AINT351 Lab Journal

Torbjorn Lab Journal

P2.1: CLASSIFICATION WITH DECISION TREES

Initial Decision tree

To get the program set up and running, this function is constructed as a skeleton function initially. Within which I, as instructed, accepted two parameters which is the data split into meas matrix and the species vector. In order to get a dataset that was easier to work with later in the program, I decided to take the approach that concatenates the matrix and vector together as well as an extra column for the initial set value. This is stored in a cell array due to the difference in data types between the variable values (floating point values) and the vector of species (string). I did this because passing around a whole data set is easier and less 'parameter' heavy, it also helps later on when filtering the data, splitting the data and comparing it as you don't have to draw all the data from different storage locations, do the operation and return to the location.

Split

The split function is very simple but is crucial for the decision tree's functionality. It takes in the set to split, the variable (column in this case) to split on and the threshold value to make the comparison in order to make the split as parameters. From there on I simply initialised two sets for the initial set to be split into, iterated every row of the set passed in and compared the value of that row, in that column, to the threshold value. If it was less

than the threshold then it would fall into set one, else it would fall into set two. Once all sets were iterated the two new sets from the split are returned by the function.

I tested the functionality in the `classificationWithDecisionTreesTests` class by creating a dummy set of data which had three sets of data all with arbitrary data. I passed in my own split condition and because I knew the data, the threshold condition, I knew what the outcome of the split would be. By doing this I ensured that the core functionality of the program is solid and robust. I always believe in writing tests and always try to do write in a test driven development way.

Editor – /Users/user/Documents/MATLAB/P2.1: CLASSIFICATION WITH DECISION TREES/split.m

split.m

```
% Splitting a data set given a rule
% S         = set (the whole matrix)
% varIdx    = variable index (column)
% threshold = what to split it on
function [S1, S2] = split(S, varIdx, threshold)

    % Initialise empty matrices for results
    S1 = {};
    S2 = {};

    % Get the amount of rows in matrix
    rows = size(S, 1);

    % For each row of S
     for row = 1:rows

        % Check if column value for this row is less than the threshold
        if S{row, varIdx} < threshold

            % Copy row accross to S1
            S1 = [S1; S(row, :)];

        % Check if the column value for this row is more or equal to
        % threshold
        elseif S{row, varIdx} >= threshold
            % Value is greater than or eqaul to threshold

            % Copy row accross to S1
            S2 = [S2; S(row, :)];

        end
     end
end
```

```
%% Test: split
function splitTest(testCase)
    %input:   S (set), varIdx (column), threshold (value to compare)
    %process: Splits the value under the varIdx into two sets based on the threshold
    %output:  Two sets, S1 with value less than threshold and S2 higher
    S = [num2cell(1.5), num2cell(1.7), cellstr('setosa');
         num2cell(2.5), num2cell(2), cellstr('virginica');
         num2cell(2), num2cell(2), cellstr('virginica')];

    [S1, S2] = split(S, 1, 2);

    assert(isequal(S1, {1.5000, 1.7000,'setosa'}));
    assert(isequal(S2, {2.5000, 2,'virginica'; 2, 2, 'virginica'}));

end
```

Entropy

I calculated the entropy of data set using the following equation `Entropy = - p(a)*log(p(a)) - p(b)*log(p(b))` but made it dynamic in order to take three sets, the three different species, and did it from first principles. Entropy is a measure of impurity in the set which is given on a scale of 0 to 1.0. 0.5 is the values of the set can be in either one set or another, 0 and 1 is completely certainty that it is or isn't in the set.

Firstly I split the equation down into multiple parts, two these into separate functions within the function class and one in a separate function altogether as it is also used in the calculation of the `majorityClass`. In order to get the probability, you have to get the number of each species that are in set that are passed in, this is done in the external function that I wrote `getNoOfPlantType` which iterates the whole set and compares the string of the desired plant type passed into the function with the current row's plant type and counts the number of instances it finds and returns that value. I extracted this into a function of its own so it can be re-used and modularised which is better practice in general, to comply with the DRY programming principle (don't repeat yourself). Secondly, using the number of the instances of that plant type in the set as a whole, you're able to calculate the probability of it occurring in that set as a whole, this was also done in the separate function `getProbOfPlantType` because it has to be done three times for each plant type. It is simply the number of that specific plant type divided by the total number of all plant types in the set provided. The last function uses the two previous values to get the entropy of each plant type by using that calculation in conjunction with the probability of the plant type. To note here, if the probability of the plant type is 0, therefore no chance of it occurring, it is given the worst possible value of entropy which is 1. Finally you can subtract all entropy values to get the final entropy which is returned.

I also wrote a test for the functionality of this, but it wasn't in true TDD style. Instead I knew a value of the entropy given a set and compared that with the outcome of calling the function.

```
function entropy = entropy(S)

    % Get the number of each plant type in the set
    noOfVirginica  = getNoOfPlantType('virginica', S);
    noOfVersicolor = getNoOfPlantType('versicolor', S);
    noOfSetosa     = getNoOfPlantType('setosa', S);

    % Get the total amount of values in set
    totalNumOfValues = size(S, 1);

    % Get the probability of each plantType
    probOfVirginica  = getProbOfPlantType(noOfVirginica, totalNumOfValues);
    probOfVersicolor = getProbOfPlantType(noOfVersicolor, totalNumOfValues);
    probOfSetosa     = getProbOfPlantType(noOfSetosa, totalNumOfValues);

    % Get entropy of each plant type
    entOfVirginica  = getEntropyOfPlantType(probOfVirginica);
    entOfVersicolor = getEntropyOfPlantType(probOfVersicolor);
    entOfSetosa     = getEntropyOfPlantType(probOfSetosa);

    % Calculate the entropy
    entropy = -entOfVirginica - entOfVersicolor - entOfSetosa;

end

% Calculates the probability of each plant type
function probOfPlantType = getProbOfPlantType(noOfPlantType, totalNoOfValues)

    % To get probability of plant type, divide its number of occarances by
    % total number of values
    probOfPlantType = noOfPlantType / totalNoOfValues;

end

% Get the entropy of specific plantType
function entropyOfPlantType = getEntropyOfPlantType(probOfPlantType)

    % Check if plant type has a probability
    if probOfPlantType == 0
        % Plant type probability is 0

        % Set entropy of plant type to worse possible value
        entropyOfPlantType = 1;

        % Exit the function
        return;

    end

    % Get the entropy per plant type by multiplying the probability of
    % plant type by the log2 of plant type
    entropyOfPlantType = probOfPlantType * log2(probOfPlantType);


end
```

```
% Get the the number of occarnces of plant type in total set
function noOfPlantType = getNoOfPlantType(plantType, S)

    % Get the amount of rows in cell array
    rows = size(S, 1);

    % Initialise number of plant type counter
    noOfPlantType = 0;

    % For each row of S
    for row = 1:rows

        % Compare the second to last element of cell array which holds the plant type
        % to the plant type passed in
        if strcmp(S{row, (end - 1)}, plantType)
            % Plant types match

            % Increment counter of plant type in set
            noOfPlantType = noOfPlantType + 1;

        end
    end
end
```

```
%% Test: entropy
function entropyTest(testCase)
    %inputs:  S (set) of all values
    %process: calculate entropy of all values
    %output:  entropy value

    % load fisher iris data set
    load fisheriris.mat

    % Concatinate the data together
    fisherIrisSet = [num2cell(meas), cellstr(species)];

    % Find the entropy before any splitting
    entropyValue = entropy(fisherIrisSet);

    % Assert equal to pre calculate entropy value
    assert(isequal(entropyValue, -3));

end
```

Improvement

Improvement is opposite to entropy, instead of measuring impurity, it measures purity. This allows use to gage a gain in the sets purity, meaning the split is getting better at filtering.

This is calculated by taking away the value of entropy before the split from the value of entropy after the split. When the value is at its highest, it is when the best split has been performed and the sets are at their 'purest'. This function simply uses the entropy function as described previously, to calculate the sets entropy before a split, the split of each split

set, and then combines those to get the entropy after. Once it has the entropy before and after the split, it calculates the information gain by subtracting one from the other and the function returns the result.

```
% originalSet : complete fisheriris set without before splitting
% S1 : First set that is a result of split
% S2 : Second set that is a result of the split
% informationGain: measure of purity that is the result of calculation from
% entropy before - entropy after
function informationGain = improvement(originalSet, S1, S2)

    % Get the entropy values for all instances before, during and after
    % the set splits
    entropyBefore = entropy(originalSet);
    entropyS1     = entropy(S1);
    entropyS2     = entropy(S2);
    entropyAfter = getEntropyAfter(originalSet, S1, S2, entropyS1, entropyS2);

    % Calculate the information gain
    informationGain = entropyBefore - entropyAfter;

end

% Wrapper function to get the size of a set
function sizeOfSet = getSizeOfSet(S)
    sizeOfSet = size(S, 1);
end

```

```
% Calculates the entropy after
function entropyAfter = getEntropyAfter(originalSet, S1, S2, entropyS1, entropyS2)

    % Get the size of all sets
    sizeOfOriginalSet = getSizeOfSet(originalSet);
    sizeOfS1          = getSizeOfSet(S1);
    sizeOfS2          = getSizeOfSet(S2);

    % Calculate the entropy after
    entropyAfter = (sizeOfS1 / sizeOfOriginalSet * entropyS1) + (sizeOfS2 / sizeOfOriginalSet * entropyS2);

end
```

Max split

Max split is where most of the programs functionality comes together. It is to get the best possible split across all variable values in a set. It does so by looping through all possible sets (where the data value's set number comes into play, being the last column of the dataset), looping through all the datasets within that set, looping through all the variables and their values, splitting on every one to get the information gain and checking if that information was higher than the information gain received from the previous split. If it was, then all the values that were used to make that split are recorded, being the threshold, the set and the variable it was split on. It uses these values to make the split once all iterations are complete as these are the best values for the split. Splitting requires using the split function described previously, and to update the set index of the sets that were split that is incrementing their value respectively as they're now in new sets. Once all of this is

complete the values of the split are returned as a rule along with the set that hast the new split sets in it.

```
1   % Loops through all possible set to find the best split and returns that a
2   % as a rule
3   function [updatedSet, setRule] = maxSplit(S)
4
5       % Get number of sets
6       noOfSets = max([S{:, end}]);
7
8       % Get lowest set no
9       lowestSet = min([S{:, end}]);
10
11      % Initialise the max information gain
12      maxInformationGain = 0;
13
14      % Intialise optimal threshold value
15      optimalThreshold = 0;
16
17      % Intialise the property holder it was split on
18      property = 0;
19
20      % Initialise the optimal set holder the split was made on
21      optimalSet = 0;
22
23      % Intialise empty array for set rule
24      setRule = zeros(1, 3);
25
26      % loop through all possible sets
27      for setIndex = lowestSet:noOfSets
28
29          % Get the current set by filtering on set index
30          currentSet = getSet(S, setIndex);
31
32          % Check if current set is empty
33          if isempty(currentSet)
34
```

```
35              % Skip set
36 -            continue;
37
38 -        end
39
40          % Loop through all possible variables
41 -        for varIdx = 1:4
42
43              % Get all values in column for that variable
44 -            thresholds = unique([currentSet{:, varIdx}]);
45
46              % Get the number of columns and rows
47 -            [thresholdRowCount, thresholdColumnCount] = size(thresholds);
48
49              % Loop through all possible values
50 -            for i = 1:thresholdColumnCount
51
52                  % Make the split on the set with the variable and threshold
53                  % value
54 -                [S1, S2] = split(currentSet, varIdx, thresholds(i));
55
56                  % Check if set contains any values
57 -                if 1 == isempty(S1) || 1 == isempty(S2)
58                      % One of the sets is empty
59
60                      % Skip information calculation
61 -                    continue;
62
63 -                end
64
65                  % Get the information gain of the split
66 -                informationGain = improvement(currentSet, S1, S2);
67
```

```
68                  % Check if new information gain is greater than the max
69                  % information gain
70 -                if maxInformationGain < informationGain
71                      % New information gain is greater
72
73                      % Set the new information gain max
74 -                    maxInformationGain = informationGain;
75
76                      % Update the threshold it was split on
77 -                    optimalThreshold = thresholds(i);
78
79                      % Update the property on which the split was made
80 -                    property = varIdx;
81
82                      % Update the set on which the split was made
83 -                    optimalSet = setIndex;
84
85 -                  end
86 -              end
87 -          end
88 -      end
89
90          % Check if the optimal set has been set
91 -      if optimalSet == 0
92          % Hasn't been set, therefore no best split available
93
94          % Initialise setRule as empty
95 -        setRule = [];
96
97          % Set updated set as set passed in, unchanged due to no split
98 -        updatedSet = S;
99
100         % Quit function execution
101 -       return;
```

```
103 -       end
104
105         % Get the set to split on based on the optimal set
106 -       setToSplitOn = getSet(S, optimalSet);
107
108         % Make the split
109 -       [S1, S2] = split(setToSplitOn, property, optimalThreshold);
110
111         % Update the subset index
112 -       updatedSet = updateSubsets(S, S1, S2, setIndex);
113
114         % Get the two new set numbers that the split has been created into
115 -       set1Split = setIndex + 1;
116 -       set2Split = setIndex + 2;
117
118         % Make array of rules based on optimal set split
119 -       setRule = [property, optimalThreshold, optimalSet, set1Split, set2Split];
120
121 -   end
122
123     % Update the set ID's respectively
124   function updatedSet = updateSubsets(S, S1, S2, setIndex)
125
126       % Update the set index + 1 of all values of S = S1
127       % For all rows of original set
128 -     for i = 1:size(S, 1)
129
130           % For all rows of set 1
131 -         for j = 1:size(S1, 1)
132
133               % Check if columns from set1 are the same as origial set
134 -             if isequal(S(i, [1:5]), S1(j, [1:5]))
135
```

```
                % Increment that set index + 1
                S{i, end} = setIndex + 1;

            end
        end
    end


    % Update the set index + 2 of all values of S = S2
    % For all rows of original set
    for i = 1:size(S, 1)

        % For all rows of set 2
        for j = 1:size(S2, 1)

            % Check if columns from set1 are the same as origial set
            if isequal(S(i, [1:5]), S2(j, [1:5]))

                % Increment that set index + 1
                S{i, end} = setIndex + 2;

            end
        end
    end

    % Return the update set
    updatedSet = S;

end
```

```
% Filter out the set from the whole data set based on set ID
function filteredSet = getSet(S, setIndex)

    % Get the amount of rows in cell array
    rows = size(S, 1);

    % Initialise new cell array for the filtered set
    filteredSet = {};

    % For each row of S
    for row = 1:rows

        % Compare the last element of cell array which holds set number to
        % the set index that it is currently iterating
        if S{row, end} == setIndex

            % Copy row accross to current set
            filteredSet = [filteredSet; S(row, :)];

        end
    end
end
```

Final decision tree

The final decision tree is an amalgamation of the functions described above so that all join to create the tree. The main part of the learn decision tree is having the max split contained in a loop. This keeps splitting the set until the stopping criteria is reached. I defined the stopping criteria as if there are no further splits to be made, that is the maximum splits have been made and there are no further nodes that can be produced. This is known when there are no set rules returned by the `maxSplit` function. There is a cell array which contains all the rules, so that one could reconstruct the tree given the data set as a whole and these rules.

I also included the call to `majorityClass` within this loop to get the majority species of each split and contain them in another different cell array for later use.

```
function [rules, majorityClassesPerExistingNodes] = learnDecisionTree(meas, species)

    % Initialise a column of 1s for the initial set number to append to
    % data set
    initialSetNo = ones(size(meas, 1), 1);

    % Concatinate the data together
    set = [num2cell(meas), cellstr(species), num2cell(initialSetNo)];

    % Initialise cell array for the majority classes
    majorityClasses = [];

    % Initialise empty array for all rules
    rules = [];

    % Initialise stopping criteria as false
    stoppingCriteria = false;

    % Call max split until stopping criteria has been achieved
    while (stoppingCriteria == false)

        % Call max split on data set
        [set, setRule] = maxSplit(set);

        % Check if best set rule has been set
        if isempty(setRule)
            % Hasn't been set, therefore no best set possible

            % Set stopping criteria to true
            stoppingCriteria = true;

            % Skip the iteration
            continue;
```

```
34
35 -            end
36
37             % Get the majority class
38 -            [majorityClassSet1, majorityClassSet2] = majorityClass(set);
39
40             % Add this to array of all majority classes
41 -            majorityClasses = [majorityClasses; majorityClassSet1];
42 -            majorityClasses = [majorityClasses; majorityClassSet2];
43
44             % Add set rules to all rules
45 -            rules = [rules; setRule];
46
47 -        end
48
49         % Remove majority classes for nodes that have been split
50 -        majorityClassesPerExistingNodes = removeMajorityClasses(majorityClasses, rules);
51
52 -    end
```

Majority class

Majority class function is used to retrieve the highest number of species found in a split. This way you can label that node with that name of species, assuming that if you were to follow the rules to split on that ends up in that node, it is likely that it will be from that species. This is where `getNoOfPlantType` function is also called as you need to get the number of each plant type in a set and compare which is the most in order to get the majority within the class.

To note, I use the function `removeMajorityClasses` once all the majority classes for each split are calculated. This is because some of these splits may be split further and therefore that 'set' doesn't exist any more. After this is performed on all of the majority classes, you are left with a cell array containing the majority classes for all remaining nodes of the final decision tree.

```
% Returns the name of the majority class for the latest split rule
function [majorityClassSet1, majorityClassSet2] = majorityClass(S)

    % Initialise majority classes
    majorityClasses = [];

    % Get the highest class number from the latest split
    highestSet = max([S{:, end}]);

    % Get all values in that set
    set1 = getSet(S, highestSet - 1);

    % Get all values in the other set made from the latest split
    set2 = getSet(S, (highestSet));

    % Get the number of each plant type in each of the lastest splits
    noOfVirginicaInSet1  = getNoOfPlantType('virginica', set1);
    noOfVersicolorInSet1 = getNoOfPlantType('versicolor', set1);
    noOfSetosaInSet1     = getNoOfPlantType('setosa', set1);

    noOfVirginicaInSet2  = getNoOfPlantType('virginica', set2);
    noOfVersicolorInSet2 = getNoOfPlantType('versicolor', set2);
    noOfSetosaInSet2     = getNoOfPlantType('setosa', set2);

    % Get the majority class for both sets
    majorityClassNameSet1 = getMajorityClassPerSplit(noOfVirginicaInSet1, noOfVersicolorInSet1, noOfSetosaInSet1);
    majorityClassNameSet2 = getMajorityClassPerSplit(noOfVirginicaInSet2, noOfVersicolorInSet2, noOfSetosaInSet2);

    % Return cell array of the majority class per that split
    majorityClassSet1 = [num2cell(highestSet - 1), cellstr(majorityClassNameSet1)];
    majorityClassSet2 = [num2cell(highestSet), cellstr(majorityClassNameSet2)];

end

```

```
 % Get the majority class for that split
function majorityClassName = getMajorityClassPerSplit(noOfVirginica, noOfVersicolor, noOfSetosa)

    % Get the highest amount of each
    if (noOfVirginica > noOfVersicolor && noOfVirginica > noOfSetosa)
        % Virginica is the highest
        majorityClassName = 'virginica';

    elseif (noOfVersicolor > noOfVirginica && noOfVersicolor > noOfSetosa)
        % Versicolor is the highest
         majorityClassName = 'versicolor';

    elseif (noOfSetosa > noOfVirginica && noOfSetosa > noOfVersicolor)
         % Setosa is the highest
         majorityClassName = 'setosa';

    else
        % No majority
         majorityClassName = '-';

    end

end
```

```
%% Test: Majority class
function majorityClassTest(testCase)
    % Input: A set
    % Process: Get the majority class plant name
    % Output: that name

    % Initialise a set
    S = [num2cell(1.5), num2cell(1.7), cellstr('setosa'), num2cell(2);
         num2cell(2.5), num2cell(2), cellstr('virginica'), num2cell(3);
         num2cell(2), num2cell(2), cellstr('virginica'), num2cell(3)];

    % Get the majority class name
    majorityClassName = majorityClass(S);



    % Assert that the majority class name is virginica
    %assert(isequal(plantName, cellstr('virginica')));

    % Initialise a set
    S1 = [num2cell(1.5), num2cell(1.7), cellstr('setosa'), num2cell(2);
         num2cell(1.5), num2cell(1.7), cellstr('setosa'), num2cell(2);
         num2cell(2.5), num2cell(2), cellstr('virginica'), num2cell(3)];

    % Get the majority class name
    majorityClassName = majorityClass(S1);

    % Get the plant name
    plantName = majorityClassName(1, 2);

    % Assert that the majority class name is setosa
    assert(isequal(plantName, cellstr('setosa')));

end
```

```
function majorityClassesPerExistingNodes = removeMajorityClasses(majorityClasses, rules)

    % Get no of rows in rules
    noOfRowsInRules = size(rules, 1);

    % Get no of rows in majority classes
    noOfRowsInMajorityClasses = size(majorityClasses, 1);

    % Iterate every row in the rules array
    for i = 1:noOfRowsInRules

        % Iterate every row in the majority classes array
        for j = 1:noOfRowsInMajorityClasses

            if (isequal(num2cell(rules(i, 3)), majorityClasses(j, 1)))
                majorityClasses(j, :) = [];

                % Break out of inner loop
                break;
            end

        end

    end

    % Return the majoirty classes for the exisiting nodes
    majorityClassesPerExistingNodes = majorityClasses;

end

```

```
1     % Filter out the set from the whole data set based on set ID
2     function filteredSet = getSet(S, setIndex)
3
4         % Get the amount of rows in cell array
5 -       rows = size(S, 1);
6
7         % Initialise new cell array for the filtered set
8 -       filteredSet = {};
9
10        % For each row of S
11 -      for row = 1:rows
12
13            % Compare the last element of cell array which holds set number to
14            % the set index that it is currently iterating
15 -          if S{row, end} == setIndex
16
17                % Copy row accross to current set
18 -              filteredSet = [filteredSet; S(row, :)];
19
20 -           end
21 -      end
22 -  end
```

Classify

Classify uses the rules of the decision tree once it is completely constructed and returns the node in which the data provided as a parameter should fall into. It does this by using the max split rules, containing the variable to split on and the threshold value to compare the data passed in value of that variable. It makes a 'dummy' split that mimics the functionality of the split function, if that variable's value is less than the threshold value of that rule's split, then it goes into first set of the sets that were a result of that rule's split, if it was more then it goes into the second set. It continues this process until it reaches a point where the set wasn't split any further, meaning that it is part of this set. It the returns that node as the classification of the data.

```
function classification = classify(decisionTree, sampleDataVector)

    % Iterate each rule
    for i = 1:size(decisionTree, 1)

        % Get the column to compare
        colToCompare = decisionTree(i, 1);

        % Get threshold to compare
        thresholdToCompare = num2cell(decisionTree(i, 2));

        % Compare the value in the column to compare of the sample data
        % with the threshold
        if cellfun(@lt, sampleDataVector(1, colToCompare), thresholdToCompare)
            % Less than threshold

            % Goes into the first set of split
            sampleDataNode = decisionTree(i, 4);

        elseif cellfun(@ge, sampleDataVector(1, colToCompare), thresholdToCompare)
            % Greater than or equal to threshold

            % Goes into the second set of the split
            sampleDataNode = decisionTree(i, 5);
        end

        % Check to see if the node that the sample data has fallen into was
        % split futher
        if (decisionTree((i + 1), 3) ~= sampleDataNode)
            % Node has not been split further

            % Break out of the loop
            break;
        end


    end

    % Return the node of comparision
    classification = sampleDataNode;

end
```

```
%% Test: Classify
function classifyTest(testcase)
    % Input: decisiontree, test data
    % Process: get the node it should fall into
    % Output: that node

    % Initialise sample data array
    sampleData = [num2cell(5.1), num2cell(3.5), num2cell(1.4), num2cell(0.2)];

    % Initialise decision tree array
    rules = [3 3 1 2 3; 3 5.2 3 4 5];

    % Get the classification
    classification = classify(rules, sampleData);

    % Assert that the the classification is correct should be in 2
    assert(isequal(classification, 2));

end
```

Final Result of Tree

I model the final tree as a set of rules in a matrix with all the information needed to reconstruct the tree. First column is the variable that it split on, in our case which column it found the best split for that set, the second column is the threshold of that split in that variable, the last three columns are the set it split and into which two new sets. From this information you could rebuild the tree on the data and it also allows you to use this information as analysis on the dataset as you are given the defining qualities. As one can see from the table, there are 21 one node in the whole tree. I have also made another matrix combining the node and the label of the plant type that has majority in that node, as seen in picture two.

	1	2	3	4	5
1	3	3	1	2	3
2	3	5.2000	3	4	5
3	4	1.5000	4	6	7
4	4	1.9000	7	8	9
5	2	3.1000	8	10	11
6	4	1.8000	10	12	13
7	2	2.9000	12	14	15
8	1	6	14	16	17
9	3	5	17	18	19
10	4	1.6000	19	20	21

	1	2
1	2	'setosa'
2	5	'virginica'
3	6	'versicolor'
4	9	'virginica'
5	11	'versicolor'
6	13	'virginica'
7	15	'versicolor'
8	16	'virginica'
9	18	'versicolor'
10	20	'virginica'
11	21	'versicolor'

Other functions included in the solution

The tests which are under `classificationWithDecisionTreesTests` and `initate` which passes the data set into the `learnDecisionTree` function and also gives an example of the `classify` function working with some dummy data.

```
function initiate

    % Load the fisher iris set
    load fisheriris.mat

    % Get the decision tree and majority nodes it produces
    [decisionTree, majorityNodes] = learnDecisionTree(meas, species);

    % Create sample data
    sampleData = [num2cell(6.2), num2cell(2.2), num2cell(4.5), num2cell(1.5)];

    % Get the classification of the sample data
    classification = classify(decisionTree, sampleData);

    % Loop through majoirty node's rows
    for i = 1:size(majorityNodes, 1)

        % Check if the current iteration's node is equal to the
        % classification
        if isequal(num2cell(classification), majorityNodes(i, 1))
            % Is the same node

            % Get the majority plant name
            plantName = majorityNodes(i, 2);

            % Exit loop
            break;

        end
    end

end
```

P2.2: Q-LEARNING

Transition Function

The task was to create a function that, given a state and an action as parameters, will return the appropriate state; effectively transitioning the agent from one state to another. Actions are given in numeric form from 1 to 4 representing north, east, south and west respectively and the states are limited to a 11 squared grid. Due to this format of actions and states I figured the transition function could be achieved by a 'look-up table' of 11x4. Each row being the state and each column being the action. From there you can assign the value of nxn to whatever that state and that action would result in. In this implementation, one is able to get the desired next state by accessing it like a regular matrix with the state as the row and action as column.

I believe this to be one of the better ways of solution as opposed to a combination of `if` or `switch` statements as it means the logic can always stay the same, if you wanted to change the grid-world or the amount of actions available you would simply have to change the matrix dimensions and values accordingly. In the test created for this function, I know the expected state from a given action and state as I know the grid-world and how to navigate it. Therefore, I call the function with known state and known action to get the known next state and compare the actual result with the expected.

```
% Returns the next state based on an action and state
function nextState = deterministicTransitionFunction(state, action)

    % Create the environment
    environment = [
        4  1  1  1;
        2  2  2  2;
        6  3  3  3;
        7  4  4  4;
        9  5  2  5;
        11 6  3  6;
        7  8  7  7;
        8  9  8  7;
        9  10 5  8;
        10 11 10 9;
        11 11 6  10
    ];

    % Get the new state
    nextState = environment(state, action);

end
```

```
%% Test: deterministicTransitionFunction
function deterministicTransitionFunctionTest(testcase)

    % Define the action and current state
    action = 1;
    state  = 1;

    % Define the next state
    expectedNextState = 4;

    % Get the next state from the deterministic transition function
    actualNextState = deterministicTransitionFunction(state, action);

    % Assert that you get the expect state
    assert(isequal(actualNextState, expectedNextState));

    % Define the action and current state
    action = 2;
    state  = 1;

    % Define the next state
    expectedNextState = 1;

    % Get the next state from the deterministic transition function
    actualNextState = deterministicTransitionFunction(state, action);

    % Assert that you get the expect state
    assert(isequal(actualNextState, expectedNextState));

end
```

Starting State

I achieved creating a function that initialises any starting state apart from the goal state which is state two in two parts. Firstly I used `randi` to generate a uniformly distributed random integer, passing the range (1 to 11) and the size (1x1) of the matrix it should create. Secondly, I check to see if the starting state has been generated as two, if it has then I use recursion to call itself to re-generate a number. This will continue to happen until 2 has not been generated which is usually after the first call. Although there are other ways to implement this, I prefer this method as it is clean, simple and easily testable. The test written for this calls the function and asserts 4 cases: that the number is whole, that the number is greater than or equal to 1, that the number is less than or equal to 11 and that it is not equal to 2.

```
1     % Get the starting state which cannot be 2 as 2 is the finish state
2     function startState = startingState()
3
4         % Get a number between 1 and 11
5 -       startState = randi([1 11], 1, 1);
6
7         % Check if number is 2 (finish state)
8 -       if startState == 2
9 -           startState = startingState();
10 -      end
11
12 -  end
```

```
36    %% Test: startingState
37    function startingStateTest(testcase)
38
39        % Get the starting state
40 -      startState = startingState();
41
42        % Test if number is whole
43 -      assert(mod(startState, 1) == 0);
44
45        % Test is number is between 1 and 11
46 -      assert(startState <= 11);
47 -      assert(startState >= 1);
48
49        % Test if number is not two
50 -      assert(startState ~= 2);
51
52 -   end
```

Reward Function

The reward function's purpose is to signify reaching the goal state and, in later use of the program, it is used to update the values of the state's value around it in order for it to learn

the shortest path towards the goal. The function is required to return value 0 as a reward unless the action and state show that is going to move into the goal state (action 3 and state 5 moves the agent into state 2). Therefore, it is only a matter of checking the parameters, action and state, values to see if they move the agent into the goal state, if so, return 10, else always return 0. The test was implemented the same way. We know if state and action equal 5 and 3 then the reward is 10, that being the expected value and so I call the function with the parameters and check against that expected value. Otherwise I choose some arbitrary numbers that are not those and check against the expected value of 0.

```
1    function reward = getReward(state, action)
2
3        % Initialise reward as 0
4 -      reward = 0;
5
6        % Check the state and action values
7 -      if (state == 5 && action == 3)
8            % Going to final state
9 -          reward = 10;
10 -     end
11
12 -     end
```

```
%% Test: rewardFunction
function getRewardTest(testcase)

    % Initialise a state and an action
    % State and action represent moving to final state
    state = 5;
    action = 3;

    % Initialise the reward for reaching final state
    expectedReward = 10;

    % Get the actual reward and function
    actualReward = getReward(state, action);

    % Assert the reward is 10
    assert(isequal(actualReward, expectedReward));

    % State and action represent moving not to a final state
    state = 4;
    action = 1;

    % Initialise the reward for reaching a the non final state
    expectedReward = 0;

    % Get the actual reward
    actualReward = getReward(state, action);

    % Assert the reward is equal to 0
    assert(isequal(actualReward, expectedReward));

end
```

Q-function Table

The purpose of the Q-function table is to maintain the values of that state and action's relation to the goal state. This is used in conjunction with the `update` function which updates the value respectively based on the reward, state and action. This function initialises the matrix of 11x4, representing the states as rows and actions as columns similar to my implementation of the `deterministicTransitionFunction` table, with random values between 0.01 and 0.1. I achieved this by using the `rand` function with an lower band, upper band and matrix dimensions to control the size and limits of the values within. It also asks to plot the matrix as a 3D surface graph, which I extracted out into a separate function `plotQFunctionInitValues` as later on in the program this function is called multiple times and therefore a graph would be created each time. Instead I call this once at the end and display it with the other graphs for the `trail` and `experiment` functions. (plot shown in **experiment section**)

I also have written a test to ensure that the sizes and values are correct when initialising. I have done this by calling the function to get the table, then checking the size against the expected size and also loop through the values getting the minimum and maximum values for each row and checking if that they're not above or below the values specified, if they

are, they set a boolean flag which is checked against at the end of the test to make sure it hasn't been set.

```
function qTable = initQ()

    % Initialise the upper and lower bound of the range
    lowerBound = 0.01;
    upperBound = 0.1;

    % Get a number between 0.01 and 0.1
    qTable = (upperBound - lowerBound).*rand(11, 4) + lowerBound;

end
```

```
%% Test: initQ
function initQTest(testcase)

    % Initialise qTable
    initQTable = initQ();

    % Get the size of the qTable
    sizeOfQTable = size(initQTable);

    % Set the expected size
    expectedSizeOfQTable = [11 4];

    % Assert sizes are as expected
    assert(isequal(sizeOfQTable, expectedSizeOfQTable));

    % Initialise flag
    valueAbove1Flag = false;

    for i = 1:11

        % Get the max value of the row
        maxRowValue = max(initQTable(i, :));

        % Get the min value of the row
        minRowValue = min(initQTable(i, :));

        % Check if the maximum value of the row is above one
        if maxRowValue > 0.1
            % Value is above 0.1

            % Set the flag
            valueAbove1Flag = true;

```

```
119 -           elseif minRowValue < 0.01
120                 % Value is below 0.01
121
122                 % Set the flag
123 -               valueAbove1Flag = true;
124 -           end
125
126 -       end
127
128         % Assert that the flag was never set true
129 -       assert(isequal(valueAbove1Flag, false));
130
131 -   end
```

Action Selection

A e-greedy action selection function by nature always tries to pick the highest value of the possible actions in that state to move the agent (hopefully) closer towards the goal state. The function is passed in the qTable and state as parameters, from that I get all possible values for that state by getting all the values of the corresponding row. Then one can simply take the highest value and the index associated with that value and use that index as the next action to take. To add the 10% random chance of moving into any state, I initialised a number between 1 and 10 every time it is called, this representing each 10% probability, then I check if the number is equal to 10, meaning that the 10% chance has been achieved and so I further pick another random number between 1 and 4 which will then be the action. Note, the if statement could have been any number between 1-10 as all have equal probability of being chosen 10% of the time, but for clarity's sake I chose 10 because I wanted 1-9 to represent 90% of the time it didn't get chosen.

```
% An e-greedy action selector function
function action = actionSelection(qTable, state)

    % Get all action values
    actionValues = qTable(state, :);

    % Get random number between 1 and 10
    randomNumber = randi([1 10]);

    % Check if number is equal to 10
    if randomNumber == 10

        % Choose random number between
        action = randi([1 4]);

        % Exit
        return;

    end

    % Get the column of the maximun value
    [value, column] = max(actionValues);

    % Get the action that corresponds to the column
    action = column;

end
```

Update

This function is to update the resulting state from the current state and action's value in the qTable using the Q-Learning update equation. This function applies a discount rate and learning rate to the current value of the table so that the new value is based on the reward and next state it has landed in. This means it will decrease over time if the reward is 0 and increase when the reward is 10. That means that over the course of the trails the values surrounding the goal and the path to the goal with increase as the goal state goes towards a value of 10 in the qTable allowing that to trickle down through the table to the values of resulting actions to states that lead to it. This twinned with the e-greedy action selection means that it will get better over time at moving towards the goal state faster as the values to it increase and ones that don't decrease.

The values needed for the equation are: the discount rate, learning rate, current value of the q table state and action and the max value of the resulting state's actions. When plugged into the formula it updates the current value correctly. Once it has that updated value, I simply assign it to the position in the qtable that corresponds to the state and action passed in as parameters.

```
function qTable = updateQTable(qTable, state, action, resultingState, reward)

    % Initialise discount rate and learning rate
    discountRate = 0.9;
    learningRate = 0.2;

    % Get the qTable value that we're in
    currentQTableValue = qTable(state, action);

    % Get the max value of the resulting state's row
    maxValueOfResultingState = max(qTable(resultingState, :));

    % Calculate the updated value according to the learning rule
    updatedValue = currentQTableValue + learningRate * (reward + (discountRate * maxValueOfResultingState) - currentQTableValue);

    % Update the table with the new value
    qTable(state, action) = updatedValue;

end
```

Episode

An episode represents an agent's full journey through the world from start to goal. To do this, I joined all the previous functions written so that a starting state is initialised along with the qTable. From there I created a while loop with an exit condition of the reward being equal to 10, meaning that the goal state has been reached. Within the loop an action is selected using the e-greedy action selection function, this is used to get the next state. The reward can then gathered from the state and action which along with the next state is passed into the update function. If the reward isn't 10, then the next state is assigned to the current state (effectively moving the agent in the world) and the loop is complete. To note, the number of steps it takes the agent to reach the goal state from the starting state is also recorded throughout the episode for later use.

```
function [numberOfSteps, qTable] = episode(qTable)

    % Get starting state
    state = startingState();

    % Initialise reward
    reward = 0;

    % Initialise number of steps
    numberOfSteps = 0;

    % Loop through episode until the goal state is reached
    while reward ~= 10

        % Get the action
        action = actionSelection(qTable, state);

        % Get the resulting state
        nextState = deterministicTransitionFunction(state, action);

        % Get the reward of the resulting action
        reward = getReward(state, action);

        % Update the qTable
        qTable = updateQTable(qTable, state, action, nextState, reward);

        % Increment number of steps
        numberOfSteps = numberOfSteps + 1;

        % Move to the next state
        state = nextState;

    end
end
```

Trial

A trial represents a given number of episodes and the result from them as the qTable is reused for each episode, meaning it can learn upon its old findings from the last episode. This meant the implementation of the `trial` function was simply calling episode 100 times in a loop and recording the number of steps taken for the episode to reach the goal and the updated q table from that episode, the q table was then fed back into the next iterations episode.

What's important here is not the implementation but the results from it. As you can see from the graph, over time the number of steps tends towards 0 as it keeps on improving (from the updated values in the qtable). It starts off quite eratic as there is either no or little previous data to use. It is only until around 20 episodes roughly does it start to improve and flatten. It will never be completely flat due to the randomness in the action selection, meaning it will never be completely efficient in always choosing the perfect path towards the goal state, but this is important to keep as it is important in the learning process. Sometimes the algorithm would get stuck on fake values due to the fact that the would is initialised randomly so some values will be higher but for no reason. If you were to

increase the number of trails then you would see it improve more so though but as previously described it will never be perfect. (plot shown in **experiment section**)

```
function episodeStepCountTable = trial

    % Initialise qTable
    qTable = initQ();

    % Initialise matrix to hold amount of steps per episode
    episodeStepCountTable = [];

    % Iterate 100 episodes
    for i = 1:100

        % Run the episode
        [episodeStepCount, updatedQTable] = episode(qTable);

        % Update the step count table
        episodeStepCountTable = [episodeStepCountTable, episodeStepCount];

        % Reassign the updated qTable
        qTable = updatedQTable;

    end

end
```

Experiment

This exercise was to get more informative results back so that one could analyse the results of the algorithm over an extended period of time. Firstly, I achieved this by calling trail 500 times to get a 500x100 table of the number of steps, each row being a trail and each row being that episode's number of steps. From that I iterated over each column to get the mean and standard deviation of that episode's step count. Using these results I could then plot the number of steps per episode number against the mean of that episode's number of steps over the 500 trials and the standard deviation of the steps from the mean.

The result shows us that after episode 30 the mean and standard deviation of the steps has greatly reduced representing the law of diminishing returns whereby no matter the further amount of episodes that the algorithm has 'bottomed-out' and won't improve further. Again both values come together as it reaches the last 70% of the episodes due to it 'learning' the best route through the world but in the first 30% it is still fairly erratic as it has minimal data to go upon and still trying multiple different routes through the world to get to the goal.

```
function experiment

    % Initialise matrix to hold mean per episode
    meanOfTrials = zeros(1, 100);

    % Initialise matrix to hold standard deviation per episode
    standardDeviationOfTrials = zeros(1, 100);

    % Initialise trail step vector
    trialStepVector = zeros(500, 100);

    % Iterate 500 episodes
    for i = 1:500

        % Run the trial
        trialStepVector(i, :) = trial();

    end

    % Iterate each episode associated steps
    for i = 1:100

        % Get the mean of the trials
        meanOfTrials(i) = mean(trialStepVector(:, i));

        % Get the standard deviation of the trials
        standardDeviationOfTrials(i) = std(trialStepVector(:, i));

    end

```

```
31          % Set x axis
32 -        x = [1:1:100];
33
34 -        figure;
35
36          % Sub plot the first graph
37 -        subplot(2, 2, 1);
38
39          % Plot the episode count over the 100 intervals
40 -        plot(x, trialStepVector(1, :));
41 -        title('Q-learning performance improvement');
42 -        xlabel('Episode number');
43 -        ylabel('Number of steps');
44
45          % Sub plot the second graph
46 -        subplot(2, 2, 2);
47
48          % Plot the mean and standard deviation of each episode count for 500
49          % trials
50 -        shadedErrorBar(x, meanOfTrials, standardDeviationOfTrials, 'black');
51 -        title('Mean and standard deviation of Q-learning performance over 500 trials');
52 -        xlabel('Episode number');
53 -        ylabel('Number of steps');
54
55          % Sub plot the third graph
56 -        subplot(2, 2, 3);
57
58          % Initialise qTable
59 -        qTable = initQ();
60
61          % Plot the initial qTable values
62 -        plotQFunctionInitValues(qTable);
63
64 -    end
```

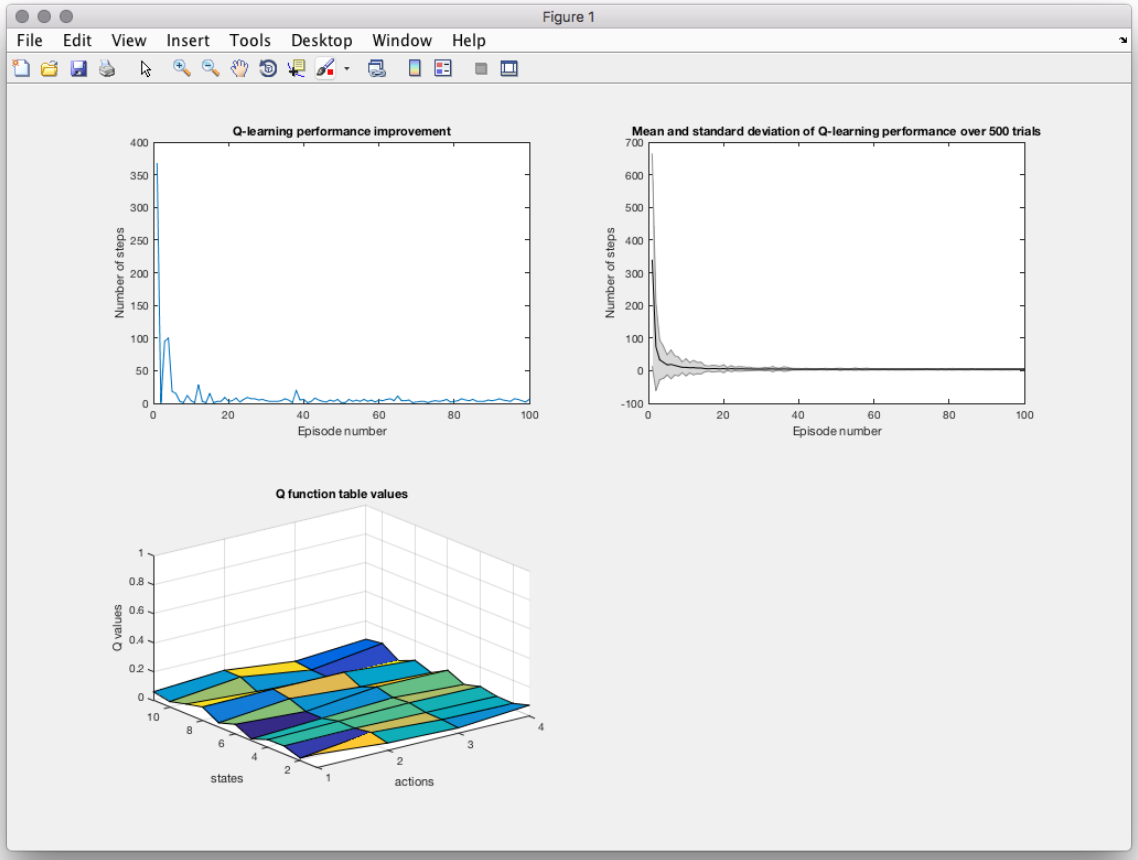


Figure 1
File Edit View Insert Tools Desktop Window Help
Q-learning performance improvement
Number of steps
Episode number
Mean and standard deviation of Q-learning performance over 500 trials
Number of steps
Episode number
Q function table values
Q values
states
actions

P2.3: NSM-LEARNING

A POMDP

The first section of this assignment was to implement the functions transition, observation and rndStartState. All of these combined help the agent navigate around a POMDP. As we had already created the functions transition and rndStartState in the previous assignment, I have copied across my definitions and reasoning for them in this section.

Observation

The purpose of this function is to return the agents observation given the state that is provided. I figured that the as the states are numbers from 1 to 11, these could correspond to the rows in a matrix with their value being the observation. Then to return the observation for the state, one simply needs to pass in the state as the first parameter of the matrix and a 1 as the second to get that rows first value which is the observation. I implemented it like this because I think it is the easiest to maintain and manage. If the POMDP was to change, then to change the state's observation value, you would only have to change that state's value row index and all of the functionality can remain the same.

```
function observation = observation(state)

 % Create the environment
    observations = [
        14;
        14;
        14;
        10;
        10;
        10;
        9;
        5;
        1;
        5;
        3
    ];

    % Get the observationr value from the state
    observation = observations(state, 1);

end
```

```
%% Test: observationTest
function observationTest(testcase)

    % Initialise state
    state = 11;

    % Initialise expected observation
    expectedObservation = 3;

    % Get the actual observation
    actualObservation = observation(state);

    % Assert that you get the expect observation
    assert(isequal(actualObservation, expectedObservation));

end

```

Transition Function

The task was to create a function that, given a state and an action as parameters, will return the appropriate state; effectively transitioning the agent from one state to another. Actions are given in numeric form from 1 to 4 representing north, east, south and west respectively and the states are limited to a 11 squared grid. Due to this format of actions and states I figured the transition function could be achieved by a 'lookup table' of 11x4. Each row being the state and each column being the action. From there you can assign the value of nxn to whatever that state and that action would result in. In this implementation, one is able to get the desired next state by accessing it like a regular matrix with the state as the row and action as column.

I believe this to be one of the better ways of solution as opposed to a combination of `if` or `switch` statements as it means the logic can always stay the same, if you wanted to change the grid-world or the amount of actions available you would simply have to change the matrix dimensions and values accordingly. In the test created for this function, I know the expected state from a given action and state as I know the grid-world and how to navigate it. Therefore, I call the function with known state and known action to get the known next state and compare the actual result with the expected.

```
% Returns the next state based on an action and state
function nextState = transition(state, action)

    % Create the environment
    environment = [
        4  1  1  1;
        2  2  2  2;
        6  3  3  3;
        7  4  4  4;
        9  5  2  5;
        11 6  3  6;
        7  8  7  7;
        8  9  8  7;
        9  10 5  8;
        10 11 10 9;
        11 11 6  10
    ];

    % Get the new state
    nextState = environment(state, action);

end
```

```
22      %% Test: transition
23      function transition(testcase)
24
25          % Define the action and current state
26 -        action = 1;
27 -        state  = 1;
28
29          % Define the next state
30 -        expectedNextState = 4;
31
32          % Get the next state from the deterministic transition function
33 -        actualNextState = transition(state, action);
34
35          % Assert that you get the expect state
36 -        assert(isequal(actualNextState, expectedNextState));
37
38          % Define the action and current state
39 -        action = 2;
40 -        state  = 1;
41
42          % Define the next state
43 -        expectedNextState = 1;
44
45          % Get the next state from the deterministic transition function
46 -        actualNextState = deterministicTransitionFunction(state, action);
47
48          % Assert that you get the expect state
49 -        assert(isequal(actualNextState, expectedNextState));
50
51 -      end
```

Start State Function

I achieved creating a function that initialises any starting state apart from the goal state which is state two in two parts. Firstly I used `randi` to generate a uniformly distributed random integer, passing the range (1 to 11) and the size (1x1) of the matrix it should create. Secondly, I check to see if the starting state has been generated as two, if it has then I use recursion to call itself to re-generate a number. This will continue to happen until 2 has not been generated which is usually after the first call. Although there are other ways to implement this, I prefer this method as it is clean, simple and easily testable. The test written for this calls the function and asserts 4 cases: that the number is whole, that the number is greater than or equal to 1, that the number is less than or equal to 11 and that it is not equal to 2.

```
% Get the starting state which cannot be 2 as 2 is the finish state
function startState = rndStartState()

    % Get a number between 1 and 11
    startState = randi([1 11], 1, 1);

    % Check if number is 2 (finish state)
    if startState == 2
        startState = rndStartState();
    end

end
```

Random Episodes

The purpose of rndEpisode function was to create episodes that weren't learning at all. Simply, they were initialises a random start state, and for every transition that did not land them onto the goal state, initialise a random action and move accordingly.

To implement this was very simple. I created a while loop that keeps iterating until the state isn't the value of two, which is the goal state. Within I use the functions previously created along with the `randi` function which initialises a random integer from values 1 to 4 for the action. Every iteration of the loop it selects a new random action and moves to that state using the transition function whilst logging the current observation. Once it has landed on the goal state it iterates back through all the steps, giving the last step a reward of 10 to be discounted throughout the previous steps by multiplying it by discount rate of 0.9, simulating the progression as the learning rate increases the closer it gets towards the goal. After, it checks the number of steps taken to reach the goal state. If it is less than 20 then it will fill the remaining steps before the first step with 0s by creating a matrix of the number of steps subtracted by 20. If it is more than 20 then it will assign to itself only the last 20 rows of the matrix. This functionality is to ensure no matter the step count, a 20x3 matrix is always returned. Once the matrix is formatted it is returned along with the number of steps this episode has taken.

```
function [numOfSteps, episode] = rndEpisode()

    % Get the starting state
    state = rndStartState();

    % Initialise episode matrix
    episode = [];

    % Loop through episode until the goal state is reached
    while state ~= 2

        % Generate a random action
        action = randi([1 4], 1, 1);

        % Get the next state based on state and action
        nextState = transition(state, action);

        % Get the observation
        currentObservation = observation(state);

        % Update the episode
        episode = [episode; currentObservation action 1.0];

        % Move to the next state
        state = nextState;

    end

    % Get number of rows of episode matrix
    [rows columns] = size(episode);

    % Iterate each row of the episode matrix
    for step = rows:-1:1

```

```
34
35          % Check to see if at last step in episode
36 -        if step == rows
37              % Give that a discountedReward of 10
38 -            episode(step, 3) = 10;
39
40              % Skip iteration
41 -            continue;
42
43 -        end
44
45          % Get the discountedReward of the previous step
46 -        prevDiscountedReward = episode((step + 1), 3);
47
48          % Do the discounted rate multiplication
49 -        discountedReward = prevDiscountedReward * 0.9;
50
51          % Assign that to this iterations step discountedReward
52 -        episode(step, 3) = discountedReward;
53
54 -    end
55
56      % Get the number of steps
57 -    numOfSteps = rows;
58
59      % Check to see if it took less steps than 20 to reach goal
60 -    if rows < 20
61          % Took less steps
62
63          % Initialise matrix of zeros that is the difference between rows
64          % and matrix size
65 -        zeroMatrix = zeros((20 - rows), 3);
66
```

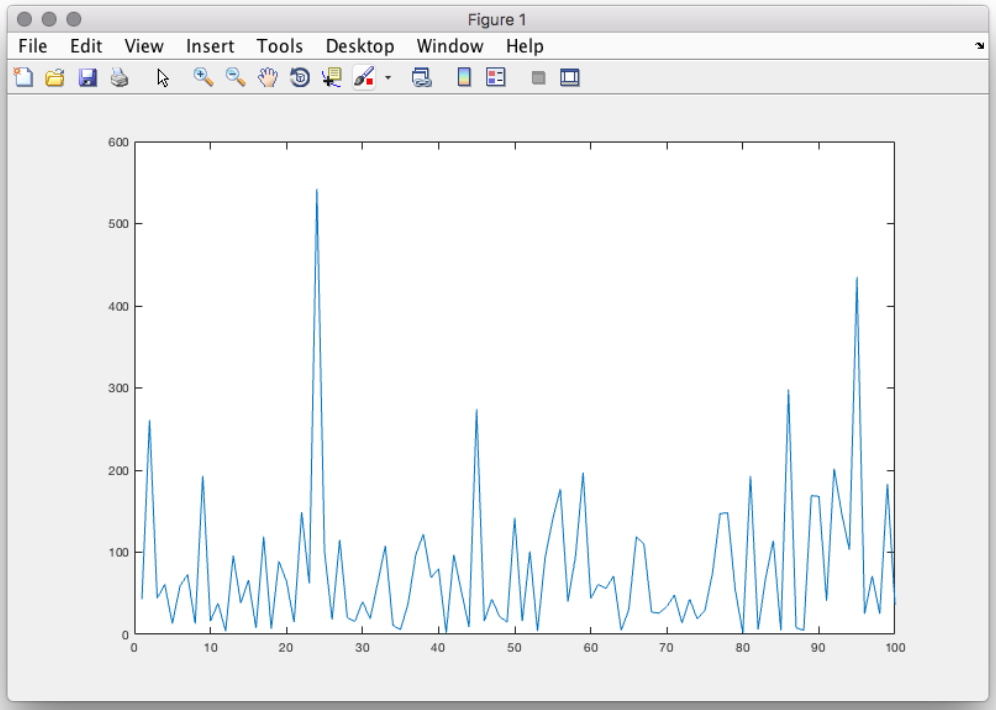
```
67              % Padd out array
68 -            episode = [zeroMatrix; episode];
69              
70 -        elseif rows > 20
71              % Took more than 20 steps
72              
73              % Get the last 20 steps
74 -            episode = episode((rows - 19):rows, :);
75              
76 -        end
77              
78 -    end
```

Random trials

This function takes the number of episodes 1 trial should run for as a parameter and uses that as the count for the iterations. It initialises two matrices, one 3 dimensional one using `zeros` function as well as the number of episodes as 3rd dimension index and the other is a matrix to hold the number of steps each episode has take, which is also created with the parameter as the index. It then loops for the number of episodes specified, every iteration it calls `rndEpisode` to get the number of steps and the episodes matrix. Once `rndEpisode` has ran through completely, it will add it to this current iterations long term memory matrix, using the iteration counter as the index for the 3rd dimension. It adds the number of steps that the current iteration `rndEpisode` has taken to the number of steps matrix so that both can be returned once the trial is completed.

As both of these matrices are returned and share the same dimensions, they can be called with the `plot` function directly from the command window that Matlab offers. This means you can see the number of the steps on the y axis and the number of episodes along the x axis and compare (as seen below). As you can see, the `rndTrail` does exactly what it is supposed to, there is no correlation between the y and x axis indicating that there is no learning happening. As the number of episodes increases the number of steps that it takes to complete it does not decrease. Passing in a higher figure into the `rndTrails` function gives a larger value on the x axis and ultimately the same on the y as there is more chance one episode will run for a longer amount of time.

```
function [numOfStepsMatrix, LTM] = rndTrial(numOfEpisodes)

    % Initialise LTM matrix
    LTM = zeros(20, 3, numOfEpisodes);

    % Initialise the number of steps matrix for each episode
    numOfStepsMatrix = zeros(1, numOfEpisodes);

    % Iterate amount of times to get desired amount of episodes
    for i = 1:numOfEpisodes

        % Get the episode
        [numOfSteps, episode] = rndEpisode();

        % Add this iterations episode to the long term memory
        LTM(:, :, i) = episode;

        % Add this episode's step count to the number of steps matrix
        numOfStepsMatrix(i) = numOfSteps;

    end

end
```



Proximity

The `proximity` function's purpose was to check if a given episode's step in the long term memory was the same as the observation passed in. If there is a match, then a proximity of 1 is rewarded and it goes on to further check the subsequent step's action and observation from the long term memory is the same as the action and observation of the short term memory. If there is a match, then the proximity is increased and the iteration backwards continues, else if there isn't a match, then the function is terminated. In theory, this gives use a numerical figure in which we can deduce how similar a set of actions and observations are, and if their discounted reward is high, one can assume that these actions lead to the goal state. As with many of the functions in this assignment, I have created a test to make sure that the basic functionality is working as expected and can will continue to work even after changes had been made to the function itself. In the test for `proximity`, I created a LTM and STM that are identical. Then I passed into the `proximity` function both the LTM and the STM, as well as the observation that was last in both of these matrices, and the step count being 20 for the last step and one for the episode index (as I only created one episode to test with). With these parameters, I could assume that the `proximity` function was working if it returned 20 as a proximity, this is because as they were identical and the last step and observation were passed in, then it should loop back through the whole STM and LTM and award a point for each step of which there are 20. As I created the LTM and STM, I also could check that it returned the correct LTM step, knowing it would be the same as the last step of both matrices.

```
% LTM:          A 3D matrix of the same format as the LTM matrix
% episodeIndex: Index of one of the episodes in the LTM matrix.
% stepIndex:    Index of one of the steps in that episode.
% episode:      2D matrix of the same format as the episode matrix above representing the current content of the algorithms STM
% observation:  Observation value
% proximity:    Value indicating how near the step at the given point in the LTM matrix is to the given STM episode and observation.
% LTMStep:      Vector describing the matching LTM step, observation, action and discounted reward.
function [proximity, LTMStep] = proximity(LTM, episodeIndex, stepIndex, STM, observation)

    % The proximity function should first initialise the proximity to 0.0.
    proximity = 0.0;

    % Get the step of the LTM
    LTMStep = LTM(stepIndex, :, episodeIndex);

    % Get the recorded observation of episode and step index
    recordedObservation = LTM(stepIndex, 1, episodeIndex);

    % Check if recorded observation is the same as observation passed in
    if (recordedObservation ~= observation)
        % Not the same

        % Terminate function
        return;
    end

    % Values are the same

    % Set proximity value
    proximity = 1;
```

```
    % Check if step index
    if (stepIndex == 1)
        % Can't compare any further back

        % terminate function
        return;

    end

    % Get the number of rows for STM
    [STMRows, STMColumns] = size(STM);

    % Iterate all previous steps in LTM
    while (stepIndex ~= 1 && STMRows ~= 1)

        % Decrement the step indexs
        stepIndex = stepIndex - 1;
        STMRows = STMRows - 1;

        % Get observation from LTM and STM for this step
        LTMobservation = LTM(stepIndex, 1, episodeIndex);
        STMobservation = STM(STMRows, 1);

        % Get action from LTM and STM for this step
        LTMaction = LTM(stepIndex, 2, episodeIndex);
        STMaction = STM(STMRows, 2);

        % Check if the observations and actions match
        if (LTMobservation == STMobservation && LTMaction == STMaction)
            % Match

            % Increment proximity
            proximity = proximity + 1;
```

```
66 -            else
67                  % No match
68
69                  % Terminate function
70 -                return;
71 -            end
72 -        end
73 -    end
```

```
73      %% Test: proximity
74      function proximityTest(testcase)
75
76          % Initalise LTM
77 -        LTM = [0         0         0;
78                 0         0         0;
79                 0         0         0;
80                 0         0         0;
81                 0         0         0;
82                 0         0         0;
83                 0         0         0;
84                 0         0         0;
85                 0         0         0;
86                 0         0         0;
87                 0         0         0;
88                 0         0         0;
89                 0         0         0;
90              10.0000    1.0000    5.3144;
91               1.0000    3.0000    5.9049;
92              10.0000    4.0000    6.5610;
93              10.0000    2.0000    7.2900;
94              10.0000    2.0000    8.1000;
95              10.0000    2.0000    9.0000;
96              10.0000    3.0000   10.0000];
97
```

```
98      % Initialise STM with same values above to test its working
99 -    STM = [0         0         0;
100            0         0         0;
101            0         0         0;
102            0         0         0;
103            0         0         0;
104            0         0         0;
105            0         0         0;
106            0         0         0;
107            0         0         0;
108            0         0         0;
109            0         0         0;
110            0         0         0;
111            0         0         0;
112       10.0000    1.0000    5.3144;
113        1.0000    3.0000    5.9049;
114       10.0000    4.0000    6.5610;
115       10.0000    2.0000    7.2900;
116       10.0000    2.0000    8.1000;
117       10.0000    2.0000    9.0000;
118       10.0000    3.0000   10.0000]
119
120     % Initialise expected values
121 -   expectedProximity = 20;
122 -   expectedLTMStep   = [10.0000    3.0000   10.0000];
123
124     % Get actual values from proximity
125 -   [actualProximity, actualLTMStep] = proximity(LTM, 1, 20, STM, 10);
126
127     % Assert proximities are the same
128 -   assert(isequal(actualProximity, expectedProximity));
129
130     % Assert LTM steps are the same
131 -   assert(isequal(actualLTMStep, expectedLTMStep));
```

```
132
133 - end
```

Nearest Sequence Memory

The purpose of this function was to create a Knearest matrix with all the complete LTM step's values as well as the proximity value associated with that LTM step. This matrix would be a maximum size of K and would contain only the highest proximity values with their LTM step.

I achieved this by first getting the whole size of the LTM, this is because I would need to know the number of episodes and the number of steps per episode in later loop counters. I created an outer loop which would go through all episodes of the LTM, checking first if the LTM episode has been created, skipping if it hasn't in order to save useless iterations. It would then enter an inner loop which would loop through all steps of that episode calling proximity for each step, passing in the whole LTM, the current episode index, the current

step index, the STM and the observation for this step. A check is made to ensure the returning proximity value is above one, if so I then check the size of the KnearestSteps matrix to make sure that it isn't full (of size K). If it is, then it will further get the minimum value of the proximity column along with the row index and replace that row with the current iterations values from the proximity and the LTM step. If the KnearestSteps matrix is smaller than size K then it can simply add it into the matrix.

This matrix signifies the highest, and therefore best, steps and their values to be checked against in order to make a prediction of the next best step later on in the program.

```
function KNearestSteps = KNearest(LTM, K, observation, STM)

    % Get number of episodes
    [steps, values, episodes] = size(LTM);

    % Initialise matrix to hold k nearest neighbours
    KNearestSteps = [];

    % Loop through all episodes and all steps of LTM
    for episode = 1:episodes

        % Get the last step of this episode from the LTM
        finalStep = LTM(20, :, episode);

        % Check if observation is 0
        if finalStep(1,1) == 0
            % LTM hasn't been set yet

            % Skip this LTM
            continue;

        end

        % Loop through all steps per episode
        for step = 1:steps

            % Get the proximity value for the current iteration
            [currentProximityValue, LTMStep] = proximity(LTM, episode, step, STM, observation);

            % Check if proximity value is greater than 0
            if currentProximityValue > 0

                % Get the size of KNearestNeighbours
                [rows, columns] = size(KNearestSteps);
```

```
35
36                  % Check if rows is the same as K
37 -                if (K == rows)
38                      % Maximum number of neighbours reached
39
40                      % Get the minimum value of proximity from the nearest
41                      % neighbours matrix
42 -                    [minimumProximityValue, index] = min(KNearestSteps(:, 4));
43
44                      % Check if minimum value is smaller than current value
45 -                    if (minimumProximityValue < currentProximityValue)
46                          % Replace the row with the smallest proximity value
47                          % with this proximity value and steps
48
49                          % Add the current proximity value onto the LTMStep
50 -                        LTMStep = [LTMStep currentProximityValue];
51
52                          % Add this LTMStep to the KnearestNeighbours matrix
53 -                        KNearestSteps(index, :) = LTMStep;
54
55 -                    end
56 -                else
57
58                      % Add the current proximity value onto the LTMStep
59 -                    LTMStep = [LTMStep currentProximityValue];
60
61                      % Add this LTMStep to the KnearestNeighbours matrix
62 -                    KNearestSteps = [KNearestSteps; LTMStep];
63 -                end
64 -            end
65 -        end
66 -    end
67 - end
```

NSM Action Selection

In this section, we were informed to use the previously created functions into a loop that instead of using the `rndEpisode` and `rndTrail` functions, to use a newly created `NSMSelectAction` function. This function's purpose is to predict the best possible step from the KnearestSteps matrix provided in the previous exercise in the assignment. For 10% of the time it will choose a random action, done by creating a random integer from 1 to 10 using `randi` function and checking if this integer is 10 later on, if it is then it could be considered as 10%. In order to prevent errors with the first iterations, a check is made to ensure that the LTM has an episode for the `Knearest` function to work with, if it doesn't then it will select a random action for this episode of the LTM. Otherwise, this function can carry on as intended. It predicts the best possible step by calling the function `getActionFromKNearest` which I created. I chose to seperate it into its own function in order to create a test to ensure that I knew from a set amount of values it would produce the correct action, the test being called `getActionFronKNearestTest`. In this function it calculates the mean for each of the actions that are found in the KnearestSteps matrix. If

there is more of one action and they have a higher mean for this step, then it can be assumed that this is the best action to take at the current step. The action is then returned.

```
function action = NSMSelectAction(LTM, STM, currentObservation)

    % Get random number between 1 and 10
    randomNumber = randi([1 10]);

    % Set k for KNearest function
    K = 4;

    % Get the last step of the episode from the first epsiode of the LTM
    finalStep = LTM(20, :, 1);

    % Check if observation is 0
    if finalStep(1,1) == 0
        % On first iteration through and therefore LTM is empty

        % Set random number to 10 so that it selects a random action
        randomNumber = 10;

    end

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

    % Check if number is equal to 10 or no rows in the STM
    if randomNumber == 10 || rows == 0

        % Get random action
        action = randi([1 4], 1, 1);

        % Terminate function
        return;

    end

```

```
35      % Get the K Nearest Steps matrix
36 -    KNearestSteps = KNearest(LTM, K, currentObservation, STM);
37
38      % Check if KNearestSteps is empty
39 -    if isempty(KNearestSteps)
40
41          % Get random action
42 -        action = randi([1 4], 1, 1);
43
44          % Terminate function
45 -        return;
46
47 -    end
48
49      % Get the best action to take from the KNearest sequence
50 -    action = getActionFromKNearest(KNearestSteps);
51
52 -  end
```

```
1   function action = getActionFromKNearest(KNearestSteps)
2
3       % Initialise matrices for each action
4 -     northKNSM = [];
5 -     eastKNSM = [];
6 -     southKNSM = [];
7 -     westKNSM = [];
8
9       % Get number of rows in Knearest
10 -    [KNearestRows, KNearestColumns] = size(KNearestSteps);
11
12      % Iterate all of KNearestSteps
13 -    for i = 1:KNearestRows
14
15          % Check if this iterations steps action
16 -        if (KNearestSteps(i, 2) == 1)
17              % Action is north
18
19              % Add this step's discount to the actions KNSM (k nearest step matrix)
20 -            northKNSM = [northKNSM; KNearestSteps(i, 3)];
21
22 -         elseif (KNearestSteps(i, 2) == 2)
23              % Action is east
24
25              % Add this step's discount to the actions KNSM (k nearest step matrix)
26 -            eastKNSM = [eastKNSM; KNearestSteps(i, 3)];
27
28 -         elseif (KNearestSteps(i, 2) == 3)
29              % Action is south
30
31              % Add this step's discount to the actions KNSM (k nearest step matrix)
32 -            southKNSM = [southKNSM; KNearestSteps(i, 3)];
33 -         elseif (KNearestSteps(i, 2) == 4)
34              % action is west
```

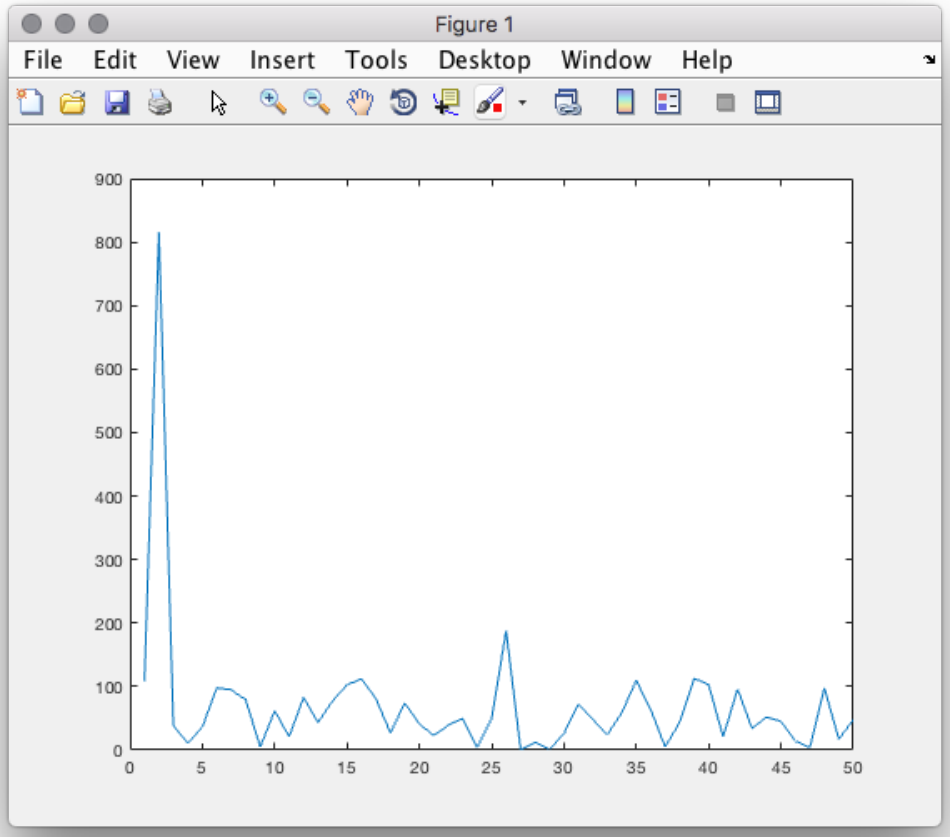
```
35
36              % Add this step's discount to the actions KNSM (k nearest step matrix)
37              westKNSM = [westKNSM; KNearestSteps(i, 3)];
38          end
39
40      end
41
42      % Get the mean of each action's discount rates
43      if (0 ~= size(northKNSM))
44          northDiscountRateMean = mean(northKNSM);
45      else
46          northDiscountRateMean = 0.0;
47      end
48
49      if (0 ~= size(eastKNSM))
50          eastDiscountRateMean = mean(eastKNSM);
51      else
52          eastDiscountRateMean = 0.0;
53      end
54
55      if (0 ~= size(southKNSM))
56          southDiscountRateMean = mean(southKNSM);
57      else
58          southDiscountRateMean = 0.0;
59      end
60
61      if (0 ~= size(westKNSM))
62          westDiscountRateMean = mean(westKNSM);
63      else
64          westDiscountRateMean = 0.0;
65      end
66
67      % Add all means to an array
68      actionMeans = [northDiscountRateMean eastDiscountRateMean southDiscountRateMean westDiscountRateMean];
```

```
69
70      % Get the index of the max value of the action means
71      [maxValue, index] = max(actionMeans);
72
73      % Assign the index which corresponds to the max value of the action mean
74      action = index;
75  end
```

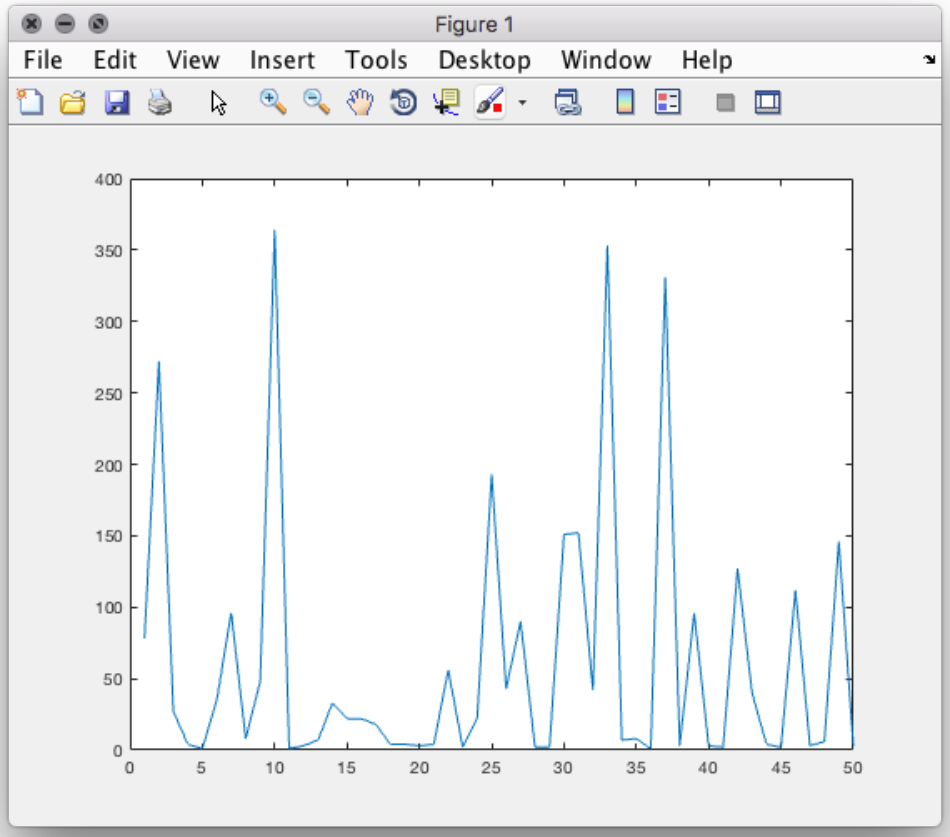
```
52      %% Test: getActionFronKNearest
53      function getActionFronKNearestTest(testcase)
54
55          % Initialise KNearest sequence with arbitary values for observation and
56          % proximity but testable values for discount rate and action
57          %KNearest = [observation action discount rate proximity];
58          KNearest = [10 1 10 1; 10 1 10 1; 10 2 9 1; 10 2 3.4868 1;
59                      10 3 1  1; 10 3 5  1; 10 4 9 1; 10 4 9 1];
60
61          % Initialise expected action (1) because of highest mean of discount
62          % rate
63          expectedAction = 1;
64
65          % Get the actual action
66          actualAction = getActionFromKNearest(KNearest);
67
68          % assert they're the same
69          assert(isequal(actualAction, expectedAction));
70
71      end
```

NSM Trial

This section of the assignment was similar to that in the previous section random trails in the way that it requires us to call a predefined function from the command window with a specified parameter and to analyse the graph. This code instead executes the portion of code that is linked to the newly created functions that when linked together, should force the program to learn over the course of the iterations. Unfortunately due to bugs that were not locatable by the multitude of tests that I wrote for previous functions, the graph only sometimes appears as desired in the brief, and quiet often looks similar to that of the `rndTrail` graph. A good example is shown below:



This does look similar to the brief, it shows progression over the iterations, that is, the number of steps tend towards zero the more it runs, showing that it is learning shorter routes to the goal state. However, as shown below, when I run some more times I get a result as such:

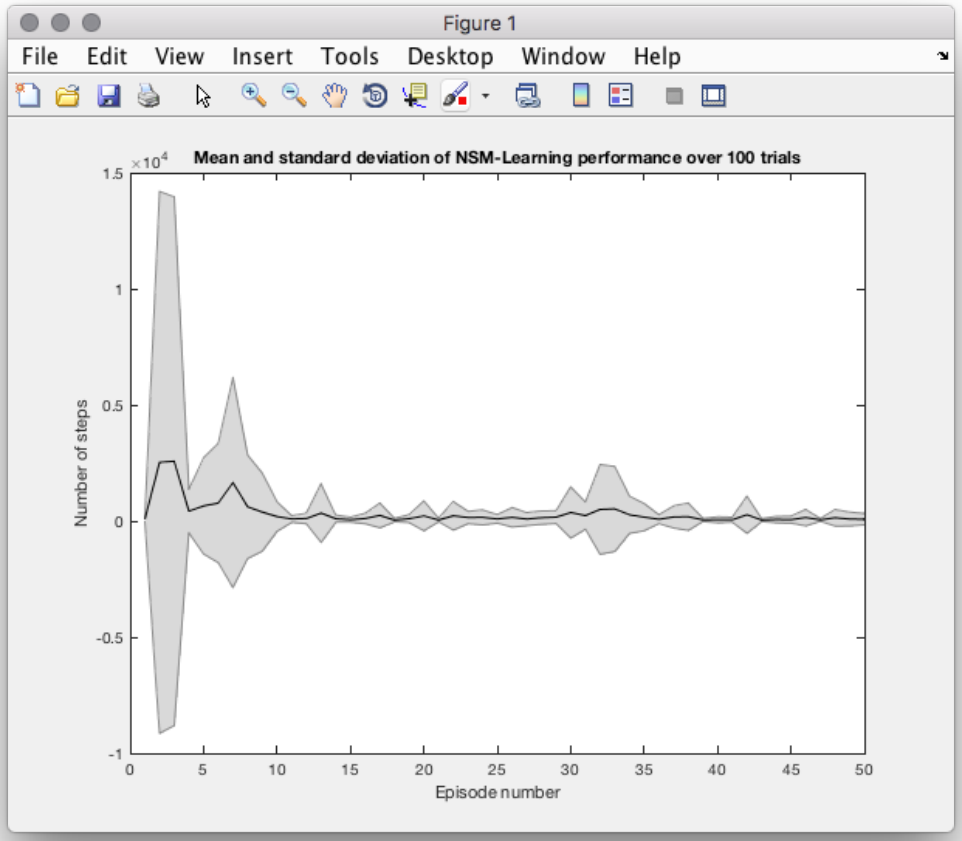


As stated, I have written tests for all of the core functionality to eradicate them as possibilities of causing this bug. However, it is hard to cover all possible edge cases and bugs can also occur when these separate parts integrate with each other. Unfortunately due external reasons I was not able to put more time in to find out the cause of this and so it is left as is. It doesn't seem to effect the `expierment` function massively as you will see, that more times than not, it looks as it should (bar the known issue of running for extended amounts of time).

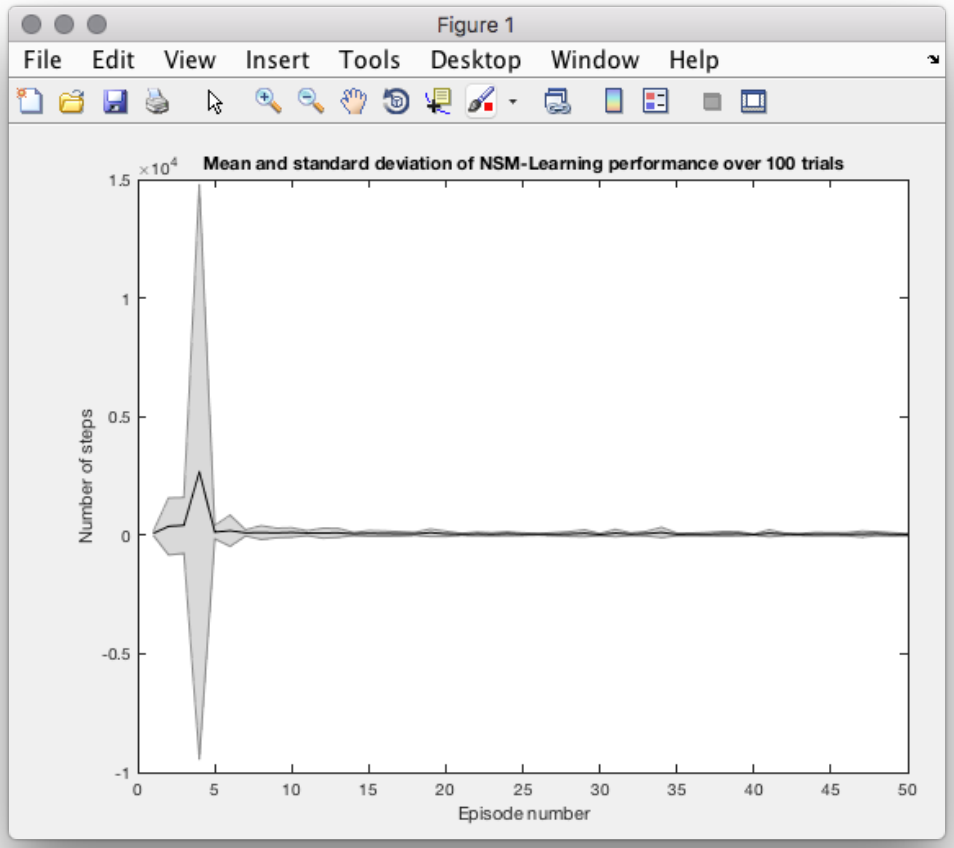
```
function [numOfStepsMatrix, LTM] = NSMTrial(numOfEpisodes)

    % Initialise LTM matrix
    LTM = zeros(20, 3, numOfEpisodes);

    % Initialise the number of steps matrix for each episode
    numOfStepsMatrix = zeros(1, numOfEpisodes);

    % Iterate amount of times to get desired amount of episodes
    for i = 1:numOfEpisodes

        % Get the episode
        [numOfSteps, episode] = NSMEpisode(LTM);

        % Add this iterations episode to the long term memory
        LTM(:, :, i) = episode;

        % Add this episode's step count to the number of steps matrix
        numOfStepsMatrix(i) = numOfSteps;

    end

end
```

NSM Experiment

The purpose of the `experiment` function was to accumulate data about how it runs multiple trials and plot to the mean and standard deviation of each trial's episode against the corresponding episode of other the other trials, this could then be plotted in a graph to analyse. As known, there has been problems with episodes getting stuck into high number of steps before breaking out and finding the goal state and mine was no exception to this problem. We were advised to lower the K value, which I believe would limit the amount it would have to find in order to predict the next best step. In the examples below I have shown trials of 25 and one of 50:



In the above, 25 trials have run and you can see that even with the apparent bug in `NSMTrial`, once run multiple times and the mean taken of it smooths out the line to a reasonable extent to show that it is in-fact learning as it should. That is the mean and standard deviation tend towards 0 when as the trials continue. There is in fact some disturbance closer to the end but this could be caused by it getting stuck for a period of time.



In the above, 50 trials have run and you can see that is very similar to the 25 trials if not closer to the desired outcome. It shows that it is working, learning through the trials and improving with the more repetitions, but also with the bug in `NSMTrial` and the fact I stated it does get stuck shows that a part of the system isn't working fully effectively. I have tried to eliminate the basic functions as possible candidates for the bugs by writing tests for them but even still I have only covered the most basic functionality and not all edge cases as I would have like to if I had more time available to me.

```
function experiment

    % Initialise matrix to hold mean per trial
    meanOfTrials = zeros(1, 50);

    % Initialise matrix to hold standard deviation per trial
    standardDeviationOfTrials = zeros(1, 50);

    % Initialise a matrix to hold all steps across all trials
    trialStepMatrix = zeros(100, 50);

    % Iterate 100 trials
    for i = 1:100

        % Run the for 50 episodes
        [numOfStepsMatrix, LTM] = NSMTrial(50);

        trialStepMatrix(i,:) = numOfStepsMatrix;

    end


    % Iterate each episode associated steps
    for i = 1:50

        % Get the mean of the trials
        meanOfTrials(i) = mean(trialStepMatrix(:, i));

        % Get the standard deviation of the trials
        standardDeviationOfTrials(i) = std(trialStepMatrix(:, i));

    end
```

```
    % Set x axis
    x = [1:1:50];

    figure;

    % Plot the mean and standard deviation of each episode count for 100
    % trials
    shadedErrorBar(x, meanOfTrials, standardDeviationOfTrials, 'black');
    title('Mean and standard deviation of NSM-Learning performance over 100 trials');
    xlabel('Episode number');
    ylabel('Number of steps');

end
```

Ian Howard Lab Journal

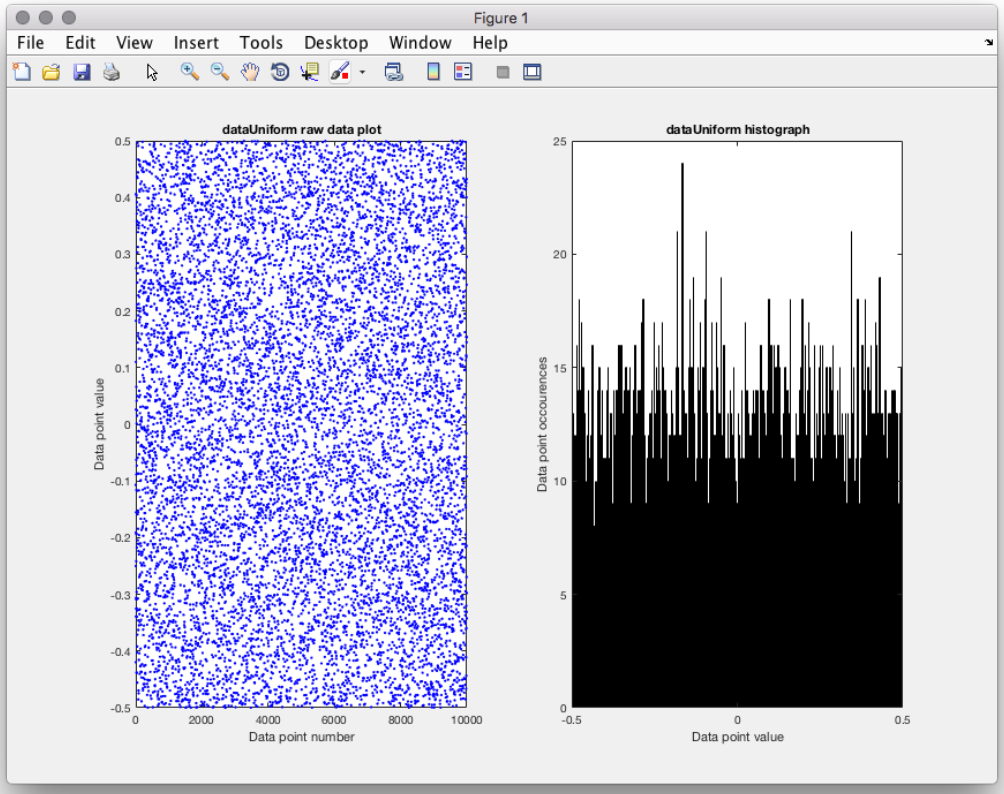
P1.1 1D and 2D distributions

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.

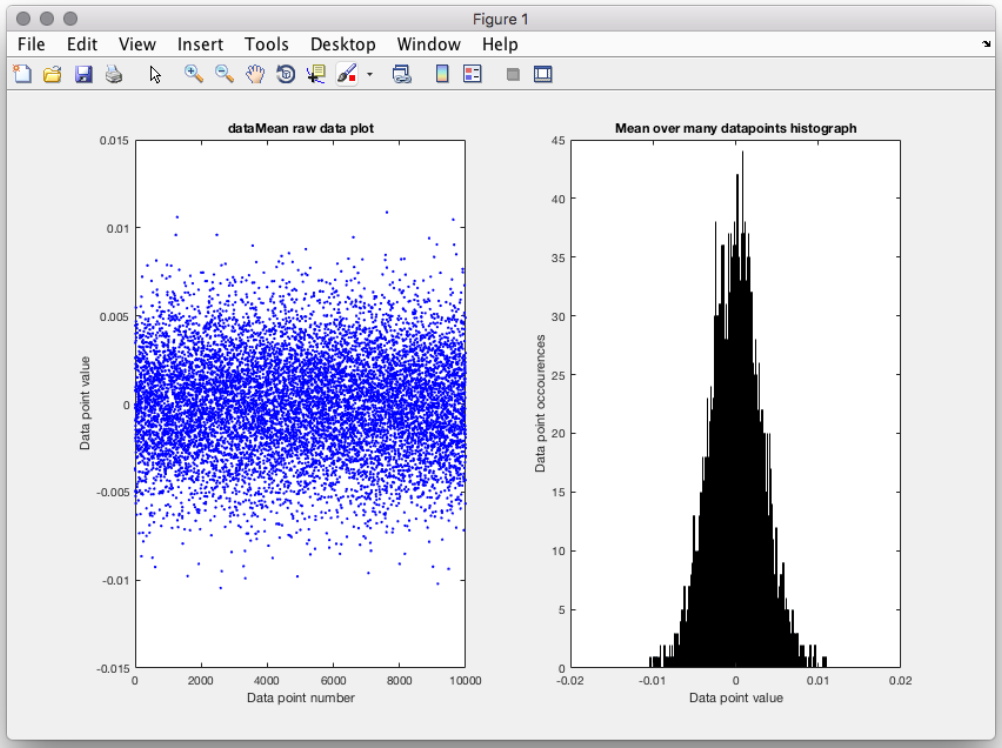
The number of samples that have been used has been consistent throughout all graph generations, standing at 10,000 values. This is large enough for there it to have a good range of values even at lower levels of bins but not high enough so that the generation of the graph takes too long nor that the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

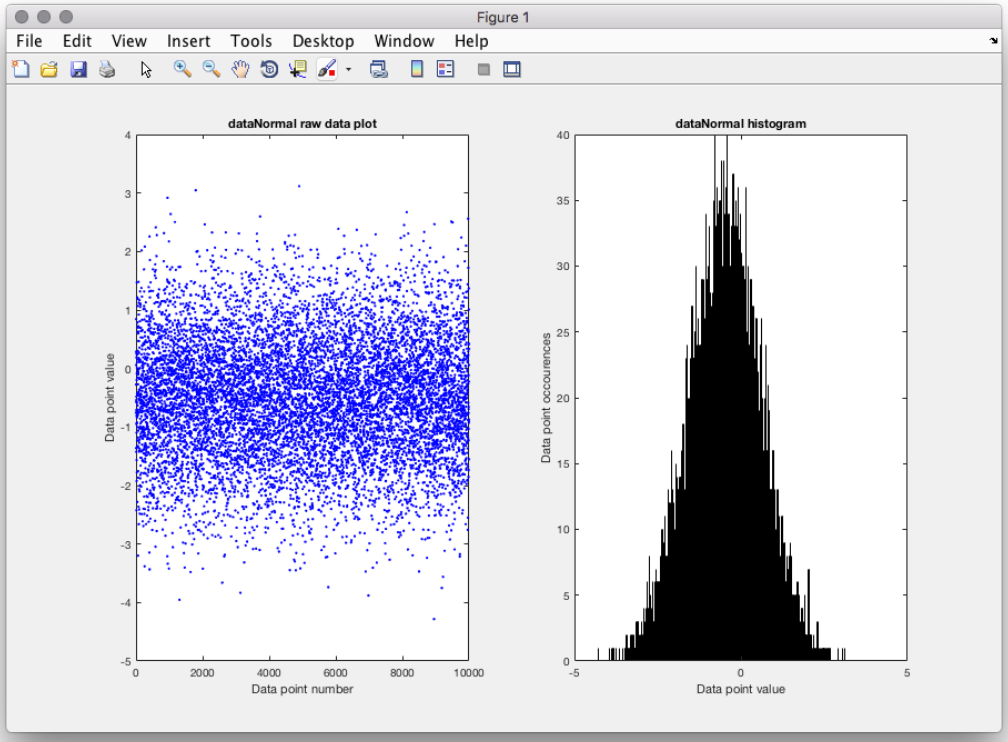
I think that 10,000 adequately show the purpose of the graph to a good degree of accuracy.

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 into 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.

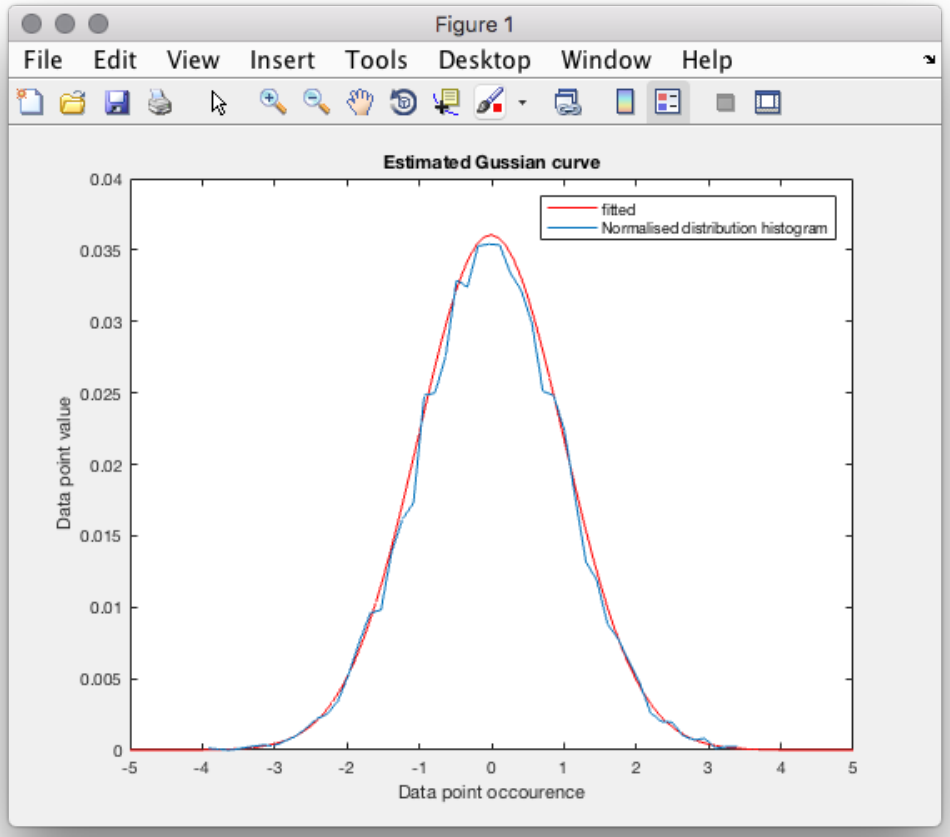
I believe that between 10,000 and 20,000 is more than enough data points to show the distribution's main characteristics.

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 trial 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

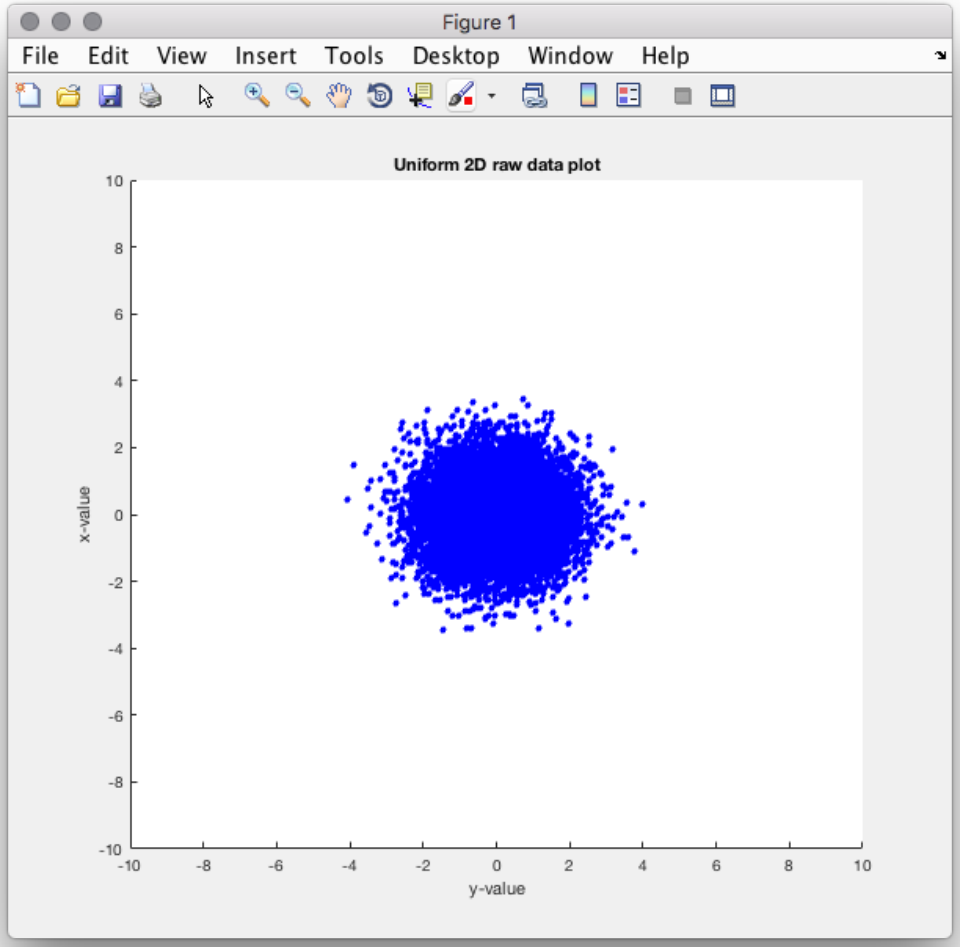
values. Having the sample size set to 10,000 means the graphs curve is of a size that demonstrates the point with clarity.

5. Generate a default 2D distribution

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

defaultTwoDimensionalDistribution.m

```
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.

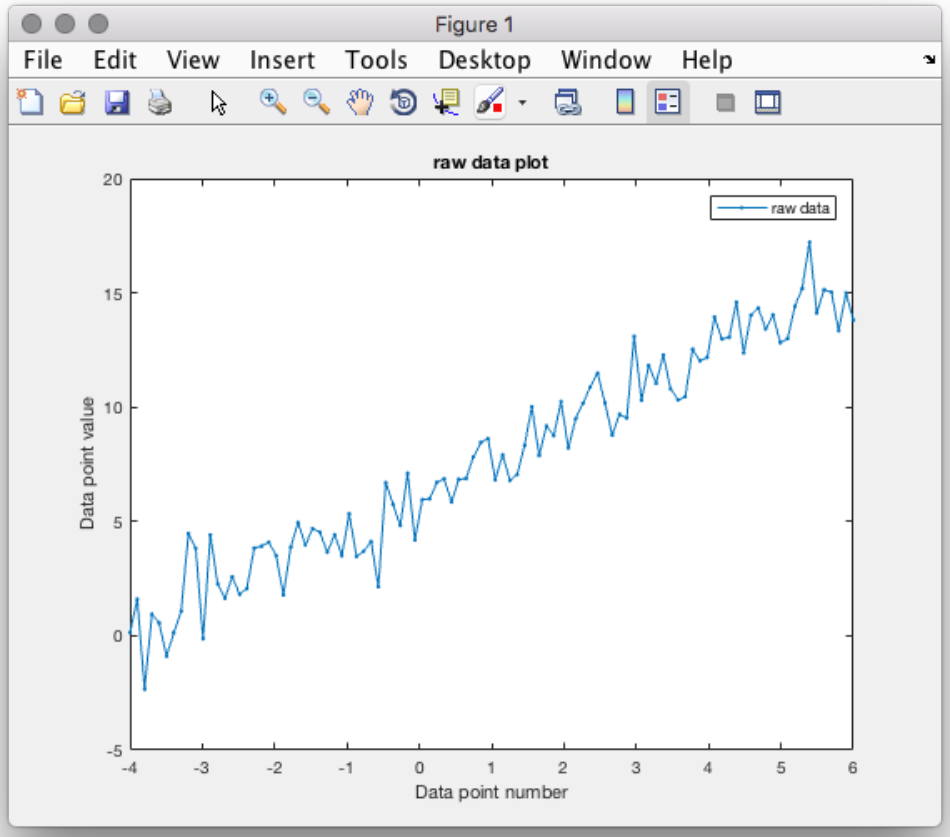
If you were to reduce the number of samples, it reduces the density of the cluster in the middle and makes the results more sparse. Increasing the standard deviation increases the distance of the data's relation to the mean. Changing the value of the mean shifts the middle of the cluster to the value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

P1.2 Linear Regression

1. Generate a noisy line

noisyLine.m

```
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);

```

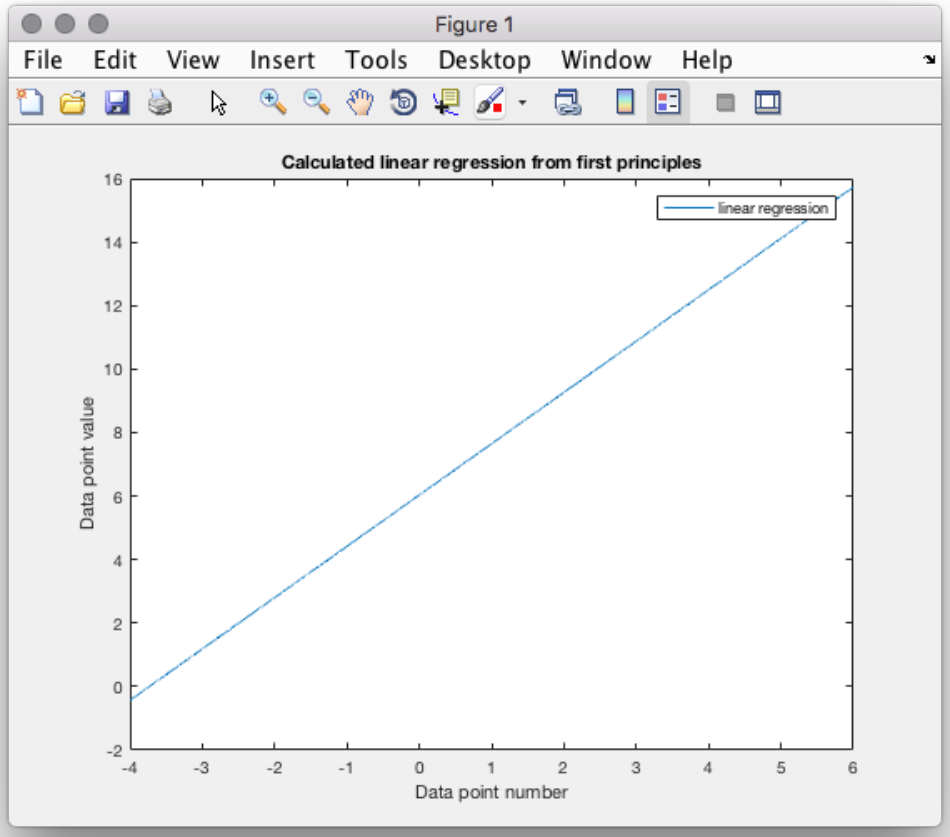
firstPrincipleLinearRegression.m

```
50      % Create the x and y data point arrays on the calculated values
51 -    xPointValues = -4:0.01:6;
52
53 -    yPointValues = yIntersect + gradient.*xPointValues;
54
55      % Plot them against each other
56 -    plot(xPointValues, yPointValues);
57 -    title('Calculated linear regression from first principles');
58 -    xlabel('Data point number');
59 -    ylabel('Data point value');
60 -    legend('linear regression');
61
62 -  end
```

firstPrincipleLinearRegression.m

```
64    function mean = getMean(data)
65
66        % Get the size of the data
67 -      [rows, columns] = size(data);
68
69        % Initiate data sum
70 -      sum = 0.0;
71
72        % Iterate all elements of data
73 -      for i = 1:columns
74
75            % Accumalate total of data
76 -          sum = sum + data(1, i);
77
78 -      end
79
80        % Get the mean
81 -      mean = sum / columns;
82
83 -  end
84
```

```
85  function deviations = getDeviation(mean, data)
86
87      % Get the size of the data
88      [rows, columns] = size(data);
89
90      % Initialise array for the deviations of the data from the mean
91      deviations = zeros(rows, columns);
92
93      % Initialise devation
94      deviation = 0;
95
96      % Iterate all elements of data
97      for i = 1:columns
98
99          % Calculate deviation
100         deviation = data(1, i) - mean;
101
102         % Add this data points deviation from mean to deviations matrix
103         deviations(1, i) = deviation;
104
105     end
106
107 end
108
109 function deviationsSquared = squareDeviations(deviations)
110
111      % Get the size of the data
112     [rows, columns] = size(deviations);
113
114     % Initialise array for the deviations of the data from the mean
115     deviationsSquared = zeros(rows, columns);
116
117     % Iterate all elements of data
118     for i = 1:columns
119
120         % Add this data points deviation from mean to deviations matrix
121         deviationsSquared(1, i) = deviations(1, i) ^ 2;
122
123     end
124
125 end
126
```



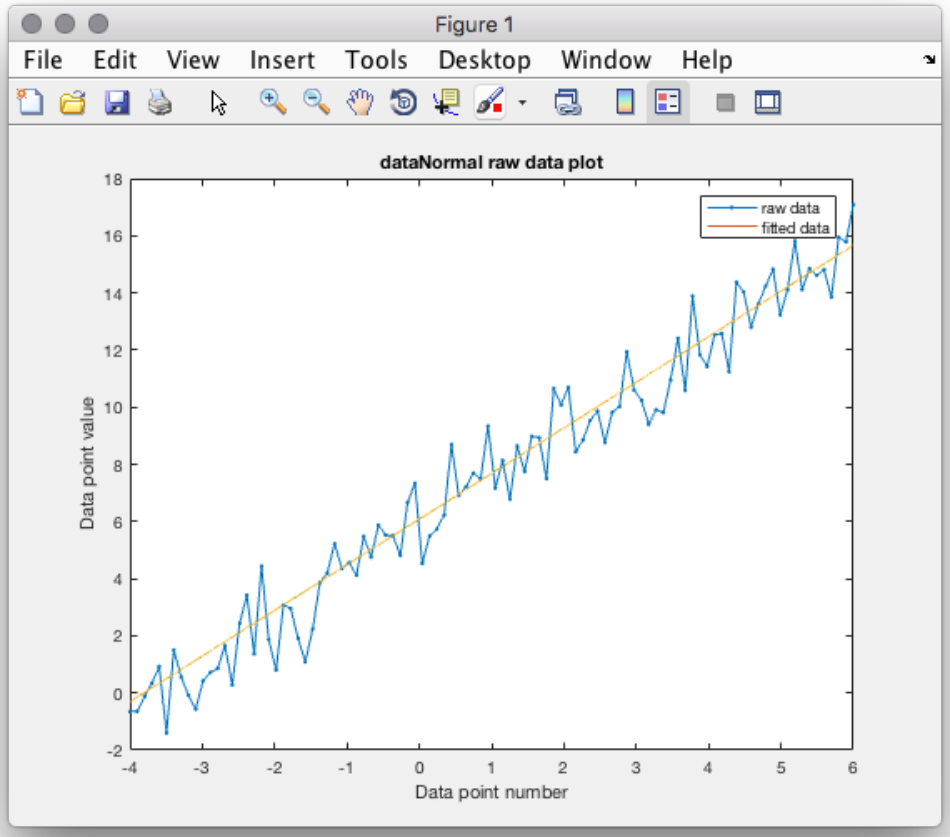
The task was to create linear regression line from first principles, this being that we had to program the fundamental methodologies of linear regressions myself, and not to use the in-built Matlab functions. I achieved this by using the least fitting square equation but programmed out step by step. 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

firstPrincipleLinearRegression.m | fittedNoisyLine.m

```
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


```

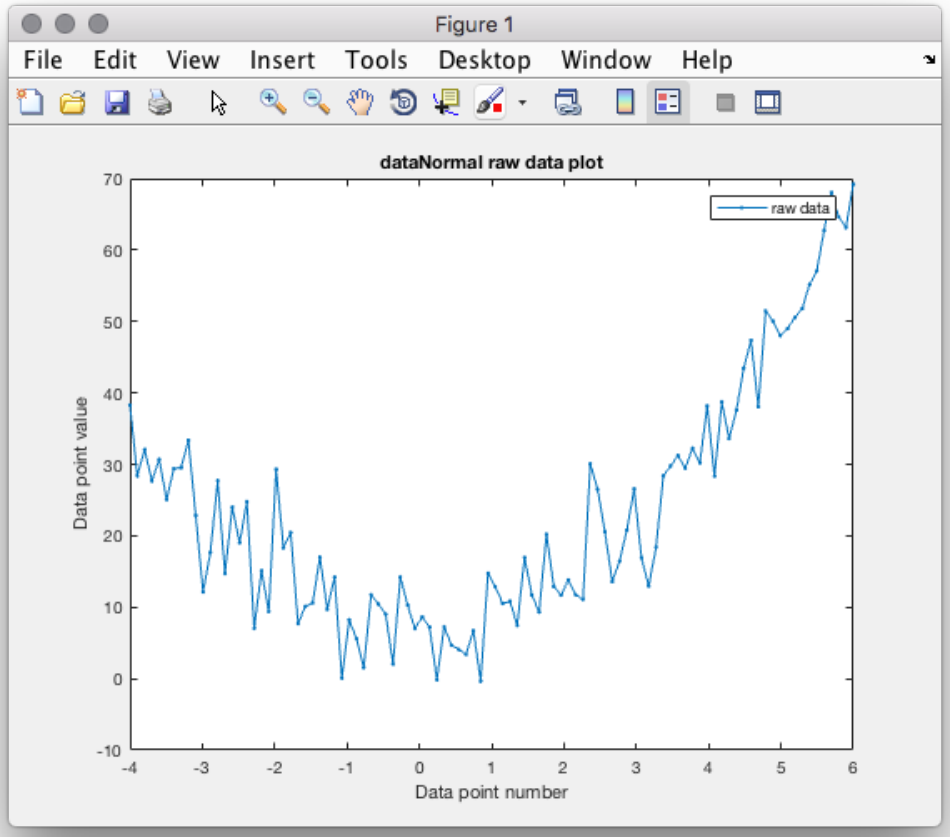


This exercise simply is an amalgamation of the previous two exercises. That is, to plot the 'fitted' line from exercise two (created from first principles using the LFS method) and the noisy linear line from exercise one. This produces a fitted noisy line, emphasising two things. Firstly, that the least 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

noisyQuadraticCurve.m

```
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 A^2 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

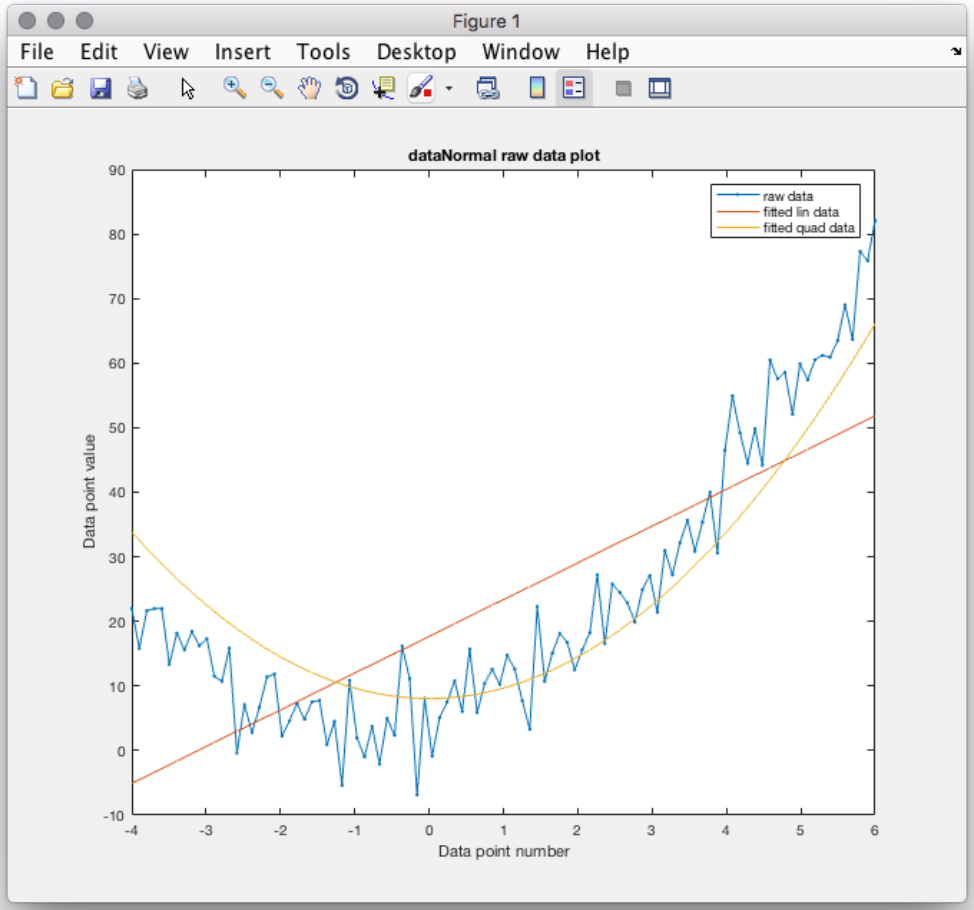
numbers. If you were to increase the standard deviation that is used in generating the noise, it would make the points more erratic, meaning they are further away from the 'line'.

5. Fit the quadratic curve using linear regression

fittedNoisyQuadraticCurve.m

```
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);
```

```
50
51 -       plot(x, yFittedLin);
52
53 -       hold on;
54
55 -       plot(x, yFittedQuad);
56
57 -       title('dataNormal raw data plot');
58 -       xlabel('Data point number');
59 -       ylabel('Data point value');
60 -       legend('raw data', 'fitted lin data', 'fitted quad data');
61
62 -   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

Generate dataset

The task was to generate a dataset of two clusters that are uncorrelated. The complete size of the whole data set was 20,000, which meant each cluster is 10,000 respectively. We were given the mean and standard deviation of both clusters. The first mean was [-4 -1] with a standard deviation of 0.75 and the second cluster mean was [3 4] and with a standard deviation of 2.0. We were then told to plot these on these on the same graph but different colours for each cluster so you could identify which was which.

You can see the effects of the mean and standard deviation of each cluster. As it is a 2D data set, the mean is effectively the coordinates for the centre of each cluster, the standard deviation effects the area around the mean that the data points are spread out into. The larger standard devation, cluster two, is greater in size compared to the first which is more concentrated around its mean. We used randn to generate random numbers that are normally distributed in conjunction with the mean and standard deviation supplied.

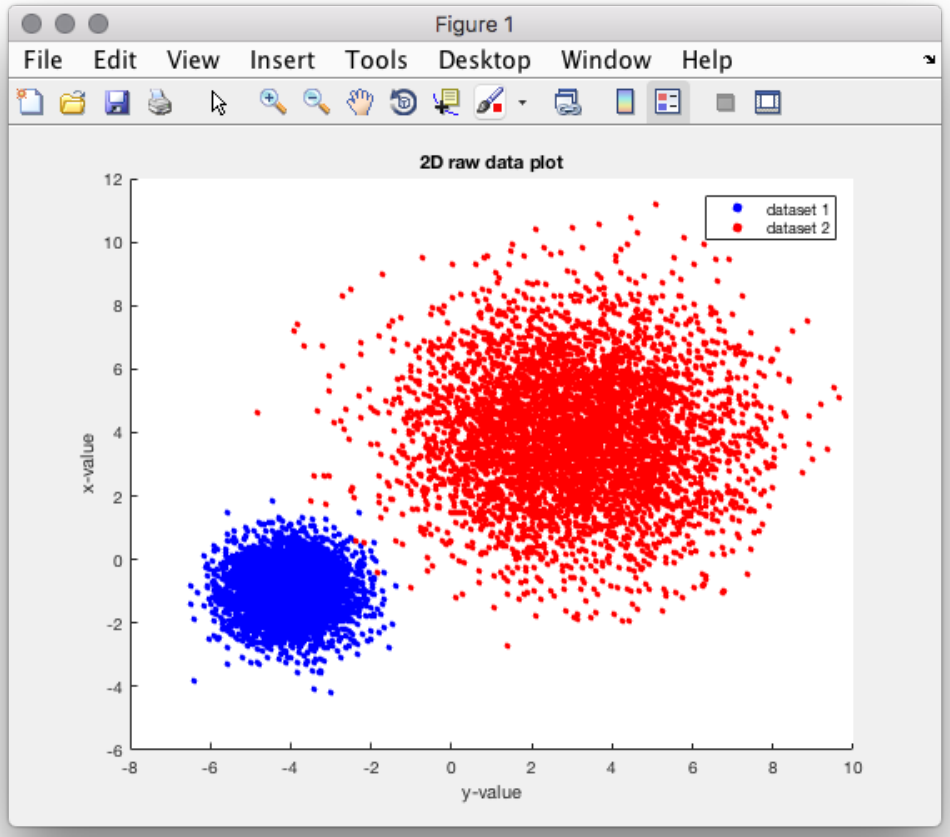
Editor - /Users/user/Documents/MATLAB/P1.3: KMEANS CLUSTERING/plotGeneratedDataset.m

concatenateDatasets.m | kmeansFirstPrinciple.m | plotGeneratedDataset.m

```
function plotGeneratedDataset

    % Get the generated data sets
    [dataSet1, dataSet2] = generateDataset();

    figure;

    % Create the size of the cirlces to be plotted
    circleSize = 15;

    % Plot the data dimension against eachother to get a 2D scatter plot
    % for data set 1
    scatter(dataSet1(1, :), dataSet1(2, :), circleSize, 'b', 'filled');

    hold on;

    % Plot the data dimension against eachother to get a 2D scatter plot
    % for data set 2
    scatter(dataSet2(1, :), dataSet2(2, :), circleSize, 'r', 'filled');

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

end
```

generateDataset.m

```
function [dataSet1Samples, dataSet2Samples] = generateDataset

    % Initialise sample size
    sampleSize = 5000;

    % Initialise first data set mean and std
    dataSet1Mean = [-4; -1];
    dataSet1Std  = 0.75;

    % Initialise second data set mean and std
    dataSet2Mean = [3; 4];
    dataSet2Std  = 2;

    % Get the samples for both X and Y data set 1
    dataSet1Samples = dataSet1Std.*randn(2, sampleSize) + dataSet1Mean;

    % Get the samples for both X and Y data set 2
    dataSet2Samples = dataSet2Std.*randn(2, sampleSize) + dataSet2Mean;

```

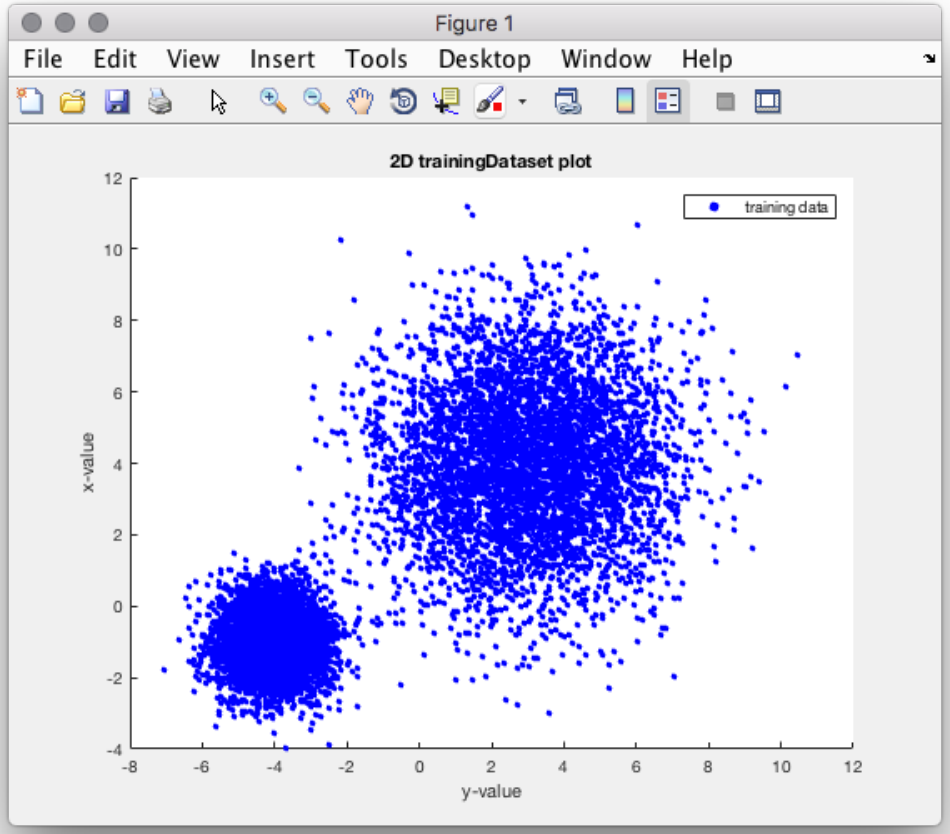


Concatenate dataset

The task was to concatenate the data sets into a single dataset, which is used later on so that the separate data sets aren't know, so that the kmeans can split them up again into their clusters. This was simply done by using matlab matrix concenate feature [a b]. It was then plotted without colour as they're now not distinct datasets, to prove that they have been concatenated successfully.

concatenateDatasets.m

```
function concatenateDatasets
/Users/user/Documents/MATLAB/P1.3: KMEANS CLUSTERING/concatenateDatasets.m
    % Get the generated data sets
    [dataNormal1, dataNormal2] = generateDataset();

    % Concatenate the datasets
    trainingDataset = [dataNormal1 dataNormal2];

    % Create the size of the cirlces to be plotted
    circleSize = 15;

    % Plot the results
    scatter(trainingDataset(1, :), trainingDataset(2, :), circleSize, 'b', 'filled');

    % Format the graph
    title('2D trainingDataset plot');
    xlabel('y-value');
    ylabel('x-value');
    legend('training data');
end
```



Implement kmeans from first principles

The task was to implement the K-means classification algorithm from first principles, that is, without the use of high level Matlab functions. This algorithm is supposed to estimate its centroids points through many repetitions until it has reached the centre and the cluster is estimated.

To do this, I initially chose K number of random locations to plot the points on the graph. Then every iteration you re-estimate the points positions in conjunction with how close that point is to the greater amount of data points. After some repetitions, the each point has moved to the centre of the data sets and once it isn't moving anymore, convergence has reached and the algorithm is complete.

concatenateDatasets.m | kmeansFirstPrinciple.m

```
1  function [trainingDataset, clusterMeans] = kmeansFirstPrinciple(K, trainingDataset)
2
3      % Get the min and max values of training data set
4      minValueOfEachCol = min(trainingDataset);
5      minValueTotalSet = min(minValueOfEachCol);
6      minValueInt = round(minValueTotalSet);
7
8      maxValueOfEachCol = max(trainingDataset);
9      maxValueTotalSet = max(maxValueOfEachCol);
10     maxValueInt = round(maxValueTotalSet);
11
12     % Initialise matrix for cluster centroids
13     clusterCentroids = zeros(2, K);
14
15     % Initialise matrix for cluster means
16     clusterMeans = zeros(2, K);
17
18     % Iterate for each cluster
19     for cluster = 1:K
20
21         % Randomly assign cluster centroids
22         clusterCentroids(:, cluster) = randi([minValueInt maxValueInt], 1, 2);
23
24     end
25
26     % Get the size of the data set
27     [rows, columns] = size(trainingDataset);
28
29     % Add extra row of zeros to bottom of training data set
30     trainingDataset = [trainingDataset; zeros(1, columns)];
```

Editor – /Users/user/Documents/MATLAB/P1.3: KMEANS CLUSTERING/kmeansFirstPrinciple.m

concatenateDatasets.m | kmeansFirstPrinciple.m

```
32     % Initialise kmeans loop break
33     finished = false;
34
35     % iterate until convergence
36     while finished == false
37
38         % Iterate for each column of the training dataset
39         for column = 1:columns
40
41             % Initialise previous cluster centroid to compare with
42             prevDiffFromClusterCentroid = 'uninitialised';
43
44             % Iterate each cluster
45             for cluster = 1:K
46
47                 % Find the difference of the current training dataset value
48                 % from cluster centroid
49                 diffFromClusterCentroid = norm(clusterCentroids(:, cluster) - trainingDataset(1:2, column));
50
51                 % Check if there is a previous value set
52                 if (strcmp(prevDiffFromClusterCentroid, 'uninitialised'))
53
54                     % Set previous value
55                     prevDiffFromClusterCentroid = diffFromClusterCentroid;
56
57                 end
58
59                 % Check if the difference from cluster centroid 1 is less
60                 % than the difference from cluster centroid 2
61                 if (diffFromClusterCentroid < prevDiffFromClusterCentroid)
62                     % It is less
63
```

Editor - /Users/user/Documents/MATLAB/P1.3: KMEANS CLUSTERING/kmeansFirstPrinciple.m

concatenateDatasets.m | kmeansFirstPrinciple.m

```
63
64                      % Datapoint belongs to this cluster
65 -                    trainingDataset(3, column) = cluster;
66
67 -                else
68
69                      % Datapoint belongs to preivous cluster
70 -                    trainingDataset(3, column) = (cluster - 1);
71
72 -                end
73
74
75 -            end
76
77 -        end
78
79
80
81          % Iterate each cluster
82 -        for cluster = 1:K
83
84              % Get all values that belong to the cluster
85 -            currentCluster = trainingDataset(3, :) == cluster;
86 -            currentCluster = trainingDataset(:, currentCluster);
87
88              % Find the mean of each cluster
89 -            currentClusterMean = mean(currentCluster, 2);
90
91              % Appened to all cluster means
92 -            clusterMeans(:, cluster) = currentClusterMean(1:2);
93
94 -        end
95
```

```
95
96          % Check to see if convergence is reached
97 -        if (clusterMeans == clusterCentroids)
98
99              % Set exit condition
100 -            finished = true;
101
102 -        end
103
104         % Assign new cluster centroids
105 -        clusterCentroids = clusterMeans;
106
107 -    end
108
109 - end
```

Run and plot k-means

The task was to run the kmeans algorithm that I implemented from first principles with the training data and to plot the results. As you can see, it has successfully classified each cluster and coloured them accordingly.

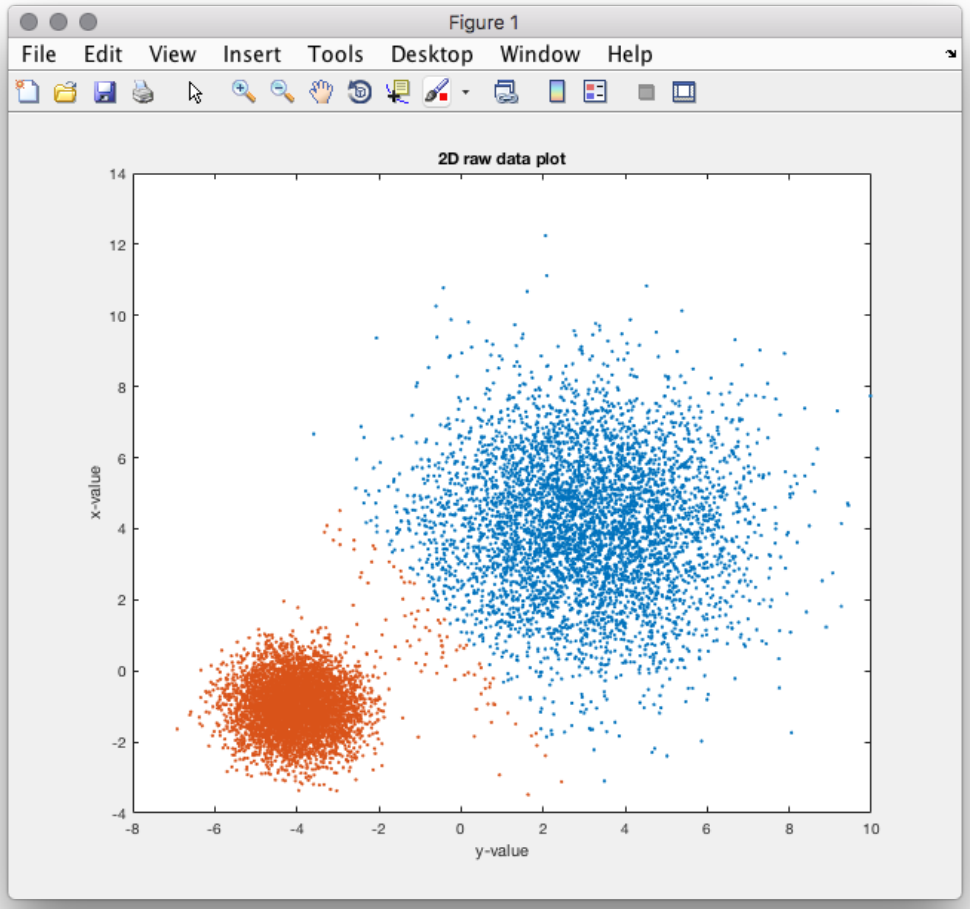
We shouldn't however, be running the algorithm on the training data, this would mean that the algorithm hasn't been trained correctly, and it doesn't provide unbiased results because it isn't using independent testing data to run on. If we were to train with one set of data and test with another, we would have true results of the algorithm.

One can also see that there is an error margin between the two clusters. This is where the data points are at the edge of their distributions and boarding on the other cluster, therefore it is harder to predict in a small amount of repetitions which these data points belong to. If we were to train further and improve the algorithm, as well as use testing data, one would assume that this error would decrease and it would provide more accurate results.

Editor - /Users/user/Documents/MATLAB/P1.3: KMEANS CLUSTERING/plotKmeansClusteredData.m

concatenateDatasets.m | kmeansFirstPrinciple.m | plotGeneratedDataset.m | plotKmeansClusteredData.m

```
function plotKmeansClusteredData

    % Get the generated data sets
    [dataNormal1, dataNormal2] = generateDataset();

    % Concatenate the datasets
    trainingDataset = [dataNormal1 dataNormal2];

    % Initalise k
    K = 2;

    % Call k means to get the clusters
    [trainingDataset, clusterMeans] = kmeansFirstPrinciple(K, trainingDataset);

    figure;

    % Iterate each cluster
    for cluster = 1:K

        % Get all values that belong to the cluster
        currentCluster = trainingDataset(3, :) == cluster;
        currentCluster = trainingDataset(:, currentCluster);

        % Plot the data dimension against eachother to get a 2D scatter plot
        % for current cluster
        plot(currentCluster(1, :), currentCluster(2, :), '.');

        hold on;

    end

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

end

```



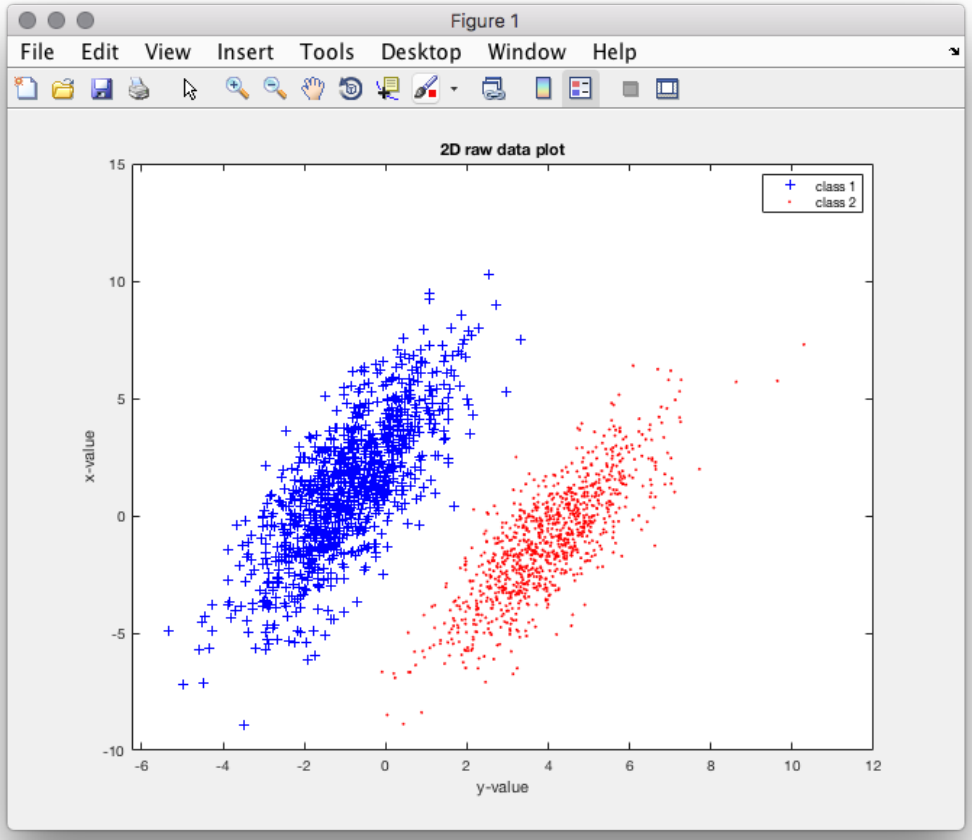
P1.4 Naive Bayes and Perceptron

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.

generateDataset.m

```
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 standard 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 priori that the data consists of two Gaussians concatenated together and so I just re-split these into separate data classes so that I can get the mean and standard deviation 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 standard deviation of each class and each attribute from the previous steps, I can calculate the probability of the attribute given the class by applying the Gaussian 1D formula to it. Given I have the probability of each attribute given each class, I now calculate the probability of each class given the attribute using that data, as well as the apriori probability of 0.5, which I assumed as the data is concatenated of two Gaussian datasets of equal length. 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. 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(100);
    [testingData, testingTarget] = GenerateGaussianData(100000);

    % 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 standard deviation of class 0's attributes
    stdOfClass0AtrA = std(class0(1,:));
    stdOfClass0AtrB = std(class0(2,:));

```

```
33      % Get the mean of class 1's attributes
34 -    meanOfClass1AtrA = mean(class1(1,:));
35 -    meanOfClass1AtrB = mean(class1(2,:));
36
37      % Get the standard deviation of class 1's attributes
38 -    stdOfClass1AtrA = std(class1(1,:));
39 -    stdOfClass1AtrB = std(class1(2,:));
40
41      % Create empty classes for the test data to fall into
42 -    classTest0 = [];
43 -    classTest1 = [];
44
45      % Iterate all columns of the testing data
46 -    for i = 1:length(testingData)
47
48          % Get the data element for testing
49 -        data = testingData(:, i);
50
51          % Get atr A and atr B
52 -        atrA = data(1, :);
53 -        atrB = data(2, :);
54
55          % Get the prob of atr A given class0 and atr B given class0
56 -        probAtrAClass0 = getProbFromGuassian(stdOfClass0AtrA, meanOfClass0AtrA, atrA);
57 -        probAtrBClass0 = getProbFromGuassian(stdOfClass0AtrB, meanOfClass0AtrB, atrB);
58
59          % Get the prob of atr A given class1 and atr B given class1
60 -        probAtrAClass1 = getProbFromGuassian(stdOfClass1AtrA, meanOfClass1AtrA, atrA);
61 -        probAtrBClass1 = getProbFromGuassian(stdOfClass1AtrB, meanOfClass1AtrB, atrB);
62
63          % Get the prob of class0 given attribute A and for B
64 -        probClass0AtrA = probAtrAClass0 * 0.5 ./ (probAtrAClass0 * 0.5 + probAtrAClass1 * 0.5);
65 -        probClass0AtrB = probAtrBClass0 * 0.5 ./ (probAtrBClass0 * 0.5 + probAtrBClass1 * 0.5);
66
```

```
66
67          % Get the prob of class1 given attribute A and for B
68 -        probClass1AtrA = probAtrAClass1 * 0.5 ./ (probAtrAClass0 * 0.5 + probAtrAClass1 * 0.5);
69 -        probClass1AtrB = probAtrBClass1 * 0.5 ./ (probAtrBClass0 * 0.5 + probAtrBClass1 * 0.5);
70
71          % Get prob of x given class 0 and x given class1 using the
72          % conditional independance assumption
73 -        P0 = probClass0AtrB * probClass0AtrA;
74 -        P1 = probClass1AtrB * probClass1AtrA;
75
76          % If class 0 given x is higher than class1 given x then it is in
77          % class 0, else it is in class 1
78 -        if (P0 > P1)
79 -             classTest0 = [classTest0 testingData(:, i)];
80 -        else
81 -             classTest1 = [classTest1 testingData(:, i)];
82 -        end
83
84 -    end
85
86      % Plot the graph
87 -    figure
88 -    hold on
89 -    plot(classTest0(1, :), classTest0(2,:), 'ro');
90 -    plot(classTest1(1, :), classTest1(2,:), 'b+');
91 -    xlabel('x-dimension');
92 -    ylabel('y-dimension');
93 -    title('Ploting class0 in red and class1 in blue');
94
95 - end
96
```

```
97    % Apply Gaussian 1D function to the attribute to get prob of being in class
98    function probOfAtr = getProbFromGuassian(stdOfAtr, meanOfAtr, atr)
99
100         % Calculate first half of equation
101 -       eq1 = (1 / (sqrt(2 * pi) * stdOfAtr)) * exp( -(atr - meanOfAtr) .^ 2);
102
103         % Apply first half to last section to get probability
104 -       probOfAtr = eq1 / (2 * stdOfAtr ^ 2);
105
106 -   end
```

Run a Naive bayes classifier and plot the result

One can see that it has successfully classified the data into the separate clusters with minimal error rate. Only when performing a fairly unfair amount of `trainingData` sample size against a large sample size of `testingData`, as seen in graph 2 and 3. In these examples I set the trainingData sample size to 10 and the testingData sample size to 100000 which is a big difference. Considered that the trainingData is so small compared to the testing data you can still see the power of Naive bayes classifier. It is relatively effective with only the top 15%-20% being wrongly classified. If the both sample sizes were scaled up to reflect a real world example then given the small trainingData size, this is still a fairly acceptable. However, when you get to a more equal sample sizes for both training and testing, then you start to see the real power of Naive bayes. It is very low computational overhead, only

requiring the mean and standard deviation of the training classes to calculate the probability of it being in either class on the testing which only requires one run through of the dataset, meaning it is suitable to large datasets. You can see the increase increase in accuracy from the first and last graph shown. In the first the sample rates are set at 100 and 100000 for training and testing respectively, it still has a small boundary error rate but it isn't a lot compared to graphs 2 and 3. It improves even further in the last graph where it is set at 1000 and 100000, where the boundary error is almost nonexistent.

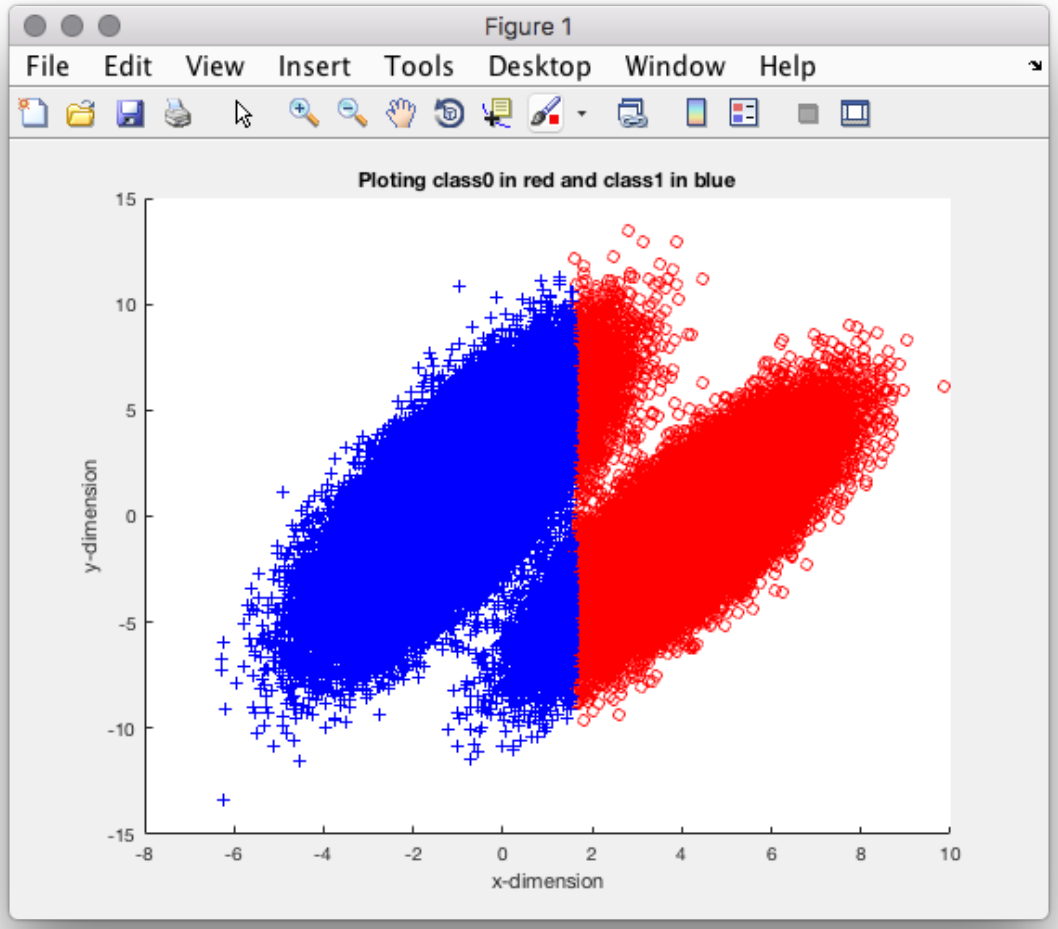


Figure 1
File Edit View Insert Tools Desktop Window Help
Ploting class0 in red and class1 in blue
y-dimension
x-dimension
15
10
5
0
-5
-10
-15
-8
-6
-4
-2
0
2
4
6
8
10

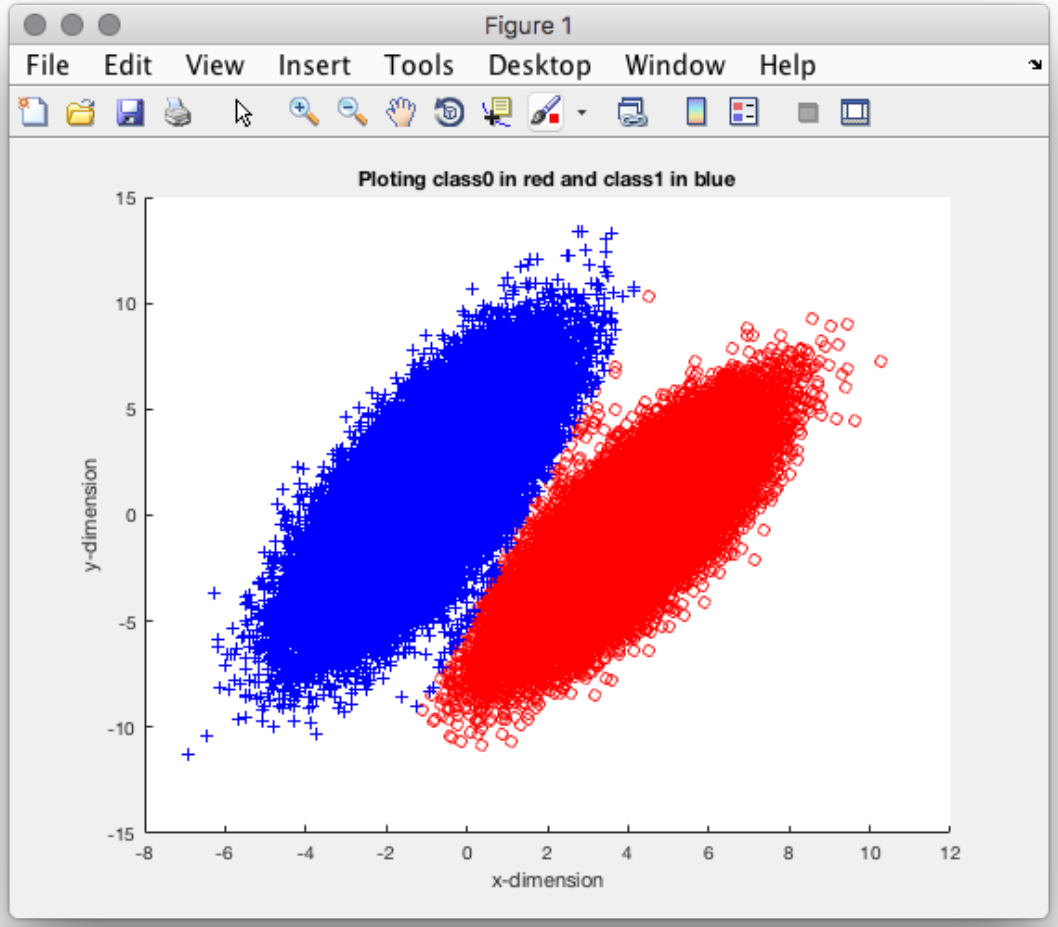


Figure 1
File Edit View Insert Tools Desktop Window Help
Ploting class0 in red and class1 in blue
y-dimension
x-dimension
15
10
5
0
-5
-10
-15
-8
-6
-4
-2
0
2
4
6
8
10
12

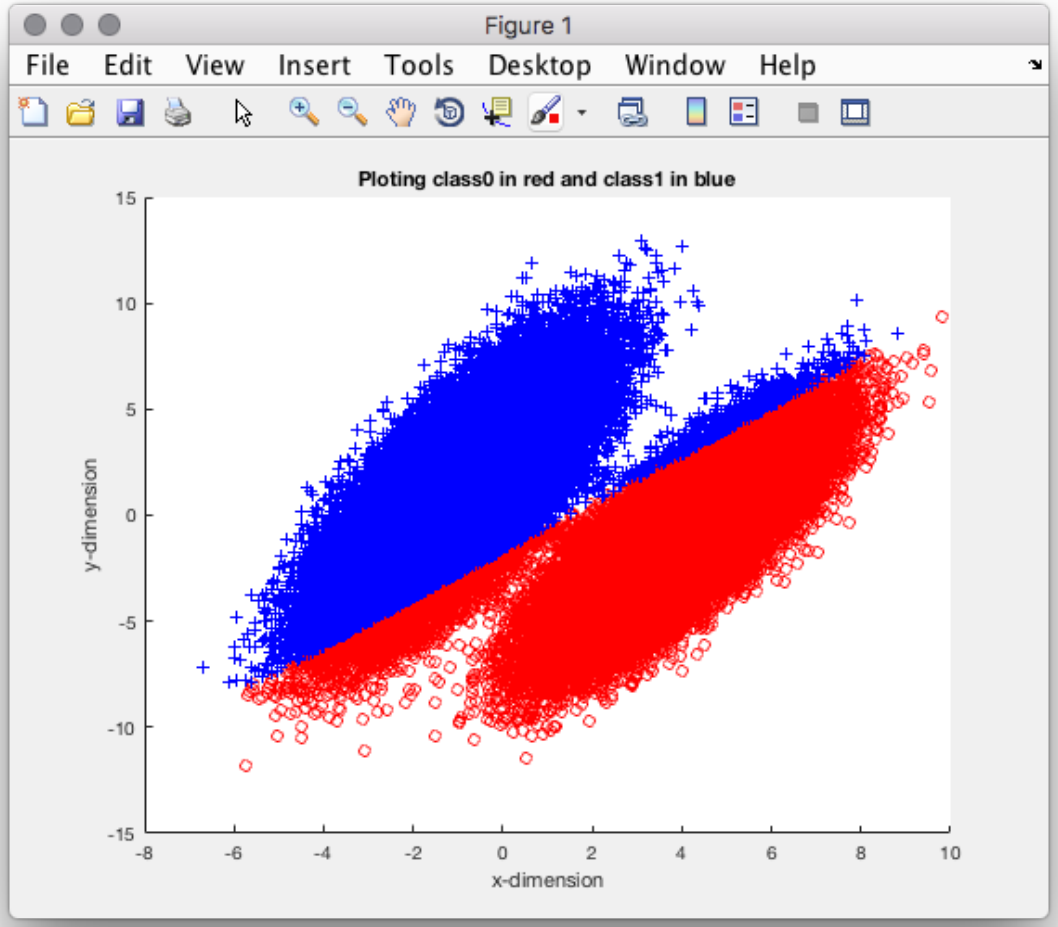
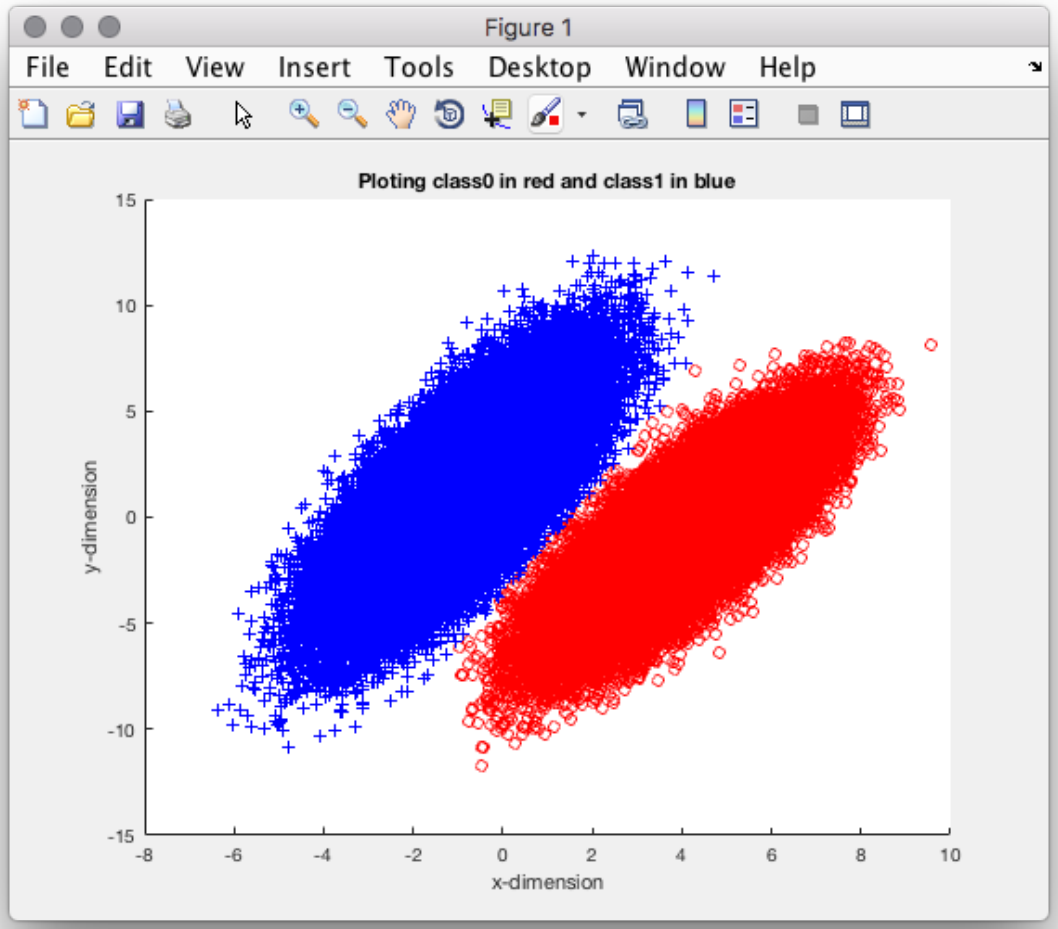


Figure 1
File Edit View Insert Tools Desktop Window Help
Ploting class0 in red and class1 in blue
y-dimension
x-dimension
15
10
5
0
-5
-10
-15
-8
-6
-4
-2
0
2
4
6
8
10



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 multiplied by the weights to get the class of this data element. With the data element class you can plug the rest of the values into the weight vector update formula as provided in the lecture slides to update the weight vector. This effectively training the perceptron, as the weights act as learning points so it can predict the output from the input with greater accuracy. This is continually done for all the `trainingData` to get the final weight vector which is applied in the same way, without the weight vector update formula as it is already trained, to the testing data to provide the classification.

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

```

```
41
42          % Initalise empty classes for the testing data
43 -        class0 = [];
44 -        class1 = [];
45
46          % Iterate all testing data
47 -        for j = 1:length(testingData)
48
49              % Get the testing data element
50 -            data = [testingData(:, j)', 1];
51
52              % Get the classifier threshold
53 -            threshold = sum(data .* weights);
54
55              % Check if threshold is above 0
56 -            if threshold > 0
57
58                  % Is above 0, assign class 1
59 -                class0 = [class0 testingData(:, j)];
60
61 -            else
62
63                  % Below 0, assign class 0
64 -                class1 = [class1 testingData(:, j)];
65
66 -            end
67
68
69 -        end
70              % Plot the graph
71 -        figure
72 -        hold on
73 -        plot(class0(1,:), class0(2,:), 'ro');
74 -        plot(class1(1,:), class1(2,:), 'b+');
75 -        xlabel('x-dimension');
76 -        ylabel('y-dimension');
77 -        title('Ploting class0 in red and class1 in blue');
78
79 -    end
80
```

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 in fact just varies on that particular run. Sometimes the boundary is extremely accurate 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.

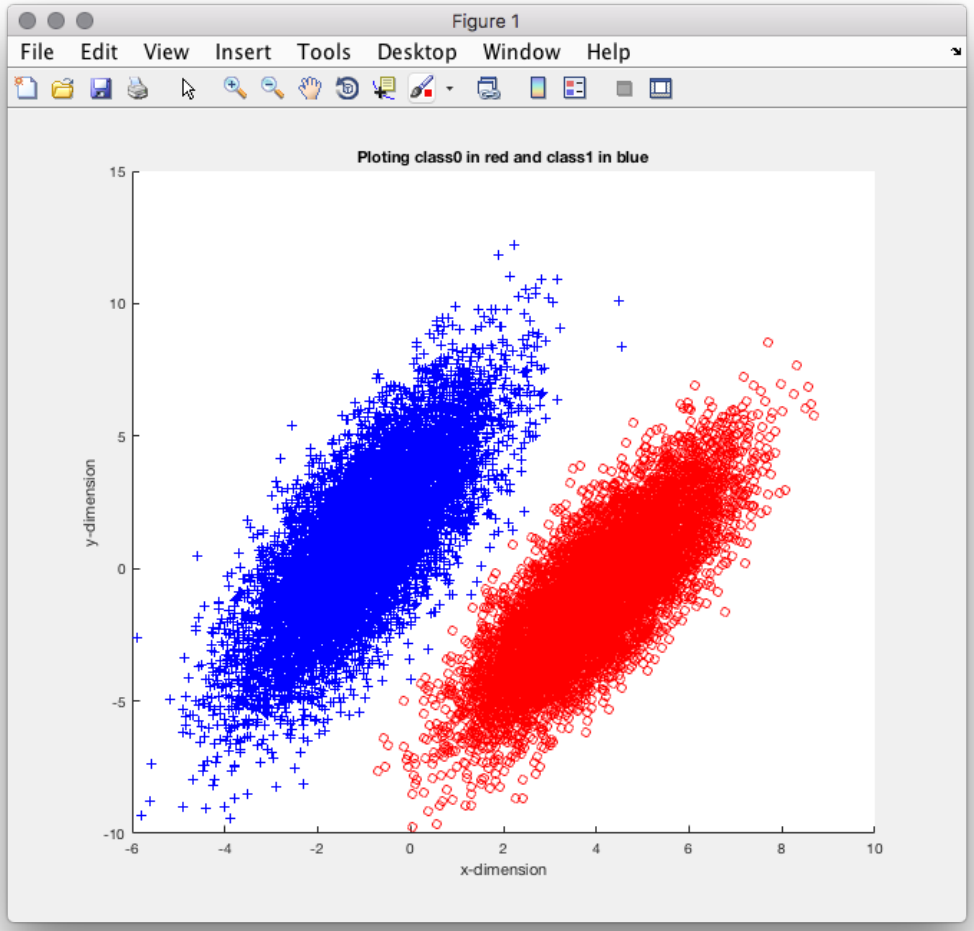


Figure 1
File Edit View Insert Tools Desktop Window Help
Ploting class0 in red and class1 in blue
y-dimension
x-dimension
15
10
5
0
-5
-10
-6
-4
-2
0
2
4
6
8
10

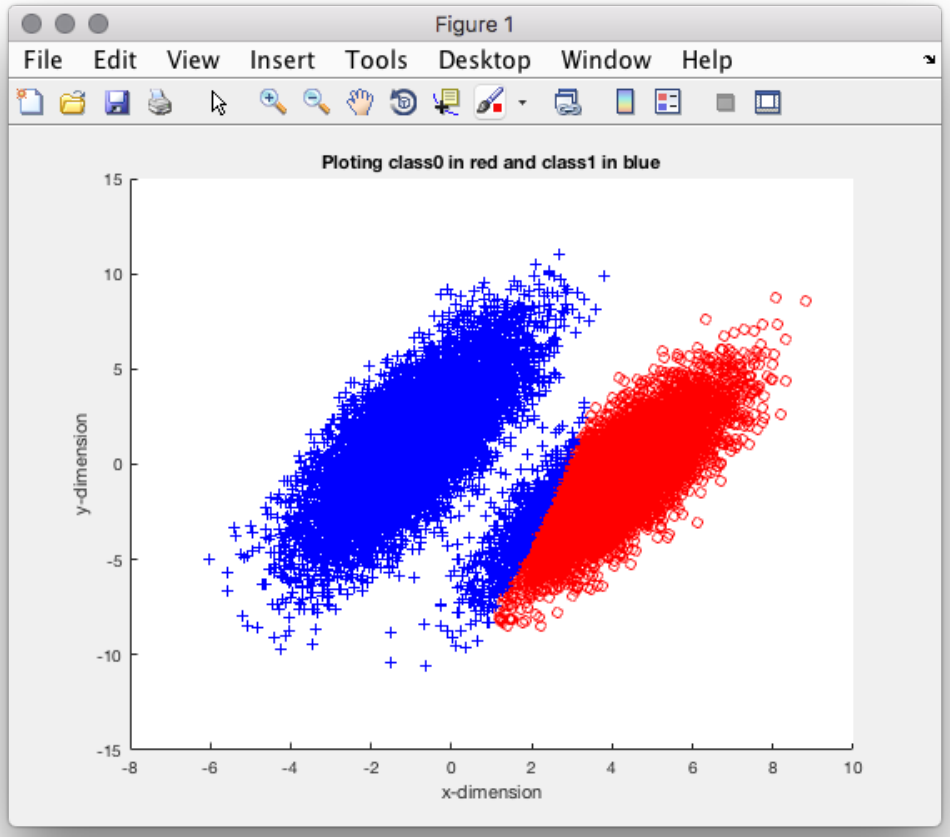


Figure 1
File Edit View Insert Tools Desktop Window Help
Ploting class0 in red and class1 in blue
y-dimension
x-dimension
15
10
5
0
-5
-10
-15
-8
-6
-4
-2
0
2
4
6
8
10