Part 1

Linear Regression with One Variable

Task 1 - Modifying the Hypothesis in Calculate_hypothesis and Gradient Descent

- In calculate_hypotheis.m – the hypothesis is updated to include one training example along with the bias term where X(training_example, 1) and theta(1) correspond to X_0 and theta_0 respectively. Whilst, X(training_example, 2) and theta(2) correspond to X_1 and theta_1 respectively.

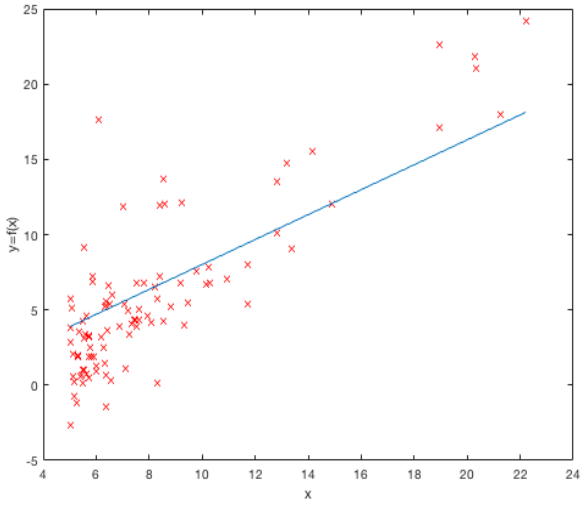
```
function hypothesis = calculate_hypothesis(X, theta, training_example)
    %CALCULATE_HYPOTHESIS This calculates the hypothesis for a given X,
    %theta and specified training example

    %hypothesis = 0:0;

hypothesis = X(training_example, 1) * theta(1) + X(training_example, 2) * theta(2);

end
```

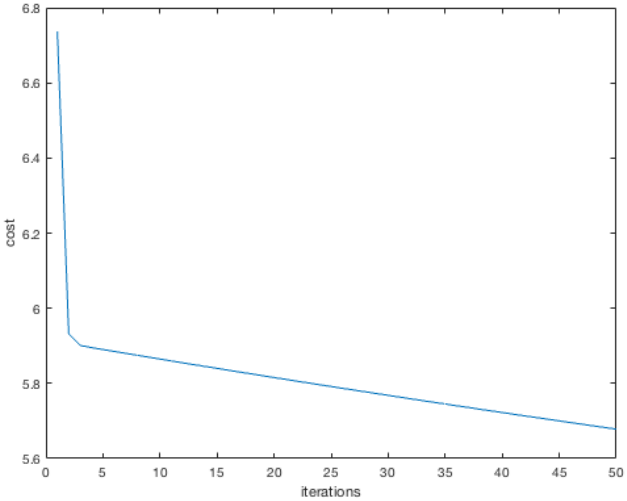
- In gradient_descent.m – the hypothesis is updated for theta(1) and theta(2) which correspong to theta_0 and theta_1 respectively.

```
%hypothesis = X(i, 1) * theta(1) + X(i, 2) * theta(2);
hypothesis = calculate_hypothesis(X, theta, i);
```

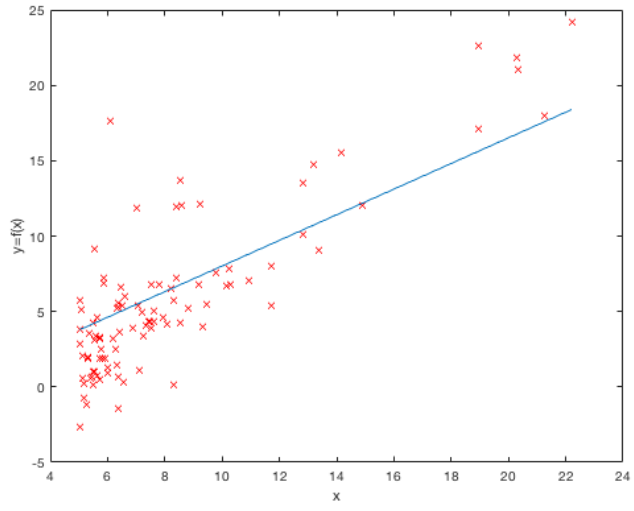
After modifying the code, mllab1.m is run again for high and low alpha values. The default alpha value which is given is 0.01. The gradient descent and cost function for the different alphas are depicted below.

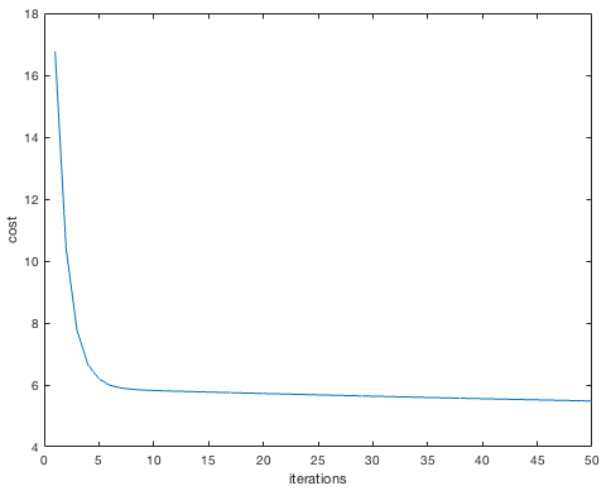
For Alpha – 0.01 (given)– the gradient descent and the cost function



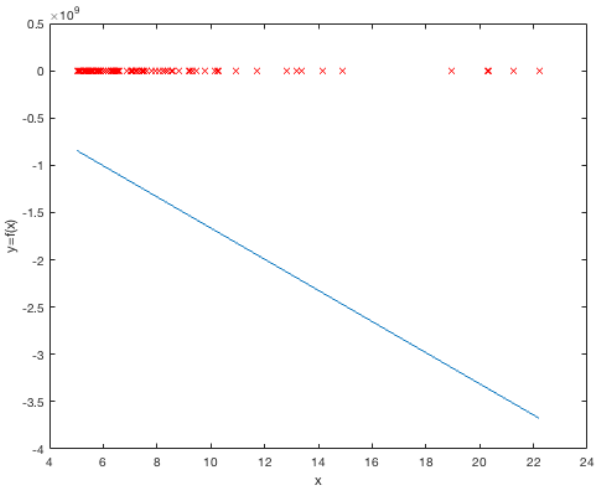


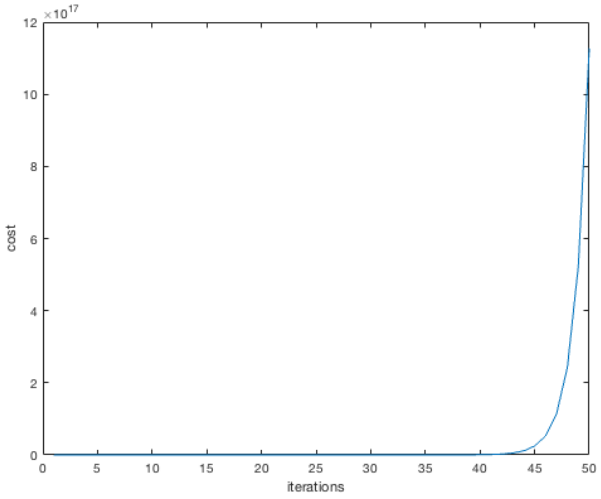
For Alpha – 0.02– the gradient descent and the cost function



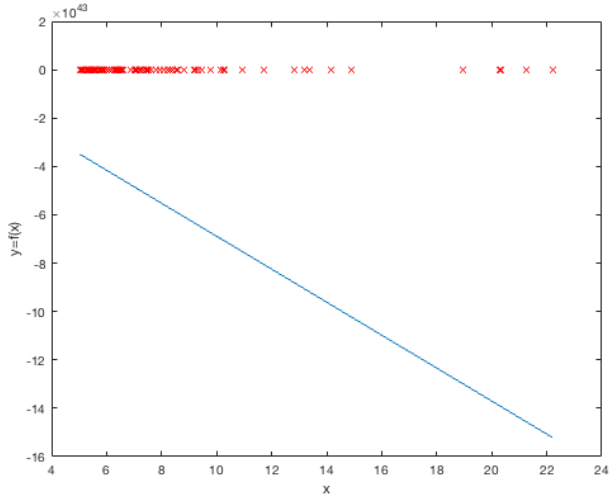


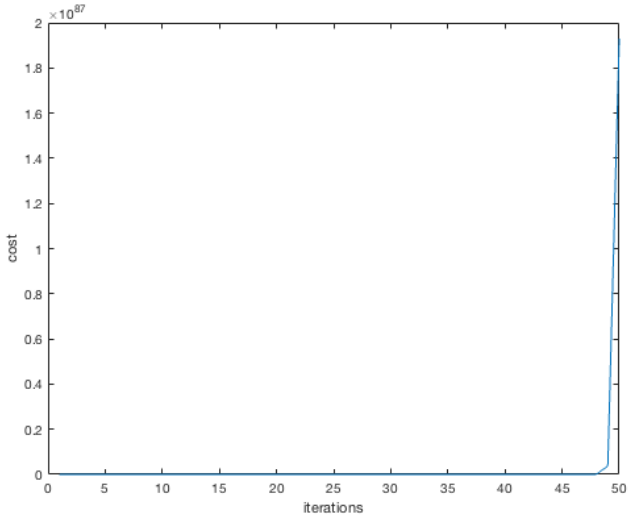
For Alpha – 0.03 – the gradient descent and the cost function



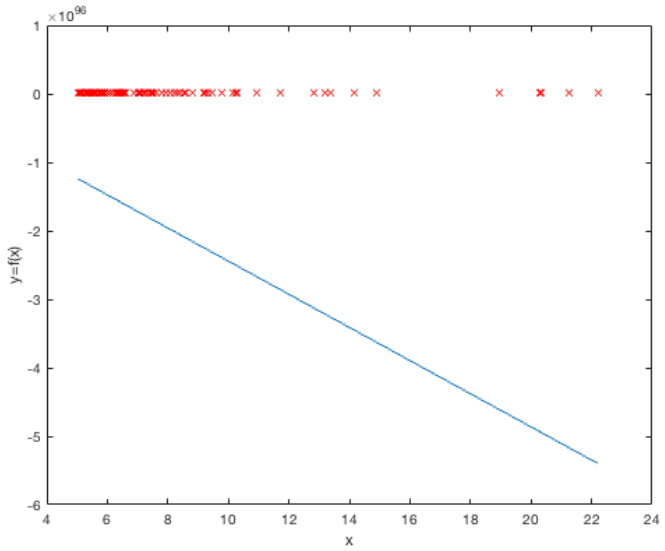


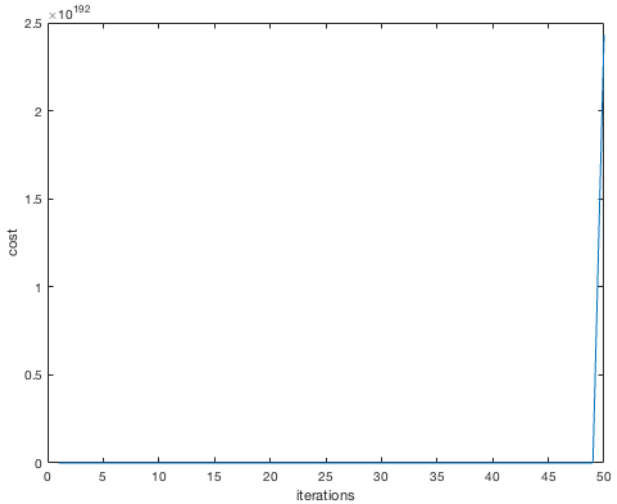
For Alpha – 0.1 – the gradient descent and the cost function



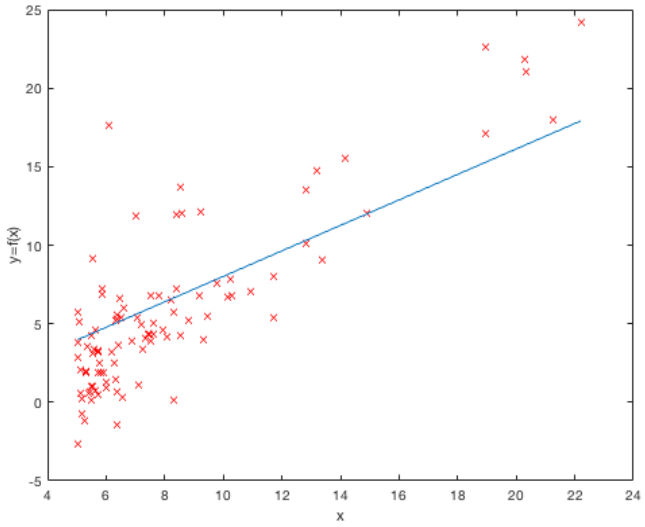


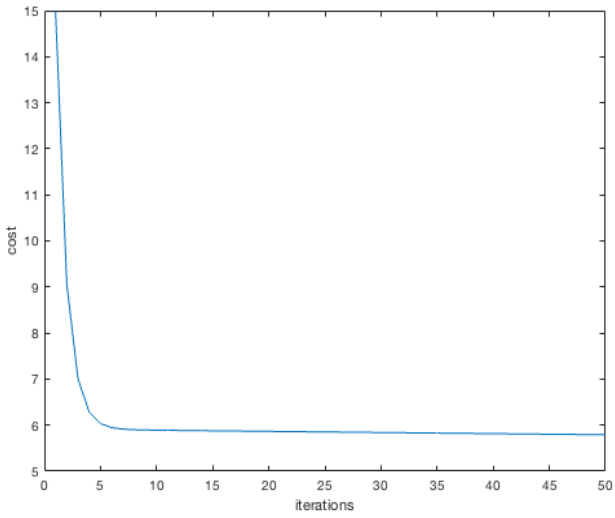
For Alpha – 1 – the gradient descent and the cost function



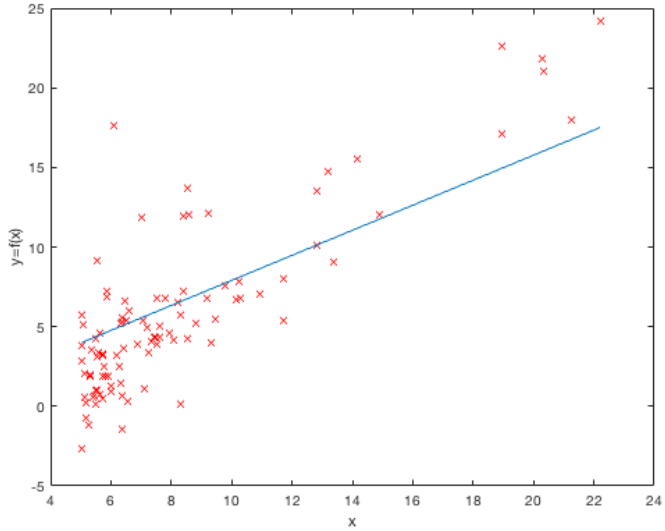


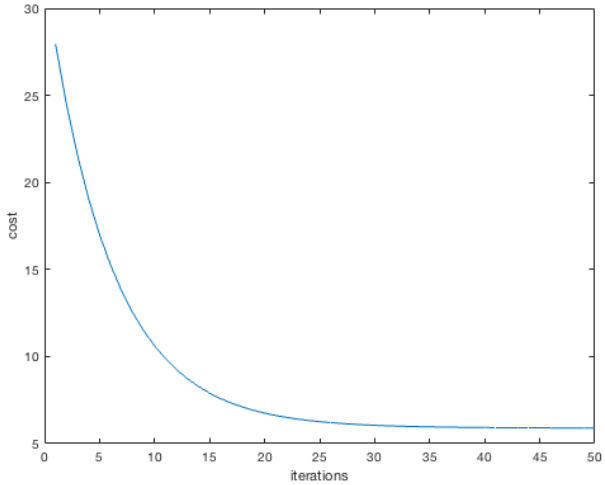
For Alpha – 0.005 – the gradient descent and the cost function



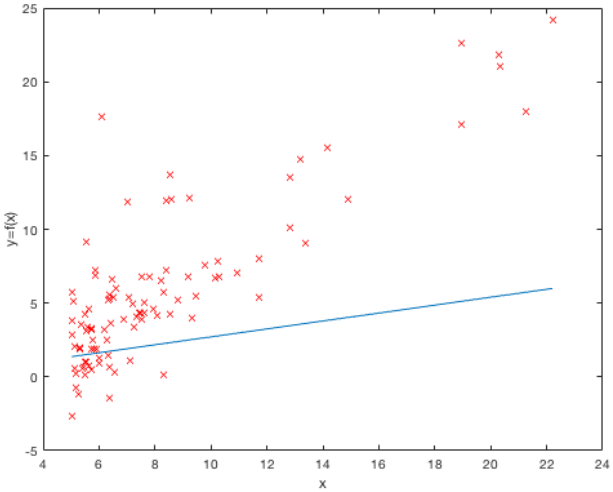


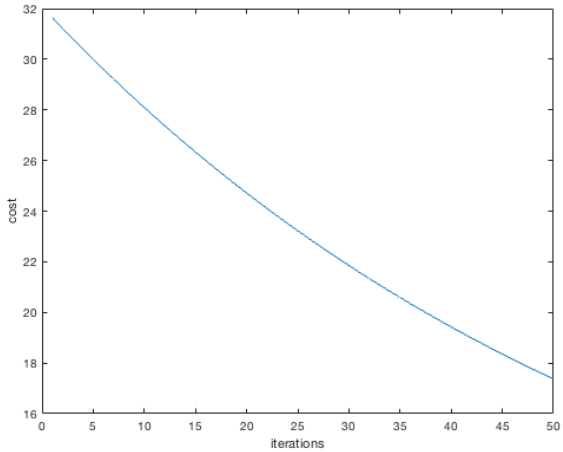
For Alpha – 0.001– the gradient descent and the cost function





For Alpha – 0.0001 – the gradient descent and the cost function





As can be seen, when alpha is too high the cost function shoots up and the gradient descent inverses. Whereas if the alpha is too low, there is over-fitting and the cost function is not optimised. Hence alpha 0.01 is the best value to fit the gradient.

Linear Regression with Two Variables

Task 2 – New Hypothesis function and gradient descent

In this task, the house prices data is included and where there are three variables now including the bias. The features are normalised.

- In calculate_hypothesis – the previous function is modified to include any number of variables.

```
function hypothesis = calculate_hypothesis(X, theta, training_example)
    %CALCULATE_HYPOTHESIS This calculates the hypothesis for a given X,
    %theta and specified training example

    %hypothesis = 0:0 ;

    hypothesis = X(training_example , :) * theta';
```

- In gradient_descent – theta is modified to include theta(3) i.e theta_2.

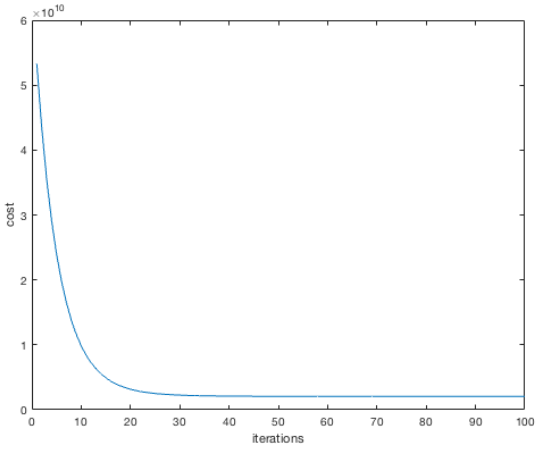
```
 %update theta(3) and store in temporary variable theta_1
        sigma = 0.0;

        for i = 1:m
            hypothesis = calculate_hypothesis(X, theta, i);
            output = y(i);
            sigma = sigma + (hypothesis - output) * X(i, 3);
        end

        theta_2 = theta_2 - ((alpha * 1.0) / m) * sigma;


        %update theta
        theta = [theta_0, theta_1, theta_2];
```

- In mllab2.m - theta is updated and is run for different values of alpha- The given alpha value is 0.1.

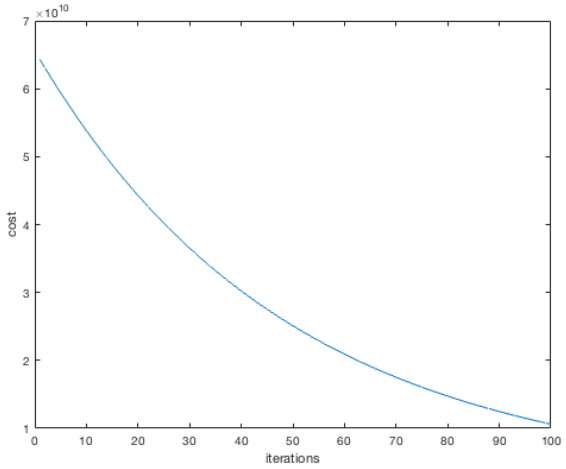
```
%initialise theta
theta = [0.0, 0.0, 0.0];
```

Alpha	Theta		
	Theta_0	Theta_1	Theta_2
0.1	3.4040e+05	1.0991e+05	-5.9311e+03
0.01	2.1581e+05	6.1384e+04	2.0274e+04
0.05	3.3840e+05	1.0396e+05	-18.4423
0.001	3.2410e+04	9.8391e+03	4.8944e+03
0.5	3.4041e+05	1.1063e+05	6.6495e+03
0.9	3.4041e+05	1.1063e+05	6.6495e+03
1	3.4041e+05	1.1063e+05	6.6495e+03

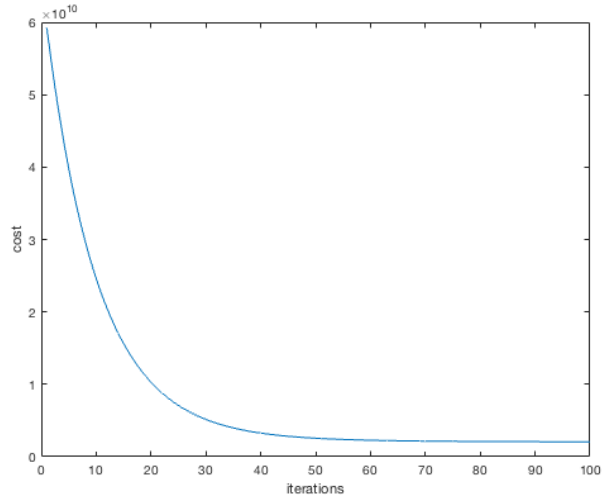
For Alpha = 0.1 (given), the cost function



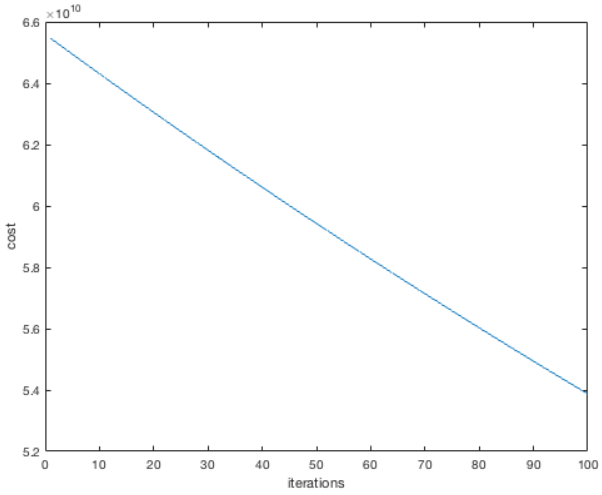
For Alpha = 0.01, the cost function



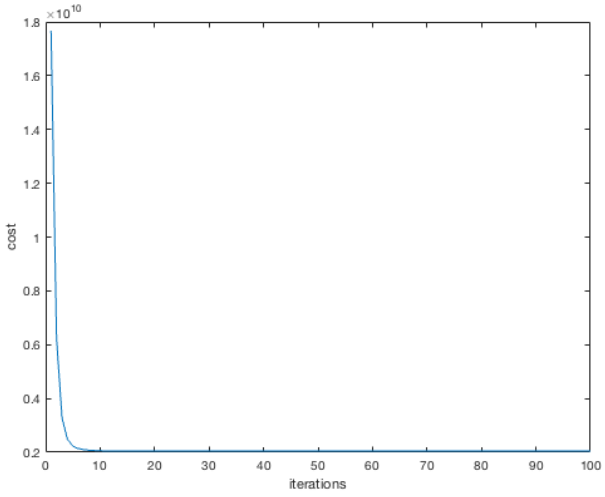
For Alpha = 0.05, the cost function



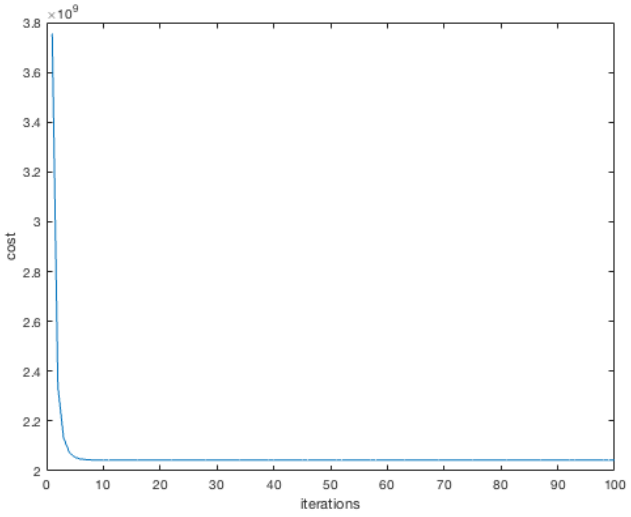
For Alpha = 0.001, the cost function



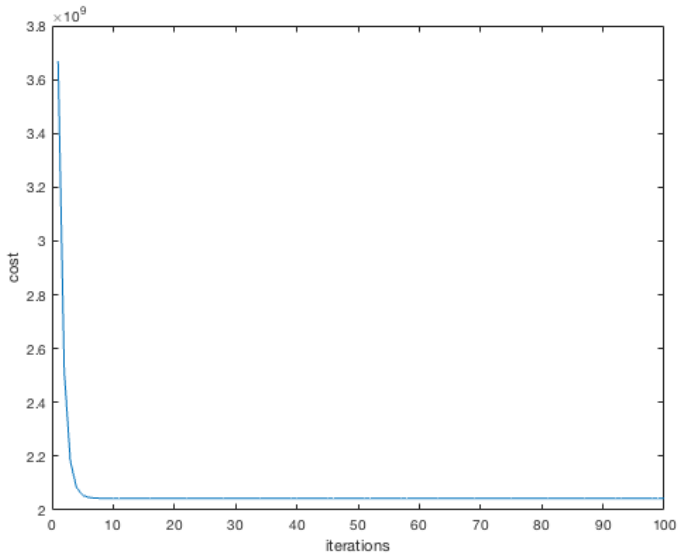
For Alpha = 0.5, the cost function



For Alpha = 0.9, the cost function



For Alpha = 1, the cost function



- It can be seen from the above, that after a certain point of alpha, the theta values almost remain constant. When the alpha value is greater than the most optimum alpha, the cost error stops to decrease effectively. Here, 0.5 is the best alpha.

- Now, in mllab2.m, we need to add a line of code to predict the house price. The two variables have to be normalised (x1 and x2). X_0 is the bias.
  Using the algorithm to predict the house price of 1650 sq.ft and 3 bedrooms,

```
%%Predict = X*t'

x0 = 1;
x1 = (1650 - mean_vec(1))/std_vec(1);
x2 = (3 - mean_vec(2))/std_vec(2);

predict = [x0 x1 x2]*t'
```

Answer = 2.9323e+05.
Thus, the cost is $293230.

We change the variables of X1 = [3000,4] to predict the house price of 3000 sq.ft. and 4 bedrooms

```
%%Predict = X*t'

x0 = 1;
x1 = (3000 - mean_vec(1))/std_vec(1);
x2 = (4 - mean_vec(2))/std_vec(2);

predict = [x0 x1 x2]*t'
```

Answer = 4.7215e+05
Thus, the cost is $472150

Regularized Linear Regression

Task 3 – Modifying Gradient Descent to use the compute_cost_regularised and regularising theta for fifth order polynomial

- In gradient_descent, we modify the cost function to use compute_cost_regularised instead of compute_cost

```
%update cost_vector
cost_vector = [cost_vector; compute_cost_regularised(X, y, theta,l)];
```

- Next, the gradient descent is modified to incorporate the new cost function with 6 variables including the bias term where all the terms except (X_0 – bias) are regularised.

```
theta_0 = theta(1);
theta_1 = theta(2);
theta_2 = theta(3);
theta_3 = theta(4);
theta_4 = theta(5);
theta_5 = theta(6);
%update theta(1) and store in temporary variable theta_0
sigma = 0.0;

for i = 1:m
   hypothesis = calculate_hypothesis(X, theta, i);
    output = y(i);
    sigma = sigma + (hypothesis - output);
end

theta_0 = theta_0 - ((alpha * 1.0) / m) * sigma;


%update theta(2) and store in temporary variable theta_1
sigma = 0.0;

for i = 1:m
     hypothesis = calculate_hypothesis(X, theta, i);
    output = y(i);
    sigma = sigma + (hypothesis - output) * X(i, 2);
end

theta_1 = theta_1*( 1 - ((alpha * l) / m)) - ((alpha * 1.0) / m)* sigma;

        sigma = 0.0;

for i = 1:m
    hypothesis = calculate_hypothesis(X, theta, i);
    output = y(i);
    sigma = sigma + (hypothesis - output)* X(i, 2)^2;
end

theta_2 = theta_2*( 1 - ((alpha * l) / m)) - ((alpha * 1.0) / m)* sigma;
          sigma = 0.0;
```

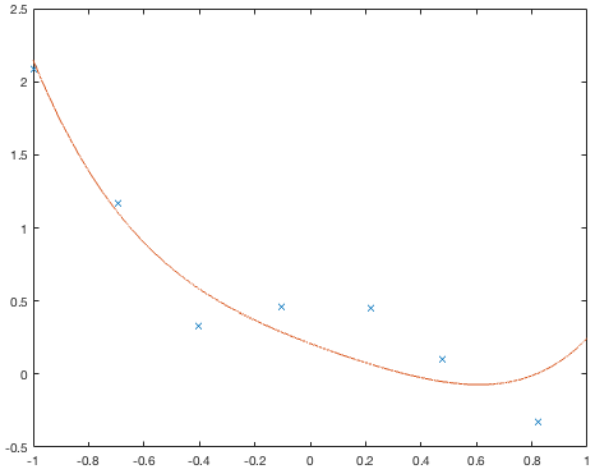
```
for i = 1:m
    hypothesis = calculate_hypothesis(X, theta, i);
    output = y(i);
    sigma = sigma + (hypothesis - output)*X(i, 2)^3;
end

theta_3 = theta_3*( 1 - ((alpha * l) / m)) - ((alpha * 1.0) / m)* sigma;
          sigma = 0.0;

for i = 1:m
    hypothesis = calculate_hypothesis(X, theta, i);
    output = y(i);
    sigma = sigma + (hypothesis - output)*X(i, 2)^4;
end

theta_4 = theta_4*( 1 - ((alpha * l) / m)) - ((alpha * 1.0) / m)* sigma;

          sigma = 0.0;

for i = 1:m
    hypothesis = calculate_hypothesis(X, theta, i);
    output = y(i);
    sigma = sigma + (hypothesis - output)*X(i, 2)^5;
end

theta_5 = theta_5*( 1 - ((alpha * l) / m)) - ((alpha * 1.0) / m)* sigma;

%update theta
theta = [theta_0, theta_1, theta_2, theta_3, theta_4, theta_5];
```

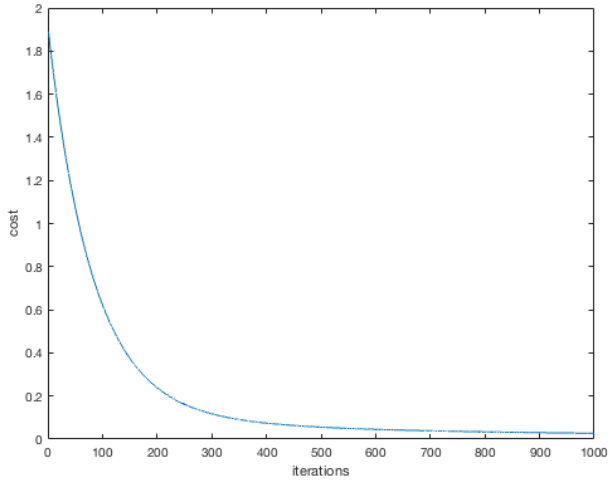
- Also, the cost-vector has to be updated in gradient_descent.m and the theta is updated to take the additional parameter lambda ‘l’ have to be updated in both mllab3.m and the gradient_descent.m

```
cost_vector = [cost_vector; compute_cost_regularised(X, y, theta,l)];
theta = gradient_descent(X, y, theta, alpha, iterations,l, do_plot)
```

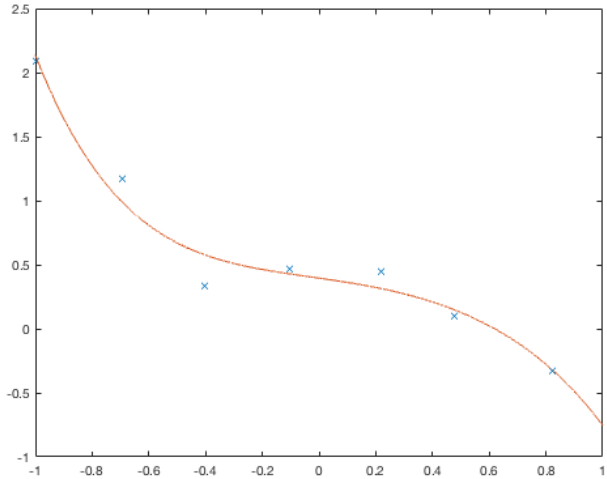
- The gradient descent and cost function is plotted for different values of alpha. The default value is 0.01- alpha and lambda = 0

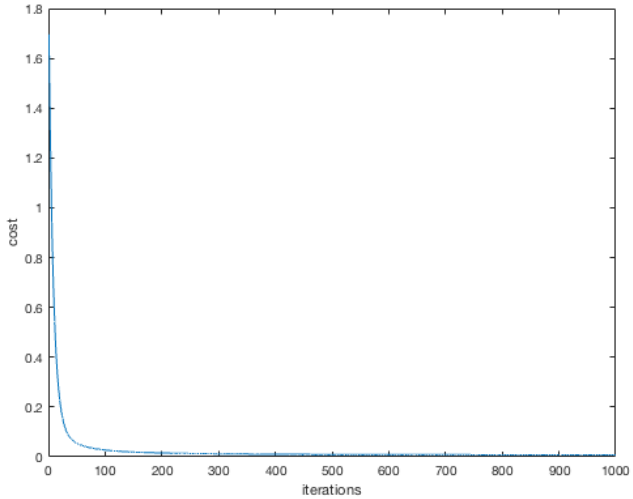
Gradient Descent and Cost function for Alpha = 0.01 (given)



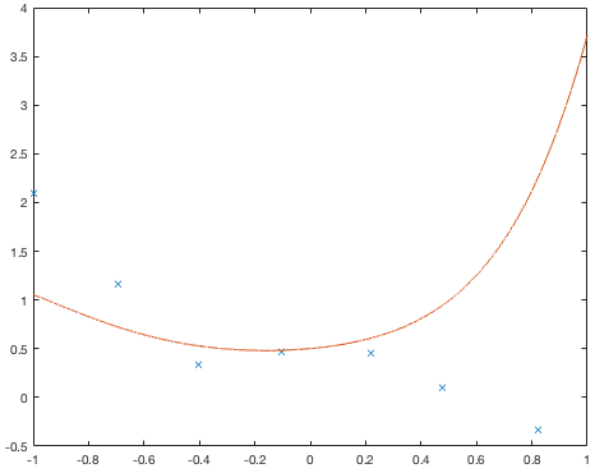


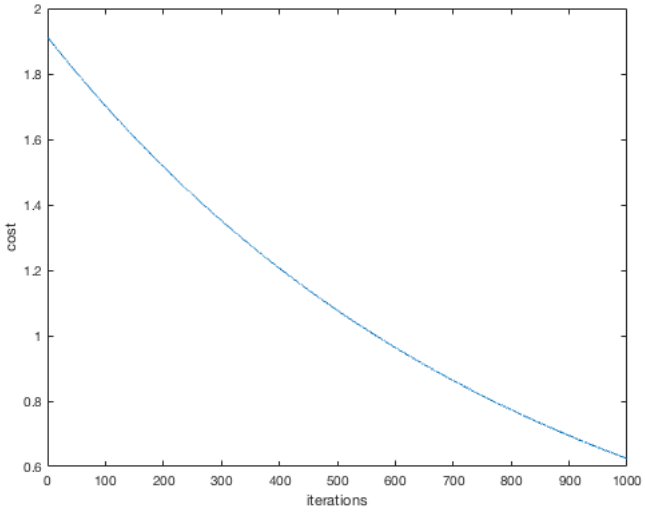
Gradient Descent and Cost function for Alpha = 0.1



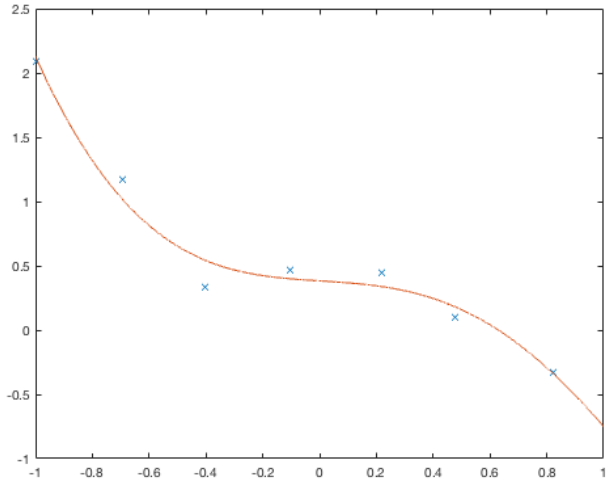


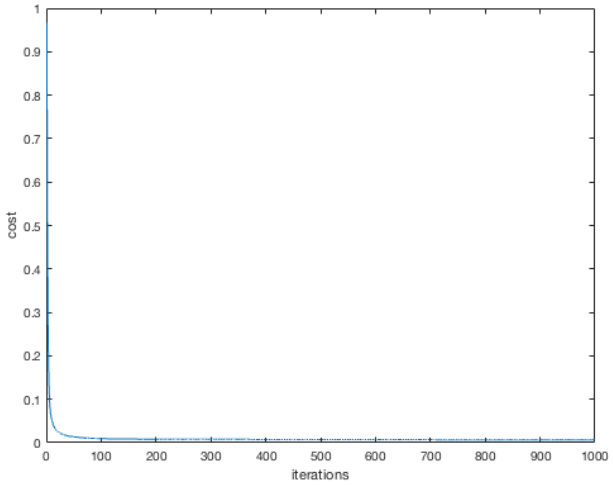
Gradient Descent and Cost function for Alpha = 0.001



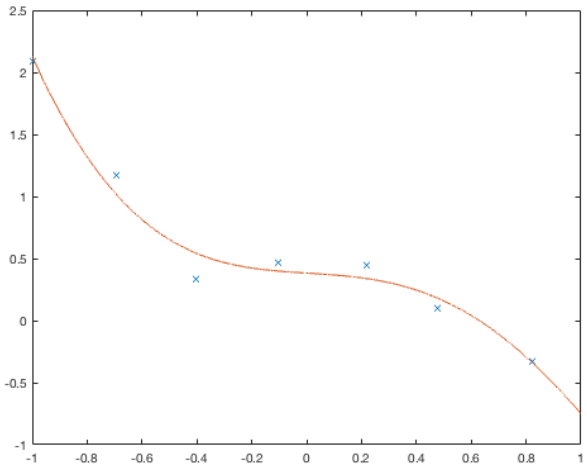


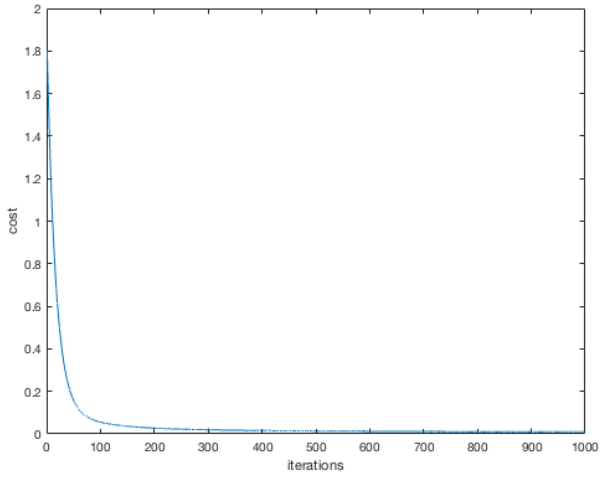
Gradient Descent and Cost function for Alpha = 0.5



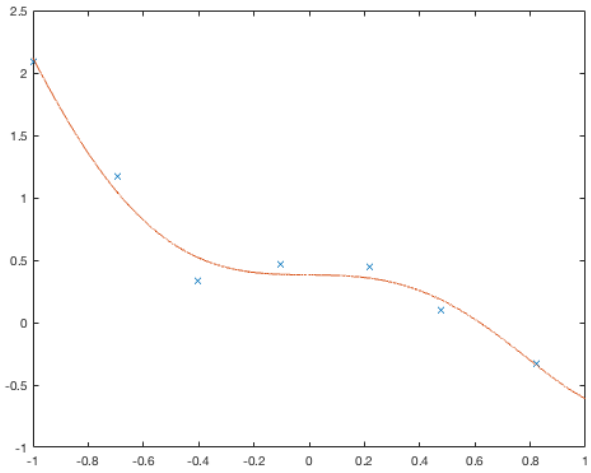


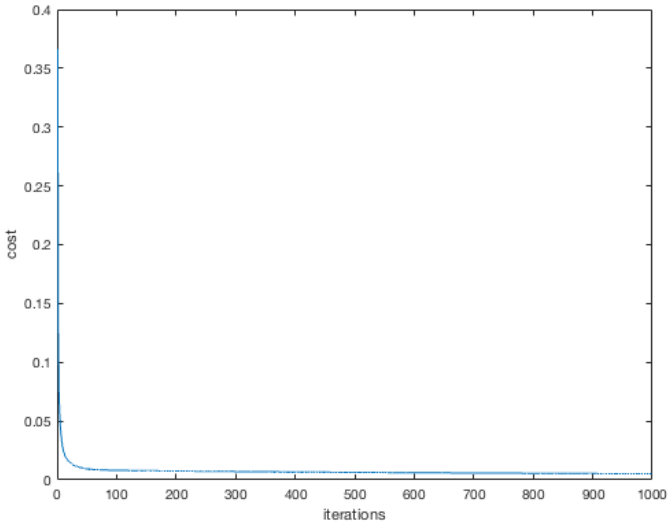
Gradient Descent and Cost function for Alpha = 0.05





Gradient Descent and Cost function for Alpha = 1

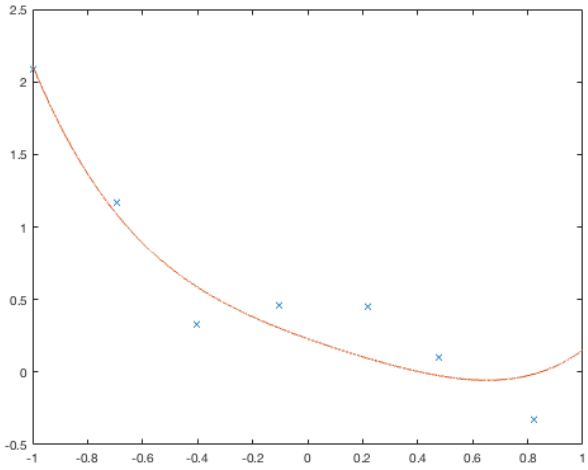


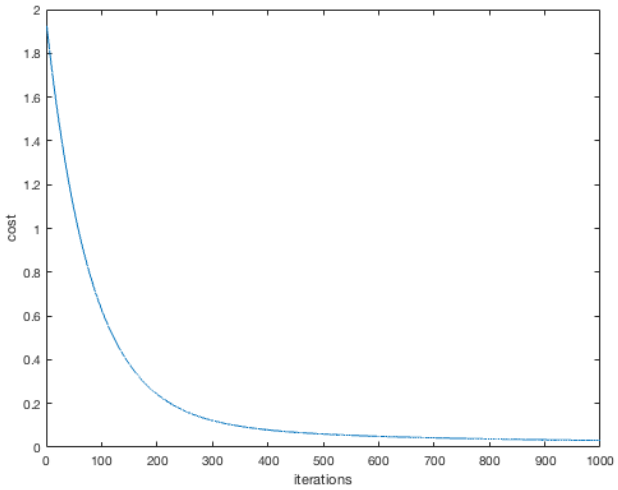


From the tested values, the best value of alpha is 0.1 where the cost error moves close to zero at a fast rate and has enough time to locate the local optima and has the best fit.

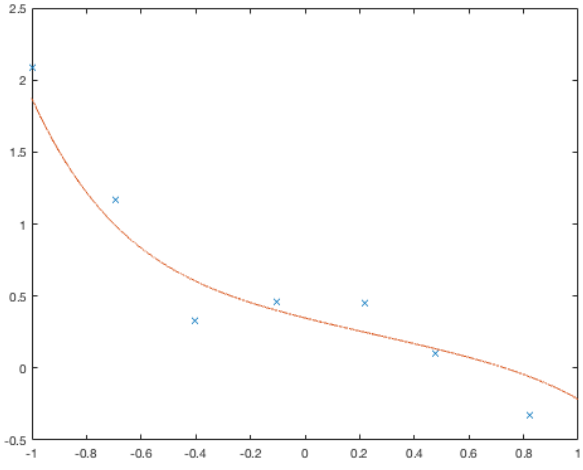
- Now, the cost function and hypothesis are tested for different values of lambda.

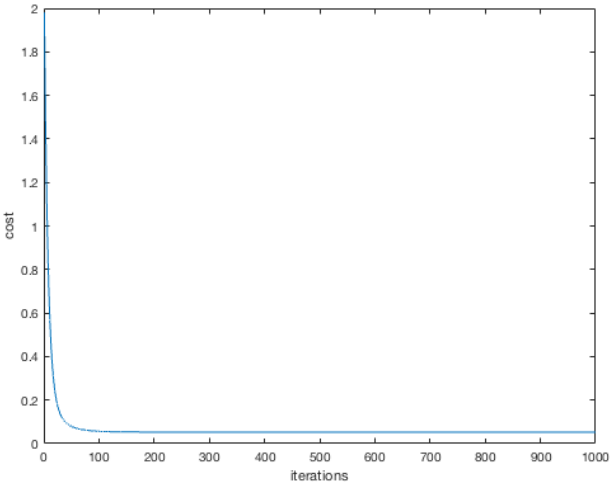
Gradient Descent and Cost function for Alpha = 0.01 and Lambda = 0.1



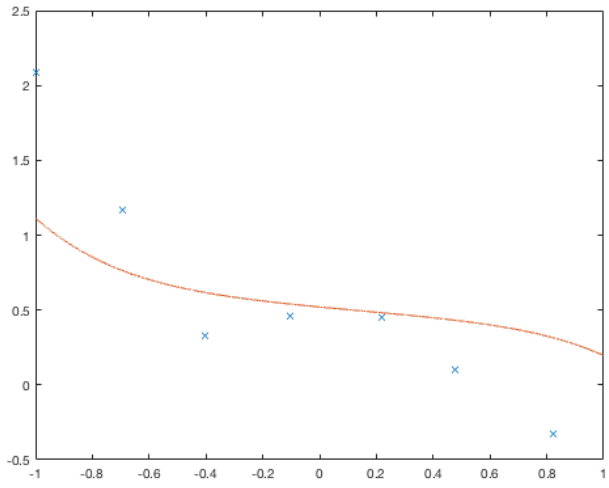


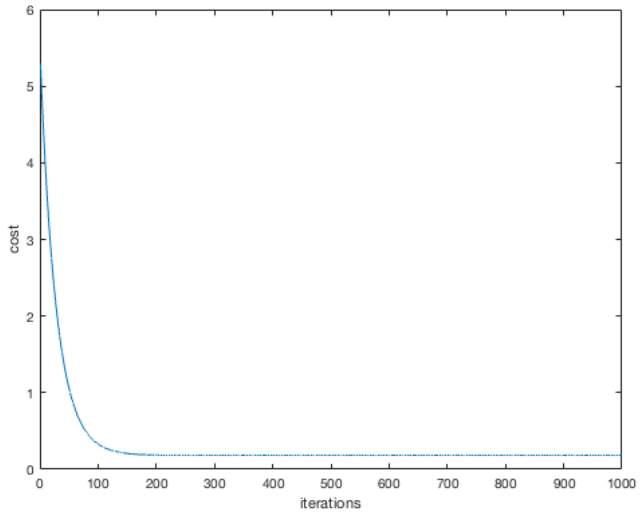
Gradient Descent and Cost function for Alpha = 0.01 and Lambda = 1



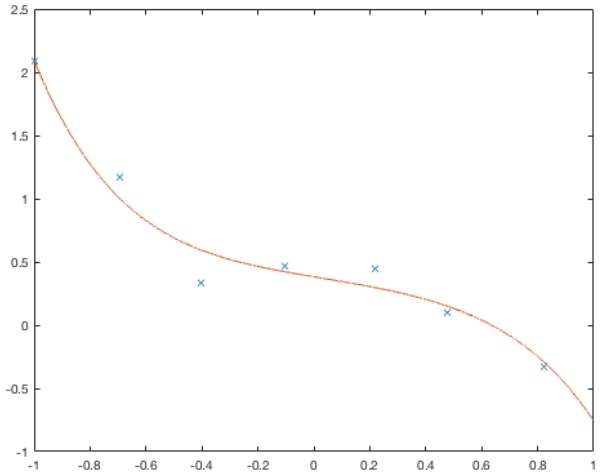


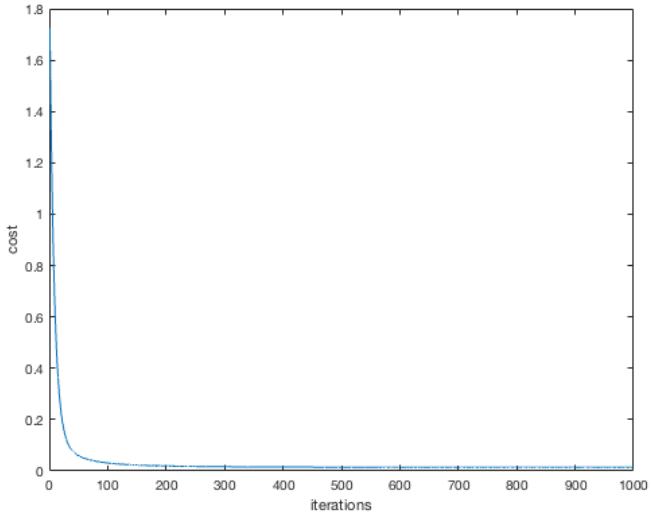
Gradient Descent and Cost function for Alpha = 0.01 and Lambda = 10



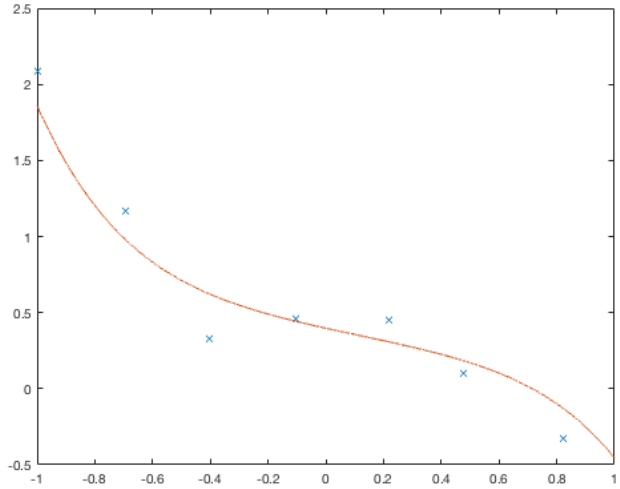


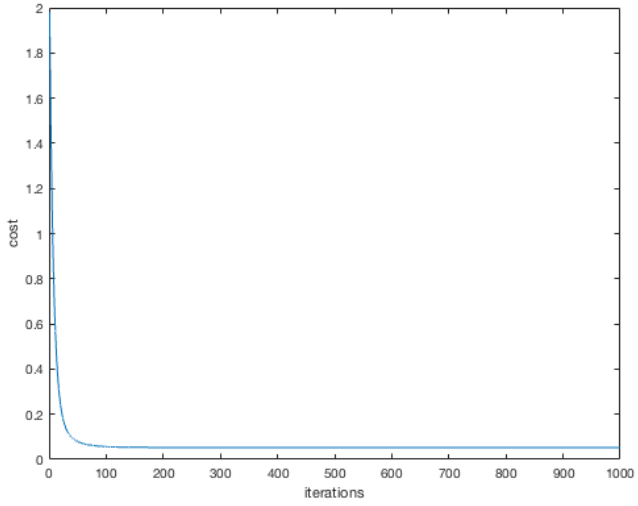
Gradient Descent and Cost function for Alpha = 0.1 and Lambda = 0.1



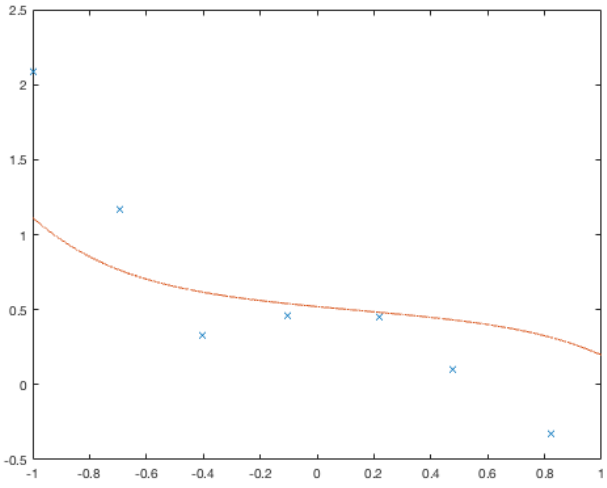


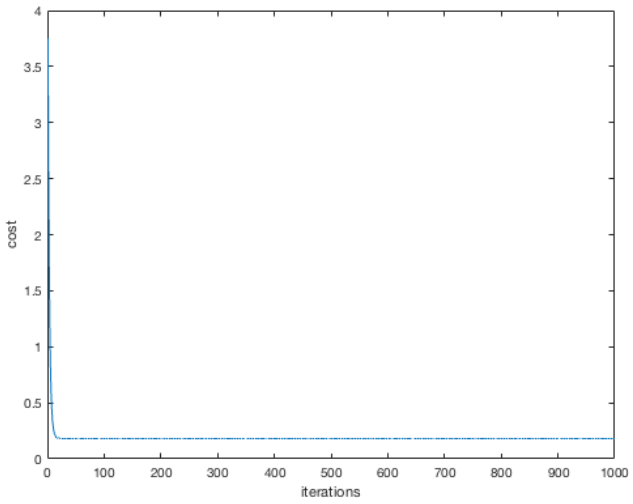
Gradient Descent and Cost function for Alpha = 0.1 and Lambda = 1





Gradient Descent and Cost function for Alpha = 0.1 and Lambda = 10





You can see that with different lambda values – the shape of the curve changes. The punishment for having more terms in your function is regularised by lambda. However, as can be seen the higher the value of lambda, the data is under-fitted. Hence, to get the best fit you would have to lower the lambda value. In this case, alpha 0.1 and lambda 0.1 gives you the best fit.

Part 2

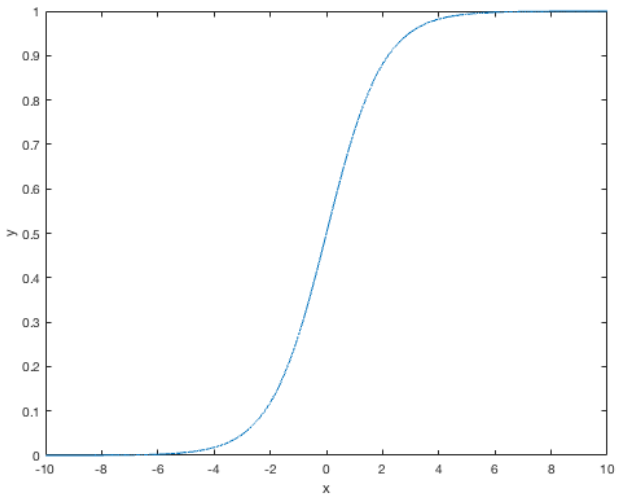
1. Logistic Regression

Task 1

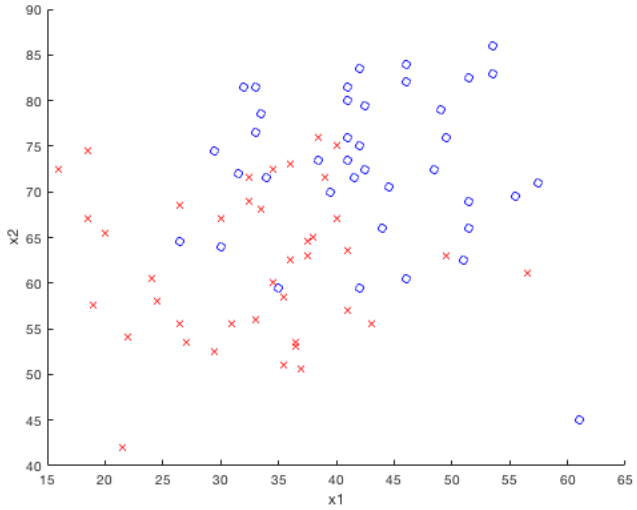
- The sigmoid function has been modified in sigmoid.m. Then the function has been plotted after running plot_sigmoid_function.m. Further, the data is also plotted after running plot_data.m

```
function output=sigmoid(z)
%output = 0;
% modify this to return z passed through the sigmoid function
%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%

output = 1 ./ (1 + exp(-z));

end
```

Sigmoid plot



Data plot

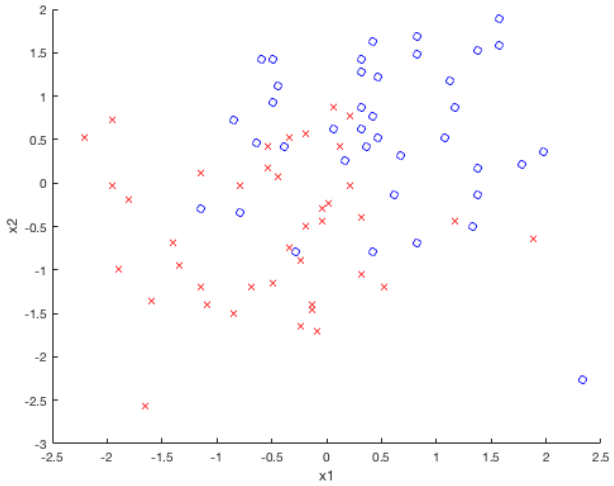


Task 2

- For the data to be optimised, the features have to be normalised. For this, the relevant normalisation code is uncommented in plot_data.m

```
% this loads our data
[X,y] = load_data_ex1();
% now we want to normalise our data
[X,mean,std] = normalise_features(X);
% after normalising we add the bias
X=[ones(size(X,1),1) X];
h=plot_data_function(X,y);
```

Data plot after normalisation



Task 3

- The calculate_hypothesis.m is modified to include the function for the prediction of any size datasets.

```
function result=calculate_hypothesis(X,theta,training_example)
   %hypothesis = 0.0;
    %%%%%%%%%%%%%%%%%%%%%%%%
    %Calculate the hypothesis for the i-th training example in X.
    hypothesis = X(training_example , :) * theta';
    %%%%%%%%%%%%%%%%%%%%%%%%
    result=sigmoid(hypothesis);
%END OF FUNCTION
end
```

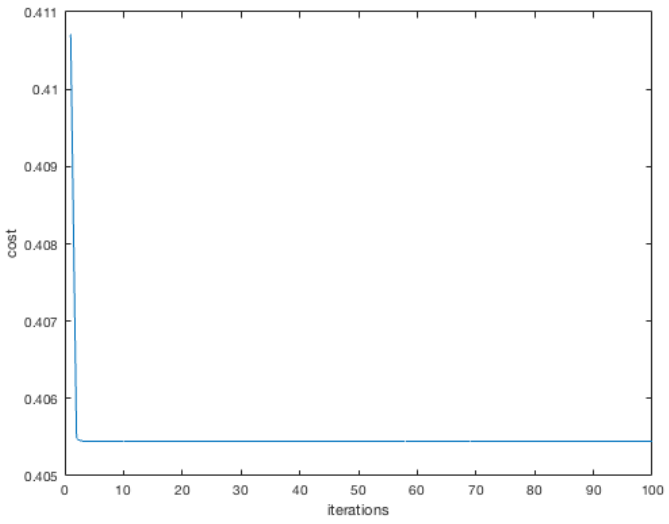
Task 4

- The cost function is updated in compute_cost.

```
    %
    %Compute cost for linear regression. Takes an input matrix X of training
    %examples, a parameter vector, theta, and an output vector y
    %
    J = 0.0; % cost
    m = size(X,1); % no. training examples
    for i=1:m
        hypothesis = calculate_hypothesis(X,theta,i);
        output = y(i);
        cost = -output*log(hypothesis)-(1-output)*log(1-hypothesis);
        % modify this to calculate the cost function, using hypothesis and output
        %cost = 0.0;
        J =J+cost;

    end
    J = J*(1.0/m);
end
    %END OF FUNCTION
```

- The final cost is computed and the graph is plotted from the gradient descent algorithm after running lab2_lr_ex1.m. The Final cost error is 0.40545

Cost function plot



Task 5

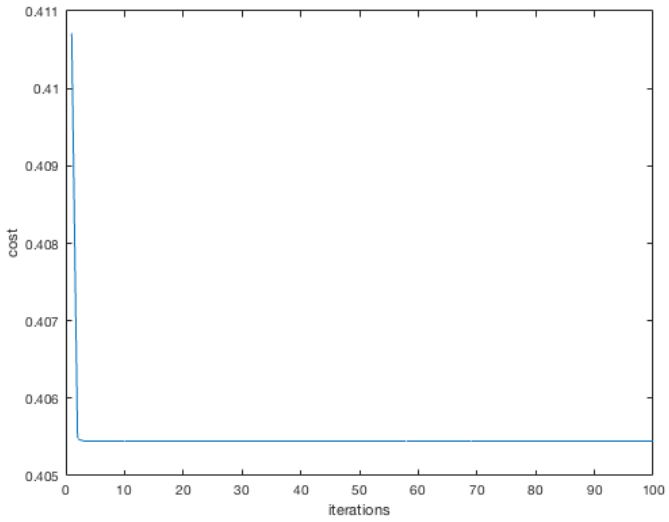
- The y1 and y2 are modified in plot_boundary.m in terms of x2.

```
%decision_boundary --- theta(1) + theta(2)*x1 + theta(3)*x2 = 0 where
%theta(1) is theta_0, theta(2) is theta_1 and theta(3) is theta_2

%y1 = 0.0;
x2 = (-theta(1) - theta(2)*min_x1)/theta(3);
y1 = x2;


% modify this: modified y2
%y2 = 0.0;
x2 = (-theta(1) - theta(2)*max_x1)/theta(3);
y2 = x2;
```

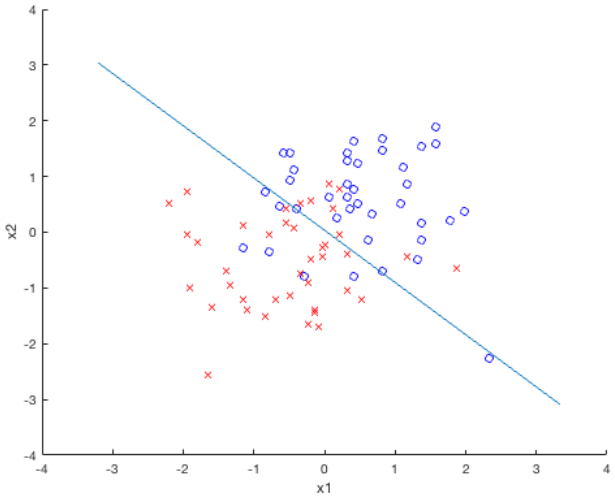
- The relevant lines are uncommented in lab2_lr_ex1.m and the cost function and the data with decision boundary are plotted.

```
% plot our data and decision boundary
plot_data_function(X,y)
plot_boundary(X,t)
```

Cost Function



Decision Boundary



Task 6

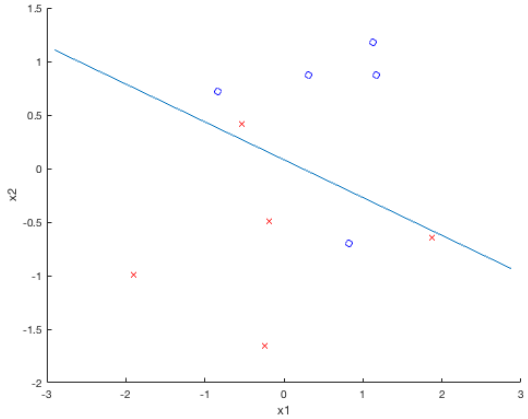
- Lab2_lr_ex2.m is run where the training is split into 10% and test is 90%. The code is run several times and the following results are generated.

Example	Train Error	Test Error
1	0.2815	0.52104
2	0.3725	0.53122
3	0.26945	0.80448
4	0.33428	0.62877
5	0.34903	0.57015
6	0.44293	0.51675
7	0.04458	0.65002
8	0.049212	1.4753
9	0.37179	0.47575
10	0.19504	0.59188
11	0.22278	0.52146
12	0.056239	0.73733
13	0.22445	0.68102
14	0.33264	0.535
15	0.025283	0.86199

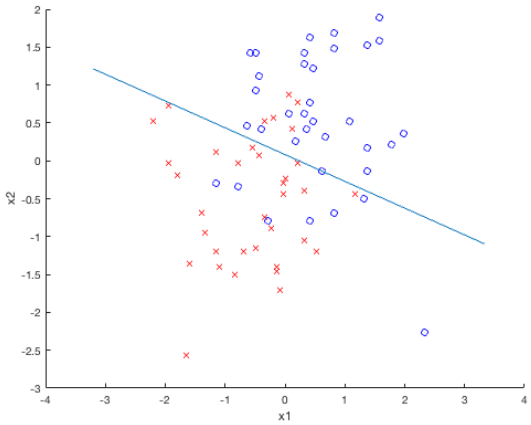
Since, it is a 90% test split, the training error is not so significant since it only comprises 10% of the data. Usually, a good generalisation is when both the train error and test error are low. If the test error is high and the train error is low, it leads to over-fitting and it is a bad generalisation. In this particular case, we focus on the test error to determine the good and bad generalisation.

- Good Generalisation is when the train error is 0.37179 and test error is 0.47575.

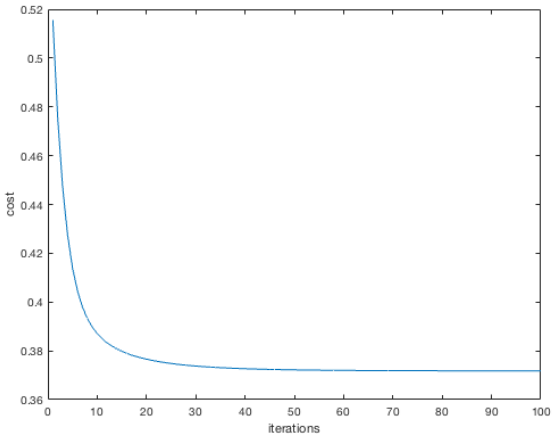
Train Data



Test Data

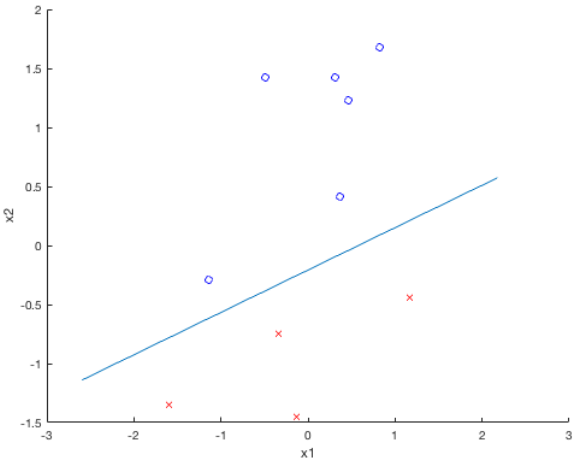


Cost Function

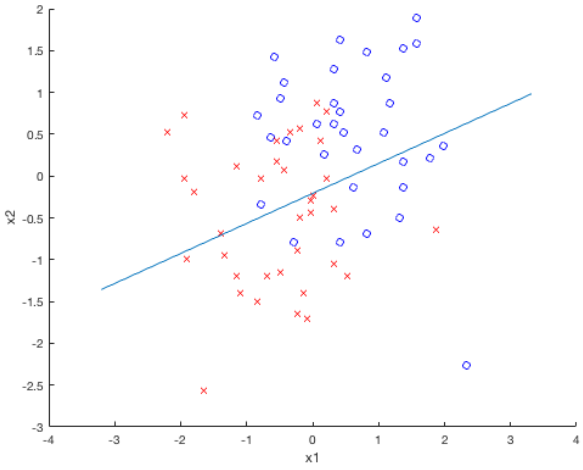


- Bad Generalisation is when the train error is 0.049212 and test error is 1.4753.

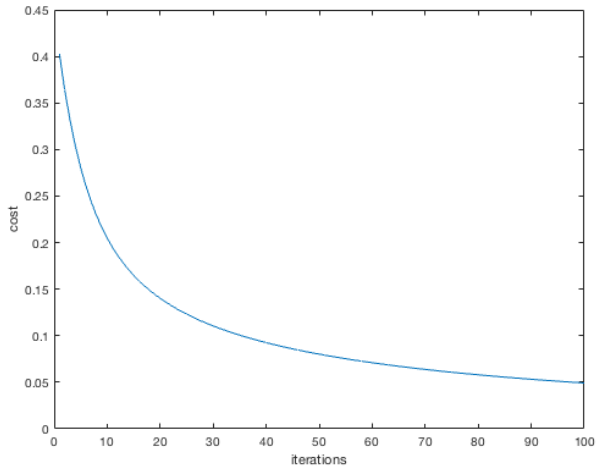
Train Data



Test Data



Cost Function

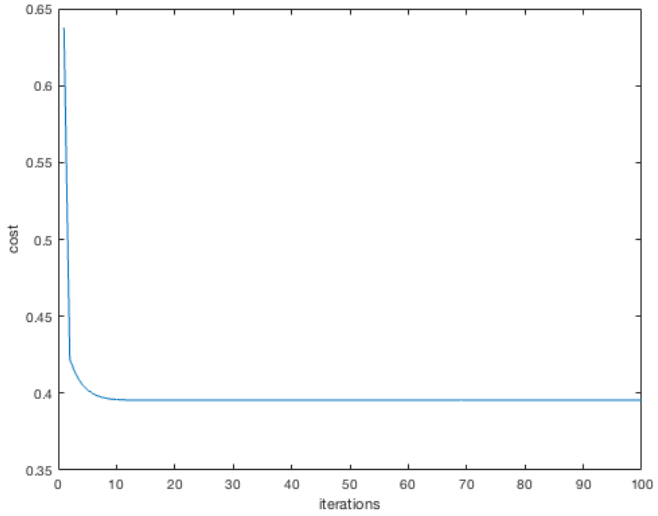


Task 7

- Firstly, in Lab2_lr_ex3a.m need to change the 2D vector to a 5D vector to incorporate the new non-linear features such as x1 * x2 * $x1^3$ and $x2^2$ and 6 parameters of theta are now used. The following lines of code have been added to modify the function.

```
% for question 7, modify the dataset X to have more features (in each row)
% append to X(i),the following features:
% here append x_2 * x_3 (remember that x_1 is the bias)

% here append x_2 * x_2 (remember that x_1 is the bias)

% here append x_3 * x_3 (remember that x_1 is the bias)
% Task 7 - to have more features


for i = 1:size(X,1)
X(i,4)=X(i,2)*X(i,3);
X(i,5)=X(i,2)*X(i,2);
X(i,6)=X(i,3)*X(i,3);
end
```

- The final cost error is computed and the cost is plotted. The final cost error is 0.39537. When compared to the cost error of task 4(is 0.40545), it is smaller as there is an increase in the size of the data due to the new non-linear features.



Task 8

- The features from task 7 are added in Lab2_lr_ex3b.m.

```
% after normalising we add the bias
X=[ones(size(X,1),1),X,ones(size(X,1),1),ones(size(X,1),1),ones(size(X,1),1)];
%X=[ones(size(X,1),1),X];
% for question 7, modify the dataset X to have more features (in each row)
    % append to X(i),the following features:
    % here append x_2 * x_3 (remember that x_1 is the bias)

    % here append x_2 * x_2 (remember that x_1 is the bias)
%
    % here append x_3 * x_3 (remember that x_1 is the bias)

% initialise theta. Remember that theta needs to be
% the same size as one row of X
% Task 7 - to have more features
% Task 8 - an additional feature is added.

for i = 1:size(X,1)
    X(i,4)=X(i,2)*X(i,3);
    X(i,5)=X(i,2)*X(i,2);
    X(i,6)=X(i,3)*X(i,3);
    %X(i,7)=X(i,6)*X(i,3);
end
```

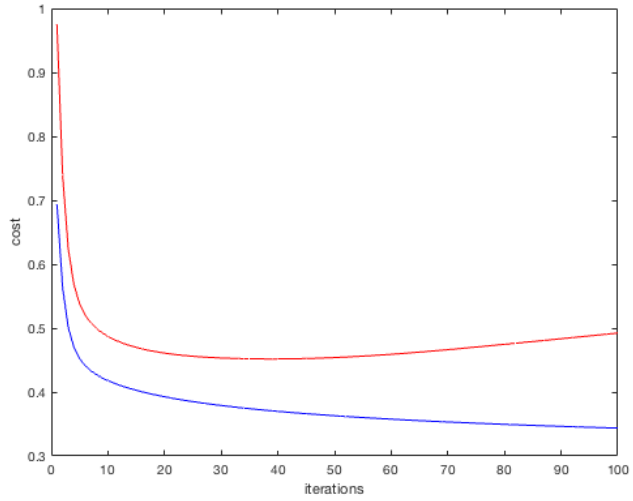
- The gradient_descent_training.m is modified to include the current cost for training and test set. These are stored in cost_array_training and cost_array_test respectively.

```
% update cost_array
        cost_array_training (it)=compute_cost(X, y, theta);
        cost_array_test (it)= compute_cost(test_X, test_y, theta);
        % code added here: to update cost array training and cost array test.
```

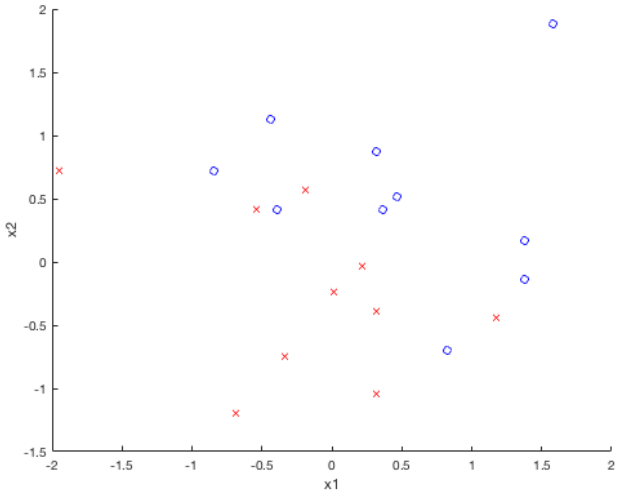
- After adding the code, Lab2_lr_ex3b.m is run and the cost function is plotted. The training cost error is blue and the test cost error is red.
  The default size given is 25% test split where the test data size is 20 and the total dataset size = 80. The remaining (75%) is train data.

  1) Test dataset size = 20 ----------75% training 25% Test
     Training Error:0.34361
     Test Error:0.49222

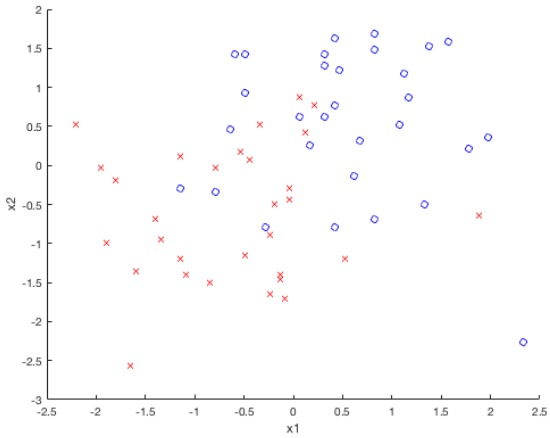
Cost Function



Training Set

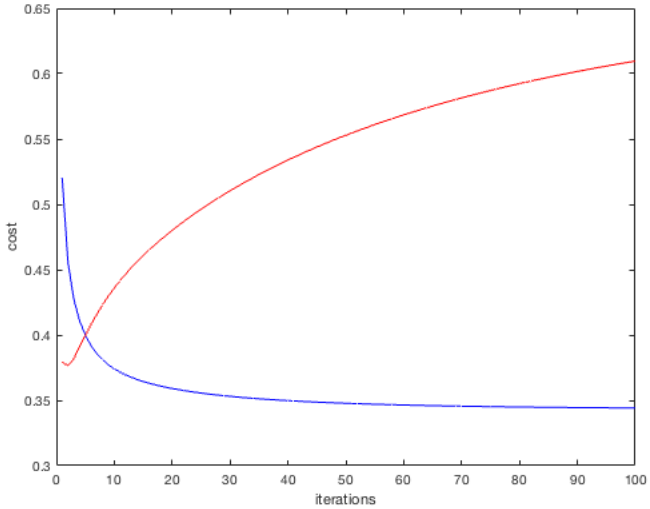


Test Set

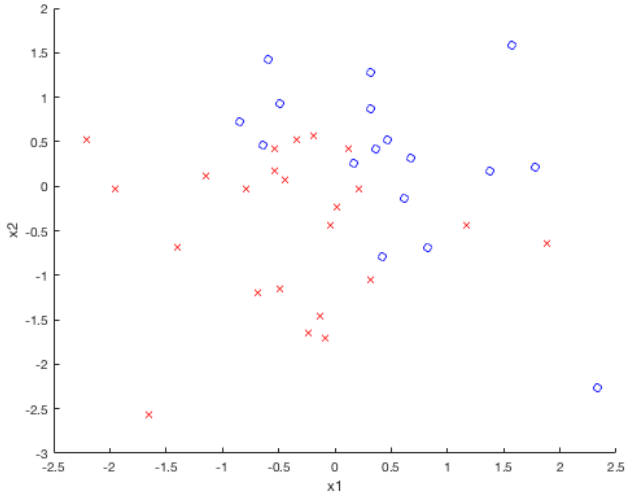


  2) Test dataset size = 40 --------- 50% training 50% test
     Training:0.34412
     Test:0.60961

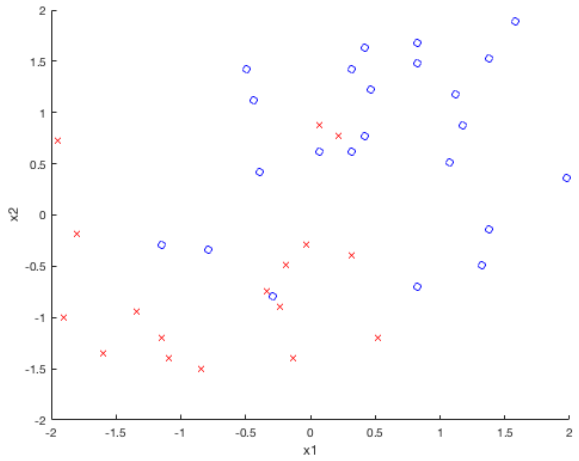
Cost Function



Training Set

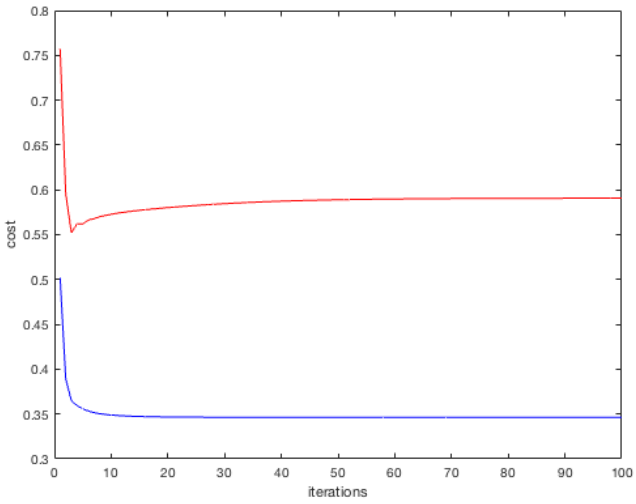


Test Set

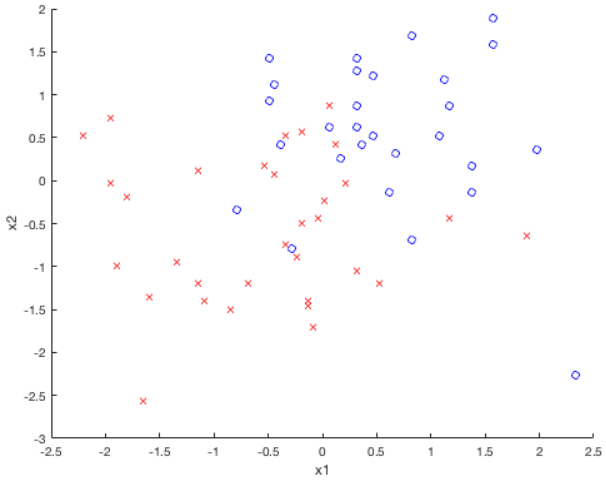


3) Test dataset size = 60 ------- 25% training 75% test
Training Error :0.34629
Test Error :0.5907

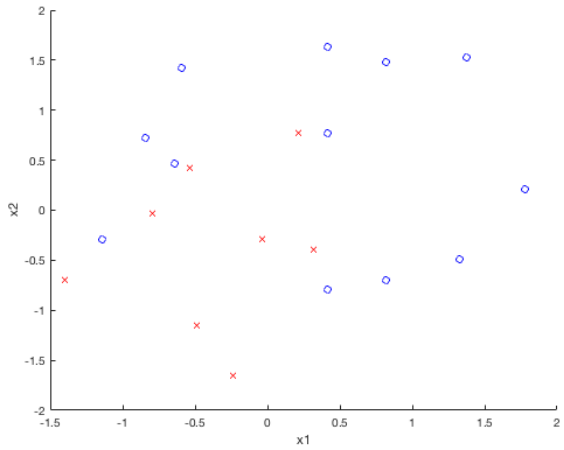
Cost Function



Training Set



Test Set



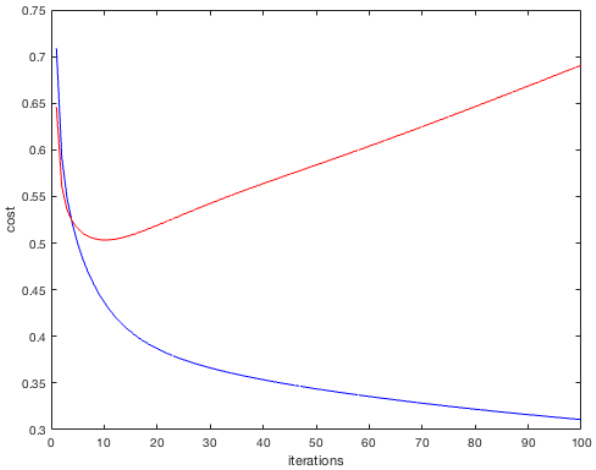
As can be seen, when the size of the training data is decreased and test data increased, it increases the cost error for the test data. Results in over-fitting. From the above, test set size 20 (25% test data and 75% training data split) gives the best generalisation.

- Now, an additional feature (3rd degree polynomial is added to the existing features) and it is again tested for different sizes. Also, now the theta has 7 parameters due to the extra variable.

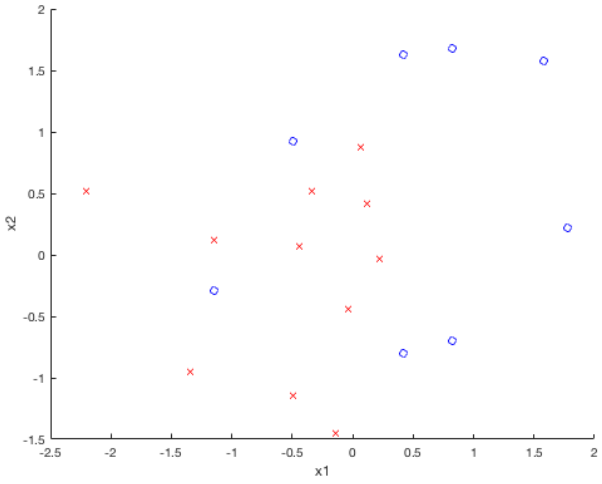
```
%Task 8 - an additional feature is added.
for i = 1:size(X,1)
X(i,4)=X(i,2)*X(i,3);
X(i,5)=X(i,2)*X(i,2);
X(i,6)=X(i,3)*X(i,3);
X(i,7)=X(i,6)*X(i,3);
end

theta=[1.0,1.0,1.0,1.0,1.0,1.0,1.0];
```

1) Test dataset size = 20 ----------75% training 25% Test
   Training Error:0.3109
   Test Error:0.69086

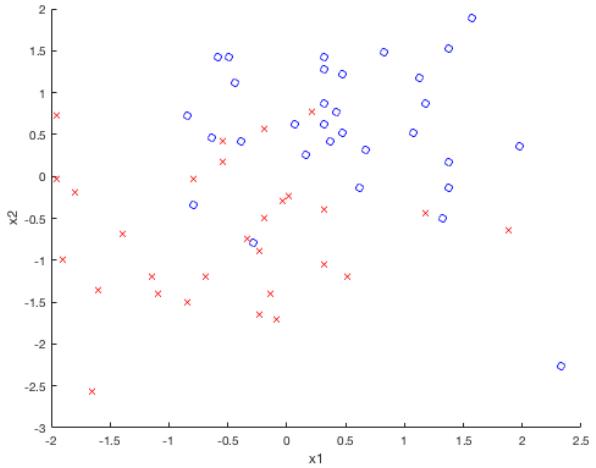
Cost Function



Training Set

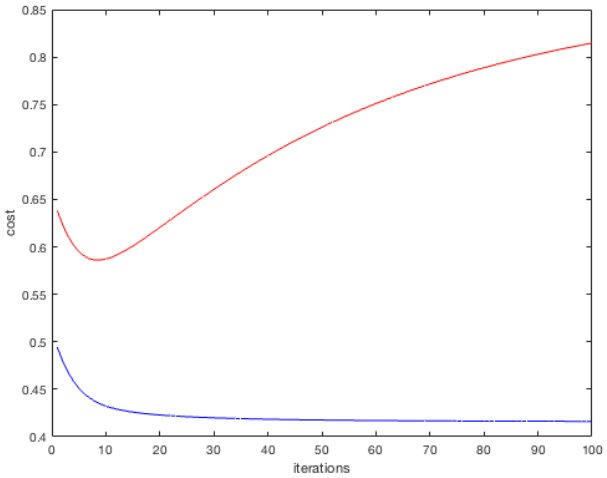


Test Set

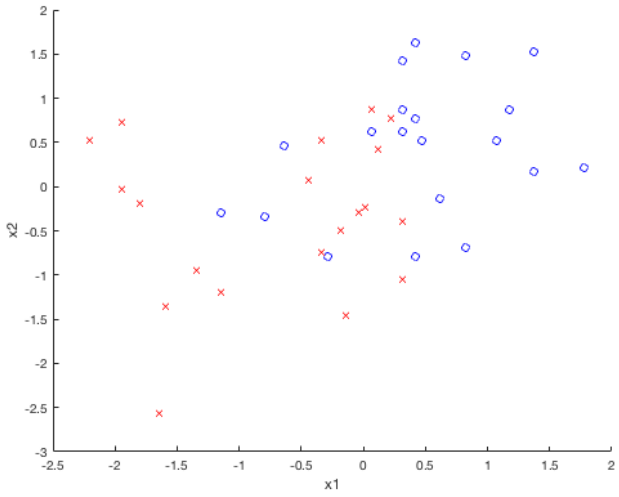


2) Test dataset size = 40 --------- 50% training 50% test
Training:0.41619
Test:0.81479

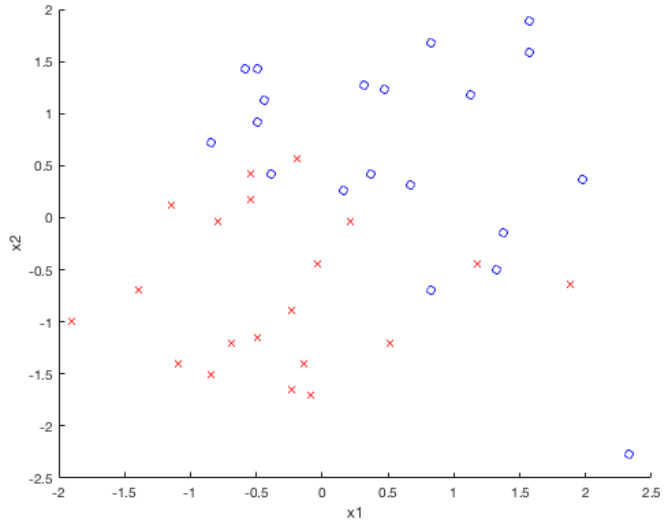
Cost Function



Training Set

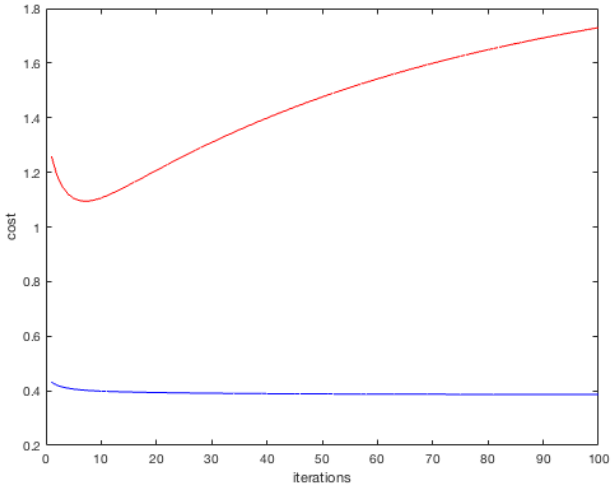


Test Set

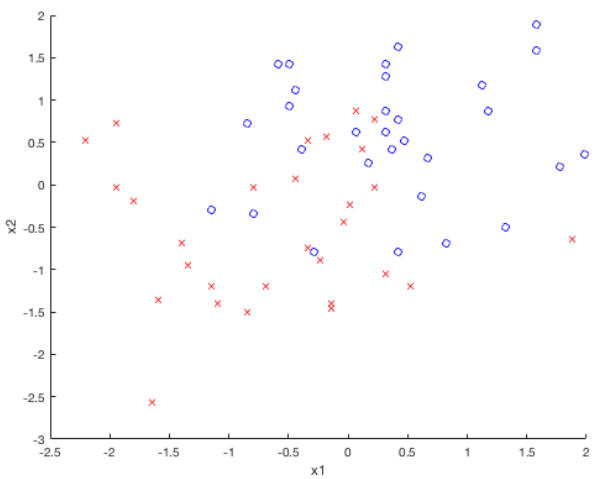


3) Test dataset size = 60 --------- 25% training 75% test
Training Error:0.38637
Test Error:1.7299

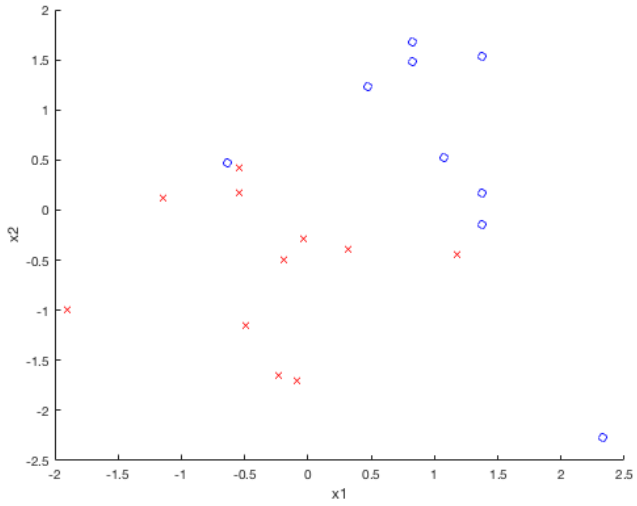
Cost Function



Training Set



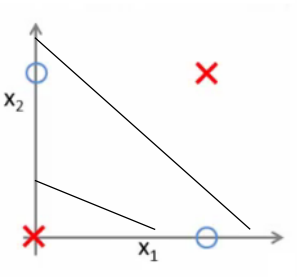
Test Set



When there is an increase in parameters (3rd order polynomial), it can be seen that the error increases and it leads to overfitting.
As can be seen from the graphs, when the training cost goes down and test goes up, there is overfitting and it is a bad generalisation.
With the increase in polynomial order, it leads to overfitting as the data remains the same but the features increase.

Task 9

Logistic Regression Unit cannot solve the XOR classification problem as can be seen the decision space diagram below. We need more than one decision boundaries at the same time to solve the problem. However, logistic regression runs only one at a time. Thus, we use neural networks for XOR classification.



2. Neural Networks

Task 10

- First, the sigmoid(z) function in sigmoid.m is modified-

```
sigmoid_output = 1 ./ (1 + exp(-z));
```

- Then, the steps 1-4 are implemented in NeuralNetwork.m

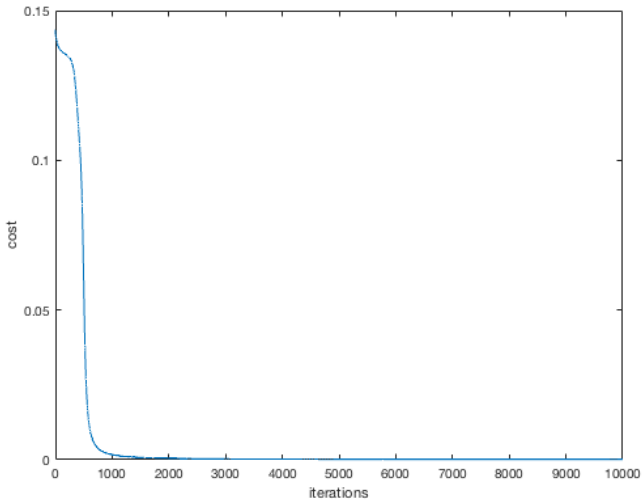
```
function J=back_propagate(inputs,nn,targets,learning_rate,batch)
%
%Backpropagates the error and performs gradient descent on the network weights
%
% Step 1. Output deltas (used to change weights from hidden --> output)
output_deltas = zeros(1,length(nn.output_neurons));
outputs=nn.output_neurons;
for i=1:length(outputs)
output_deltas(i) = (outputs(i)-targets(i))*sigmoid_derivative(outputs(i));
end

% Step 2. Hidden deltas (used to change weights from input --> output).
hidden_deltas = zeros(1,length(nn.hidden_neurons));
% hint... create a for loop here to iterate over the hidden neurons and for each
% hidden neuron create another for loop to iterate over the ouput neurons
%hidden = nn.hidden_neurons;
for i = 1: length(nn.hidden_neurons)
for j = 1:length(outputs)
hidden_deltas(i) = sum(nn.output_weights(i,j)*output_deltas(j))*sigmoid_derivative(nn.hidden_neurons(i));
end
end
% Step 3. update weights output --> hidden
for i=1:length(nn.hidden_neurons)
for j=1:length(output_deltas)
nn.output_weights(i,j) =nn.output_weights(i,j) -(output_deltas(j) * nn.hidden_neurons(i) * learning_rate);
end
end
```

```
% here we are removing the bias from the hidden neurons as there is no
% connection to it from the layer below
hidden_deltas = hidden_deltas(2:end);

% Step 4. update weights input --> hidden.
% hint, use a similar process to step 3, except iterate over the input neurons and hidden deltas
for i=1:length(inputs)
for j=1:length(hidden_deltas)
nn.hidden_weights(i,j) =nn.hidden_weights(i,j) -(hidden_deltas(j) * inputs(i) * learning_rate);
end
end
```

- Then, backpropagation is implemented on xorExample.m and the cost is tested for different learning rates (alpha). The given learning rate is 1.0 in xorExample.m. The graphs for different learning rates have been plotted.

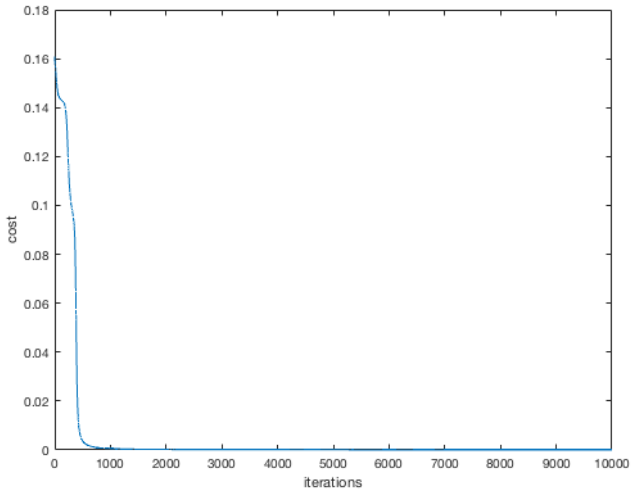
For alpha = 1(given)

target output:0actual output0.012794
target output:1actual output0.98895
target output:1actual output0.98901
target output:0actual output0.011506
cost = 6.7402e-05

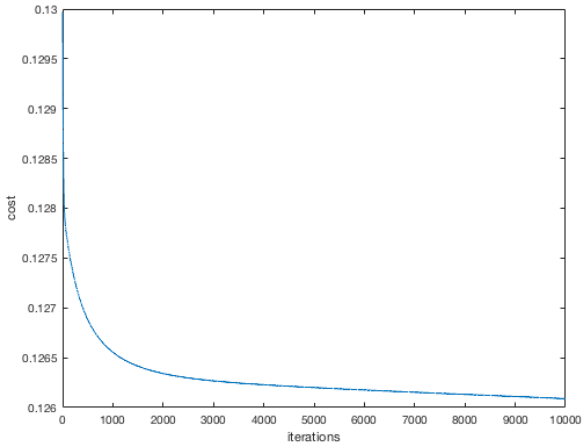


For alpha = 2

cost = 3.2196e-05
target output:0actual output0.008005
target output:1actual output0.99079
target output:1actual output0.99246
target output:0actual output0.0071988

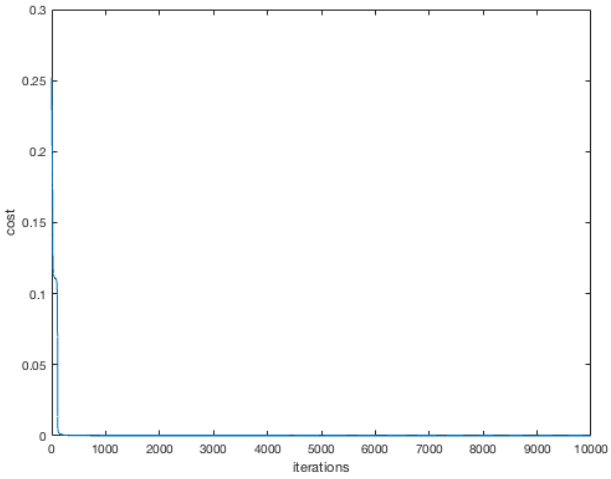


For alpha = 0.1
cost = 0.12609
target output:0actual output0.50634
target output:1actual output0.49889
target output:1actual output0.50167
target output:0actual output0.49408



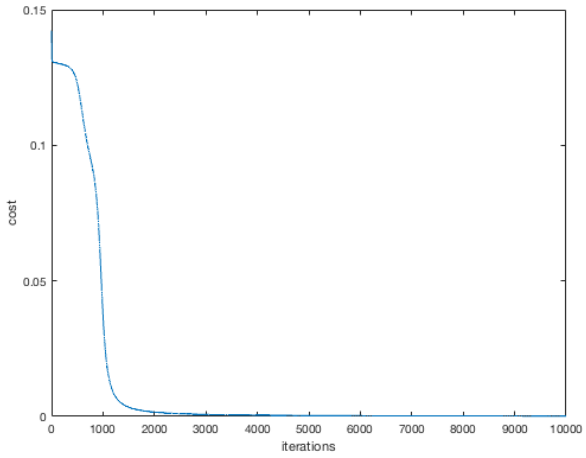
For alpha = 10

cost = 6.2368e-06
target output:0actual output0.0032648
target output:1actual output0.99669
target output:1actual output0.99662
target output:0actual output0.0040989



For alpha = 0.5

cost = 0.00014662
target output:0actual output0.017112
target output:1actual output0.98402
target output:1actual output0.98029
target output:0actual output0.015357



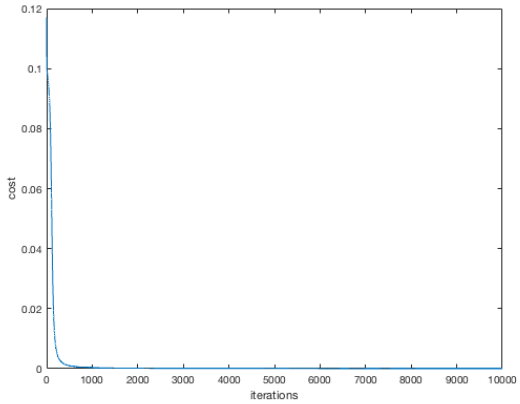
- From the above graphs, you can tell that learning rate 1 is the best as it minimises the cost to zero at an increasing rate, but at the same it is not dropping too fast. When it drops too fast, you cannot locate the global or local minima.
- Sometimes, the backpropagation can get stuck in local optima. This means the cost error decreases but at a decreasing rate. From the graphs above, it can be seen that for learning rate 0.1, backpropagation gets stuck in local optima.
  Alpha = 0.1
  cost = 0.12609
  target output:0actual output0.50634
  target output:1actual output0.49889
  target output:1actual output0.50167
  target output:0actual output0.49408

Task 11

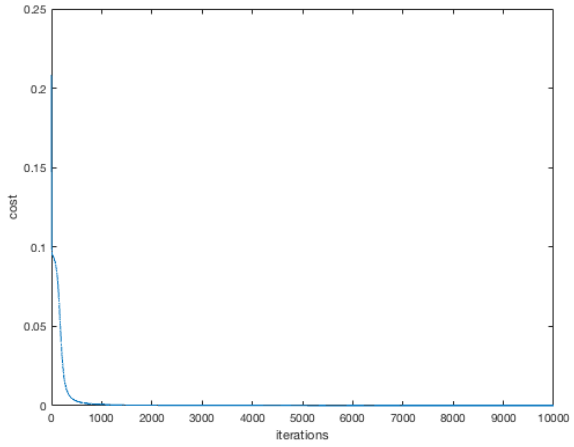
- Similar to XOR, now two other logical functions NOR [1000] and AND[0001] are implemented. The logical operation is modified in the training_set_output. These are given in norExample.m and andExmaple.m respectively.
- The functions have been tested for different values of alphas and the cost error function has been plotted for given (1.0) alpha value and the best alpha value.

NOR

Alpha – 1 (given)
cost = 2.4524e-05
target output:1actual output0.98887
target output:0actual output0.0059495
target output:0actual output0.0059588
target output:0actual output0.0012038



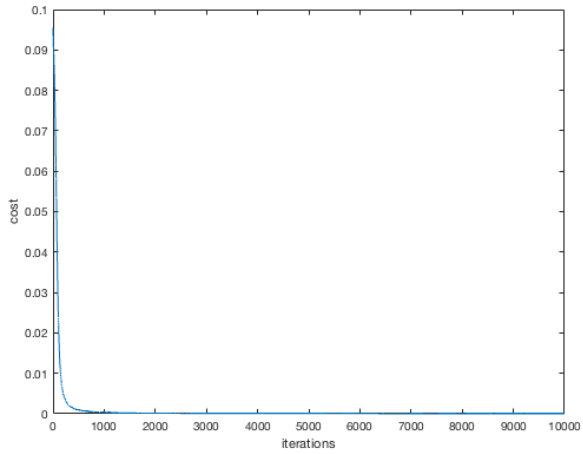
Alpha – 0.5
cost = 4.0871e-05
target output:1actual output0.98672
target output:0actual output0.0090523
target output:0actual output0.0081102
target output:0actual output0.0016878



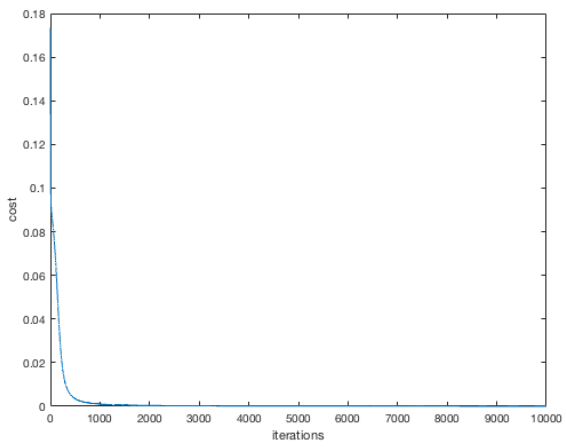
Learning rate 0.5 gives the best successive trial since it drops fast enough but also has enough time to capture the local/global minima.

AND

Alpha – 1 (given)
cost = 2.1516e-05
target output:0actual output0.00024702
target output:0actual output0.0061656
target output:0actual output0.0064967
target output:1actual output0.99042



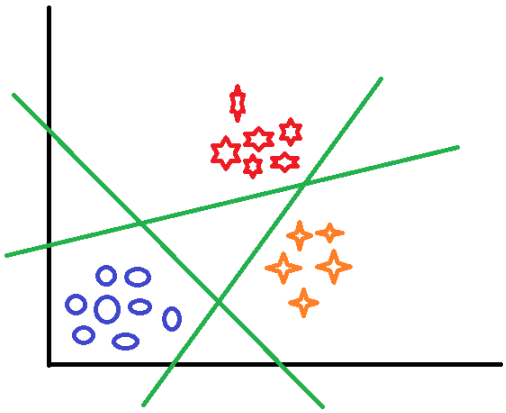
Alpha – 0.5
cost = 5.1651e-05
target output:0actual output0.00043246
target output:0actual output0.0092088
target output:0actual output0.01026
target output:1actual output0.98508



Learning rate 0.5 gives the best successive trial since it drops fast enough but also captures the local optima.

Task 12

- If we used a logistic regression units to solve the iris.m classification problem, then it would look like the diagram below. It can be solved by using one against all (OAA) logistic regression. Whereas, a neural network is a is a group of logistic regression units forming a network. So, when using a neural network, it can be solved in one go as they are connected set of logistic regression units/neurons.

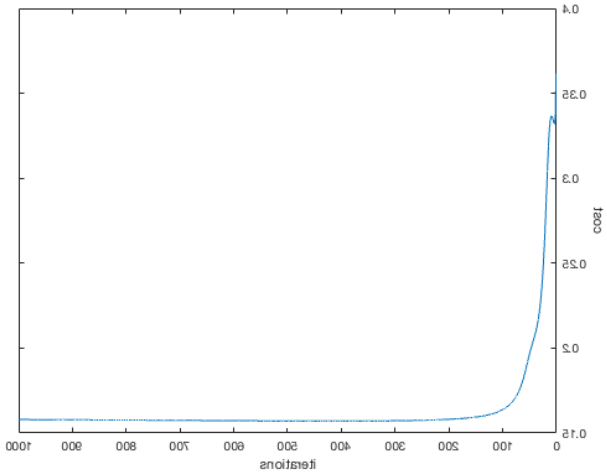


Task 13

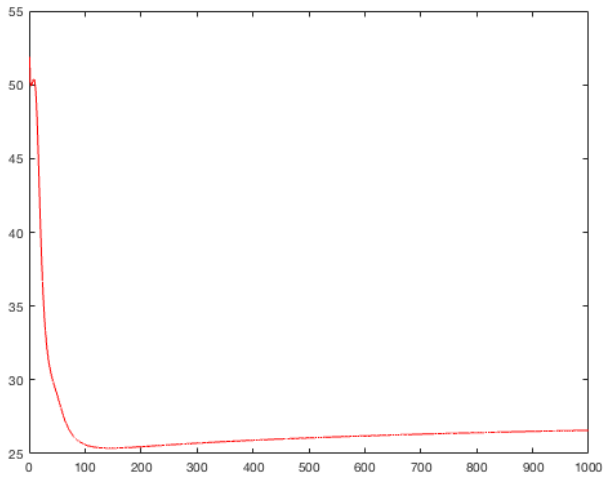
- Iris_example.m is run for different numbers of hidden neurons – 1,2 ,3 , 5, 7, 10. The training costs are plotted in blue and test cost in red.

For hidden neuron =1
Error training:23.7101
Error testing:26.5851
plotting errors

Training Cost Plot

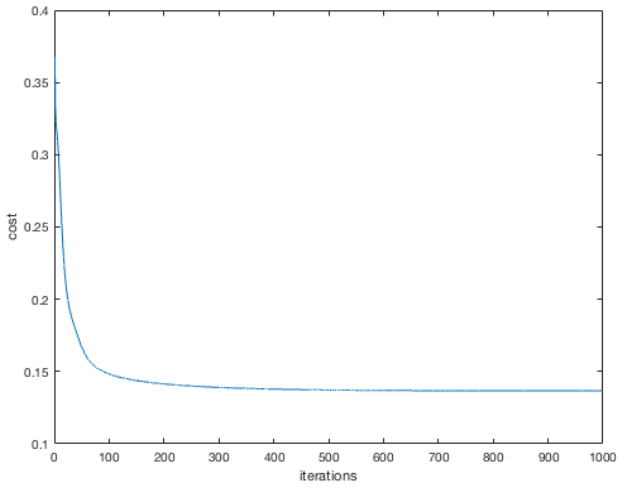


Test Cost Plot

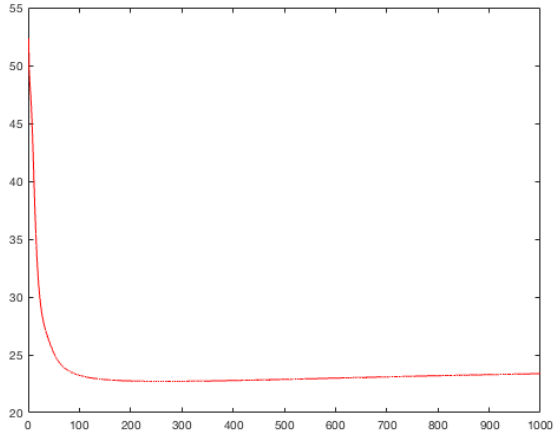


For hidden neuron =2(given)
Error training:20.4944
Error testing:23.3716
plotting errors

Training Cost Plot

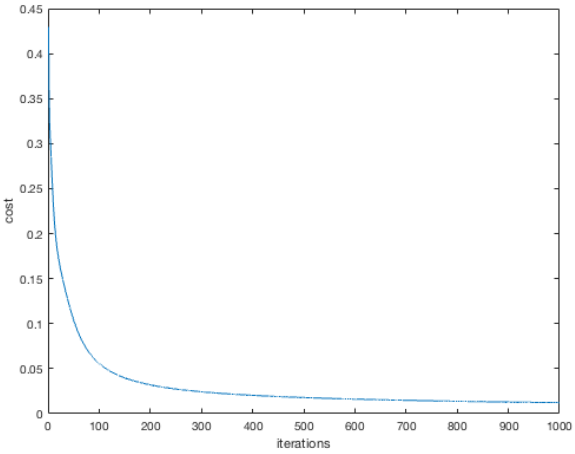


Test Cost Plot

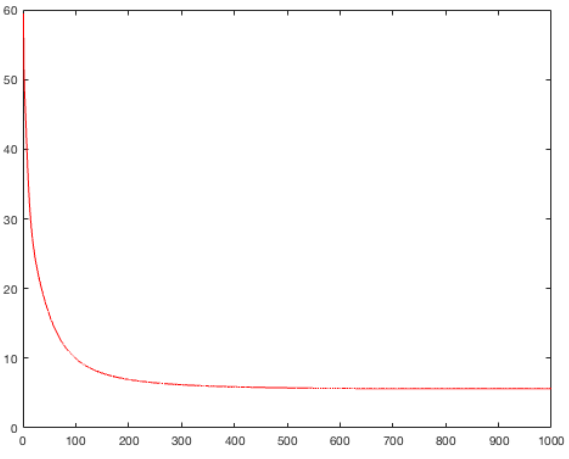


For hidden neuron = 3
Error training:1.8151
Error testing:5.6342
plotting errors

Training Cost Plot

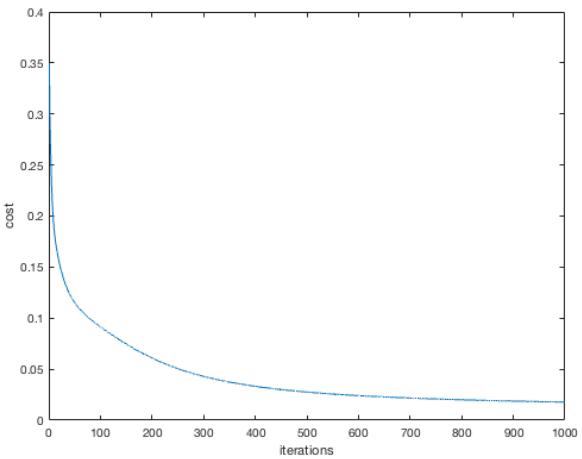


Test Cost Plot

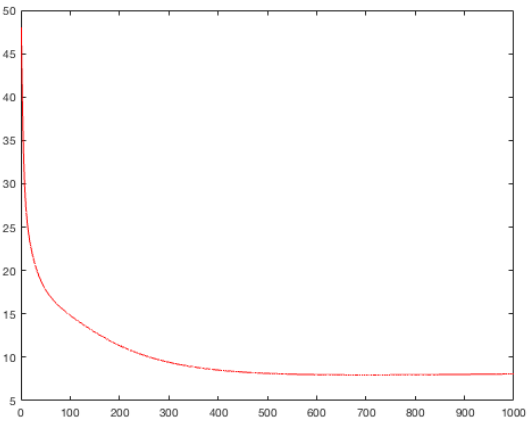


For hidden neuron = 5
Error training:2.6354
Error testing:8.0797
plotting errors

Training Cost Plot

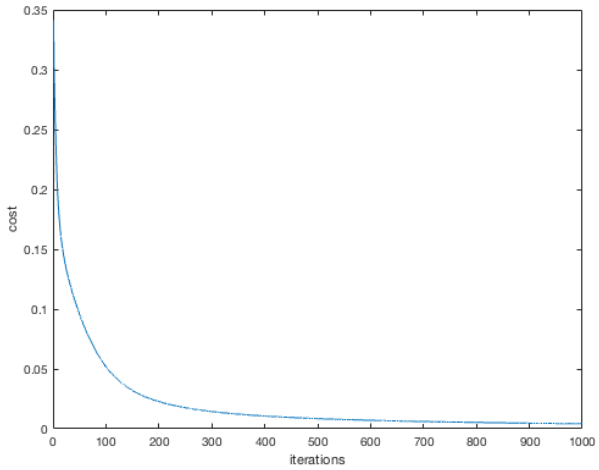


Test Cost Plot

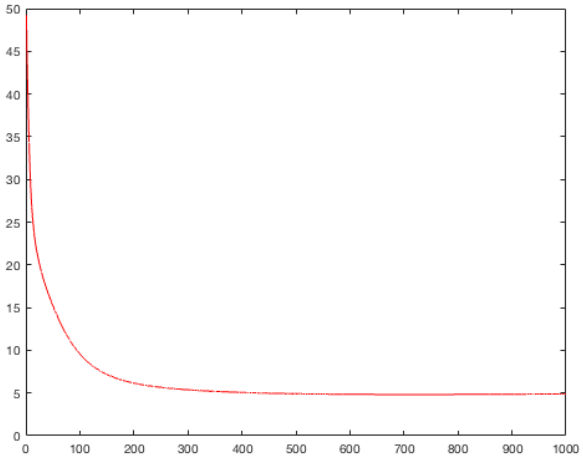


For hidden neuron = 7
Error training:0.61787
Error testing:4.8929
plotting errors

Training Cost Plot

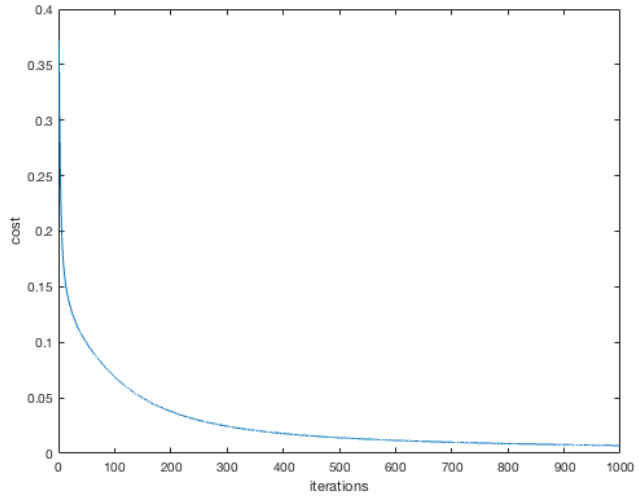


Test Cost Plot

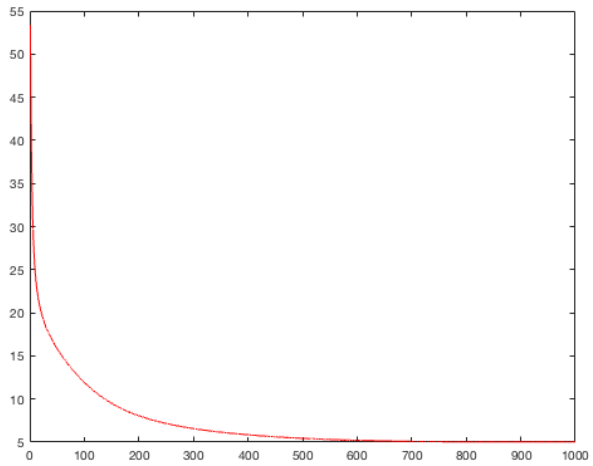


For hidden neuron = 10
Error training:1.0305
Error testing:5.0426
plotting errors

Training Cost Plot



Test Cost Plot



Since it is a 50% training and 50% test split, it can be seen from the graphs above that using 10 hidden neurons would give the best generalisation where both the training and test error are low. The differences between the different number of neurons is that the test error and train error decrease as we increase the hidden layers For, number 1, 2 the errors are very high. Whereas from 3 onwards, the error decreases.