Part 1 Linear Regression

1. Linear regression with one variable

Task 1

As you can see the hypothesis is computed for a particular example in **line 5 on calculate_hypothesis.m**, The direction of the vectors is important (whether they were row or column vectors).

Lines 21 and 41 have been modified of gradient_descent.m to use the calculate_hypothesis.m function.

When modifying the learning rate it is noticed that even with a slight increase to **0.03** the step is too large a step in the wrong direction causing a match of a line that **doesn't fit the data at all**, in fact the result was a line with a very negative slope (in the range of -10^9 for the bias term and theta_1).
If instead one reduces the granularity of the search to say **0.001** convergence happens to the same result but at a much slower pace, the curve of cost / iteration is a smooth curve of reducing costs that reach the optimum, in the case of 0.01 this curve was a very steep drop corresponding the quick convergence.
The problem with setting a granularity to an even smaller rate, although would avoid missing the solution as in 0.03 case is that for a fixed number of iterations it may not reach the minimum cost and might estimate a suboptimal line (underfit).

2. Linear regression with Multiple variables

Task 2

2. First, as a minor step, the **gradient_descent.m** function was modified to not use the the do_plot flag and to instead just always plot. The **calculate_hypothesis.m** function was already general enough to handle any size example by using the vectorial form.
However the ad-hoc logic for computing the theta updates was computing each theta update independently. It now uses vectorial form to compute all the weight update summations in one go. by using a vectorial form for the theta parameters. This can be seen in **lines 26-35. Line 23-29 were commented** which was previously initialising and setting theta0 and theta_1 manually.
While tweaking the alpha value for convergence, It was noticed that the previous value of 0.01 was not
converging fast enough to yield a result after 100 iterations. Increasing the step to **0.3** allowed the gradient descent function to reach convergence.

As a result:
at 0.01
bias=2.158106e+05, theta1=6.138403e+04 theta2=2.027355e+04
at 0.1
bias=3.404036e+05, theta1=1.099127e+05, theta2=-5.931109e+03

with a step of 0.2 we are close to convergence
bias=3.404127e+05, theta1= 1.106239e+05, theta2=-6.642289e+03
with a step of 0.3 are are at convergence. With theta of

bias= 3.404127e+05 ,theta1= 1.106310e+05 , theta2=-6.649417e+03
all step sizes up till 1 result in the same coefficients being computed after which it overshoots.

one can now use these weights to make a prediction:
bias= 3.404127e+05 ,theta1= 1.106310e+05 , theta2=-6.649417e+03

lines 20-30 in mllab2.m make use of the supplied normalise_feature functions where the features were encoded as examples.

The data that was supplied was it outputs the followed matrix of normalised features:
1.0000 -0.7071 -0.7071
1.0000 0.7071 0.7071

Which in turn means the costs are:

House 1 (1650 sq2m,3 beds) costs: 266886.58
House 2 (3000 sq2m,4 beds) costs: 413938.74

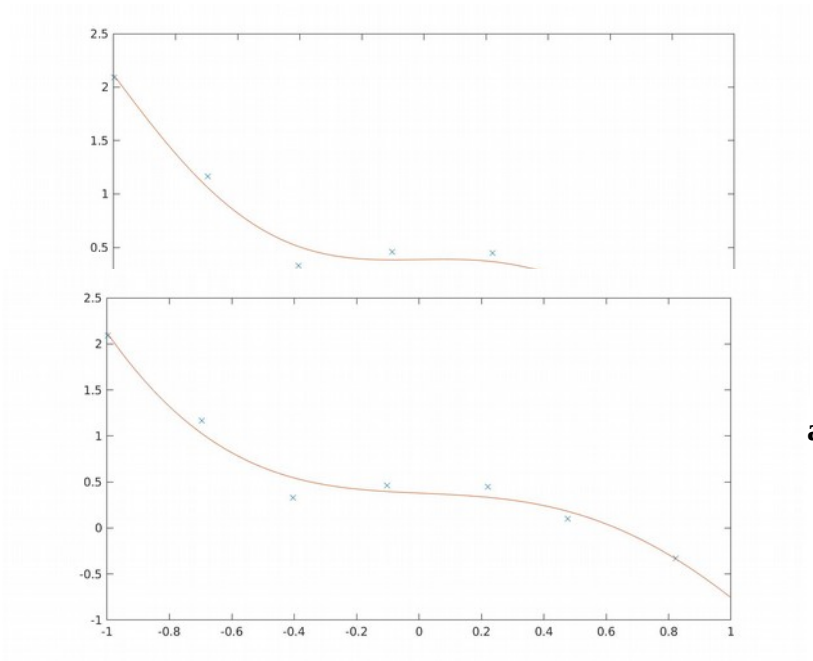
Which looks like a good prediction of pricing.

3. Regularised Linear regression
Task 3

In order to use the regularised cost function in our gradient descent, first I had to add a regularised parameter l to the gradient_descent.m as well as the do_plot flag from Task 1. I then had to change **line 54. In lines 21, 41,** it was necessary to add the bias term to our initial theta vector in **line 17 of mllab3.m.**

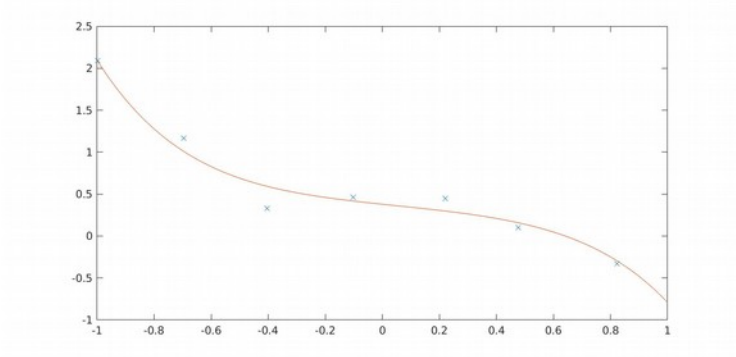
In order to incorporate the new method for updating weights for regularisation there **lines 23-47** were changed to add regularisation. The computation of theta_0 was extracted in order to avoid regularising on it. Regularisation was applied on the rest of the weights by slicing the thetas and input examples taking out the bias term.

After tweaking the alpha parameter we notice that the line of optimal fit was reached at **alpha=1.4**, after which the learning rate is too high and the cost curve minima was overshot.
At which **theta =0.3870 0.0462 -0.1238 -2.3851 0.5537 1.039**
INITIAL CURVE:

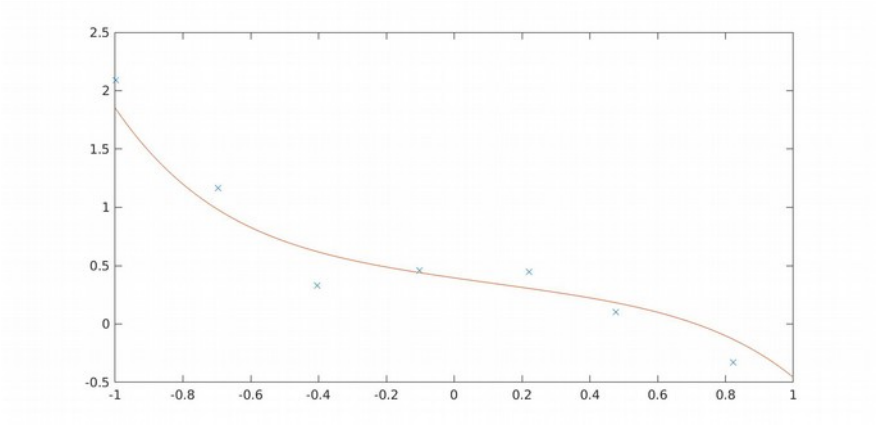


after adding l=0.01:

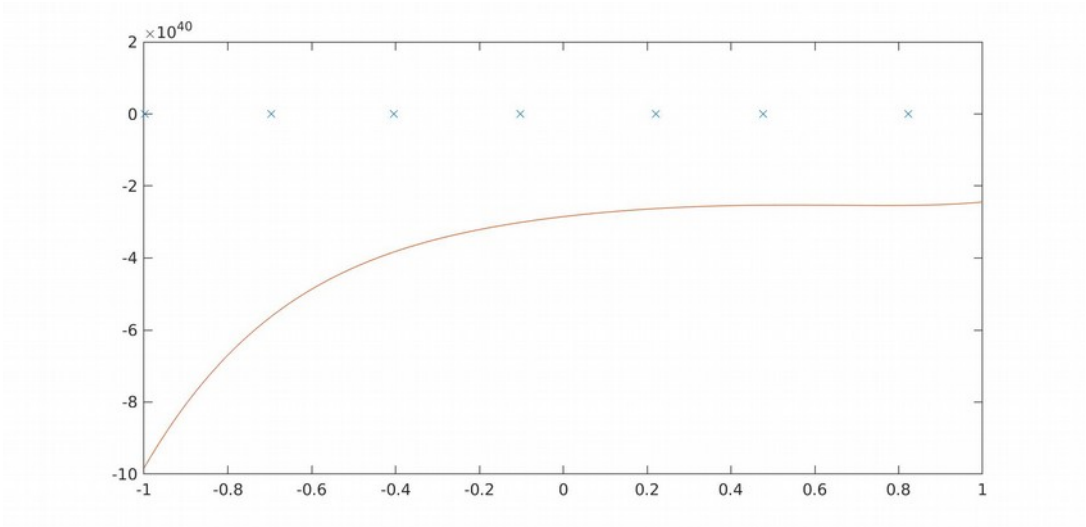
at l = 0.1



at l=1



at l =2 (underfit)



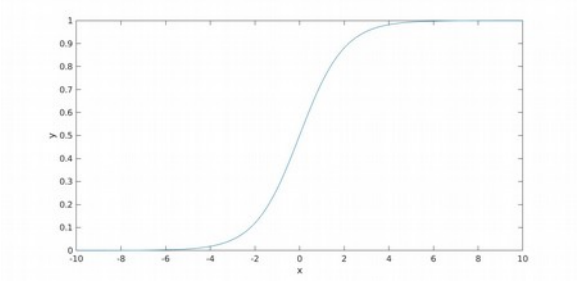
As you can see from the plots above one can **notice the effect of regularisation from l=0.1 and at 1** a significant regularisation to the weights has been made which should be robust to overfitting.

However when **l=2** too great a constraint was imposed on the weights and an **underfitting** situation occurs.

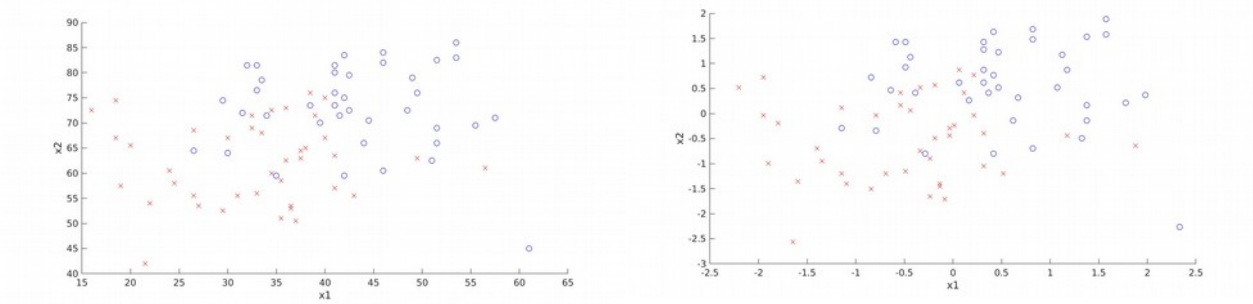
Part 2 Logistic regression and neural networks

1. Logistic regression

Task 1.
I have modified plot_sigmoid.m to compute the sigmoid of scalar input. This is the result:



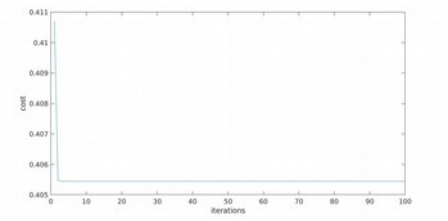
Task 2.
Plot with data not normalised: **Data normalised:**



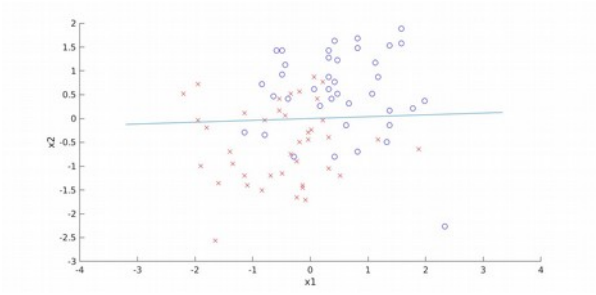
Task 3.
I have modified **calculate_hypothesis.m** to compute the hypothesis in **line 2.**

Task 4.
compute_cost.m was correct and no modification was needed.

Final error:0.40545



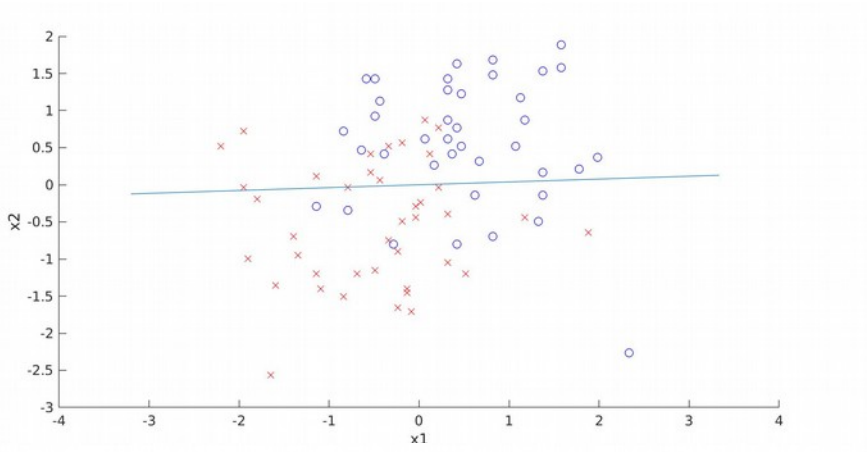
Task 5.
As described in question I have computed y1 corresponding to x1 was minimum and y2 for maximum x2, **lines 9-11 of plot_boundary.m, the result is the following line:**

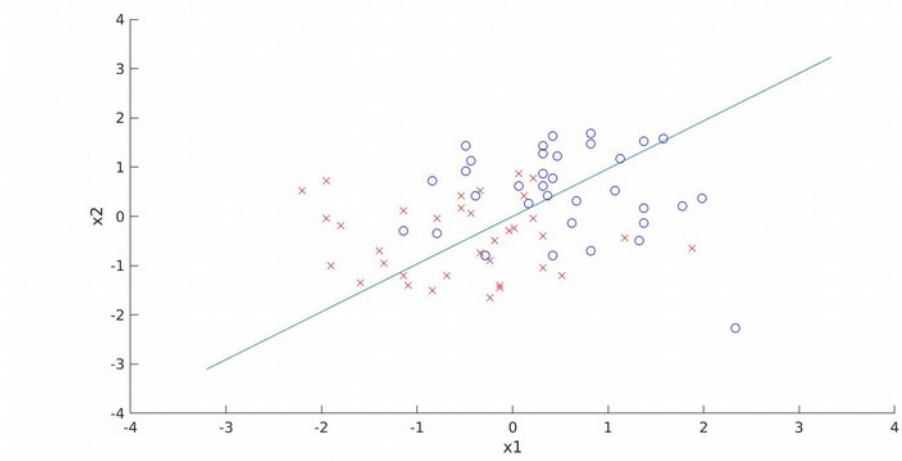


Task 6.

This run represents a good generalisation. It is not an overfitting curve. However it is unclear if the training error can get any lower.

Test error:0.69181
Training error:0.11993





The above situation illustrates a bad generalisation, also known as overfitting.

Training error:0.22767
Test error:0.59765

Task 7.

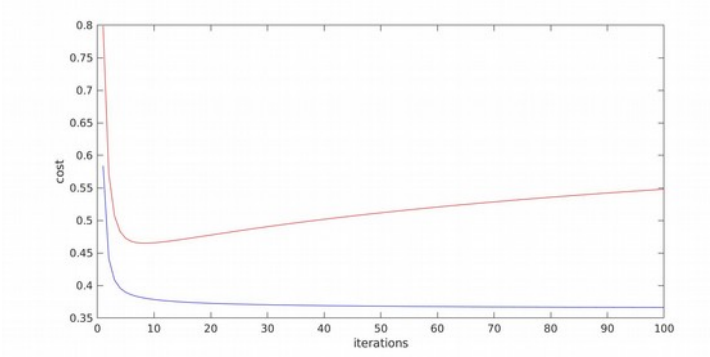
After enriching the examples with the derived parameters x1* x2, x1^2 and x2^2, modified in lines

we obtain a smaller error than before :

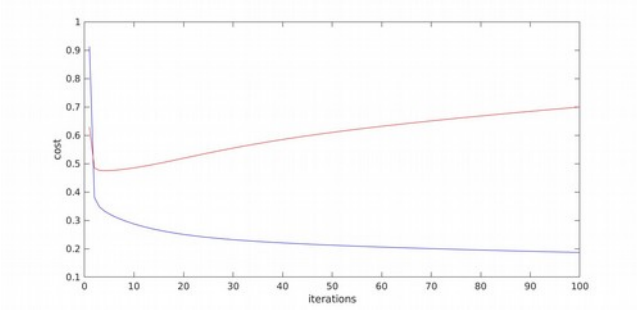
Error:0.39537

Although no new data was added our new parameters allow us to explore polynomial separation boundary to the data which has resulted in a better fit.

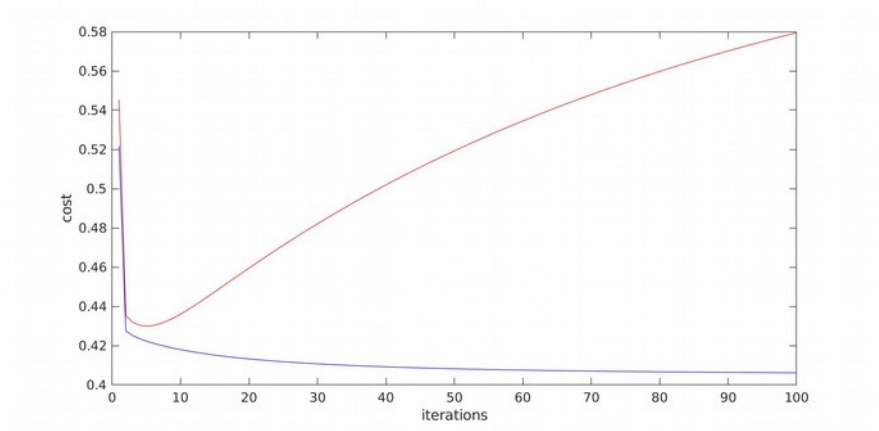
Task 8.



As you can see in the diagram as we are training our example with more data we are making the cost decrease for both until it starts to increase again for the test error this is due to the gradient descent overfitting on our data. This is with the initial split (60 training set 20 test set).
When the split is of 10 test set and 70 training the result is the following:



Which is the same sort of effect which reaches a slightly higher test error error of around 0.65, this is because we have more overfitting over more data points.
When we split with 30 test set and 50 training, we obtain the following curve:



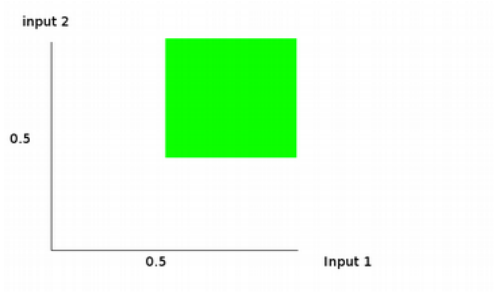
This graph seems similar to the rest but reaches a lesser error than the previous split, we have not overfitted the data .. it seems that the initial split of 60-20 was reaching an optimal of using more data points and hence achieving a good fit, and not overfitting as much as in the other cases.

They all seem to reach that peak after around 10 iterations which suggests that the accuracy can be improved by reducing the number of iterations.

Task 9.

We cannot solve the XOR problem because the plot for XoR is not linearly seperable

here's a way to visualise it.. Imagine a 2d plot where areas in green represent the target output or height.. If input encodes < 0.5 = 0 and > 0.5 = 1
Then for deciding the correct decision boundary the algorithm would need to come up with a line that must span the upper right rectangle from 0.5 to 1. which can never be a single line or plane.
The following shows that:



2. Neural network

Task 10.
Step 1. The correct method for computing the output layer delta is to multiple the error by the sigmoid derivative of the **pre-sigmoidal signal** of the output layer, however this derivative has a form of **g(a) * g(a) – 1,** this means that the parameter to sigmoid_derivative is the activation in (output_neurons), this can be seen in line 121.
Step 2.
As explained in hint I iterated over each neuron of the hidden layer recomputing the delta for that neuron by multiplying the sigmoid derivative of the pre-sigmoidal signal (recomputed using the helper function and the output of the neurons (a * a- 1) with a being g(a). The activation is stored in **hidden_neurons.**
The derivative is multiplied with the sum of output deltas times the weights on the connections from the neuron to the output neuron. The code in **124-139** implements that.

Step 3.
There is nothing to edit here because it seems that the code already does the right thing.
Step 4.
Similar to step 3 except iterating over inputs to update the hidden_weights, the changes are in **line 149-153**

2.1 Implement back-propagation on XOR

What is found is that the optimal learning rate for the pre-defined number of iterations is around **3-4**. if the learning rate is lesser than that then the ANN equivalent of under-fitting occurs even though the solutions are still well within the threshold of a XOR classifier.
In the best case the cost is:
cost = 0.00022251
target output:0actual output0.011358
target output:1actual output0.97915
target output:1actual output0.97911
target output:0actual output0.024659
with a very accurate XOR learned.

Occasionally due to the random element at initialisation, outputs being correct usually on ¾ of the test and a fixed cost curve stuck in a local minima like in the below cases:
this is a run stuck on a local minima it is wrong:
cost = 0.071202
target output:0actual output0.020912
target output:1actual output0.46647
target output:1actual output0.97902
target output:0actual output0.46672
If we instead set the learning rate to be very high say 10.
cost = 2.485e-05
target output:0actual output5.0209e-05
target output:1actual output0.99332
target output:1actual output0.99331
target output:0actual output0.010049
a very close fit is possible (but there is a danger of overfit) for this value of learning parameter the learner getting stuck in a bad configuration happens more frequently.

Task 11.

By setting the following training set to learn a NAND in NANDExample.m:

```
training_set_input = [
    0,0;
    0,1;
    1,0;
    1,1
];

training_set_output = [
    1;
    1;
    1;
    0
];
```

A good NAND gate function ANN is learned:

cost = 1.1367e-05
target output:1actual output0.99938
target output:1actual output0.99556
target output:1actual output0.99452
target output:0actual output0.0063798

2.2 Implement backpropagation on Iris

Task 12.

This machine learning problem requires a multi-label classifier, where the features are the attributes of the flower (sepal length, width..etc) and the target variable the class of type of flower.'

One can use a K-NN classifier or logistic regression for example.

There are two ways to use logistic regression for multi-class classification, being a 1-vs-all approach where a classification is learned for each class separately vs all the other classes. This is expensive and presents the problem of ambiguities. Instead what one can use is the softmax logistic regression algorithm which learns a matrix of coefficients one for each boundary. This means that the output layer has as many neurons as there are classes. This algorithm presents no ambiguities like the previous solution.

An artificial neural network has neurons each learning a LRU unit but then applies some non-linearity at the activation function and also chains such neurons to be able to learn arbitrarily complex functions. Any continuous function can be learnt.

Task 13.

When running **iris.m** with 1 hidden neuron one can see that the error is in the 50 range, the best run we had is bellow:
Error training:49.3299
Error testing:49.3617
There is little difference in the graph between these two errors. Except at the later stages of iterations.

When increasing to 2 hidden neurons, the overall error is lesser and the trend is still very similar between train and test error.
Error training:48.4119
Error testing:48.2418

At 3 hidden neurons the error is substantially smaller and the difference between train and test set is at 2. the plot is showing a similar evolution of the cost functions and the divergence happens in later iterations.
Error training:33.9361
Error testing:35.563

at 5 hidden neurons:
Error training:43.1221
Error testing:39.5296

at 7 hidden neurons:
Error training:36.5045
Error testing:35.7261

at 10 hidden neurons:
Error training:35.0806
Error testing:34.275

After many runs we obtain different results every time because of the randomness in initialisation but the trend is that adding **more hidden neurons increases overall accuracy** although this is not a linear relationship and at **10 hidden neurons the best generalisation is achieved**.