# AINT351 MACHINE LEARNING 2016

## STUDENT NUMBER: 10450559

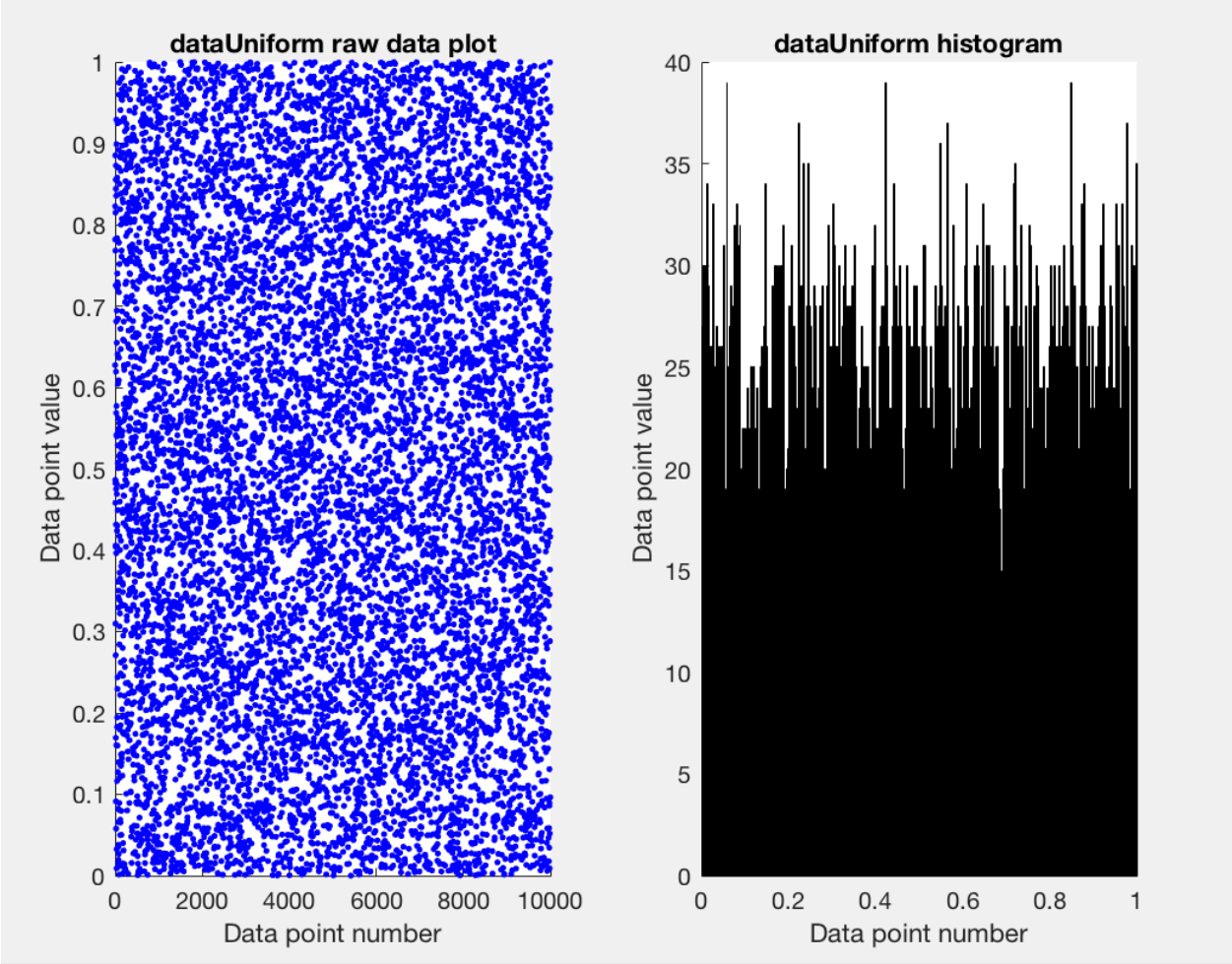
## Table of Contents

# AINT351 MACHINE LEARNING 2016
# STUDENT NUMBER:10450559

## P1.1 1D and 2D distributions

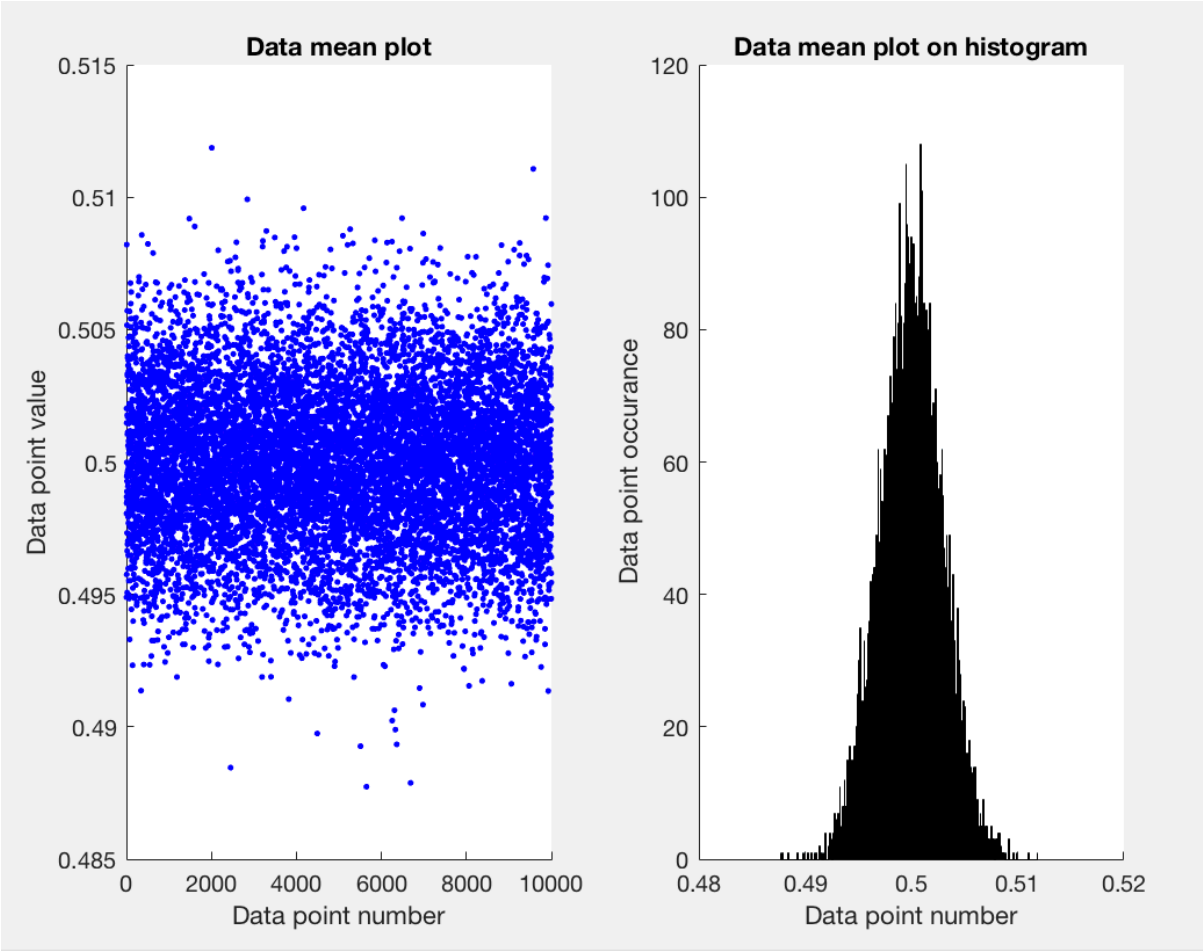
1. The results are of a uniform probability distribution of random samples to make up a 1xn matrix. This data is taken from Matlab using the rand function which is used for uniform data. The 1xn matrix is plotted on both the histogram and the plot diagram in a uniform fashion which produces the results. Plotting a 1xn data set means you are plotting the data points on one value i.e. in each column of the matrix, therefore this is a point plot of each column of this matrix. The plot is very random and not correlated or focused on any particular area due to the rand function being used to generate data on a 1D matrix. Doing so results in an uncorrelated, un-clustered set of results with no real statistical value being gained from these plots. By doing this it stands to show that when initially started the data set was truly random, therefore this can be used as a base to build on for the following tasks. The number of samples determine the size of the matrix/ the number of values to be plotted and the bins determines the number of intervals for the data in order for them to be organised and plotted onto the histogram. An increase of samples will only full the graph more with more random plotted points.

```
%Number of datapoints
samples = 10000;

%Create a Matrix of 1 X samples (1xn) from a uniform distribution
data = rand(1,samples);

%Specify the number of bins
nbins = 400;

%Print the matrix size
message = sprintf('dataUniform Matrix size %d x %d', size(data,1), size(data,2));
fprintf(message);

%Plot data on figure position 1
figure
subplot(1,2,1);
hold on
h = plot(data,'b.');
set(h,'linewidth',3);
title('dataUniform raw data plot');
xlabel('Data point number');
ylabel('Data point value');

%Plot data for the histogram on figure position 2
subplot(1,2,2);
hold on
histogram(data,nbins);
title('dataUniform histogram');
xlabel('Data point number');
ylabel('Data point value');
```


dataUniform raw data plot
Data point value
Data point number
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
2000
4000
6000
8000
10000
dataUniform histogram
Data point value
Data point number
0
5
10
15
20
25
30
35
40
0
0.2
0.4
0.6
0.8
1

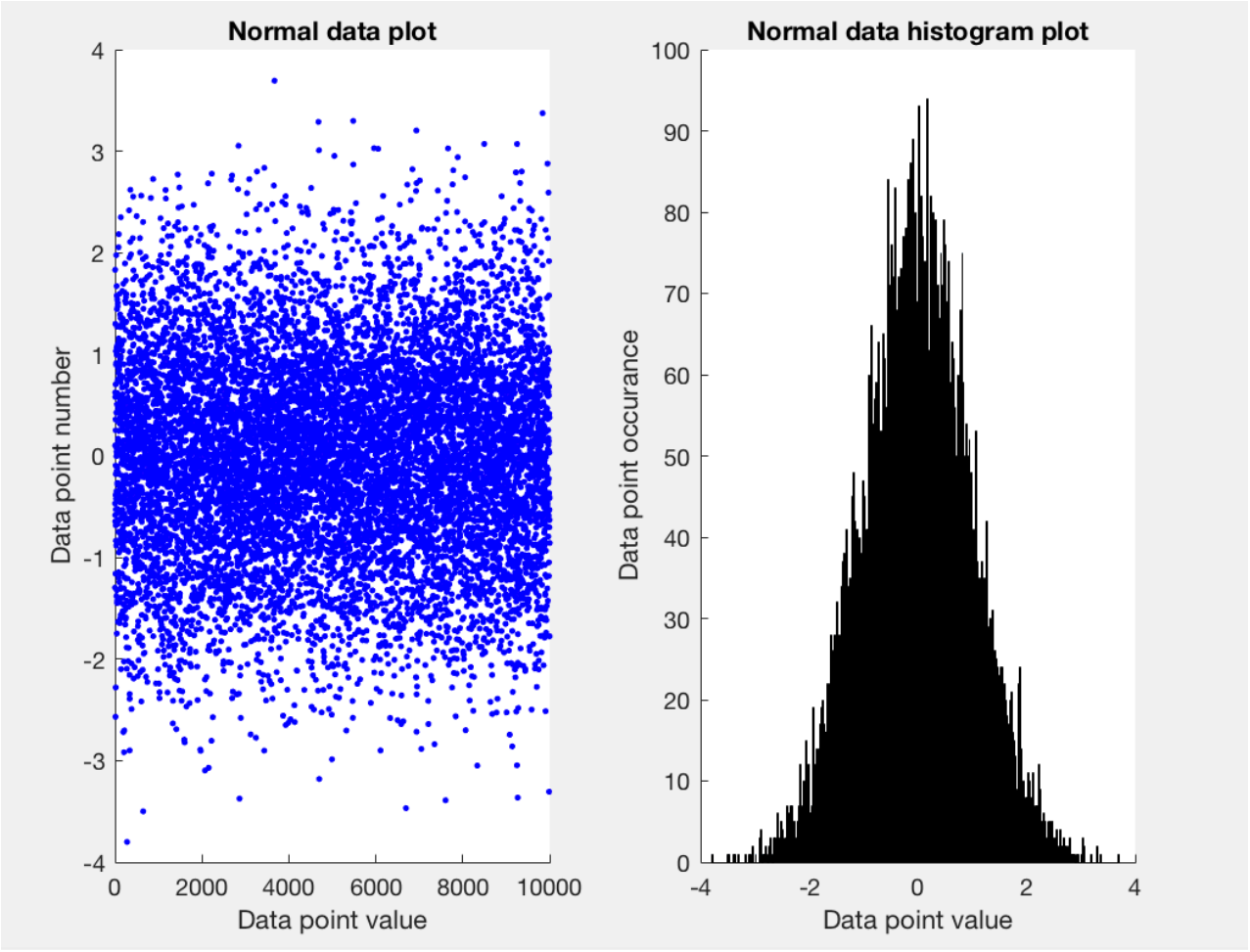
2. This shows the central limit theorem using a uniform distribution of samples similar to that of before. Central limit theorem states that given a large sample size from a population with finite level of variance, the mean of all the samples from the same population will approximately equal to the mean of the population. We took a large sample value (10000) and calculated the mean of this sample. After plotting the results, we can use central limit theorem to come to the conclusion that the approx. mean of the entire population is roughly 0.5. This was done by the following, a matrix of nxn is created from a random sample of the uniform distribution data. The 2nd dimension of the nxn matrix is used to calculate the mean. The 2nd dimension represents the columns of the matrix, therefore when the columns of the matrix have a mean applied across them each column uses the mean function through the entire matrix. This mean data was plotted on both a scatter plot and a histogram. From this we can see that the data is as if it is a normal distribution in both the plots cases. This is also a show of the central limit theorem as stated above. As before the increase sample size causes more plots to be plotted and an increase in bins causes more histogram bars to be plotted causing the plots to look even more like a normally distributed data set and giving a more accurate mean value for the population.

```
%Number of datapoints
samples = 10000;

%Create a Matrix of samples (nxn) from a uniform distribution
data = rand(samples);

%Get the mean of the 2nd dimension of the data set
meanData = mean(data,2);

%Specify the number of bins
nbins = 400;

%Print the matrix size
message = sprintf('data Matrix size %d x %d', size(data,1), size(data,2));
fprintf(message);

%Plot data on figure position 1
figure
subplot(1,2,1);
hold on
h = plot(meanData,'b.');
set(h,'linewidth',3);
title('Data mean plot');
xlabel('Data point number');
ylabel('Data point value');

%Plot histogram data on figure position 2
subplot(1,2,2);
hold on
histogram(meanData,nbins);
title('Data mean plot on histogram');
xlabel('Data point number');
ylabel('Data point occurance');
```


Data mean plot
Data point value
Data point number
0.515
0.51
0.505
0.5
0.495
0.49
0.485
0
2000
4000
6000
8000
10000
Data mean plot on histogram
Data point occurance
Data point number
120
100
80
60
40
20
0
0.48
0.49
0.5
0.51
0.52

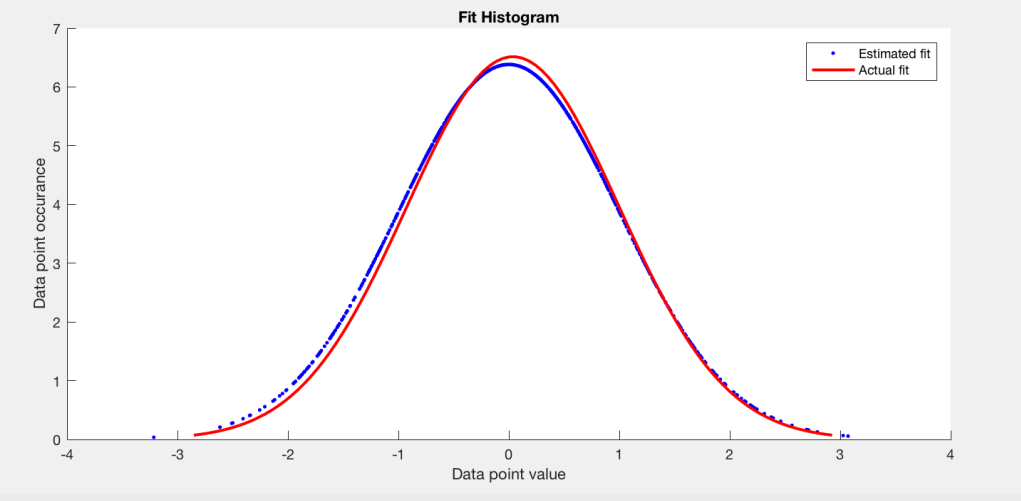
3. These results show a normal probability distribution. A normal probability distribution will create a data set will a mean of 0 and a variance of 1. The plot will look similar to that of central limit theorem above. In this case a 1xn matrix is created from a sample of data from Matlab. By using the randn function this specifies to take the data from Matlabs in built set of normally distributed data sets. This plot is similar to that of the previous central limit theorem but there is no need to manually calculate the mean as this has already been done by creating a normally distributed data set, the data from the matrix generated by the randn function is simply plotted onto both the scatter plot and histogram. As can be seen from the plots below the mean is in fact 0 as expected with a normally distributed data set. This shows that as the data approaches the mean value it starts to grow and then peak as it falls back down as it strays from the mean value, this shows how the data was created to conform to the normal distribution expected. The larger the samples size used the clearer the normal distribution becomes. Similarly, with the increase of samples the increase of bins for the histogram will also make the normal distribution more clear. Both of these will show more clearly that the mean is in fact 0, as we are sampling from the population we can also say that the mean for the whole population is approximately 0 also.

```
 %Number of datapoints
samples = 10000;

%Create a Matrix of 1 X samples (1xn) from a normal distributon
data = randn(1,samples);

%Specify the number of bins
nbins = 400;

%Print the matrix size
message = sprintf('data Matrix size %d x %d', size(data,1), size(data,2));
fprintf(message);

%Plot data on figure position 1
figure
subplot(1,2,1);
hold on
h = plot(data,'b.');
set(h,'linewidth',3);
title('Normal data plot');
xlabel('Data point value');
ylabel('Data point number');

%Plot data on figure position 2
subplot(1,2,2);
hold on
i = histogram(data,nbins);
title('Normal data histogram plot');
xlabel('Data point value');
ylabel('Data point occurance');
```


Normal data plot
Data point number
Data point value
4
3
2
1
0
-1
-2
-3
-4
0
2000
4000
6000
8000
10000
Normal data histogram plot
Data point occurance
Data point value
100
90
80
70
60
50
40
30
20
10
0
-4
-2
0
2
4

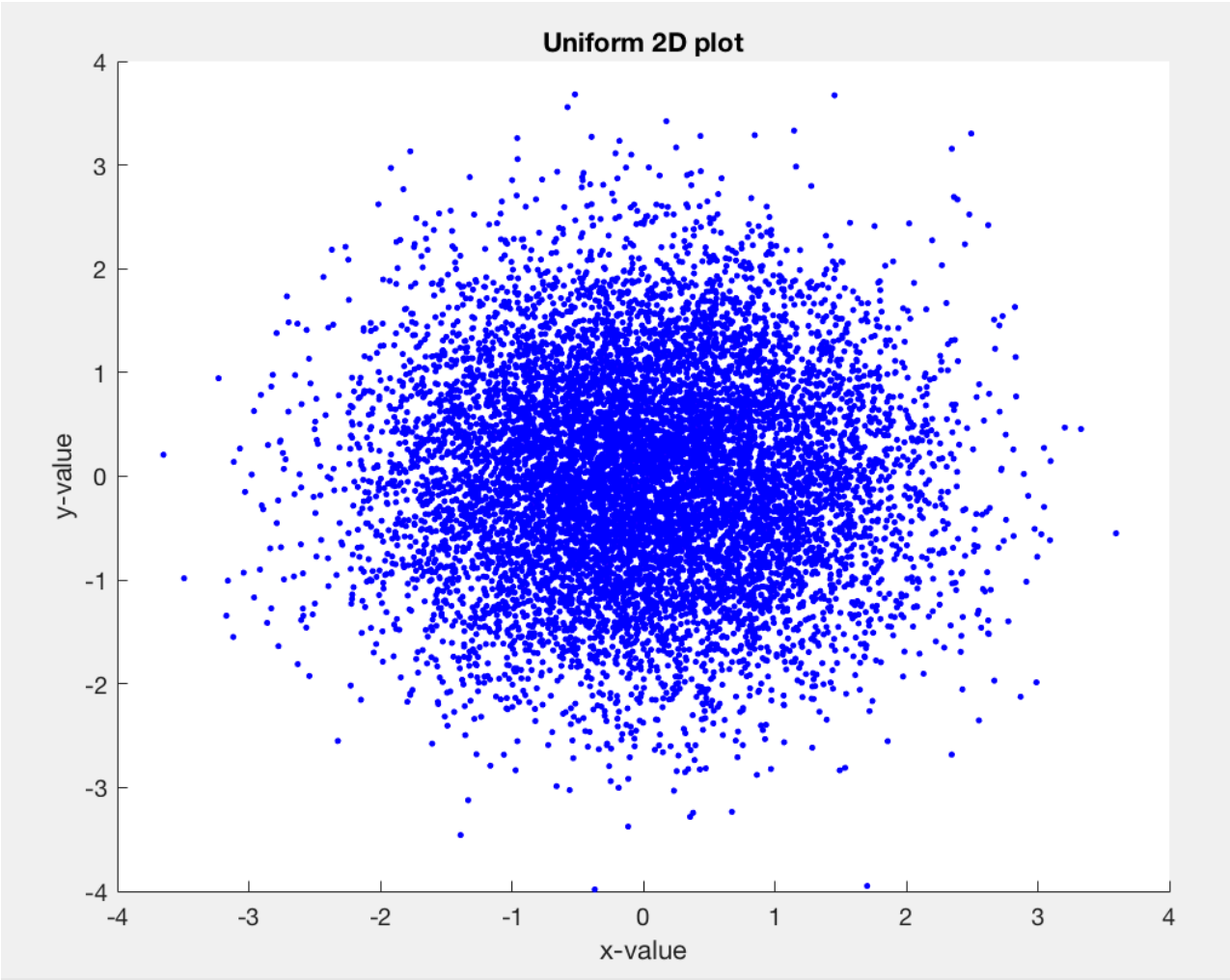
4. The same as before this graphs shows the normal distribution along with the estimated values for the distribution. The actual distribution line has been fitted to the histogram where an increase in samples and bins will result in a more well fitted line and a better representation of the normal distribution. This plot is also known as a Gaussian distribution. By estimating the values of the mean and variance (square root of the standard deviation) we can estimate the plot and fitted line that the normal distribution will create. In order to have the estimation fit as closely as possible to the actual fit we have to scale the data appropriately. Otherwise the estimated plotted data will not provide much value when compared against the actual fit. Here the normal probability distribution is calculated (with the estimated variance and mean) and this is plotted against the data generated and once scaled will give an accurate representation of the normal distribution/ Gaussian plot.

```
1    %Number of datapoints
2 -  samples = 1000;
3
4    %Create a Matrix of 1 X samples (1xn)
5 -  data = randn(1,samples);
6
7    %Specify the number of bins
8 -  nbins = 400;
9
10   %the normal probability density function set mean to 0 and standard deviation to 1
11 - norm = normpdf(data,0,1)*16;
12
13   %Split the data using the automatic algorithm
14 - N = histcounts(data);
15
16   %Print the matrix size
17 - message = sprintf('data Matrix size %d x %d', size(data,1), size(data,2));
18 - fprintf(message);
19
20   %Plot the norm and the histogram with a fitted line on a figure.
21 - figure
22 - hold on
23 - plot(data,norm,'b.');
24 - h = histfit(data,nbins);
25 - title('Fit Histogram');
26 - ylabel('Data point occurance');
27 - xlabel('Data point value');
```



5. In order to create a 2D distribution plot a 2D matrix is needed. Therefore, we again use the randn function but this time to create a 2xn matrix with a normal set of data. The randn function has a default standard deviation of 1 and mean of 0 which is applied to the 2D matrix. In order to represent this data, we must plot the dimensions against each other. To do this we take all of the first dimension as x and all of the second dimension as y. From the plot (once x and y are plotted against each other) we can see that the distribution is very focused on 0 on both axis this is due to the mean of 0 and the set of normal data which is applied by the randn function when the matrix was created. By doing this the mean for the first and second dimensions are both 0 and standard deviations of 1. This results in a clustering of data around the (0,0) point on the plot. Most data are very closely cluster and is quite highly correlated. An increase in samples will result on a more clustered distribution allowing the focus of 0 to become more clear by showing a much more concentrated, higher correlated and closer clustered data points. With the large sample number that was used it can be assumed that the mean of the overall population is also 0.

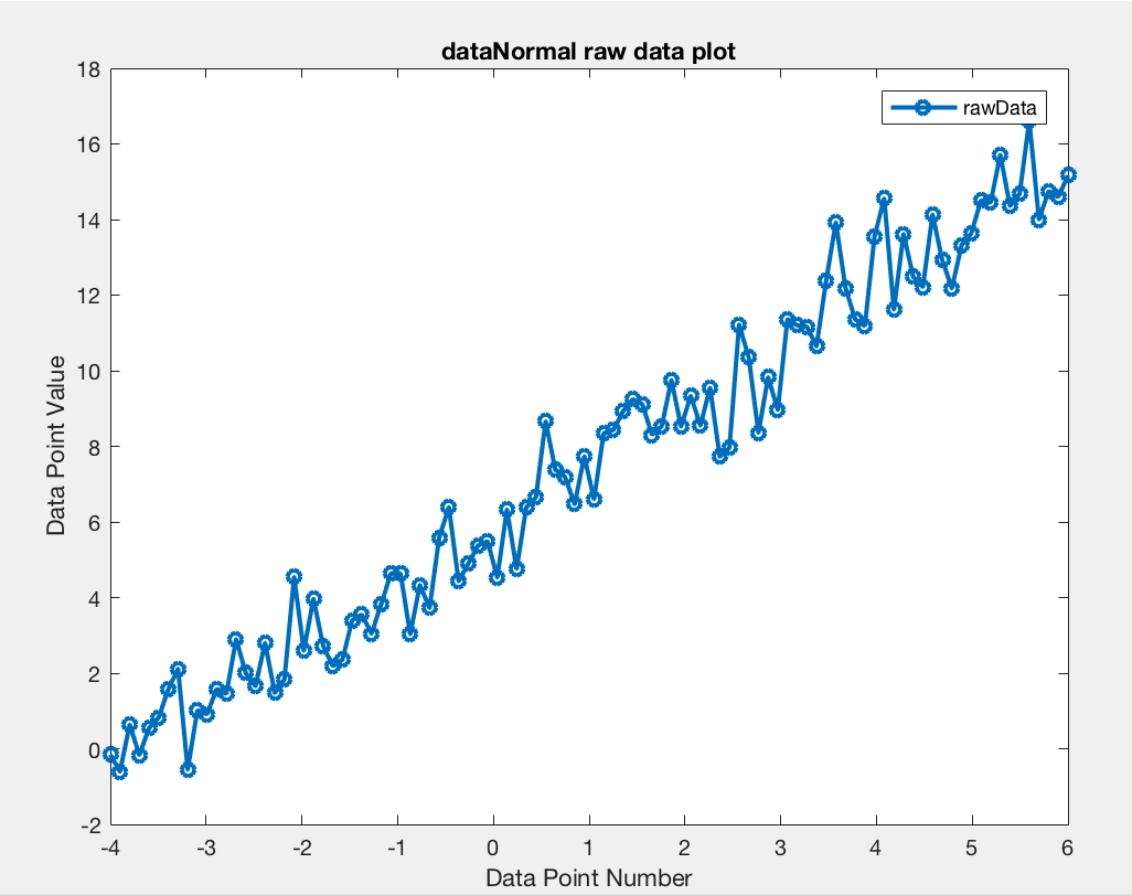
```
%Number of datapoints
samples = 10000;

%Create a Matrix of 2 X samples (2xn) (randn is produced with a mean of 0
%and standard deviation of 1, otherwise it would be: standardDeviation * randn(2,n) + mean;)
data = randn(2,samples);

%Set the x and y points to be plotted. One dimension against another.
x = data(1,:);
y = data(2,:);

%Print the matrix size
message = sprintf('data Matrix size %d x %d', size(data,1), size(data,2));
fprintf(message);

%Plot data on figure position 1
figure
subplot(1,1,1);
hold on
scatter(x,y,'b.');
title('Uniform 2D plot');
xlabel('x-value');
ylabel('y-value');
```


Uniform 2D plot
y-value
x-value
4
3
2
1
0
-1
-2
-3
-4
-4
-3
-2
-1
0
1
2
3
4

## P1.2 Linear regression

1. A basic line equation was used to get a set of y values. This equation used the following values: gradient (m) of 1.6, y-intercept (c) 6 and a sample of x between -4 and 6. A normally distributed data set was created in order to add noise to the line. This was done by generating a data set of matrix size 1xn, by using randn the mean is 0 and standard deviation is 1 resulting in a Gaussian data set. The sample number is the number of sample points generated, this sample number must be 100 as the linspace function will generate 100 evenly spaced points between the two values specified for x. This noise was then added to the previously generated value of y in order to be plotted. By doing this the straight line is skewed with the noise to produce the plot as seen below.

```
%Specify number of samples to take
samples = 100;

%Set limits for x value
minX = -4;
maxX = 6;

%Sample x and specify other variable
x = linspace(minX,maxX);
c = 6;
m = 1.6;

%Combine variables to calculate y in a linear equation
y = m*x+c;

%Create noise for data in a Gaussian format (default std = 1 mean = 0)
noise = randn(1,samples);

%Plot x against y adding on the Gaussian noise
figure
plot(x,y+noise,'-o','linewidth',2)
ylabel('Data Point Value');
xlabel('Data Point Number');
title('dataNormal raw data plot');
legend('rawData');
```

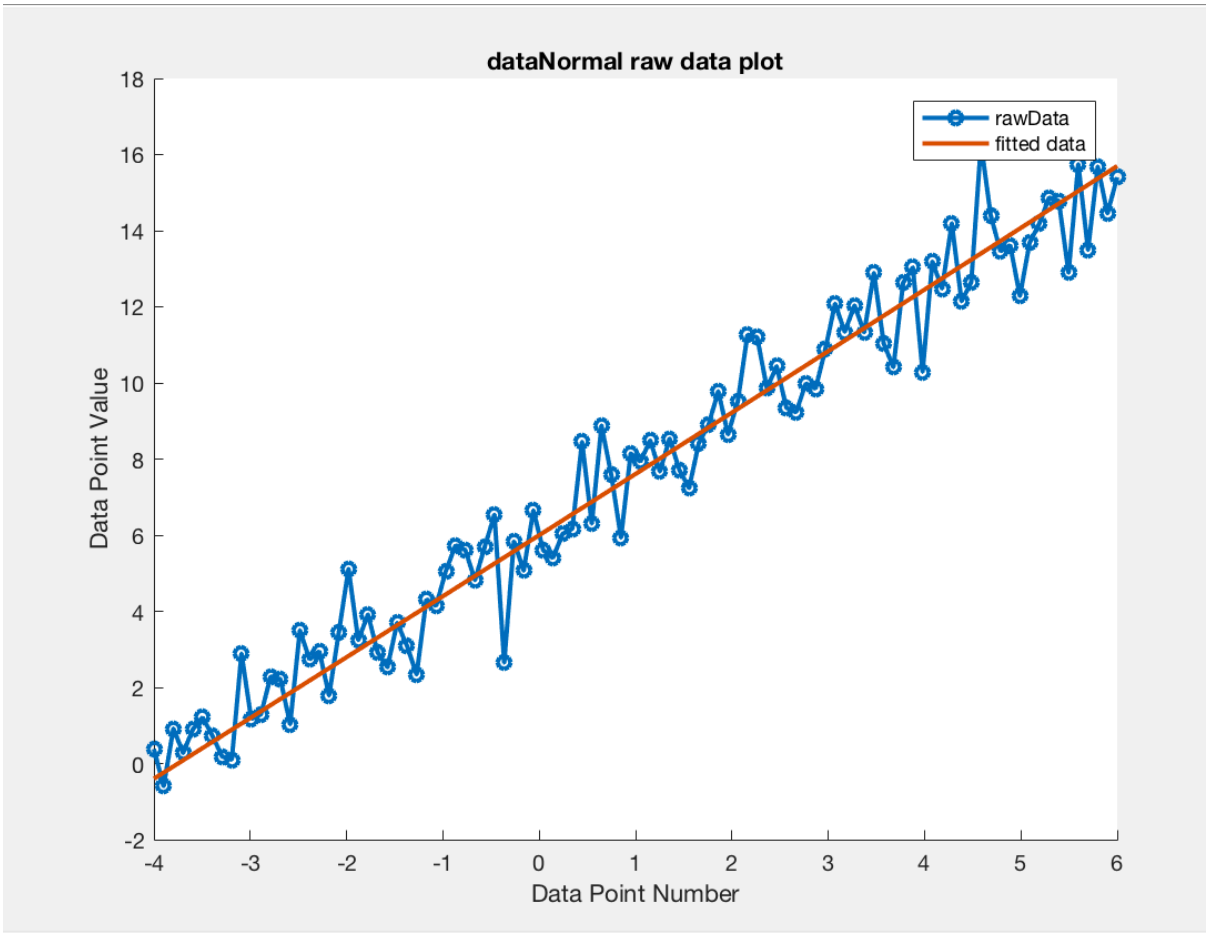

dataNormal raw data plot
rawData
Data Point Value
Data Point Number
18
16
14
12
10
8
6
4
2
0
-2
-4
-3
-2
-1
0
1
2
3
4
5
6

2. In this task, linear regression was implemented on the above noisy data above. Linear regression is an approach for modelling the relationship between a variable y and one or more variables denoted X. The coding for this is the same as the previous task with the addition of the computed weight matrix (highlighted). With this noisy line, linear regression can be applied in order to find the best fitted line. In order to do this the weight matrix at minimum error needs to be calculated. This will determine how far off the point is from what is expected and is to be used to plot the best fitted line. When the weight matrix is calculated, it is the same size as the original data set which in this case is 1x100. This is due to the fact that there are 100 points and the error value for each point is calculated and stored. Least square fitting is used to produce how far from the expected points is, the square and sum of these values can then be used to calculate a fitted line. This can be done is either terms of c or m in order to get the gradient of the line.

```
1     %Specify number of samples to take
2 -   samples = 100;
3
4     %Set limits for x value
5 -   minX = -4;
6 -   maxX = 6;
7
8     %Sample x and specify other variable
9 -   x = linspace(-4,6);
10 -  c = 6;
11 -  m = 1.6;
12
13    %Combine variables to calculate y in a linear equation
14 -  y = m*x+c;
15
16    %Create noise for data in a Gaussian format (default std = 1 mean = 0)
17 -  noise = randn(1,samples);
18
19    %compute weight matrix at min error
20 -  W = inv(x * x')*x.*y;
```

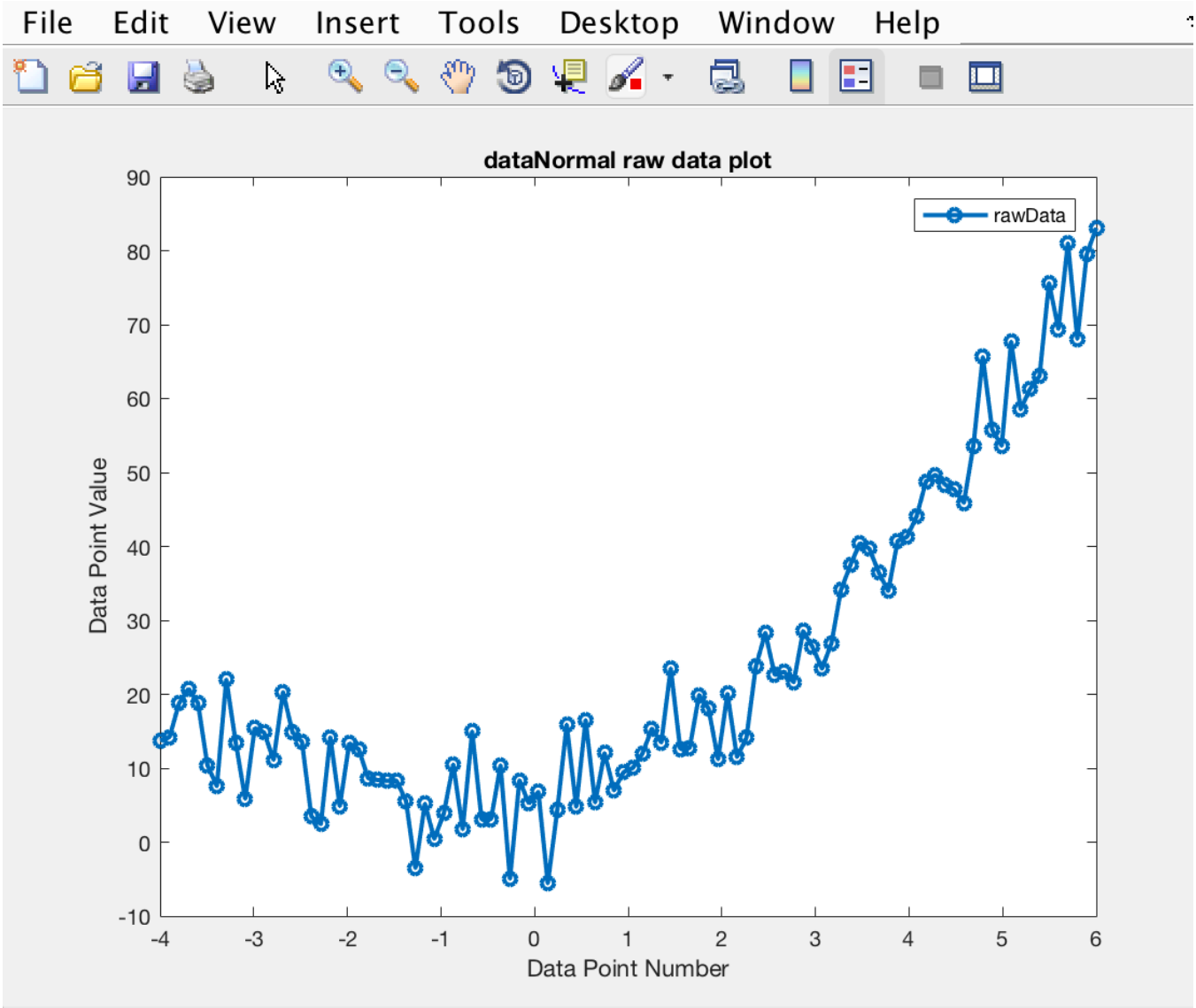
3. The above weight matrix is applied to the y value from the line equation used. This done in order to create the best fitted line to the noise data set. The new y values are plotted against the x values on top of the noisy line. A best fitted line is produced for this data given data which is plotted as expected. As the noisy data isn't straying too far apart, which is due to the randn function as the standard deviation is 1, the fitted line is plotted essentially straight through the middle of the data points set. The fitted line shows a linear positive overall gradient. The gradient and y-intercept is approximately that of the stated m and c values respectively at the start of the script. The code for this is the same as previously with the addition of the 'figure' code which has been highlighted.

```
6 -    maxX = 6;
7
8      %Sample x and specify other variable
9 -    x = linspace(-4,6);
10 -   c = 6;
11 -   m = 1.6;
12
13     %Combine variables to calculate y in a linear equation
14 -   y = m*x+c;
15
16     %Create noise for data in a Gaussian format (default std = 1 mean = 0)
17 -   noise = randn(1,samples);
18
19     %compute weight at min error
20 -   W = inv(x * x')*x.*y;
21
22     %Plot x against y adding on the Gaussian noise and plot the fitted line
23     %ontop of the noisy line.
24 -   figure
25 -   hold on
26 -   plot(x,y+noise,'-o','linewidth',2);
27 -   plot(x,y+W,'-','linewidth',2);
28 -   legend('rawData','fitted data');
29 -   ylabel('Data Point Value');
30 -   xlabel('Data Point Number');
31 -   title('dataNormal raw data plot');
```


dataNormal raw data plot
rawData
fitted data
Data Point Value
Data Point Number
18
16
14
12
10
8
6
4
2
0
-2
-4
-3
-2
-1
0
1
2
3
4
5
6

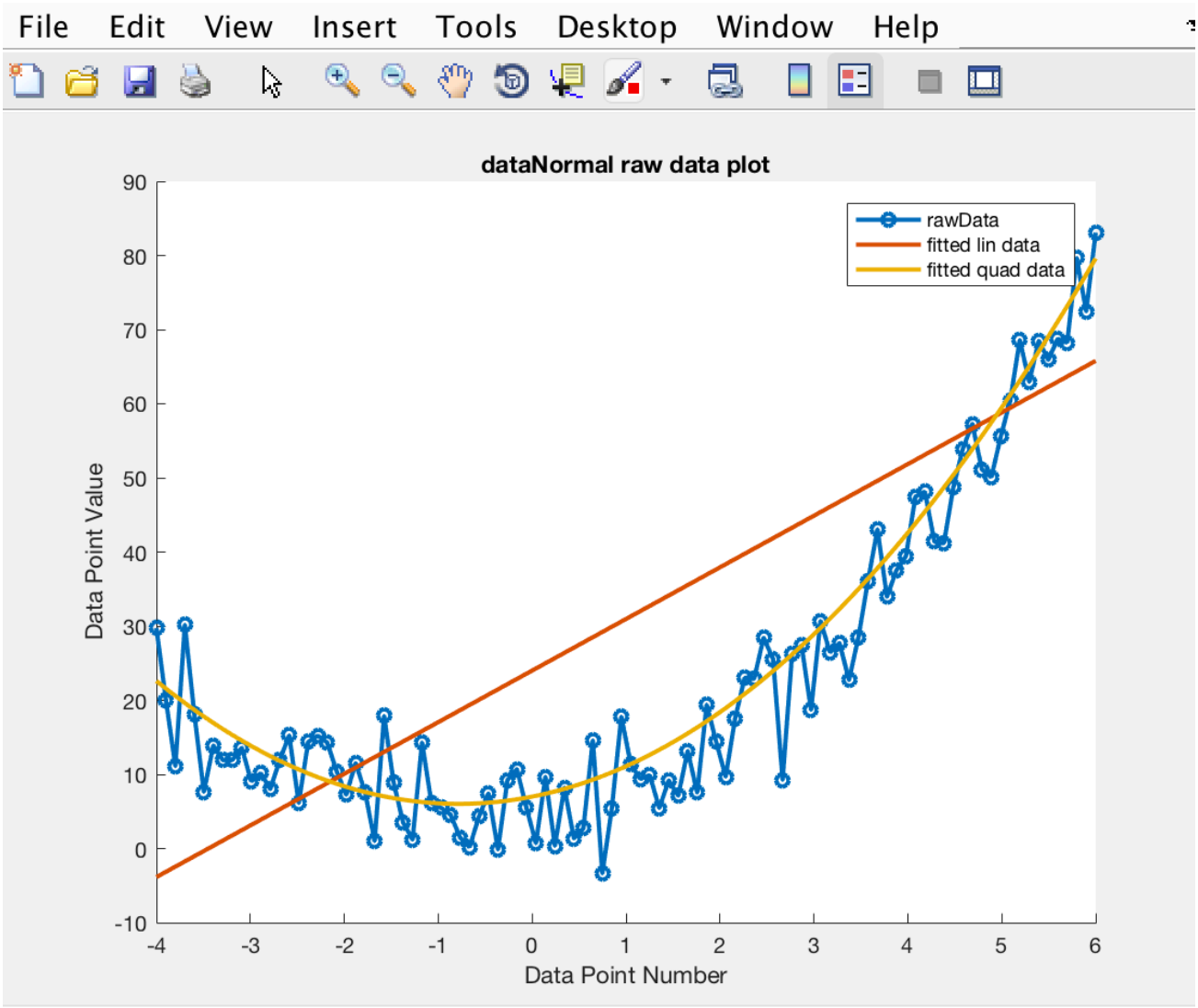
4. A quadratic curve equation was used to get a set of y values. This equation used the quadratic coefficients A = 1.6, B= 2.5 and C= 6 with the equation being 'y=A*x^2+B*x+C;'. This kind of equation is called a parabola; a parabola is a plotted line that opens either upwards or downwards in a 'U' shape. The constants A and B affects whether the parabola opens upwards or downwards (positive and negative respectively). The magnitude of A defines how steep the parabola is. The closer it is to 0, the flatter the parabola will be, and the more it will resemble a straight-line. In this case, approximately half of the 'U' will be plotted and it will open upwards. A normally distributed data set was created in order to add noise to the curve. This was done by generating a data set of matrix size 1xn, by using randn the mean with the set mean of 0 and standard deviation of 5 resulting in a Gaussian data set to be used as noise. Same as the linear regression on a straight line the sample number is the number of sample points generated, this sample number must be 100 as the linspace function will generate 100 evenly spaced points between the two values specified for x. This noise was then added to the previously generated value of y in order to be plotted. By doing this the curve is skewed with the noise to produce the plot as seen below. From the plot, it can be seen that the lowest point of the curve is approximately at 0, which is the same as the mean. This noisy data set has a general upwards gradient.

```
%Specify number of samples to take
samples = 100;

%Set limits for x value
minX = -4;
maxX = 6;

%Quadratic coefficients
A = 1.6;
B = 2.5;
C = 6;

%Sample x
x = linspace(minX,maxX);

%Mean and stanDev
mean = 0;
stanDev = 5;

%Combine variables to calculate y in a linear equation
y = A*x.^2+B*x+C;

%Create noise for data in a Gaussian format (default std = 5 mean = 0)
noise = stanDev.*randn(1,samples)+mean;

%Plot x against y adding on the Gaussian noise
figure
plot(x,y+noise,'-o','linewidth',2);
ylabel('Data Point Value');
xlabel('Data Point Number');
title('dataNormal raw data plot');
legend('rawData');
```


File Edit View Insert Tools Desktop Window Help
dataNormal raw data plot
rawData
Data Point Value
Data Point Number
90
80
70
60
50
40
30
20
10
0
-10
-4
-3
-2
-1
0
1
2
3
4
5
6

5. After the noisy data has been plotted the best fit lines are then to be calculated and fitted. A straight line and a quadratic line can both be fitted to this noisy data plot. In order to do this, we use the same code as before, which has the quadratic noisy data plot, as a base and build on this to fit the lines. To do this first a linear and quadratic base is to be created as the variables xLin and Quad respectively. The base is needed for the Matlab regress function to work. The base made up of a series of 1s, the results of the regress function are calculated under the assumption of this and if there is no base the resulting values will be incorrect. To plot the quadratic, curve the result of the regress is used to plot against in order to find the best fit line. The quadratic best fit line is seen to be an upwards gradient with no local minima or maxima but has the lowest point on the fitted line at approximately 0 which coincides with the mean value that was set when generating the data. This fit runs roughly through the centre of the set of points plotted fitting to the noisy data. The straight fitted line has a positive gradient also cutting the noisy quadratic curve and the quadratic fitted line at approximately (-2, 10) and again at (5,60). From all of this it can be said that this data set has a general upwards trend at a relatively steep steady gradient.

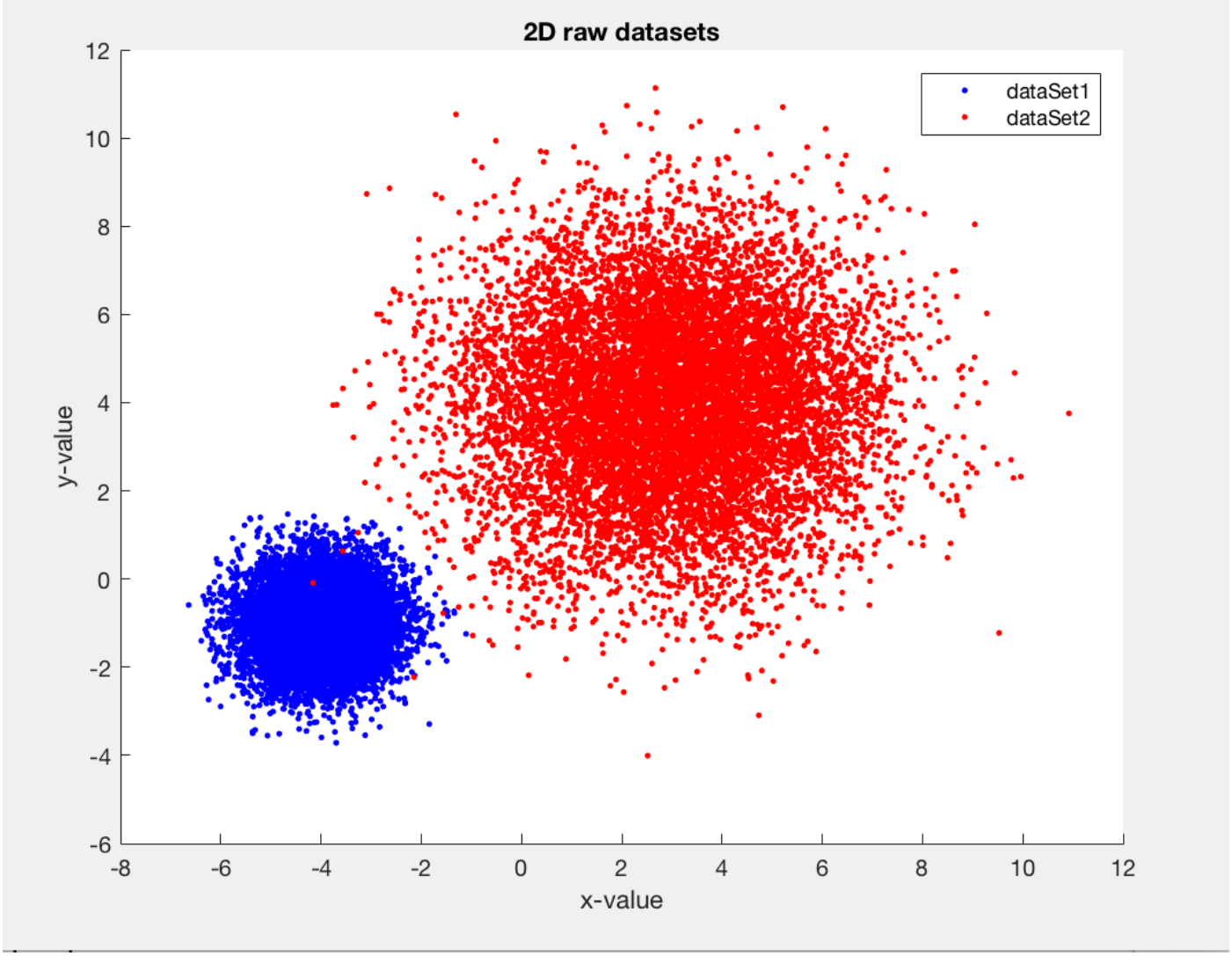
```
%Specify number of samples to take
samples = 100;

%Set limits for x value
minX = -4;
maxX = 6;

%Quadratic coefficients
A = 1.6;
B = 2.5;
C = 6;

%Sample x and specify other variable
x = linspace(minX,maxX);

%Mean and stanDev
mean = 0;
stanDev = 5;

%Combine variables to calculate y in a quadratic equation
y = A*x.^2+B*x+C;

%Find the gradient for the quadratic curve
m = (y - C)/x;

%Use the gradient to find yL
yL = m*x + C;

%Create noise for data in a Gaussian format (default std = 5 mean = 0)
noise = stanDev.*randn(1,samples)+mean;

%Linear basis for function
xLin = [x; ones(1,samples);];

%Fit test data with linear line
linFit = regress(y',xLin');

%Quadratic basis for function
Quad = [x.*x;x;ones(1,samples);];

%Fit test data with quadratic line
quadFit = regress(y',Quad');

%Plot x against y adding on the Gaussian noise and plot the two best fit
%lines for both the straight line and quadratic.
disp(size(linFit));
figure
hold on
plot(x,y+noise,'-o','linewidth',2);
plot(x,yL+linFit(2),'-','linewidth',2);
plot(x,y+Quad(3),'-','linewidth',2);
legend('rawData','fitted lin data','fitted quad data');
ylabel('Data Point Value');
xlabel('Data Point Number');
title('dataNormal raw data plot');
```


File Edit View Insert Tools Desktop Window Help
dataNormal raw data plot
rawData
fitted lin data
fitted quad data
Data Point Value
Data Point Number
90
80
70
60
50
40
30
20
10
0
-10
-4
-3
-2
-1
0
1
2
3
4
5
6

## P1.3 KMeans Clustering

1. Two datasets have been created each of which are 2D uncorrelated datasets. The two datasets have overlapping clusters where a few points in each cluster are seen to be in the cluster of the other. Each data set is 10000 points totalling 20000 points in total to be plotted. The two sets both have their own mean and standard deviation values which are used to determine the centroid (mean) and spread of the values (standard deviation). The x and y values are split out from the datasets based on the dimensions of 1 and 2 of the 2D datasets. This could also have been done in the plot rather than assigning x1, x2, y1 and y2 ahead of time. As can be seen from the plot below the two clusters have been plotted and they have indeed some overlapping points. Dataset2 (red cluster) is less clustered than dataset1 (blue cluster), meaning that the red points are more spread out from the centroid than the corresponding blue points. This is expected due to the fact that dataset2 had a higher standard deviation value than dataset1. The blue points are much more correlated with each other but do still overlap into the red data points.

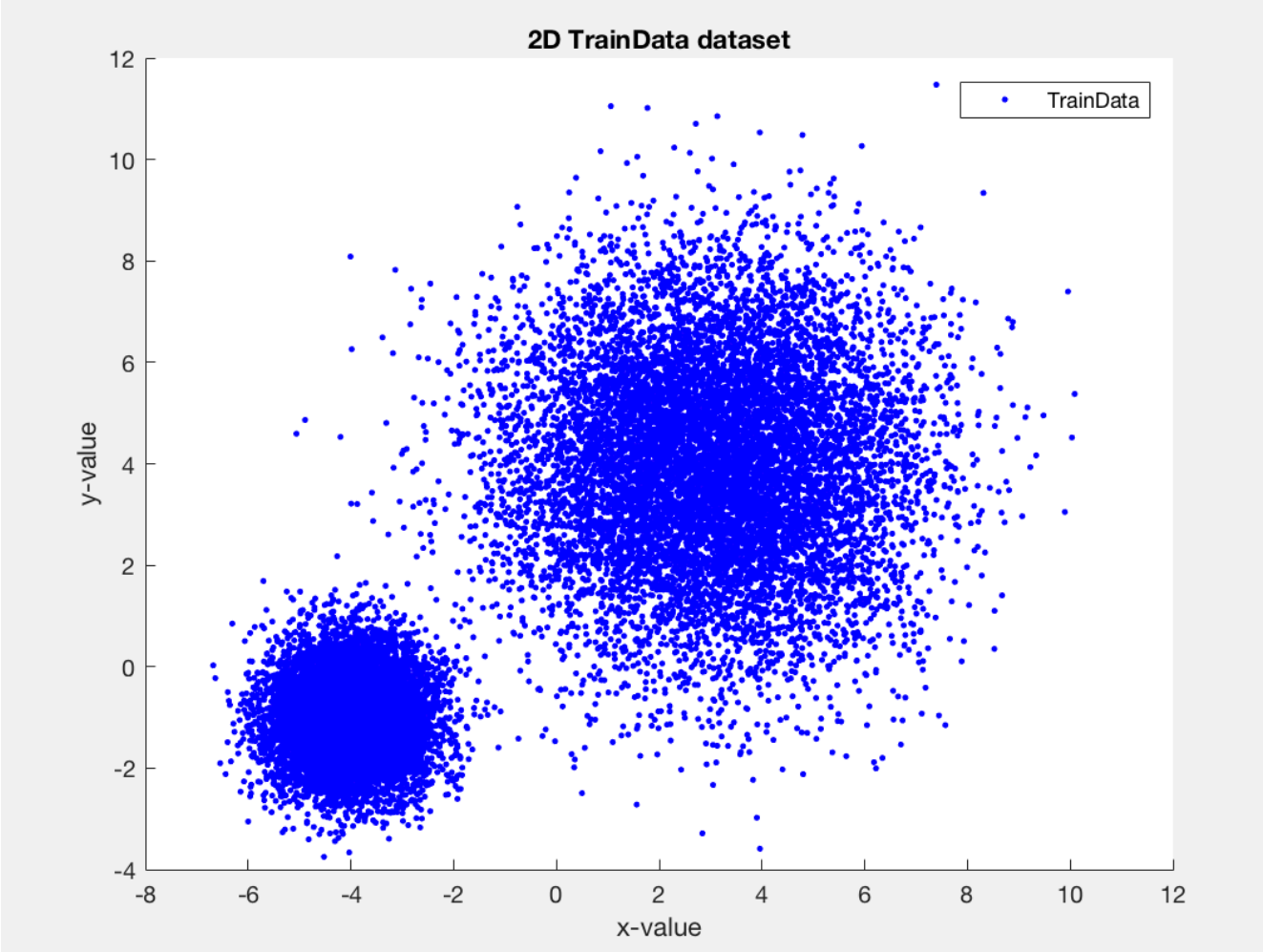
```
%select number of samples for each set
samples1 = 10000;
samples2 = 10000;

%define the mean for each set
meanSet1 = [-4 -1]';
meanSet2 = [3 4]';

%define the standard deviation for each set
stanDevSet1 = 0.75;
stanDevSet2 = 2;

%Create a Matrix of 2 X samples (2xn) with the specified mean and standard
%deviation using 'standardDeviation * randn(2,n) + mean;' for each data set
dataSet1 = stanDevSet1 * randn(2,samples1) + meanSet1;
dataSet2 = stanDevSet2 * randn(2,samples2) + meanSet2;

%Define the x and y values for set 1 and 2 respectively
x1 = dataSet1(1,:);
y1 = dataSet1(2,:);

x2 = dataSet2(1,:);
y2 = dataSet2(2,:);

%plot the two data sets with dataset1 being plotted as blue dots and
%dataset2 being plotted as red dots.
figure
hold on
plot(x1,y1,'b.');
plot(x2,y2,'r.');
title('2D raw datasets');
xlabel('x-value');
ylabel('y-value');
legend('dataSet1','dataSet2');
```


2D raw datasets
dataSet1
dataSet2
y-value
x-value
12
10
8
6
4
2
0
-2
-4
-6
-8
-6
-4
-2
0
2
4
6
8
10
12

2. The two datasets have been concatenated into 1 dataset called TrainData which now has a size of 4x20000. This TrianData can now be split into 4 components based on each dimension. This will give 2 sets of coordinates one for cluster seen above. By plotting this it all shows as the one colour merging the two previous clusters in separate sets to two clusters in one set. This TrainData is to be used to implement KMeans on new sets of data in order to carry out correct classification. It will use the TrainData as a base and an example and build the new sets based on this. By doing this and training on this set the overlapping points can organised and reclassified into the correct cluster resulting in no overlapping points. The clusters are the same as before but overlaps cannot be distinguished to see if there is indeed an overlap.

```
%select number of samples for each set
samples1 = 10000;
samples2 = 10000;

%define the mean for each set
meanSet1 = [-4 -1]';
meanSet2 = [3 4]';

%define the standard deviation for each set
stanDevSet1 = 0.75;
stanDevSet2 = 2;

%Create a Matrix of 2 X samples (2xn) with the specified mean and standard
%deviation using 'standardDeviation * randn(2,n) + mean;' for each data set
dataSet1 = stanDevSet1 * randn(2,samples1) + meanSet1;
dataSet2 = stanDevSet2 * randn(2,samples2) + meanSet2;

%Concatenate the two sets into a single dataset
TrainData = [dataSet1; dataSet2];

%Define the x and y values for the 4 dimensions
x1 = TrainData(1,:);
y1 = TrainData(2,:);
x2 = TrainData(3,:);
y2 = TrainData(4,:);
%Plot both sets of coordinates
figure
hold on
plot(x1,y1,'b.');
plot(x2,y2,'b.');
title('2D TrainData dataset');
xlabel('x-value');
ylabel('y-value');
legend('TrainData');
```

# AINT351 MACHINE LEARNING 2016
# STUDENT NUMBER:10450559



3. At this point the classification and reclassification of the data sets are done. The end result is to correctly classify the data sets so there is no overlap. The initial process is the same as above i.e. creating the TrainData set, defining the mean and standard deviation etc. To start the TrainData set us used to initiate the process, this is done do create a basis for the actual datasets and the initial clusters. After the first iteration, the datasets that are to be classified are used.

   With each loop through the series the classification is done per dataset classifying each point. The process is as follows:

   - Take the mean value and use this as the initial centroid for each cluster (this is due to the TrainData run first so we know the centroids are here).
   - Loop through each point in each of the data sets and for each point:
     - Find the distance from the point to the centroid of cluster one.
     - Find the distance from the point to the centroid of cluster to.
     - Determine which distance is shortest and therefore which cluster centroid is closest.
     - Assign the point to the closest centroid cluster and this is the classification for that point.
   - This will result in two new sets being made of the new classifications.
   - Recalculate the mean for each of these new sets which will produce the centroid of these new clusters.
   - Use these new means/ centroid and loop through the process again.

   This process is done until the centroids no longer move i.e. there is no change to the clusters, and therefore all the points have been classified. The results of this means that each point is assigned to the correct cluster being the one it is closest to. This will element the overlap seen before and will find the best classification for these sets. This script could be changed to a function and the datasets can be passed in so this can be done for any datasets.

```
%select number of samples for each set
samples1 = 10000;
samples2 = 10000;

%define the mean for each set
meanSet1 = [-4 -1]';
meanSet2 = [3 4]';

%Abrituaryly set the mean for this at first
newMeanSet1 = [1 1]';
newMeanSet2 = [1 1]';

%define the standard deviation for each set
stanDevSet1 = 0.75;
stanDevSet2 = 2;

%Create a Matrix of 2 X samples (2xn) with the specified mean and standard
%deviation using 'standardDeviation * randn(2,n) + mean;' for each data set
uncorrelatedData1 = stanDevSet1 * randn(2,samples1) + meanSet1;

uncorrelatedData2 = stanDevSet2 * randn(2,samples2) + meanSet2;

%Concatenate the two sets into a single dataset
TrainData = [uncorrelatedData1; uncorrelatedData2];

%Define the x and y values for the 4 dimensions
x1 = TrainData(1,:);
y1 = TrainData(2,:);

x2 = TrainData(3,:);
y2 = TrainData(4,:);

%Assigned arbitrary values for the means of the norm data
oldMean1 = 0;
oldMean2 = 1;
newMean1 = 2;
newMean2 = 3;

%Number of iterations through the outer while loop starting at 1
iterationNumber = 1;

%Define an empty matrix for the cluster datasets
classidx1 = [];
classidx2 = [];

%Variable for the while loop
n = 1;

%Compare the norm means for the cluster to determine if there is a change
%or not
while oldMean1 ~= newMean1 && oldMean2 ~= newMean2
    %Variables to show which column in the matrix to insert the data. Start
    %at 1 with each new iteration
    classidx1Column = 1;
    classidx2Column = 1;

    %If on the first iteration enter this in order to use the TrainData
    %as the intial plot/ clusters.
    if iterationNumber == 1
        while n <= size(TrainData,2)
            %Assign the x and y values from the Train data
            x = TrainData(1,n);
            y = TrainData(2,n);

            %Calculate the distance from the centroid of each clusters
            %for each point in the provided dataset
```

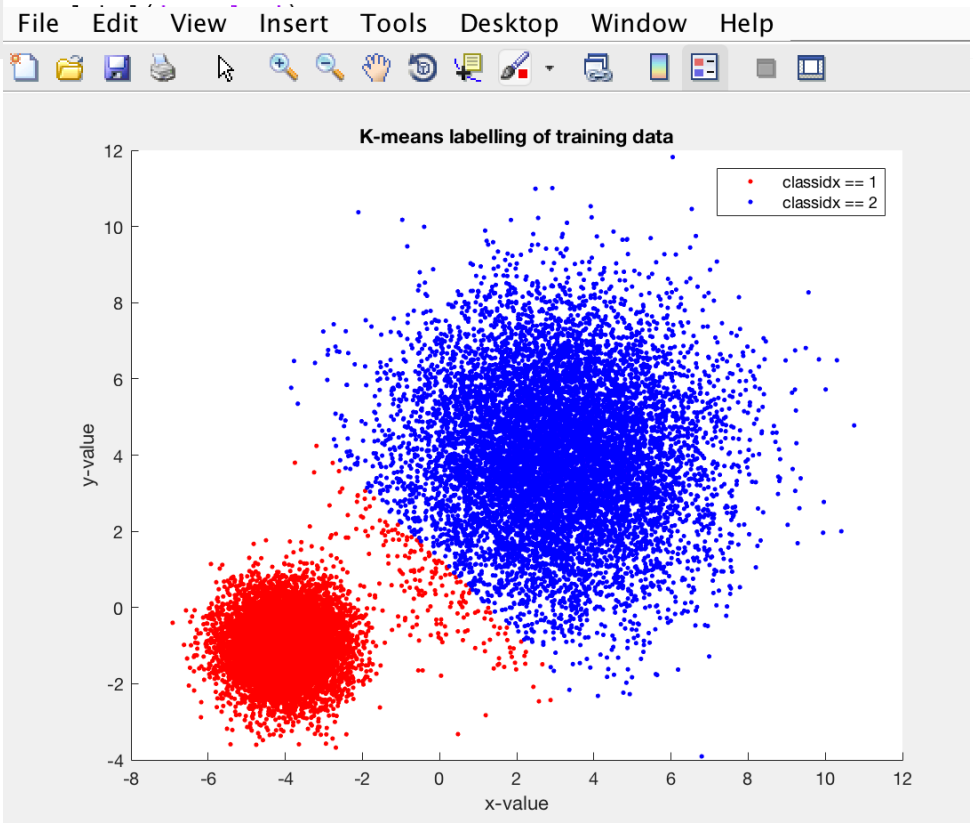
```
            distance1 = norm(meanSet1 - [x y]);
            distance2 = norm(meanSet2 - [x y]);
            %Assign the values to the cluster which it is closest to
            if(distance1 < distance2)
                classidx1(1,classidx1Column) = x;
                classidx1(2,classidx1Column) = y;
                classidx1Column = classidx1Column + 1;
            else
                classidx2(1,classidx2Column) = x;
                classidx2(2,classidx2Column) = y;
                classidx2Column = classidx2Column + 1;
            end
            n = n + 1;
        end
        %Reset n back to one for the next dataset
        n = 1;
        %---The below while block is the same as above for the second
        %dataset
        while n <= size(TrainData,2)
            x = TrainData(3,n);
            y = TrainData(4,n);

            distance1 = norm(meanSet1 - [x y]);
            distance2 = norm(meanSet2 - [x y]);

            if(distance1 < distance2)
                classidx1(1,classidx1Column) = x;
                classidx1(2,classidx1Column) = y;
                classidx1Column = classidx1Column + 1;
            else
                classidx2(1,classidx2Column) = x;
                classidx2(2,classidx2Column) = y;
                classidx2Column = classidx2Column + 1;
            end
```

```
        %Reset n to one for the next iteration so the matrix can be looped
        %through again.
        n=1;
    %Enter this else statement on any iteration number except the first
    else
        %Iterate through each element of the matrix in the first class
        %dataset
        while n <= size(classidx1,2)
            %Assign the x and y values
            x = classidx1(1,n);
            y = classidx1(2,n);

            %Calculate the distance from the centroid of each cluster
            distance1 = norm(meanSet1 - [x y]);
            distance2 = norm(meanSet2 - [x y]);
            %Assign the values to the cluster which it is closest to
            if(distance1 < distance2)
                classidx1(1,classidx1Column) = x;
                classidx1(2,classidx1Column) = y;
                classidx1Column = classidx1Column + 1;
            else
                classidx2(1,classidx2Column) = x;
                classidx2(2,classidx2Column) = y;
                classidx2Column = classidx2Column + 1;
            end
            n = n + 1;
        end
        %Reset for the next while loop
        n = 1;
        %---Do the same as above for the possible second dataset and assign
        %them to the cluster they are closest to
        while n <= size(classidx2,2)
            x = classidx2(1,n);
            y = classidx2(2,n);
```

```
137 
138 -             distance1 = norm(meanSet1 - [x y]);
139 -             distance2 = norm(meanSet2 - [x y]);
140 -             if(distance1 < distance2)
141 -                 classidx1(1,classidx1Column) = x;
142 -                 classidx1(2,classidx1Column) = y;
143 -                 classidx1Column = classidx1Column + 1;
144 -             else
145 -                 classidx2(1,classidx2Column) = x;
146 -                 classidx2(2,classidx2Column) = y;
147 -                 classidx2Column = classidx2Column + 1;
148 -             end
149 -             n = n + 1;
150 -         end
151 -         n=1;
152 -     end
153 
154       %Calculate the means for the new data sets again.
155 -     newMeanSet1 = nanmean(classidx1);
156 -     newMeanSet2 = nanmean(classidx2);
157 
158       %Assign these newly calculated mean values to the variables used above
159       %in the calculations for the next iteration.
160 -     meanSet1 = newMeanSet1';
161 -     meanSet2 = newMeanSet2';
162 
163       %Assign this mean the was just used in above loops to a new varible.
164       %This is done to compare if the mean has changed (centroid) in the
165       %outer most while loop
166 -     oldMean1 = newMean1;
167 -     oldMean2 = newMean2;
168 
169       %Create the norm value for the new mean values. These will be used with
170       %the oldMean values to determine if the centroids  have moved.
171 -     newMean1 = norm(newMeanSet1);
172 -     newMean2 = norm(newMeanSet2);
173 
174       %Add one to the iteration number to keep track of how many times the
175       %process has been executed.
176 -     iterationNumber = iterationNumber+1;
177 - end
```

4. The code is the same as above but with the addition of the plots. This will give a visual representation of the new classification with no overlapping. When this is plotted, we are looking for two clusters as this is number of clusters in the TrainData set and this is number of datasets passed in. As can be seen from the plot below there are indeed two cluster with no overlap. There is a distinct separation of the two clusters which can be seen where the red points meet the blue points. This is due to the classification to the nearest cluster. At this point the distance from the point to each of the two centroids is the closest resulting in this split. By classifying like this the clusters are now more clustered and focused around their centroids. In this case the TrainData was used for testing. By comparing the testing set and clusters against the training set and clusters can result in overfitting. This means that the noise is picked up by the set and this is learned, this can have a negative impact on the performance of the model. These concepts, noise and fluctuation that are picked up by the training data cannot be used on other data sets which results in an impact on the model to generalize. To avoid this a validation data set can be used. A validation data set is a subset of the training data that is withheld until the end of the testing data fitting. By then running the validation set you can get an idea of the overall performance of the system with this new different set from the testing set. Cross validating unseen data is a great way to help element this problem.

```
177        %Plot the results once the centroids no longer change showing convergence.
178 -      figure
179 -      hold on
180 -      plot(classidx1(1,:),classidx1(2,:),'r.');
181 -      plot(classidx2(1,:),classidx2(2,:),'b.');
182 -      title('K-means labelling of training data');
183 -      xlabel('x-value');
184 -
185 -
```

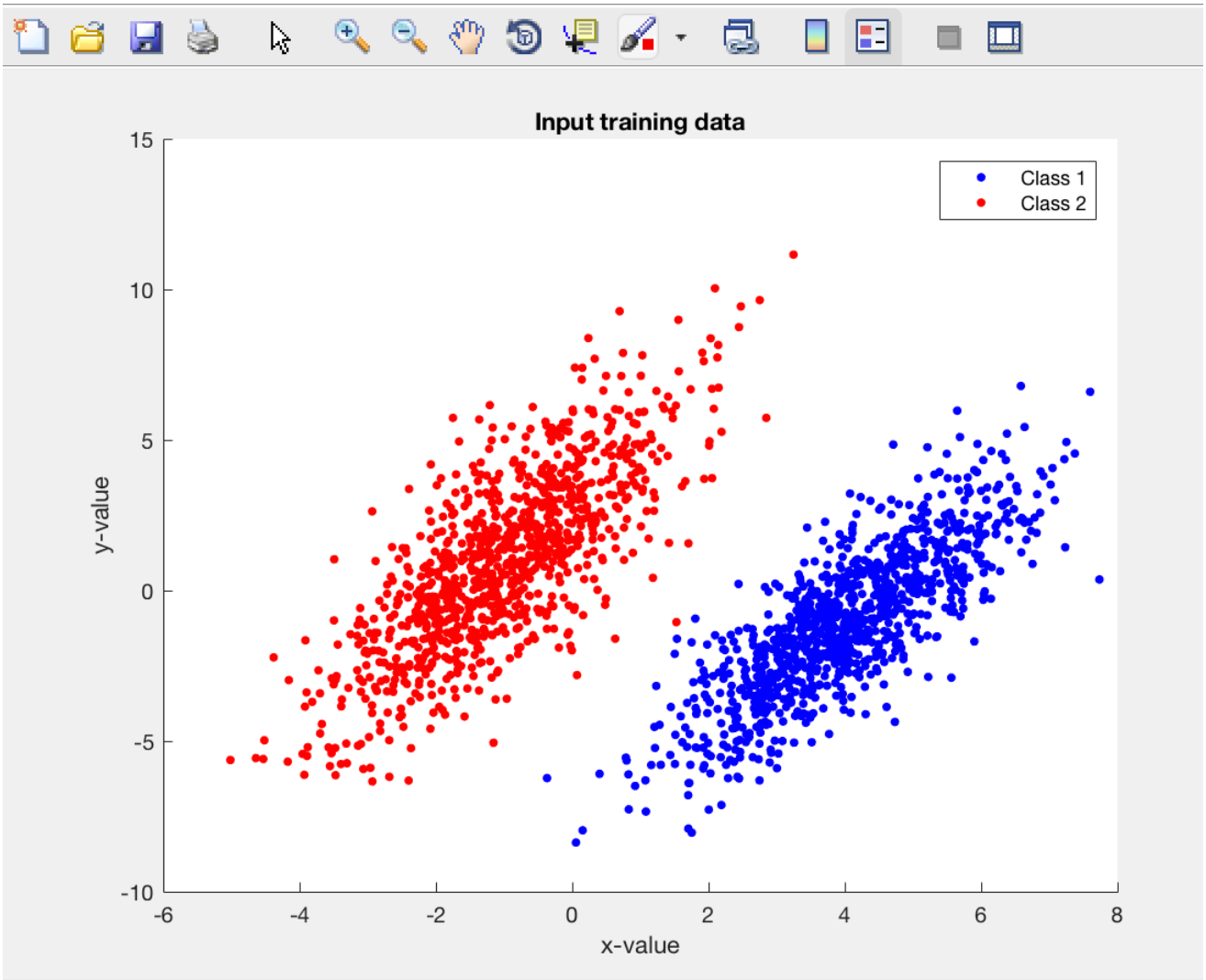


## P1.4 Naïve Bayes and perceptron classifier

1. Initially two sets are created by passing the sample sizes required to the GenerateGaussianData function which will return a trainingData and target and a testingData and target. From this the two classes, can be separated out and plotted independently. The separation is needed in order train the classifier. When the two classes are plotted it can be seen that both have a positive upward trend and slightly, but not heavily, clustered within themselves.

```
function Part1BayesClassifier
% Set the sample numbers for the trainging and testing data
trainingSamples = 1000;
testingSamples = 100000;

%Get the two sets
[trainingData, trainingTarget] = GenerateGaussianData(trainingSamples);
[testingData, testingTarget] = GenerateGaussianData(testingSamples);

%Separate out the classes
label1 = repmat([1; 0;] , 1, trainingSamples);
label2 = repmat([0; 1;] , 1, trainingSamples);
%Concate the two
targetVec = [label1 label2];

%vectorized example to extract all class 1 patterns
% examine first dimension which is 1 for class 1
fidx = find(targetVec(1,:) == 1);
c1data = trainingData(:,fidx);

%vectorized example to extract all class 2 patterns
% examine first dimension which is 0 for class 2
fidx = find(targetVec(1,:) == 0);
c2data = trainingData(:,fidx);

    %Plot the two separated sets on same plot.
    figure
    hold on
    plot(c1data(1,:), c1data(2,:), 'b.','markersize', 10);
    plot(c2data(1,:), c2data(2,:), 'r.','markersize', 10);
    xlabel('x-value');
    ylabel('y-value');
    title('Input training data');
    legend('Class 1','Class 2');
```

# AINT351 MACHINE LEARNING 2016
# STUDENT NUMBER:10450559



2. Naïve Bayes is used for constructing classifiers based on Bayes' theorem. These classifiers are highly scalable. By taking into account prior probability with a naïve Bayes approach, two data sets can be classified and separated. Naïve Bayes does very well with large data sets but is widely used for a variety of problems all across the machine learning environment. Naïve Bayes assumes that the predictor on a class is independent of the values of the other predictor i.e. the value of one data sets has no impact on the values of the other data set. Due to its simplicity, Naïve Bayes is often seen to outperform other, more complicated, classification methods.

3. From the plot, it can be seen that there is a distinct decision boundary between the two classes. The classes are relatively well clustered within themselves but there is some overlap. There is more overlap in the class 1 data than that into the class 2 data, these points that 'leak' into the other class are not being classes as expected. This is due to the fact that this is a classification error, multiple of these can be seen as some of the plots are not as expected as stated above. The decision boundary seems to start at 0 and cure upwards and to the right slightly, it would be expected that the boundary would cut between the two data sets in a straight diagonal line.

4/5. The single layer perceptron will take multiple input values, apply a weight to each and an output will be produced. Unlike multilayer perceptions there are not many hidden layers between the input and the output there is just one where the weight is applied. To apply this to a data set a perceptron of N inputs would be created and the training set would be applied across them. The output would be calculated from this and the rule of the perceptron to update the weights would be applied. This is done may times with each iteration through the training set being carried out until the total training set error ceases to improve. The output of a perceptron can either be a 1 or a 0 with the weight update given by $\Delta w_i = c(t - z)\, x_i$. Two forms of update can be adopted either batch or sequential, batch is where all the data is presented for each iteration and sequential is where it is presented one at a time. If the data is linearly separable then the decision boundary will correctly classify all points. The algorithm will stop where there are no incorrectly classified training points and therefore, a converge to a solution is guaranteed.

## P2.1 Classification with decision trees

1. To start this task the Fisher Iris set was downloaded from online and loaded into the Matlab workspace. From here the first function was created, learn decision tree. This function is to be the base function of the entire program. This is where the tree is to be built and other functions are to be called. Mostly this is made up of a lot of variables and function calls.

```
function learnDescisionTree(variables, classification)

global variableSet;
variableSet = variables;

measData = num2cell(variables);

global dataSet;
dataSet =[measData,classification];

global variableNum;
variableNum = 1;

global thresholdValue;
thresholdValue = 1;

global rowNumber;
rowNumber = 1;

global bestSplitSet1;
global bestSplitSet2;
global bestVariableSet;
global bestVariable;

split(variableNum,variableSet,thresholdValue);



end
```

2. An important function in this task is the split function. This is where the splitting of the data sets based on a threshold value is conducted. The data sets are split accordingly and assigned into new sets which will later be used for further splitting. The code of the split function below has comment which explain what is happening. However, to do this I looped through the data set using a for loop and compared the value of the corresponding data with that of the threshold value. Depending on where the value was greater than or less than the threshold determined which new split data set it goes into. The new split sets created were called irisSet1 and irisSet2.

```
function split(variableNum, variableSet, thresholdValue)
%Get the VarName of the data (column), index to show the cell of the data (row), and
%threshold for the greater or less than
%Output value of the classification depending on the threshold and value
global dataSet;
rowNum = 1;
value = variableSet(rowNum, variableNum);
global irisSet1;
irisSet1 = [];
global irisSet2;
irisSet2 = [];

%For loop to loop through the data set intervals of records going from 0
%(value passed from learnDecisisonTree) to the max number. Use this to
%determine the type and populate into the next column.
for n=1:size(dataSet,1)
    if (value < thresholdValue)
        %add to subset 1
        irisSet1 = [irisSet1;
dataSet(rowNum,1),dataSet(rowNum,2),dataSet(rowNum,3),dataSet(rowNum,4),dataSet(rowNum,5)];
        value = variableSet(rowNum,variableNum);
        rowNum = rowNum+1;
    else
        %add to subset 2
        irisSet2 = [irisSet2;
dataSet(rowNum,1),dataSet(rowNum,2),dataSet(rowNum,3),dataSet(rowNum,4),dataSet(rowNum,5)];
        value = variableSet(rowNum,variableNum);
        rowNum = rowNum+1;
    end
end

entropyFunction(irisSet1,irisSet2,dataSet);

end
```

3. Entropy function is used as a measure of randomness in a data set. For the previously split data sets (and any other data set passed in) this entropy function will calculate the entropy for both sets. First I counted the number of each classification in each data set, including the original un-split data set. Then I calculated the probability of each of the classifications, an issue arose due to the fact that if there were no classifications of a certain time the probability would cause an error. Therefore, to avoid this a series of if statements were used to check for the 0 value. After this the entropy, can be calculated for each classification in each set. A sum of the entropies can then be used to find an improvement value.

```
function entropyFunction(irisSet1,irisSet2,dataSet)

%Count number of each classification in the dataSets
setosaCount = sum(strncmp('setosa',dataSet,6));
virginicaCount = sum(strncmp('virginica',dataSet,6));
versicolorCount = sum(strncmp('versicolor',dataSet,6));

setosaCountSplit1 = sum(strncmp('setosa',irisSet1,6));
virginicaCountSplit1 = sum(strncmp('virginica',irisSet1,6));
versicolorCountSplit1 = sum(strncmp('versicolor',irisSet1,6));

setosaCountSplit2 = sum(strncmp('setosa',irisSet2,6));
virginicaCountSplit2 = sum(strncmp('virginica',irisSet2,6));
versicolorCountSplit2 = sum(strncmp('versicolor',irisSet2,6));

%Calculate the probability of each classification
%Need the if statements as if there is not split for that threshold in the
%split set an error will be thrown due to the undefined number.
if(size(dataSet,1)~=0)
    probSetosa = setosaCount(1,5)/size(dataSet,1);
    probVirginica = virginicaCount(1,5)/size(dataSet,1);
    probVersicolor = versicolorCount(1,5)/size(dataSet,1);
else
    probSetosa = 0;
    probVirginica = 0;
    probVersicolor = 0;
end

if(size(irisSet1,1) ~= 0)
    probSetosaSplit1 = setosaCountSplit1(1,size(setosaCountSplit1,2))/size(irisSet1,1);
    probVirginicaSplit1 = virginicaCountSplit1(1,size(virginicaCountSplit1,2))/size(irisSet1,1);
    probVersicolorSplit1 = versicolorCountSplit1(1,size(versicolorCountSplit1,2))/size(irisSet1,1);
else
    probSetosaSplit1 = 0;
    probVirginicaSplit1 = 0;
    probVersicolorSplit1 = 0;
end

if(size(irisSet2,1) ~= 0)
    probSetosaSplit2 = setosaCountSplit2(1,size(setosaCountSplit2,2))/size(irisSet2,1);
    probVirginicaSplit2 = virginicaCountSplit2(1,size(virginicaCountSplit2,2))/size(irisSet2,1);
    probVersicolorSplit2 = versicolorCountSplit2(1,size(versicolorCountSplit2,2))/size(irisSet2,1);
else
    probSetosaSplit2 = 0;
    probVirginicaSplit2 = 0;
    probVersicolorSplit2 = 0;
end

%Calculate the entropy for each classification
entropySetosa = (probSetosa * log2(probSetosa));
entropyVirginica = (probVirginica * log2(probVirginica));
entropyVersicolor = (probVersicolor * log2(probVersicolor));

%Check if probability is zero because if it is zero and the log is
%carried out in next step the value will return infinity.
if(probSetosaSplit1 ~= 0)
    entropySetosaSplit1 = (probSetosa * log2(probSetosaSplit1));
else
    entropySetosaSplit1 = 0;
end

if(probVirginicaSplit1 ~= 0)
    entropyVirginicaSplit1 = (probVirginica * log2(probVirginicaSplit1));
else
```

```
    entropyVirginicaSplit1 = 0;
end

if(probVersicolorSplit1 ~= 0)
    entropyVersicolorSplit1 = (probVersicolor * log2(probVersicolorSplit1));
else
    entropyVersicolorSplit1 = 0;
end

if(probSetosaSplit2 ~= 0)
    entropySetosaSplit2 = (probSetosa * log2(probSetosaSplit2));
else
    entropySetosaSplit2 = 0;
end

if(probVirginicaSplit2 ~= 0)
    entropyVirginicaSplit2 = (probVirginica * log2(probVirginicaSplit2));
else
    entropyVirginicaSplit2 = 0;
end

if(probVersicolorSplit2 ~= 0)
    entropyVersicolorSplit2 = (probVersicolor * log2(probVersicolorSplit2));
else
    entropyVersicolorSplit2 = 0;
end

%Sum all of the above entropies
entropy = -(entropySetosa + entropyVirginica + entropyVersicolor);
entropySplit1 = -(entropySetosaSplit1 + entropyVirginicaSplit1 + entropyVersicolorSplit1);
entropySplit2 = -(entropySetosaSplit2 + entropyVirginicaSplit2 + entropyVersicolorSplit2);

improvement(entropy, entropySplit1, entropySplit2,irisSet1,irisSet2,dataSet);

 end
```

4. Improvement is known as the change in entropy. By using the values, I calculated in the entropy function, the calculation of the entropy was very straight forward. The improvement is calculated by the original entropy minus the sum of the other entropies. However, the values needed to be waited, therefore, to do this I used the size of the split sets to calculate the percentage difference between the split and the original and used this as the weighted value to multiply by.

```
function improvement(entropy, entropySplit1,
entropySplit2,irisSet1,irisSet2,dataSet)

%Calcualte the improvement
%For improvement this will be the original minus the sum of the other
%entropies
%The entropies must be weighted for example if set1 has 90% of the values
%the entropy of set 1 should be multiplied by 0.9 and same for set 2. Total
%records is 150 split 1 has 51 split 2 has 99. Using size here to get the
%percentage of records in each set.
improvementValue = entropy - ((entropySplit1 * size(irisSet1,1)/size(dataSet,1)) +
(entropySplit2 * size(irisSet2,1)/size(dataSet,1)));

disp('improvement on this split is')
disp(improvementValue);

maxSplit(improvementValue);

end
```

5. The max split function will loop through every value of every variable and calculate the best split that can be produced, this is found by the improvement value. This process was long but straight forward. A number of for loops were used to propagate through each value in each variable. Each time the split function was called in order to carry out the split which in turn called the entropy function and then the improvement value. The improvement value is returned to the maxSplit function and this is what was used to determine if the split was better or not. If the improvement was better the variable column, variable position (value) and the split sets were all stored as the new best split matrix. This process is repeated until all variable values have been rotated through, at this point the best possible split value, variable and sets have been produced.

```
function maxSplit(improvementValue)
%Must loop through each variable and each value of each variable and split
%on this and calculate the improvement (see improvement function).
%Substitute out the threshold value in the learnDecisionTree function for
%the value in that variable list, this is what will be split on. Do this
%for each value and keep record of only the max improvement value, others
%can be deleted. Use the unique function and sub out values to loop through
%in a for statement and then pass to improvement function to calculate the
%improvement in order to get the max improvement.
global variableNum;
global thresholdValue;
global variableSet;
global rowNumber;
global irisSet1;
global irisSet2;
global bestSplitSet1;
global bestSplitSet2;
global bestVariableSet;
global bestVariable;

%Create the array to store the max improvement value on first iteration
persistent improvementStorage
if(isempty(improvementStorage))
  disp('Improvement storage is not initialized');
  improvementStorage = zeros(1);
end

%Check the improvement value to determine if it is larger than the current
if(improvementValue > improvementStorage(1,1))
    improvementStorage(1,1)= improvementValue;
    disp('Improvement is better. New improvement is:')
    disp(improvementStorage(1,1));
    bestSplitSet1 = irisSet1;
    bestSplitSet2 = irisSet2;
    bestVariableSet = variableNum;
    bestVariable = variableSet(rowNumber,variableNum);
else
    disp('Improvement is not better. The improvement is still:')
    disp(improvementStorage(1,1));
end

%Loop through all values in each varibale of the dataSet
if (variableNum <= 4 && rowNumber <= size(variableSet,1))
    for n=1:size(variableSet,1)
        value = variableSet(rowNumber,variableNum);
        disp(rowNumber);
        rowNumber = rowNumber+1;
        thresholdValue = value;
        split(variableNum, variableSet, thresholdValue);
    end
else
    if(variableNum <= 4 && rowNumber > size(variableSet,1))
        variableNum = variableNum+1;
        rowNumber = 1;
        maxSplit(improvementValue);
else
```

```
            disp('Program finished max improvement is:');
            disp(improvementStorage(1,1));
            bestSplitting = [bestSplitSet1; bestSplitSet2; bestVariableSet; bestVariable];
            disp('Best sets, variable and value');
            disp(bestSplitting);
        end
    end
end

%Need to further split the now two split sets. The values for the max
%improvement split should be stored permanently in a new matrix i.e. the
%variable set, threshold value etc. There should be 3 nested loops to do
%this and should stop when there is no further splits. End result should be
%a matrix of all these values stated above. Summary need to loop through
%all split sets again doing the same and storing the values listed in
%another matrix.
```

6. The values stored from the maxSplit function above are then passed back to the very first function. These are then used to create the splitting rules for the decision tree. This is how the tree is created with this series of best splits to be carried out. As this becomes the decision tree it should be passed back to the original learnDescisionTree function to be used further.

7. The majorityClass function simply count the number of each classification in a given set. The set that is passed to this function is scanned through comparing strings with the different possible classifications in the data set. From here the majorityClass function show be applied to each new rule, therefore, this function should be used within the original learnDescisionTree function.

```
function MajorityClass(set)

%Count number of each classification in the dataSets
setosaCount = sum(strncmp('setosa',set,6));
virginicaCount = sum(strncmp('virginica',set,6));
versicolorCount = sum(strncmp('versicolor',set,6));



if setosaCount > virginicaCount && setosaCount > versicolorCount
    disp('Majority class is setosa');
    majorityClass = setosa;
end

if virginicaCount > setosaCount && virginicaCount > versicolorCount
    disp('Majority class is virginica');
    majorityClass = virginica;
end

if versicolorCount > setosaCount && versicolorCount > versicolorCount
     disp('Majority class is versicolor');
     majorityClass = versicolor;
end
end
```

## P2.2 Q-learning

1. The transition function is used to create the classic reinforcement learning grid. This function uses a series of if statements to determine the next state you will go to. This is done by taking the current state and an action. Depending on the state and action the resulting state is given. There is no if statement for state 2 as this is the goal state. If the action when in a state results in no change of state i.e. a wall is hit, the same state is returned and you stay in the current state.

```
1.  function TransistionFunction(state,action)
2.
3.  %TransistionFunction: building the environment for the transitions. Taking
4.  %an action and state and returning the next state from taking that action
5.  %from that current state.
6.
7.  global reward;
8.  global nextState;
9.
10. %Possibilities for state 1
11. if(state == 1)
12.     if(action == 1)
13.         nextState = 4;
14.     else
15.         nextState = state;
16.     end
17. end
18.
19. %Possibilities for state 3
20. if(state == 3)
21.     if(action == 1)
22.         nextState = 6;
23.     else
24.         nextState = state;
25.     end
26. end
27.
28. %Possibilities for state 4
29. if(state == 4)
30.     if(action == 1)
31.         nextState = 7;
32.     else
33.         if(action == 3)
34.             nextState = 1;
35.         else
36.             nextState = state;
37.         end
38.     end
39. end
40.
41. %Possibilities for state 5
42. %Action 3 from state 5 will result in the goal state (2) being achieved.
43. if(state == 5)
44.     if(action == 1)
45.         nextState = 9;
46.     else
47.         if(action == 3)
48.             nextState = 2;
49.         else
50.             nextState = state;
51.         end
52.     end
53. end
54.
55. %Possibilities for state 6
56. if(state == 6)
57.     if(action == 1)
58.         nextState = 11;
59.     else
60.         if(action == 3)
61.             nextState = 3;
62.         else
63.             nextState = state;
64.         end
65.     end
66. end
67.
```

```
%Possibilities for state 7
if(state == 7)
    if(action == 2)
        nextState = 8;
        else
        nextState = state;
    end
end

%Possibilities for state 8
if(state == 8)
    if(action == 2)
        nextState = 9;
    else
        if(action == 4)
            nextState = 7;
        else
            nextState = state;
        end
    end
end

%Possibilities for state 9
if(state == 9)
    if(action == 2)
        nextState = 10;
    else
        if(action == 4)
            nextState = 8;
        else
            if(action == 3)
                nextState = 5;
                    else
                        nextState = state;
                    end
            end
        end

        %Possibilities for state 10
        if(state == 10)
            if(action == 2)
                nextState = 11;
            else
                if(action == 4)
                    nextState = 9;
                else
                    nextState = state;
                end
            end
        end

        %Possibilities for state 11
        if(state == 11)
            if(action == 3)
                nextState = 6;
            else
                if(action == 4)
                    nextState = 10;
                else
                    nextState = state;
                end
            end
        end

        qLearning(state,action,nextState,reward);

        end
```

2. The Starting state is to return a random starting state except for the goal state of 2. To do this:

- A matrix of all the possible states was created.
- Random number was selected by choosing a random number from 1 to the length of the matrix that has the starting states.
- The random number is then used to select that position in the 1D matrix of starting states. This selection is then used as the starting state.

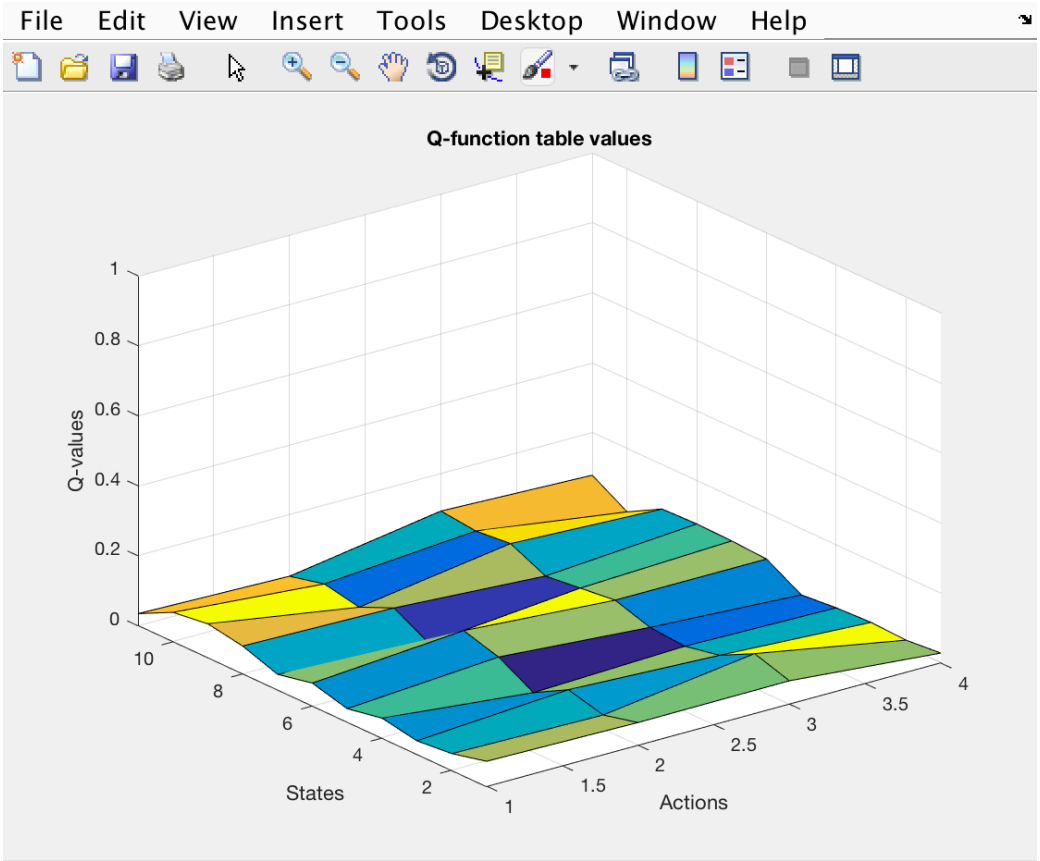
```
function StartingState
%STARTINGSTATE: Randomly choose a starting state (not 2)

global qTable;

%List of possible states.
startingStates = [1,3,4,5,6,7,8,9,10];

%Generate a random position.
position = randi(length(startingStates));

%Use the random position to select a state from the list.
state = startingStates(position);

eGreedyActionSelection(qTable,state);

end
```

3. The reward function is to take a state and an action and return 10 if the state is 5 and the action is three. This is due to the fact that if you are in state 5 and take action 3 (south) you will end up in the goal state. All other states and action combinations will get a reward value of 0. To do this a basic if statement was used where the provided state is checked to determine if it is 5, if so, the provided action is checked to determine if it is 3. If both of these are true the reward is 10, else the reward is 0.

```
function rewardFunction(state, action)
%REWARDFUNCTION: Determine the reward given for each state. Only award 10
%if in state 5 and action 3 is taken as this results in the goal state.

global reward;

if(state == 5)
    if(action == 3)
        reward = 10;
    else
        reward = 0;
    end
else
    reward = 0;
end

TransistionFunction(state,action);

end
```

4. This function of initQ is to create the initial Q-table of 11x4 with random values between 0.01 and 0.1. To do this the boundaries were set from 0.01 to 0.1 and a rand function was applied for these boundaries. The size of the rand function to populate was specified at 11x4. This Q-table is essential for the rest of the tasks as this is used as the storage and basis for all calculations. A surface plot was then created for this matrix by plotting the actions vs states vs Q-values. The limits of the axis were limited in order to scale the plot so it can be seen more clearly. The boundaries were set from 1 to 4 for actions (due to there being 4 possible actions), 1 to 10 for states (due to there being 10 states) and 0 to 1 for the Q-table in order to scale accordingly.

```
function initQ

%INITQ: Function to initial a new Q-table of random values between 0.01 and
%0.1 of size 11 by 4.
global qTable;
qTable = 0.01 + (0.1-0.01)*rand(11,4);

%Plot the results
figure
surf(qTable)
title('Q-function table values')
xlabel('Actions')
ylabel('States')
zlabel('Q-values')
axis([1,4,1,11,0,1]);

end
```



5. The greedy action selection function is to select the action with the highest Q value given Q-table. During this 90% of the time the highest Q-value should be returned and the remaining 10% of the time a random action should be returned. This is done to prevent a loop occurring, where the actions returned keep going back and forth in an unstoppable loop preventing progress. To do this I did the following:
    - Firstly, set the maxActionValue to 0 so this is the base and where it starts.
    - Declare a matrix to store the maxAction found and a matrix to use in the 10% random selection.
    - Use a for loop to loop through all options of the Q-Table provided
    - For a given state and action set the value to the value of that position in the Q-table.
    - If the value is greater than the current max value, then set the max value to this value and make a record of the action it is in.
    - Add one to the action to iterate through all actions.

    For the random selection decimals were used to determine the 90% and 10%. A random number was generated between 0 and 1 using the rand function and if this number fell below or equalled 0.9 the max action was returned, else a random action was returned.

```
function eGreedyActionSelection(qTable,state)

action = 1;
%initially have the value start at 0
maxActionValue = 0;
maxAction = zeros(1,1);
randomNum = rand(1,1);

%Loop through all actions for the state (1-4) provided finding the max
value
%action and storing this in an array.
for n=1:size(qTable,2)
    value = qTable(state,action);
        if(value>maxActionValue)
            maxActionValue = value;
            %This becomes the action with the highest value.
            maxAction(1,1) = action;
        end
    action = action+1;
end

%Used for percentage of the time. 90% of time return max value and 10% of
%the time return a random value.
if(randomNum <= 0.9)
    %Get the action from the maxAction matrix which will contain the action
    %with the higest value.
    rewardFunction(state,maxAction(1,1));
else
    randomAction = randi(length(qTable(2)));
    rewardFunction(state,randomAction);
end

end
```

6. The Q-learning update algorithm is to be applied to the provided values and the results should be an updated cell of the Q-table based on the algorithm. This function inserts the values given into the equation so with each iteration you are updating the values of random q-table. s' (resultingState) being the next state, a' being the next action, s (state) being the initial state and a (action) being the initial action. Goal is to update the values based on the rewards so the closer to the goal you get the closer to 10 you get. You update the cell of s' and a' with the new value. To start it will be shuffling around the values but with each iteration you start to propagate through updating the cells to get closer to the reward. To start reward value will be 10 or 0 (based on the reward function). In this case a temporal discount rate of 0.9 and a learning rate of 0.2 was used in the algorithm.

```
function qLearning(state,action,resultingState,reward)

global qTable;
global endTrialState;
%Use a temporal discount rate of 0.9 and a learning rate of 0.2
learningRate = 0.2;
temporalDiscount = 0.9;

%Implement the Q-learning algorithm using the values provided
qValue = qTable(state,action) + learningRate * (reward + temporalDiscount *
max(qTable(resultingState,:))-qTable(state,action));

qTable(state,action) = qValue;

endTrialState = resultingState;

end
```

7. The episode function is to implement a Q-learning episode using the algorithm in the previous task. It should choose a random start state and loop until the goal state is reached. The number of steps taken to reach the goal state should be stored. In the episode function, the trail is also taken into consideration allowing for 100 episodes to run per trial in this case. As long as the goal state has not been reached the eGreedy function is being called passing the new updated qTable and current state each time. For each iteration through this function the number of steps taken to reach the goal state is recorded in an ever-growing matrix called stepNumberStore.

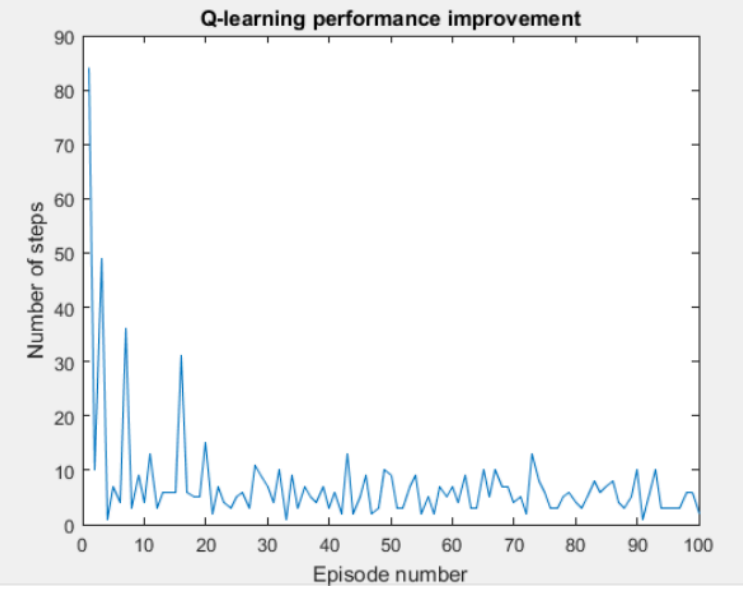
```
function Episode
%EPISODE: Run the number of episode specified by the trial and count and
%store the number of steps taken to reach the goal state from the random
%starting state.
global endTrialState;
global qTable;
global TrialNumber;
global stepNumberStore
global columnNum;

%Run 100 episodes in each trial.
while TrialNumber <= 100
stepNumber = 1;
StartingState();

%While the state is not the goal state.
while endTrialState ~= 2
    state = endTrialState;
    stepNumber = stepNumber + 1;
    eGreedyActionSelection(qTable,state);
end
%Store the number of steps taken to reach the goal state.
stepNumberStore(columnNum,TrialNumber) = stepNumber;
TrialNumber = TrialNumber + 1;
end
end
```

8. The next step up from the episode is the Trial. Each trial will run a certain number of episodes, in this case this number is 100. The trial function will initialize the Q-table for each trial and then use this table throughout the trial. The matrix that was storing the steps for each episode is then used here to plot the results. As can be seen from the results the number of steps for the first episode is a large number. With each episode, the number of steps taken quickly drops. This is due to the Q-table that is being shared between each episode and updated every time. By doing this the goal state is reached significantly faster resulting in a fewer step number needed.

```
function Trial
%TRIAL: Ensure the Q-table is initialized again for each new Trial and run
%episodes while keeping track of the storage of the results.

global TrialNumber
TrialNumber = 1;

%Position used to store the number of steps.
global columnNum;

initQ();

Episode();
columnNum = columnNum + 1;

end
```

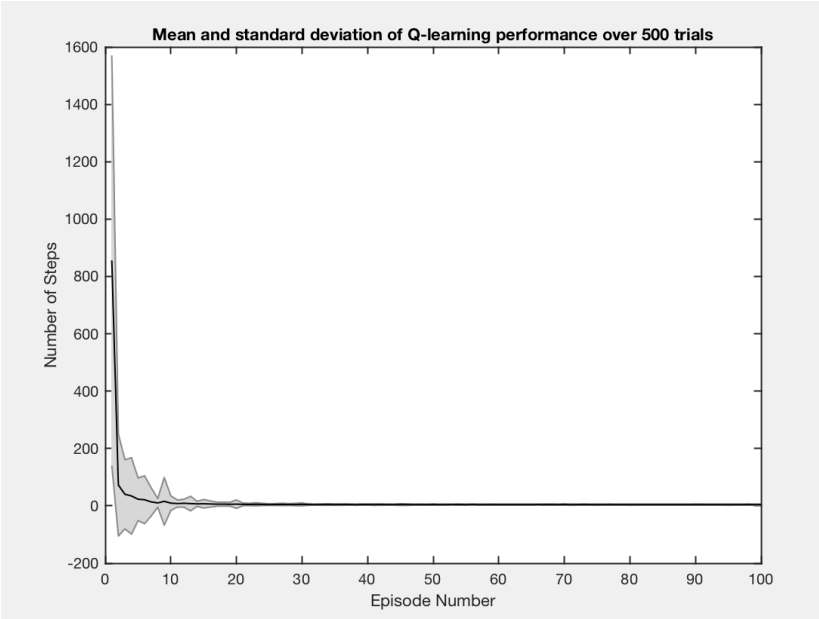
## Code to Plot the results

```
figure
plot(stepNumberStore);
title('Q-learning performance Improvement')
xlabel('Episode Number')
ylabel('Number of Steps')
```



9. The highest most level is the Experiment. Each experiment is to run 500 trials with each trial running 100 episodes, 50000 episodes in total. For each loop through the standard deviation and mean are calculated which is used to plot a shaded error bar graph. As can been seen from the code this is done in the inside the while loop. This loop also keeps track of how many trials have been run and will stop once 500 have been run. The stores for the mean and standard deviation are declared here. The function of shadedErrorBar is called in order to plot the results, the results can also be seen below. These results show that with each episode iteration the number of steps taken is less and less resulting in a faster process of getting to the end goal state. This downwards trend continues until the minimum number of steps needed to reach the goal state has been reach. At this point the plot then levels off and stays constant.

```
function Experiment
%Experiment: Run 500 trials and calculate the standard deviation and mean for
%these trials.

global ExperimentNumber
ExperimentNumber = 1;

global stepNumberStore
stepNumberStore = [];

global columnNum
columnNum = 1;

meanStore = [];
stdStore = [];

%Run 500 trials per experiment.
while ExperimentNumber <= 500
    Trial();
    stdStore = std(stepNumberStore);
    meanStore = mean(stepNumberStore);
    ExperimentNumber = ExperimentNumber + 1;
end

%Plot a shaded error bar graph of the results.
shadedErrorBar([1:100],meanStore,stdStore);
title('Mean and standard deviation of Q-learning performance over 500 trials')
xlabel('Episode Number')
ylabel('Number of Steps')

end
```



## P2.3 NSM-learning

1. The transition function is used to create the classic reinforcement learning grid. This function uses a series of if statements to determine the next state you will go to. This is done by taking the current state and an action. Depending on the state and action the resulting state is given. There is no if statement for state 2 as this is the goal state. If the action when in a state results in no change of state i.e. a wall is hit, the same state is returned and you stay in the current state.

```
function Transistion(state,action)

%Transistion: building the environment for the transitions. Taking
%an action and state and returning the next state from taking that action
%from that current state.

global nextState;

%Possibilities for state 1
if(state == 1)
    if(action == 1)
        nextState = 4;
    else
        nextState = state;
    end
end

%Possibilities for state 3
if(state == 3)
    if(action == 1)
        nextState = 6;
    else
        nextState = state;
    end
end

%Possibilities for state 4
if(state == 4)
    if(action == 1)
        nextState = 7;
    else
        if(action == 3)
            nextState = 1;
        else
            nextState = state;
        end
    end
end

%Possibilities for state 5
%Action 3 from state 5 will result in the goal state (2) being achieved.
if(state == 5)
    if(action == 1)
        nextState = 9;
    else
        if(action == 3)
            nextState = 2;
        else
            nextState = state;
        end
    end
end

%Possibilities for state 6
if(state == 6)
    if(action == 1)
        nextState = 11;
    else
        if(action == 3)
            nextState = 3;
        else
```

```
            nextState = state;
        end
    end
end

%Possibilities for state 7
if(state == 7)
    if(action == 2)
        nextState = 8;
        else
        nextState = state;
    end
end

%Possibilities for state 8
if(state == 8)
    if(action == 2)
        nextState = 9;
    else
        if(action == 4)
            nextState = 7;
        else
            nextState = state;
        end
    end
end

%Possibilities for state 9
if(state == 9)
    if(action == 2)
        nextState = 10;
    else
        if(action == 4)
            nextState = 8;
        else
            if(action == 3)
                nextState = 5;
            else
                nextState = state;
            end
        end
    end
end

%Possibilities for state 10
if(state == 10)
    if(action == 2)
        nextState = 11;
    else
        if(action == 4)
            nextState = 9;
        else
            nextState = state;
        end
    end
end

%Possibilities for state 11
if(state == 11)
    if(action == 3)
        nextState = 6;
    else
        if(action == 4)
            nextState = 10;
        else
            nextState = state;
        end
    end
end

end
```

1. This continuation of task 1 has another function be created and it is a function that creates a POMP version of McCallum's grid-world. It gets the state and will return an observation. Due to the fact that there are multiple states with the same observation we cannot know which state we are in. Therefore, all possible states are listed for each observation. This is done with a series of if statements and defined the possible states for each observation with observation being the output.

```
function Observations(state)

global observation;

if state == 1 || state == 2 || state == 3
    observation = 14;
end

if state == 4 || state == 5 || state == 6
    observation = 10;
end

if state == 7
    observation = 9;
end

if state == 8 || state == 10
    observation = 5;
end

if state == 9
    observation = 1;
end

if state == 11
    observation = 3;
end

end
```

1. As a part of task one a third function is to be creates. The rndStartState is to return a random starting state except for the goal state of 2. To do this:
    - A matrix of all the possible states was created.
    - Random number was selected by choosing a random number from 1 to the length of the matrix that has the starting states.
    - The random number is then used to select that position in the 1D matrix of starting states. This selection is then used as the starting state.

```
function rndStartState

%List of possible states.
startingStates = [1,3,4,5,6,7,8,9,10];

%Generate a random position.
position = randi(length(startingStates));

%Use the random position to select a state from the list.
startState = startingStates(position);

rndEpisode(startState);

end
```

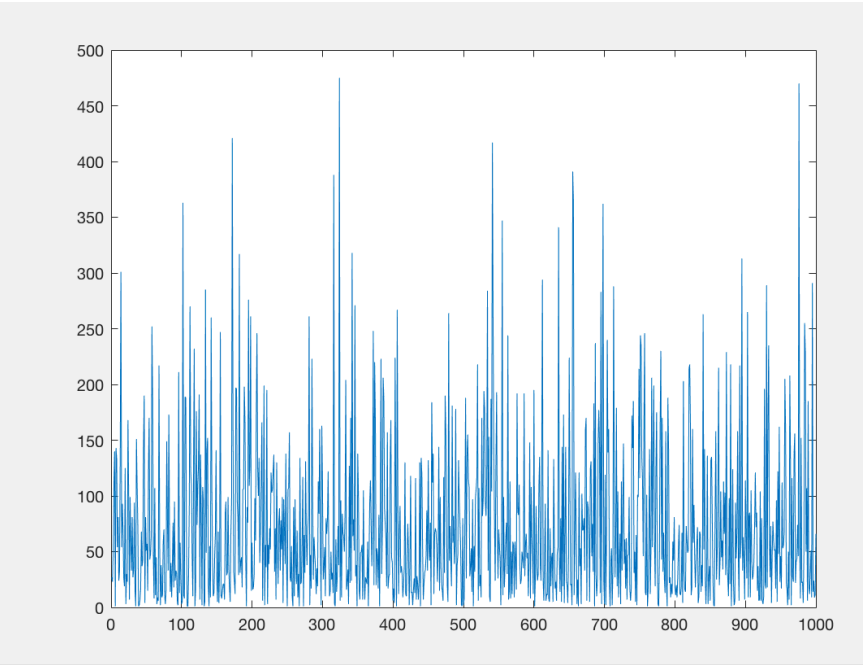
2. For the rndEpisode function the objective was to generate a series of observations, actions and discounted reward values. It will take the start state generated by rndStartState and use this to find the observation, action and reward. This state is passed to the other functions already created i.e. the transition function, with a random action to get the next state. Once a matrix has been made of all these values the rows needs to be discounted, starting with the last and working back. This was done using a for loop to back propagate through the result to apply the discount. The final part of this function was to limit the number of rows to 20, if there was more than 20, the last 20 were simply taken from the matrix. If there are less than the rows before this set were to be set to zero to make it up to 20. To do this I create a 20x3 matrix of zeros (the max size that is needed) and then took away the size of the matrix i.e. if the matrix had a length of 5 then 20 – 5 = 15 which is how many rows of zeros that are needed. Once I had this I took the 15 rows from the zero matrix and concatenated them with the episode matrix. This was then added to the LTM at the appropriate episode.

```
function rndEpisode(startState)

global observation;
global nextState;
global stepStore;

episode = [];
state = startState;
rowNum = 1;
stepCount = 0;
global LTM;
global iterationNumber;

while state ~= 2
    %List of possible actions.
    actions = [1,2,3,4];

    %Generate a random position.
    position = randi(length(actions));

    %Use the random position to select a action from the list.
    action = actions(position);

    %Get observations and the next state
    Observations(state);
    Transistion(state,action);
    state = nextState;

    %Write results to episode matrix
    episode(rowNum,1) = observation;
    episode(rowNum,2) = action;
    episode(rowNum,3) = 1;

    rowNum = rowNum + 1;
    stepCount = stepCount + 1;
end

disp('state 2 has been reached!!');
episode(size(episode,1),3) = 10;

%For discount you do last is 10 then one before is 10 * 0.9 and one before
%that is the next one times 0.9. e.g. last = 10 second last = 10 * 0.9 = 9 third
last = 9 * 0.9 = 8.1 etc.
for n= 1:size(episode,1)-1
    episode(size(episode,1)-n,3) = episode(size(episode,1)-(n-1),3) * 0.9;
```

```
end

stepStore(1,iterationNumber) = stepCount;

%Create a matrix of zeros with size of this being 20 - size of episode e.g.
%episode has 15 rows matrix of zeros will be 20-15=5. Concat this (with zero matrix
first)
%to populate the beginning of episode with zeros so its size comes to (20,3)
if size(episode,1)<20
    zeroMatrix = zeros(20-size(episode,1),3);
    episode = [zeroMatrix;episode];
    last20Steps = episode;
else
    last20Steps = episode(size(episode,1)-19:size(episode,1),:);
end

%Set the LTM matrix at the specific iteration to the last 20 steps.
LTM(:,:,iterationNumber) = last20Steps;

end
```

3. The rndTrials creates a 3D matrix made up of the last 20 steps matrix created in the previously task. This function will take a value, being the number of episodes wanted, and will run the rndEpisode function to reach that number of episodes. This is done in a while loop as indicated with the code below. The number of steps taken per episode is recorded into another matrix. This becomes the output for this function, as a result of this a graph of steps against episode can be plotted. As stated this was done for 1000 random episodes with the graph being seen below. The graph doesn't show much improvement in the number of steps as it progresses, which is expected due to the fact that it is random episodes that are being run.

```
function [output] = rndTrial(numberOfEpisodesAdded)

%This should take the number of episodes to run and therefore the number of
%results sets of 20x3 to add to the 3D matrix.
global LTM
LTM = zeros(20,3,numberOfEpisodesAdded);

global iterationNumber
iterationNumber = 1;

global stepStore
stepStore = zeros(1,numberOfEpisodesAdded);


%Store results of the 20x3 matrix in the LTM for each iteration. Each
%episode will generate a new 20X3 matrix which should be added to LTM. When
%the LTM is called with (:,:,3) it should return the 3rd episodes results
%matrix (3rd episodes 20x3 matrix). See diagram in notepad for visual.

while iterationNumber <= numberOfEpisodesAdded
    rndStartState();
    iterationNumber = iterationNumber + 1;
end

output = stepStore;

KNearest();
end
```



4. The basic function of the proximity function is to count the number of steps are the same in the STM as are in the LTM. It takes in a series of values including LTM, index of episode in LTM, step in episode, STM and observation value. Using this we can compare the LTM to the STM. The matches are initially done on the observation values in the LTM and STM at the specified point. The comments below explain in detail what each part of the code does but essentially what I did was loop through each matrix (LTM and STM) and worked backwards through them matching the value. If there was a match, then the row in the LTM was stored in a new matrix which in this case is called proximityAndMatchVector with the proximity also getting stored and incremented by 1. If there is one mismatch, then this process terminated and the program moves on.

```
function proximity(LTM, EpisodeLTM, StepInEpisode, STM, ObservationValue)
global proximityValue
proximityValue = 0.0;

global proximityAndMatchVector
proximityAndMatchVector = [];

rowNum = 1;

currentObservation = LTM(StepInEpisode,1,EpisodeLTM);

%If the observation in the LTM specified by the input is the same as the
%observation specified then its a match
if currentObservation == ObservationValue
    disp('Match');
    proximityValue = 1;
    %get the last row of the STM for an index
    StmIndex = size(STM,1);
    disp(StmIndex);
    %store the match in a vector
    proximityAndMatchVector(rowNum,1) = proximityValue;
    proximityAndMatchVector(rowNum,2) = LTM(StepInEpisode,1,EpisodeLTM);
    proximityAndMatchVector(rowNum,3) = LTM(StepInEpisode,2,EpisodeLTM);
    proximityAndMatchVector(rowNum,4) = LTM(StepInEpisode,3,EpisodeLTM);
    rowNum = rowNum +1;
    %Do this so it stops once it hits the first row of the LTM
    while StepInEpisode >= 1
    %if the observations and action values match for both the LTM and STM
    %row then add one to proximity
    if LTM(StepInEpisode,1,EpisodeLTM) == STM(StmIndex,1) && LTM(StepInEpisode,2,EpisodeLTM)
== STM(StmIndex,2)
        proximityValue = proximityValue + 1;
        proximityAndMatchVector(rowNum,1) = proximityValue;
        proximityAndMatchVector(rowNum,2) = LTM(StepInEpisode,1,EpisodeLTM);
        proximityAndMatchVector(rowNum,3) = LTM(StepInEpisode,2,EpisodeLTM);
        proximityAndMatchVector(rowNum,4) = LTM(StepInEpisode,3,EpisodeLTM);
        %take one away from each index to move back through STM and LTM
        StepInEpisode = StepInEpisode - 1;
        StmIndex = StmIndex - 1;
        rowNum = rowNum + 1;
    else
        %if there is a mismatch terminate the loop.
        disp('Terminate process')
        StepInEpisode = 0;
    end
    end
end
disp('Proximity : ')
disp(proximityValue)
disp('The final output Vector is : ')
disp(proximityAndMatchVector);

end
```

5. The KNearest function calls the proximity function for all the steps stored that were a match, in my case this is proximityAndMatchVector. I then had it loop through every episode and every step in the LTM and get the proximity for each and store this value. As the proximity is incremented by 1 in the proximity function I checked if the returned proximity value was greater than 0, if it was each part of the returned entire proximity matrix was copied into the nearest steps matrix. Once this was done I then added in logic for keeping the size of the nearest steps matrix the size of 10, do this I first checked if it was already at size 10, if not the proximity matrix was added to it until the size was 10. Upon this happening I used the min function in Matlab and the proximity function that the program was on. I compare the two and if the proximity value is greater than the min value that is already stored this row was replaced. This is done simply by finding the row that the min value is on and overwriting it with the value that is larger. This process was repeated until all episodes had been run and the end results in a 20 row matrix of nearest steps.

```
function KNearest

global proximityValue
global LTM
global proximityAndMatchVector

EpisodeLTM = 1;
StepInEpisode = 1;

nearestSteps = [];
nearestStepsRowNum = 1;
proximityAndMatchVectorRowNum = 1;

%for every episode in LTM
while EpisodeLTM <= size(LTM,3)
    %for every step in episode
    while StepInEpisode <= 20
        STM = LTM(:,:,EpisodeLTM);
        stmObservationValue = LTM(StepInEpisode,1,EpisodeLTM);
        %pass to proximity the LTM, step to start, 20x3 STM (current episode)
        %and the observation of this step in LTM
        proximity(LTM, EpisodeLTM, StepInEpisode,STM, stmObservationValue);
        StepInEpisode = StepInEpisode + 1;

        if proximityValue > 0
            if size(nearestSteps,1) == 10
                while proximityAndMatchVectorRowNum <= size(proximityAndMatchVector,1)
                    %find min proximity value in nearest steps
                    minValue = min(nearestSteps(:,1));
                    %get the proximity of the row stated from proximityAndMatchVector
                    possibleReplaceProximity =
proximityAndMatchVector(proximityAndMatchVectorRowNum,1);
                    if possibleReplaceProximity > minValue
                        %get the row the min nearestSteps proximity is in
                        [row, column] = find(nearestSteps == minValue);
                        %replace the nearestStep row with the values in the
                        %proximityAndMatchVector for that row.
                        nearestSteps(row,1) =
proximityAndMatchVector(proximityAndMatchVectorRowNum,1);
                        nearestSteps(row,2) =
proximityAndMatchVector(proximityAndMatchVectorRowNum,2);
                        nearestSteps(row,3) =
proximityAndMatchVector(proximityAndMatchVectorRowNum,3);
                        nearestSteps(row,4) =
proximityAndMatchVector(proximityAndMatchVectorRowNum,4);
                    end
                    %iterate through all rows in the proximityAndMatchVector
                    proximityAndMatchVectorRowNum = proximityAndMatchVectorRowNum + 1;
                end
                %reset the row num for the next call.

            else
                %loop through each value in the proximity vector while
                %nearestStep is less than 10
```

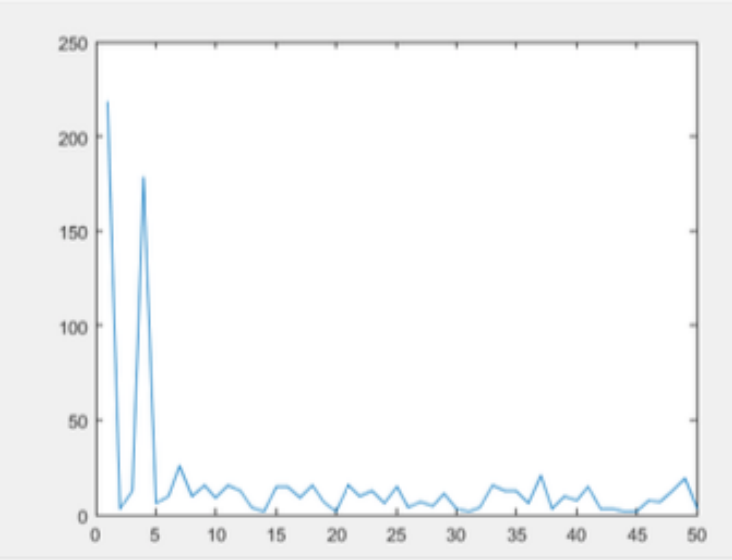
```
                while size(nearestSteps,1) < 10 && proximityAndMatchVectorRowNum <=
size(proximityAndMatchVector,1)
                %if any iteration has any matches resulting in proximities more
                %than one then store the values.
                nearestSteps(nearestStepsRowNum,1) =
proximityAndMatchVector(proximityAndMatchVectorRowNum,1);
                nearestSteps(nearestStepsRowNum,2) =
proximityAndMatchVector(proximityAndMatchVectorRowNum,2);
                nearestSteps(nearestStepsRowNum,3) =
proximityAndMatchVector(proximityAndMatchVectorRowNum,3);
                nearestSteps(nearestStepsRowNum,4) =
proximityAndMatchVector(proximityAndMatchVectorRowNum,4);
                nearestStepsRowNum = nearestStepsRowNum +1;
                proximityAndMatchVectorRowNum = proximityAndMatchVectorRowNum +1;
                end
                proximityAndMatchVectorRowNum = 1;
                disp('LESS THAN 10')
                disp(nearestSteps);
            end
            proximityAndMatchVectorRowNum = 1;
        end
    end
    EpisodeLTM = EpisodeLTM + 1;
    %Reset the step number to 1 for next episode
    StepInEpisode = 1;
end

disp('Knearest vector is :')
disp(nearestSteps);
end
```

6. In order to prepare for the NSM action select a few current functions were copied and changed. These included the rndTrial and rndEpisode which code can be seen below. A new Action Selection function was created where instead of taking a random action, as before, a calculated action is selected. This is done by discounting the KNearest matrix created in the previous task. 10% of the time a random action is returned though, this is to prevent looping where it goes back and forth between two actions and makes no progress.

```
function NSMTrial(numberOfEpisodesAdded)
%This should take the number of episodes to run and therefore the number of
%results sets of 20x3 to add to the 3D matrix.
global LTM
LTM = zeros(20,3,numberOfEpisodesAdded);

global iterationNumber
iterationNumber = 1;

global stepStore
stepStore = zeros(1,numberOfEpisodesAdded);


%Store results of the 20x3 matrix in the LTM for each iteration. Each
%episode will generate a new 20X3 matrix which should be added to LTM. When
%the LTM is called with (:,:,3) it should return the 3rd episodes results
%matrix (3rd episodes 20x3 matrix). See diagram in notepad for visual.

while iterationNumber <= numberOfEpisodesAdded
    rndStartState();
    iterationNumber = iterationNumber + 1;
end

NSMEpisode(LTM);

End


function NSMEpisode(LTM)

global observation;
global stepStore;

episode = [];
stepCount = 0;
global iterationNumber;

NSMSelectAction(LTM,STM,observation);

disp('state 2 has been reached!!');
episode(size(episode,1),3) = 10;

%For discount you do last is 10 then one before is 10 * 0.9 and one before
%that is the next one times 0.9. e.g. last = 10 second last = 10 * 0.9 = 9 third last = 9 *
0.9 = 8.1 etc.
for n= 1:size(episode,1)-1
    episode(size(episode,1)-n,3) = episode(size(episode,1)-(n-1),3) * 0.9;
end

disp('Number of steps taken = ')
disp(stepCount);
stepStore(1,iterationNumber) = stepCount;

%Create a matrix of zeros with size of this being 20 - size of episode e.g.
%episode has 15 rows matrix of zeros will be 20-15=5. Concat this (with zero matrix first)
%to populate the beginning of episode with zeros so its size comes to (20,3)
if size(episode,1)<20
    zeroMatrix = zeros(20-size(episode,1),3);
    disp(zeroMatrix);
    episode = [zeroMatrix;episode];
```

```
    last20Steps = episode;
else
    last20Steps = episode(size(episode,1)-19:size(episode,1),:);
end
LTM(:,:,iterationNumber) = last20Steps;

end



function NSMSelectAction(LTM, STM, observation)

randomNum = rand(1,1);

% 10% of the time get a random aciton.
if(randomNum >= 0.9)
while state ~= 2
    %List of possible actions.
    actions = [1,2,3,4];

    %Generate a random position.
    position = randi(length(actions));

    %Use the random position to select a action from the list.
    action = actions(position);

    %Get observations and the next state
    Observations(state);
    Transistion(state,action);
    state = nextState;

    %Write results to episode matrix
    episode(rowNum,1) = observation;
    episode(rowNum,2) = action;
    episode(rowNum,3) = 1;

    rowNum = rowNum + 1;
    stepCount = stepCount + 1;
end
    rewardFunction(state,maxAction(1,1));
end


end
```

7. From plotting the 50 trials it can be seen that there is an improvement. The number of steps required drops significantly as the trial progresses through the episode. This a contrast to the previous graph of the random episodes which were taking the random actions. In this case the actions are being calculated with the best being used. To start there are still a lot of steps due to the fact the it is starting out so has nothing to learn from, but as it starts to learn it efficiency improves greatly.




8. When experiments are implemented in a similar fashion as a previous part of the assignment the improvement is similar. As the experiment progresses, and the shaded error bar is plotted, the number of steps taken greatly reduces. This continue to fall until the limit is reached. This limit being that the minimum number of steps required to take has been found and therefore the optimal route and number of steps to take is always being used.