AINT351 – MACHINE LEARNING

Elliott White | 10467243 | MEng Robotics | 4th December 2017

COURSEWORK 1 – DISCRIMINANT PATTERN CLASSIFIER

1. Generate Training & Testing Datasets

Using the supplied Matlab function GenerateGuassianData, we will generate a training and testing dataset of 1000 samples each. The training dataset will then be plotted in two dimensions. The original sample code used the same values for Mean and Sigma for both the training and testing datasets. I have added in two more Means and two more Sigmas so that the testing data is different to the training data.

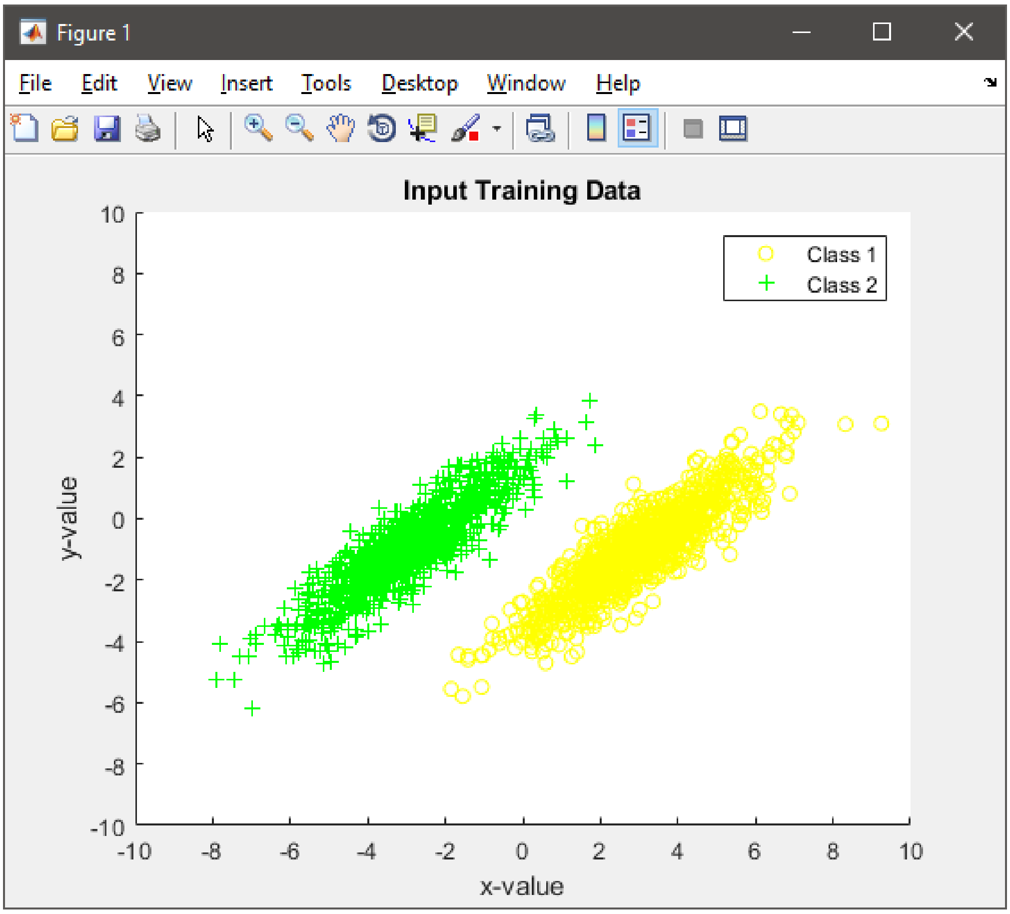
Function: GenerateTrainingAndTestData

Inputs: trainOrTest

Outputs: class1Data, class2Data

```
function [ class1Data, class2Data ] = GenerateTrainingAndTestData(trainOrTest)
%FUNCTION TO GENERATE A TRAINING AND TEST DATASET
% Student Number:   10467243
% Module:           AINT351
% Date:             18/11/2017
% Generate a training dataset of 1000 samples and a testing dataset of 1000
% samples

      % set the mean and covariance for class 0 data points for training data
          Mean1 = [3; -1;];
          Sigma1 = [0.5 0.95; 0.95 4];

          % set the mean and covariance for class 1 data points for training data
          Mean2 = [-3; -1;];
          Sigma2 = [0.5 0.95; 0.95, 4];

          % set the mean and covariance for class 0 data points for testing data
          Mean3 = [5; 2;];
          Sigma3 = [0.6 0.9; 0.25 3];

          % set the mean and covariance for class 1 data points for testing data
          Mean4 = [-6; 2;];
          Sigma4 = [0.1 0.3; 0.35, 1];

          % generate the training dataset
          trainingSamples = 1000;
          [trainingData, trainingTarget] = GenerateGaussianData(trainingSamples,
      Mean1, Sigma1, Mean2, Sigma2);

          % generate the testing dataset
          testingSamples = 1000;
          [testingData, testingTarget] = GenerateGaussianData(testingSamples,
      Mean3, Sigma3, Mean4, Sigma4);


      if strcmp(trainOrTest, 'train')
              % extract all class 1 patterns
              % examine first dimension which is 1 for class 1
              fidx = find(trainingTarget(1,:) == 1);
              class1Data = trainingData(:,fidx);

              % extract all class 2 patterns
              % examine first dimension which is 0 for class 2
              fidx = find(trainingTarget(1,:) == 0);
              class2Data = trainingData(:,fidx);
          elseif strcmp(trainOrTest, 'test')
               % extract all class 1 patterns
```

```
            % examine first dimension which is 1 for class 1
            fidx = find(testingTarget(1,:) == 1);
            class1Data = testingData(:,fidx);

            % extract all class 2 patterns
            % examine first dimension which is 0 for class 2
            fidx = find(testingTarget(1,:) == 0);
            class2Data = testingData(:,fidx);
        end

    % now plot separated classes on a figure
    figure;
    hold on;
    plot(class1Data(1,:), class1Data(2,:), 'yo');
    plot(class2Data(1,:), class2Data(2,:), 'g+');
    axis([-10 10 -10 10]);
    xlabel('x-value');
    ylabel('y-value');
    legend('Class 1', 'Class 2');
    title('Input Training Data');

end
```



2. Generate a Uniform Dataset

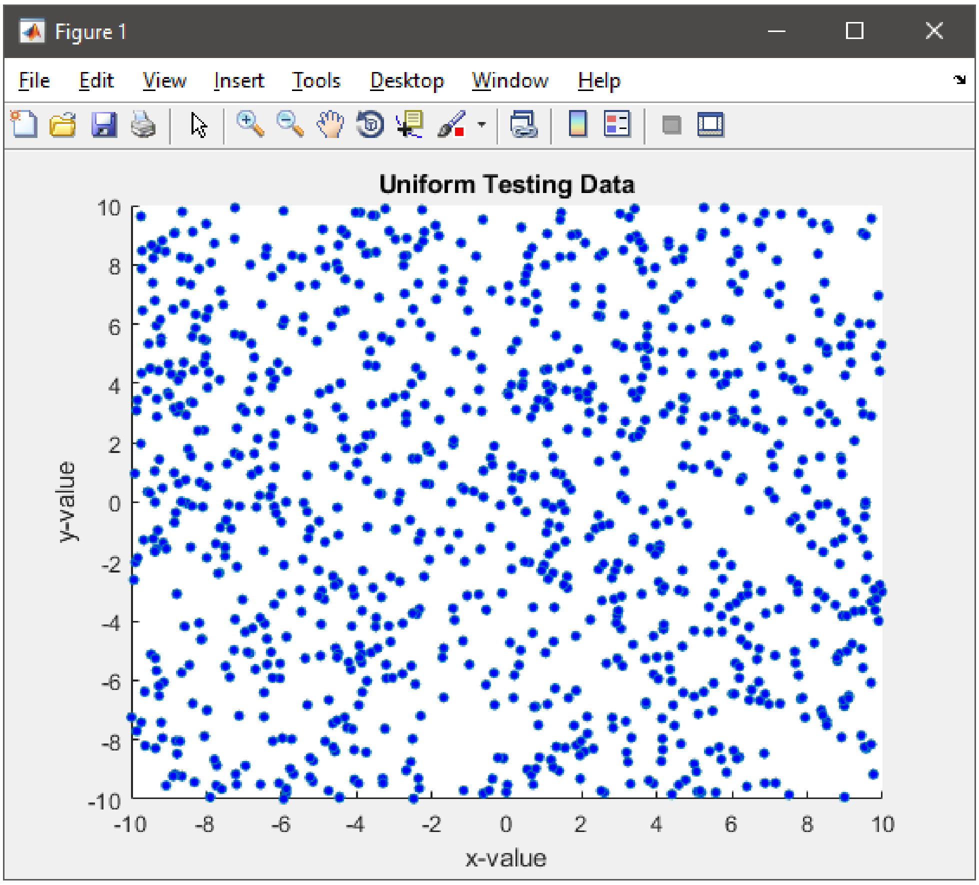
Using the supplied Matlab 'rand' function, we will generate a training dataset of 1000 data points that are uniformly sampled from the interval (-10,-10) to (+10,+10). The points will then be plotted in two dimensions.

Function: GenerateUniformData

Inputs: plotOrNot, dimension

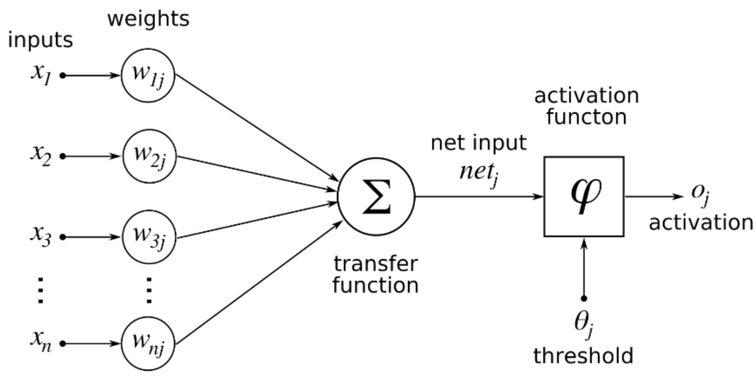
Outputs: uniformData

```
function [ uniformData ] = GenerateUniformData(plotOrNot, dimension)
%FUNCTION TO GENERATE A UNIFORM DATASET
% Student Number:   107467243
% Module:           AINT351
% Date:             18/11/2017

    samples = 1000;             %number of data samples to use

    %create range
    lowerInterval = -10;
    upperInterval = 10;

    %create uniform dataset within the specified range
    uniformData = (upperInterval - lowerInterval).*rand(dimension,samples) +
lowerInterval;

    %decide whether to plot the graph based on plotOrNot input.
    %plots are different based on the dimension given. Limited to one or two
    %dimensions.
    if strcmp(plotOrNot, 'plot')
        if dimension == 1
            % new figure
            figure;
            hold on;
            %plot uniformData
            plot(uniformData, 'o', 'MarkerFaceColor', 'b', 'MarkerSize', 4);
            xlabel('x-value');
            ylabel('y-value');
            title('Uniform Testing Data');
        elseif dimension == 2
            % new figure
            figure;
            hold on;
            %plot first row of uniformData against second row
            plot(uniformData(1,:), uniformData(2,:), 'o', 'MarkerFaceColor', 'b',
'MarkerSize', 4);
            xlabel('x-value');
            ylabel('y-value');
            title('Uniform Testing Data');
        else
            % dimension variable is out of range
            disp('ERROR - Dimension is not 1 or 2');
        end
    elseif strcmp(plotOrNot, 'noplot')
        disp('Not plotting uniform data')
    end


end
```



3. Expression for the Output of a Single Layer Network

Here we have a single layer linear network:



https://upload.wikimedia.org/wikipedia/commons/6/60/ArtificialNeuronModel_english.png

Given the input data vector x = $\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$ and weight vector w = $\begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$:

$$net_j = \sum_{j=1}^{k} w_j x_j = W^T X_j \quad \Longrightarrow \quad o_j(x) = \begin{cases} 1 \; if \; (net_j + \theta_j) > 0 \\ \quad 0 \; otherwise \end{cases}$$

We can incorporate the threshold bias (*Θ_j*) into the input of the transfer function. The new transfer function can then be derived:

$$net_j = \sum_{j=1}^{k} w_j x_j + \theta$$

We can add theta *Θ* to the weights vector by augmenting the W and X vectors:

Input data vector $\tilde{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \\ 1 \end{bmatrix}$ Weight vector $\tilde{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_n \\ \theta \end{bmatrix}$

The full transfer function with the new bias can then be written as:

$$net_j = \sum_{j=1}^{k} w_j x_j + \theta = \tilde{W}^T \tilde{X}_j \quad \Longrightarrow \quad o_j(x) = \begin{cases} 1 \; if \; net_j > 0 \\ 0 \; otherwise \end{cases}$$

4. Network Cost Function

The error function for any point of the input dataset is the difference between the output target vector *(T)*, and the actual output.

Output function: $o_j = net_j = \tilde{W}^T \tilde{X}_j$

Error function: $e_j = T_j - net_j = T_j - \tilde{W}^T \tilde{X}_j$

The overall cost function of all data points can be derived as follows:

Squared error function: $e_j^2 = (T_j - net_j)^2$ This eliminates the negative errors.

Sum squared error function: $e = \sum_{j=1}^{k} e_j^2 = \sum_{j=1}^{k} (T_j - net_j)^2$

$$e = \sum_{j=1}^{k} (T_j - \tilde{W}^T \tilde{X}_j)^2$$

When the cost function decreases, this means that the weights are becoming better, and the output is converging towards the target.

5. Cost Function Gradient

If we have *k* features in the input vector *X* and correspondingly *k* weights in the weight vector *W* then:

$$W = [w_1 \; w_2 \; \cdot \; w_k] \quad \Longrightarrow \quad \frac{\partial e}{\partial W} = \left[\frac{\partial e}{\partial w_1} \; \frac{\partial e}{\partial w_2} \; \cdot \; \frac{\partial e}{\partial w_k}\right]$$

We will re-write the squared error function as:

$$e_j = \frac{1}{2}\left(t_j - o_j\right)^2$$

This is done so the multiplication of 2 is removed when we differentiate the equation.

Let $u_j = t_j - o_j \quad \Longrightarrow \quad e_j = \frac{1}{2}(u_j)^2$

To differentiate this, we use the chain rule:

$$\frac{\partial e_j}{\partial W} = \frac{\partial e_j}{\partial u_j} * \frac{\partial u_j}{\partial W}$$

$$\Rightarrow \frac{\partial e_j}{\partial W} = \frac{\partial}{\partial W}\left[\frac{1}{2}(u_j)^2\right]$$

$$\Rightarrow \frac{\partial e_j}{\partial u_j} = u_j = t_j - o_j$$

$$\Rightarrow \frac{\partial u_j}{\partial \tilde{\mathrm{W}}} = -\frac{\partial o_j}{\partial \tilde{\mathrm{W}}}$$

$$\Rightarrow \frac{\partial e_j}{\partial \tilde{\mathrm{W}}} = -(t_j - o_j)\frac{\partial o_j}{\partial W}$$

$$\frac{\partial o_j}{\partial \tilde{\mathrm{W}}} = \tilde{\mathrm{X}}_j^T$$

Therefore, the full equation for the cost function gradient is:

$$\frac{\partial e_j}{\partial \tilde{\mathrm{W}}} = -(t_j - o_j)\tilde{\mathrm{X}}_j^T$$

$$\Rightarrow \frac{\partial e_j}{\partial \tilde{\mathrm{W}}} = -(t_j - \tilde{\mathrm{W}}^T \tilde{\mathrm{X}}_j)\tilde{\mathrm{X}}_j^T$$

6. Weight Update – The Delta Rule

Using the cost function gradient in association with the weights can help us achieve better classification.

The delta rule updates the weights over iterations of the input data points that are put into the network. The equation for the update per data point is:

$$\widetilde{\mathrm{W}}_{j+1} = \widetilde{\mathrm{W}}_j + \alpha\frac{\partial e_j}{\partial \widetilde{\mathrm{W}}} \qquad \mathit{Where}\ \alpha\ \mathit{is\ the\ learning\ rate. Usually\ between}\ 0\ \&\ 1.$$

As we have previously derived $\frac{\partial e_j}{\partial \tilde{W}}$, the full equation is as follows:

$$\tilde{W}_{j+1} = \tilde{W}_j - \alpha(t_j - \tilde{W}^T\tilde{X}_j)\tilde{X}_j^T$$

7. Implement Single Layer Network Recognition

Implement a single layer network recogniser in Matlab. Using dummy data, test and debug the implementation. The recogniser will generate an overall output activation which will then correspond to one of the two classes.

To achieve binary classification, we need to set the activation function threshold to be 0.5. Values above this correspond to class 1, and the values below the threshold belong to class 2.

Function: DummyDataForSLNRecog

Dependencies: GenerateUniformData

Inputs: plotOrNot

Outputs: inputDataSet, outTargetData

```
function [ inputDataSet, outputTargetData ] = DummyDataForSLNRecog(plotOrNot)
%FUNCTION TO CREATE TEST DATA AND TARGET DATA FOR A SLN RECOGNISER
% Student Number:   10467243
% Module:           AINT351
% Date:             18/11/2017

    inputDataX = GenerateUniformData('noplot',1);   %generate uniform X data values
    inputDataY = GenerateUniformData('noplot',1);   %generate unfiorm Y data values

    inputDataX = inputDataX';   %transpose X data
    inputDataY = inputDataY';   %transpose Y data

    inputDataSet = [inputDataX,  inputDataY];   %concatenate data arrays into 2D
data matrix


    %generate target vectors based on y = mx + c
    m = 2;          %gradient of the target boundary
    c = -3;         %offset of the target boundary
    %generate logical vector indicating which point belongs to which class
    outputTargetData = inputDataY > inputDataX.*m+c;    %target class


    decisionBoundary = inputDataX.*m + c;   %decision boundary is defined by y=mx+c

    %decide whether to plot the graph based on plotOrNot input.
    if strcmp(plotOrNot, 'plot')
        %new figure
        figure;
        hold on;
        plot(inputDataX,decisionBoundary, '-b','LineWidth',3); %plot decision
boundary against X data
        plot(inputDataSet(:,1), inputDataSet(:,2),'or'); %plot Input Data
        axis([-10 10 -10 10]);
        xlabel('x-value');
```

```
        ylabel('y-value');
        title('Input Test Data for Single Layer Network Recognition');
        legend('Decision Boundary','Input Test Data');
     elseif strcmp(plotOrNot, 'noplot')
        disp('Not plotting test data')
    end

end
```

Function: SingleLayerNetworkRecog

Dependencies: DummyDataForSLNRecog, GenerateUniformData

Inputs: [None]

Outputs: [None]

```
function [] = SingleLayerNetworkRecog()
%FUNCTION TO IMPLEMENT A SINGLE LAYER NETWORK RECOGNISER
% Student Number:   10467243
% Module:           AINT351
% Date:             18/11/2017

    %generate uniform input data
    [inputData, ~] = DummyDataForSLNRecog('plot');

    %randomise weights vector
    weights = rand(3,1);

    %create augmented bias vector. One point for every input data point
    biasVector = ones(1000,1);

    %concatenate inputData and biasVector into 1000x3 matrix
    inputData = [inputData, biasVector];

    %network sum function of the network
    NetworkSumFunction = inputData*weights;

    %extract the class 1 points
    Class1Points = NetworkSumFunction > 0.5;        %find class 1 data points
    class1DataPoints = inputData(Class1Points,:);   %create vector containing class
1 points

    %extract the class 2 points
    Class2Points = NetworkSumFunction <= 0.5;       %find class 2 data points
    class2DataPoints = inputData(Class2Points,:);   %create vector containing class
2 points

    %new figure
    figure;
    hold on;
    axis([-10 10 -10 10]);
    xlabel('x-value');
    ylabel('y-value');
    title('Test Data Classified');
    plot(class1DataPoints(:,1),class1DataPoints(:,2),'bx')  %plot class1 X against
Y in blue crosses
    plot(class2DataPoints(:,1),class2DataPoints(:,2),'rx')  %plot class2 X against
Y in red crosses
    legend('Class 1', 'Class 2');    %create legend

end
```

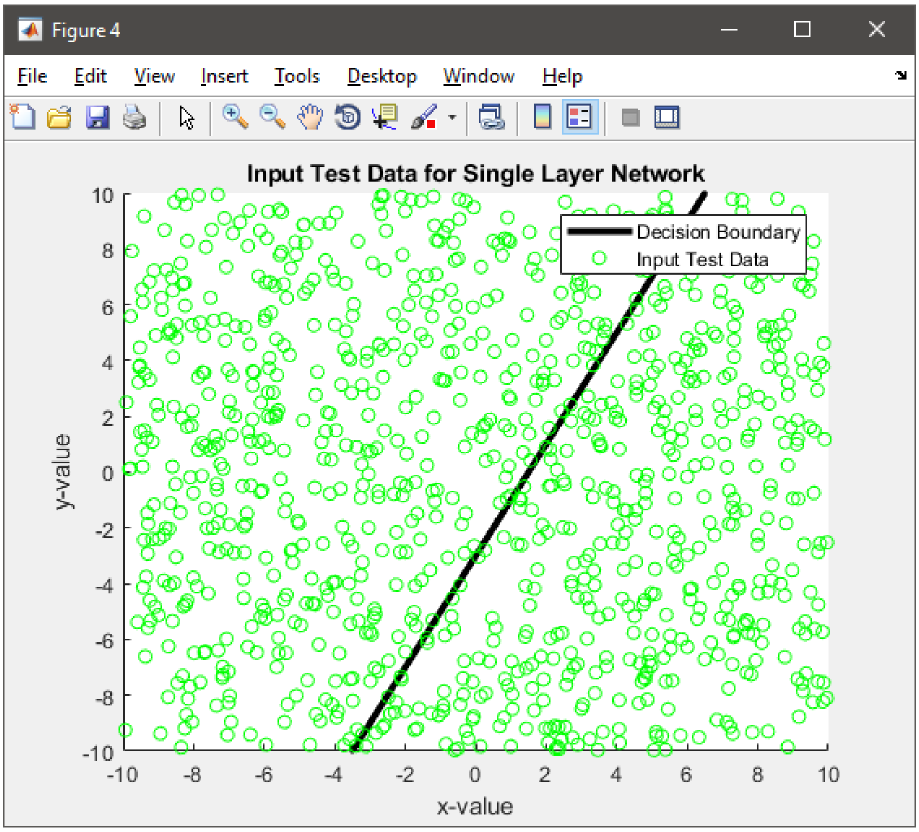


Figure 4
File Edit View Insert Tools Desktop Window Help
Input Test Data for Single Layer Network
Decision Boundary
Input Test Data
y-value
x-value
10
8
6
4
2
0
-2
-4
-6
-8
-10
-10 -8 -6 -4 -2 0 2 4 6 8 10

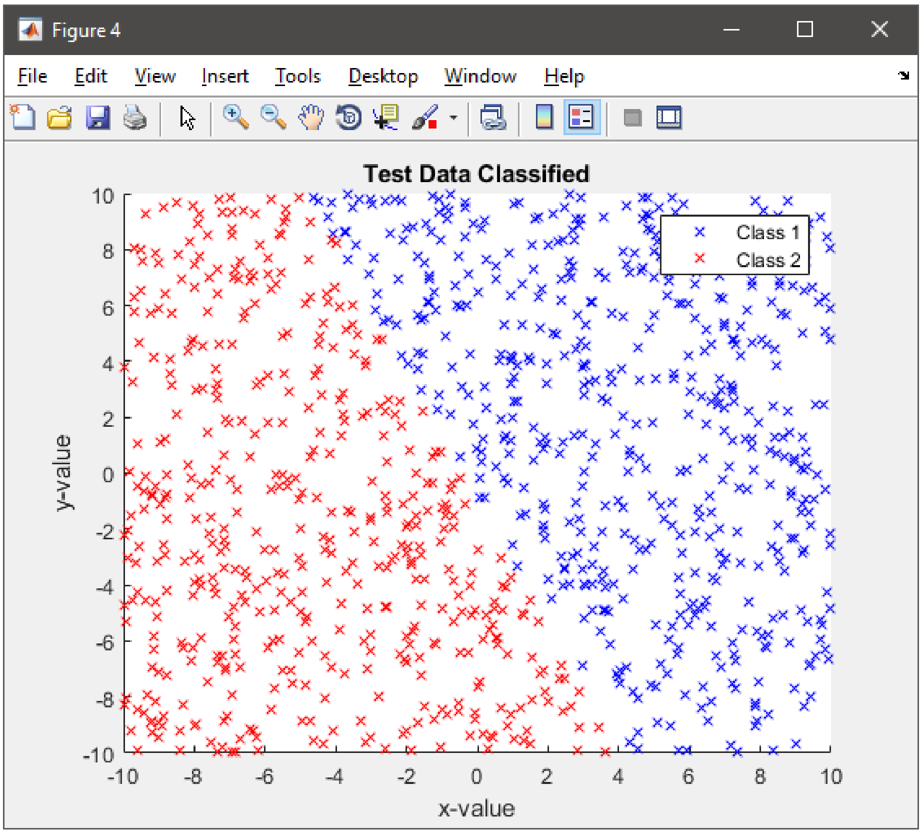


Figure 4
File Edit View Insert Tools Desktop Window Help
Test Data Classified
Class 1
Class 2
y-value
x-value
10
8
6
4
2
0
-2
-4
-6
-8
-10
-10 -8 -6 -4 -2 0 2 4 6 8 10

The classification is very far off due to not training the network. The classification decision boundary is just based on the randomised weights, and if the algorithm is run multiple times, the output classified figure will be different every time.

8. Implement Single Layer Network Training

We will now implement the weight update equations we have previously derived. Updating our weights allows us to train our classifier on the training data. We will run the algorithm several times to see how much the randomised initial weights affects our final classification if we only go through our data once.

Function: SingleLayerNetworkTrain

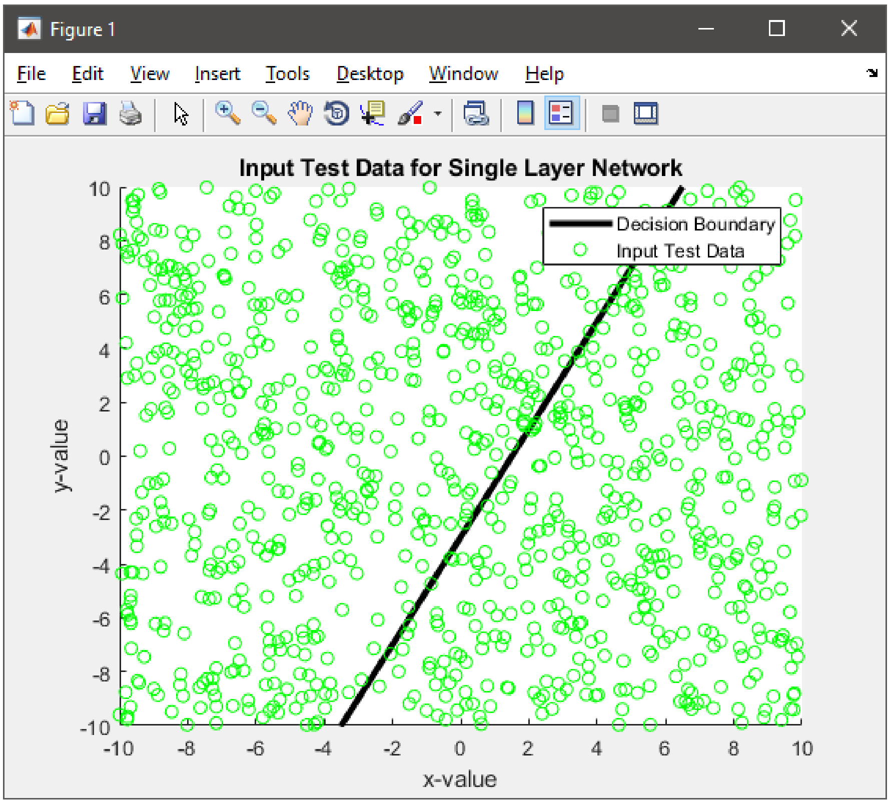
Dependencies: DummyDataForSLNRecog, GenerateUniformData

Inputs: [None]

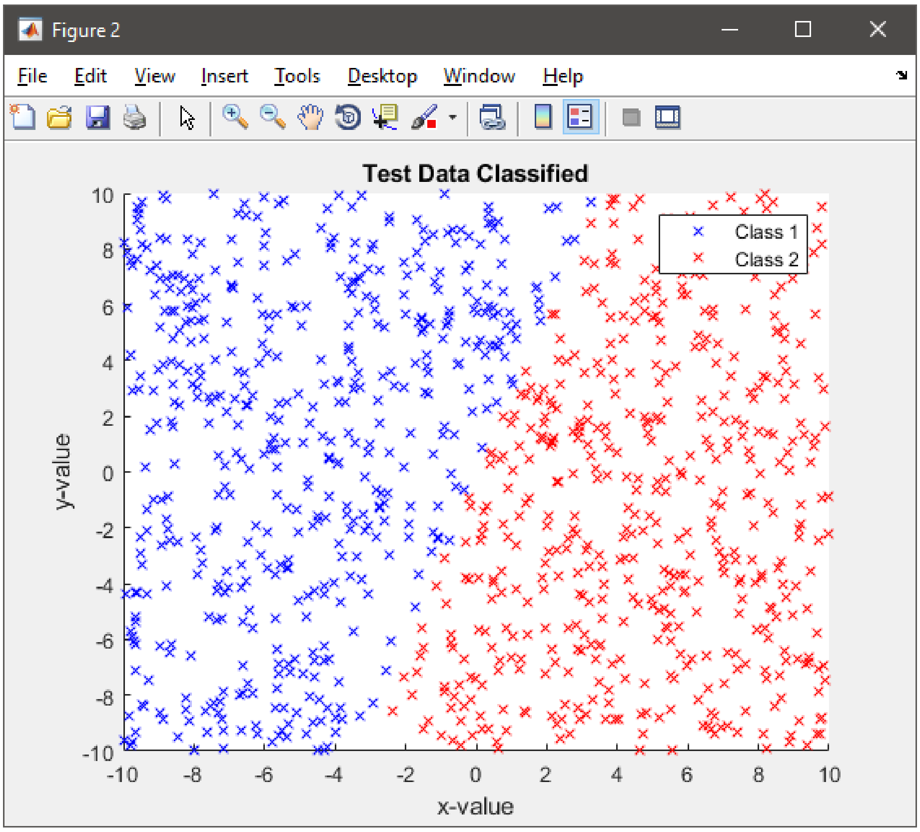
Outputs: [None]

```
function [] = SingleLayerNetworkTrain()
%FUNCTION TO TRAIN A SINGLE LAYER NETWORK
% Student Number:   10467243
% Module:           AINT351
% Date:             18/11/2017

    %generate uniform input data
    [inputData, outputTargetData] = DummyDataForSLNRecog('plot');

    %number of samples
    dataPoints = 1000;

    %randomise initial weights vector
    weights = rand(3,1);

    %learning rate
    alpha = 0.0005;

    %create augmented bias vector. One point for every input data point
    biasVector = ones(dataPoints,1);

    %concatenate inputData and biasVector into 1000x3 matrix
    inputData = [inputData, biasVector];


    %Delta rule for a single layer network
    for i = 1:dataPoints    %iterate through all data points

        %determine output based on input and weights
        actualOutput =  inputData(i,:)*weights;

        %cost function gradient. deltaError / deltaWeights
        dEdW = -(outputTargetData(i)-actualOutput)*(inputData(i,:));

        %update the weights based on the delta rule
        weights = weights - alpha.*dEdW';

    end

    %network sum function of the network based on final weights
```

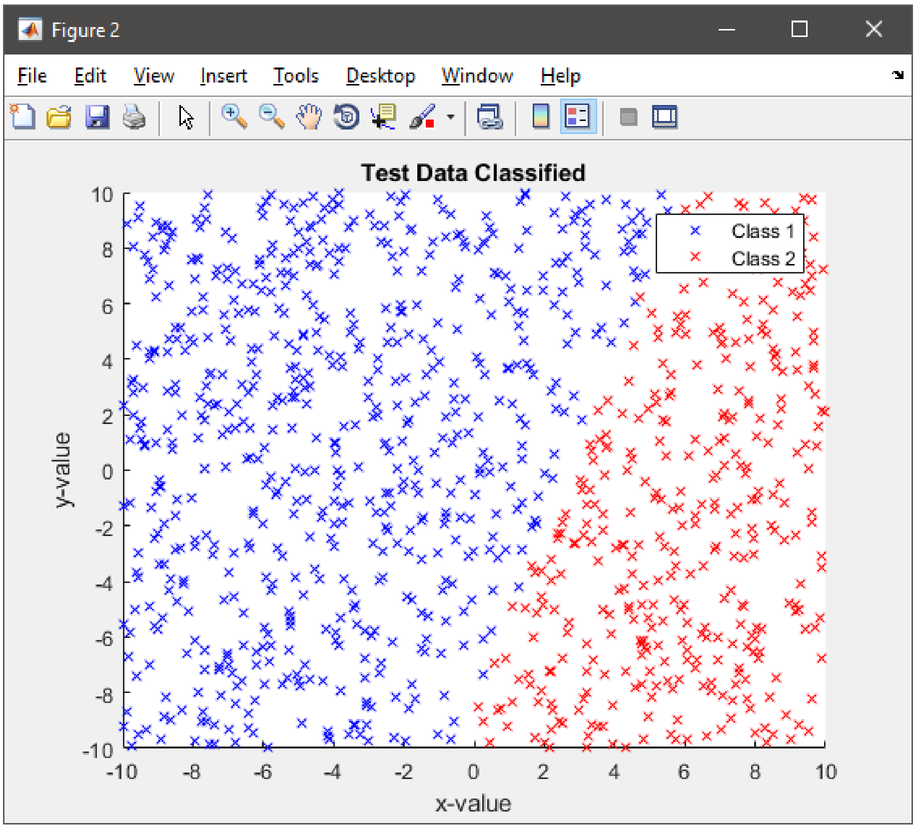
```
    NetworkSumFunction = inputData*weights;

    %extract the class 1 points
    xClassPoints = NetworkSumFunction > 0.5;
    class1DataPoints = inputData(xClassPoints,:);

    %extract the class 2 points
    yClassPoints = NetworkSumFunction <= 0.5;
    class2DataPoints = inputData(yClassPoints,:);

    %new figure
    figure;
    hold on;
    axis([-10 10 -10 10]);
    xlabel('x-value');
    ylabel('y-value');
    title('Test Data Classified');
    plot(class1DataPoints(:,1),class1DataPoints(:,2),'bx')  %plot class1 X against
Y in blue crosses
    plot(class2DataPoints(:,1),class2DataPoints(:,2),'rx')  %plot class2 X against
Y in red crosses
    legend('Class 1', 'Class 2');                           %create legend


end
```



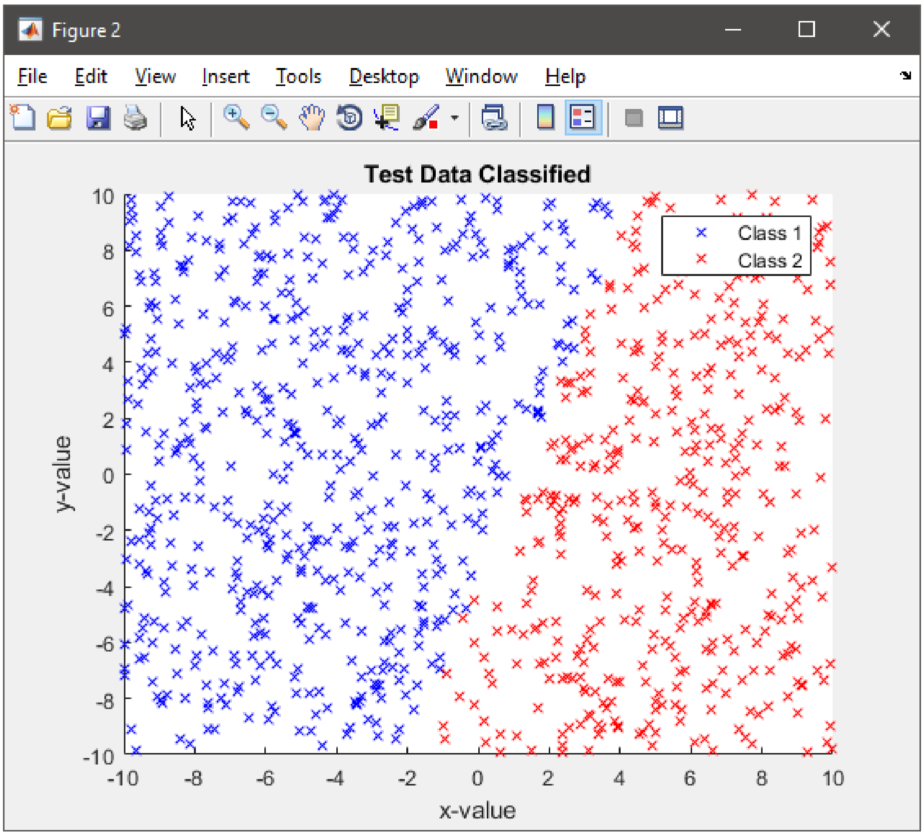
First run through:



Second run through:



Third run through:



We can see that if we only iterate over our dataset once we still do not achieve perfect classification.

9. Train the Network and Plot the Error

We will now train the linear single layer network using the delta rule on the training dataset we generated in part 1. We will then calculate and plot the sum square error on each iteration of the algorithm. We will be able to see how much the error improves every time we go over the dataset.

We will then change the learning rate to see how much it affects the rate at which the error improves.

Function: SingleLayerNetworkTrain_WithErr

Dependencies: GenerateTrainingAndTestData, GenerateGaussianData

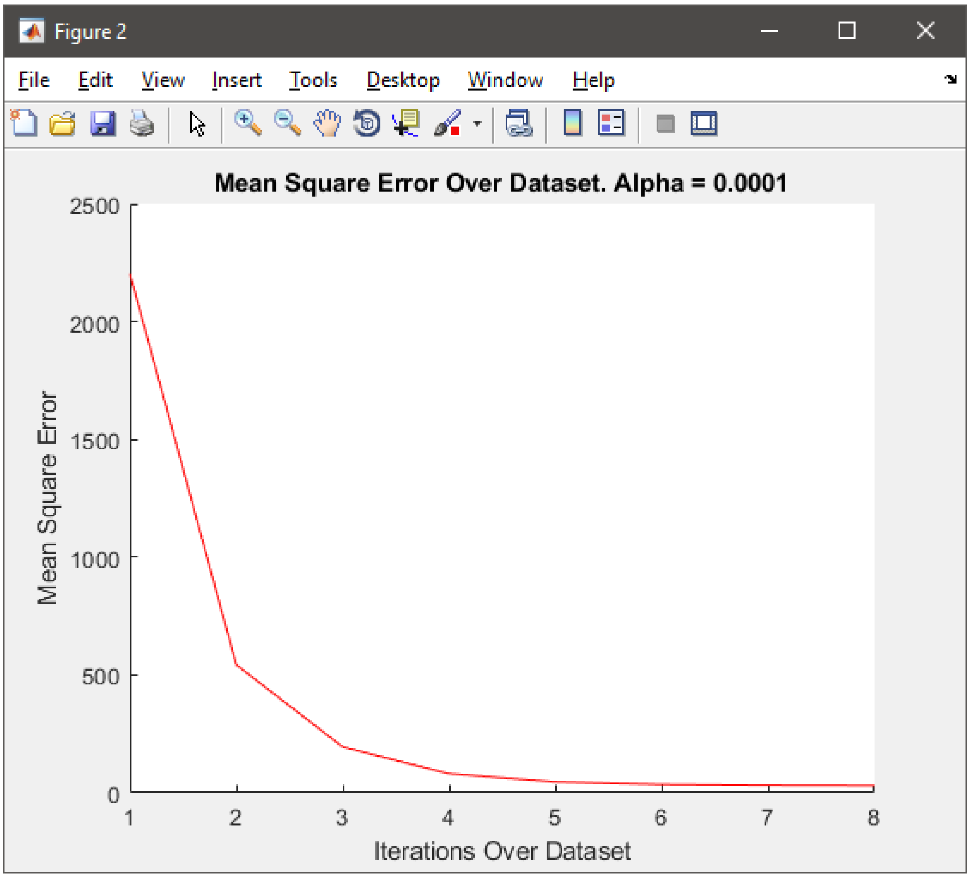
Inputs: [None]

Outputs: finalWeights

```
function [ finalWeights ] = SingleLayerNetworkTrain_WithErr()
%FUNCTION TO TRAIN A SINGLE LAYER NETWORK AND PLOT THE ERROR
% Student Number:   10467243
% Module:           AINT351
% Date:             18/11/2017
```

```
%generate class 1 & class 2 training data
[ class1Data, class2Data ] = GenerateTrainingAndTestData('train');


%number of data points
dataPoints = 1000;

%randomise initial weights vector
weights = rand(3,1);

%learning rate
alpha = 0.00005;

%create augmented bias vector. One point for every input data point
biasVector = ones(1, dataPoints);

%concatenate classData and biasVector into 1000x3 matrix. Do this for
%class 1 and class 2
class1Data = [class1Data', biasVector'];
class2Data = [class2Data', biasVector'];

%create target data. 1s for class 1. 0s for class 2
c1Target = ones(1,dataPoints);
c2Target = zeros(1,dataPoints);

%concatenate both classDatas into inputData which is now a 3x2000 matrix
inputData = [class1Data', class2Data'];
%transpose inputData into 2000x3
inputData = inputData';
%concatenate the target data into outputTargetData which is now 1000x2
%matix
outputTargetData = [c1Target', c2Target'];

%create empty vector of error values. The index corresponds to the time
%we iterate over the dataset
finalError = [];

%for loop to iterate over the whole dataset
for j = 1:20

    %reset the error for the current iteration
    networkError = 0;

    %iterate over all the data points
    for I = 1:2000

        %determine output based on input and weights
        actualOutput =  inputData(I,:)*weights;

        %cost function gradient. deltaError / deltaWeights
        dEdW = -(outputTargetData(i)- actualOutput)*(inputData(I,:));

        %update the weights based on the delta rule
        weights = weights - alpha.*dEdW';

        %calculate the sum squared error
        networkError = networkError + (outputTargetData(i) -  actualOutput)^2;

    end

    %put the error for this iteration into the finalError array
    finalError(j) = networkError;

    %check if the difference between the current error and the last
    %error is small (less than 2)
    %if it is, then break out of overall for-loop
    if j > 1
```

```
            if finalError(j) > (finalError(j-1) - 2)
                break
            end
        end

    end

    %output our final weights so we can use them to test on other datasets
    finalWeights = weights;

    %new figure
    figure;
    hold on;
    plot(finalError,'r')     %plot our finalError array
    xlabel('Iterations Over Dataset');
    ylabel('Mean Square Error');
    title('Mean Square Error Over Dataset');


end
```



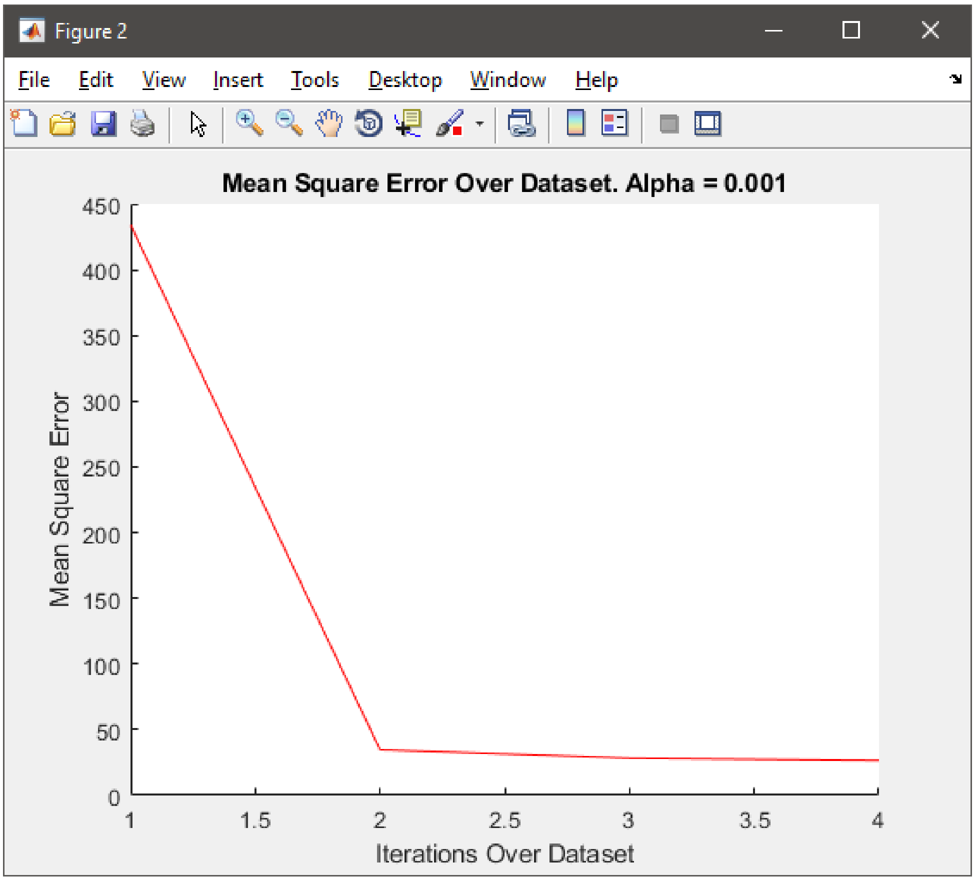


Figure 2
File Edit View Insert Tools Desktop Window Help
Mean Square Error Over Dataset. Alpha = 0.001
Mean Square Error
Iterations Over Dataset
0
50
100
150
200
250
300
350
400
450
1
1.5
2
2.5
3
3.5
4

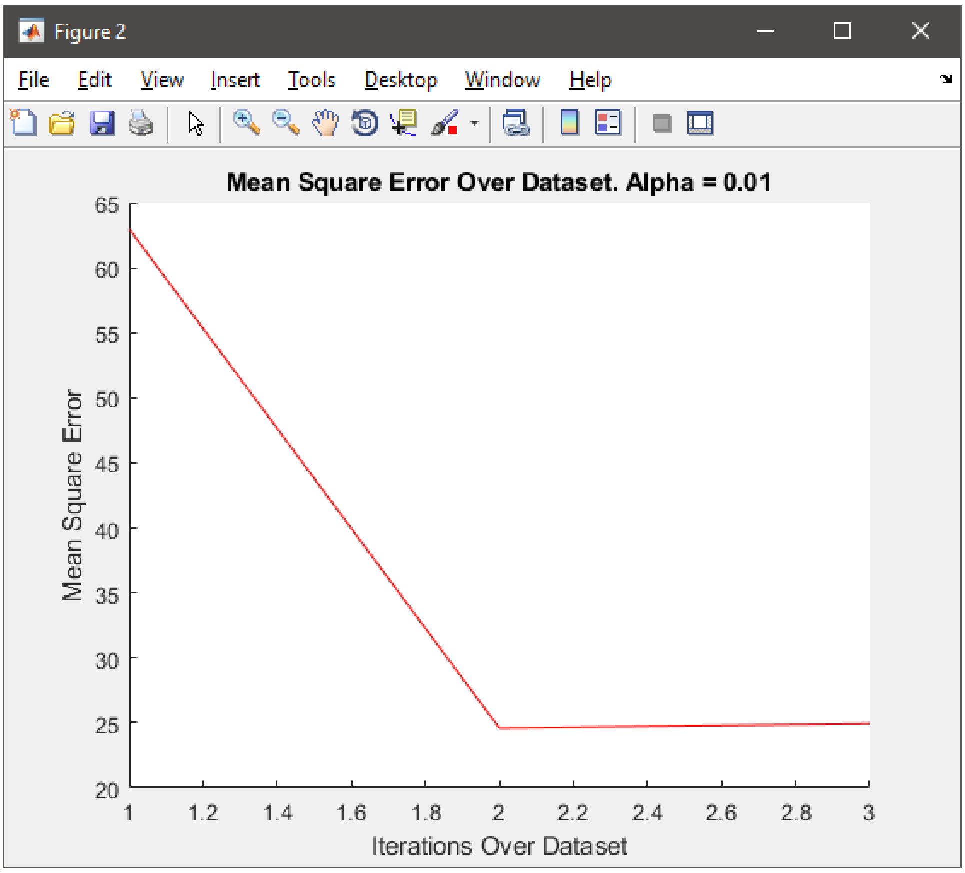
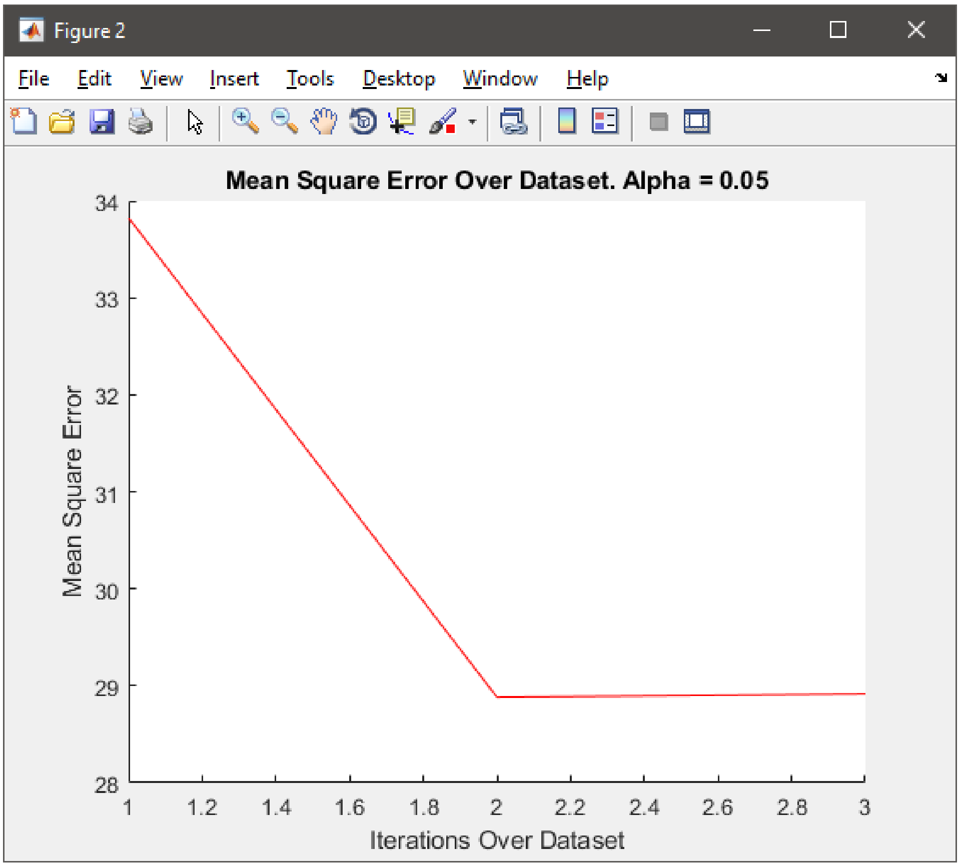


Figure 2
File Edit View Insert Tools Desktop Window Help
Mean Square Error Over Dataset. Alpha = 0.01
Mean Square Error
Iterations Over Dataset
20
25
30
35
40
45
50
55
60
65
1
1.2
1.4
1.6
1.8
2
2.2
2.4
2.6
2.8
3



Having our learning rate too small means that we may take quite a few iterations over our dataset before the difference in error between iterations is small enough. A very small learning rate also causes the initial mean square error to be very large.

10. Train the Network and Plot the Error

We will now test our network on the testing data provided by our GenerateGuassianData function. We will pass ‘test’ to the function so it produces different data than that we trained on. We will then also determine the correct classification percentage.

Function: SingleLayerNetworkTest

Dependencies: GenerateTrainingAndTestData, GenerateGaussianData

Inputs: finalWeights

Outputs: [None]

```
function [] = SingleLayerNetworkTest(finalWeights)
%FUNCTION TO TEST A SINGLE LAYER NETWORK
% Student Number:   10467243
% Module:           AINT351
% Date:             18/11/2017

    %generate class 1 & class 2 testing data
    [ class1Data, class2Data ] = GenerateTrainingAndTestData('test');

    %number of data points
    dataPoints = 1000;
```

```
    %weights are the final weights used from the training
    weights = finalWeights;

    %create augmented bias vector. One point for every input data point
    biasVector = ones(1, dataPoints);

    %concatenate classData and biasVector into 1000x3 matrix. Do this for
    %class 1 and class 2
    class1Data = [class1Data', biasVector'];
    class2Data = [class2Data', biasVector'];

    %concatenate both classDatas into inputData which is now a 3x2000 matrix
    inputData = [class1Data', class2Data'];
    %transpose inputData into 2000x3
    inputData = inputData';

    %network sum function of the network
    NetworkSumFunction = inputData*weights;

    %extract the class 1 points
    Class1Points = NetworkSumFunction > 0.5;        %find class 1 data points
    class1DataPoints = inputData(Class1Points,:);   %create vector containing class
1 points

    %extract the class 2 points
    Class2Points = NetworkSumFunction <= 0.5;       %find class 2 data points
    class2DataPoints = inputData(Class2Points,:);   %create vector containing class
2 points

    %count number points in the first 1000 of NetworkSumFunction that are
    %bigger than 0.5. This indicates the number of correctly classed
    %Class1 points
    numberOfClass1Correct = sum(NetworkSumFunction(1:1000) > 0.5);
    %count number points in the second 1000 of NetworkSumFunction that are
    %less than or equal to 0.5. This indicates the number of correctly classed
    %Class2 points
    numberOfClass2Correct = sum(NetworkSumFunction(1001:2000) <= 0.5);
    %calculate overal percentage of correctly classed points
    percentageOfClassesCorrect = ((numberOfClass1Correct + numberOfClass2Correct) /
2000) * 100;

    %print to console
    fprintf('Percentage of correctly classified data: %d%%',
percentageOfClassesCorrect);

    %new figure
    figure;
    hold on;
    plot(class1DataPoints(:,1), class1DataPoints(:,2), 'bo');   %plot class1 X
against Y in blue crosses
    plot(class2DataPoints(:,1), class2DataPoints(:,2), 'r+');   %plot class2 X
against Y in red crosses
    axis([-10 10 -10 10]);
    xlabel('x-value');
    ylabel('y-value');
    legend('Class 1', 'Class 2');   %create legend
    title('Output Single Layer Network Tested on Guassian Data');

end
```

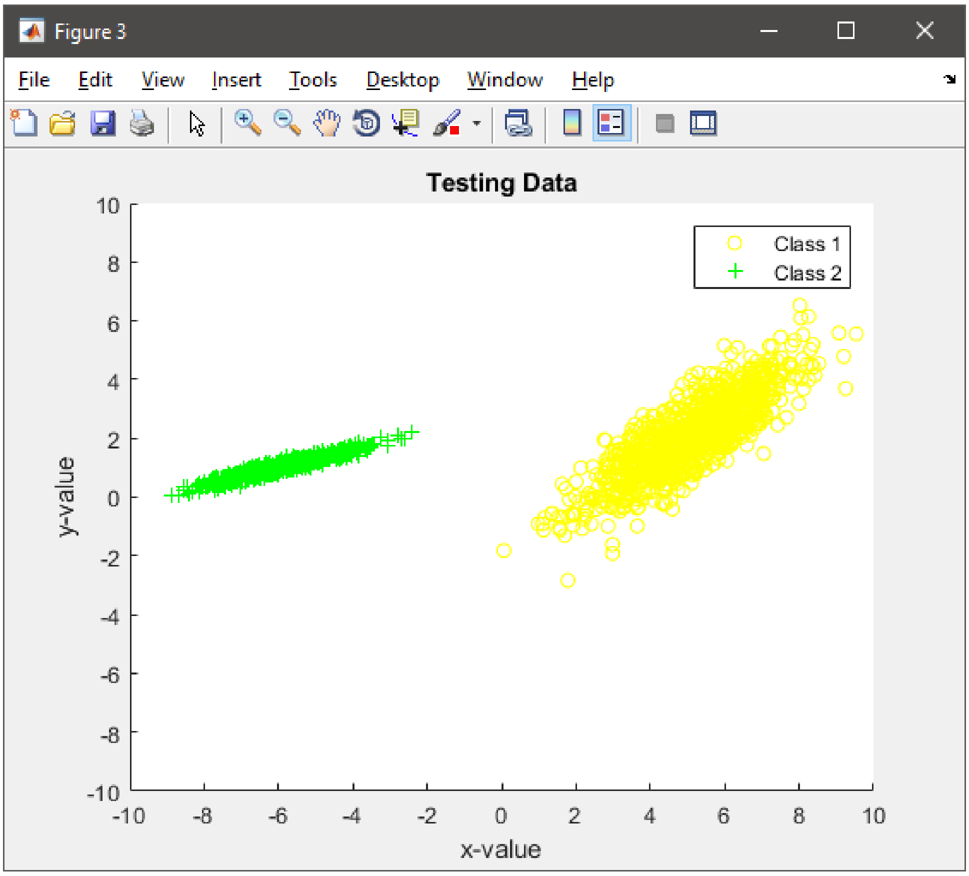


Figure 3
File Edit View Insert Tools Desktop Window Help
Testing Data
Class 1
Class 2
y-value
x-value
10
8
6
4
2
0
-2
-4
-6
-8
-10
-10 -8 -6 -4 -2 0 2 4 6 8 10

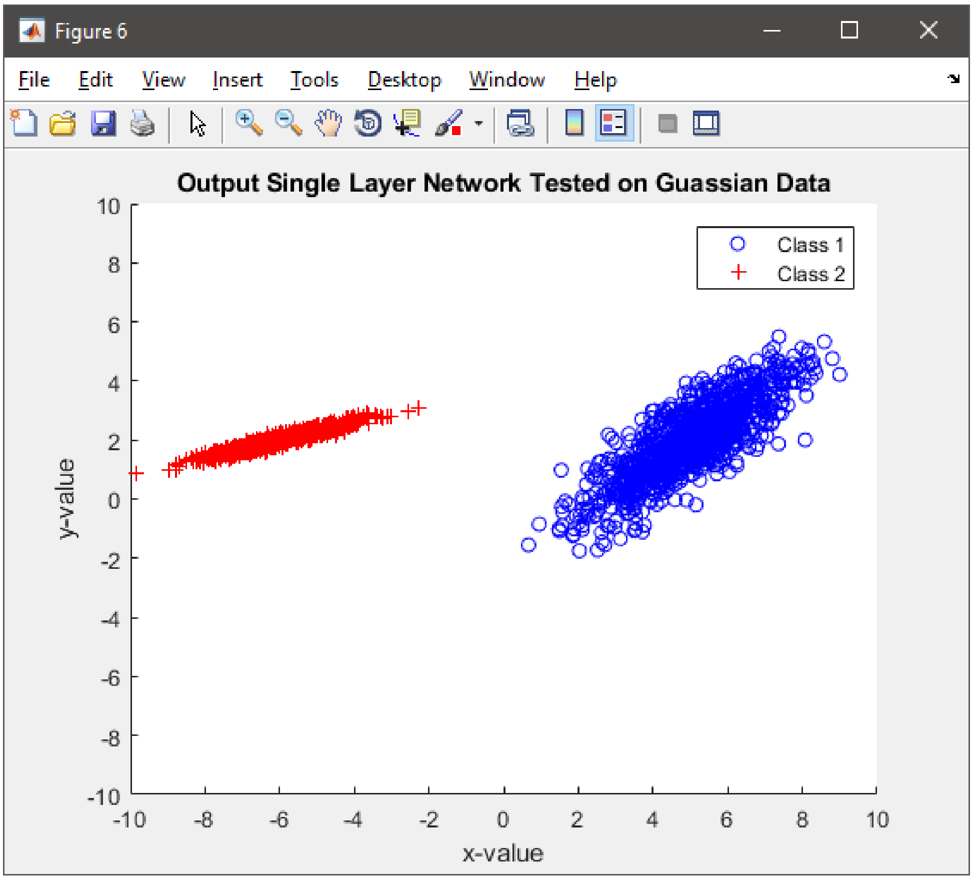


Figure 6
File Edit View Insert Tools Desktop Window Help
Output Single Layer Network Tested on Guassian Data
Class 1
Class 2
y-value
x-value
10
8
6
4
2
0
-2
-4
-6
-8
-10
-10 -8 -6 -4 -2 0 2 4 6 8 10

Console output:

```
fx Percentage of correctly classified data: 100%K>>
```

The reason for perfect classification is due to the fact that the two data sets are clearly separate. If I had not changed the mean and sigma parameters for the testing dataset then the reason for perfect classification would be because the testing data is exactly the same as the training data.

As soon as there is any overlap in the two classes of the testing data, then the percentage of correctly classified data decreases. This is due to the network trying to apply a linear boundary based on the training data, and cannot do this when there is overlap.

11. Test the Network on the Uniform Data Set

We will now test our network on the uniform dataset we have created, using GenerateUniformData.

Function: SingleLayerNetworkTestOnUniformData

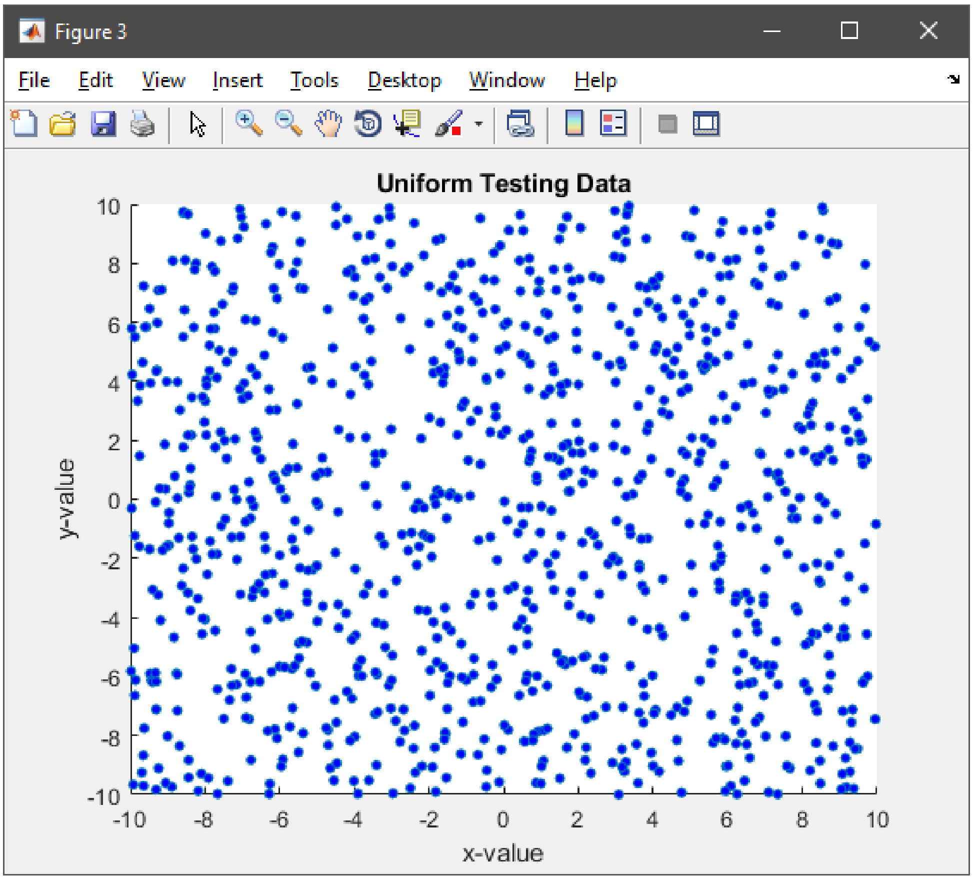
Dependencies: GenerateUniformData

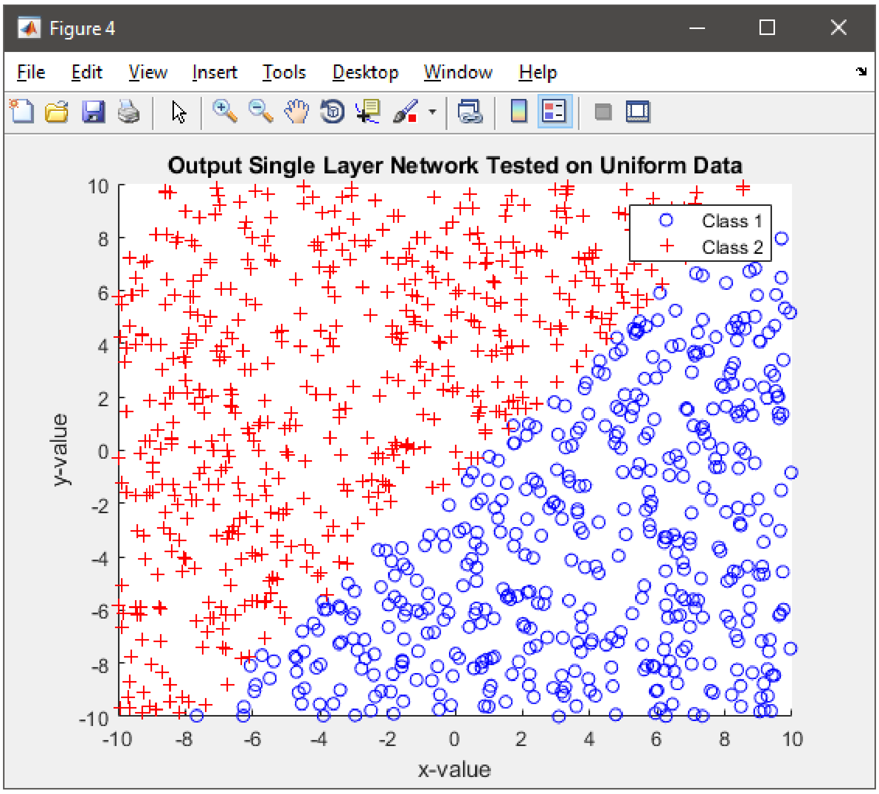
Inputs: finalWeights

Outputs: [None]

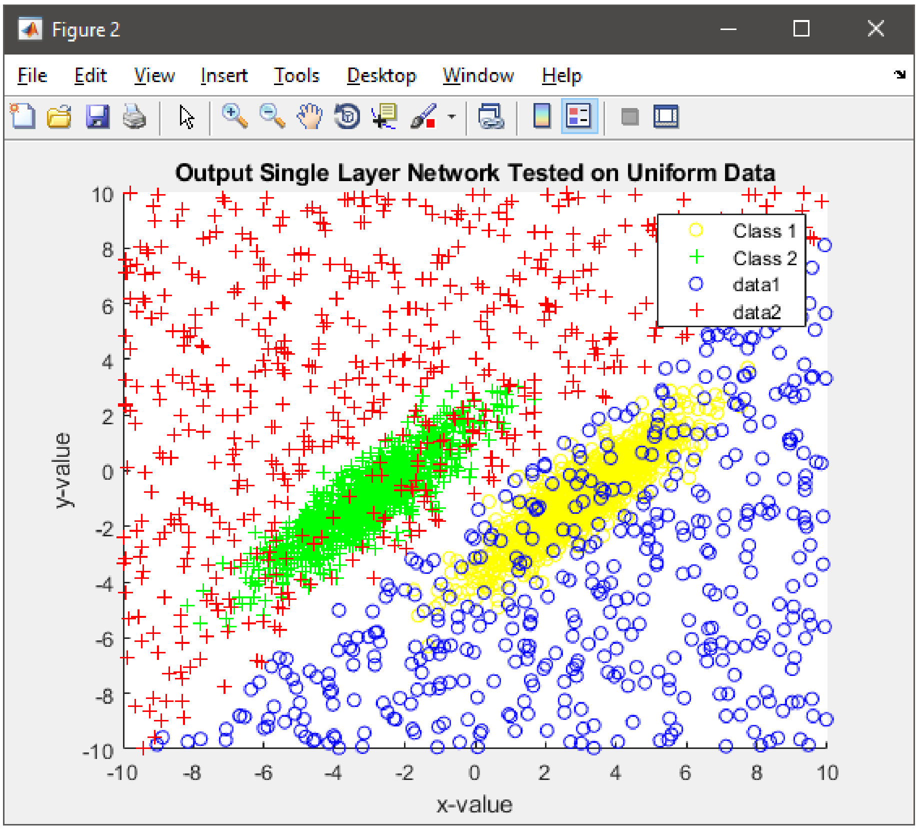
```
function [] = SingleLayerNetworkTestOnUniformData(finalWeights)
%FUNCTION TO TEST A SINGLE LAYER NETWORK ON UNIFORM DATA
% Student Number:   10467243
% Module:           AINT351
% Date:             18/11/2017

    %generate uniform data with 2 dimensions
    [ uniformData ] =  GenerateUniformData('plot', 2);

    %number of data points
    dataPoints = 1000;

    %weights are the final weights used from the training
    weights = finalWeights;

    %create augmented bias vector. One point for every input data point
    biasVector = ones(1, dataPoints);

    %concatenate uniformData and biasVector into 1000x3 matrix
    inputData = [uniformData', biasVector'];

    %network sum function of the network
    NetworkSumFunction = inputData*weights;

    %extract the class 1 points
    Class1Points = NetworkSumFunction > 0.5;         %find class 1 data points
    class1DataPoints = inputData(Class1Points,:);    %create vector containing class
1 points

    %extract the class 2 points
```

```
    Class2Points = NetworkSumFunction <= 0.5;        %find class 2 data points
    class2DataPoints = inputData(Class2Points,:);     %create vector containing
class 2 points

    %new figure
    figure;
    hold on;
    plot(class1DataPoints(:,1), class1DataPoints(:,2), 'bo');    %plot class1 X
against Y in blue crosses
    plot(class2DataPoints(:,1), class2DataPoints(:,2), 'r+');    %plot class2 X
against Y in red crosses
    axis([-10 10 -10 10]);
    xlabel('x-value');
    ylabel('y-value');
    legend('Class 1', 'Class 2');
    title('Output Single Layer Network Tested on Uniform Data');

end
```

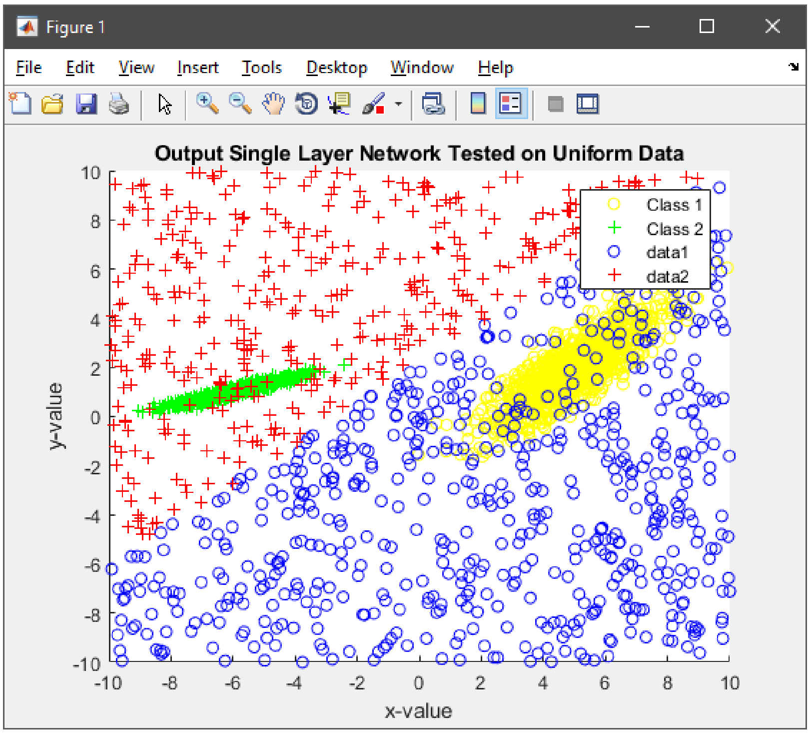




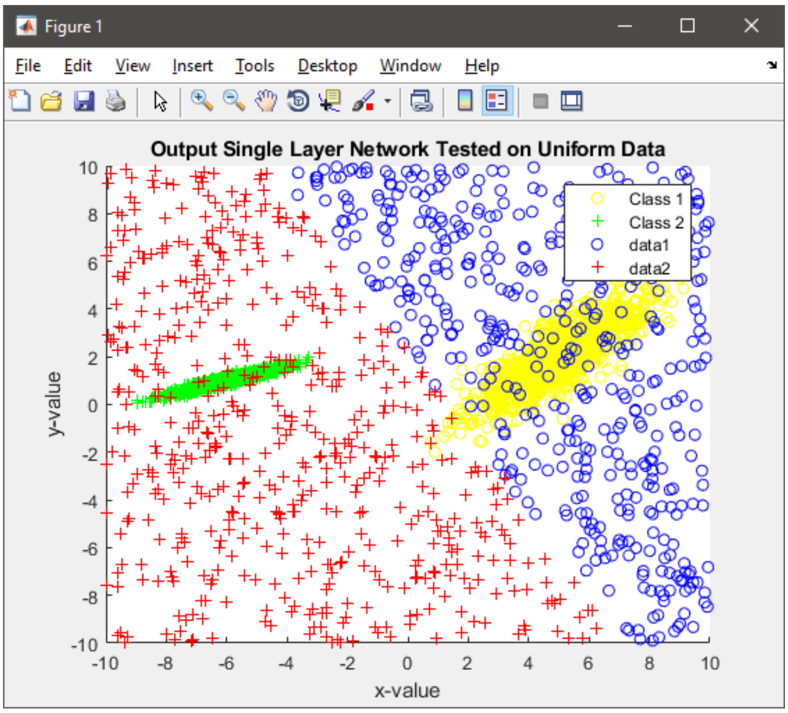
The classification boundary on the uniform dataset is determined by the Gaussian training data. If we super-imposed the decision boundary from the training dataset onto our uniform data, we would find that it fits along the classification boundary.



If we change the mean and sigma values from our training input, we can see how the decision boundary tries to change with the new training data. We can then see how this changes the decision boundary of our uniform data.



If we restart and run the algorithm a few times, we can see how the decision boundary changes every time we re-train the network. As the two datasets are far apart, the decision boundary can vary and won't be the same every time.



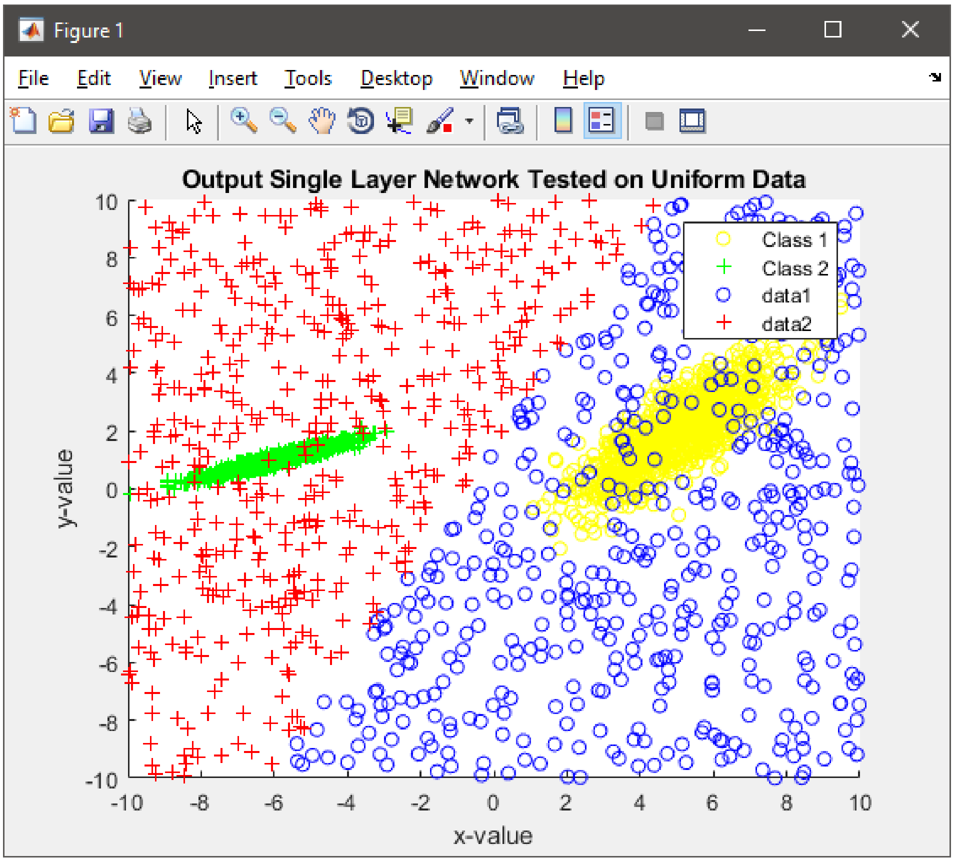


Figure 1
File Edit View Insert Tools Desktop Window Help
Output Single Layer Network Tested on Uniform Data
Class 1
Class 2
data1
data2
y-value
x-value
10
8
6
4
2
0
-2
-4
-6
-8
-10
-10
-8
-6
-4
-2
0
2
4
6
8
10

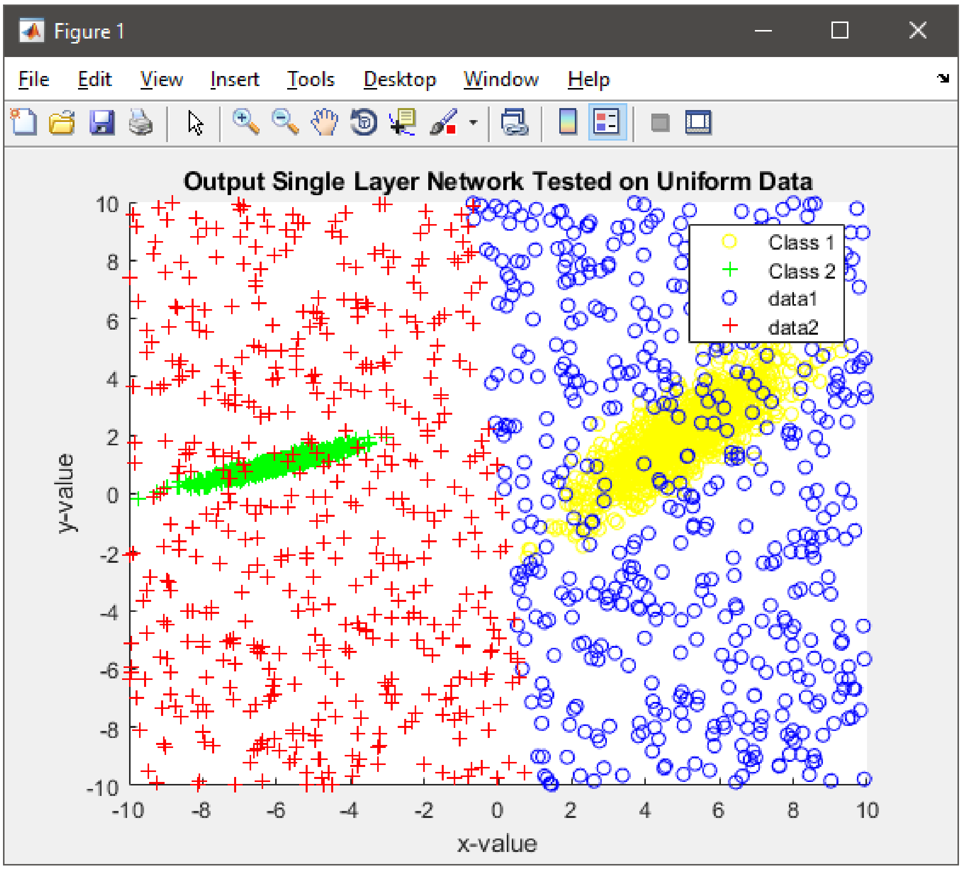


Figure 1
File Edit View Insert Tools Desktop Window Help
Output Single Layer Network Tested on Uniform Data
Class 1
Class 2
data1
data2
y-value
x-value
10
8
6
4
2
0
-2
-4
-6
-8
-10
-10
-8
-6
-4
-2
0
2
4
6
8
10

COURSEWORK 2 – REINFORCEMENT LEARNING

Due to me completing the lab work before a solution to the Q-Learning algorithm was given, I have written my own implementation. The following pages will show and describe my solution.

File: Q_Learning_Exercises.m

Dependencies: initQ.m, initQPOMDP.m, initQSTM.m, trialTrainer.m, shadedErrorBar.m,

Description: Top level file to execute different Q-Learning algorithms. The file is split into three primary parts: one to execute a single variation of Q-Learning, another to execute another variation of Q-Learning used to compare to the previous algorithm (this contains the ttest() and Mann-Whitney U-Test functions), and lastly a part used to plot the quartile performance of a selected algorithm. I will embed the first part here then embed the other two parts when needed for later questions.

Inputs: [None]

Outputs: Multiple figures

FIRST PART:

```
%=================================================================
%{
TOP LEVEL FILE TO EXECUTE VARIATIONS OF Q-LEARNING
    Author:             Elliott White
    Student Number:     10467243
    Date:               04/12/2017
    Module:             AINT351
%}
%=================================================================

close all;

global episodeTrack;
global episodeStepMeanSTD;
global doDynaQ;
global doPOMDP;
global doSTM;
global goalState;
global numberOfTrials;
global numberOfEpisodes;
global temporalDiscountRate;
global explorationRate;
global learningRate;
global doComparison;
global doTTest;
global doUTest;
global doQuarPerf;

%select main training algorithm only one of these can be true based on which task to execute
%if all are false then we execute normal Q-learning
doDynaQ = false;
doPOMDP = false;
doSTM = false;

%logic to decide if comparing two algorithms or not
doComparison = false;
%logic to decide which comparison test to do
doTTest = true;
doUTest = false;

%logic to decide to do quartile performance plotting
```

```
doQuarPerf = true;

numberOfTrials = 50;
numberOfEpisodes = 200;

goalState = 2;

temporalDiscountRate = 0.9;
explorationRate = 0.1;
learningRate = 0.2;

%array of episode numbers used to store mean and standard deviation for these episodes only
episodeTrack = [1 2 5 10 15 20 30 50 75 100 150 200];

%matrix to store means and standard deviations for the episodes on episodeTrack
%first row is means, second row is standard deviation
episodeStepMeanSTD = zeros(2, size(episodeTrack,2));

%array to store to total number of steps per episode of every trial. This is used to calculate
the average number of steps per episode
totalStepsPerTrial = zeros(1,numberOfEpisodes);

%matrix to store all steps for every episode for every trial
rawData = zeros(numberOfTrials,numberOfEpisodes);

%matrices to store the number of steps of every trial for the episopdes in episodeTrack. This
will be used to compare the two algorithms using test algorithms and also used to plot the
quartile performances
rawDataForEpisodeTrackAlg1 = zeros(numberOfTrials,size(episodeTrack,2));
rawDataForEpisodeTrackAlg2 = zeros(numberOfTrials,size(episodeTrack,2));
rawDataForEpisodeTrackAlg3 = zeros(numberOfTrials,size(episodeTrack,2));


%=============================================================================
%INITIAL Q-LEARNING
for i=1:numberOfTrials

    disp('Trial Number');
    disp(i);

    %initialise the Q-Table for each trial. Q-Table is different size depending on which
algorithm is implemented
    if doDynaQ == true
        QTable = initQ(0.01,0.1);
    elseif doPOMDP == true
        QTable = initQPOMDP(0.01,0.1);
    elseif doSTM == true
        QTable = initQSTM(0.01,0.1);
    else %normal Q-Learning
        QTable = initQ(0.01,0.1);
    end

    %used to set limits on the surf plot
    QTableSize = size(QTable);
    QTableSizeCol = QTableSize(1);

    %begin a trial of the selected Q-Learning algorithm
    [stepsPerEpisode,finalQTable] = trialTrainer(QTable,numberOfEpisodes);

    %store the number of steps it took per episode
    totalStepsPerTrial = totalStepsPerTrial + stepsPerEpisode;

    %put the number of steps per episode into the rawData matrix
    rawData(i,:) = stepsPerEpisode;

    %store the mean number of steps for an episode in episodeTrack into the rawData..Alg1
matrix
    rawDataForEpisodeTrackAlg1(i,:) = episodeStepMeanSTD(1,:);

    if (i ~= 1)
        for j=2:i
            %as episodeStepMeanSTD is accumulative, so the means can be
            %calculated, we need to find out the difference in step count
            %between two trials and adjust the matrix accordingly
```

```
            rawDataForEpisodeTrackAlg1(i,:) = rawDataForEpisodeTrackAlg1(i,:) -
rawDataForEpisodeTrackAlg1(j-1,:);
        end
    end

end

%calculate the average number of steps per episode
averageStepsPerEpisode = totalStepsPerTrial / numberOfTrials;

%calculate the means and standard deviations for the episodes in
%episodeTrack
episodeStepMeanSTD(1,:) = episodeStepMeanSTD(1,:) / numberOfTrials;
episodeStepMeanSTD(2,:) = std(rawDataForEpisodeTrackAlg1);

%create cell matrix to store the means and standard deviations
meanAndSTD = cell(3, size(rawDataForEpisodeTrackAlg1,2) + 1);
cellRowTitles = {'Episode Number', 'Mean', 'Standard Deviation'};


for i=1:3
    meanAndSTD{i,1} = cellRowTitles{i};
end
for i=2:size(episodeTrack,2)+1
    meanAndSTD{1,i} = episodeTrack(i-1);
end
for i=2:size(episodeTrack,2)+1
    meanAndSTD{2,i} = episodeStepMeanSTD(1,i-1);
end
for i=2:size(episodeTrack,2)+1
    meanAndSTD{3,i} = episodeStepMeanSTD(2,i-1);
end

meanAndSTD = meanAndSTD';

%plot the intial and final Q-Table on surf plots
figure;
subplot(1,2,1);
surf(QTable);
zlim([0 1]);
ylim([0 QTableSizeCol]);
title('Initial Q Table');
xlabel('Action');
ylabel('State')

pause(0.1);

subplot(1,2,2);
max1 = max(finalQTable);
max2 = max(max1);
surf(finalQTable);
zlim([0 max2]);
ylim([0 QTableSizeCol]);
title('Final Q Table');
xlabel('Action');
ylabel('State')

pause(0.1);

%plot the Q-Learning performance improvement of the last trial
figure;
plot(stepsPerEpisode);
if doDynaQ == true
    title('Dyna-Q-Learning Performance Improvement (of last trial)');
elseif doPOMDP == true
    title('Q-Learning for POMDP Performance Improvement (of last trial)');
elseif doSTM == true
    title('Q-Learning for STM Performance Improvement (of last trial)');
else
    title('Q-Learning Performance Improvement (of last trial)');
end
xlabel('Episode Number');
ylabel('Number of Steps');

%plot the average number of steps per episode to see the improvement over time
figure;
```

```
x=1:size(averageStepsPerEpisode,2);
shadedErrorBar(x,averageStepsPerEpisode,std(rawData),'lineprops','r');
ylim([0 (max(averageStepsPerEpisode))+50]);
if doDynaQ == true
    title(['Mean and Standard Deviation of Dyna-Q-Learning Performance Over '
num2str(numberOfTrials) ' Trials']);
elseif doPOMDP == true
    title(['Mean and Standard Deviation of Q-Learning for POMDP Performance Over '
num2str(numberOfTrials) ' Trials']);
elseif doSTM == true
    title(['Mean and Standard Deviation of Q-Learning for STM Performance Over '
num2str(numberOfTrials) ' Trials']);
else
    title(['Mean and Standard Deviation of Q-Learning Performance Over '
num2str(numberOfTrials) ' Trials']);
end
xlabel('Episode Number');
ylabel('Number of Steps');

pause(0.1)

%===========================================================================
.
.
.
```

File: initQ.m

Dependencies: [None]

Description: Initialise a Q-Table for Dyna-Q or regular Q-Learning

Inputs: min, max

Outputs: initQTable

```
function initQTable = initQ( min, max )

    %initialise 11x4 Q-Table
    initQTable = (max - min).*rand(11,4) + min;

end
```

File: trialTrainer.m

Dependencies: doQLearning.m

Description: Execute a trial of the selected Q-Learning algorithm

Inputs: QTable, numberOfEpisodes

Outputs: stepsPerEpisode, finalQTable

```
function [ stepsPerEpisode, finalQTable ] = trialTrainer( QTable, numberOfEpisodes )

    global episodeTrack;
    global episodeStepMeanSTD;
```

```
    y = 1;

    %create empty array to store the number of steps per episode
    stepsPerEpisode = zeros(1,numberOfEpisodes);

    %execute Q-Learning algorithm. This is done outside the later for-loop
    %so the original QTable is not altered. This is done for visualy
    %comparing the intitial and final Q-Table later
    [stepCounter,finalQTable] = doQLearning(QTable);

    %store number of steps of current episode
    stepsPerEpisode(1) = stepCounter;

    %store number of steps into the means and standard deviation matrix
    episodeStepMeanSTD(1,y) = episodeStepMeanSTD(1,y) + stepCounter;

    y = y + 1;

    for x = 2:numberOfEpisodes

        %execute Q-Learning algorithm
        [stepCounter,finalQTable] = doQLearning(finalQTable);

        %store number of steps of current episode
        stepsPerEpisode(x) = stepCounter;

        %check if the episode number is a member of episodeTrack. If it is
        %then store the number of steps for that episode
        if (ismember(x, episodeTrack))
            episodeStepMeanSTD(1,y) = episodeStepMeanSTD(1,y) + stepCounter;
            y = y + 1;
        end

    end

end
```

File: doQLearning.m

Dependencies: randomStartingState.m. findObservationIdx.m, findObservation.m, findSubsequentObservationIdx.m, eGreedySelection.m, nextState.m, tUpdate.m, rUpdate.m, nextObservation.m, QTableUpdate.m, QTableUpdateSTM.m. QTableUpdatePOMDP.m

Description: Execute an episode of the selected Q-Learning algorithm

Inputs: QTable

Outputs: numberOfSteps, finalQTable

```
function [ numberOfSteps, finalQTable ] = doQLearning( QTable )

    global doDynaQ;
    global doPOMDP;
    global goalState;
    global doSTM;
```

```
    %array of states
    state = [1,2,3,4,5,6,7,8,9,10,11];

    %array of actions
    action = [1,2,3,4];

    %number of steps taken to reach goal
    stepCounter = 0;

    %find a random starting state and its associated index
    startingState = randomStartingState(state);
    state_idx = find(state==startingState);

    if doPOMDP == true || doSTM ==true
        %find observation index for POMDPs
        observation_idx = findObservationIdx(state_idx);

        %find observation give the state index
        observation = findObservation(state_idx);
        %initial previous obersavation is same as current observation
        previous_observation = findObservation(state_idx);
        %find index of the combination of the previous and current observation
        currentSubsequentObservationIdx =
findSubsequentObservationIdx(observation,previous_observation);
    end

    if doDynaQ == true
        %create empty matrices for Dyna-Q algorithm
        TCount = zeros(11, 4, 11);
        TProbability = zeros(11, 4, 11);
        rewardMeans = zeros(11, 4);
        rewardCounts = zeros(11, 4);

        %loops for Dyna-Q modelling
        modelLoopCount = 10;
    end

    while state_idx ~= goalState

        if doPOMDP == true
            current_action = eGreedySelection(QTable,observation_idx,action);
        elseif doSTM == true
            current_action = eGreedySelection(QTable,currentSubsequentObservationIdx,action);
        else
            current_action = eGreedySelection(QTable,state_idx,action);
        end

        %index of the chosen action
        action_idx = find(action==current_action);

        %determine next state and reward
        [next_state, next_reward] =  nextState(state(state_idx),action(action_idx));

        %index of next state
        next_state_idx = find(state==next_state);

        if doDynaQ == true

            %update transfer function
            [TCount, TProbability] = tUpdate(TCount, TProbability, state_idx, action_idx,
next_state_idx);

            %update reward function
            [rewardMeans, rewardCounts] = rUpdate( rewardCounts, rewardMeans, state_idx,
action_idx, next_reward );

            for i=0:modelLoopCount
```

```
                % take random previously observed state and action previously
                % taken in s
                [state_find,~,~] = ind2sub( size(TCount),find(TCount > 0) );

                %generate a random state based on previously observed
                %states
                random_state = datasample(state_find,1);

                TCount_row_extracted = squeeze(TCount(random_state,:,:));

                %rows now are the actions, columns are the next states
                [action_find, next_state_find] = ind2sub(
size(TCount_row_extracted),find(TCount_row_extracted > 0) );

                %take random previously observed action in s
                random_action = datasample(action_find,1);

                random_action_idx = find(random_action == action_find);

                %generate next state given random action
                next_state = next_state_find(random_action_idx);

                %update transfer function based on randomly previously
                %observed state and action
                [TCount, TProbability] = tUpdate( TCount, TProbability, random_state,
random_action, next_state );

                %generate reward
                next_reward = rewardGen( random_state, random_action );

                %update reward function
                [rewardMeans, rewardCounts] = rUpdate( rewardCounts, rewardMeans,
random_state, random_action, next_reward );

                %update Q-Table
                QTable = QTableUpdate( QTable,  random_state, random_action, next_state,
next_reward );

            end


        elseif doPOMDP == true

            %find current observation and its associated index
            observation = findObservation(state_idx);
            observation_idx = findObservationIdx(state_idx);

            %determine the next observation and its associated index
            [~, next_reward] =
nextObservation(observation,state(state_idx),action(action_idx));
            next_observation_idx = findObservationIdx(next_state_idx);

            %update Q-Table accordingly
            QTable = QTableUpdatePOMDP(QTable, observation_idx, action_idx,
next_observation_idx, next_reward);


        elseif doSTM == true

            %find current observation
            observation = findObservation(state_idx);

            %calculate the next observation and reward based on current
            %observation, state and action
```

```
            [next_observation, next_reward] =
nextObservation(observation,state(state_idx),action(action_idx));

            %find the index of the current subsequent observations and next
            %subsequent observations
            currentSubsequentObservationIdx =
findSubsequentObservationIdx(observation,previous_observation);
            nextSubsequentObservationIdx =
findSubsequentObservationIdx(next_observation,observation);

            %update Q-Table accordinly
            QTable = QTableUpdateSTM(QTable, currentSubsequentObservationIdx,
nextSubsequentObservationIdx, action_idx, next_reward);

            %remember previous observation
            previous_observation = findObservation(state_idx);


        else

            %update Q-Table for regular Q-Learning
            QTable = QTableUpdate(QTable, state_idx, action_idx, next_state_idx, next_reward);

        end

        %new current state is assigned calculated next state
        state_idx = next_state_idx;

        %new current observation based on the new state
        observation_idx = findObservationIdx(state_idx);

        %increase step counter
        stepCounter = stepCounter + 1;


    end

    %return final Q-Table
    finalQTable = QTable;

    %return total number of steps for episode
    numberOfSteps = stepCounter;

end
```

File: randomStartingState.m

Dependencies: [None]

Description: Create a random starting state for an episode

Inputs: x

Outputs: startingState

```
function [ startingState ] = randomStartingState( x )

    %return random starting state
    startingState = randsample(x,1);

end
```

File: eGreedySelection.m

Dependencies: [None]

Description: Decides whether to take an action based on largest Q-Value or take a random action

Inputs: QTable, state_idx, action

Outputs: current_action

```
function [ current_action ] = eGreedySelection( QTable, state_idx, action )

    global explorationRate;

    %create random number between 0-1
    r = rand;

    %if r is more than exploration rate, take action based on largest
    %Q-Value of the given state
    if r>=explorationRate
        %exploitation
        [~,umax]=max(QTable(state_idx,:));
        current_action = action(umax);
    else %take random action
        %exploration
        current_action=datasample(action,1);
    end

end
```

File: nextState.m

Dependencies: [None]

Description: Calculates next state given input state and action. Also generates a reward.

Inputs: inputState, inputAction

Outputs: next_state, reward

```
function [ next_state, reward ] = nextState( inputState, inputAction )

    %calculate the next observation and reward depending on the given state and
    %action

    %needs to be assigned so if the state doesn't change then we sit in that state
    next_state = inputState;

    switch inputState

        case {1,2,3}
            if inputAction == 1
                next_state = inputState + 3;
            end
        case 4
            if inputAction == 1
                next_state = inputState + 3;
            elseif inputAction == 3
                next_state = inputState - 3;
            end
        case 5
            if inputAction == 1
                next_state = inputState + 4;
            elseif inputAction == 3
                next_state = inputState - 3;
            end
        case 6
            if inputAction == 1
                next_state = inputState + 5;
            elseif inputAction == 3
                next_state = inputState - 3;
            end
        case 7
            if inputAction == 2
                next_state = inputState + 1;
            elseif inputAction == 3
                next_state = inputState - 3;
            end
        case {8,10}
            if inputAction == 2
                next_state = inputState + 1;
            elseif inputAction == 4
                next_state = inputState - 1;
            end
        case 9
            if inputAction == 2
                next_state = inputState + 1;
            elseif inputAction == 3
                next_state = inputState - 4;
            elseif inputAction == 4
                next_state = inputState - 1;
            end
        case 11
            if inputAction == 3
                next_state = inputState - 5;
            elseif inputAction == 4
                next_state = inputState - 1;
```

```
            end

    end

    %Create a reward function that takes a state and an action and returns
    %10 if the state is 5 and the action is 3. In all other cases it should
    %return 0 >>
    if (inputState == 5) && (inputAction == 3)
            reward = 10;
        else
            reward = 0;
    end

    %When the following code is uncommented, it creates a new figure with a
    %dot moving around the McCallum's grid world.

    %{
    points = [0,0;2,0;4,0;0,1;2,1;4,1;0,2;1,2;2,2;3,2;4,2];
    minX = min(points(:,1)) - 0.5;
    maxX = max(points(:,1)) + 0.5;
    minY = min(points(:,2)) - 0.5;
    maxY = max(points(:,2)) + 0.5;

    plot(points(inputState,1), points(inputState, 2), 'b*', ...
    'MarkerSize', 10, 'LineWidth', 3);
    grid on;
    xlim([minX, maxX]);
    ylim([minY, maxY]);
    pause(0.00001);
    %}

end
```

File: QTableUpdate.m

Dependencies: [None]

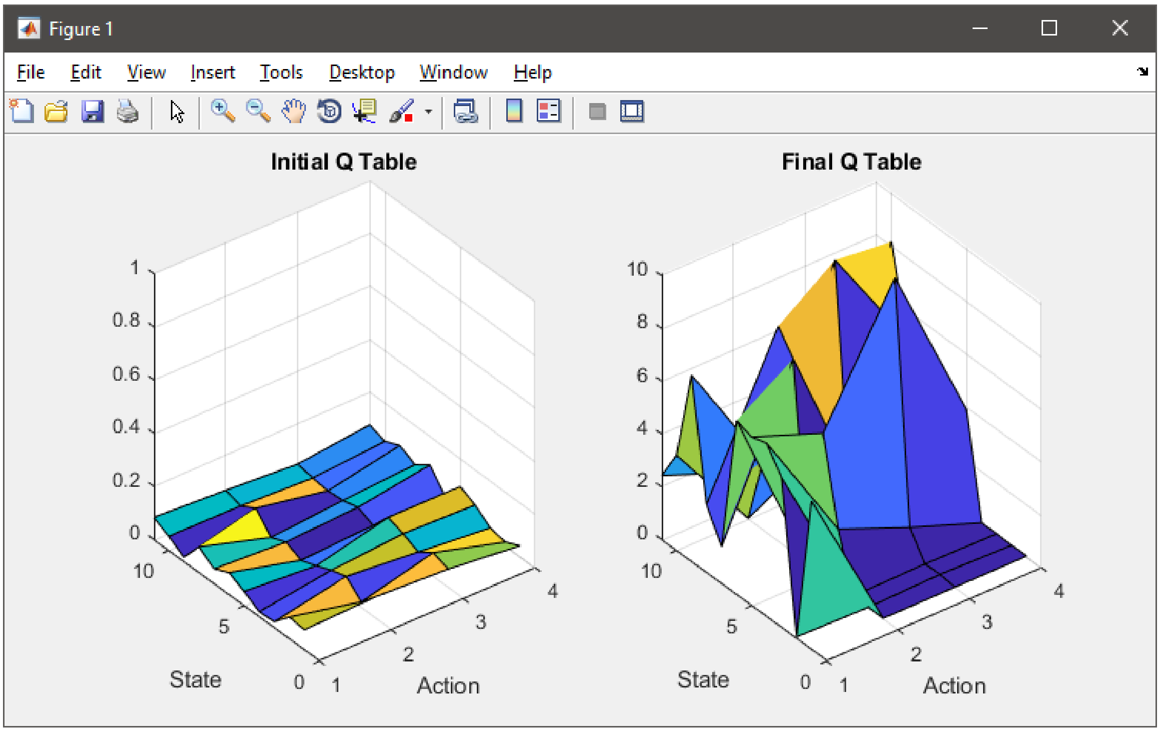
Description: Updates the Q-Table.

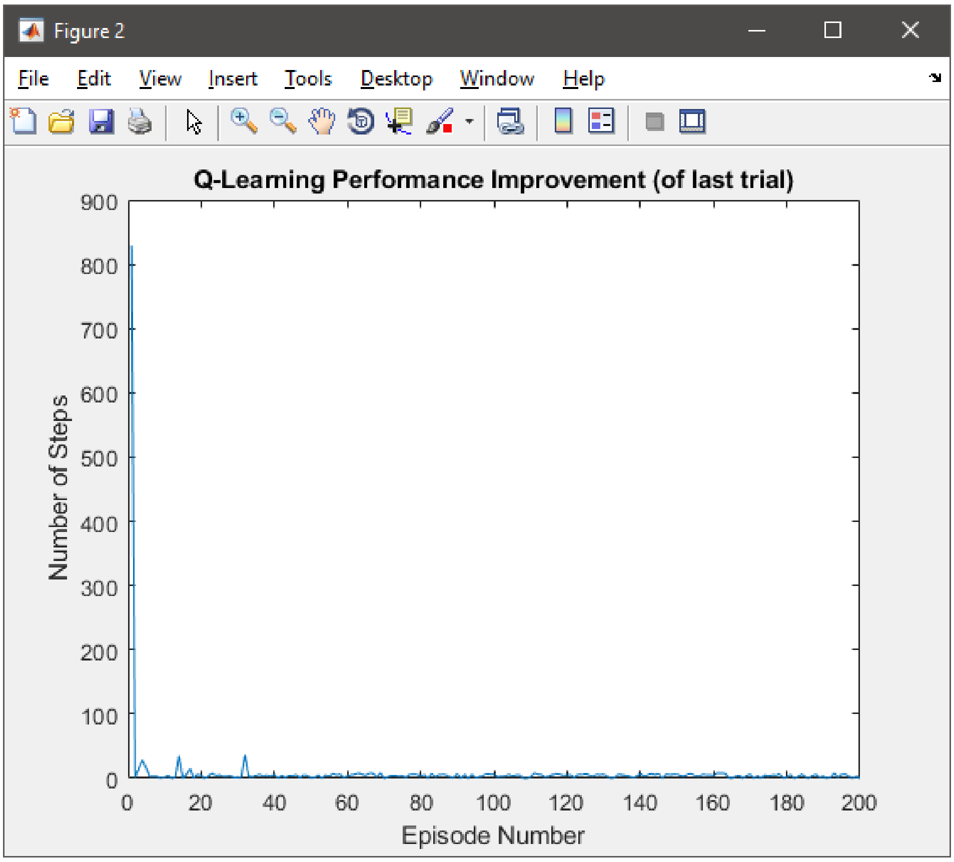
Inputs: QTable, state_idx, action_idx, next_state_idx, next_reward

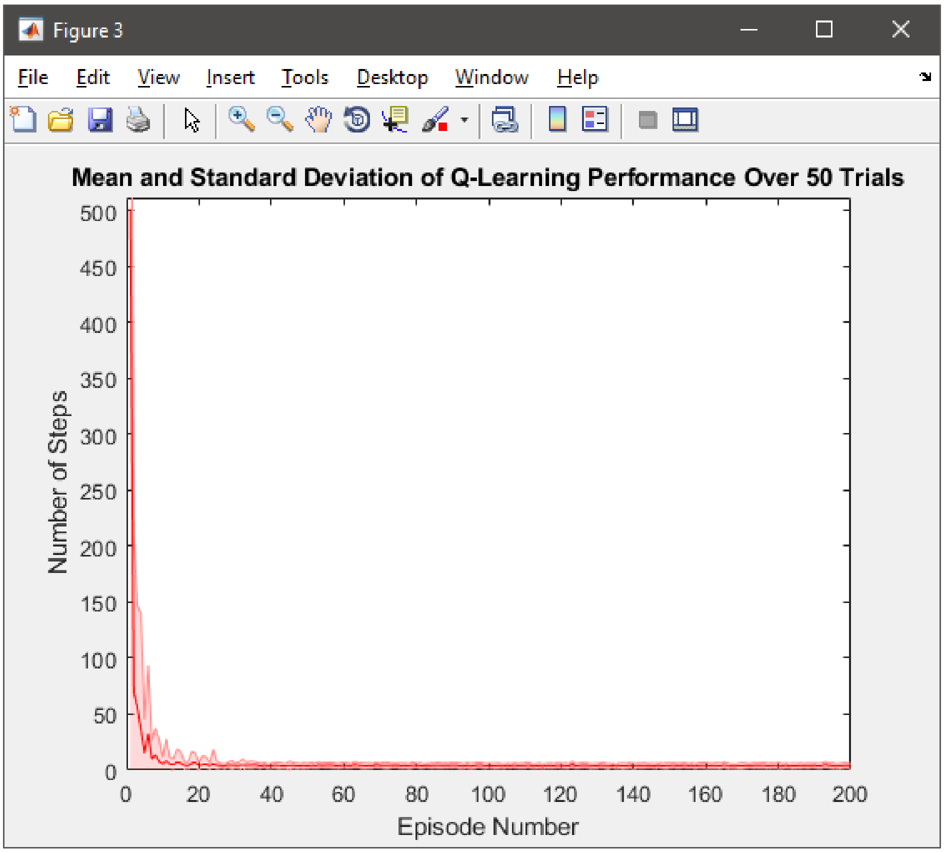
Outputs: QTable

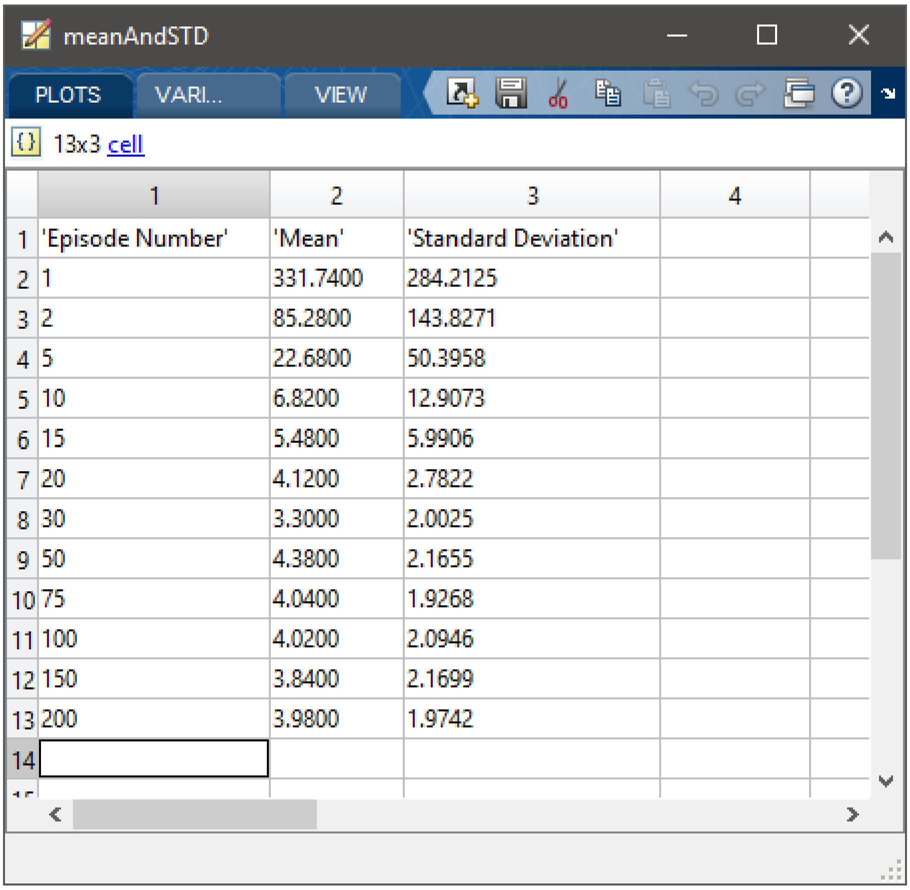
```
function [ QTable ] = QTableUpdate( QTable, state_idx, action_idx, next_state_idx, next_reward
)

    global temporalDiscountRate;
    global learningRate;

    QTable(state_idx,action_idx) = QTable(state_idx,action_idx) + learningRate * (next_reward
+ temporalDiscountRate* max(QTable(next_state_idx,:)) - QTable(state_idx,action_idx));

end
```

REGULAR Q-LEARNING OUTPUTS:








meanAndSTD

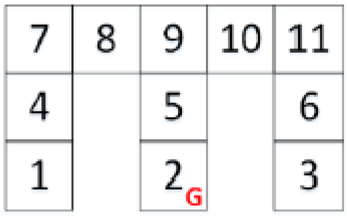
PLOTS VARI... VIEW

13x3 cell

	1	2	3	4
1	'Episode Number'	'Mean'	'Standard Deviation'	
2	1	331.7400	284.2125	
3	2	85.2800	143.8271	
4	5	22.6800	50.3958	
5	10	6.8200	12.9073	
6	15	5.4800	5.9906	
7	20	4.1200	2.7822	
8	30	3.3000	2.0025	
9	50	4.3800	2.1655	
10	75	4.0400	1.9268	
11	100	4.0200	2.0946	
12	150	3.8400	2.1699	
13	200	3.9800	1.9742	
14				

1. Dyna-Q for MDPs

Implement the Dyna-Q algorithm and the MDP version of McCallum's grid world as show in the following figure:



We will be using an exploration rate, ε, of 0.1, a temporal discount rate, γ, of 0.9 and a learning rate, α, of 0.2.

a)

EXTRACT FROM MAIN FILE: doQLearning.m:

```
if doDynaQ == true
        %create empty matrices for Dyna-Q algorithm
        TCount = zeros(11, 4, 11);
        TProbability = zeros(11, 4, 11);
        rewardMeans = zeros(11, 4);
        rewardCounts = zeros(11, 4);

        %loops for Dyna-Q modelling
        modelLoopCount = 10;
    end
```

.

.

.

```
if doDynaQ == true

            %update transfer function
            [TCount, TProbability] = tUpdate(TCount, TProbability, state_idx, action_idx,
next_state_idx);

            %update reward function
            [rewardMeans, rewardCounts] = rUpdate( rewardCounts, rewardMeans, state_idx,
action_idx, next_reward );

            for i=0:modelLoopCount

                % take random previously observed state and action previously
                % taken in s
                [state_find,~,~] = ind2sub( size(TCount),find(TCount > 0) );

                %generate a random state based on previously observed
                %states
                random_state = datasample(state_find,1);

                TCount_row_extracted = squeeze(TCount(random_state,:,:));

                %rows now are the actions, columns are the next states
```

```
                [action_find, next_state_find] = ind2sub(
size(TCount_row_extracted),find(TCount_row_extracted > 0) );

                %take random previously observed action in s
                random_action = datasample(action_find,1);

                random_action_idx = find(random_action == action_find);

                %generate next state given random action
                next_state = next_state_find(random_action_idx);

                %update transfer function based on randomly previously
                %observed state and action
                [TCount, TProbability] = tUpdate( TCount, TProbability, random_state,
random_action, next_state );

                %generate reward
                next_reward = rewardGen( random_state, random_action );

                %update reward function
                [rewardMeans, rewardCounts] = rUpdate( rewardCounts, rewardMeans,
random_state, random_action, next_reward );

                %update Q-Table
                QTable = QTableUpdate( QTable,  random_state, random_action, next_state,
next_reward );

            end
```

File: tUpdate.m

Dependencies: [None]

Description: Updates the transfer functions

Inputs: TCount, TProbability, state_idx, action_idx, next_state_idx

Outputs: TCount, TProbability

```
function [ TCount, TProbability ] = tUpdate( TCount, TProbability, state_idx, action_idx,
next_state_idx )

    % T matrix is the probablity that we go to next state based on input state
    % and action

    % initialise Tcount[] = 0.000001
    % Tcount[] is a 3d matrix
    % observe s,a,s'
    % increment T[s,a,s'] by adding one

    %increment by one in the position of [s,a,s']
    TCount(state_idx, action_idx, next_state_idx) = TCount(state_idx, action_idx,
next_state_idx) + 1;

    %create second table of probabilities based on TCount.
    %i.e probability of going to next state given action and state
    %add up values of next_state_idx given current state and action
    %divide current value in TCount by previous addition
```

```
    stepsSum = sum(TCount(state_idx, action_idx, :));

    TProbability(state_idx, action_idx, next_state_idx) = TCount(state_idx, action_idx,
next_state_idx) / stepsSum;

end
```

File: rewardGen.m

Dependencies: [None]

Description: Generates reward based on given state and action

Inputs: state, action

Outputs: reward

```
function [ reward ] = rewardGen( state, action )

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

end
```

File: rUpdate.m

Dependencies: [None]

Description: Updates the reward estimation functions

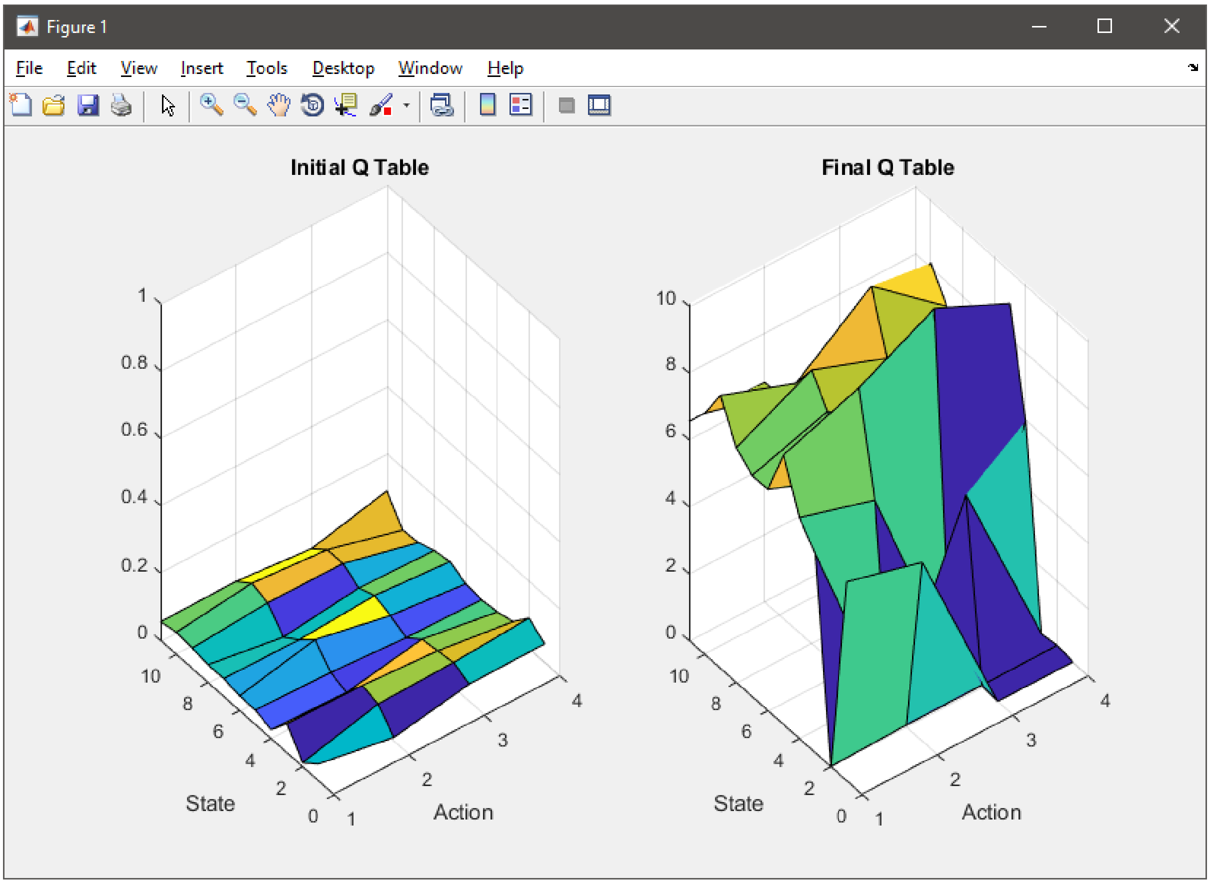
Inputs: rewardCounts, rewardMeans, state_idx, action_idx, next_reward

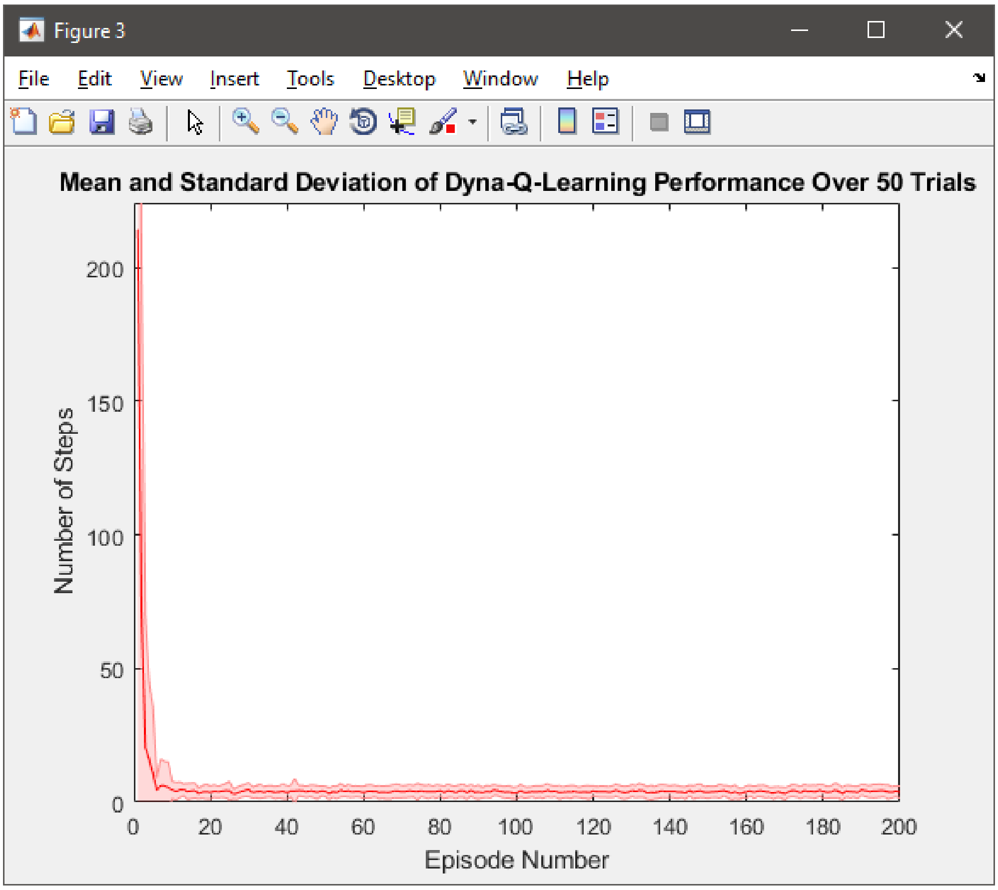
Outputs: rewardMeans, rewardCounts

```
function [ rewardMeans, rewardCounts ] = rUpdate( rewardCounts, rewardMeans, state_idx,
action_idx, next_reward )

    if next_reward ~= 0

        %add 1 to rewardCounts in corresponding location
        rewardCounts(state_idx, action_idx) = rewardCounts(state_idx, action_idx) + 1;

        %calculate the average reward of given state and action
        rewardMeans(state_idx, action_idx) = next_reward / rewardCounts(state_idx,
action_idx);
    end

end
```

b)

Now we will analyse the performance of the Dyna-Q algorithm on the MDP version of McCallum's grid world by running 50 trials of 200 episodes. A table containing the means and standard deviations of the number of steps required to complete and episode for the following episodes 1, 2, 5, 10, 15, 20, 30, 50, 75, 100, 150, and 200 will be provided. The means and standard deviations across all episodes will also be plotted, with the y-axis limited to the means of the first episode plus 10.





	1	2	3
1	'Episode Number'	'Mean'	'Standard Deviation'
2	1	265.0400	266.4214
3	2	40.2800	84.5475
4	5	11.2000	21.3513
5	10	3.6000	2.2678
6	15	5.3600	4.2125
7	20	4.3600	3.6798
8	30	4.6200	2.5144
9	50	3.8600	2.0703
10	75	3.4400	1.9183
11	100	4.3200	2.2080
12	150	3.8600	2.5476
13	200	3.8800	2.0765

Dyna-Q works very similar to regular Q-learning but has a lower initial number of steps for the first episode. This is due to looping using the model before taking a real step, so the algorithm learns faster. Using the model, the algorithm can reduce the number of steps taken to reach the goal faster than regular Q-learning. After about 20 episodes, both algorithms perform almost identical and reach the same number of average minimum steps taken to reach the goal.

c)

We will now compare the performances of Q-Learning to Dyna-Q learning on the MDP version of McCallum's grid world. This will be done using student-t test to evaluate whether the difference in means can be said to be significant for each of the episodes on part 1.b).

The following extract is the second part of the top-level file Q-Learning-Exercises.m that I have fully written myself.

SECOND PART OF Q_LEARNING_EXERCISES.m:

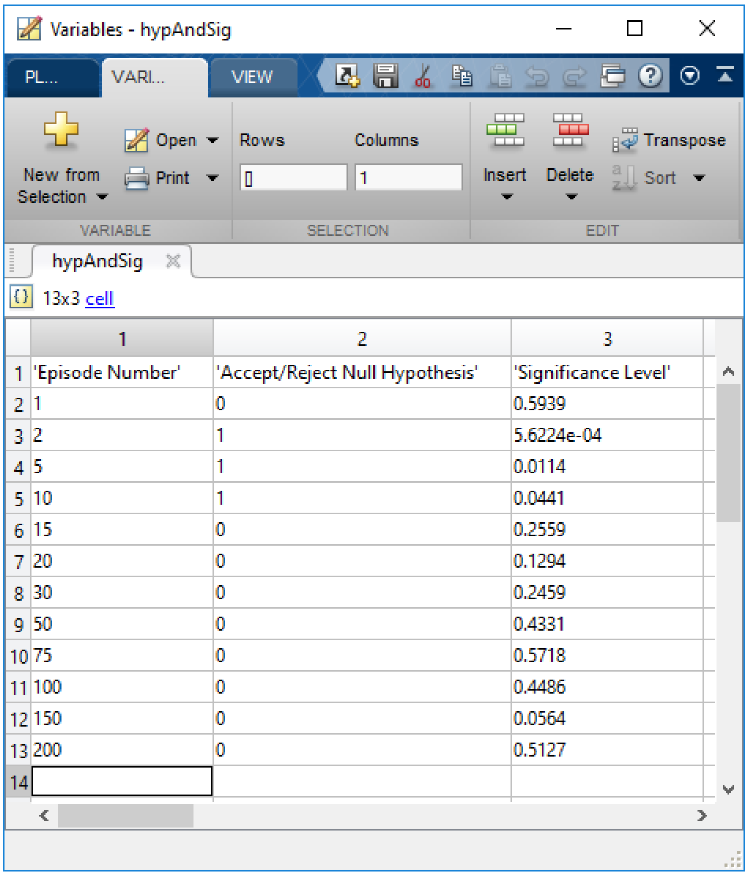
```
%==============================================================================
%SECONDARY Q-LEARNING FOR COMPARISION USING TTEST()
if doComparison == true

    %select which comparison to do. If all false compare algorithm 1 with
    %normal Q-Learning
    doDynaQ = true;
    doPOMDP = false;
    doSTM = false;

    %matrix to store means and standard deviations for the episodes on
    %episodeTrack
    %first row is means, second row is standard deviation
    episodeStepMeanSTD = zeros(2, size(episodeTrack,2));

    %array to store to total number of steps per episode of every trial. This
    %is used to calculate the average number of steps per episode
    totalStepsPerTrial = zeros(1,numberOfEpisodes);

    %matrix to store all steps for every episode for every trial
    rawData = zeros(numberOfTrials,numberOfEpisodes);

    for i=1:numberOfTrials

        disp('Trial Number');
        disp(i);

        %initialise the Q-Table for each trial. Q-Table is different size
        %depending on which algorithm is implemented
        if doDynaQ == true
            QTable = initQ(0.01,0.1);
        elseif doPOMDP == true
            QTable = initQPOMDP(0.01,0.1);
        elseif doSTM == true
            QTable = initQSTM(0.01,0.1);
        else    %normal Q-Learning
            QTable = initQ(0.01,0.1);
        end

        %begin a trial of the selected Q-Learning algorithm
        [stepsPerEpisode,finalQTable] = trialTrainer(QTable,numberOfEpisodes);

        %store the number of steps it took per episode
        totalStepsPerTrial = totalStepsPerTrial + stepsPerEpisode;

        %put the number of steps per episode into the rawData matrix
        rawData(i,:) = stepsPerEpisode;
```

```
        %store the mean number of steps for an episode in episodeTrack into the
        %rawData..Alg2 matrix
        rawDataForEpisodeTrackAlg2(i,:) = episodeStepMeanSTD(1,:);

        if (i ~= 1)
            for j=2:i
                %as episodeStepMeanSTD is accumulative, so the means can be
                %calculated, we need to find out the difference in step count
                %between two trials and adjust the matrix accordingly
                rawDataForEpisodeTrackAlg2(i,:) = rawDataForEpisodeTrackAlg2(i,:) -
rawDataForEpisodeTrackAlg2(j-1,:);
            end
        end

    end

    %calculate the average number of steps per episode
    averageStepsPerEpisode = totalStepsPerTrial / numberOfTrials;

    %calculate the means and standard deviations for the episodes in
    %episodeTrack
    episodeStepMeanSTD(1,:) = episodeStepMeanSTD(1,:) / numberOfTrials;
    episodeStepMeanSTD(2,:) = std(rawDataForEpisodeTrackAlg2);

    %test that the pairwise difference between the two Mean vectors have a mean equal to zero
    p = zeros(1,size(rawDataForEpisodeTrackAlg2,2));
    h = zeros(1,size(rawDataForEpisodeTrackAlg2,2));

    %create cell matrix to store H and P from the test algorithms
    hypAndSig = cell(3, size(rawDataForEpisodeTrackAlg2,2) + 1);
    cellRowTitles = {'Episode Number', 'Accept/Reject Null Hypothesis', 'Significance Level'};
    for i=1:3
        hypAndSig{i,1} = cellRowTitles{i};
    end
    for i=2:size(episodeTrack,2)+1
        hypAndSig{1,i} = episodeTrack(i-1);
    end

     %decide which test algorithm to use
    if doTTest == true
        %we want to look at the columns of rawDataForEpisodeTrack as our
        %samples and compare this to the rawDataForEpisodeTrack of the previous
        %algorithm
        for i=1:size(rawDataForEpisodeTrackAlg2,2)
            [h(i), p(i)] = ttest2(rawDataForEpisodeTrackAlg1(:,i),
rawDataForEpisodeTrackAlg2(:,i));
        end
    elseif doUTest == true
        for i=1:size(rawDataForEpisodeTrackAlg2,2)
            [p(i), h(i)] = ranksum(rawDataForEpisodeTrackAlg1(:,i),
rawDataForEpisodeTrackAlg2(:,i));
        end
    end

    %put the results from the test algorithm into the hypothesis and
    %significance cell matrix
    for i=2:size(episodeTrack,2)+1
        hypAndSig{2,i} = h(i-1);
    end
    for i=2:size(episodeTrack,2)+1
        hypAndSig{3,i} = p(i-1);
    end

    hypAndSig = hypAndSig';

    %plot the average number of steps per episode to see the improvement over
    %time
    figure;
    x=1:size(averageStepsPerEpisode,2);
    shadedErrorBar(x,averageStepsPerEpisode,std(rawData),'lineprops','r');
```

```
    ylim([0 (max(averageStepsPerEpisode))+50]);


    if doDynaQ == true
        title(['Mean and Standard Deviation of Dyna-Q-Learning Performance Over '
num2str(numberOfTrials) ' Trials']);
    elseif doPOMDP == true
        title(['Mean and Standard Deviation of Q-Learning for POMDP Performance Over '
num2str(numberOfTrials) ' Trials']);
    elseif doSTM == true
        title(['Mean and Standard Deviation of Q-Learning for STM Performance Over '
num2str(numberOfTrials) ' Trials']);
    else
        title(['Mean and Standard Deviation of Q-Learning Performance Over '
num2str(numberOfTrials) ' Trials']);
    end
    xlabel('Episode Number');
    ylabel('Number of Steps');

end
%============================================================================
```

ttest2(x,y) returns a test decision for the null hypothesis that the data in x-y comes from a normal distribution with mean equal to zero and unknown variance, using the paired-sample t-test.

Our null hypothesis is that “there is no significant difference between the means of the two data vectors at a 5% confidence level”. This means if the test returns a 1, we can reject it and say there is a significant difference between the two means. However, if the test returns a 0 we can say there is no significant difference between the two means at the 5% confidence level.

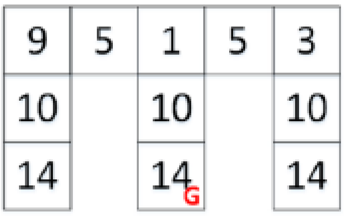


	1	2	3
1	'Episode Number'	'Accept/Reject Null Hypothesis'	'Significance Level'
2	1	0	0.5939
3	2	1	5.6224e-04
4	5	1	0.0114
5	10	1	0.0441
6	15	0	0.2559
7	20	0	0.1294
8	30	0	0.2459
9	50	0	0.4331
10	75	0	0.5718
11	100	0	0.4486
12	150	0	0.0564
13	200	0	0.5127
14			

The output of the ttest2() gives quite an expected result. At the early episodes we do expect there to be a large difference in the mean number of steps due to the variation in initial weights of the Q-Table and the initial starting position, but as the episodes increase, the number of steps for both algorithms decreases and stabilises so the difference between the two means becomes insignificant.

2. Q-Learning for POMDPs

I will now implement the POMDP version of McCallum's grid world by adding an observation function. The Q-Learning implementation will be changed so that each row in the Q-table represents an observation rather than a state.



a)

EXTRACT FROM MAIN FILE: doQLearning.m:

```
if doPOMDP == true || doSTM ==true
        %find observation index for POMDPs
        observation_idx = findObservationIdx(state_idx);

        %find observation give the state index
        observation = findObservation(state_idx);
        %initial previous obersavation is same as current observation
        previous_observation = findObservation(state_idx);
        %find index of the combination of the previous and current observation
        currentSubsequentObservationIdx =
findSubsequentObservationIdx(observation,previous_observation);
    end
.

.

.

while state_idx ~= goalState

        if doPOMDP == true
            current_action = eGreedySelection(QTable,observation_idx,action);
        elseif doSTM == true
            current_action = eGreedySelection(QTable,currentSubsequentObservationIdx,action);
        else
            current_action = eGreedySelection(QTable,state_idx,action);
        end

        %index of the chosen action
        action_idx = find(action==current_action);

        %determine next state and reward
        [next_state, next_reward] =  nextState(state(state_idx),action(action_idx));

        %index of next state
        next_state_idx = find(state==next_state);

.

.

.
```

```
        elseif doPOMDP == true

                %find current observation and its associated index
                observation = findObservation(state_idx);
                observation_idx = findObservationIdx(state_idx);

                %determine the next observation and its associated index
                [~, next_reward] =
        nextObservation(observation,state(state_idx),action(action_idx));
                next_observation_idx = findObservationIdx(next_state_idx);

                %update Q-Table accordingly
                QTable = QTableUpdatePOMDP(QTable, observation_idx, action_idx,
        next_observation_idx, next_reward);

        end


.

.

.


%new current state is assigned calculated next state
        state_idx = next_state_idx;

        %new current observation based on the new state
        observation_idx = findObservationIdx(state_idx);

        %increase step counter
        stepCounter = stepCounter + 1;


    end

    %return final Q-Table
    finalQTable = QTable;

    %return total number of steps for episode
    numberOfSteps = stepCounter;

end
```

File: initQPOMDP.m

Dependencies: [None]

Description: Initialises a Q-Table for POMDP. This is 6x4 as oppose to 12x4 as there are 6 different observations.

Inputs: min, max

Outputs: initQTable

```
function initQTable = initQPOMDP( min, max )

    %initialise 6x4 Q-Table
    initQTable = (max - min).*rand(6,4) + min;

end
```

File: QTableUpdatePOMDP.m

Dependencies: [None]

Description: Updates the Q-Table based on the observation and action

Inputs: QTable, observation_idx, action_idx, next_obersavation_idx, next_reward

Outputs: QTable

```
function [ QTable ] = QTableUpdatePOMDP( QTable, observation_idx, action_idx,
next_observation_idx, next_reward )

    global temporalDiscountRate;
    global learningRate;

    QTable(observation_idx,action_idx) = QTable(observation_idx,action_idx) + learningRate *
(next_reward + temporalDiscountRate* max(QTable(next_observation_idx,:)) -
QTable(observation_idx,action_idx));


end
```

File: findObservation.m

Dependencies: [None]

Description: Returns the observation based on the given state

Inputs: state_idx

Outputs: observation

```
function [ observation ] = findObservation( state_idx )

    observations = [14,14,14,10,10,10,9,5,1,5,3];

    %find current observation based on the state index
    observation = observations(state_idx);

end
```

File: findObservationIdx.m

Dependencies: [None]

Description: Returns the observation index based on the given state. The rows in the Q-Table now represent the observations so we need to find the index of the observation given the state. Row 1 is observation 14, 2 is 10, 3 is 9, 4 is 5, 5 is 1, 6 is 3.

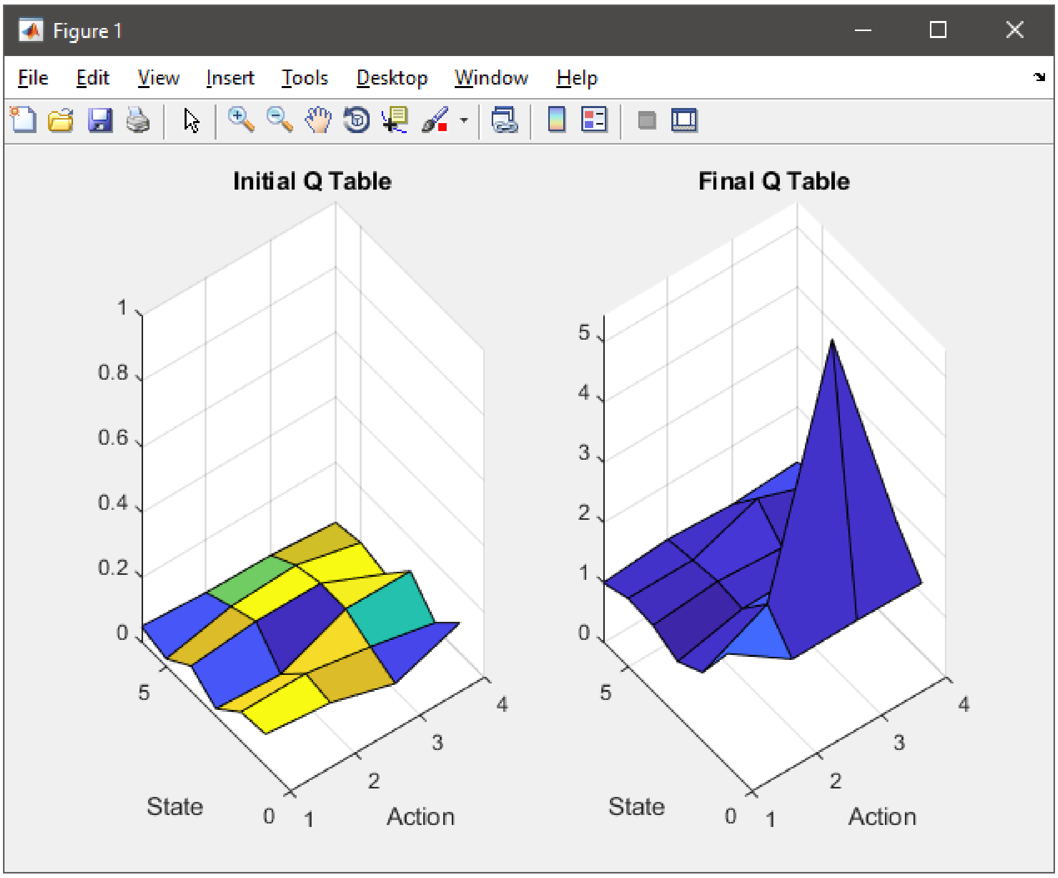
Inputs: state_idx

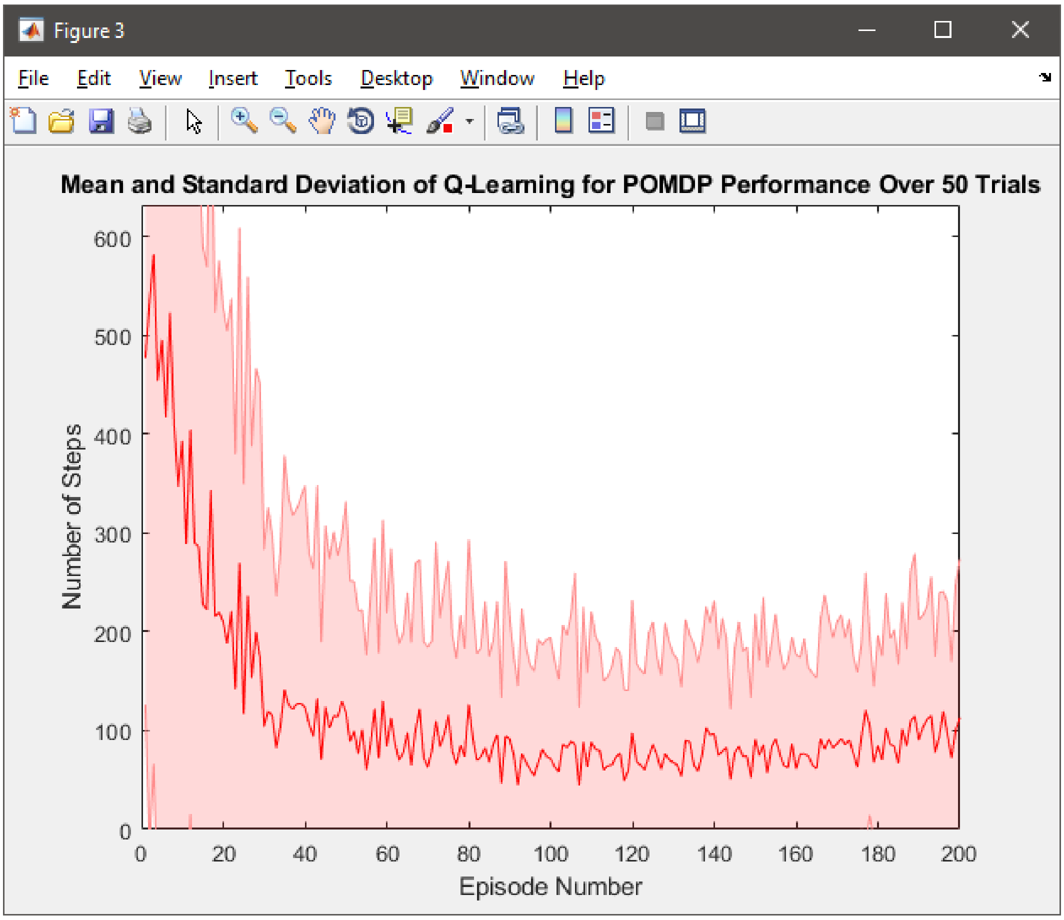
Outputs: observation_idx

```
function [ observation_idx ] = findObservationIdx( state_idx )

    %determine the observation index based on the state index
    if (1 <= state_idx)  && (state_idx <= 3)
        observation_idx = 1;
    elseif (4 <= state_idx)  && (state_idx <= 6)
        observation_idx = 2;
    elseif state_idx == 7
        observation_idx = 3;
    elseif state_idx == 8 || state_idx == 10
        observation_idx = 4;
    elseif state_idx == 9
        observation_idx = 5;
    elseif state_idx == 11
        observation_idx = 6;
    end

end
```

b)

Now we will analyse the performance of the POMDP algorithm on the MDP version of McCallum's grid world by running 50 trials of 200 episodes. A table containing the means and standard deviations of the number of steps required to complete and episode for the following episodes 1, 2, 5, 10, 15, 20, 30, 50, 75, 100, 150, and 200 will be provided. The means and standard deviations across all episodes will also be plotted, with the y-axis limited to the means of the first episode plus 50.





meanAndSTD

13x3 cell

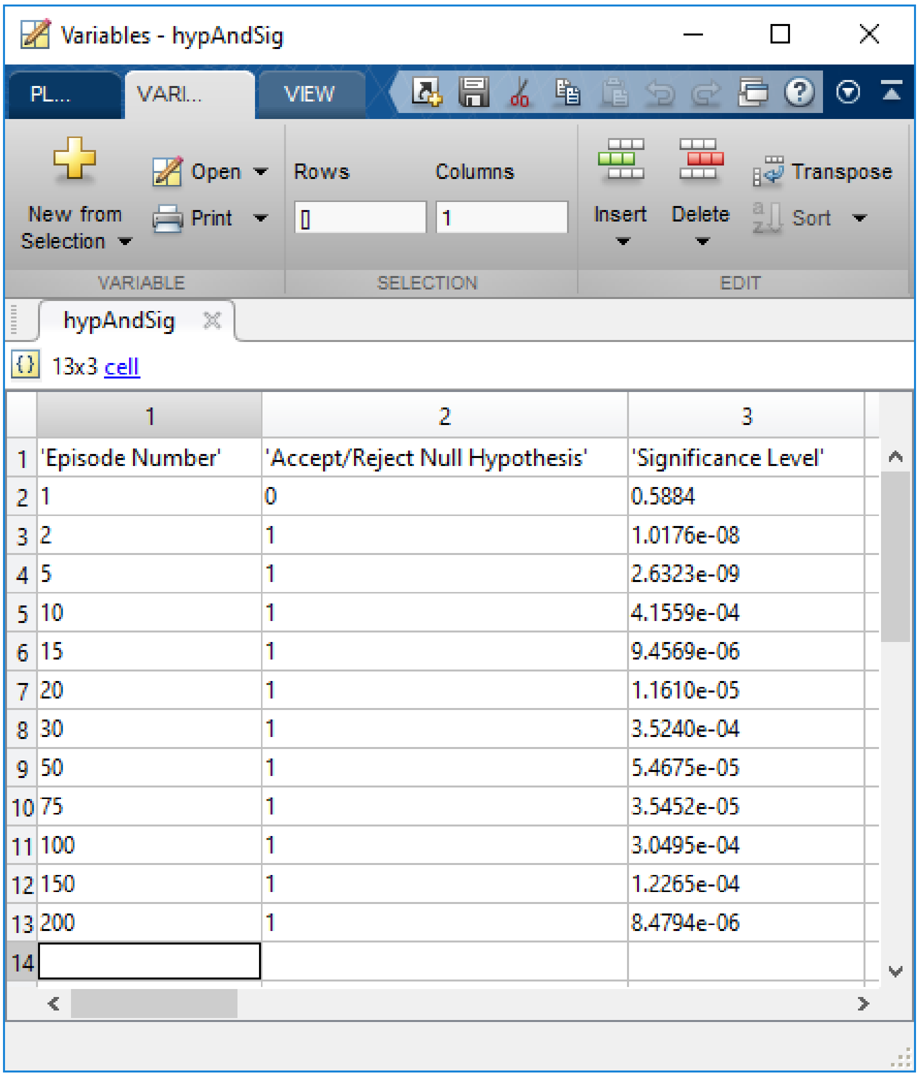
	1	2	3
1	'Episode Number'	'Mean'	'Standard Deviation'
2	1	369.6400	314.8610
3	2	388.8200	508.7232
4	5	413.5800	504.6825
5	10	345.1800	428.3166
6	15	223.9400	308.7580
7	20	202.7000	249.7113
8	30	116.9600	192.1356
9	50	76.3200	126.4174
10	75	70.6200	117.0215
11	100	82.5200	116.2219
12	150	109.1000	142.0732
13	200	88.8200	125.7855
14			

We expect the POMDP version of Q-Learning to be worse than regular Q-Learning due to the algorithm not knowing which state it is in, given the observation. Going from observation 10 to 14 is rewarding from only one state so the Q-Table will reflect this action, given the observation. This means when doing the eGreedySelection the algorithm is more likely to pick action 3 whenever it sees observation 10, but this is not rewarding for two states that have observation 10 so will get stuck in the corners and take longer to reach the goal.

The large standard deviation shows that the algorithm is not very consistent, and the number of steps between trials varies greatly. The algorithm also takes much longer to plateau, and doesn't reach a reasonably constant average number of steps until around episode 70, compared to around 20 for regular Q-learning.

c)

We will now compare the performances of Q-Learning to POMDP on the MDP version of McCallum's grid world. This will be done using student-t test to evaluate whether the difference in means can be said to be significant for each of the episodes on part 1.b). This is done using the second part of Q_Learning_Exercises.m, which can be found in part 1.c) of Dyna-Q learning.



	1	2	3
1	'Episode Number'	'Accept/Reject Null Hypothesis'	'Significance Level'
2	1	0	0.5884
3	2	1	1.0176e-08
4	5	1	2.6323e-09
5	10	1	4.1559e-04
6	15	1	9.4569e-06
7	20	1	1.1610e-05
8	30	1	3.5240e-04
9	50	1	5.4675e-05
10	75	1	3.5452e-05
11	100	1	3.0495e-04
12	150	1	1.2265e-04
13	200	1	8.4794e-06
14			

We expect the hypothesis to be rejected for most, if not all, of the episodes as POMDP is much slower than regular Q-Learning. The average number of steps taken for POMDP is a lot higher than Q-learning, so we expect there to be a significant difference in means for all episodes. We can see from the plot of means and standard deviation of POMDP that it differs greatly from the plot of regular Q-Learning. The accepting of the null hypothesis for episode 1 might be due randomness from the initial Q-Table causing the two algorithms to take a very similar number of steps to reach the goal.

3. Q-Learning with Short-Term Memory

We will now extend the POMDP version of the Q-Learning algorithm with a simple short-term memory (STM) so that it remembers one step back in time. The Q-Table will now represent all possible combinations of two subsequent observations.

a)

EXTRACT FROM MAIN FILE: doQLearning.m:

```
if doPOMDP == true || doSTM ==true
        %find observation index for POMDPs
        observation_idx = findObservationIdx(state_idx);

        %find observation give the state index
        observation = findObservation(state_idx);
        %initial previous obersavation is same as current observation
        previous_observation = findObservation(state_idx);
        %find index of the combination of the previous and current observation
        currentSubsequentObservationIdx =
findSubsequentObservationIdx(observation,previous_observation);
    end

.

.

.

while state_idx ~= goalState

        if doPOMDP == true
            current_action = eGreedySelection(QTable,observation_idx,action);
        elseif doSTM == true
            current_action = eGreedySelection(QTable,currentSubsequentObservationIdx,action);
        else
            current_action = eGreedySelection(QTable,state_idx,action);
        end

        %index of the chosen action
        action_idx = find(action==current_action);

        %determine next state and reward
        [next_state, next_reward] =  nextState(state(state_idx),action(action_idx));

        %index of next state
        next_state_idx = find(state==next_state);

.

.

.

        elseif doSTM == true

                    %find current observation
                    observation = findObservation(state_idx);

                    %calculate the next observation and reward based on current
                    %observation, state and action
                    [next_observation, next_reward] =
        nextObservation(observation,state(state_idx),action(action_idx));
```

```
                %find the index of the current subsequent observations and next
                %subsequent observations
                currentSubsequentObservationIdx =
findSubsequentObservationIdx(observation,previous_observation);
                nextSubsequentObservationIdx =
findSubsequentObservationIdx(next_observation,observation);

                %update Q-Table accordinly
                QTable = QTableUpdateSTM(QTable, currentSubsequentObservationIdx,
nextSubsequentObservationIdx, action_idx, next_reward);

                %remember previous observation
                previous_observation = findObservation(state_idx);



.

.

.

         %new current state is assigned calculated next state
         state_idx = next_state_idx;

         %new current observation based on the new state
         observation_idx = findObservationIdx(state_idx);

         %increase step counte
         stepCounter = stepCounter + 1;


        end

  %return final Q-Table
    finalQTable = QTable;

    %return total number of steps for episode
    numberOfSteps = stepCounter;

end
```

File: initQSTM.m

Dependencies: [None]

Description: Initialises a Q-Table for STM. This is a 20x4 as there are 20 possible subsequent observations.

Inputs: min, max

Outputs: initQTable

```
function [ initQTable ] = initQSTM( min, max )

    %{
    Q Table possible combinations:
    All states will have themselves as a previous observation, this starts us
    off with 12 combinations. All states connected to two other states each
    have two previous states, so there are 2x (states with two connected
    states).
    State with the '1' observation with also only have 2 previous state
    observations due to '5' being two of the three possibilities.

    QTable will have 1 row per possible subsequent observations.

    QTable =  SubsequentObservations =  [ 14  14;
                                          10  10;
                                          9   9;
                                          5   5;
                                          3   3;
                                          1   1;
                                          14  10;
                                          10  14;
                                          10  9;
                                          10  1;
                                          10  3;
                                          9   10;
                                          9   5;
                                          5   9;
                                          5   1;
                                          5   3;
                                          3   10;
                                          3   5;
                                          1   10;
                                          1   5
                                        ];

    %}

    %initialise 20x4 Q-Table
    initQTable = (max - min).*rand(20,4) + min;
end
```

File: findSubsequentObservationIdx.m

Dependencies: [None]

Description: Finds the index of two subsequent observations to index the Q-Table correctly

Inputs: current_observation, previous_observation

Outputs: SubsequentObservationIdx

```
function [ SubsequentObservationIdx ] = findSubsequentObservationIdx( current_observation,
previous_observation )

    %find the subsequent observation index based on the current and previous
    %observation
    X = [current_observation previous_observation];

    %all posibilities of the current and previous observations
    SubsequentObservations =  [ 14  14;
                10  10;
                9   9;
                5   5;
                3   3;
                1   1;
                14  10;
                10  14;
                10  9;
                10  1;
                10  3;
                9   10;
                9   5;
                5   9;
                5   1;
                5   3;
                3   10;
                3   5;
                1   10;
                1   5
              ];

    %find row index of X
    [~,SubsequentObservationIdx] = ismember(X,SubsequentObservations,'rows');

end
```

File: nextObservation.m

Dependencies: [None]

Description: Determines the next observation and reward based on given observation, state and action.

Inputs: inputObservation, inputState, inputAction

Outputs: next_observation, reward

```
function [ next_observation, reward ] = nextObservation( inputObservation, inputState, 
inputAction )

    %calculate the next observation and reward depending on the given state and
    %action

    %needs to be assigned so if the state doesn't change then we sit in that state
    next_observation = inputObservation;

    switch inputObservation

        case 1
            if (inputAction == 2) || (inputAction == 4)
                next_observation = inputObservation + 4;
            elseif inputAction == 3
                next_observation = inputObservation + 9;
            end
        case 3
            if inputAction == 3
                next_observation = inputObservation + 7;
            elseif inputAction == 4
                next_observation = inputObservation + 2;
            end
        case 5
            if inputState == 8
                if inputAction == 2
                    next_observation = inputObservation - 4;
                elseif inputAction == 4
                    next_observation = inputObservation + 4;
                end
            elseif inputState == 10
                if inputAction == 2
                    next_observation = inputObservation - 2;
                elseif inputAction == 4
                    next_observation = inputObservation - 4;
                end
            end
        case 9
            if inputAction == 2
                next_observation = inputObservation - 4;
            elseif inputAction == 3
                next_observation = inputObservation + 1;
            end
        case 10
            if inputAction == 3
                next_observation = inputObservation + 4;
            elseif inputAction == 1
                if inputState == 4
                   next_observation = inputObservation - 1;
                elseif inputState == 5
                   next_observation = inputObservation - 9;
                elseif inputState == 6
                   next_observation = inputObservation - 7;
                end
            end
```

```
        case 14
            if inputAction == 1
                next_observation = inputObservation - 4;
            end

    end

    %Create a reward function that takes a state and an action and returns
    %10 if the state is 5 and the action is 3. In all other cases it should
    %return 0 >>
    if (inputObservation == 10) && (inputAction == 3) && (inputState == 5)
            reward = 10;
        else
            reward = 0;
    end



end
```

File: QTableUpdateSTM.m

Dependencies: [None]

Description: Updates the Q-Table using the Subsequent observation index and the next subsequent observation index

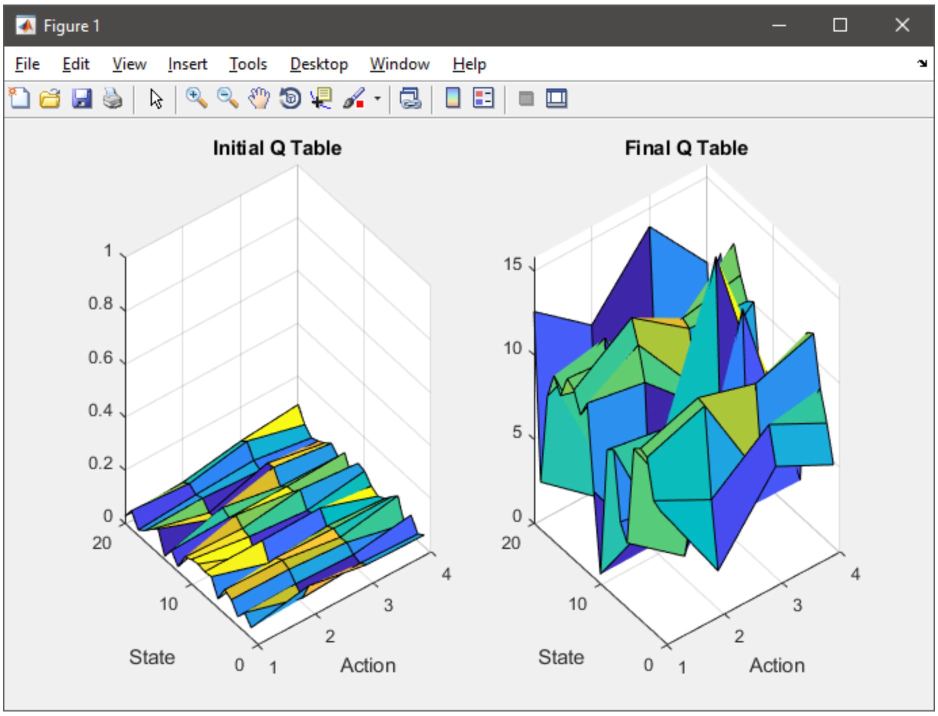
Inputs: QTable, SubsequentObservation_idx, nextSubsequentObservation_idx, action_idx, next_reward

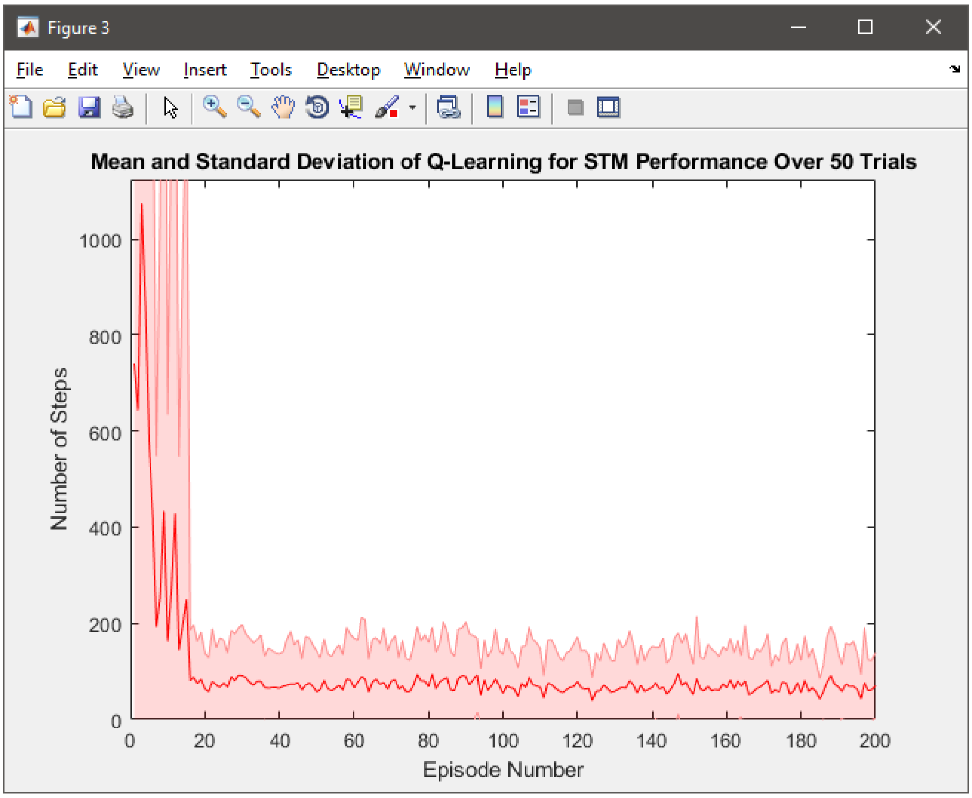
Outputs: QTable

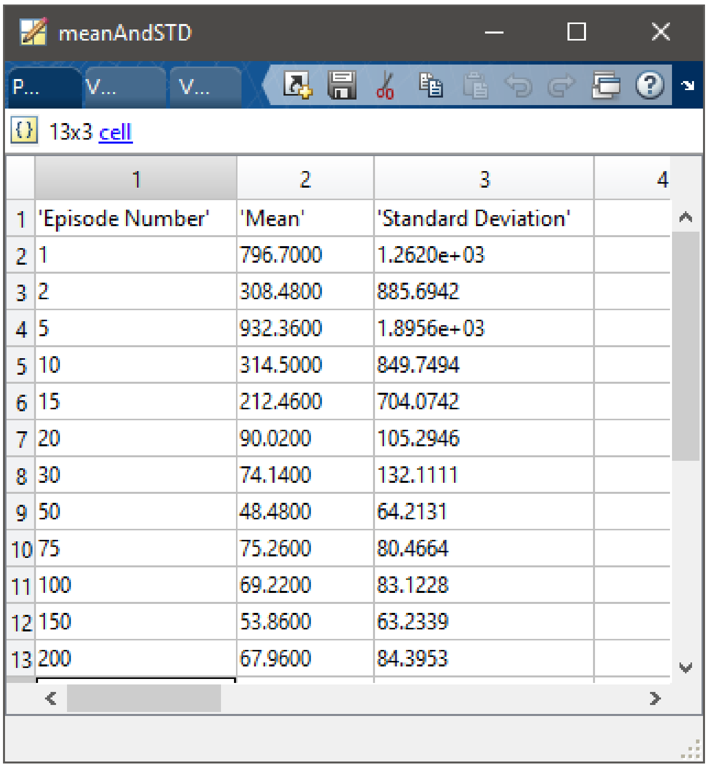
```
function [ QTable ] = QTableUpdateSTM( QTable, SubsequentObservation_idx,
nextSubsequentObservation_idx, action_idx, next_reward )

    global temporalDiscountRate;
    global learningRate;

    QTable(SubsequentObservation_idx,action_idx) =
QTable(SubsequentObservation_idx,action_idx) + learningRate * (next_reward +
temporalDiscountRate* max(QTable(nextSubsequentObservation_idx,:)) -
QTable(SubsequentObservation_idx,action_idx));


end
```

b)

Now we will analyse the performance of the STM algorithm on the MDP version of McCallum's grid world by running 50 trials of 200 episodes. A table containing the means and standard deviations of the number of steps required to complete and episode for the following episodes 1, 2, 5, 10, 15, 20, 30, 50, 75, 100, 150, and 200 will be provided. The means and standard deviations across all episodes will also be plotted, with the y-axis limited to the means of the first episode plus 50.







	1	2	3
1	'Episode Number'	'Mean'	'Standard Deviation'
2	1	796.7000	1.2620e+03
3	2	308.4800	885.6942
4	5	932.3600	1.8956e+03
5	10	314.5000	849.7494
6	15	212.4600	704.0742
7	20	90.0200	105.2946
8	30	74.1400	132.1111
9	50	48.4800	64.2131
10	75	75.2600	80.4664
11	100	69.2200	83.1228
12	150	53.8600	63.2339
13	200	67.9600	84.3953

We expect the STM version of Q-Learning to be worse than regular Q-Learning due to the algorithm not knowing which state it is in, given the observation, and the fact that the Q-Table is now much larger. Going from observation 10 to 14 is rewarding from only one state so the Q-Table will reflect this action, given the observation. This means when doing the eGreedySelection the algorithm is more likely to pick action 3 whenever it sees observation 10, but this is not rewarding for two states that have observation 10 so will get stuck in the corners and take longer to reach the goal.

The large standard deviation shows that the algorithm is not very consistent, and the number of steps between trials varies greatly. The algorithm, however, does plateau at roughly the same episode number as regular Q-Learning (episode 20).

c)

We will now compare the performances of Q-Learning to STM on the MDP version of McCallum's grid world. This will be done using student-t test to evaluate whether the difference in means can be said to be significant for each of the episodes on part 1.b). This is done using the second part of Q_Learning_Exercises.m, which can be found in part 1.c) of Dyna-Q learning.

Variables - hypAndSig

hypAndSig

13x3 cell

	1	2	3
1	'Episode Number'	'Accept/Reject Null Hypothesis'	'Significance Level'
2	1	1	0.0270
3	2	1	0.0389
4	5	1	0.0196
5	10	1	0.0026
6	15	1	0.0212
7	20	1	0.0320
8	30	1	7.7179e-05
9	50	1	2.0127e-06
10	75	1	9.1345e-07
11	100	1	6.5816e-08
12	150	1	5.2308e-08
13	200	1	6.0276e-08
14			
15			

We expect the hypothesis to be rejected for most, if not all, of the episodes as STM is much slower than regular Q-Learning. The average number of steps taken for STM is a lot higher than Q-learning, so we expect there to be a significant difference in means for all episodes. We can see from the plot of means and standard deviation of STM that it differs greatly from the plot of regular Q-Learning.

4. Non-symmetric Metrics

Reinforced-learning measured across multiple trials are typically not normally distributed. I shall now plot the quartiles of the performance values from sections 1.b), 2.b) and 3.b), instead of the means and standard deviations.

a)

The code that I have written executes another 50 trials of the selected Q-Learning algorithm and calculates the performance values of these new trials as oppose to calculating them on the original 50 trials of the selected algorithm.

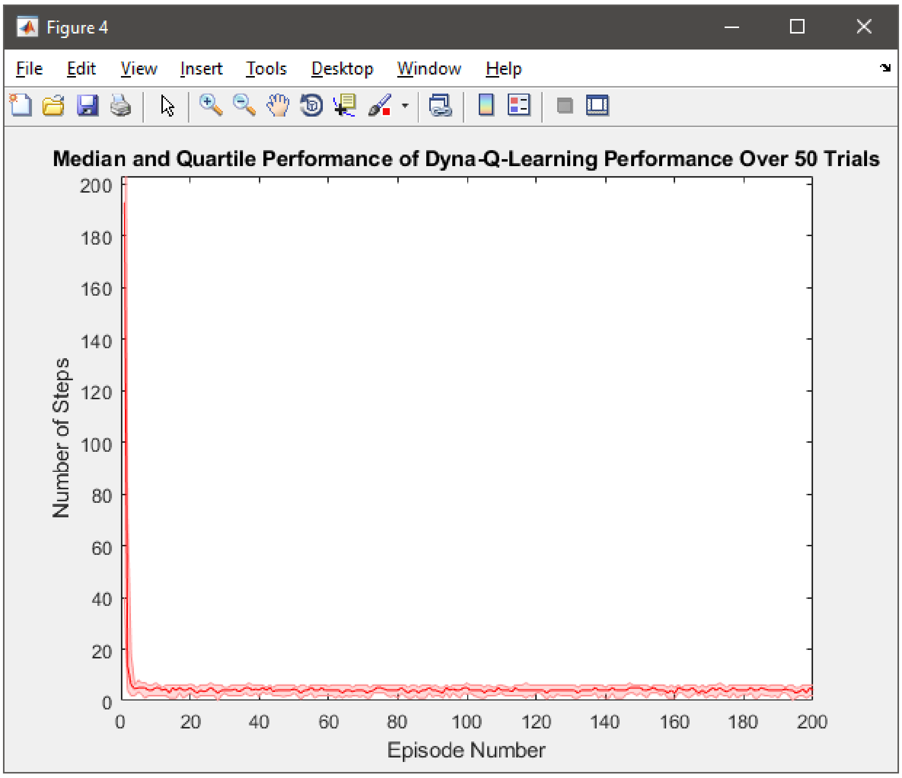
THIRD PART OF Q_LEARNING_EXERCISES.m:

```
%==============================================================================
%QUARTILE PERFORMANCE PLOT
if doQuarPerf == true

    %select which performance plot to do. If all false then plot normal
    %Q-Learning
    doDynaQ = true;
    doPOMDP = false;
    doSTM = false;

    %matrix to store means and standard deviations for the episodes on
    %episodeTrack
    %first row is means, second row is standard deviation
    episodeStepMeanSTD = zeros(2, size(episodeTrack,2));

    %array to store to total number of steps per episode of every trial. This
    %is used to calculate the average number of steps per episode
    totalStepsPerTrial = zeros(1,numberOfEpisodes);

    %matrix to store all steps for every episode for every trial
    rawData = zeros(numberOfTrials,numberOfEpisodes);

    for i=1:numberOfTrials

        disp('Trial Number');
        disp(i);

        %initialise the Q-Table for each trial. Q-Table is different size
        %depending on which algorithm is implemented
        if doDynaQ == true
            QTable = initQ(0.01,0.1);
        elseif doPOMDP == true
            QTable = initQPOMDP(0.01,0.1);
        elseif doSTM == true
            QTable = initQSTM(0.01,0.1);
        else    %normal Q-Learning
            QTable = initQ(0.01,0.1);
        end

        %begin a trial of the selected Q-Learning algorithm
        [stepsPerEpisode,finalQTable] = trialTrainer(QTable,numberOfEpisodes);

        %store the number of steps it took per episode
        totalStepsPerTrial = totalStepsPerTrial + stepsPerEpisode;

        %put the number of steps per episode into the rawData matrix
        rawData(i,:) = stepsPerEpisode;

        %store the mean number of steps for an episode in episodeTrack into the
```

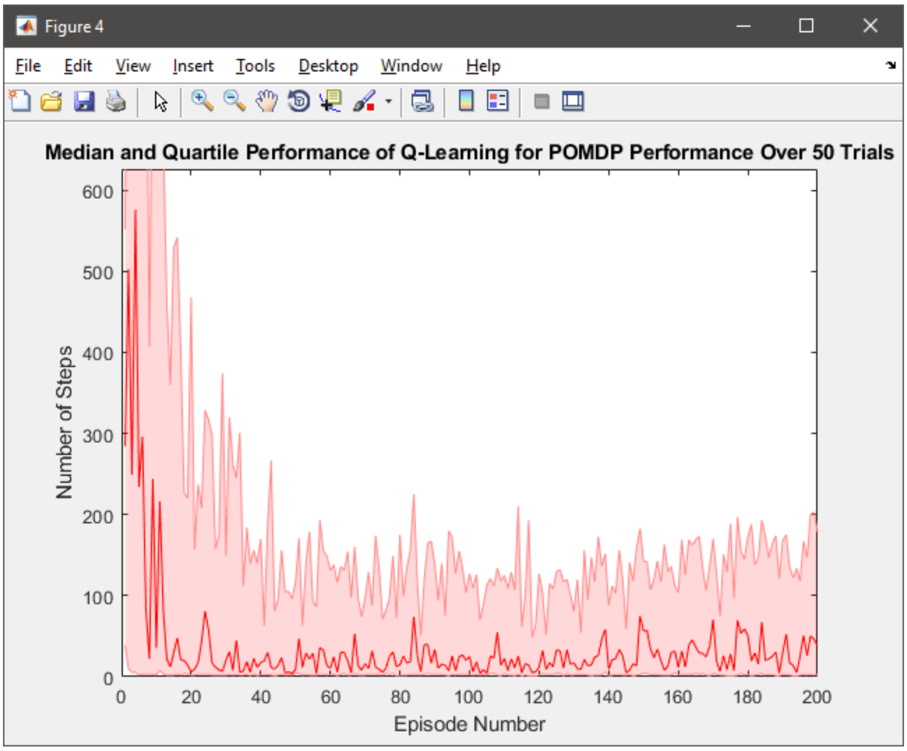
```
        %rawData..Alg2 matrix
        rawDataForEpisodeTrackAlg3(i,:) = episodeStepMeanSTD(1,:);

        if (i ~= 1)
            for j=2:i
                %as episodeStepMeanSTD is accumulative, so the means can be
                %calculated, we need to find out the difference in step count
                %between two trials and adjust the matrix accordingly
                rawDataForEpisodeTrackAlg3(i,:) = rawDataForEpisodeTrackAlg3(i,:) -
rawDataForEpisodeTrackAlg3(j-1,:);
            end
        end

    end

    %calculate the average number of steps per episode
    averageStepsPerEpisode = totalStepsPerTrial / numberOfTrials;

    %calculate the means and standard deviations for the episodes in
    %episodeTrack
    episodeStepMeanSTD(1,:) = episodeStepMeanSTD(1,:) / numberOfTrials;
    episodeStepMeanSTD(2,:) = std(rawDataForEpisodeTrackAlg3);

    %calculate the median steps per episode of all trials
    medianStepsPerEpisode = median(rawData);

    %create empty matrix to store the upper and lower quartile performances
    quartilePerformance = zeros(2,size(rawData,2));

    for i=1:size(rawData,2)
        %sort data by size
        sortedData = sort(rawData(:,i));
        % compute 25th percentile (first quartile). Row 2 for
        % shadedErrorBar needs to be lower bar
        quartilePerformance(2,i) = median(sortedData(find(sortedData<median(sortedData))));
        % compute 75th percentile (third quartile). Row 1 for
        % shadedErrorBar needs to be upper bar
        quartilePerformance(1,i) = median(sortedData(find(sortedData>median(sortedData))));
    end

    %if I plot the quartilePerformance without the following algorithm then
    %it will plot the quartiles above and below the median. e.g median = 5.
    %lower quartile = 2, upper quartile = 6. shadedErrorBar would then plot
    %5+6=11 as the upper quartile, and 5-2=3 as the lower quartile, which is
    %incorrect. We want to actually plot 2 and 6, not 2 below and 6 above
    %the median.
    quartilePerformance(1,:) = quartilePerformance(1,:) - medianStepsPerEpisode;
    quartilePerformance(2,:) = medianStepsPerEpisode - quartilePerformance(2,:);

    %plot the median number of steps per episode to see the improvement over
    %time, this time with the shaded bar being the upper and lower quartile
    figure;
    x=1:size(medianStepsPerEpisode,2);
    shadedErrorBar(x,medianStepsPerEpisode,quartilePerformance,'lineprops','r');
    ylim([0 max(medianStepsPerEpisode)+50]);
    if doDynaQ == true
        title(['Median and Quartile Performance of Dyna-Q-Learning Performance Over '
num2str(numberOfTrials) ' Trials']);
    elseif doPOMDP == true
        title(['Median and Quartile Performance of Q-Learning for POMDP Performance Over '
num2str(numberOfTrials) ' Trials']);
    elseif doSTM == true
        title(['Median and Quartile Performance of Q-Learning for STM Performance Over '
num2str(numberOfTrials) ' Trials']);
    else
        title(['Median and Quartile Performance of Q-Learning Performance Over '
num2str(numberOfTrials) ' Trials']);
    end
    xlabel('Episode Number');
    ylabel('Number of Steps');
```

```
end
%==========================================================================
```

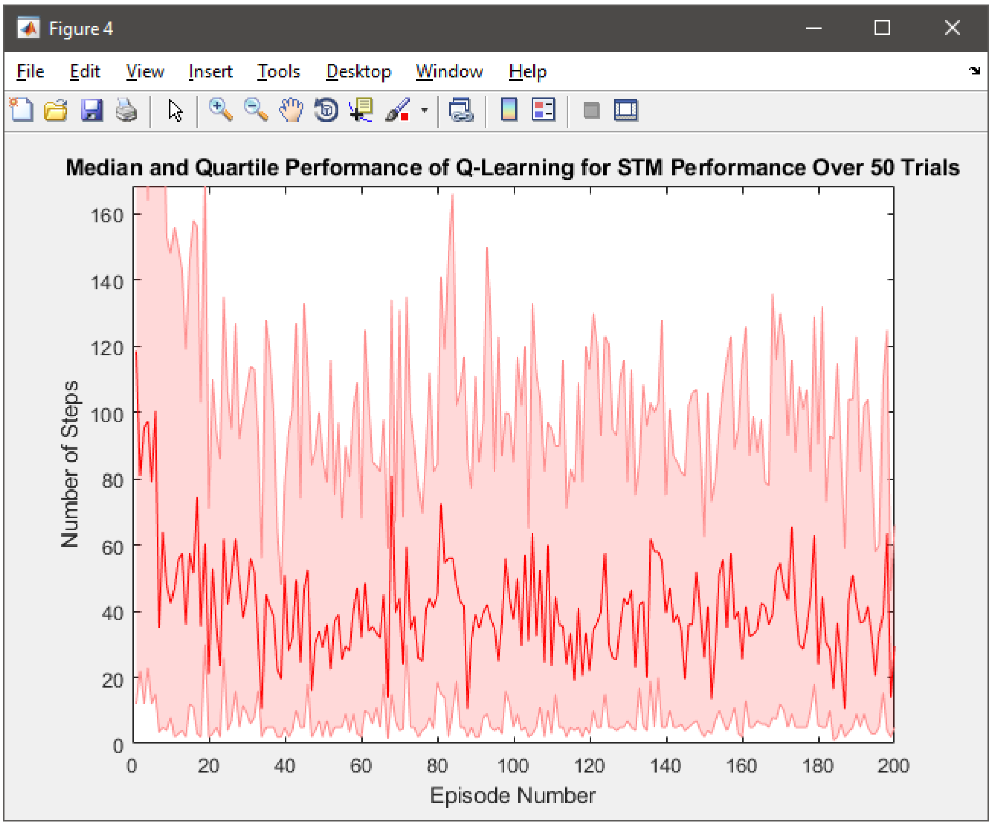
MEDIAN AND QUARTILE PERFORMANCE FOR DYNA-Q:



MEDIAN AND QUARTILE PERFORMANCE FOR POMDP:



MEDIAN AND QUARTILE PERFORMANCE FOR STM:



The two graphs show different information because one is plotting the means and the other plotting the median. Plotting the median is going to be more accurate in representing the average number of steps as single outliers will not greatly affect the median. Outliers will, however, greatly affect the mean.

I have written a small test script to show this:

```
data = [ 1 1 1 2 2 2 3 3 3 5 40];

meanData = mean(data)
stdData = std(data)

medianData = median(data)

sortedData = sort(data);

lowerquartilePerformance = median(sortedData(find(sortedData<median(sortedData))))

upperquartilePerformance = median(sortedData(find(sortedData>median(sortedData))))
```

With the output of:

```
meanData =

  5.7273

stdData =
```

```
  11.4288

medianData =

   2

lowerquartilePerformance =

   1

upperquartilePerformance =

   3
```

We can see that the mean is 5.73. This is quite high when looking at the dataset, and this is due to the outlier of 40. The median, however, is 2, and this better represents the dataset.

Plotting the median and quartile performance doesn't show us how greatly the data varies, whilst plotting the standard deviation shows us how much our average number of steps varies for a given episode.

b)

I will now redo the comparisons in sections 1.c), 2.c) and 3.c) using a metric test that is robust to non-symmetric distributions, and for this I will use the Mann-Whitney U Test. This is implemented by using the ranksum() function. Apart from Dyna-Q, we expect to reject the null hypothesis for most, if not all, episodes.

The Mann–Whitney U test remains the logical choice when the data are ordinal but not interval scaled, so that the spacing between adjacent values cannot be assumed to be constant. As it compares the sums of ranks, the Mann–Whitney U test is less likely than the t-test to randomly indicate significance because of the presence of outliers. Therefore Mann–Whitney U test is more robust.

EXTRACT FROM Q_LEARNING_EXERCISES.m PART 2:

```
%decide which test algorithm to use
    if doTTest == true
        %we want to look at the columns of rawDataForEpisodeTrack as our
        %samples and compare this to the rawDataForEpisodeTrack of the previous
        %algorithm
        for i=1:size(rawDataForEpisodeTrackAlg2,2)
            [h(i), p(i)] = ttest2(rawDataForEpisodeTrackAlg1(:,i),
rawDataForEpisodeTrackAlg2(:,i));
        end
    elseif doUTest == true
        for i=1:size(rawDataForEpisodeTrackAlg2,2)
            [p(i), h(i)] = ranksum(rawDataForEpisodeTrackAlg1(:,i),
rawDataForEpisodeTrackAlg2(:,i));
        end
    end
```

.

First, we will compare Q-Learning to Dyna-Q Learning:

Variables - hypAndSig

hypAndSig

13x3 cell

	1	2	3
1	'Episode Number'	'Accept/Reject Null Hypothesis'	'Significance Level'
2	1	0	0.2275
3	2	0	0.3386
4	5	0	0.1417
5	10	0	0.0573
6	15	0	0.4679
7	20	0	0.4772
8	30	0	0.6062
9	50	1	0.0134
10	75	0	0.3825
11	100	0	0.7431
12	150	0	0.0517
13	200	0	0.9085
14			
15			

Next, we will compare Q-Learning to the POMDP algorithm:

Variables - hypAndSig

hypAndSig

13x3 cell

	1	2	3
1	'Episode Number'	'Accept/Reject Null Hypothesis'	'Significance Level'
2	1	0	0.7225
3	2	1	0.0171
4	5	1	8.9503e-06
5	10	1	0.0114
6	15	1	8.2216e-05
7	20	1	2.0267e-06
8	30	1	0.0014
9	50	1	0.0021
10	75	1	4.2767e-05
11	100	1	6.1078e-05
12	150	1	0.0071
13	200	1	9.6473e-04
14			
15			

And finally, we will compare Q-Learning to Q-Learning with STM:

	1	2	3
1	'Episode Number'	'Accept/Reject Null Hypothesis'	'Significance Level'
2	1	0	0.2509
3	2	1	0.0109
4	5	1	4.9949e-04
5	10	1	1.3774e-05
6	15	1	6.4871e-06
7	20	1	5.4365e-07
8	30	1	0.0016
9	50	1	1.5070e-06
10	75	1	5.0223e-06
11	100	1	2.8219e-07
12	150	1	1.6344e-06
13	200	1	6.1666e-09

Again, we expect to reject the null hypothesis for most of the episodes of the POMDP and STM versions of Q-Learning. This is due to both algorithms being much slower than regular Q-Learning and taking significantly more steps to reach the goal.