Ahmet Narman, ahmet.narman18@imperial.ac.uk, CID: 01578741, MSc. HBR

Machine Learning and Neural Computation: Coursework 2

All figures and the Matlab code for this assignment is given in the corresponding Appendix.

1)

Before starting with the classifier, the data was visualised in different plotting methods to better understand what we are dealing with and tailor our classifier to the data accordingly. A boxplot was created to compare how the data is distributed on every dimension for different classes (Appx. A, 1). It is visible that the data of different classes overlap for some dimensions and doesn't overlap for others. For example, on the last four dimensions, data belonging to five classes mostly overlap, which may make it harder to extract relevant class information from these dimensions (see Appx. C). Though this boxplot figure gives an idea of where the distribution of each class sits compared to each other, it does not give much of an information about how the distributions look like. For that, the data for each class was plotted on histogram figures (Appx. A, 2.1-2.5). Moreover, the data was plotted for 5 classes using scatter plot for two chosen dimensions (Appx. A, 3.1-3.3) and for some dimensions there are separated clusters for each class while for others, the class clusters mostly overlap (Appx, A, 3.1).

One classification method we could use is the Naïve Bayes approach. However, after analysing the figures, it was seen that most of the dimensions does not have a Gaussian distribution for different classes so representing the classes with different multivariate Gaussian distributions will be somewhat wrong. We could try to use the actual probability distributions on Naïve Bayes based on the histogram data we have but coming up with a multivariate distribution with the derived distributions doesn't have a straightforward way (we can't just multiply the actual distributions for each dimension as they are not independent, covariances can be found but for non-gaussian distributions, it is hard to utilize them). Yet, since the Naïve Bayes classifier (using Gaussian) is computationally efficient, easy to code and understand, it will be used as a base classifier according to which the actual classifier of our choice will be evaluated.

The main classifier of choice will be k-nearest neighbour classifier as there are some clear clusters on certain dimensions of the data for different classes. Using KNN, we can exploit this clustering and come up with an efficient and accurate classification pipeline. For this classifier we should determine the optimal number of neighbours and we should consider how to pre-process the data (normalization can be a good approach as the distribution and the range of the data is different for different dimensions which may interfere with the accuracy of the classifier). Also, distance weighted KNN will be used to see if the accuracy will be better.

2)

a. Naïve Bayes Classifier

For this classifier, we should represent the distribution of data for different classes as a multivariate gaussian distribution whose formula is given below:

$$P(\boldsymbol{x}|C_k) = a \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu_k})^T \sum_k^{-1} (\boldsymbol{x}-\boldsymbol{\mu_k})\right)$$

where $\boldsymbol{\mu_k}$ is the mean vector of class k data across dimensions and $\sum_k$ is the covariance matrix of class k for all dimensions. After finding that, we can assign the class of the input according to the maximum likelihood principle given below:

$$Class(\boldsymbol{x}) = \underset{k}{\operatorname{argmax}}\big(P(C_k|\boldsymbol{x})\big) = \underset{k}{\operatorname{argmax}}\left(\frac{P(\boldsymbol{x}|C_k)P(C_k)}{P(\boldsymbol{x})}\right)$$

The probability P(**x**) is irrelevant for this context because the input is chosen without a bias.

This classification method was implemented by finding the mean vector and the covariance matrix of the training data for different classes. Then, using these information, the likelihood of the test input was calculated for each class in ClassifyX.m script, and the class that had maximum likelihood was assigned as the class of the test input (see Appx. B, 1.1-1.2).

b. k-Nearest Neighbour Classifier

In the TrainClassifierX.m code, the training data was taken as one of our parameters. The normalization parameters (mean and standart deviation to do z-scale normalization) were found according to the training input as well which will be used in classification. The other parameter that is needed is the value 'k' which will be chosen after calculating the optimal amount of neighbours for the given problem (for this, a modified TrainClassifierX method will be used in the validation code Appx. B, 2.3). In the ClassifyX.m script, the distance between every individual testing data and every individual training data was found. Then the k closest element to the each testing point was found by finding the element with the minimum distance and then doing this k times and recording the classes of these neighbours. Additionally, the distances of the closest neighbours are saved as well in order to do distance weighted KNN which favours the neighbours that are closer using the following equations:

$$W(k) = \sum_{i=1}^{N_i} \frac{1}{\left(\boldsymbol{x} - \boldsymbol{t_{k,i}}\right)^2} \quad , \quad k \in \{1,2,3,4,5\}$$

where N_i is the amount of data points in training set that belongs to class 'i', and $\boldsymbol{t_{i,k}}$ is the training points of class 'i'. This equation ensures that the shorter the distance, the more effect it has on the weighted sum. Knowing that, the classification is done using the following:

$$Class(\boldsymbol{x}) = \underset{k}{\operatorname{argmax}}\, W(k)$$

The two classifiers were implemented, and they passed the SanityCheck.

3)

For the Naïve Bayes Classifier, the data was simply separated into training and testing groups to see how well the classifier performs in terms of accuracy and speed. The average classification accuracy (for 10 trials) of Naïve Bayes was found to be 97.06% with standard deviation 0.304, for 19200 training data and 4800 test data(4:1 ratio, the ratio of training and testing data matters because the more training data there is, the better the system learns, especially for KNN classifier) and on average, one training and classification sequence took

less than 5 seconds for 4800 testing points (though this depends heavily on the computers processing speed). This performance will be compared to the KNN classifier's performance.

For KNN classifier, the given data was separated into two groups which are training group and testing group. Moreover, the training data was partitioned to do n-fold cross-validation so every part of the training data is used for testing as well as training without the data overlapping. Three versions of KNN were tested, one that doesn't include pre-processing (no normalization), one where the data is normalized according to the training input distribution, and one where we utilize distance weighted KNN with normalization. The validation results (Appx. A, 4.1) shows us the optimal performance is achieved when k=4, using the distance weighted KNN method, but the testing performance (Appx. A, 4.2) says that the best performance is for k=1 using normalised regular KNN. We will use k=4, weighted KNN model, as in real life, the test data is not given to us and we can just use the validation results for optimizing the model, and also, using n-fold validation, we eliminated some randomness by averaging, which made validation results less variable (as it can be seen on the plots Appx. A, 4.1-4.2, validation plots show less abrupt changes because they are the average of n trials).

Now we can examine the specified KNN model better. The same testing procedure that was used on Naïve Bayes method gave an average classification accuracy of 98.62% with standard deviation 0.201 for 19200 training data and 4800 testing data, which means it works 1.56% better with less variability. The time it takes for the KNN algorithm to train and classify (though the training is virtually immediate) is around 60 seconds for 4800 testing points, which is considerably higher compared to the Naïve Bayes method. The confusion matrix given below shows us which classes are confused with others during classification.

	Actual Class					
Predicted Class	%	Class1	Class2	Class3	Class4	Class5
	Class1	99.91	0	0	0	0
	Class2	0	99.36	0.18	0	0.18
	Class3	0.09	0.64	99.82	1.34	2.37
	Class4	0	0	0	97.07	4.55
	Class5	0	0	0	1.59	92.90

It was shown that the KNN performs well for classes 1,2,3 and most of the error happens for classes 4, 5. That's because data of these classes spread a lot in most dimensions (see Appx. A, 1-3.2-3.3) which causes some data from these classes being neighbours to other classes.

The great success of the KNN method is its performance despite its simplicity. It performed very decently with classification error less than 1.5%, most of which can be explained by the outlier elements in class 4 and 5. The disadvantage of this method is the fact that it takes a lot of time to classify the given data which may be problematic in some applications. For applications requiring fast classification, Naïve Bayes model (or multi-layer perceptron model which uses a lot of time in training but gives fast results in classifying) can be used.

References:

[1] Hechenbichler, Schliep. *Weighted k-Nearest-Neighbor Techniques and Ordinal Classification*, 2004. Sonderforschungsbereich 386

[2] Faisal, Aldo. *Machine learning & Neural Computation Lecture Notes.* 2018

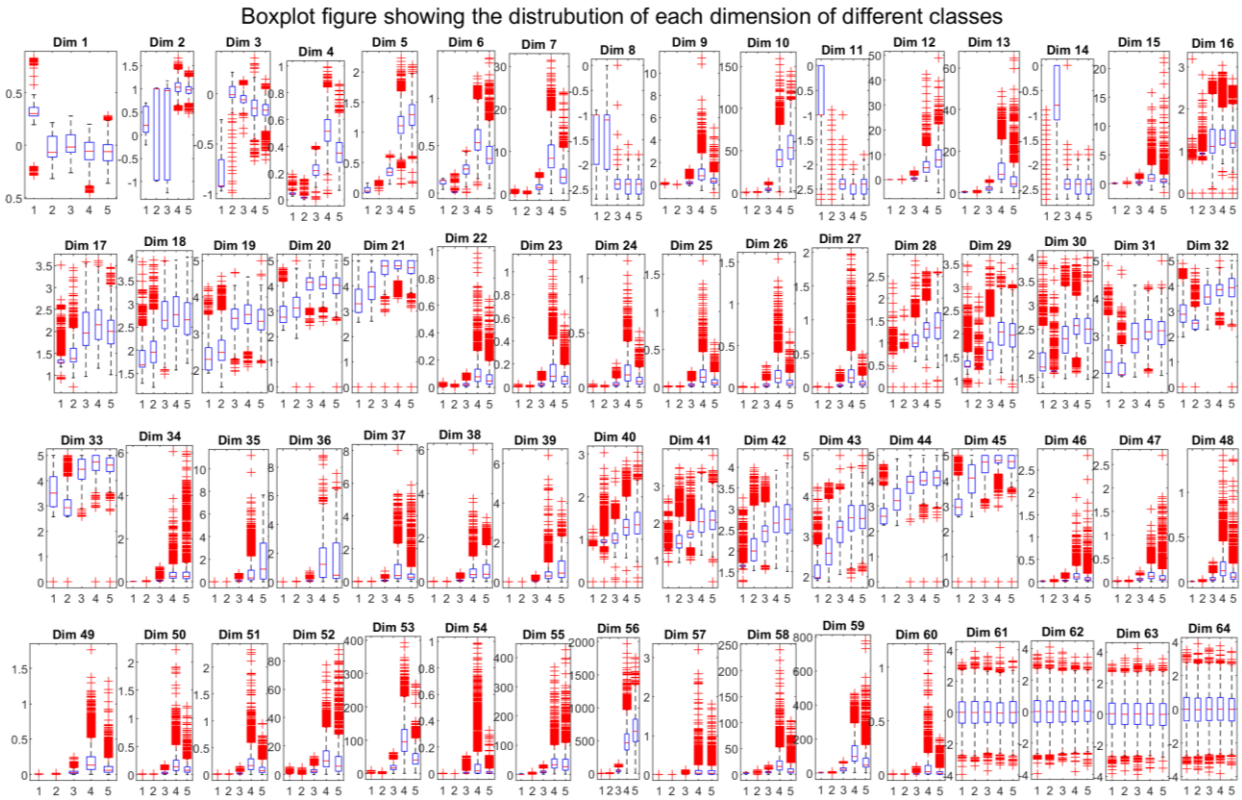
[3] https://en.wikipedia.org/wiki/Naive_Bayes_classifier

[4] https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

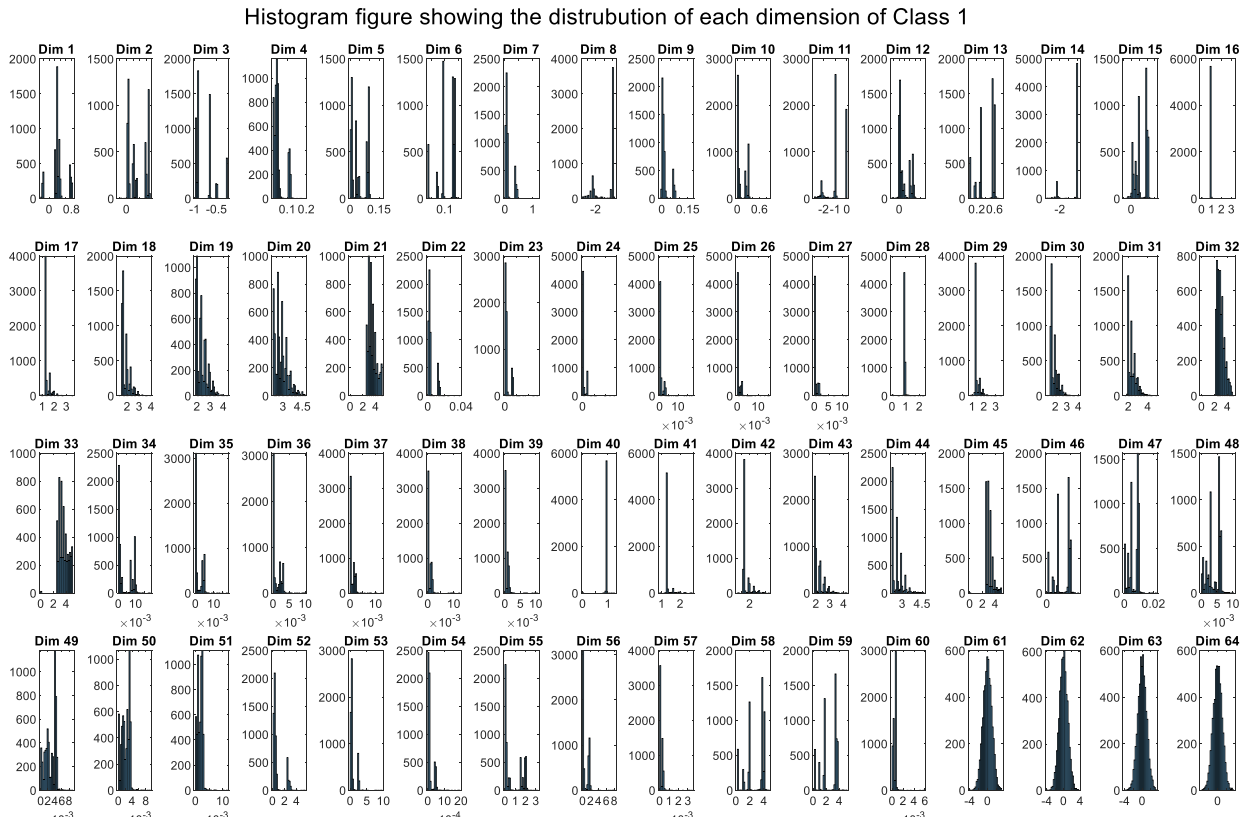
[5] https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/#parameter-tuning-with-cross-validation

APPENDIX A: Plots

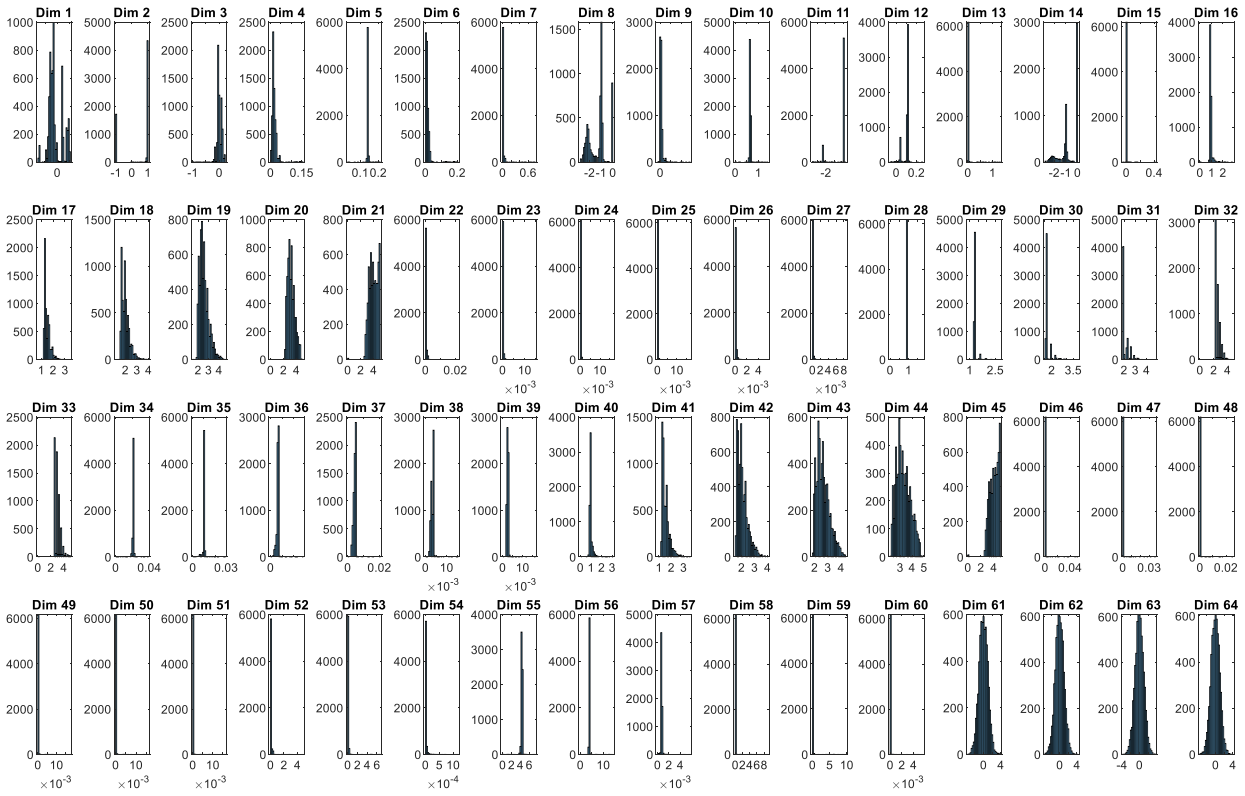
1)



2.1)

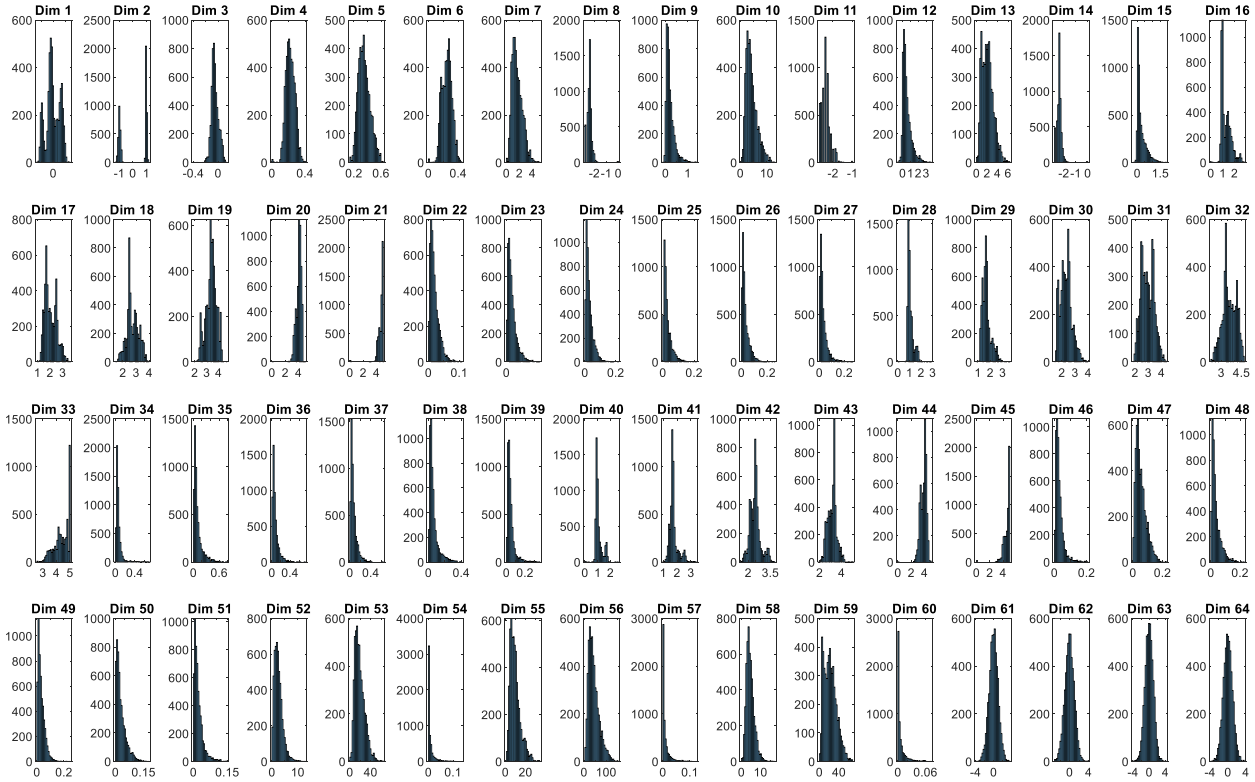

Histogram figure showing the distrubution of each dimension of Class 1
Dim 1
Dim 2
Dim 3
Dim 4
Dim 5
Dim 6
Dim 7
Dim 8
Dim 9
Dim 10
Dim 11
Dim 12
Dim 13
Dim 14
Dim 15
Dim 16
Dim 17
Dim 18
Dim 19
Dim 20
Dim 21
Dim 22
Dim 23
Dim 24
Dim 25
Dim 26
Dim 27
Dim 28
Dim 29
Dim 30
Dim 31
Dim 32
Dim 33
Dim 34
Dim 35
Dim 36
Dim 37
Dim 38
Dim 39
Dim 40
Dim 41
Dim 42
Dim 43
Dim 44
Dim 45
Dim 46
Dim 47
Dim 48
Dim 49
Dim 50
Dim 51
Dim 52
Dim 53
Dim 54
Dim 55
Dim 56
Dim 57
Dim 58
Dim 59
Dim 60
Dim 61
Dim 62
Dim 63
Dim 64

2.2)


Histogram figure showing the distrubution of each dimension of Class 2
Dim 1
Dim 2
Dim 3
Dim 4
Dim 5
Dim 6
Dim 7
Dim 8
Dim 9
Dim 10
Dim 11
Dim 12
Dim 13
Dim 14
Dim 15
Dim 16
Dim 17
Dim 18
Dim 19
Dim 20
Dim 21
Dim 22
Dim 23
Dim 24
Dim 25
Dim 26
Dim 27
Dim 28
Dim 29
Dim 30
Dim 31
Dim 32
Dim 33
Dim 34
Dim 35
Dim 36
Dim 37
Dim 38
Dim 39
Dim 40
Dim 41
Dim 42
Dim 43
Dim 44
Dim 45
Dim 46
Dim 47
Dim 48
Dim 49
Dim 50
Dim 51
Dim 52
Dim 53
Dim 54
Dim 55
Dim 56
Dim 57
Dim 58
Dim 59
Dim 60
Dim 61
Dim 62
Dim 63
Dim 64

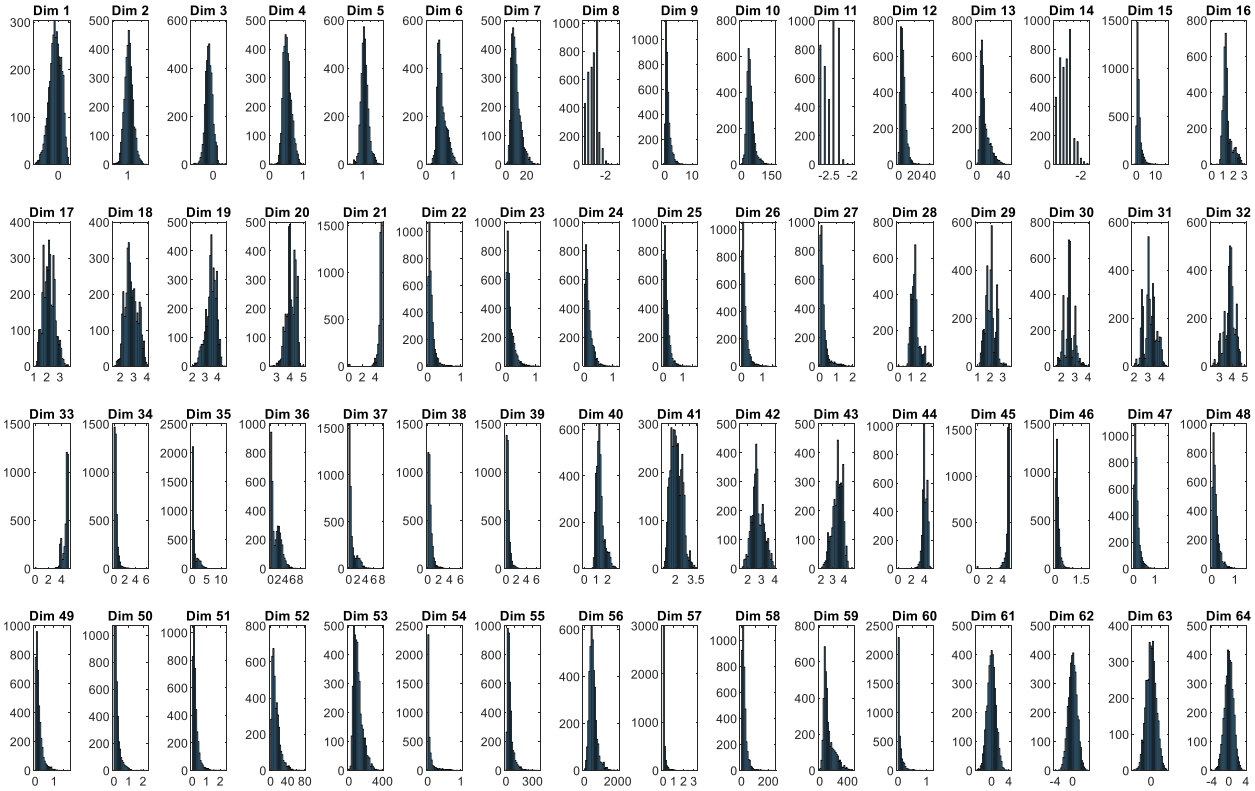
2.3)

Histogram figure showing the distrubution of each dimension of Class 3

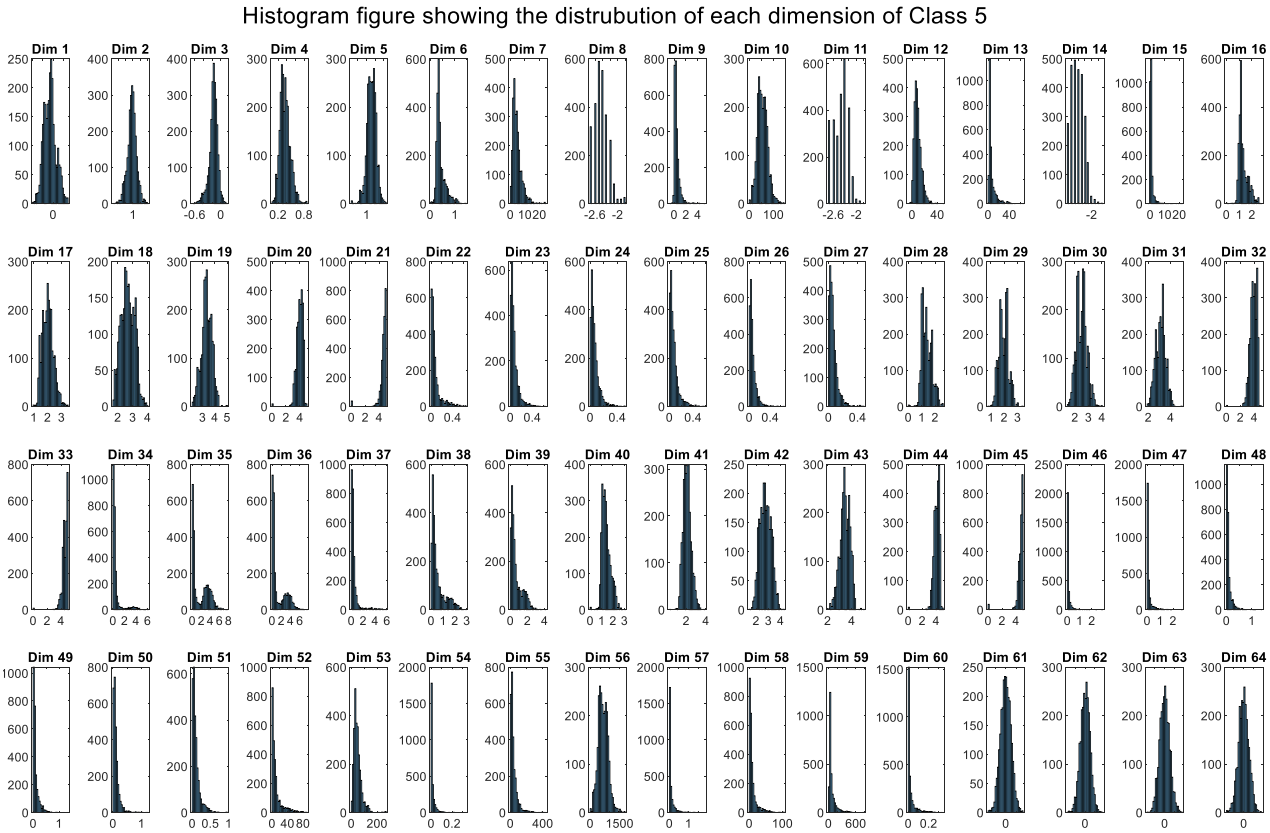


2.4)

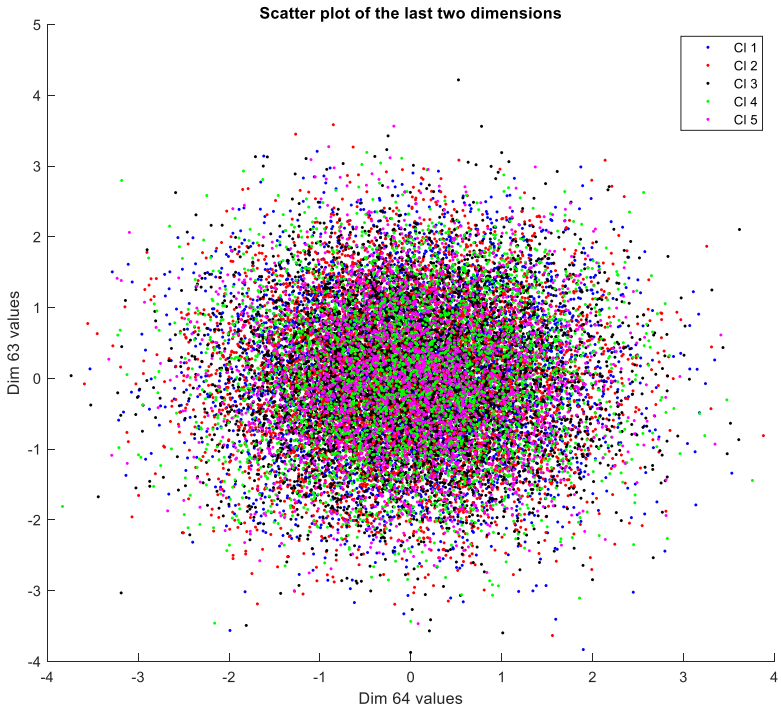
Histogram figure showing the distrubution of each dimension of Class 4



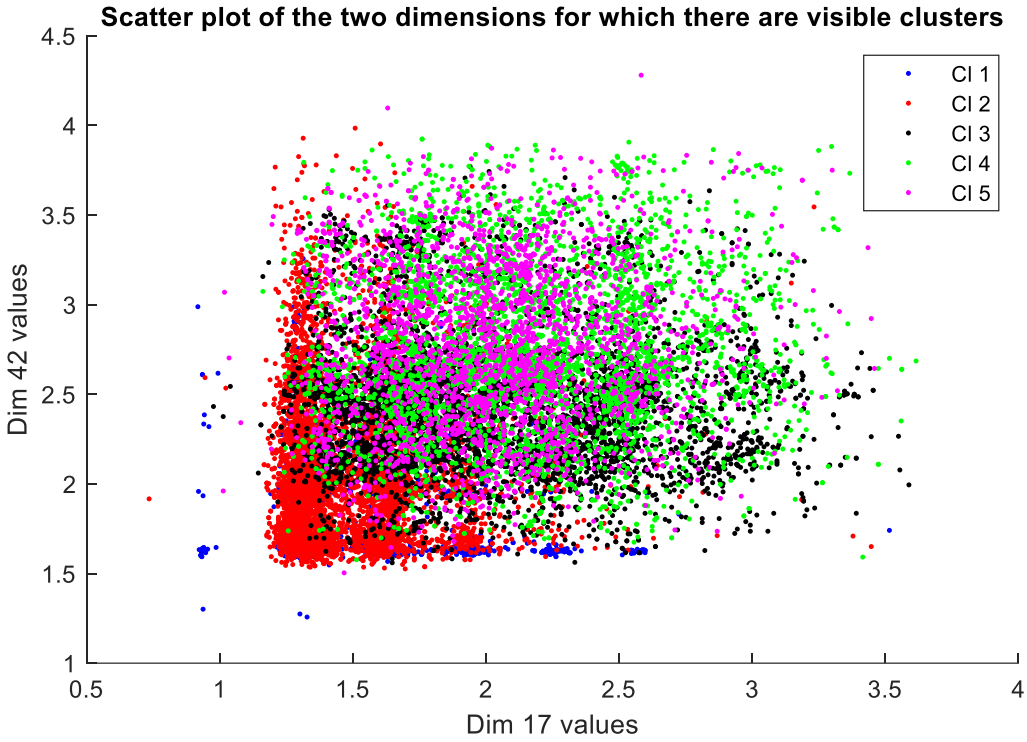
2.5)


Histogram figure showing the distrubution of each dimension of Class 5
Dim 1
Dim 2
Dim 3
Dim 4
Dim 5
Dim 6
Dim 7
Dim 8
Dim 9
Dim 10
Dim 11
Dim 12
Dim 13
Dim 14
Dim 15
Dim 16
Dim 17
Dim 18
Dim 19
Dim 20
Dim 21
Dim 22
Dim 23
Dim 24
Dim 25
Dim 26
Dim 27
Dim 28
Dim 29
Dim 30
Dim 31
Dim 32
Dim 33
Dim 34
Dim 35
Dim 36
Dim 37
Dim 38
Dim 39
Dim 40
Dim 41
Dim 42
Dim 43
Dim 44
Dim 45
Dim 46
Dim 47
Dim 48
Dim 49
Dim 50
Dim 51
Dim 52
Dim 53
Dim 54
Dim 55
Dim 56
Dim 57
Dim 58
Dim 59
Dim 60
Dim 61
Dim 62
Dim 63
Dim 64

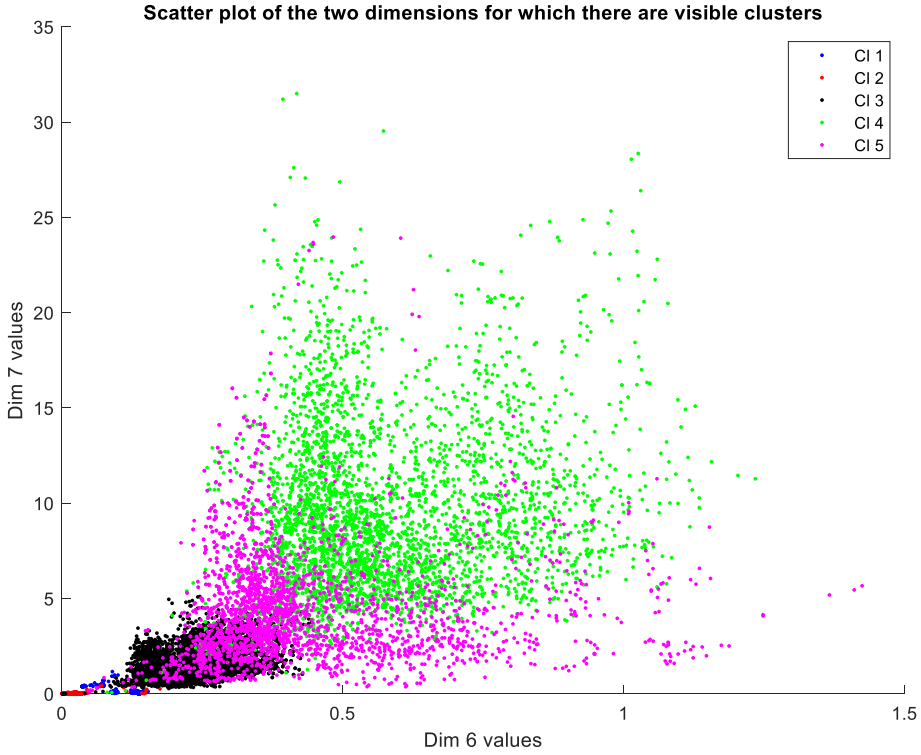
3.1)


Scatter plot of the last two dimensions
Cl 1
Cl 2
Cl 3
Cl 4
Cl 5
Dim 63 values
Dim 64 values

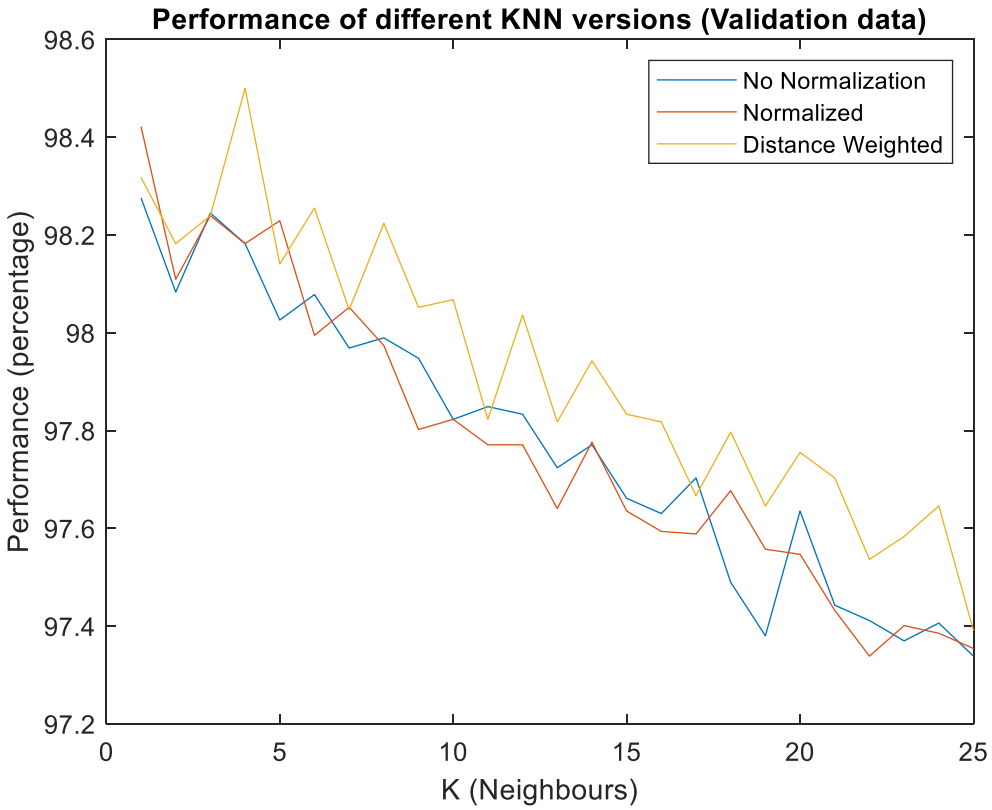
3.2)


Scatter plot of the two dimensions for which there are visible clusters
Cl 1
Cl 2
Cl 3
Cl 4
Cl 5
Dim 42 values
Dim 17 values
4.5
4
3.5
3
2.5
2
1.5
1
0.5
1
1.5
2
2.5
3
3.5
4

3.3)


Scatter plot of the two dimensions for which there are visible clusters
Cl 1
Cl 2
Cl 3
Cl 4
Cl 5
Dim 7 values
Dim 6 values
35
30
25
20
15
10
5
0
0
0.5
1
1.5

4.1)


Performance of different KNN versions (Validation data)
No Normalization
Normalized
Distance Weighted
Performance (percentage)
K (Neighbours)
98.6
98.4
98.2
98
97.8
97.6
97.4
97.2
0
5
10
15
20
25

4.2)


Performance of different KNN versions (Testing data)
No Normalization
Normalized
Distance Weighted
Performance (percentage)
K (Neighbours)
98.8
98.6
98.4
98.2
98
97.8
97.6
97.4
97.2
97
0
5
10
15
20
25

APPENDIX B: Matlab Codes

1.1) Naïve Bayes TrainClassifierX.m code

```
%%% Ahmet Narman,
%%% ahmet.narman18@imperial.ac.uk,
%%% CID: 01578741,
%%% MSc. HBR
%%% Imperial College London

function P = TrainClassifierX(input, label)

cl1 = input(find(label==1),:); % All data belonging to class 1
cl2 = input(find(label==2),:); % All data belonging to class 2
cl3 = input(find(label==3),:); % All data belonging to class 3
cl4 = input(find(label==4),:); % All data belonging to class 4
cl5 = input(find(label==5),:); % All data belonging to class 5

[trainSize, trainDim] = size(input); % Size of the training sample

P = struct; % All the parameters, which is the output of the training func.

P.size.all = trainSize;  % size of the training data
P.size.cl1 = length(cl1); % Size of the sample that belongs to class 1
P.size.cl2 = length(cl2); % Size of the sample that belongs to class 2
P.size.cl3 = length(cl3); % Size of the sample that belongs to class 3
P.size.cl4 = length(cl4); % Size of the sample that belongs to class 4
P.size.cl5 = length(cl5); % Size of the sample that belongs to class 5

P.mean.all = mean(input); % Means of the training data dimensions
P.mean.cl1 = mean(cl1); % Means of the sample that belongs to class 1
P.mean.cl2 = mean(cl2); % Means of the sample that belongs to class 2
P.mean.cl3 = mean(cl3); % Means of the sample that belongs to class 3
P.mean.cl4 = mean(cl4); % Means of the sample that belongs to class 4
P.mean.cl5 = mean(cl5); % Means of the sample that belongs to class 5

P.cov.all = cov(input); % Covariance betwen the training data dimensions
P.cov.cl1 = cov(cl1); % Covariance of the sample that belongs to class 1
P.cov.cl2 = cov(cl2); % Covariance of the sample that belongs to class 2
P.cov.cl3 = cov(cl3); % Covariance of the sample that belongs to class 3
P.cov.cl4 = cov(cl4); % Covariance of the sample that belongs to class 4
P.cov.cl5 = cov(cl5); % Covariance of the sample that belongs to class 5

end
```

1.2) Naïve Bayes Classify.m code

```
%%% Ahmet Narman,
%%% ahmet.narman18@imperial.ac.uk,
%%% CID: 01578741,
%%% MSc. HBR
%%% Imperial College London

function label = ClassifyX(input, P)

[testSize, testDim] = size(input); % Size of the testing sample

prob = zeros(testSize, 5); % Probability of datapoints being in each one of
                           % the five classes

% Prior class probabilities found using class sizes in training sample
PC1 = P.size.cl1/P.size.all;
PC2 = P.size.cl2/P.size.all;
```

```
PC3 = P.size.cl3/P.size.all;
PC4 = P.size.cl4/P.size.all;
PC5 = P.size.cl5/P.size.all;

for i = 1:testSize
    % The class probabilities are found under the assumption of gaussian
    % distrubution. Thus, for every class, a probability was found using a
    % jointly gaussian distrubution. The normalization constant is omited
    % because it wont affect the final comparison
    prob(i,1) = exp((-1/2)*(input(i,:)-P.mean.cl1)...
        *inv(P.cov.cl1)*(input(i,:)-P.mean.cl1)')*PC1;
    prob(i,2) = exp((-1/2)*(input(i,:)-P.mean.cl2)...
        *inv(P.cov.cl2)*(input(i,:)-P.mean.cl2)')*PC2;
    prob(i,3) = exp((-1/2)*(input(i,:)-P.mean.cl3)...
        *inv(P.cov.cl3)*(input(i,:)-P.mean.cl3)')*PC3;
    prob(i,4) = exp((-1/2)*(input(i,:)-P.mean.cl4)...
        *inv(P.cov.cl4)*(input(i,:)-P.mean.cl4)')*PC4;
    prob(i,5) = exp((-1/2)*(input(i,:)-P.mean.cl5)...
        *inv(P.cov.cl5)*(input(i,:)-P.mean.cl5)')*PC5;
end

logProb = log10(prob'); % Probabilities converted to log scale

[MaxVal, label] = max(logProb); % Index of the max probability is the
                                % assigned label for the corresponding data

end
```

1.3) Naïve Bayes Validation code

```
%%% Ahmet Narman,
%%% ahmet.narman18@imperial.ac.uk,
%%% CID: 01578741,
%%% MSc. HBR
%%% Imperial College London

close all;
clear all;

load('data.mat');

[dataSize,dataDim] = size(data); % Size of the dataset

t = 10; % How many times the classification will repeat

corr = zeros(1,t); % Correctness for t iterations

for i=1:t
    % We repeat the classification here to find an average performance and
    % eliminate variability
    tic % To see how much time it takes for training-classification

    random = data(randperm(dataSize),:); % The dataset is shuffled
    trainSize = 0.8*dataSize; % The amount of data to be used for training

    % The dataset is split into training and testing groups
    train_data = random(1:trainSize, 2:end);
    train_label = random(1:trainSize, 1);
    test_data = random(trainSize+1:end, 2:end);
    test_label = random(trainSize+1:end, 1);

    % The Classifier is trained
    param = TrainClassifierX(train_data, train_label);
    % The classifier is tested
    label_out = ClassifyX(test_data, param);
```

```
    % The correct classifications are found
    Correctness = label_out' == test_label;
    corr(i)=sum(Correctness);
    toc
end

% Performance is calculated for every iteration in terms of (%)
prf = corr*100/(dataSize-trainSize);
% Performance average and its standard deviation was found
avgPerf = mean(prf);
stdPerf = std(prf);
```

2.1) k-Nearest Neighbour TrainClassifierX.m code

```
%%% Ahmet Narman,
%%% ahmet.narman18@imperial.ac.uk,
%%% CID: 01578741,
%%% MSc. HBR
%%% Imperial College London

function P = TrainClassifierX(input, label)

P.mean = mean(input); % Mean of training samle for normalization
P.std = std(input); % Std of training sample for normalization

allData = [label input]; % Labels combined with the inputs

P.data = allData; % All the input and the labels are taken as parameters

P.k = 4; % The optimal 'k' for distance weighted KNN

end
```

2.2) k-Nearest Neighbour ClassifyX.m code

```
%%% Ahmet Narman,
%%% ahmet.narman18@imperial.ac.uk,
%%% CID: 01578741,
%%% MSc. HBR
%%% Imperial College London

function label = ClassifyX(input, P)

% Training data and labels combined, training data normalized
train = [P.data(:,1) (P.data(:,2:65)-P.mean)./P.std];

[trainSize, trainDim] = size(train); % Training sample size


test = (input-P.mean)./P.std; % Normalized testing data
[testSize,testDim] = size(test); % Testing sample size

% Closest neighbours will be put in this matrix
closestNeighbours = zeros(P.k, testSize);
% Neighbour distances will be put in this array
NeighDist = zeros(1,P.k);
% Distance weighted sums will be put in this matrix for every class
classDistSum = zeros(testSize,5);

for i = 1:testSize
    % Below parameter is the Euclidian distances of training data
    % their corresponding classes are included as well.
    dist=[sqrt(sum((train(:,2:65) - test(i,:)).^2'))' train(:,1)];
```

```
    % The minimum distance was found 'k' times in this loop
    for j = 1:P.k
        [M,I] = min(dist(:,1)); % Finding the neighbour with min distance
        closestNeighbours(j,i) = dist(I,2); % Adding to the neighbour list
        dist(I,1)=max(dist(:,1)); % Increasing that value to find the
                                  % next minimum distance in the dist var.
        NeighDist(1,j) = M; % Neighbour distance recorded
    end
    % Weighted sum for each class was found where weights are inverse
    % squared distances. Lower the distance, higer the weight
    classDistSum(i,1) = sum(1./NeighDist(closestNeighbours(:,i)==1));
    classDistSum(i,2) = sum(1./NeighDist(closestNeighbours(:,i)==2));
    classDistSum(i,3) = sum(1./NeighDist(closestNeighbours(:,i)==3));
    classDistSum(i,4) = sum(1./NeighDist(closestNeighbours(:,i)==4));
    classDistSum(i,5) = sum(1./NeighDist(closestNeighbours(:,i)==5));
end

[M,label] = max(classDistSum'); % Assigning the class label according to
                                %  argmax rule
end
```

2.3) k-Nearest Neighbour Method comparison code

Note: This code will not work with the given TrainClassifyX methods

```
%%% Ahmet Narman,
%%% ahmet.narman18@imperial.ac.uk,
%%% CID: 01578741,
%%% MSc. HBR
%%% Imperial College London

close all;
clear all;

load('data.mat');

[dataSize,dataDim] = size(data); % Size of the dataset
trainSize = 0.8*dataSize; % Training data size
testSize = dataSize - trainSize; % Testing data size

kmax = 25; % The 'k' value until which the performance will be calculated
n = 5; % for n-fold cross validation

perform = zeros(kmax,n); % To record  accuracy for validation
testPerform = zeros(kmax,1); % to record accuracy for testing

for k = 1:kmax

    randomData = data(randperm(size(data,1)),:); % The dataset is shuffled

    % Data is partitioned into training and testing
    train_data = randomData(1:trainSize, 2:end);
    train_label = randomData(1:trainSize, 1);
    test_data = randomData(trainSize+1:end, 2:end);
    test_label = randomData(trainSize+1:end, 1);

    for i=1:n
        tic % To calculate the time for classification
        % The training data is further partitioned into validation and
        % training sets
        validate = train_data(1:trainSize/n,:); % Validation set
        valLab = train_label(1:trainSize/n,:); % Validation labels
        train = train_data((trainSize/n)+1:end,:); % Training set
```

```
        trainLab = train_label((trainSize/n)+1:end,:); % Training labels

        % TrainClassifierX function is modified here to accept 'k' as an
        % argument because we want to see the effect of 'k' on training
        param = TrainClassifierX(train, trainLab, k);
        label_out = ClassifyX(validate, param);

        Corr = label_out' == valLab;% To find correctly classified elements

        perform(k,i) = sum(Corr)*100/(trainSize/n);% Percentage correctness

        % Shifting the datasets to do the n-fold valdation
        train_data = circshift(train_data, trainSize/n, 1);
        train_label = circshift(train_label, trainSize/n, 1);
        toc % To calculate the time for classification
    end

    % After validation, now we move to testing using training and testing
    % datasets
    param = TrainClassifierX(train_data, train_label, k);
    label_out = DClassifyX(test_data, param);
    Corr = label_out' == test_label;% To find correctly classified elements
    testPerform(k) = sum(Corr)*100/testSize; % Percentage correctness
    k % To see which part of the loop we ar in
end

perfMean = mean(perform'); % Using mean for average performances

figure
plot(1:kmax, perfMean); % Validation results plotted
hold on;
plot(1:kmax, testPerform'); % Testing results plotted
xlabel('k (neighbours)');
ylabel('Classification Performance (Percentage)');
title('Validation and test results for KNN classifier');
legend('validation', 'test');
```

To work this code, use the following Train ClassifierX code:

```
%%% Ahmet Narman,
%%% ahmet.narman18@imperial.ac.uk,
%%% CID: 01578741,
%%% MSc. HBR
%%% Imperial College London

function P = TrainClassifierX(input, label,k)

P.mean = mean(input); % Mean of training samle for normalization
P.std = std(input); % Std of training sample for normalization

allData = [label input]; % Labels combined with the inputs

P.data = allData; % All the input and the labels are taken as parameters

P.k = k; % The optimal 'k' for distance weighted KNN

end
```

2.4) k-Nearest Neighbour Validation code

```
%%% Ahmet Narman,
%%% ahmet.narman18@imperial.ac.uk,
%%% CID: 01578741,
```

```
%%% MSc. HBR
%%% Imperial College London

close all;
clear all;

load('data.mat');

[dataSize,dataDim] = size(data); % Size of the dataset
trainSize = 0.8*dataSize; % Training data size
testSize = dataSize - trainSize; % Testing data size

t=10; % Number or repetition for training-testing

perf = zeros(1,t); % Testing performance to be stored here

for i=1:t
    tic % To keep track of time

    randomData = data(randperm(size(data,1)),:); % The dataset is shuffled

    train_data = randomData(1:trainSize, 2:end);
    train_label = randomData(1:trainSize, 1);
    test_data = randomData(trainSize+1:end, 2:end);
    test_label = randomData(trainSize+1:end, 1);

    % Training and classifying
    param = TrainClassifierX(train_data, train_label);
    label_out = ClassifyX(test_data, param);

    Corr = label_out' == test_label;% To find correctly classified elements
    perf(i) = sum(Corr)*100/testSize; % Performance in percentage
    toc % To find the time for training and classifying
end

% Average and std values of the performance
avgPerf = mean(perf);
stdPerf = std(perf);

% To calculate the confusion matrix
cl1Err = zeros(1,5);
cl2Err = zeros(1,5);
cl3Err = zeros(1,5);
cl4Err = zeros(1,5);
cl5Err = zeros(1,5);
%Confusion matrix elements for the last train-test pair is calculated
for i=1:5
    cl1Err(i)=sum(label_out(test_label==1)'==i)*100/sum(test_label==1);
    cl2Err(i)=sum(label_out(test_label==2)'==i)*100/sum(test_label==2);
    cl3Err(i)=sum(label_out(test_label==3)'==i)*100/sum(test_label==3);
    cl4Err(i)=sum(label_out(test_label==4)'==i)*100/sum(test_label==4);
    cl5Err(i)=sum(label_out(test_label==5)'==i)*100/sum(test_label==5);
end
```

3.1) Plot Code

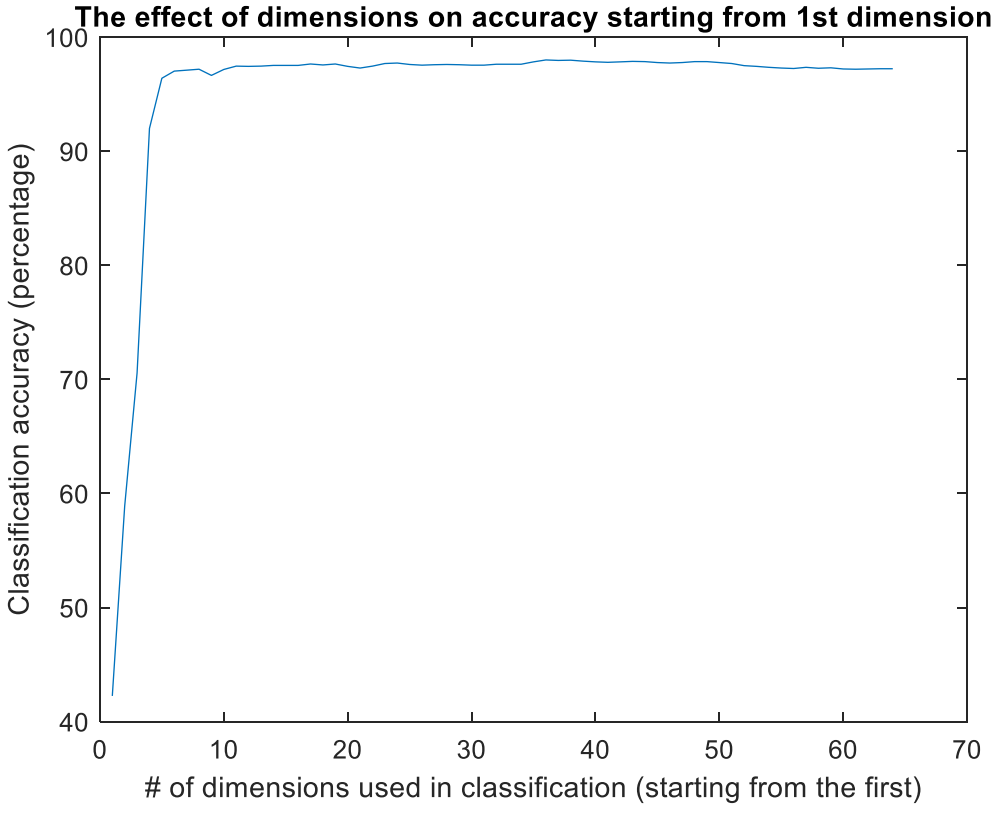
```
close all;
clear all;

load('data.mat');

% Seperating the data for different classes
cl1 = data(find(data(:,1)==1),:);
cl2 = data(find(data(:,1)==2),:);
cl3 = data(find(data(:,1)==3),:);
cl4 = data(find(data(:,1)==4),:);
cl5 = data(find(data(:,1)==5),:);
```

```
fig = figure;
fig.WindowState = 'maximized';
% Boxplot
sgtitle("Boxplot figure showing the distrubution of each dimension of different classes");
for i=1:64
    subplot(4,16,i);
    x = [cl1(:,i+1); cl2(:,i+1); cl3(:,i+1); cl4(:,i+1); cl5(:,i+1);];
    g = [ones(length(cl1(:,i+1)), 1); 2*ones(length(cl2(:,i+1)), 1);...
        3*ones(length(cl3(:,i+1)), 1); 4*ones(length(cl4(:,i+1)), 1);...
        5*ones(length(cl5(:,i+1)), 1)];
    boxplot(x, g);
    title(strcat("Dim ",int2str(i)));
end

fig = figure;
fig.WindowState = 'maximized';
% Individual classes plot
sgtitle("Histogram figure showing the distrubution of each dimension of Class 1");
for i=1:64
    subplot(4,16,i);
    histogram(cl1(:,i+1), 30);
    title(strcat("Dim ",int2str(i)));
end
fig = figure;
fig.WindowState = 'maximized';
% Individual classes plot
sgtitle("Histogram figure showing the distrubution of each dimension of Class 2");
for i=1:64
    subplot(4,16,i);
    histogram(cl2(:,i+1), 30);
    title(strcat("Dim ",int2str(i)));
end
fig = figure;
fig.WindowState = 'maximized';
% Individual classes plot
sgtitle("Histogram figure showing the distrubution of each dimension of Class 3");
for i=1:64
    subplot(4,16,i);
    histogram(cl3(:,i+1), 30);
    title(strcat("Dim ",int2str(i)));
end
fig = figure;
fig.WindowState = 'maximized';
% Individual classes plot
sgtitle("Histogram figure showing the distrubution of each dimension of Class 4");
for i=1:64
    subplot(4,16,i);
    histogram(cl4(:,i+1), 30);
    title(strcat("Dim ",int2str(i)));
end
fig = figure;
fig.WindowState = 'maximized';
% Individual classes plot
sgtitle("Histogram figure showing the distrubution of each dimension of Class 5");
for i=1:64
    subplot(4,16,i);
    histogram(cl5(:,i+1), 30);
    title(strcat("Dim ",int2str(i)));
end

fig = figure;
fig.WindowState = 'maximized';
% Scatters plot of the data for the last two dimensions
scatter(cl1(:,65), cl1(:,64), 'b', '.');
hold on
scatter(cl2(:,65), cl2(:,64), 'r', '.');
scatter(cl3(:,65), cl3(:,64), 'k', '.');
scatter(cl4(:,65), cl4(:,64), 'g', '.');
scatter(cl5(:,65), cl5(:,64), 'm', '.');
xlabel("Dim 64 values");
```

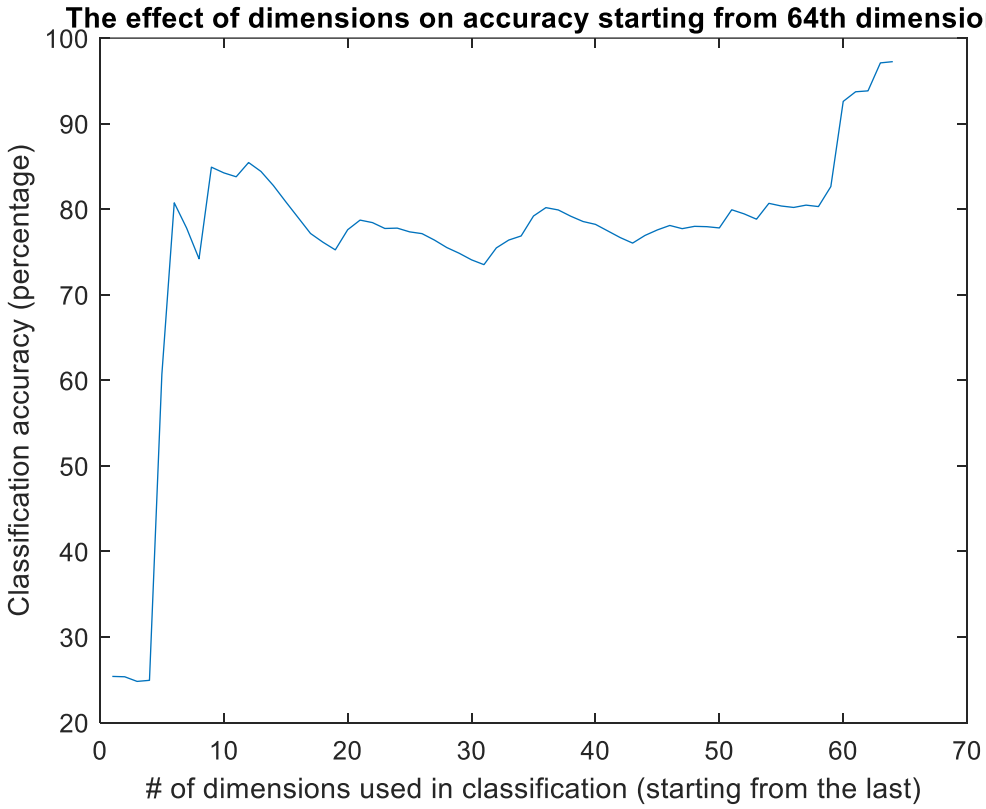
```
ylabel("Dim 63 values");
title("Scatter plot of the last two dimensions");
legend('Cl 1','Cl 2','Cl 3','Cl 4','Cl 5');

fig = figure;
fig.WindowState = 'maximized';
% Scatters plot of the data for the last two dimensions
scatter(cl1(:,7), cl1(:,8), 'b', '.');
hold on
scatter(cl2(:,7), cl2(:,8), 'r', '.');
scatter(cl3(:,7), cl3(:,8), 'k', '.');
scatter(cl4(:,7), cl4(:,8), 'g', '.');
scatter(cl5(:,7), cl5(:,8), 'm', '.');
xlabel("Dim 6 values");
ylabel("Dim 7 values");
title("Scatter plot of the two dimensions for which there are visible clusters");
legend('Cl 1','Cl 2','Cl 3','Cl 4','Cl 5');
```

APPENDIX C: Useless dimensions vs. Useful dimensions

As we have seen in the boxplot, the last 4 dimensions of the data has overlapping gaussian distributions. This means that they will probably be useless in terms of determining the class of the test inputs. To test this, a modified Naïve Bayes ClassifyX function was used that would only use the certain number of dimensions when constructing the multivariate gaussian distribution. For example, it could do classification just using the first data dimension, or the first 10 dimensions, or the last 5 dimensions, etc. Using this classifier, two plots were generated. The first plot shows how classification accuracy changes as the number of dimensions used in classifier increases (starting from dimension {1}, {1,2},{1,2,3},…, {1,…64}). It is given below.



It was seen that using just the first 6 dimensions, we could achieve a classification performance of approximately 97%. So, the first few dimensions gives pretty good information about the class of the inputs. What if we started from the end of the dimensions i.e. {64},{64,63},{64,63,62},…{64,…,1}. The below plot shows the results of such dimension selection.



It was seen that using only the last four dimensions, the classification performance stays around 25% which means it is just a little better that randomly assigning labels. Looking at the end of this graph, we see that the classification performance doesn't go up to 97% until it includes the very first dimensions.

These graphs show that the first few dimensions in the data has very distinctive features while the last dimensions does not. This information could be properly utilized in a dimensionality reduction algorithm.