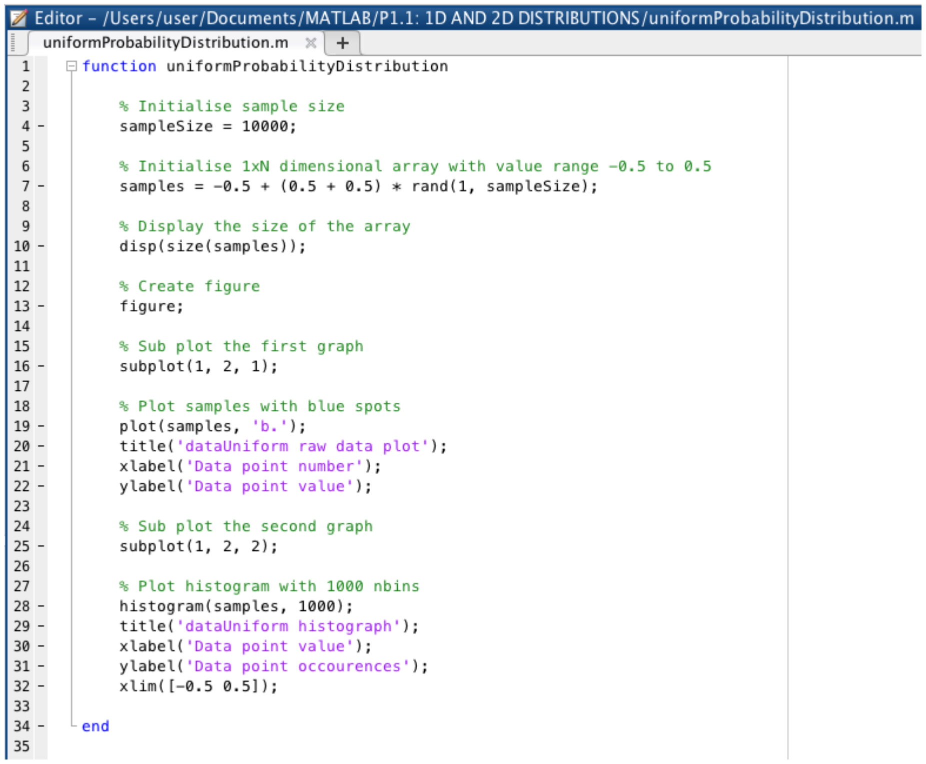
AINT351 - Ian Howard Lab Journel

P1.1 1D and 2D distributions

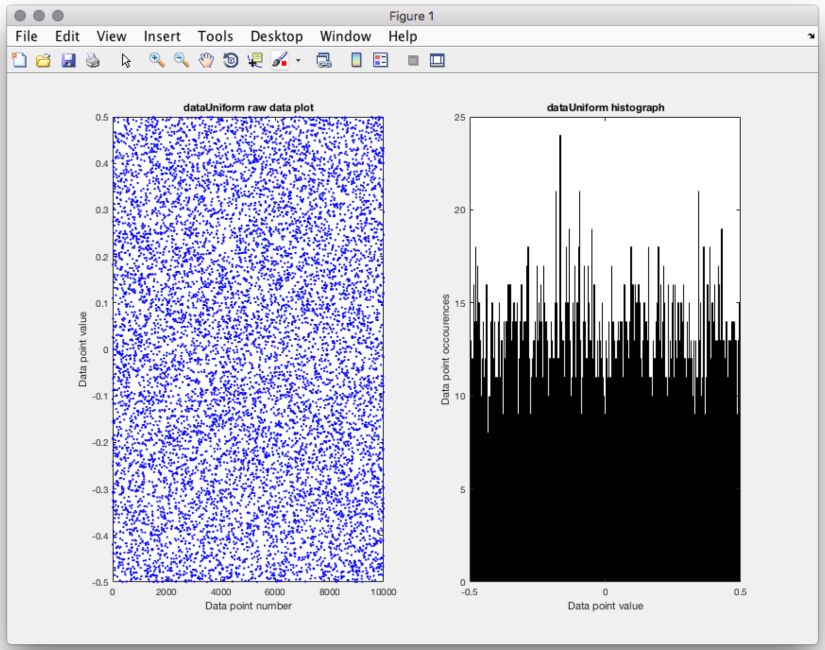
1. Generate a uniform probability distribution



Editor – /Users/user/Documents/MATLAB/P1.1: 1D AND 2D DISTRIBUTIONS/uniformProbabilityDistribution.m

uniformProbabilityDistribution.m

```
function uniformProbabilityDistribution

    % Initialise sample size
    sampleSize = 10000;

    % Initialise 1xN dimensional array with value range -0.5 to 0.5
    samples = -0.5 + (0.5 + 0.5) * rand(1, sampleSize);

    % Display the size of the array
    disp(size(samples));

    % Create figure
    figure;

    % Sub plot the first graph
    subplot(1, 2, 1);

    % Plot samples with blue spots
    plot(samples, 'b.');
    title('dataUniform raw data plot');
    xlabel('Data point number');
    ylabel('Data point value');

    % Sub plot the second graph
    subplot(1, 2, 2);

    % Plot histogram with 1000 nbins
    histogram(samples, 1000);
    title('dataUniform histograph');
    xlabel('Data point value');
    ylabel('Data point occourences');
    xlim([-0.5 0.5]);

end

```

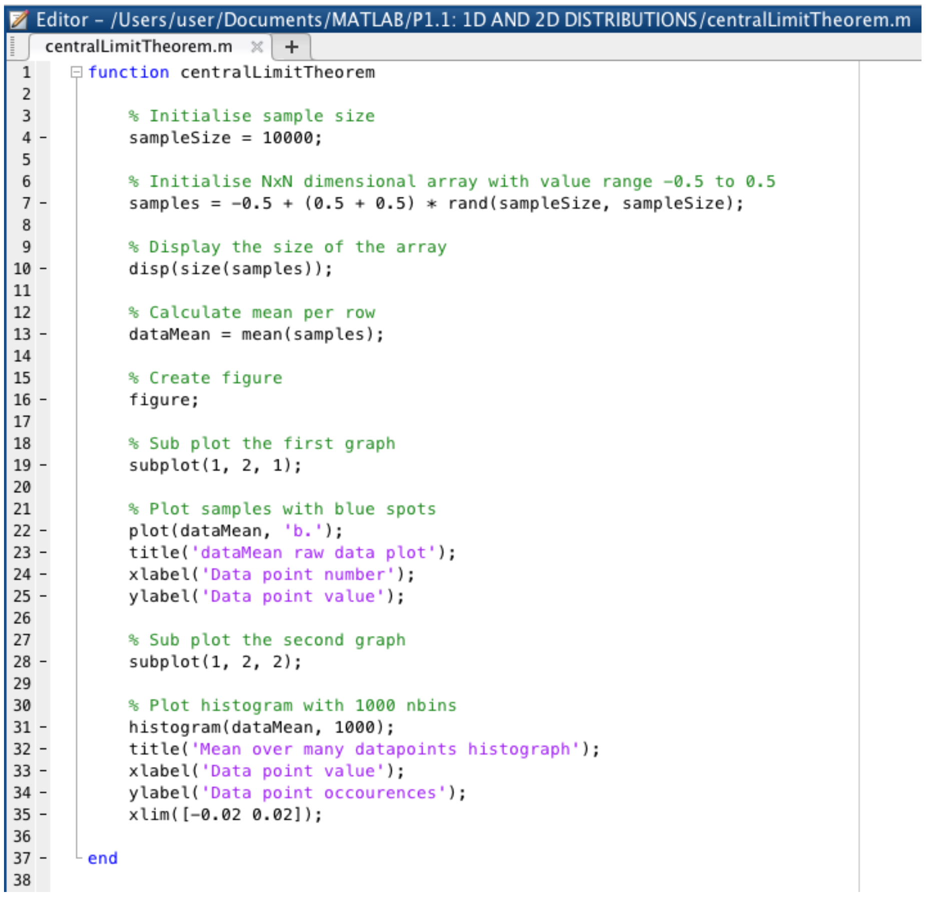


The task was to create a uniform probability distribution graph which demonstrates the theory that all data points that are equal in number of occurrences, have an equal probability to be chosen. This is partly because we fill the matrix with data, no other calculations, therefore all numbers is likely to appear the same amount of times.

Changing the number of bins increases the size of the intervals on the x-axis, meaning that the data point values have a larger group to fall in. This pushes the occurrence of each bin much higher the less bins there are. If I were to change the bin from 1000 to 100 then the data point occurrence range goes from the maximum of 25 to 140. This, effectively, decreasing the accuracy. It also means that the peaks and dips of the ranges have a greater difference between them.

The number of samples that have been used has been consistent throughout all graph generations, standing at 10,000 values. This is large enough for there it to have a good range of values even at lower levels of bins but not high enough so that the generation of the graph takes too long nor that the graph is over saturated with values that are unnecessary to prove the point of the graph.

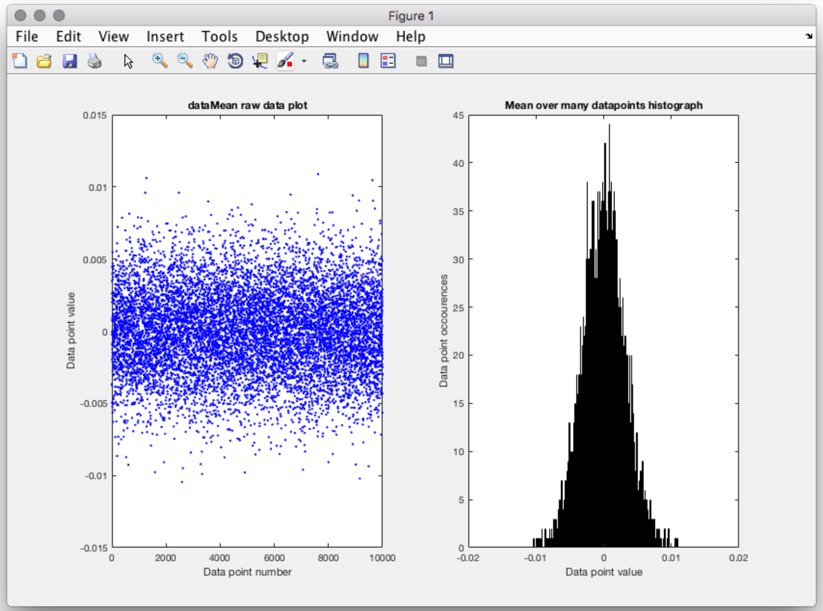
2. The central limit theorem



Editor – /Users/user/Documents/MATLAB/P1.1: 1D AND 2D DISTRIBUTIONS/centralLimitTheorem.m

centralLimitTheorem.m

```
function centralLimitTheorem

    % Initialise sample size
    sampleSize = 10000;

    % Initialise NxN dimensional array with value range -0.5 to 0.5
    samples = -0.5 + (0.5 + 0.5) * rand(sampleSize, sampleSize);

    % Display the size of the array
    disp(size(samples));

    % Calculate mean per row
    dataMean = mean(samples);

    % Create figure
    figure;

    % Sub plot the first graph
    subplot(1, 2, 1);

    % Plot samples with blue spots
    plot(dataMean, 'b.');
    title('dataMean raw data plot');
    xlabel('Data point number');
    ylabel('Data point value');

    % Sub plot the second graph
    subplot(1, 2, 2);

    % Plot histogram with 1000 nbins
    histogram(dataMean, 1000);
    title('Mean over many datapoints histograph');
    xlabel('Data point value');
    ylabel('Data point occourences');
    xlim([-0.02 0.02]);

end

```

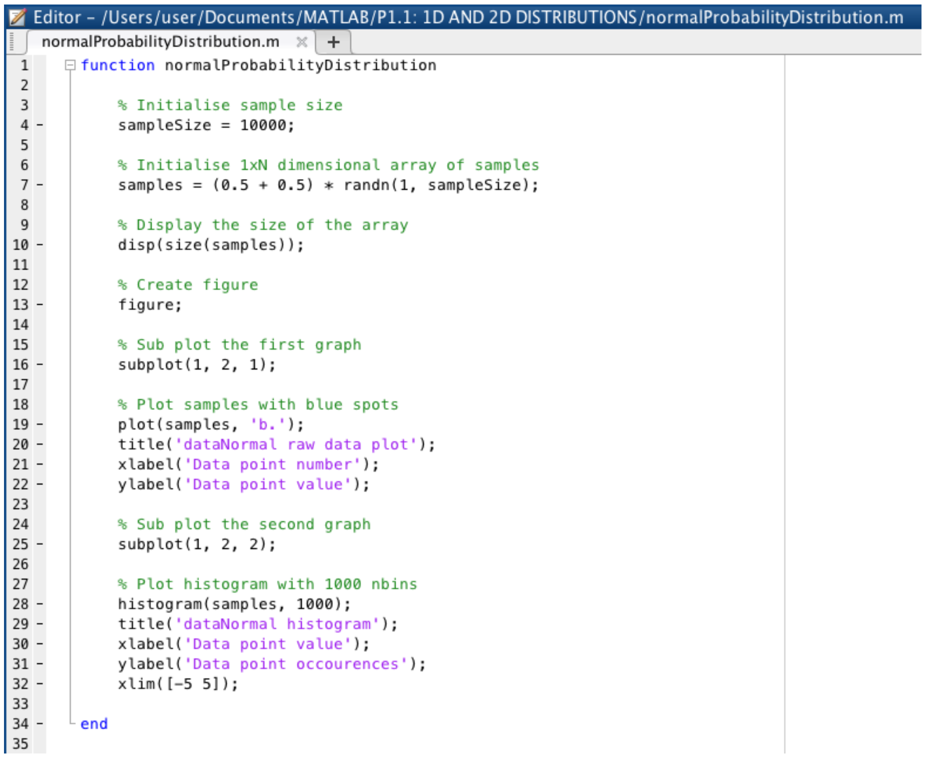


The task was to create a central limit theorem graph which demonstrates the theorem that the mean of a large number of iterations of data that are independent, will approximately be normally distributed. That is, no matter data distribution, the mean will gravitate toward the middle point of the data and will have a greater probability than that of the data closer to the limits of data boundaries.

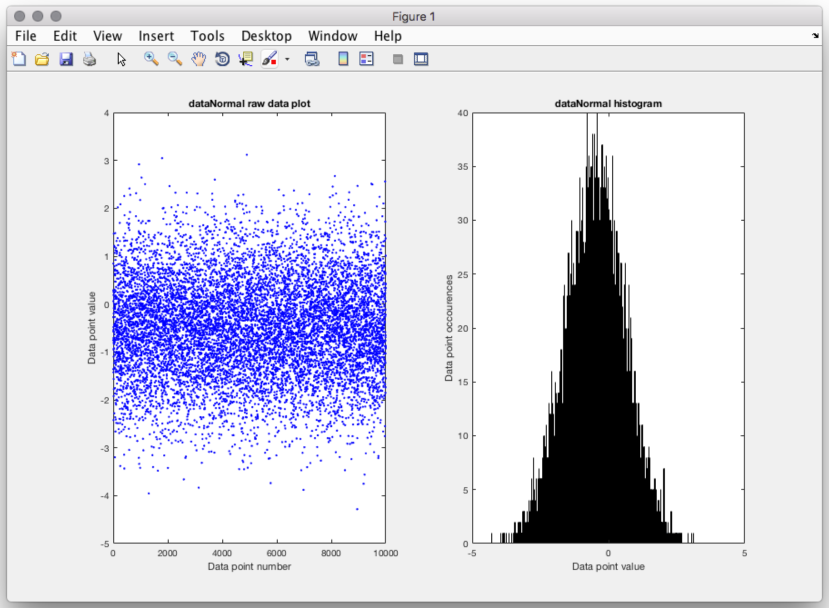
Increasing the size of data used increases the height of the centre of the distribution, where as decreasing not only shortens the height of the centre but widens the standard deviation from the centre of the distribution. Keeping the same number of samples but decreasing the number of bins from 1000 to 10 dramatically changes the appearance of the graph. Although the rise of the curve from outer limits to the centre limit is already steep, decreasing the number of bins only emphasises that feature. As always, it pushes the data occurrences limit up greatly due to there being more chance of falling in a bin.

I think that 10,000 adequately show the purpose of the graph to a good degree of accuracy.

3. Generate a normal probability distribution


Editor – /Users/user/Documents/MATLAB/P1.1: 1D AND 2D DISTRIBUTIONS/normalProbabilityDistribution.m
normalProbabilityDistribution.m

```
function normalProbabilityDistribution

    % Initialise sample size
    sampleSize = 10000;

    % Initialise 1xN dimensional array of samples
    samples = (0.5 + 0.5) * randn(1, sampleSize);

    % Display the size of the array
    disp(size(samples));

    % Create figure
    figure;

    % Sub plot the first graph
    subplot(1, 2, 1);

    % Plot samples with blue spots
    plot(samples, 'b.');
    title('dataNormal raw data plot');
    xlabel('Data point number');
    ylabel('Data point value');

    % Sub plot the second graph
    subplot(1, 2, 2);

    % Plot histogram with 1000 nbins
    histogram(samples, 1000);
    title('dataNormal histogram');
    xlabel('Data point value');
    ylabel('Data point occourences');
    xlim([-5 5]);

end

```

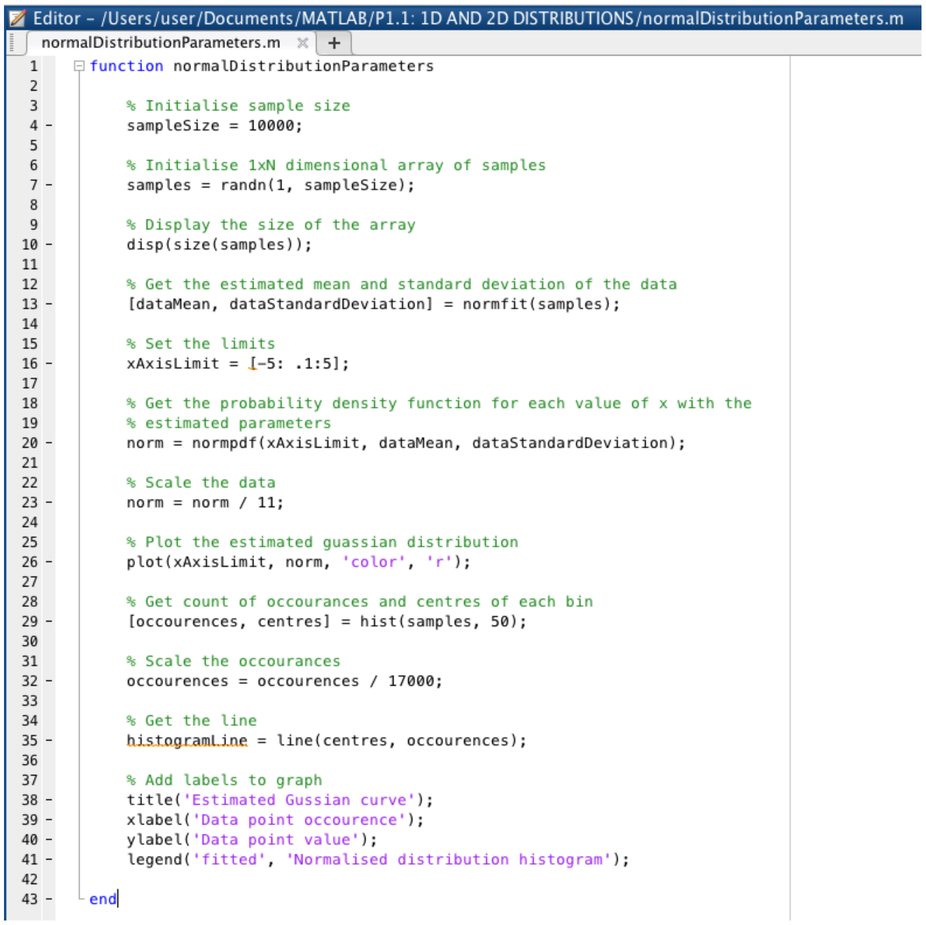


The task was to create a normal probability distribution graph which demonstrates that the data mean is greatest at its median. Many categories of data fall into this distribution such as average height, average weight, etc. This graph was generated like this because of the use of randn function, which generates randomly normally distributed numbers.

If I were to double the amount of samples used from 10,000 to 20,000, the occurrence of the data at the median greatly increases, almost doubles in fact, but the graph does not widen. The same is for the decreasing of the number of bins, it does not widen the graph, but only heighten the peak at the median. This is because I've restricted the range in which the random numbers were generated in to 5 and -5, so it only further proves that adding more data to such distribution only strengthens the concept that it the mean of it all is falls close to the median.

I believe that between 10,000 and 20,000 is more than enough data points to show the distribution's main characteristics.

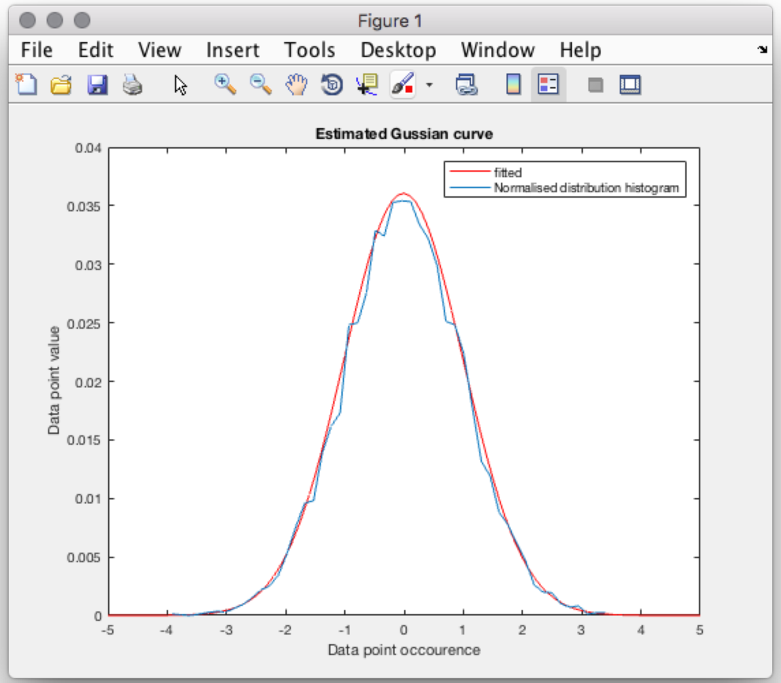
4. Estimate a normal distribution parameters



Editor – /Users/user/Documents/MATLAB/P1.1: 1D AND 2D DISTRIBUTIONS/normalDistributionParameters.m

normalDistributionParameters.m

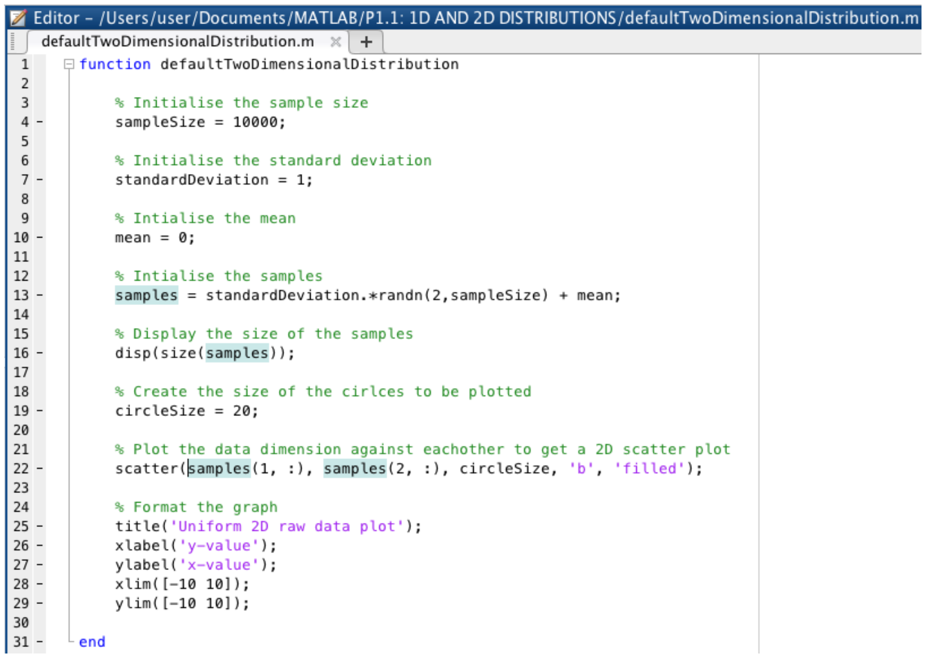
```
function normalDistributionParameters

    % Initialise sample size
    sampleSize = 10000;

    % Initialise 1xN dimensional array of samples
    samples = randn(1, sampleSize);

    % Display the size of the array
    disp(size(samples));

    % Get the estimated mean and standard deviation of the data
    [dataMean, dataStandardDeviation] = normfit(samples);

    % Set the limits
    xAxisLimit = [-5: .1:5];

    % Get the probability density function for each value of x with the
    % estimated parameters
    norm = normpdf(xAxisLimit, dataMean, dataStandardDeviation);

    % Scale the data
    norm = norm / 11;

    % Plot the estimated guassian distribution
    plot(xAxisLimit, norm, 'color', 'r');

    % Get count of occourances and centres of each bin
    [occourences, centres] = hist(samples, 50);

    % Scale the occourances
    occourences = occourences / 17000;

    % Get the line
    histogramLine = line(centres, occourences);

    % Add labels to graph
    title('Estimated Gussian curve');
    xlabel('Data point occourence');
    ylabel('Data point value');
    legend('fitted', 'Normalised distribution histogram');

end
```



The task was to plot a scaled version of estimated mean and variance of the samples against a scaled normal distribution probability graph. As explained previously, the normal probability distribution graph shows that when data is normally distributed, its mean is greatest at the median, giving the Gaussian or "bell" shaped curve when plotted. The estimated Gaussian distribution was created using MatLab's "normfit()" function which returns the estimated mean and standard variance of the data passed in. Then the data had to be scaled to fit in the data occurrence scale provided. I achieved the right scaling parameters purely by trail and error, I was unsure if there was a formula that gives the right scaling values for both graphs for them to match up. The estimated Gaussian line gives a line of best fit on the raw data samples when they are scaled the same.

Decreasing the total number samples used doesn't effect the estimated Gaussian graph greatly, but it dramatically decreases the height of the data point occurrences of the raw data. This is because the calculation to find the estimated mean remains the same but the number of data that could possibly occur reduces giving a lower total spread across all data values. Having the sample size set to 10,000 means the graphs curve is of a size that demonstrates the point with clarity.

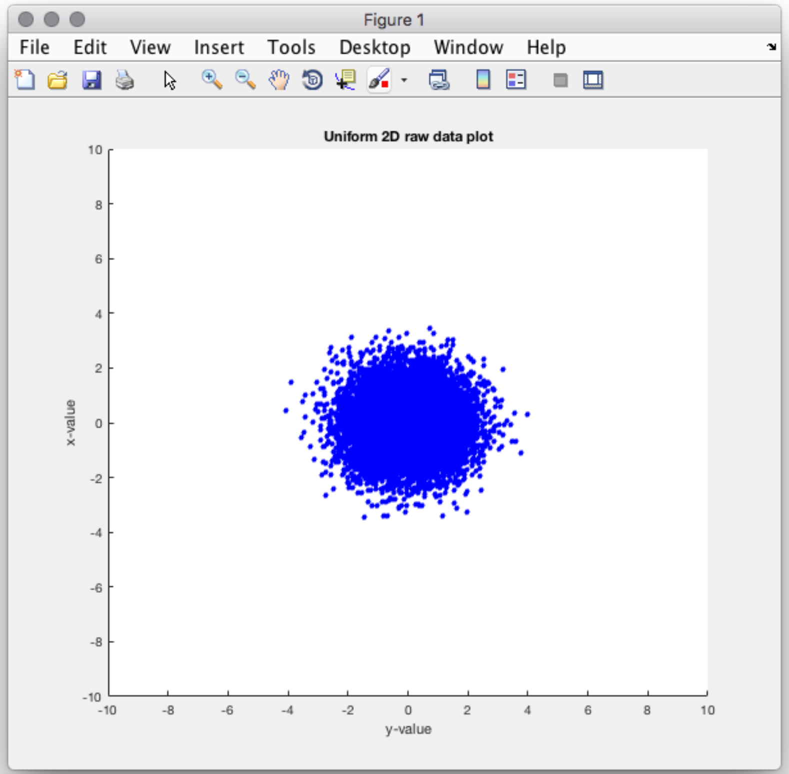
5. Generate a default 2D distribution



Editor – /Users/user/Documents/MATLAB/P1.1: 1D AND 2D DISTRIBUTIONS/defaultTwoDimensionalDistribution.m

defaultTwoDimensionalDistribution.m

```matlab
function defaultTwoDimensionalDistribution

    % Initialise the sample size
    sampleSize = 10000;

    % Initialise the standard deviation
    standardDeviation = 1;

    % Intialise the mean
    mean = 0;

    % Intialise the samples
    samples = standardDeviation.*randn(2,sampleSize) + mean;

    % Display the size of the samples
    disp(size(samples));

    % Create the size of the cirlces to be plotted
    circleSize = 20;

    % Plot the data dimension against eachother to get a 2D scatter plot
    scatter(samples(1, :), samples(2, :), circleSize, 'b', 'filled');

    % Format the graph
    title('Uniform 2D raw data plot');
    xlabel('y-value');
    ylabel('x-value');
    xlim([-10 10]);
    ylim([-10 10]);

end
```

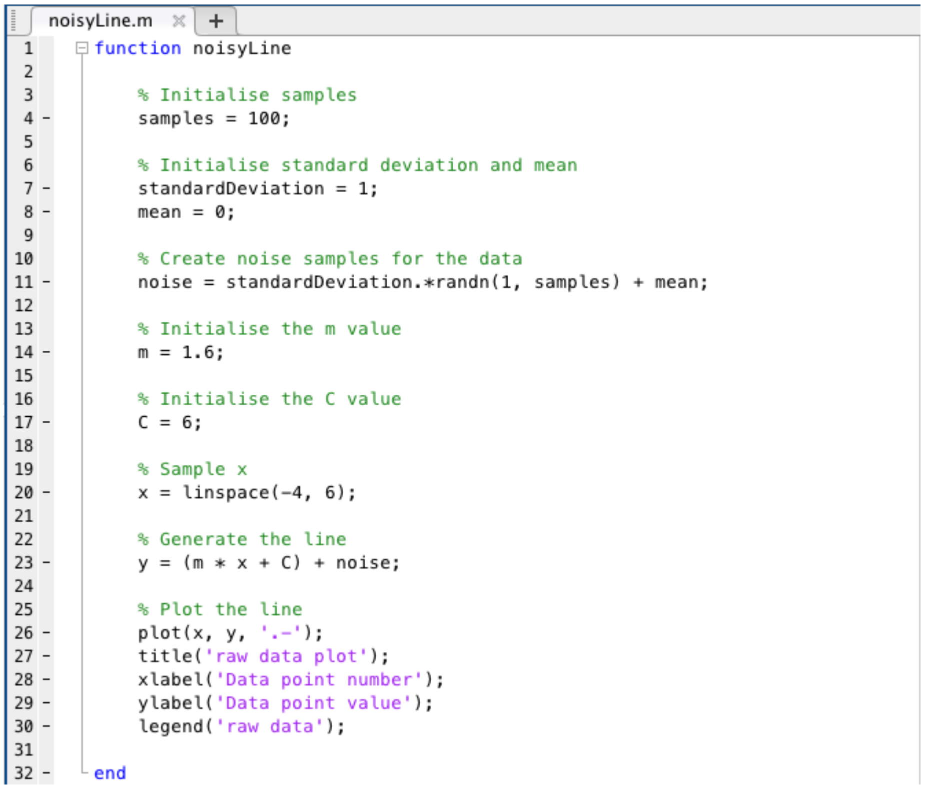


The task was to graph a representation of a default 2D distribution, which, in essence, is a normal probability distribution but of a higher dimension. At its heart, it derives from the central limit theorem as it shows two independently correlated set of random normal distributed data's mean still relatively equal to the median of the data values.

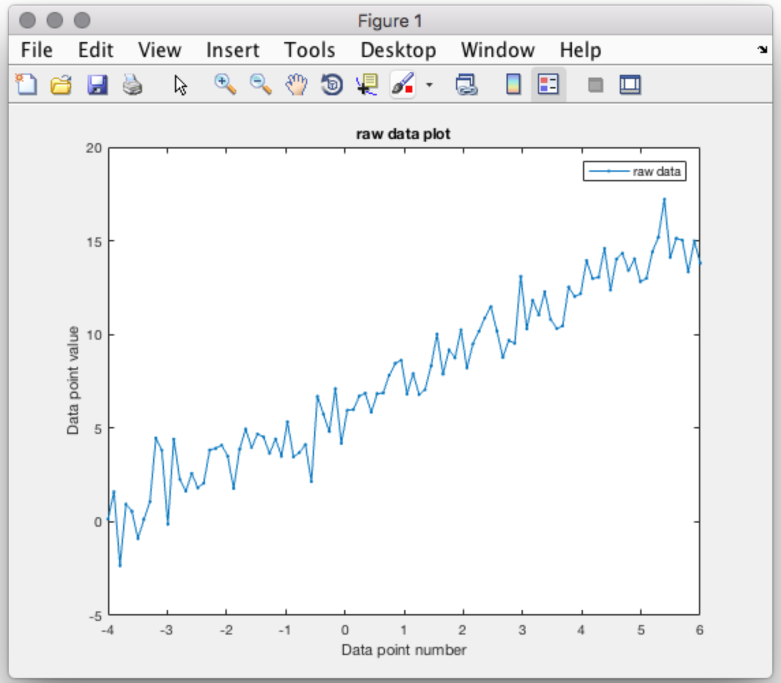
If you were to reduce the number of samples, it reduces the density of the cluster in the middle and makes the results more sparse. Increasing the standard deviation increases the distance of the data's relation to the mean. Changing the value of the mean shifts the middle of the cluster to the value set.

P1.2 Linear Regression

1. Generate a noisy line



```
function noisyLine

    % Initialise samples
    samples = 100;

    % Initialise standard deviation and mean
    standardDeviation = 1;
    mean = 0;

    % Create noise samples for the data
    noise = standardDeviation.*randn(1, samples) + mean;

    % Initialise the m value
    m = 1.6;

    % Initialise the C value
    C = 6;

    % Sample x
    x = linspace(-4, 6);

    % Generate the line
    y = (m * x + C) + noise;

    % Plot the line
    plot(x, y, '.-');
    title('raw data plot');
    xlabel('Data point number');
    ylabel('Data point value');
    legend('raw data');

end
```

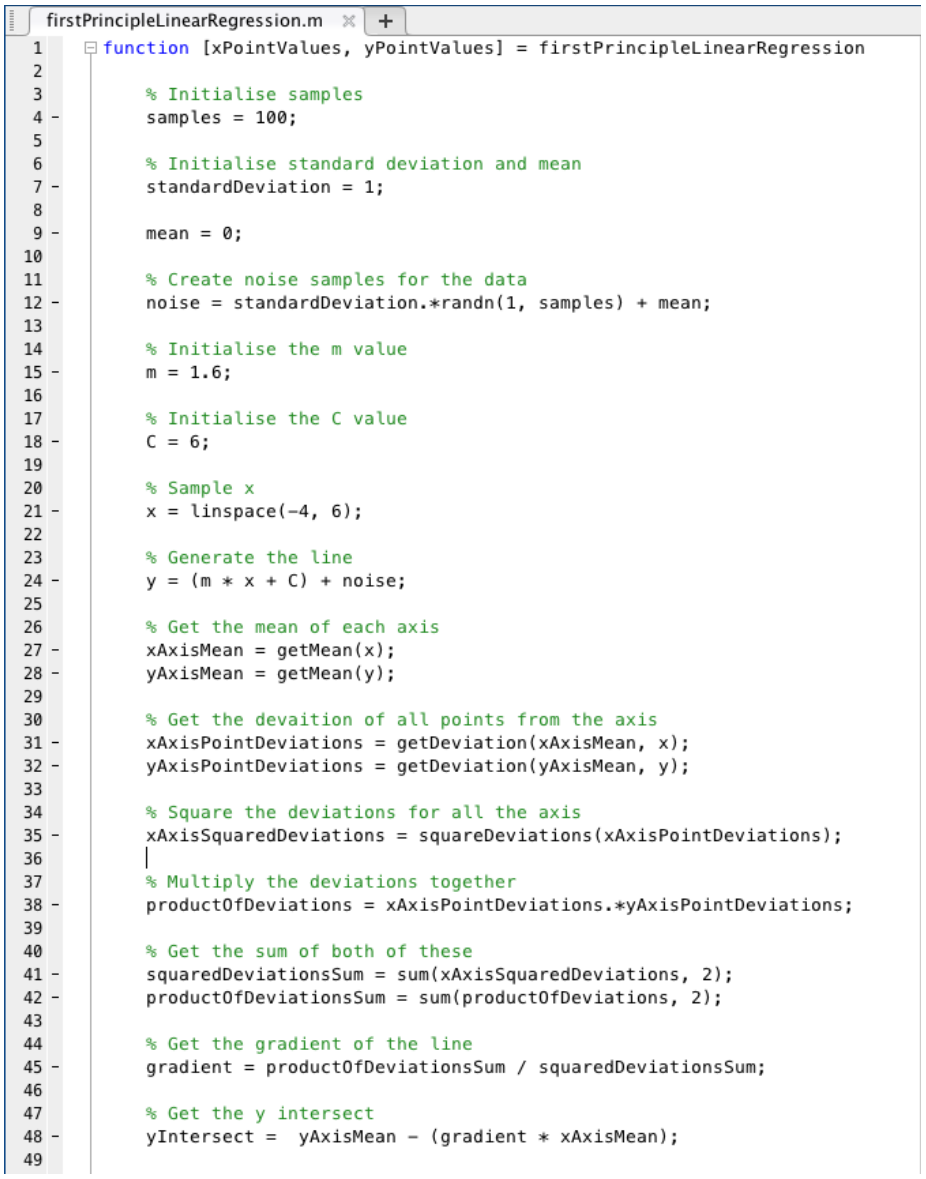


The task was to create and plot a linear regression line with noise. I achieved this by using the equation of a 1D line which stipulates that the line is composed of the gradient of the line, in this case given to us at a value of 1.6, multiplied by the x values with the Y intersect at 0, which was also provided to us at the value of 6, added onto the result of that. The samples to add Gaussian noise to the line was generated using the `randn` function in conjunction with the sample size, stated at 100, the mean, value 0, and standard deviation, value of 1.

This noise generation returns a matrix of normally distributed numbers in a matrix of 1 by the size of samples with a mean and standard deviation of the values stated. The fact we used `randn` gives us the Gaussian noise, instead of using `rand` which would give us uniformly distributed numbers, making the line's noise smoother because the probability of the range of the errors would be evenly distributed. Changing the sample size means we also have to change the range in which the x values are generated in order to make the matrices dimensions match, but in doing so, increasing both makes for a far noisier line, with the gaps between each error point much tighter due to the fact of the increase in data as a whole.

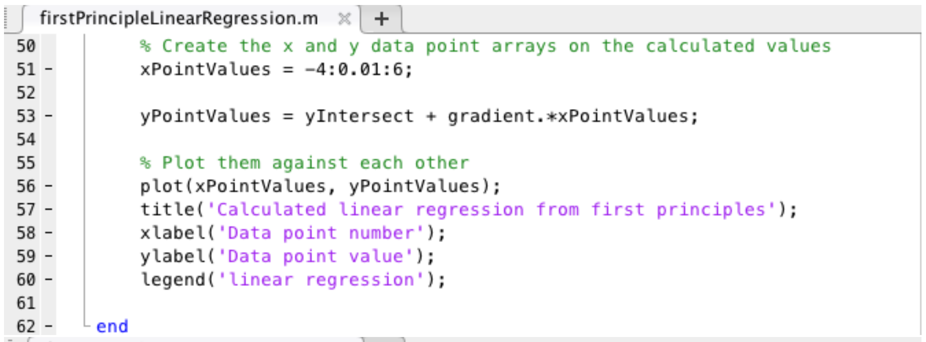
This line demonstrates highly positive correlated data that has noise in the form of errors.

2. Implement linear regression from first principles

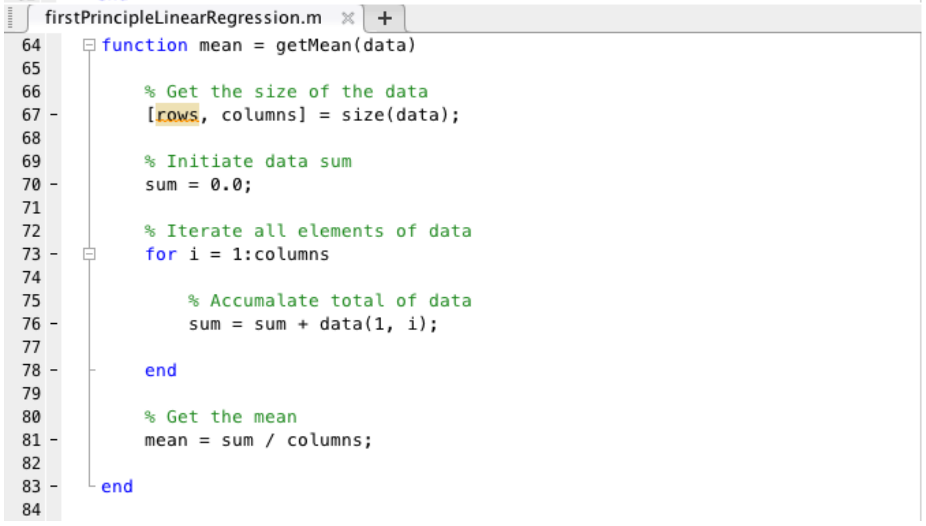


firstPrincipleLinearRegression.m

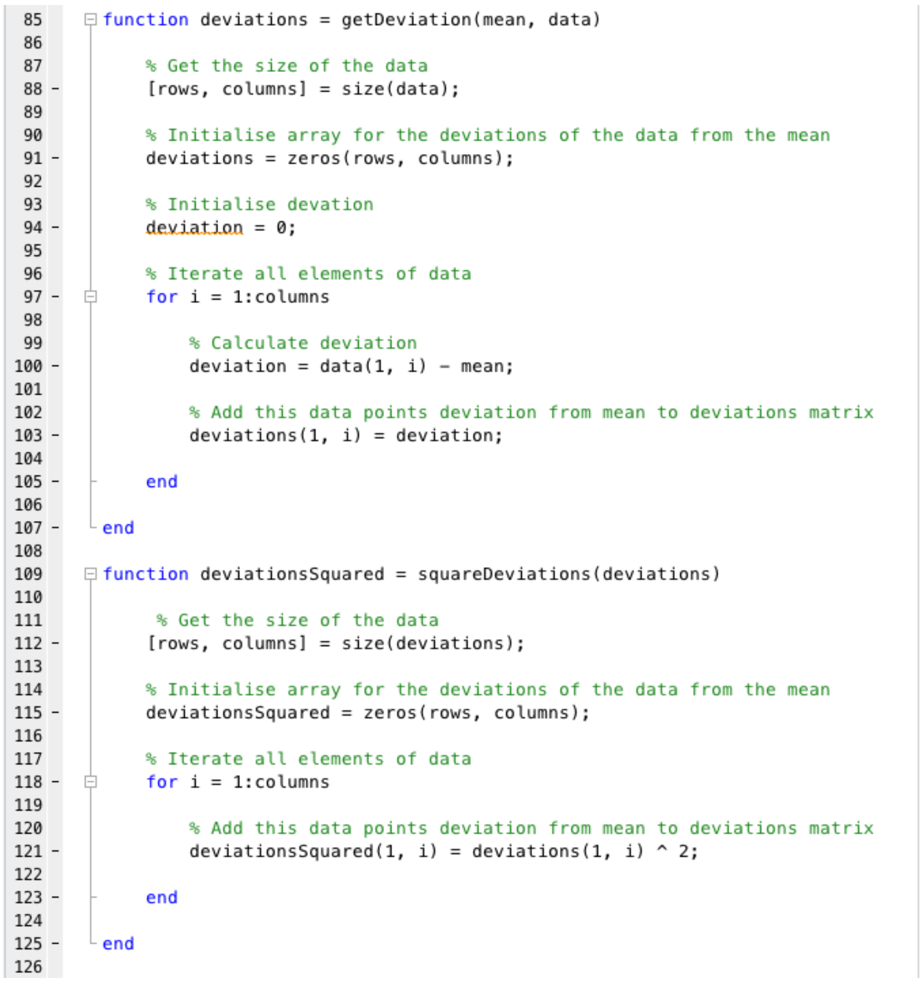
```
function [xPointValues, yPointValues] = firstPrincipleLinearRegression

    % Initialise samples
    samples = 100;

    % Initialise standard deviation and mean
    standardDeviation = 1;

    mean = 0;

    % Create noise samples for the data
    noise = standardDeviation.*randn(1, samples) + mean;

    % Initialise the m value
    m = 1.6;

    % Initialise the C value
    C = 6;

    % Sample x
    x = linspace(-4, 6);

    % Generate the line
    y = (m * x + C) + noise;

    % Get the mean of each axis
    xAxisMean = getMean(x);
    yAxisMean = getMean(y);

    % Get the devaition of all points from the axis
    xAxisPointDeviations = getDeviation(xAxisMean, x);
    yAxisPointDeviations = getDeviation(yAxisMean, y);

    % Square the deviations for all the axis
    xAxisSquaredDeviations = squareDeviations(xAxisPointDeviations);

    % Multiply the deviations together
    productOfDeviations = xAxisPointDeviations.*yAxisPointDeviations;

    % Get the sum of both of these
    squaredDeviationsSum = sum(xAxisSquaredDeviations, 2);
    productOfDeviationsSum = sum(productOfDeviations, 2);

    % Get the gradient of the line
    gradient = productOfDeviationsSum / squaredDeviationsSum;

    % Get the y intersect
    yIntersect =  yAxisMean - (gradient * xAxisMean);

```



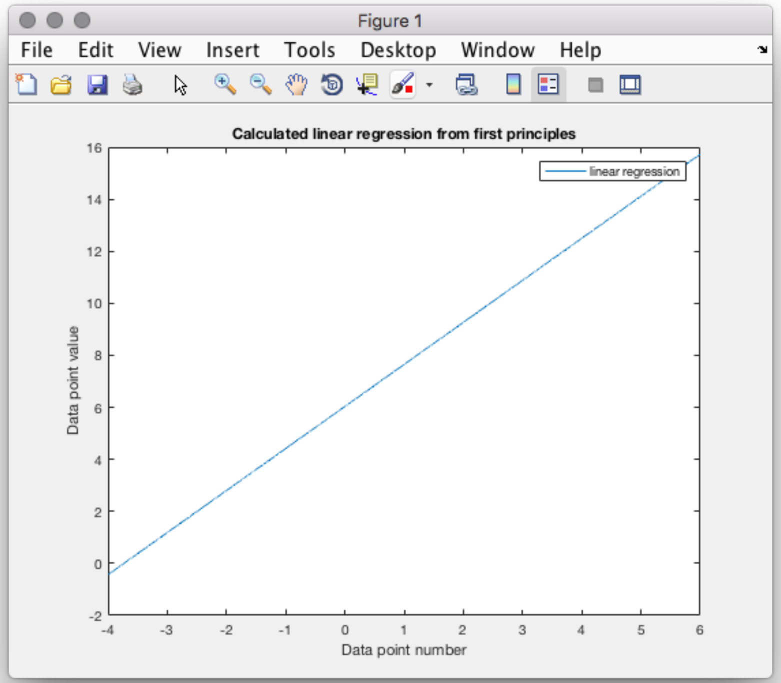
```
    % Create the x and y data point arrays on the calculated values
    xPointValues = -4:0.01:6;

    yPointValues = yIntersect + gradient.*xPointValues;

    % Plot them against each other
    plot(xPointValues, yPointValues);
    title('Calculated linear regression from first principles');
    xlabel('Data point number');
    ylabel('Data point value');
    legend('linear regression');

end
```



```
function mean = getMean(data)

    % Get the size of the data
    [rows, columns] = size(data);

    % Initiate data sum
    sum = 0.0;

    % Iterate all elements of data
    for i = 1:columns

        % Accumalate total of data
        sum = sum + data(1, i);

    end

    % Get the mean
    mean = sum / columns;

end

```

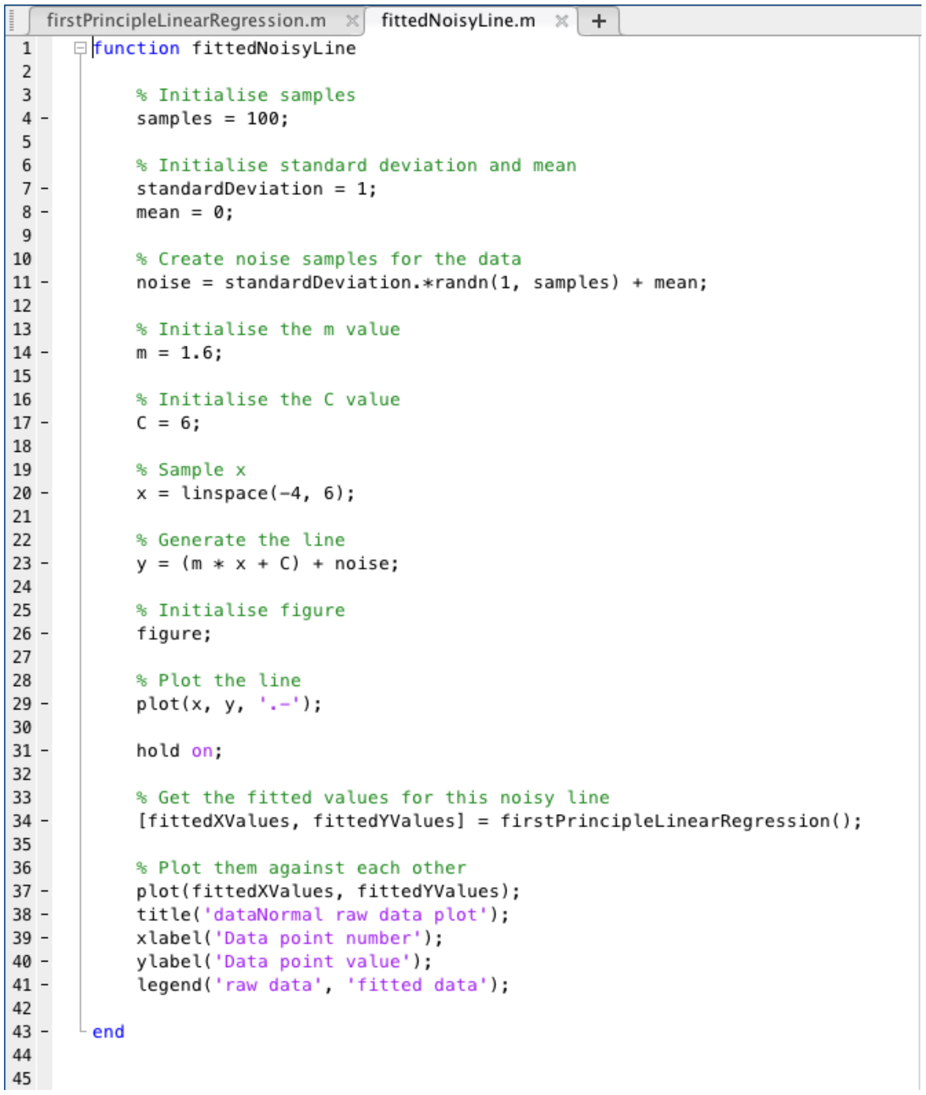


```
function deviations = getDeviation(mean, data)

    % Get the size of the data
    [rows, columns] = size(data);

    % Initialise array for the deviations of the data from the mean
    deviations = zeros(rows, columns);

    % Initialise devation
    deviation = 0;

    % Iterate all elements of data
    for i = 1:columns

        % Calculate deviation
        deviation = data(1, i) - mean;

        % Add this data points deviation from mean to deviations matrix
        deviations(1, i) = deviation;

    end

end

function deviationsSquared = squareDeviations(deviations)

     % Get the size of the data
    [rows, columns] = size(deviations);

    % Initialise array for the deviations of the data from the mean
    deviationsSquared = zeros(rows, columns);

    % Iterate all elements of data
    for i = 1:columns

        % Add this data points deviation from mean to deviations matrix
        deviationsSquared(1, i) = deviations(1, i) ^ 2;

    end

end
```

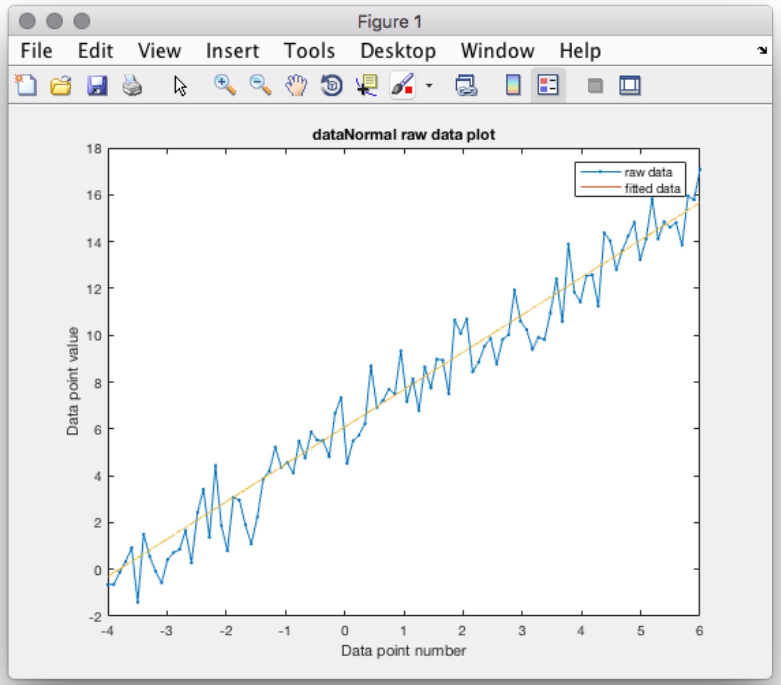


The task was to create linear regression line from first principles, this being that we had to program the fundamental methodologies of linear regressions myself, and not to use the in-built Matlab functions. I achieved this by using the least fitting square equation but programmed out step by step. The LFS method can be used to find the 'line of best fit' for a set of correlated data. It does so by minimising the residuals of the points from the curve. I first had to re-create the noisy line as specified in the first question of this practical, again, using `randn` to get the noise for the line as it generates a matrix of normally distributed (Gaussian) values, that being the probability of occurrence tend towards the mean and median of the data. From that I could work out the mean of both axes, mean being the average value over a given data set, which works out to be 1 and ~7 for x and y axes accordingly. The mean was needed to find the deviation of each point of the noisy line away from each axis's mean, deviation being the literal distance from one point to another (distance on each axes current data point from the respective axis mean). These values gave me enough information to calculate the linear regression line with only a few more manipulations to them. First by squaring the x-axis deviations, then multiplying the deviations of each axis together, summing both of these values and dividing together to get the gradient of the line (which works out to be 1.6 as given in the question above) and finally deriving the y-intersect by subtracting the product of the gradient and x-axis mean from the y-axis mean. With the gradient and intersect calculated, you can plug that into the standard y = mx + C equation to get the line. The resulting line essentially reverses the noise giving you a nice smooth line of best fit.

3. Fit the test line using your linear regression function

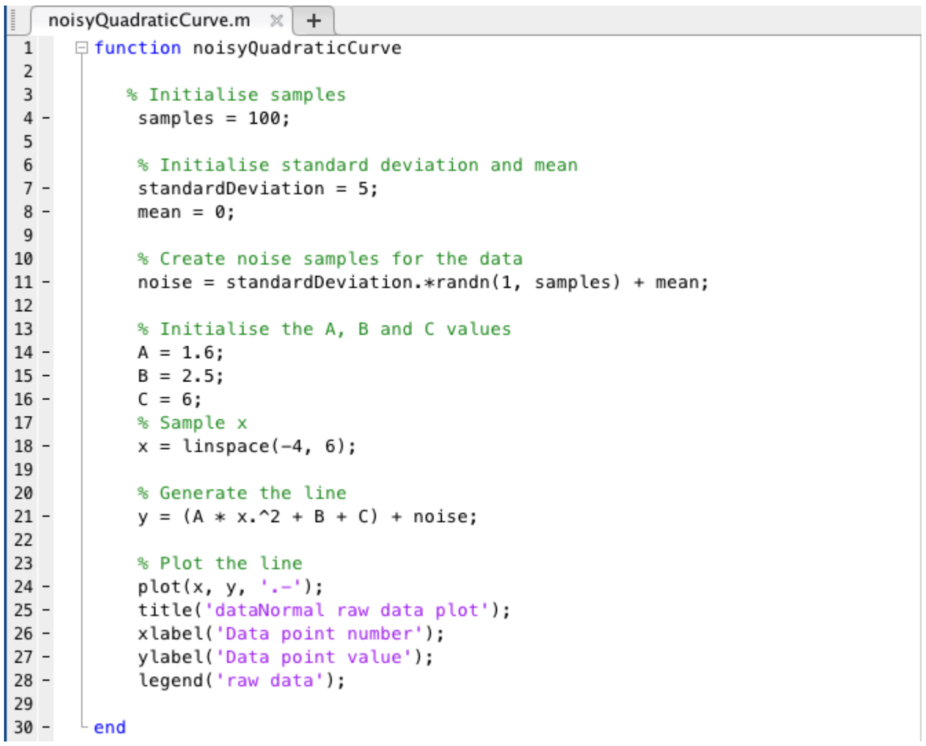


```
function fittedNoisyLine

    % Initialise samples
    samples = 100;

    % Initialise standard deviation and mean
    standardDeviation = 1;
    mean = 0;

    % Create noise samples for the data
    noise = standardDeviation.*randn(1, samples) + mean;

    % Initialise the m value
    m = 1.6;

    % Initialise the C value
    C = 6;

    % Sample x
    x = linspace(-4, 6);

    % Generate the line
    y = (m * x + C) + noise;

    % Initialise figure
    figure;

    % Plot the line
    plot(x, y, '.-');

    hold on;

    % Get the fitted values for this noisy line
    [fittedXValues, fittedYValues] = firstPrincipleLinearRegression();

    % Plot them against each other
    plot(fittedXValues, fittedYValues);
    title('dataNormal raw data plot');
    xlabel('Data point number');
    ylabel('Data point value');
    legend('raw data', 'fitted data');

end
```

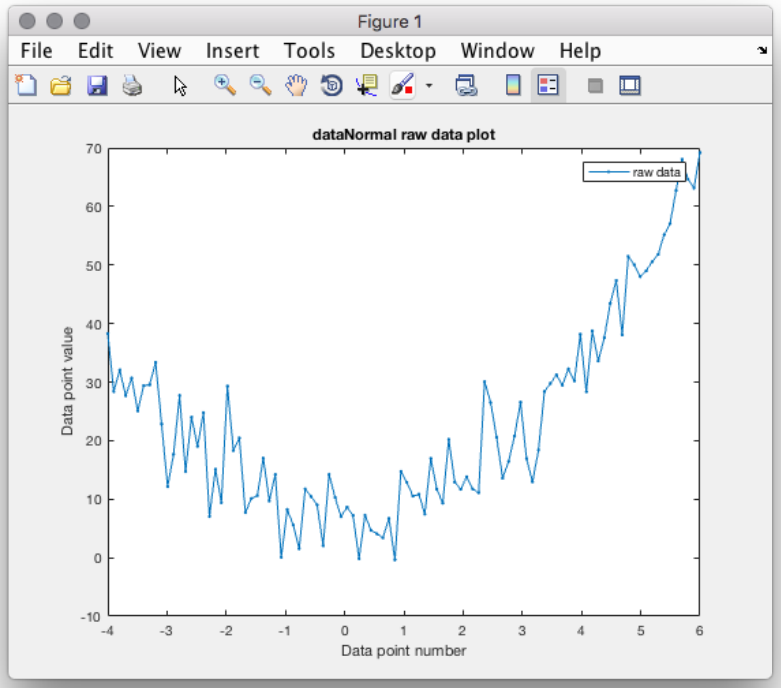


This exercise simply is an amalgamation of the previous two exercises. That is, to plot the 'fitted' line from exercise two (created from first principles using the LFS method) and the noisy linear line from exercise one. This produces a fitted noisy line, emphasising two things. Firstly, that the least fitting square method really does produce a reasonable line of best fit for the data and secondly, that draws attention to the fact the data is so positively correlated.

4. Generate a noisy quadratic curve



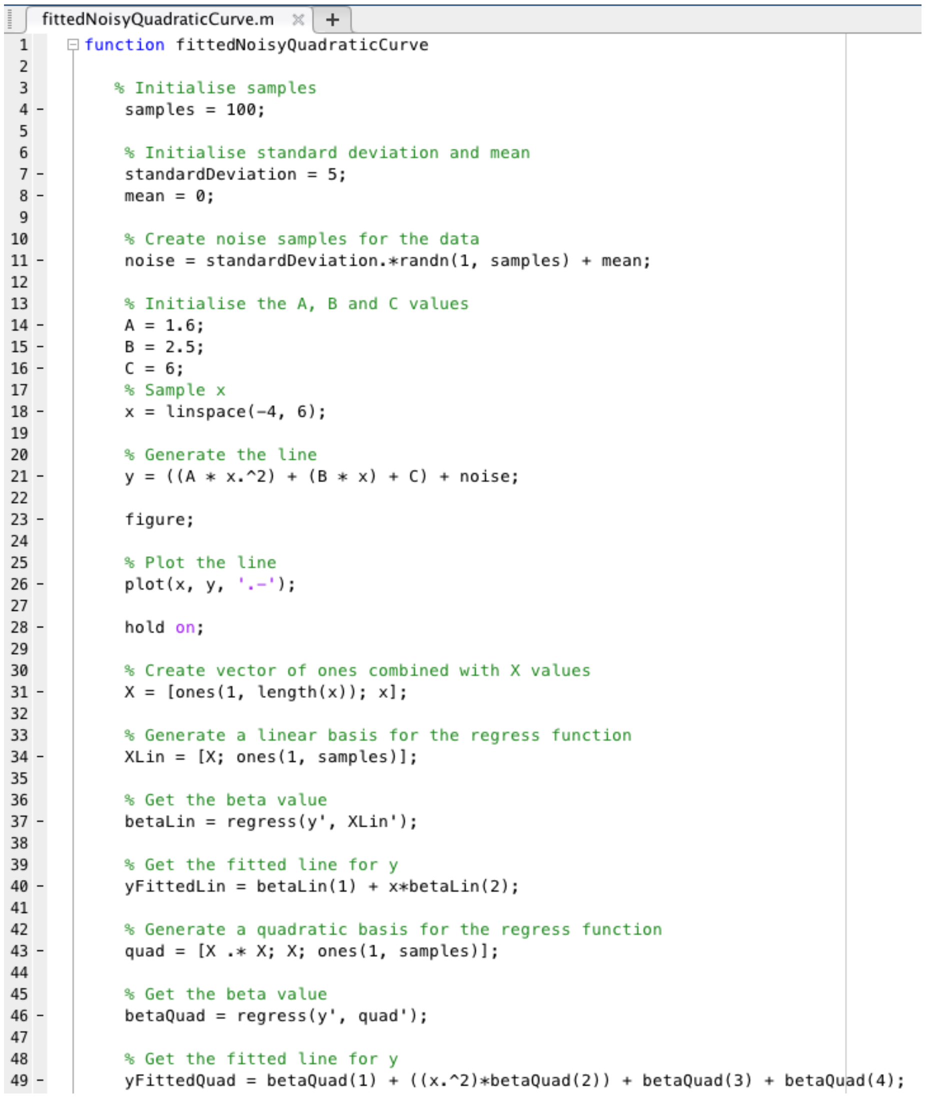
```
function noisyQuadraticCurve

   % Initialise samples
    samples = 100;

    % Initialise standard deviation and mean
    standardDeviation = 5;
    mean = 0;

    % Create noise samples for the data
    noise = standardDeviation.*randn(1, samples) + mean;

    % Initialise the A, B and C values
    A = 1.6;
    B = 2.5;
    C = 6;
    % Sample x
    x = linspace(-4, 6);

    % Generate the line
    y = (A * x.^2 + B + C) + noise;

    % Plot the line
    plot(x, y, '.-');
    title('dataNormal raw data plot');
    xlabel('Data point number');
    ylabel('Data point value');
    legend('raw data');

end
```



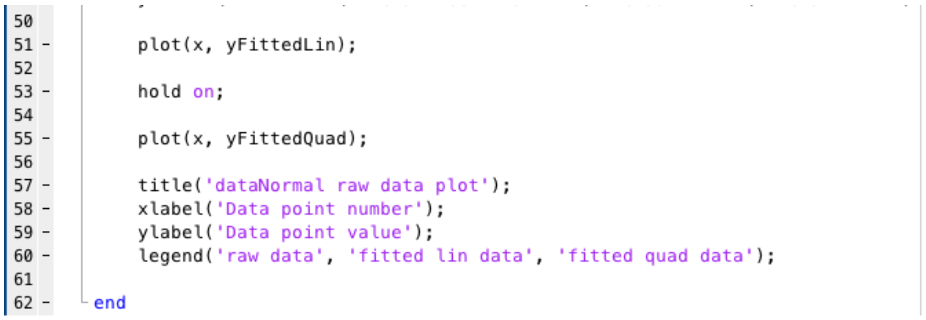
Generating a noisy quadratic curve simply, as the name suggests, using the second power in order to get the curve, in more formal notation, it generates a parabolic graph. If the graph was extended you would see, as quadratic curves show, that the lines (although noisy) would be symmetrical from when it passes in through the vertex; which in this case would be approximately between -1 and 0. This was achieved using the equation and values provided, as stated it is the Aˆ2 which gives it the curve. B in the equation determines the vertical placement of the graph and C is the constant of the equation which gives us the y-intersect. The graph shows us that as X gets exponentially bigger, Y increases until it reaches a limit, after which it decreases with the exact same (if it were not noisy) intervals as it increased.

If we were to change the values of A it would change the shape of the curve. A cannot be 0, but if we were to increase the value of A, it would reduce the standard deviation from the medium of the graph, if we were to decrease A it would make for a greater standard deviation. If A were negative, the curve would be “flipped”. As used in previous exercises, the noise was generated with `randn` function, to give us a matrix of normally distributed numbers. If you were to increase the standard deviation that is used in generating the noise, it would make the points more erratic, meaning they are further away from the ‘line’.

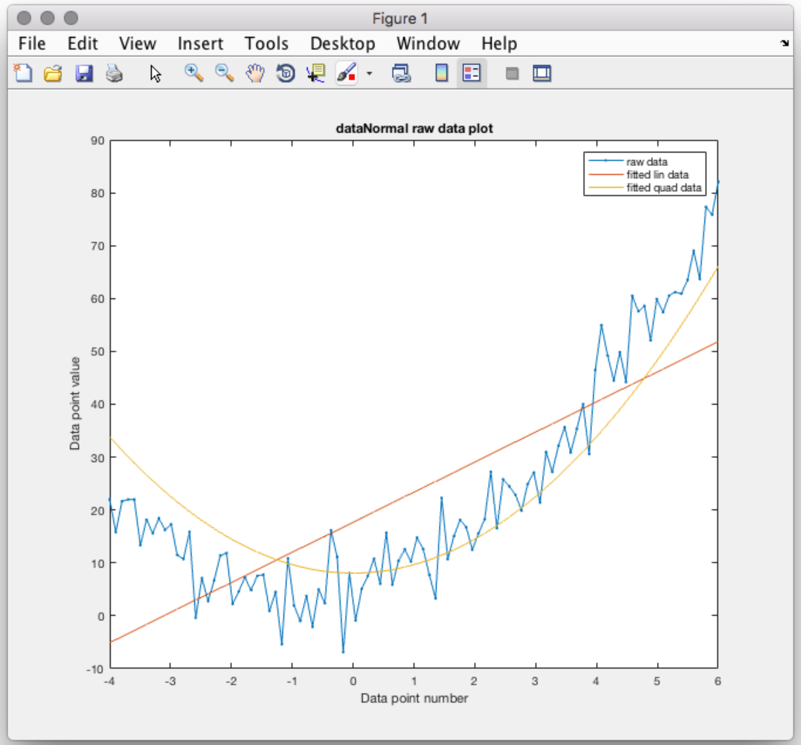
5. Fit the quadratic curve using linear regression



```
function fittedNoisyQuadraticCurve

   % Initialise samples
    samples = 100;

    % Initialise standard deviation and mean
    standardDeviation = 5;
    mean = 0;

    % Create noise samples for the data
    noise = standardDeviation.*randn(1, samples) + mean;

    % Initialise the A, B and C values
    A = 1.6;
    B = 2.5;
    C = 6;
    % Sample x
    x = linspace(-4, 6);

    % Generate the line
    y = ((A * x.^2) + (B * x) + C) + noise;

    figure;

    % Plot the line
    plot(x, y, '.-');

    hold on;

    % Create vector of ones combined with X values
    X = [ones(1, length(x)); x];

    % Generate a linear basis for the regress function
    XLin = [X; ones(1, samples)];

    % Get the beta value
    betaLin = regress(y', XLin');

    % Get the fitted line for y
    yFittedLin = betaLin(1) + x*betaLin(2);

    % Generate a quadratic basis for the regress function
    quad = [X .* X; X; ones(1, samples)];

    % Get the beta value
    betaQuad = regress(y', quad');

    % Get the fitted line for y
    yFittedQuad = betaQuad(1) + ((x.^2)*betaQuad(2)) + betaQuad(3) + betaQuad(4);
```



```matlab

    plot(x, yFittedLin);

    hold on;

    plot(x, yFittedQuad);

    title('dataNormal raw data plot');
    xlabel('Data point number');
    ylabel('Data point value');
    legend('raw data', 'fitted lin data', 'fitted quad data');

end
```



This exercises was to replicate the noisy quadratic curve but have it fitted using the `regress` function which returns a vector of coefficient estimates for a multi-linear regression of the responses in Y on the predictors in X. To do this, I had to create a vector of 1s combined with the value of X. We attached them together to make sure we get the first beta value. Then we get the beta value by using the `regress` function with the parameters of our newly created X values and the Y values. This gives us the fitted values of the line. The only change in the fitted quadratic line is the fact, as it is quadratic, we use the power of two to give us the parabolic curve.

The fitted linear line represents a positive correlation between the values of X and Y. But, as described in the last exercise, the quadratic fitted line, although shows correlation, it is both positive and negative to a limit, giving us the curve.

P1.3 Kmeans Clustering

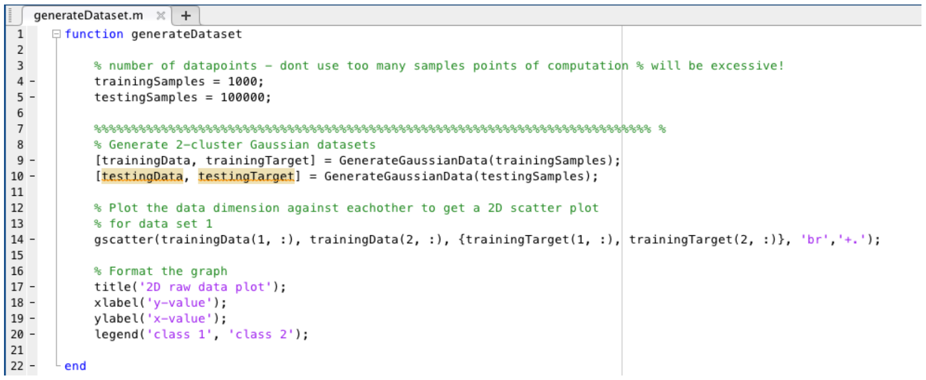
TO REMEMBER

- understanding of either rand or randn
- understanding of mean var
- insight into task
- (matlab code)
- What is the task
- How i solve it
- What does it mean
- **Eplain**
- How does changing number samples, bins effect what you see
- Gaussian adding no tegether, doesn't matter what limit they tend to go to a guassain form
- what constitutes a sensible choice
- model data by estimate parameters

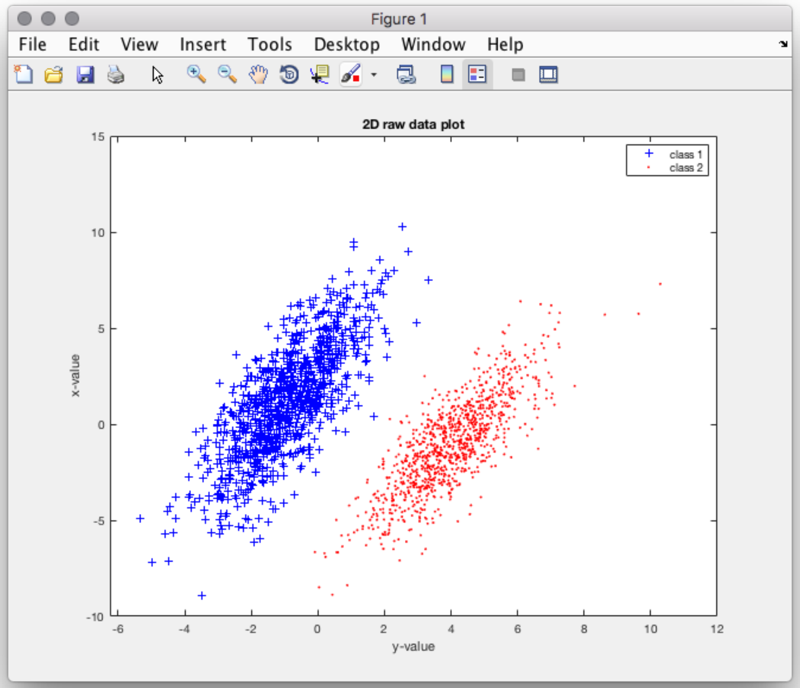
P1.4 Naive Bayes and Perceptron

Generate dataset

The task was to plot the data that is provided by the already implemented function `GenerateGaussianData`. The `GenerateGaussianData` function uses the sample size provided as a parameter to create a two Gaussian data sets with their respective targets concatenated together. To plot the data, I used the `gscatter` function of Matlab that allows you to group data together to plot it. For this function, I grouped the `trainingData` plots with the `trainingTarget` values to define which class they were in. The plot shows gives us a visual reference for what the dataset should actually look like when we train the classifiers and use the test data to see how well the classifiers work.



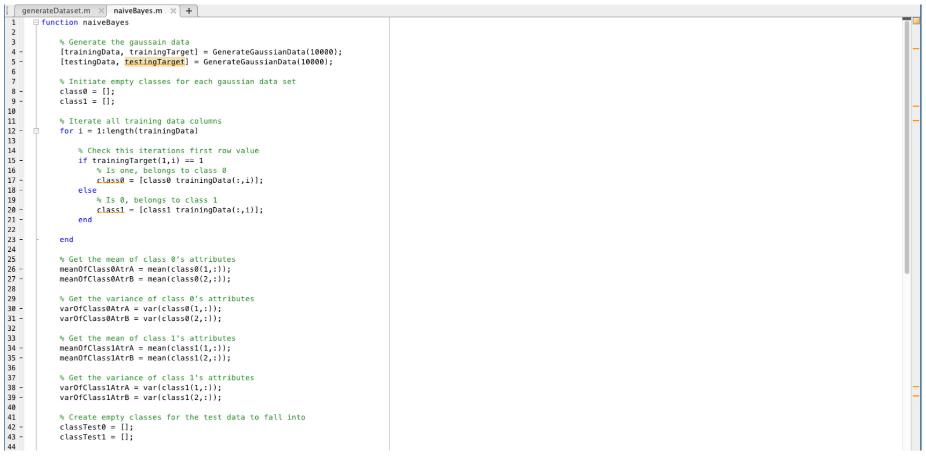
```matlab
function generateDataset

    % number of datapoints - dont use too many samples points of computation % will be excessive!
    trainingSamples = 1000;
    testingSamples = 100000;

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% %
    % Generate 2-cluster Gaussian datasets
    [trainingData, trainingTarget] = GenerateGaussianData(trainingSamples);
    [testingData, testingTarget] = GenerateGaussianData(testingSamples);

    % Plot the data dimension against eachother to get a 2D scatter plot
    % for data set 1
    gscatter(trainingData(1, :), trainingData(2, :), {trainingTarget(1, :), trainingTarget(2, :)}, 'br','+.');

    % Format the graph
    title('2D raw data plot');
    xlabel('y-value');
    ylabel('x-value');
    legend('class 1', 'class 2');

end
```



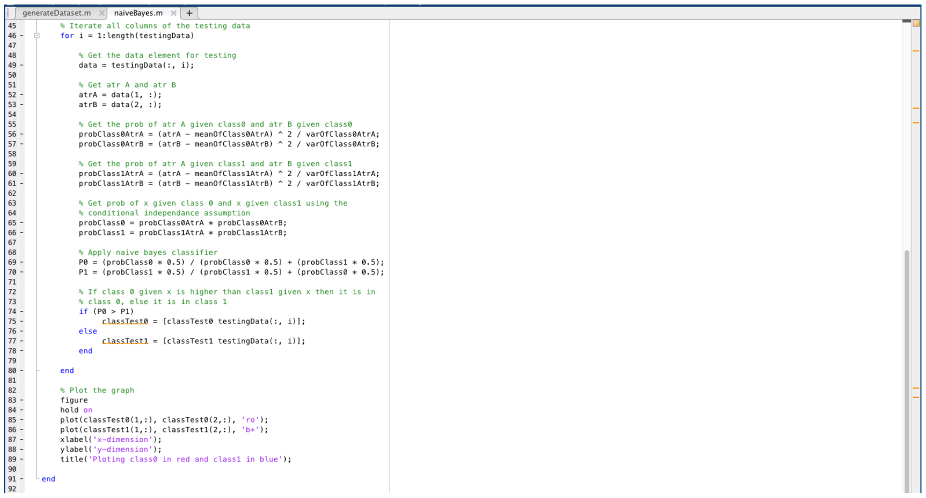
Implement a Naive bayes classifier

Our task was to implement a Naive bayes classifier, training and testing it with data generated from `GenerateGaussianData` function. A Naive bayes classifier is based on Baye's theorem that assumes there is independence between each predictor. That is, that us knowing the value of one attribute does not tell us anything about the other attribute. It also is Naive because it ignores the order of attribute values as we multiple across them to get the probability of that class. Doing this for all classes means that we have the ratio or probability that it is in one class over the other classes.

As we are working with continuous data, to implement it requires you to know the classes of each data, which are provided in the `trainingTarget`, the mean and the variance of the data before being able to classify the testing data. Firstly, I split the Gaussain data into their classes using the `trainingTarget` as a reference. I know a prior that the data consists of two Gaussians concatenated together and so I just re-split these into seperate data classes so that I can get the mean and variance of both classes and both attributes (rows) of the classes, leaving me with four variables in total to use in the classification. After this, I iterate every data element (column) in the testing data set to get both attributes of that data element. Then using the mean and variance of each class and each attribute from the previous steps, I can calculate the probability of the attribute given the class. From that, I can then further calculate the probability of any attribute, `x`, given the class by multiplying the probability of each attribute given each class. This is where it is Naive, as you are multiplying you disregard the order, this is the conditional independence assumption. Given I have the probability of x given the class, I now calculate the probability of it being in both classes, using that data, as well as the a prior probability of 0.5, which I assumed as the data is concatenated of two Gaussian datasets of equal length. This gives us the ratio of it being in either class, so one assumes that it if one is higher than the other, then it belongs to that class accordingly.



```
function naiveBayes

    % Generate the gaussain data
    [trainingData, trainingTarget] = GenerateGaussianData(10000);
    [testingData, testingTarget] = GenerateGaussianData(10000);

    % Initiate empty classes for each gaussian data set
    class0 = [];
    class1 = [];

    % Iterate all training data columns
    for i = 1:length(trainingData)

        % Check this iterations first row value
        if trainingTarget(1,i) == 1
            % Is one, belongs to class 0
            class0 = [class0 trainingData(:,i)];
        else
            % Is 0, belongs to class 1
            class1 = [class1 trainingData(:,i)];
        end

    end

    % Get the mean of class 0's attributes
    meanOfClass0AtrA = mean(class0(1,:));
    meanOfClass0AtrB = mean(class0(2,:));

    % Get the variance of class 0's attributes
    varOfClass0AtrA = var(class0(1,:));
    varOfClass0AtrB = var(class0(2,:));

    % Get the mean of class 1's attributes
    meanOfClass1AtrA = mean(class1(1,:));
    meanOfClass1AtrB = mean(class1(2,:));

    % Get the variance of class 1's attributes
    varOfClass1AtrA = var(class1(1,:));
    varOfClass1AtrB = var(class1(2,:));

    % Create empty classes for the test data to fall into
    classTest0 = [];
    classTest1 = [];

```



```
    % Iterate all columns of the testing data
    for i = 1:length(testingData)

        % Get the data element for testing
        data = testingData(:, i);

        % Get atr A and atr B
        atrA = data(1, :);
        atrB = data(2, :);

        % Get the prob of atr A given class0 and atr B given class0
        probClass0AtrA = (atrA - meanOfClass0AtrA) ^ 2 / varOfClass0AtrA;
        probClass0AtrB = (atrB - meanOfClass0AtrB) ^ 2 / varOfClass0AtrB;

        % Get the prob of atr A given class1 and atr B given class1
        probClass1AtrA = (atrA - meanOfClass1AtrA) ^ 2 / varOfClass1AtrA;
        probClass1AtrB = (atrB - meanOfClass1AtrB) ^ 2 / varOfClass1AtrB;

        % Get prob of x given class 0 and x given class1 using the
        % conditional independance assumption
        probClass0 = probClass0AtrA * probClass0AtrB;
        probClass1 = probClass1AtrA * probClass1AtrB;

        % Apply naive bayes classifier
        P0 = (probClass0 * 0.5) / (probClass0 * 0.5) + (probClass1 * 0.5);
        P1 = (probClass1 * 0.5) / (probClass1 * 0.5) + (probClass0 * 0.5);

        % If class 0 given x is higher than class1 given x then it is in
        % class 0, else it is in class 1
        if (P0 > P1)
            classTest0 = [classTest0 testingData(:, i)];
        else
            classTest1 = [classTest1 testingData(:, i)];
        end

    end

    % Plot the graph
    figure
    hold on
    plot(classTest0(1,:), classTest0(2,:), 'ro');
    plot(classTest1(1,:), classTest1(2,:), 'b+');
    xlabel('x-dimension');
    ylabel('y-dimension');
    title('Ploting class0 in red and class1 in blue');

end
```

Run a Naive bayes classifier and plot the result

As you can see from my graph, I have a bug in my code or have miscalculated one aspect as it seems that their is a intersection into each cluster of the other class. It has a curved linearity and seems to be symmetrical which leads me to believe it is a miscalculation of the data. Unfortunately due to external reasons I cannot spend any more time in working through the problem and so it is left as such. One can see that it works for the majority of the results. If these intersections were ignored, then you would see it divides the data as it should, given the examples in the assignment. I can see with `trainingData` sample rate set to 1000 and the `testingData` set to 10000 that it is almost result for them both being set at 10000. I figure this is because the difference in sample rate isn't large as compared to the third graph which is were the `trainingData` is set to 1000 and the `testingData` is set to 100000. Here you can see the clusters are a lot more dense and so the lines of intersection are well defined. Again however, if these intersections are ignored, then you can see that the majority of the data is in the correct class and the training data has been successful. Ignoring the incorrect intersection of the classes, the boundaries are sufficient given the small training sample size. They small percentage of error at the top of the first cluster and the bottom second cluster. This is due to these points overlapping more so than the rest of the cluster and it is where the mean and variance of each are close to one another so therefore it is harder to define whether or not it is in either class. One thing to note about naive bayes and the reason that it is so popular is because it only requires one iteration through the dataset with a low expense in computations to give high accuracy results, meaning that it is suitable for large dataset.

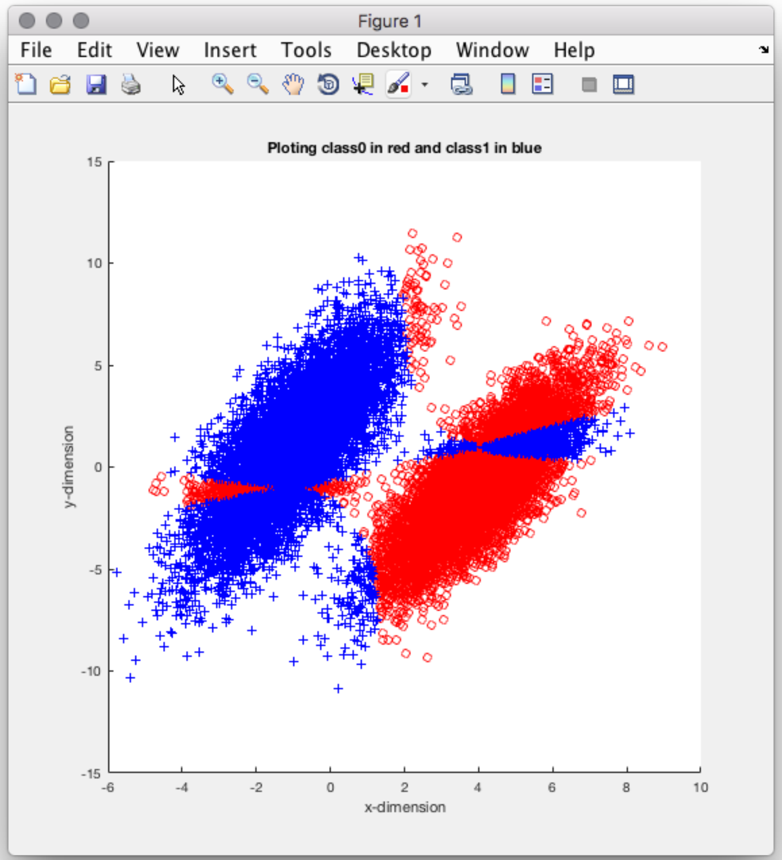


Figure 1
File Edit View Insert Tools Desktop Window Help
Ploting class0 in red and class1 in blue
y-dimension
x-dimension
15
10
5
0
-5
-10
-15
-6
-4
-2
0
2
4
6
8
10

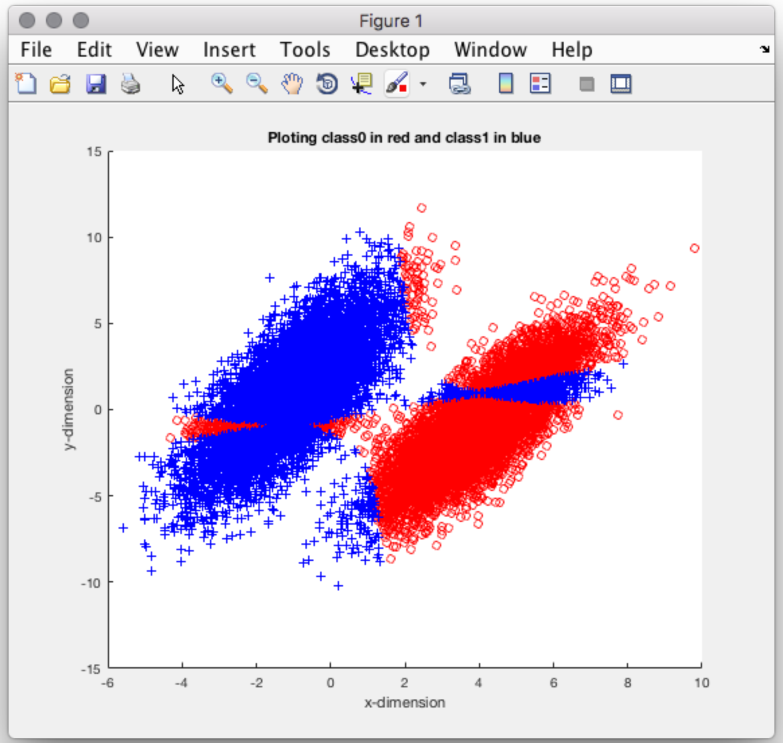
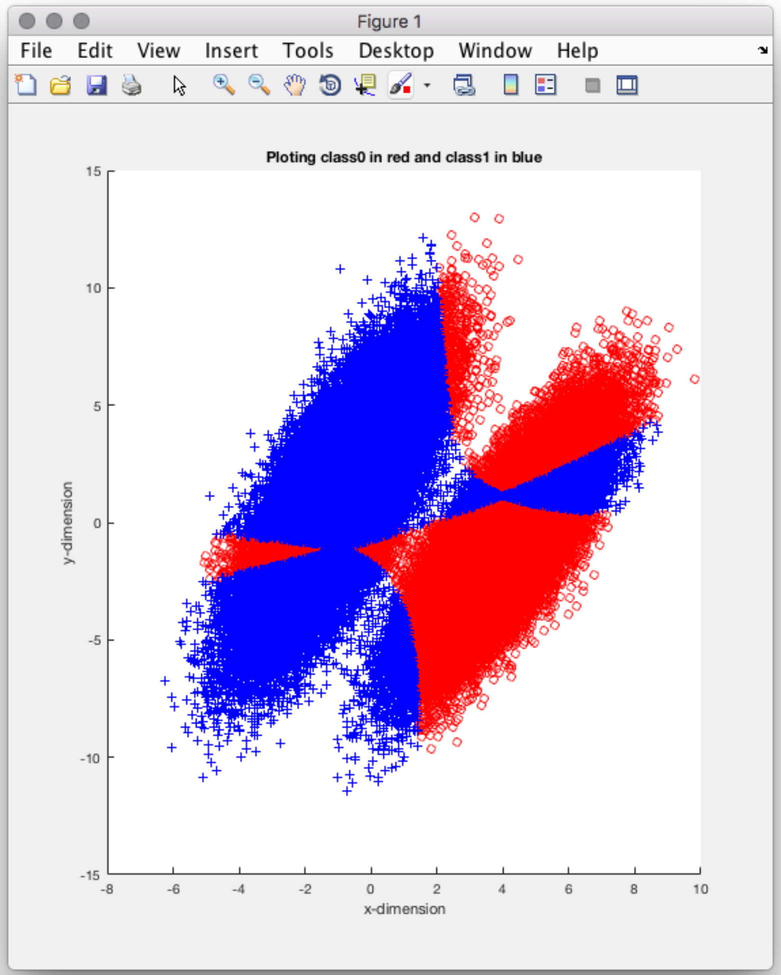
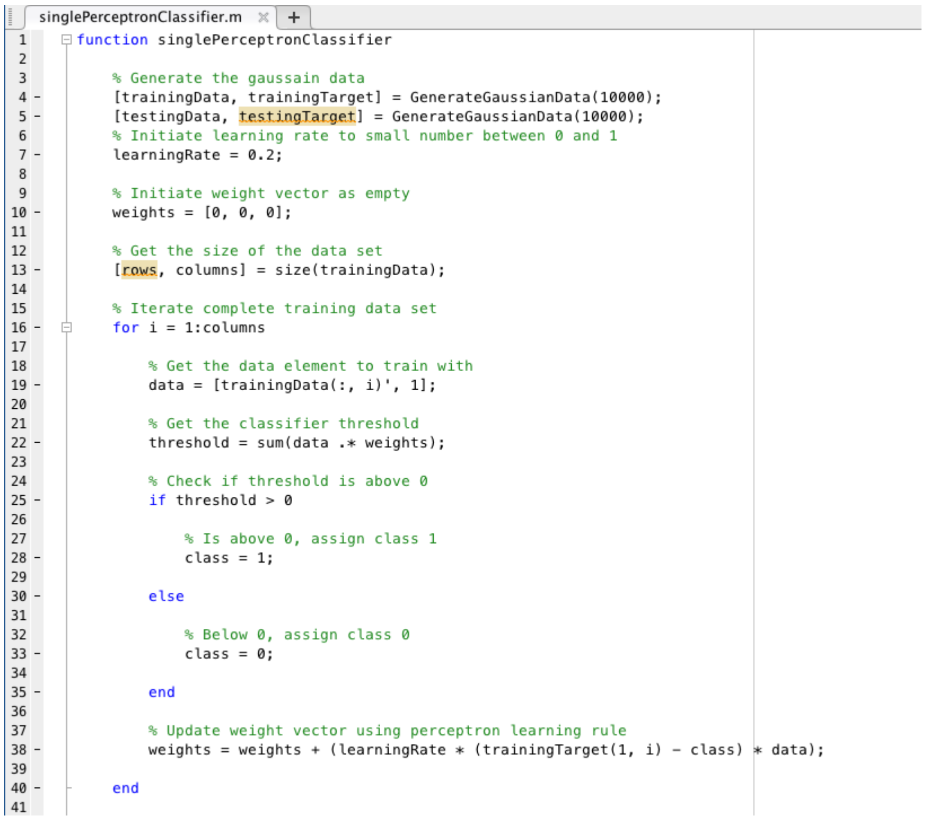


Figure 1
File
Edit
View
Insert
Tools
Desktop
Window
Help
Ploting class0 in red and class1 in blue
y-dimension
x-dimension
15
10
5
0
-5
-10
-15
-6
-4
-2
0
2
4
6
8
10



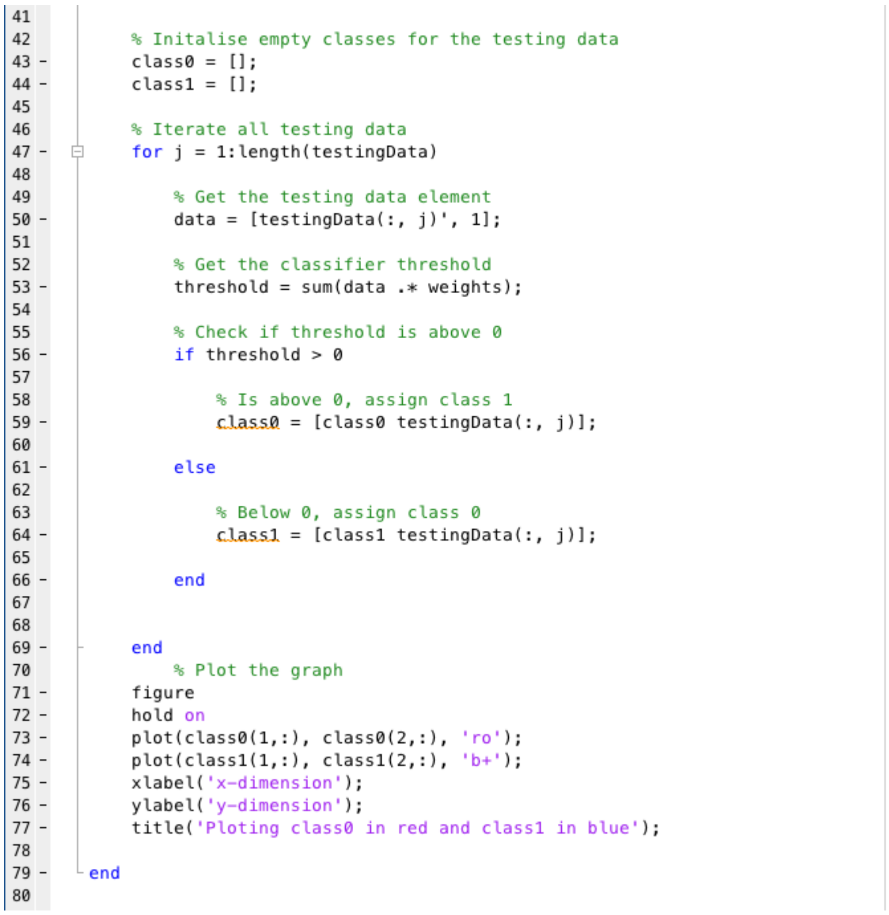
Implement a simple single layer perceptron

The task here was train a single layer perceptron using the weight vector update formula provided in the lectures. I did this by setting a small learning rate of 0.2, reasoning for this is discussed in the next section, as well as initiating the weight vectors to 0. Then you iterate all elements in the training data set, summing the data set



singlePerceptronClassifier.m

```
function singlePerceptronClassifier

    % Generate the gaussain data
    [trainingData, trainingTarget] = GenerateGaussianData(10000);
    [testingData, testingTarget] = GenerateGaussianData(10000);
    % Initiate learning rate to small number between 0 and 1
    learningRate = 0.2;

    % Initiate weight vector as empty
    weights = [0, 0, 0];

    % Get the size of the data set
    [rows, columns] = size(trainingData);

    % Iterate complete training data set
    for i = 1:columns

        % Get the data element to train with
        data = [trainingData(:, i)', 1];

        % Get the classifier threshold
        threshold = sum(data .* weights);

        % Check if threshold is above 0
        if threshold > 0

            % Is above 0, assign class 1
            class = 1;

        else

            % Below 0, assign class 0
            class = 0;

        end

        % Update weight vector using perceptron learning rule
        weights = weights + (learningRate * (trainingTarget(1, i) - class) * data);

    end

```



```

    % Initalise empty classes for the testing data
    class0 = [];
    class1 = [];

    % Iterate all testing data
    for j = 1:length(testingData)

        % Get the testing data element
        data = [testingData(:, j)', 1];

        % Get the classifier threshold
        threshold = sum(data .* weights);

        % Check if threshold is above 0
        if threshold > 0

            % Is above 0, assign class 1
            class0 = [class0 testingData(:, j)];

        else

            % Below 0, assign class 0
            class1 = [class1 testingData(:, j)];

        end


    end
        % Plot the graph
    figure
    hold on
    plot(class0(1,:), class0(2,:), 'ro');
    plot(class1(1,:), class1(2,:), 'b+');
    xlabel('x-dimension');
    ylabel('y-dimension');
    title('Ploting class0 in red and class1 in blue');

end

```

Run the perceptron and plot the results

The learning rate is supposed to be a small number between 0 and 1. However, it is more ideal to have the learning rate smaller and for it to still provide sufficient results to show that the algorithm can work efficiently without the aid of a predefined constant learning rate. I decided to settle on 0.2 (as seen in the first two graphs) because it is as close to 0 with little boundary errors. If the learning rate is higher (as seen in graph three with a learning rate of 0.5) then the results are far more accurate and the boundaries are much more defined with less error. The difference in boundary error at learning rate of 0.2 doesn't differ between sample size but infact just varies on that particular run. Sometimes the boundary is extremely acurate and sometimes it has a higher margin of error. This can be argued it is because of the Gaussian data set varying slightly so the threshold is lower or higher on different runs. As you can see though, the single layer perceptron is an effective way of classifying data as the computation is relatively low but also very accurate.

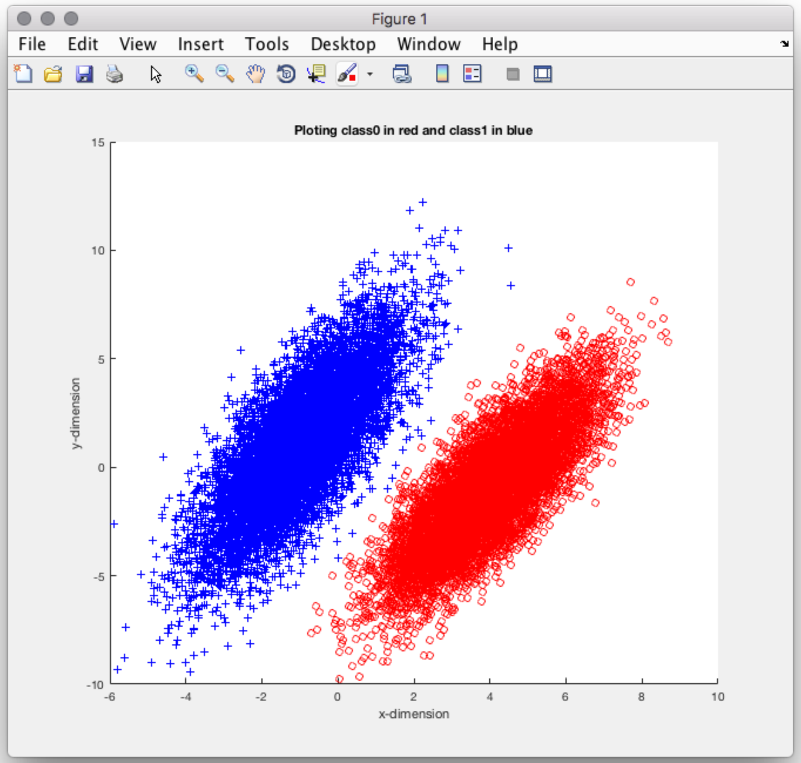


Figure 1
File Edit View Insert Tools Desktop Window Help
Ploting class0 in red and class1 in blue
y-dimension
x-dimension
15
10
5
0
-5
-10
-6
-4
-2
0
2
4
6
8
10

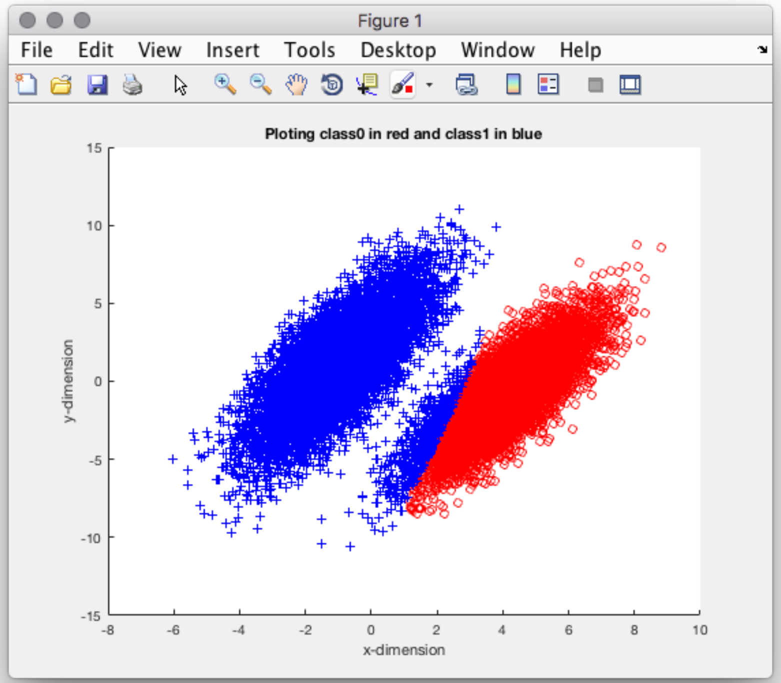


Figure 1
File Edit View Insert Tools Desktop Window Help
Ploting class0 in red and class1 in blue
y-dimension
x-dimension
15
10
5
0
-5
-10
-15
-8
-6
-4
-2
0
2
4
6
8
10