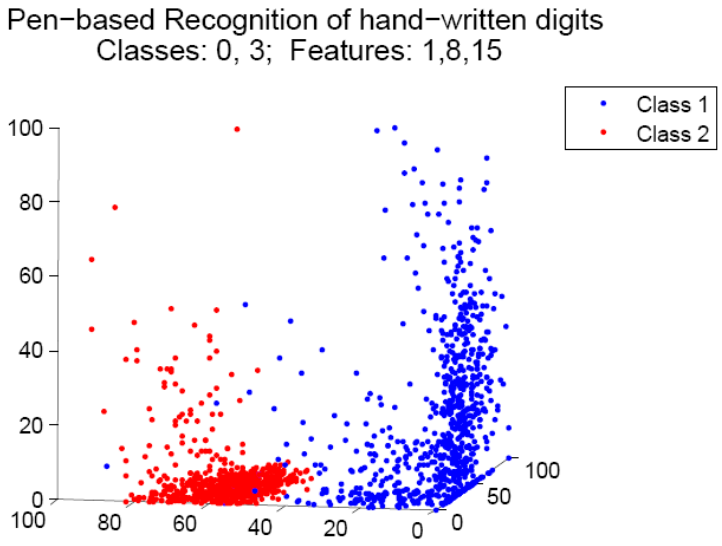
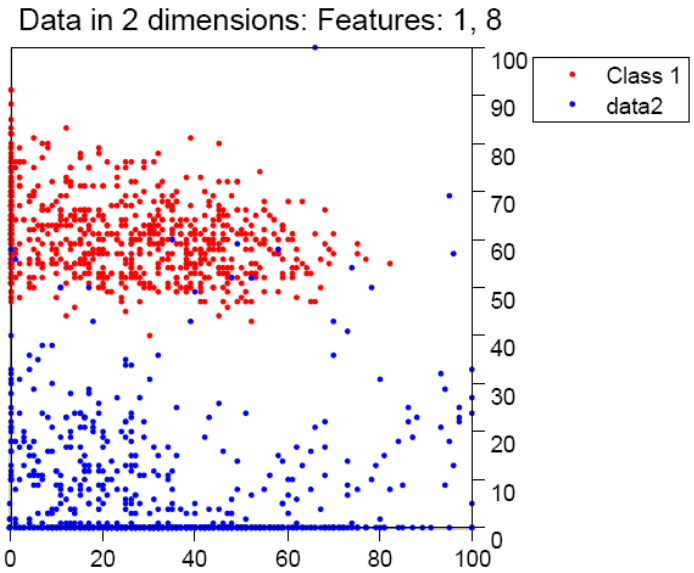
Solution 1:

As discussed in class, the cost function J(w) was chosen for minimization to obtain the optimal classifier between two classes because it helps ensure that the difference between means is large relative to some measure of the standard deviations within the classes. The other cost function namely with $S_W=I_{nxn}$, considers only the means. Therefore, a classifier based on J(w) will, in principle give lower error rate on the data on which it is trained.

For experimenting with this, I took a real data set from ***UCI-Machine Learning Repository***. The data is about ***"Pen-based recognition of hand-written digits"***. Although the original data set contains 10 classes, and 16 features, I chose two of the classes (class 0 and 3) and with 3 feature vectors (1, 8, 15) which appear to be linearly separable. In the following discussion, I refer the class labeled '0' as class 1 and class labeled '3' as class 2.

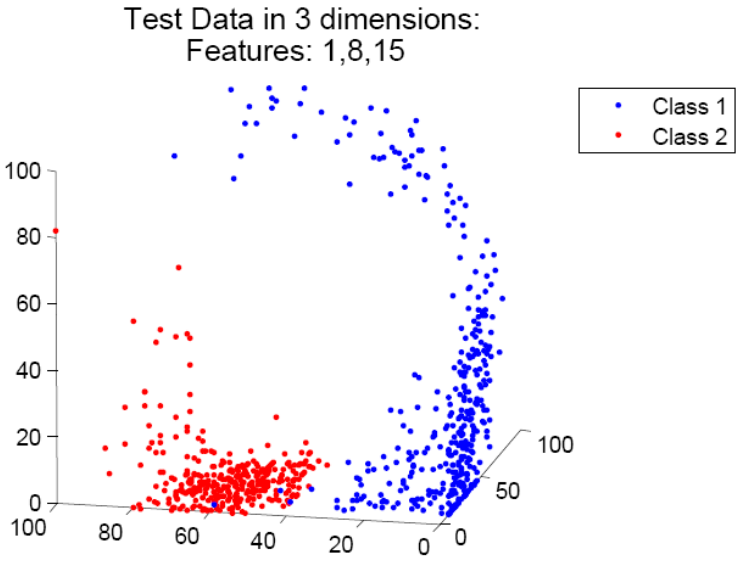


Training Samples in 3 dimensions

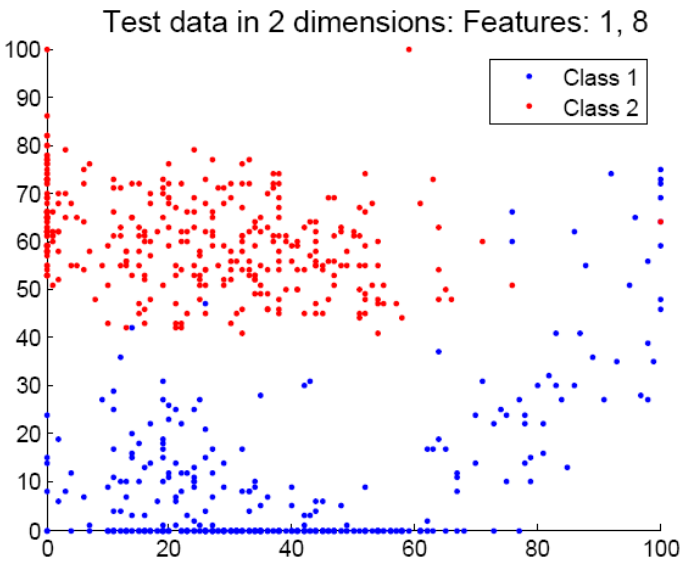


Training Samples in 2 dimensions

As we see from the above figure, the data is reasonably linearly separable on the basis of chosen features, whether we take features1 & 8 or 1, 8 &15.
Above shown data is the training data.
Below I show test data for the same.



Test Samples in 3 dimensions



Test Samples in 2 dimensions

Summary Information of the number of samples

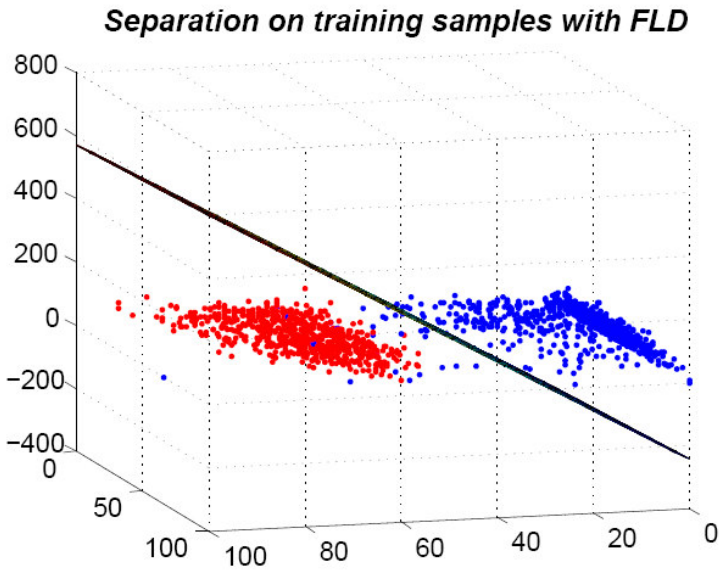
Number of samples	**Class 1**	**Class 2**	**Total**
Training samples	780	719	1499
Test samples	363	336	699

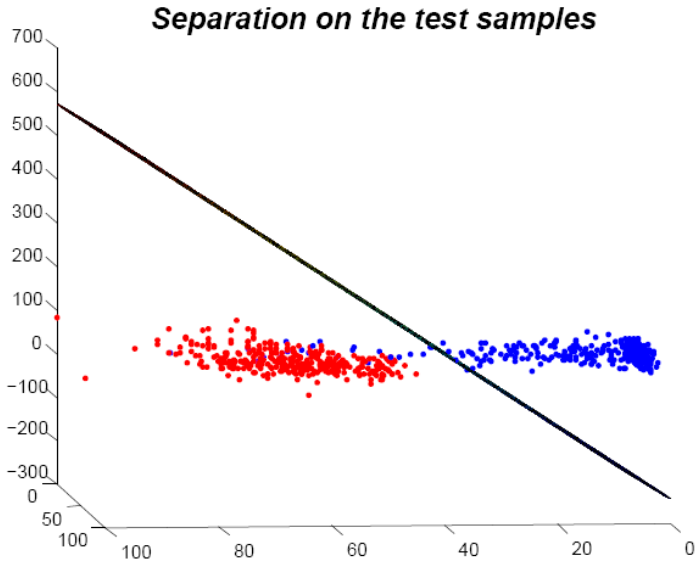
Result of Fisher Linear Discriminant (FLD) on 3 dimensions:

m_1 = [33.8846, 6.4205, 19.1462]
m_2 = [25.0821, 60.6815, 3.1794]
m = [29.6624, 32.4470, 11.4877]
σ
Σ_1 = [512.5662 -5.8669 186.8181
-5.8669 138.6180 -24.8320
186.8181 -24.8320 373.6889]

Σ_2 = [388.7457 -45.3828 -78.7727
-45.3828 66.3422 18.5815
-78.7727 18.5815 95.0568]

ω = 1.0e-003 * [-0.0116, -0.3494, 0.0414]
ω_0 = 0.0112
Error rate on the training samples = **0.0200**
Error rate on the test samples = **0.0343**





Result of Mean-Difference Discriminant (MDD) on 3 dimensions:

m_1 =[33.8846, 6.4205, 19.1462]
m_2 = [25.0821, 60.6815, 3.1794]
m = [29.6624, 32.4470, 11.4877]

Σ_1 = [512.5662 -5.8669 186.8181
-5.8669 138.6180 -24.8320
186.8181 -24.8320 373.6889]

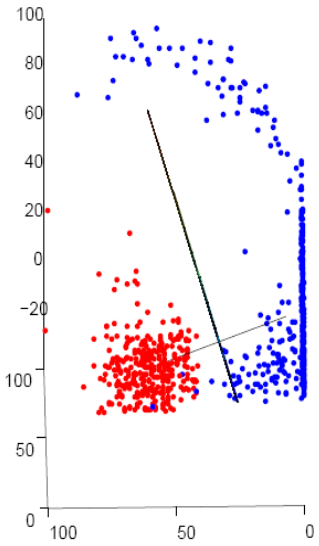
Σ_2 = [388.7457 -45.3828 -78.7727
-45.3828 66.3422 18.5815
-78.7727 18.5815 95.0568]

ω = [8.8026, -54.2610, 15.9667]

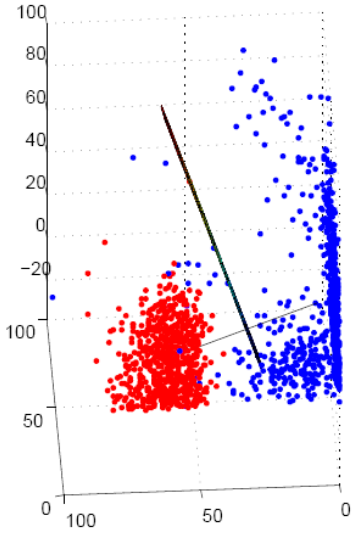
ω_0 = 1.3161e+003

Training error rate for MDD = **0.0273**

Test error rate for MDD = **0.0186**



Training Data Classification



Test Data Classification

The bold-appearing line in between is the edge of separating hyper-plane. The line joining m_1 and m_2 passes perpendicular through this hyper-plane.

Here, we observe that
Training error for MDD > Training error for FLD, as expected from theory.
Test error for MDD < Test error for FLD!
This may happen when the test data behaves quite differently from the training data.

We can verify if this is the reason here by comparing the mean and variance of test data.
m_1 = [38.5702, 9.1708, 28.4353]
m_2 = [24.1488, 60.3929, 4.0863]
From the mean vectors for test data, it appears that, if anything, the test data are further apart in comparison with the training data. This means that FLD should have performed better than MDD, even on the test cases. But what we observe is the opposite. I don't know of any precise reason of this, but one guess is that the samples which lie as exceptions outside the main distributions, cause this discrepancy. Since, our classifier is based on mean of distribution, and not on support vectors (like in SVM), it fails to resolve this.

But one thing that is important to observe is that FLD does perform better on the data on which it is trained. i.e. Training samples are always better classified by FLD, in comparison with FLD. The following experiment shows this

Now, let us determine FLD and MDD weight vectors for test samples themselves.
Applying FLD:
ω = 1.0e-003 * [0.0104, -0.6097, 0.2425]

ω_0 = 0.0162
Training error rate (over the test samples) = **0.0243**
Test error rate (over the training samples) = **0.0267**

Applying MDD:
ω = [14.4214, -51.2221, 24.3490]
ω_0 = 867.2749
Training error rate (over the test samples) = **0.0257**
Test error rate (over the training samples) = **0.0380**

Again we see that on reversing the test and training data, FLD gives better accuracy over the data used for training.

If we see the figures above showing separation between classes by a plane, we see that there are blue points on the red side also, which could easily be taken on that side by choosing ω_0 properly. Again, the reason behind this deficiency in performance of this method, is that it depends on means and variances (assuming homogeneous distribution of data), and does not take individual samples into account, as a support vector machine does. My guess is that a SVM would do better at this choice.

Solution 2

For this problem, I take a different dataset from the same source as cited above.

Classes: 5, 8

Features: <varying number of features ranging from 1 to 16>

First, I take feature {1}, then {1,2}, and add features in this way.

Neural Network Classification

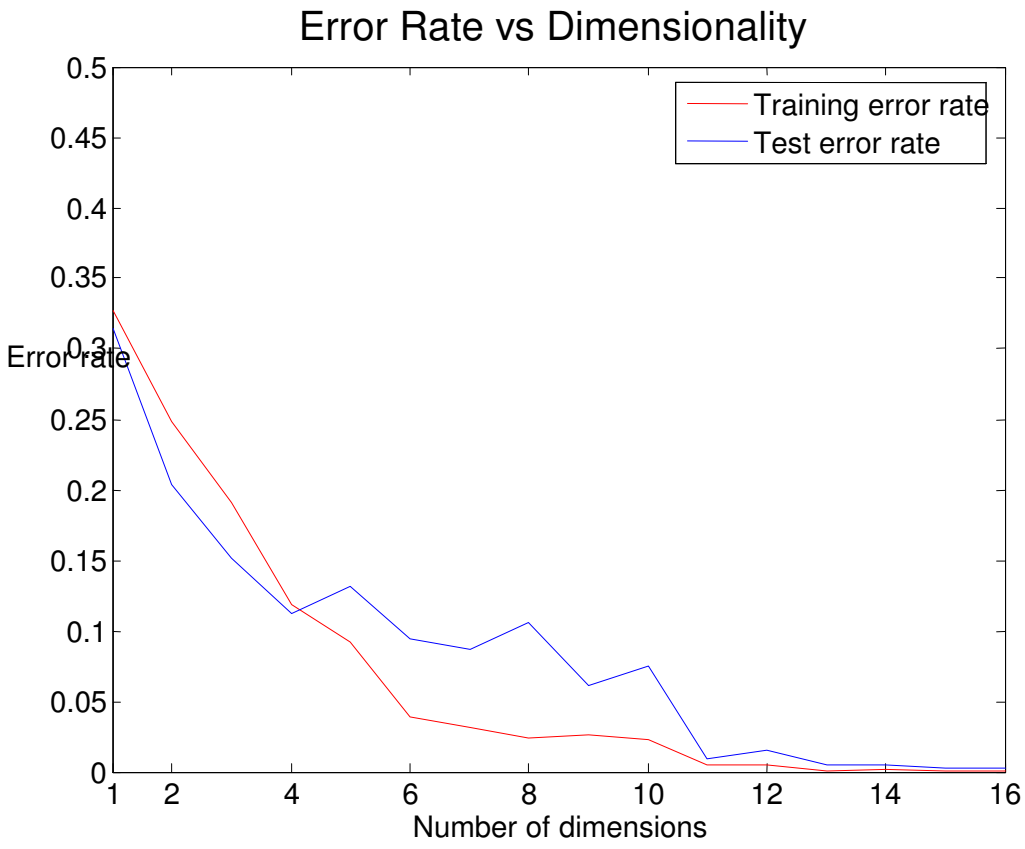
For Neural Network Classification, I use Matlab's toolbox interfaced with ***newff*** and ***train***. It provides for the choice of number of hidden layers, and number of neurons in each layer.

Matlab Program Options Used	
Matlab interface functions	***newff, train***
Transfer function for the hidden layers	***tansig***
Transfer function for the output layer	***Purelin***
Back-propagation network training function	***Trainlm***
Back-propagation weight/bias learning function	***Learngdm***

The procedure used for training is back-propagation based on **Levenberg-Marquardt optimization**. It is an iterative technique locating a local minimum of a multivariate function that is expressed as the sum of squares of several non-linear, real-valued functions by updating weight and bias values in the network.

Experiment 1

First experiment, I performed was to test the effect of number of dimensions on accuracy of the classifier. As discussed previously, this depends on the dataset. In my case, the dataset was not so strongly correlated, (as we will see for three dimensions) therefore, with increase in the dimensionality of feature vectors, the accuracy grew.
The network consisted of one hidden layer, with 20 neurons.
Following graph shows the resulting error rates.



From the above graph, we see that **for dimensions <= 3, the test error rate is less than training error rate**. This is because of the fact that test data is more well-behaved (i.e. grouped together separate from the complementary class) in comparison with training data.
This is shown in 3D below.

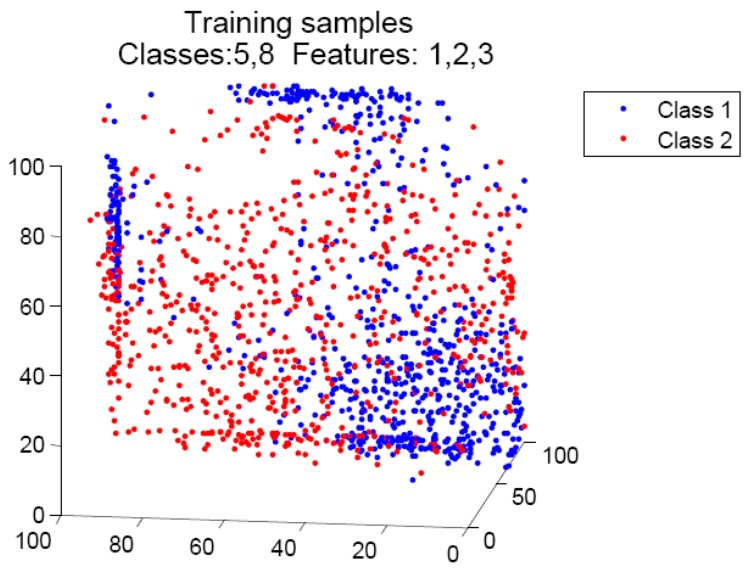


Figure showing training samples projected in 3 dimensions are not ‘well-classifiable’

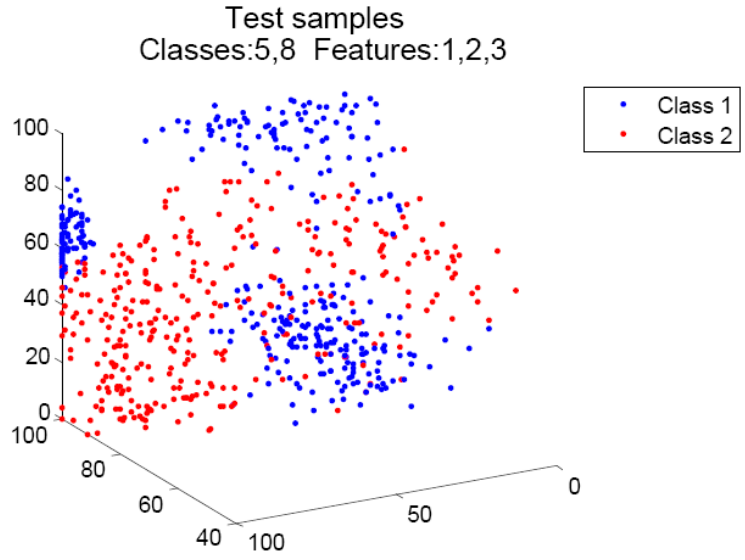
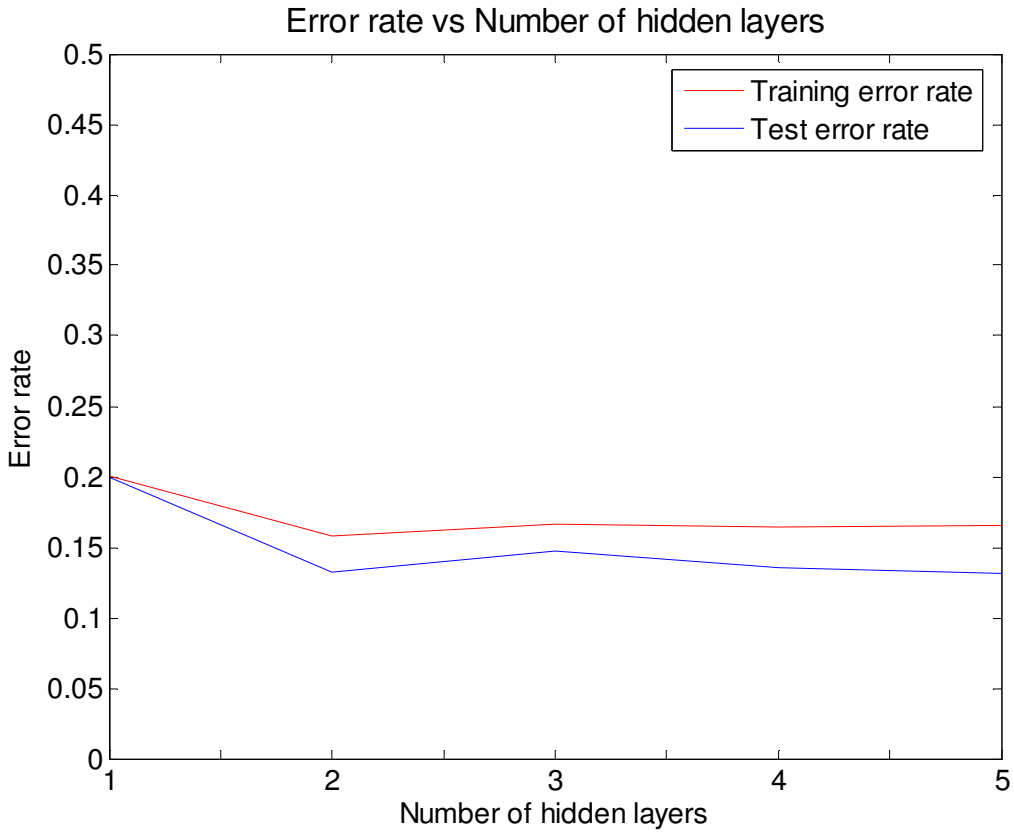


Figure showing test samples projected in 3 dimensions are more ‘classifiable’ than training samples

Experiment 2:

The next experiment I performed by varying the number of layers in the neural network.

As we see in the result of previous experiment, error rate is quite high for Dimensions<=3, I tried in this experiment to increase the number of layers from 1 through 5, each layer containing 5 neurons, and try to classify samples projected in 3 dimensions.
Following graph shows the result:



As discussed in the class, **the error rate does not decrease with increasing the number of layers**, because theoretically a one-hidden-layer network is as powerful as a multiple-hidden-layer network.

Experiment 3:
The third experiment is to compare error rates achieved by neural network on the original dataset of Problem 1, with Fisher Linear Discriminant.

	Training error rate	**Test error rate**
Neural Network classifier	0.0087	0.0286
Fisher Linear Discriminant	0.0200	0.0343

The above table shows that error rates are smaller for neural network. This is because of two reasons.

i) FLD is only a linear classifier and forms a hyperplane. Neural Network virtually forms a hypersurface.
ii) Neural Network performs well when the dataset is big. Here we have ~1500 samples, which gives the Neural network sufficient information about the

spread of data

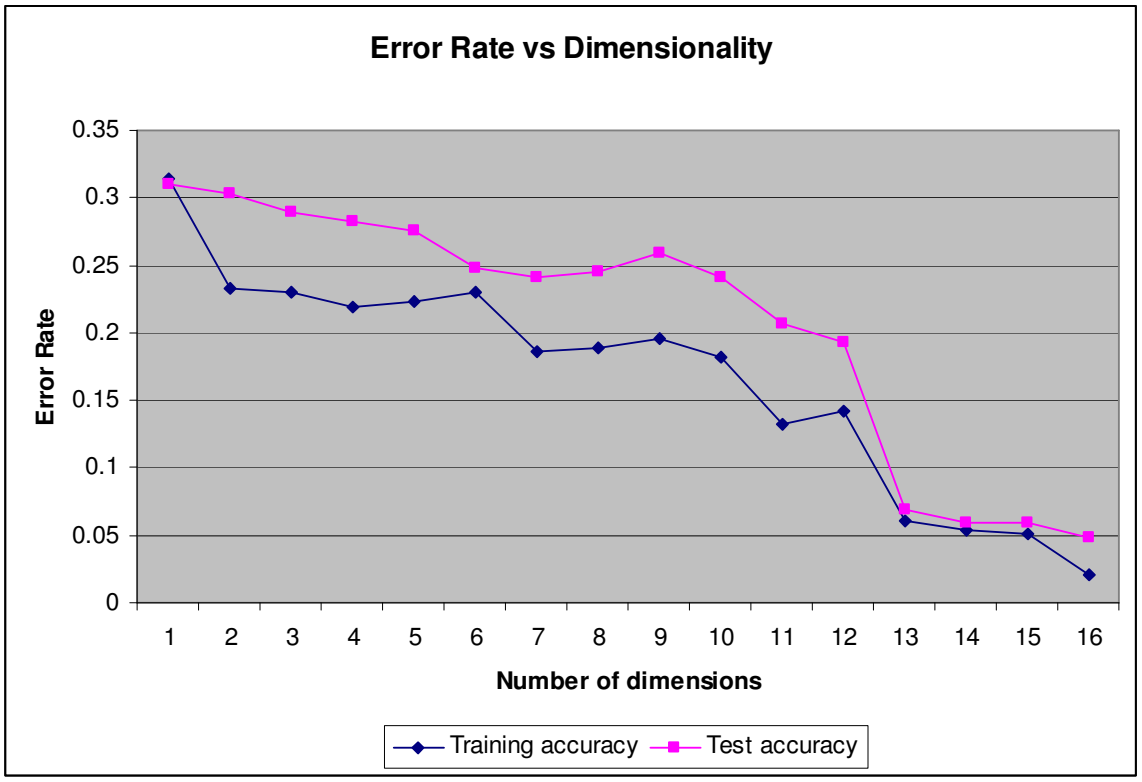
SVM Classification

The tool used for SVM classification is SVM *light*, by Cornell University. It is a C program. Along with classification it also solves the problem of regression and ranking, but I use it here only to do the classification. It is quite fast, being in C, and because it does not output any pictorial representation of classifier.

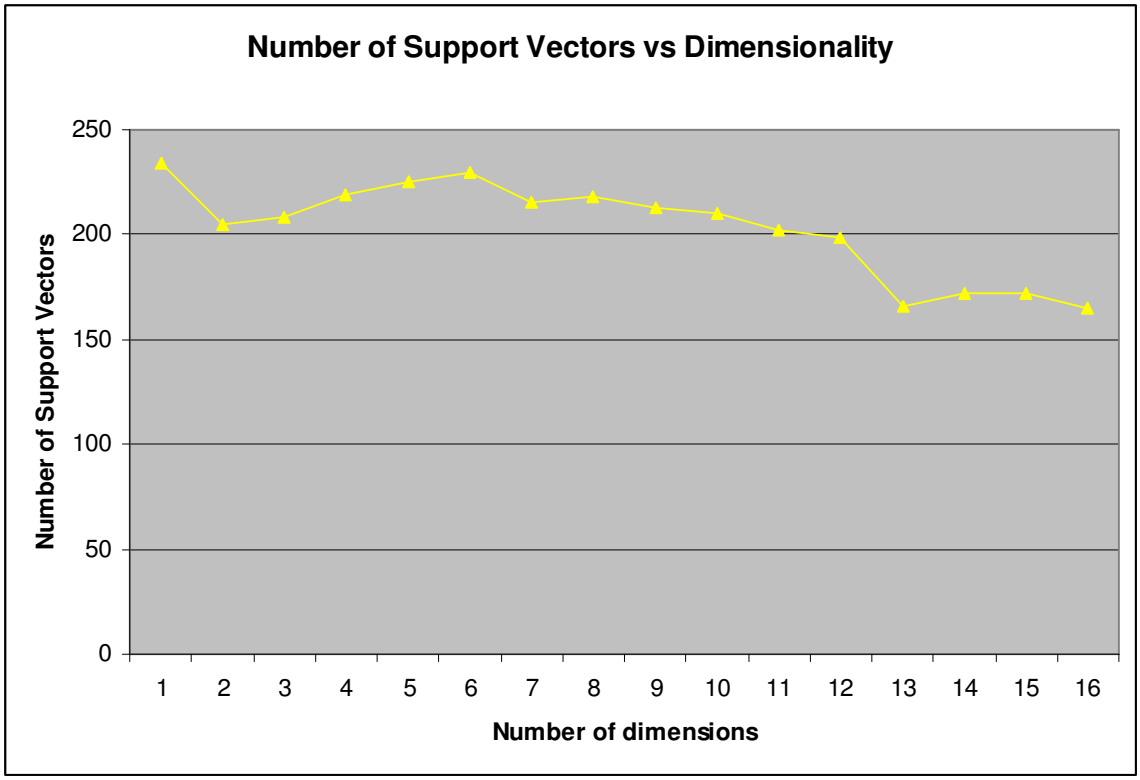
Experiment 1

The first experiment I did with SVM is the same as first experiment of Neural network classification, i.e. vary the dimensions. The data set is the same as used in Neural Network problem.

Following graph shows the summarized results:

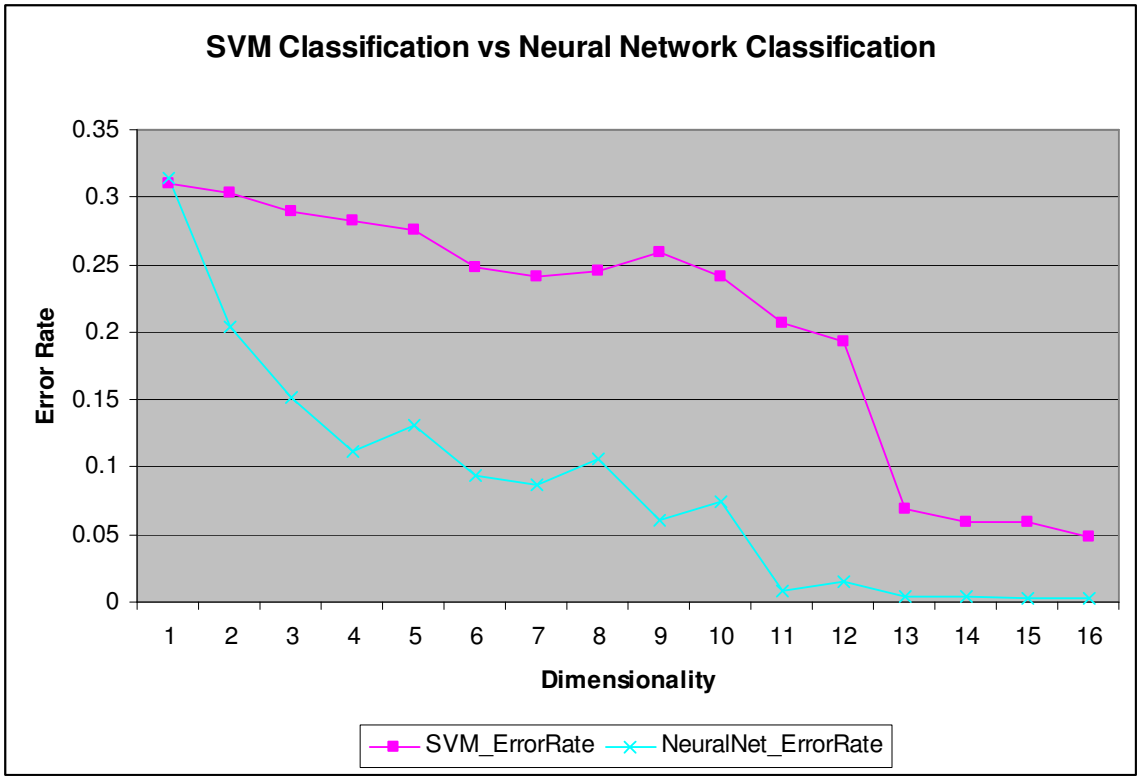


Both training and test error rates decrease with increase in dimensions. This is because of the dataset we have.



Number of support vectors identified for classification decrease slightly with increase in dimensionality.

Experiment 2:
The next experiment is to compare the error rates of SVM with Neural Network classifier. The following graph shows this.



Error rates achieved by Neural Network are lower than SVM because SVM finds a single discriminating hypersurface. The separating hypersurface determined by Neural Network is better than the separating hypersurface determined by SVM, because of having more freedom.

Experiment 3:
The next experiment is to compare the error rate achieved by SVM over the original dataset of problem 1, with error rates achieved by Neural Network and FLD.

	Training error rate	**Test error rate**
Support Vector Machine	0.0153	0.0215
Neural Network classifier	0.0087	0.0286
Fisher Linear Discriminant	0.0200	0.0343

The table shows that over the original dataset of problem 1,
Neural Network > SVM > FLD

Solution 3:

1) Parzen Window Method (PWM)

Experiment 1: The first experiment is to design a parzen window method-based density estimation program. Since, in PWM, we have to choose volume according to the number of samples, $V = \alpha / \sqrt{i}$, I had to select an appropriate value for the constant of proportionality α. The measure of nearness is based on Euclidean

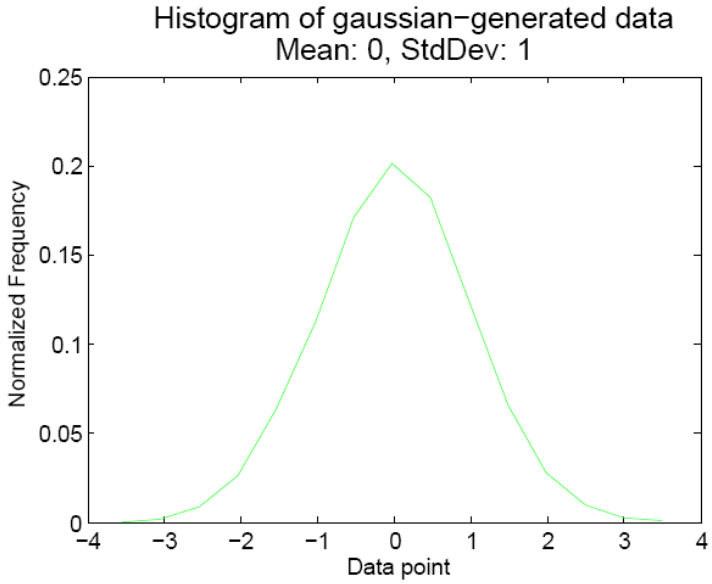
distance, because it seems OK for the dataset I have, and the shape of window is cubic.

I first apply this density estimation over 1D Gaussian. For this problem α=1 was sufficient, because the data was synthetically generated from a known distribution, so we had a good number of samples everywhere.

Code

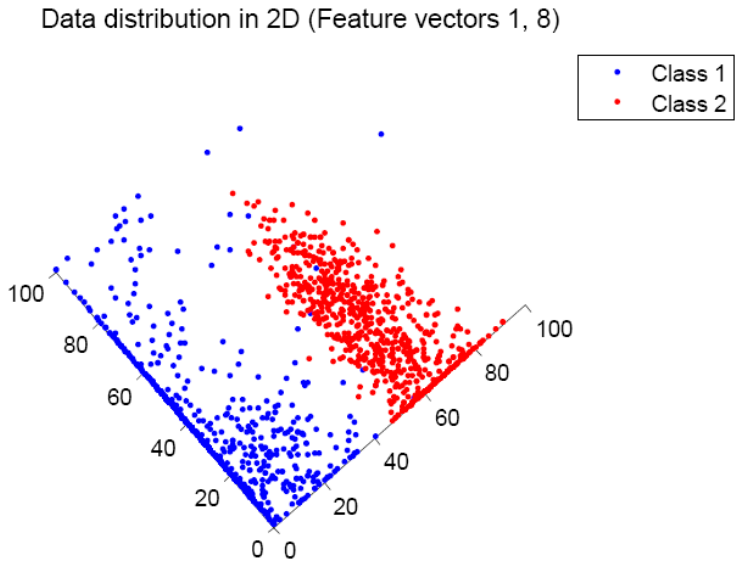
```
x0=[0];
X=[];
Dimension=1;
p_true=1/sqrt(2*pi);
alpha=1;
i=0;
flag=1;
while flag==1
    x=random('norm',0,1,Dimension,1);
    X=[X,x];
    i=i+1;
    V=alpha/sqrt(i);
    h=(V)^(1/Dimension);
    ki=0;
    for j=1:i
        tempflag=1;
        for k=1:Dimension
            if abs(X(k,j))> h/2
                tempflag=0;
            end
            if tempflag==1
                ki=ki+1;
            end
        end
    end
    if mod(i,10000)==0
        if Dimension==1,
            [tempn,temploc]=hist(X,15);
            figure;plot(temploc,tempn/i,'g');
        end
         pix=ki/(i*V)
         p_true
         ki,V
         i
         pause();
    end
    close();
end
```

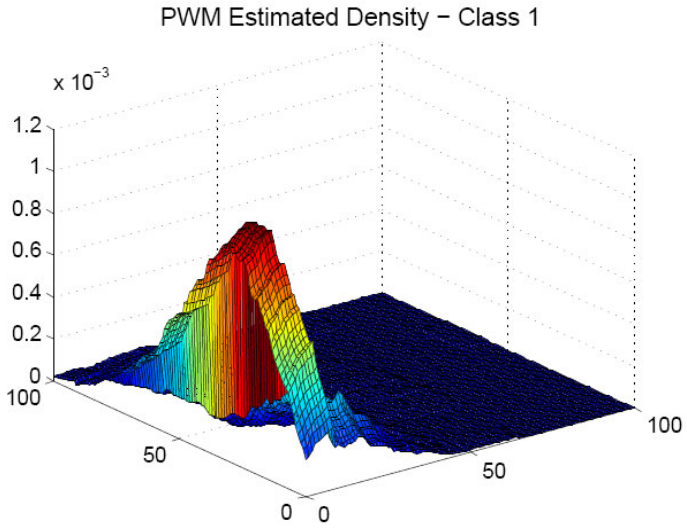
The result is as follows:

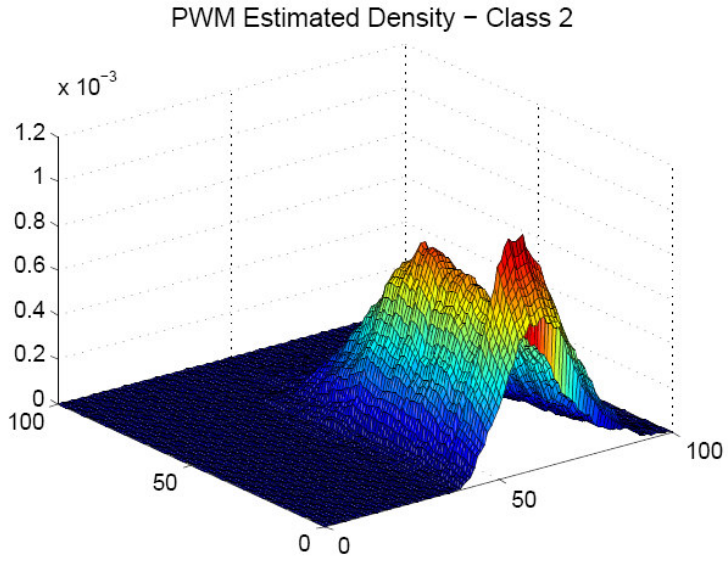


The density was estimated at mean point (x=0)
Value estimated by PWM = **0.4300**
Theoretical value = **0.3989**

Experiment 2:
The next experiment is to determine density over space containing my training and test data. The dataset is the same as in problem 1, which was also used for problem 2. The sample space is of 2 dimensions, representing feature space of feature vectors {1,8}. Because the dataset is real, and not well distributed, I had to take α=10000 (equal to actual volume of entire space) to make sure that most of the time, there are atleast some (>0) samples in the selected region.
I generate density estimates over all points in the space containing training data, as shown in following figures.


Data distribution in 2D (Feature vectors 1, 8)
Class 1
Class 2
100
80
60
40
20
0
0
20
40
60
80
100


PWM Estimated Density − Class 1
x 10^-3
1.2
1
0.8
0.6
0.4
0.2
0
100
50
0
0
50
100



Code for PWM-based density estimation

```
Dimension=2;alpha=10000;
train=uint32(load('train.dat'));
[N_train,temp]=size(train);
I1=find(train(:,4)==0);
I2=find(train(:,4)==3);
Density1=zeros(101,101);
Density2=zeros(101,101);
V=alpha/sqrt(N_train);
h=V^(1/Dimension);
for m=0:100
  for n=0:100
    x=double([m,n]);
    n1=0;n2=0;
    for j=1:N_train
        y=double(train(j,1:Dimension));
        if max(abs(x-y))<h/2
            if train(j,4)==0
                n1=n1+1;
            else n2=n2+1;
            end
        end
    end
    pw1x=n1/(numel(I1)*V);
    pw2x=n2/(numel(I2)*V);
    Density1(m+1,n+1)=pw1x;
    Density2(m+1,n+1)=pw2x;
  end
end
figure;hold on;
plot(train(I1,1),train(I1,2),'b.');
plot(train(I2,1),train(I2,2),'r.');
```

```
hold off;
[X,Y]=meshgrid(0:1:100,0:1:100);
figure;surf(X,Y,Density1);
figure;surf(X,Y,Density2);
```

Experiment 3:

In this step, I actually perform the classification in 2 and 3 dimensions, with feature vectors {1,8} and {1,8,15}.

Again, the constant of proportionality for volume is 10000 for 2D, and 1000000 for 3D.

Since, there are regions in the sample space, where data is very sparse, it is not possible to compare the probabilities of both classes. For the test data points lying in those regions, I declare a doubt instead of performing the classification.

The number of training samples is 1499. (Class 1: 720; Class 2: 719)
The number of test samples is 699. (Class 1: 363; Class 2: 336)

	Training Error Rate	**Test Error Rate**	**Alpha**
2D	16/1499 (0.0107) Errors + 3/1499 Doubts	9/699 (0.0129) Errors + 6/699 Doubts	10000
3D	29/1499 (0.0193) Errors + 0/1499 Doubts	4/699 (0.0057) Errors + 7/699 Doubts	1000000

Code for PWM-Classification

```
Dimension=2;alpha=10000;
train=uint32(load('train.dat'));
test=uint32(load('test.dat'));
[N_train,temp]=size(train);
[N_test,temp]=size(test);
Classified=zeros(N_test,1);
V=alpha/sqrt(N_train);
h=V^(1/Dimension);
err_count=0;
doubt_count=0;
for i=1:N_test
    x=double(test(i,1:Dimension));
    n1=uint32(0);n2=uint32(0);
    for j=1:N_train
        y=double(train(j,1:Dimension));
        if max(abs(x-y))<h/2
            if train(j,4)==0
                n1=n1+1;
            else n2=n2+1;
            end
        end
    end
    if n1+n2>0
      pw1x=n1/(n1+n2);
      pw2x=n2/(n1+n2);
      if pw1x>pw2x
        Classified(i)=1;
        if test(i,4)==3
```

```
            err_count=err_count+1;
        end
      else
        Classified(i)=2;
        if test(i,4)==0
            err_count=err_count+1;
        end
      end
    else
        Classified(i)=0; %Doubt
        doubt_count=doubt_count+1;
    end
end
err_rate=err_count/N_test
doubt_count
doubt_rate=doubt_count/N_test
```

2) K-Nearest Neighbor (KNN)

Experiment 1: In KNN estimation and classification, we are required to increase the volume till it encompasses k points. Because the number of points in the volum is always non-zero, we do not encounter the same problem as we did in PWM. So, the constant of proportionality α in $k = \alpha\sqrt{i}$ in all cases is 1. The measure of nearness is based on Euclidean distance. The first experiment is to estimate density at the mean point in 1D Gaussian.

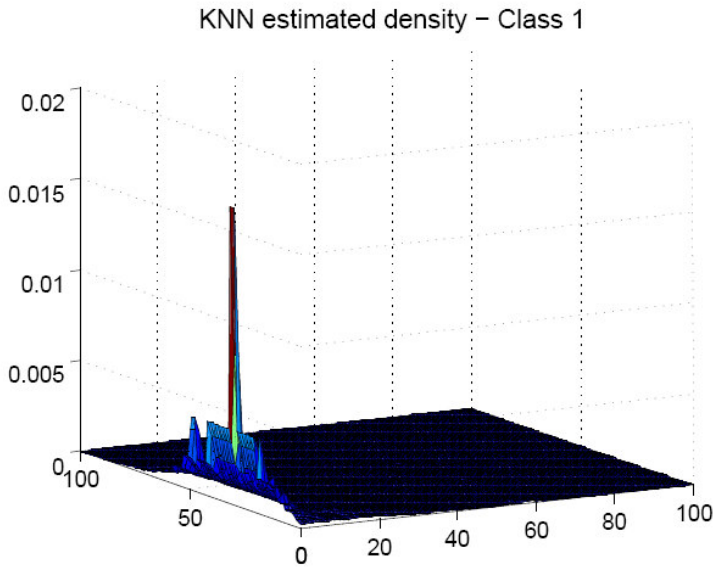
The mean point is x=0.
KNN estimated density = **0.4242**
Theoretical value of density = **0.3989**

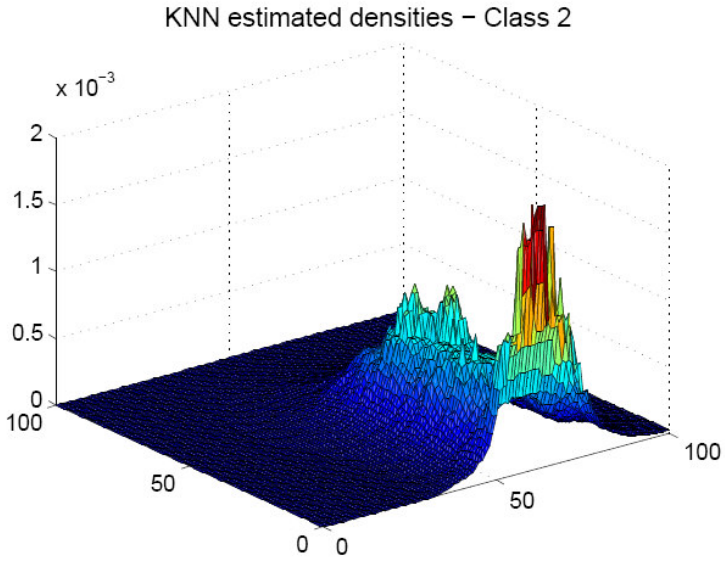
Code for Density Estimation for Gaussian data

```
Dimension=1;
p_true=(1/(2*pi))^(Dimension/2);
X=zeros(Dimension,1);
normX=[0];

alpha=1;
i=1;
flag=1;
while flag==1
    x=random('norm',0,1,Dimension,1);
    normx=norm(x,2);
    if normx<normX(1)
        index=0;
        normX(index+1:i+1)=normX(index:i);
        normX(index)=normx;
        X(:,index+1:i+1)=X(:,index:i);
        X(:,index)=x;
    else if normx>=normX(i)
            index=i;
            normX(i+1)=normx;
            X(:,i+1)=x;
        else
```

```
            index=find(normX>normx,1);
            normX(index+1:i+1)=normX(index:i);
            normX(index)=normx;
            X(:,index+1:i+1)=X(:,index:i);
            X(:,index)=x;
        end
    end
    i=i+1;
    k=uint32(alpha*sqrt(i))+1;
    distance=normX(k);
    if Dimension==2
        V=pi*distance*distance;
    else if Dimension==1
            V=2*distance;
        end
    end

    if mod(i,1000)==0
         if Dimension==1,
             [tempn,temploc]=hist(X,15);
             figure;plot(temploc,tempn/i,'g');
         end
         pix=double(k-1)/(double(i*V));
         p_true,pix
         k,V
         i
         pause();
    end
    close();
end
```

Experiment 2:
As next experiment, I use KNN method to estimate density of the samples of my dataset. The resulting plots are in following figures.





Code for KNN-based Probability Estimation

```
Dimension=2;alpha=1;
train=load('train.dat');
[N_train,temp]=size(train);
Density1=zeros(101,101);
Density2=zeros(101,101);
```

```
I1=find(train(:,4)==0);
I2=find(train(:,4)==3);

err_count=0;
for i1=0:100
  for i2=0:100

    x=[i1,i2];
    k=uint32(alpha*sqrt(N_train))+1;
    normX=[];
    X=[];
    for j=1:N_train
        y=train(j,1:Dimension);
        normx=norm(x-y,2);
        if numel(normX)==0
            normX=[normx];
            X=[train(j,4)];
            else if normx<normX(1)
             normX(2:j)=normX(1:j-1);
             normX(1)=normx;
             X(2:j)=X(1:j-1);
             X(1)=train(j,4);
             else if normx>=normX(j-1)
                normX(j)=normx;
                X(j)=train(j,4);
                 else
                  index=find(normX>normx,1);
                normX(index+1:j)=normX(index:j-1);
                normX(index)=normx;
                X(index+1:j)=X(index:j-1);
                X(index)=train(j,4);
                 end
                end
        end
    end
    distance=normX(k);
    if Dimension==2
        V=pi*distance*distance;
    else if Dimension==1
            V=2*distance;
        end
    end

    n1=0;n2=0;
    for j=1:k
        if X(j)==0
            n1=n1+1;
        else n2=n2+1;
        end
    end
    Density1(i1+1,i2+1)=n1/(numel(I1)*V);
    Density2(i1+1,i2+1)=n2/(numel(I2)*V);
  end
end
[X,Y]=meshgrid(0:1:100,0:1:100);
figure;surf(X,Y,Density1);
figure;surf(X,Y,Density2);
```

Experiment 3:
Here, I use KNN method to classify the test data from same datasets used in PWM. Here because I always able to find k neighbors, unlike PWM, I do not declare a doubt. Instead, I give the sample the class of majority of neighbors.

	Training Error Rate	**Test Error Rate**
2D	23/1499 (0.0153)	18/699 (0.0258)
3D	23/1499 (0.0153)	4/699 (0.0057)

The error rate of KNN is comparable to PWM, for this dataset. But it is much simpler to code KNN, than PWM.

Code for KNN Classification

```
Dimension=3;alpha=1;
load 'train.dat';
load 'test.dat';
[N_train,temp]=size(train);
[N_test,temp]=size(test);
Classified=zeros(N_test,1);


err_count=0;
for i=1:N_test
    x=test(i,1:Dimension);
    k=uint32(alpha*sqrt(N_train))+1;
    normX=[];
    X=[];
    for j=1:N_train
        y=train(j,1:Dimension);
        normx=norm(x-y,2);
        if numel(normX)==0
            normX=[normx];
            X=[train(j,4)];
            else if normx<normX(1)
             normX(2:j)=normX(1:j-1);
             normX(1)=normx;
             X(2:j)=X(1:j-1);
             X(1)=train(j,4);
             else if normx>=normX(j-1)
                normX(j)=normx;
                X(j)=train(j,4);
                 else
                  index=find(normX>normx,1);
                normX(index+1:j)=normX(index:j-1);
                normX(index)=normx;
                X(index+1:j)=X(index:j-1);
                X(index)=train(j,4);
                 end
             end
```

```
    end
    end

    n1=0;n2=0;
    for j=1:k
        if X(j)==0
            n1=n1+1;
        else n2=n2+1;
        end
    end
    pw1x=n1/(n1+n2);
    pw2x=n2/(n1+n2);
    if pw1x>pw2x
        Classified(i)=1;
        if test(i,4)==3
            err_count=err_count+1;
        end
    else
        Classified(i)=2;
        if test(i,4)==0
            err_count=err_count+1;
        end
    end
end
err_rate=err_count/N_test
```

3) Nearest Neighbor (NN)

The training error rate is "Don't care" in this case. So, I have shown only test error rate.
The dataset used is the same as in other sections of this problem.
The measure of nearness is based on Euclidean distance.

	Test Error Rate
2D	17/699 (0.0243)
3D	10/699 (0.0143)

Code for Nearest Neighbor Classification

```
Dimension=3;
load 'train.dat';
load 'test.dat';
[N_train,temp]=size(train);
[N_test,temp]=size(test);
Classified=zeros(N_test,1);

err_count=0;
for i=1:N_test
    x=test(i,1:Dimension);
    y=train(1,1:Dimension);
    mindist=norm(x-y,2);
```

```
    closest=1;
    for j=2:N_train
        y=train(j,1:Dimension);
        normx=norm(x-y,2);  %Euclidean distance
        if normx<mindist
            mindist=normx;
            closest=j;
        end
    end
    if train(closest,4)==0
        Classified(i)=1;
        if test(i,4)==3
            err_count=err_count+1;
        end
    else
        Classified(i)=2;
        if test(i,4)==0
            err_count=err_count+1;
        end
    end
end
err_rate=err_count/N_test
```