiments. The samples generated are divided into two with half used for testing and the other training. The total number of samples point = 200 and the number of neighbors = 5.

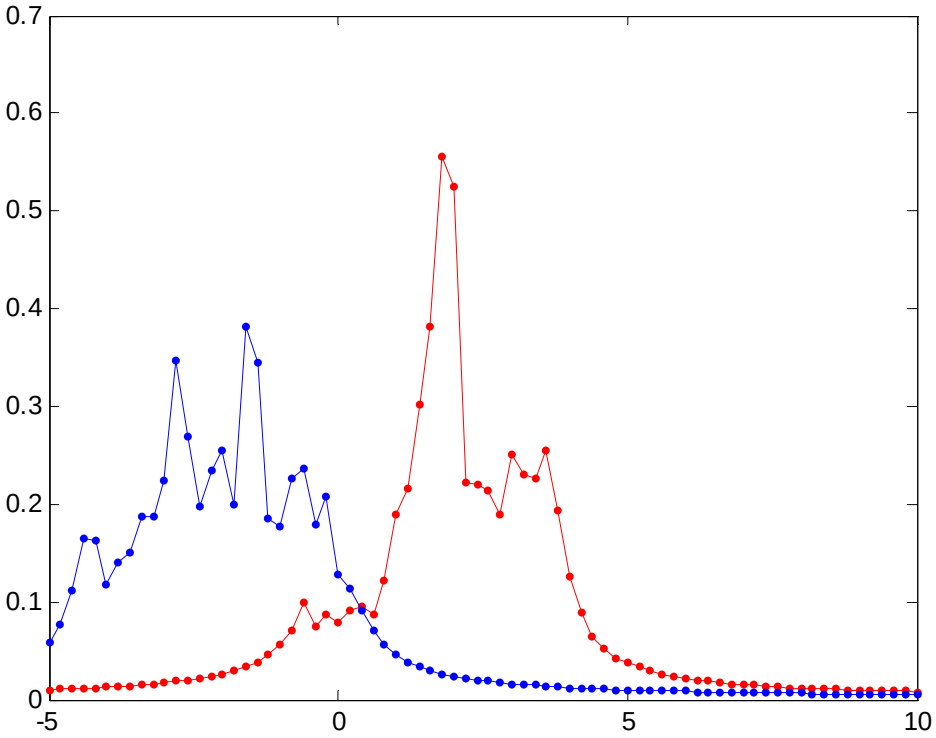

Figure 11 – kNN

In the figure above, we plot the probabilities of Class 1(red) and Class 2(blue) verses an unknown point X_0.

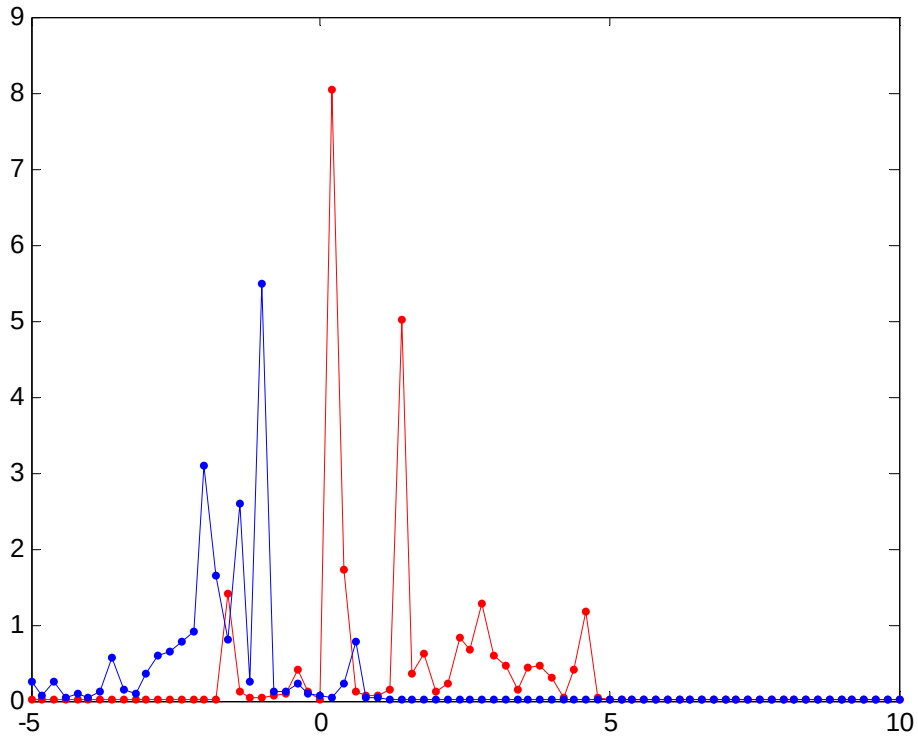

Figure 12 – NN

As with figure 11, we show the probabilities of Class 1 and Class 2 versus the unknown point X_0.

Classification is based on the $P(x \mid w_i)$, i=1, 2 where $p_n(x \mid w_i) = \frac{k_i / n}{V}$. The respective $P(x \mid w_i)$'s are compared and the class with larger probability is chosen for the class of X_0.

When we compare kNN and NN the difference lies in the Error rate. kNN has the advantage here in that it is more robust to handle noise in the training data. This is because K points are used instead of 1, as is the case in NN. While this may be advantageous, there are drawbacks when using large values for K. This value of K depends on the data being used. For Normal distributions, it depends on the dimension of the data.

	K=1	K=3	K=5	K=7
Error Rate	0.35	0.29	0.25	0.24

Experiments show that as K get larger in normally distributed data the error also gets larger. The optimal value for K is found to be $k_n = \sqrt{n\_training}$.

Parzen Windows

Parzen windows classification is a technique for nonparametric density estimation, which can also be used for classification. Using a given kernel function, the technique approximates a given training set distribution via a linear combination of kernels centered on the observed points. In this work, we separately approximate densities for each of the two classes, and we assign a test point to the class with maximal posterior probability.

The resulting algorithm is extremely simple and closely related to support vector machines. The decision function is

$$f(\mathbf{X}) = sign(\sum y_i K(\mathbf{X}_i, \mathbf{X})),$$

where the kernel function *K* is the radial basis function of:

$$K(\mathbf{X}, \mathbf{Y}) = exp\left(\frac{-\|\bar{\mathbf{X}} - \bar{\mathbf{Y}}\|^2}{2\sigma^2}\right),$$

The Parzen windows classification algorithm does not require any training phase; however, the lack of sparseness makes the test phase quite slow. Furthermore, although asymptotical convergence guarantees on the performance of Parzen windows classifiers exist, no such guarantees exist for finite sample sizes.

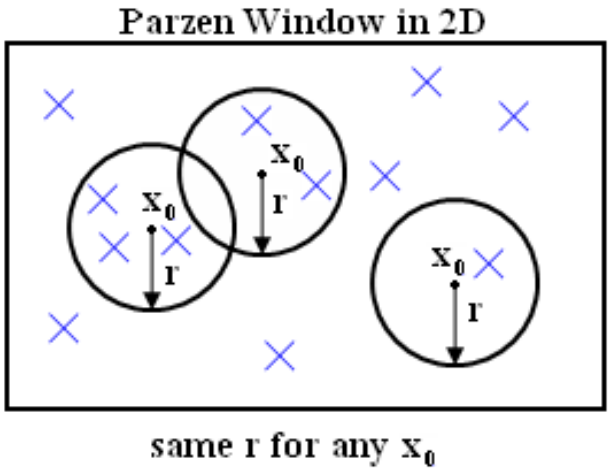

Figure 13

Figure 13 shows the unknown data point is classified into either Class 1 or Class 2. If the density of samples is high near x, the cell will capture more samples and conversely if the density if low the cell will capture less samples.

In order to estimate density we need a window function $\phi(v)$. The estimated density is:

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

Examples of a window functions are Gaussian, Unit, and Triangle functions. For our purposes the density is:

$$p(x) \approx \frac{k / N}{V}$$

Where x is the sample inside some region R, K is the volume of the samples, N is the total number of samples, and V is the volume of R.

When choosing the window size h, we are making a guess the region where density is approximately constant. The smaller h is there exist n sharp pulses centered on the data. These are super imposed so that the result will be noisy and not smooth. Conversely is h is too large, the result will lack detail and will be over smoothed.

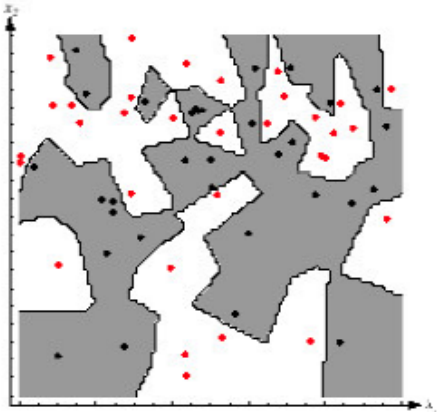

Figure 14 – Small h

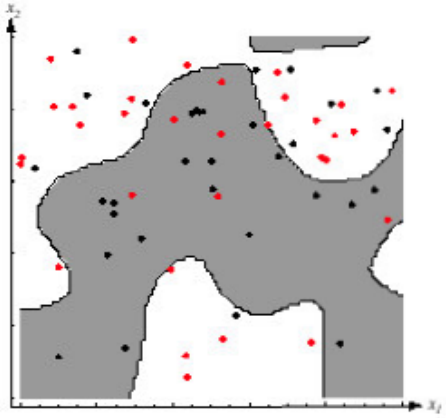

Figure 15 – Large h

For our experiment we used N(1,sqrt(2)) and N(-1, sqrt(2)) for our samples, along with different windows sizes, number of samples. This is to show their effect on estimation. The results from the experiments are shown below.

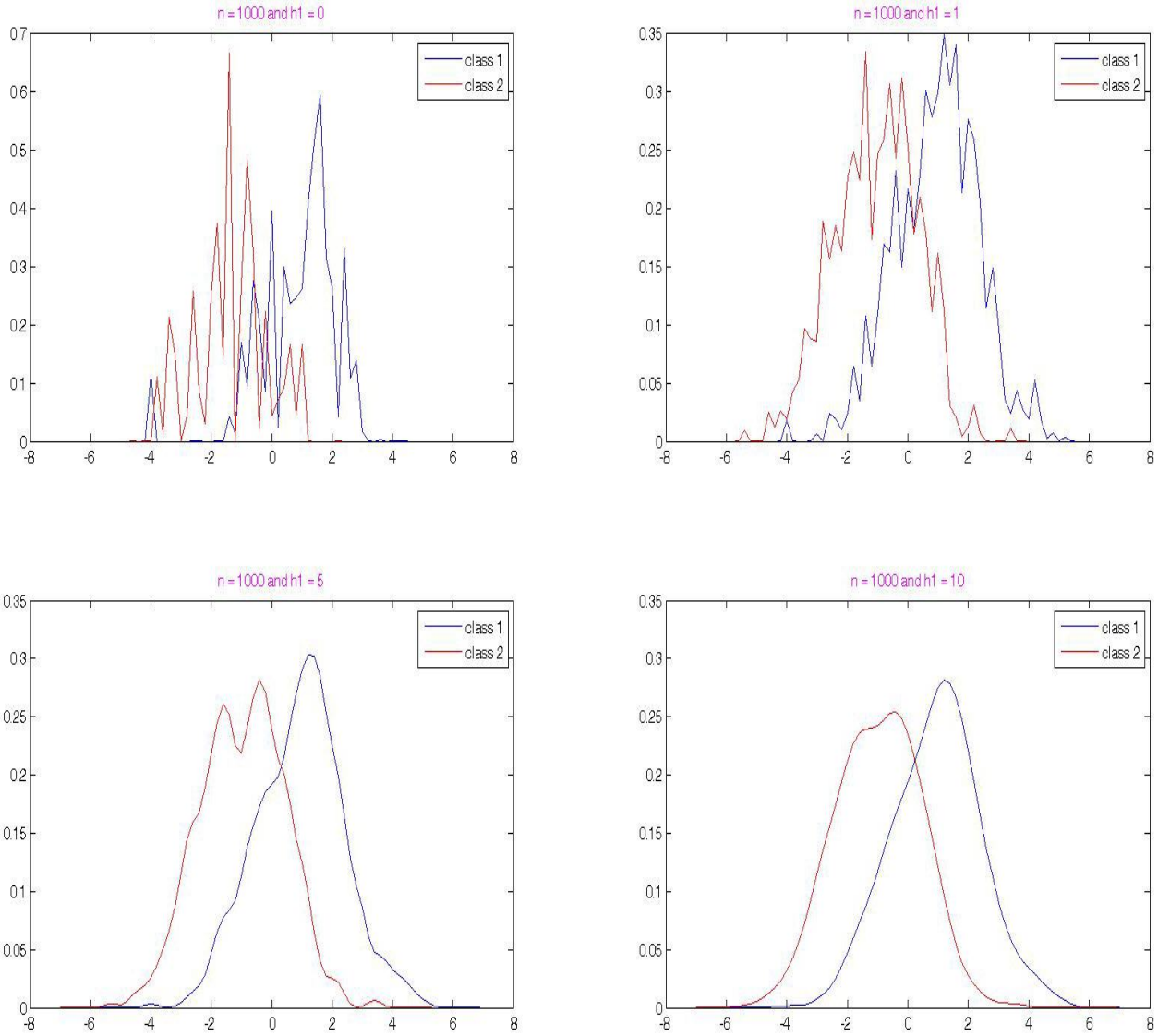

Figure 16: Number of samples used = 1000 and window size = 0.1,1 , 5, 10

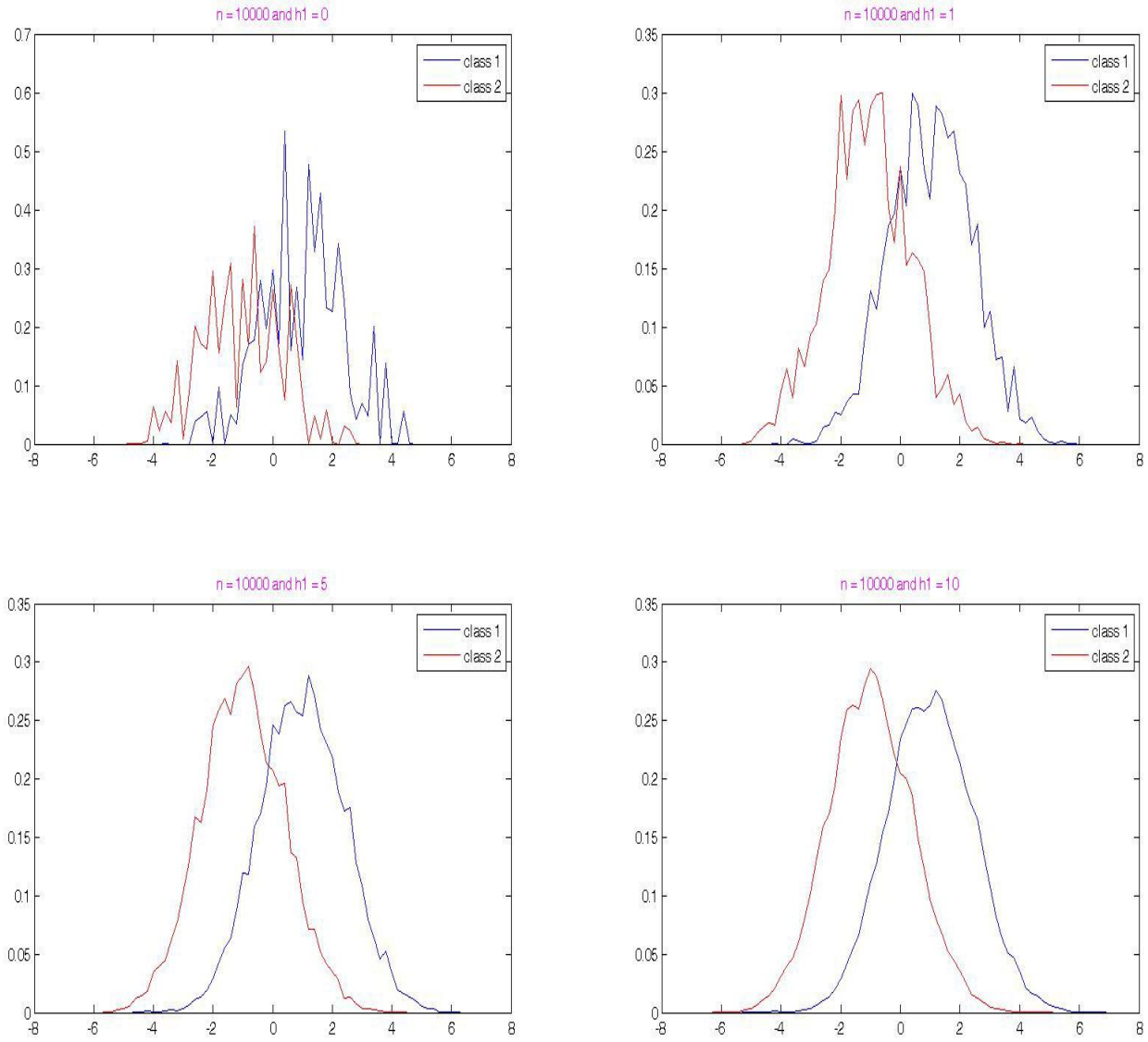

Figure 17: Number of samples used = 10000 and window size = 0.1,1 , 5, 10

We can see clearly from the results that if the window size is too large, the figure looses much of it's details.

Parzen windows can be regarded as a generalization of *k*-nearest neighbor techniques. Rather than choosing the *k* nearest neighbors of a test point and labelling the test point with the weighted majority of its neighbors' votes, one can consider all points in the voting scheme and assign their weight by means of the kernel function. With Gaussian kernels, the weight decreases exponentially with the square of the distance, so far away points are practically irrelevant.

Both Parzen and kNN methods can generate very complex decision boundaries. The main difference is that instead of looking at the k closest points to a piece of training data, all points within a fixed distance are considered. In practice, one difference is that datasets with large gaps get treated much different. kNN will pick far away points, but it is possible that relatively small Parzen Windows will actually enclose zero points. In this case, only the priors can be used to classify.

Code:

Problem 1:

```
% 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 0]';
Mean2 = [ 2 2]';
std1 = [1 0; 0 1];
std2 = [1 0; 0 1];
data_class1 = mvnrnd(Mean1,std1,n1);
data_class2 = mvnrnd(Mean2,std2,n2);
plot(data_class1(:,1),data_class1(:,2),'b*');hold on;
plot(data_class2(:,1),data_class2(:,2),'r*');
x1=data_class1;
x2=data_class2;

figure(2),
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);


w_opt=S_W\bet_scatter';

S_W_I=eye(f,c);  % try with Sw set to ident matrix
w_opt_I=S_W_I\bet_scatter';


% Projections
y1 = x1*w_opt;
y2 = x2*w_opt;

bin = 0.1;
x = -25:bin:25;
xa = 1:length(y1);
xb = 1:length(y2);
plot(xa,y1,'b',xb,y2,'r');
%plot(y1,'k');hold on;
%plot(y2,'g');hold off;

figure(3),

% Projections
y1_I = x1*w_opt_I;
y2_I = x2*w_opt_I;

bin = 0.1;
x = -25:bin:25;
xa= 1:length(y1_I);
xb=1:length(y2_I);
plot(xa,y1_I,'b',xb,y2_I,'r');
```

Problem 2a

```
% performs backpropagation algorithm
```

```
close all;
clear all;
%rand('state',100);
% the neurons have a sigmoid function activation
% data length
N1 = 3;
N2 = 12;
N3 = 6;
% length training set
% iter = epochs
iter = 50;
iter_test = 50;
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 = 2;
Mean2 = -2;
std1 = 2;
std2 = 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,'k'); 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,y,'b'); hold on;
plot(x,error_epoch_test,'y'); hold off;
W_hid_in,
W_hid_out,
```

Problem 2b

```
clear;    % clear variables from memory
close all;
nsample = 100;
X = zeros(nsample,1);
Y = zeros(nsample,1);
Mean1 = 0;
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,'ko');hold on;
plot(data_class2,'g+');
p = randperm(nsample);
Y(p(1:nsample/2)) = -1;
```

```
Y(p(nsample/2+1:nsample)) = 1;

% the trade-off weights we want to investigate , 2000, 5000, 10000, 100000
C = [0.1, 1, 5, 10, 20, 50, 100, 200, 500, 1000];

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)),
   % construct Hessian matrix; Hessian matrix is the Q matrix in our slides; also called Kernel
matrix
   H = zeros(nsample, nsample);    % initialize H; set H to a nsample * nsample zero matrix
   for i = 1 : nsample,
      for j = 1 : nsample,
         H(i,j) = X(i)*X(j)*Y(i)*Y(j);            % !!! please write your answer here !!!
      end
   end
   H = H+1e-10*eye(size(H));       % add 1e-10 to the main diagonal of H; a trick to make H stable
   F = -ones(nsample,1);          % F' * Alpha  corresponds to sigma_i(Alpha_i) in object function
   % set up equality constraints
   A = Y';    % corresponds to sigma_i(Alpha_i * Y_i) = 0
   b = 0;

   % set up upper and lower bounds for alpha: LB <= Alpha <= UB
   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 = qp(H, F, A, b, LB, UB, Alpha0, 1);

   % 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)];    % the operation A = [A, v] appends v to matrix A
```

```
   % number of support vectors
   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

% plot C_margin, C_trainingerror, C_misclassification, C_nsupportvector
% please use your code to make better plots instead of ours

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)');
```

Problem 3a,b

```
% Nearest Neighbor
clear all
close all

% sample data
n = 200;
train_data = n/2;
test_data = n/2;

% Data set 1:
mean_x1 = 2;
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 = -2;
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.-')


% 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
```

Problem 3c

```
clear all
close all

n = 500;
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 = 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


figure
plot(x,prob1_train,'k.-', x,prob2_train,'g.-')


% 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)
```