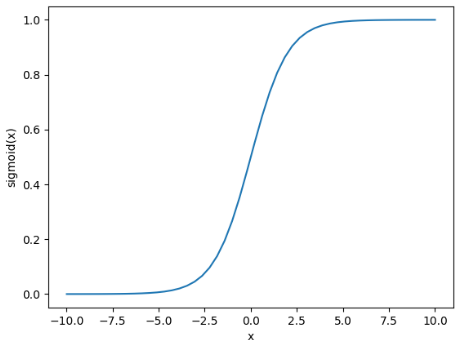
Neeraj Vashistha (190573735)

ECS708 Machine Learning

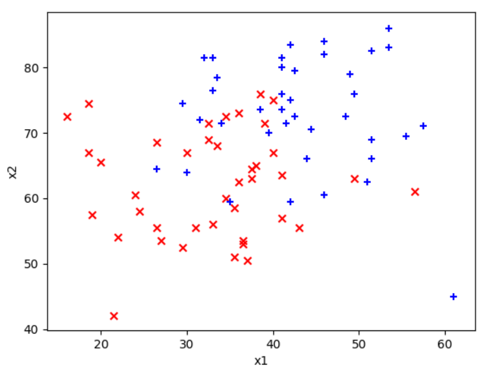
Assignment 1 – Part 2 - Logistic Regression and Neural Networks

Task 1: Include in your report the relevant lines of code and the result of the running the plot_sigmoid_function.py.

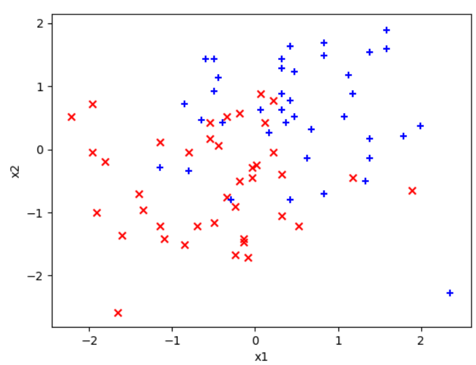
```
def sigmoid(z):
    output = 0.0
    output = 1. / (1. + np.exp(-z))
    return output
```



Task 2. Plot the normalized data to see what it looks like. Plot also the data, without normalization. Enclose the plots in your report.



Unnormalized X



normalized X

Task 3. Modify the calculate_hypothesis.m so that for a given dataset, theta and training example it returns the hypothesis.

```
    hypothesis = np.dot(X[i],theta)
    result = sigmoid(hypothesis)
```

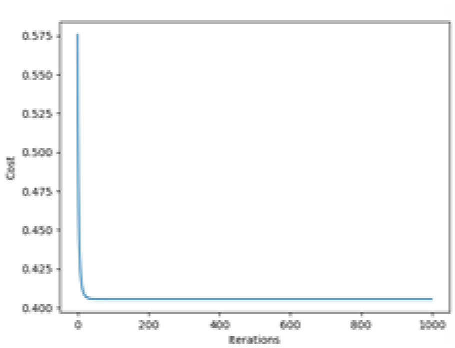
Task 4. Modify the line:

cost = 0.0 in compute_cost(X,y,theta so that it uses the cost function:

$$J(\theta) = \frac{1}{m}\sum_{\{i=1\}}^{m}[\left(-y^{(i)}\log\left(h_\theta\left(x^{(i)}\right)\right) - \left(1 - y^{(i)}\right)\log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right]$$

To calculate a logarithm, you can use log(x). Now run the file assgn1_ex1.py. What is the final cost found by the gradient descent algorithm? In your report include the modified code and the cost graph.

```
cost = ((-1 * output * np.log(hypothesis)) - ((1 - output) * np.log(1 - hypothesis)))

alpha = 1
```

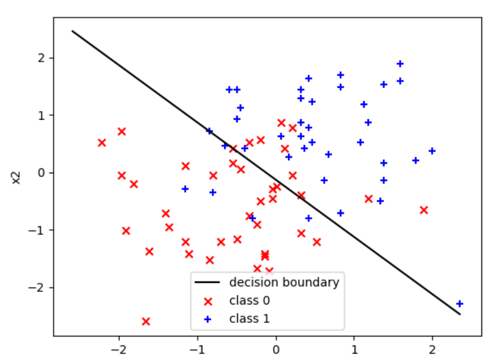



```
Dataset normalization complete.
Gradient descent finished.
Minimum cost: 0.40545, on iteration #306

The best cost the model could attain was at alpha =1 and it came out to be 0.40545
```

Task 5. Plot the decision boundary. This corresponds to the line where $\theta^T x = 0$

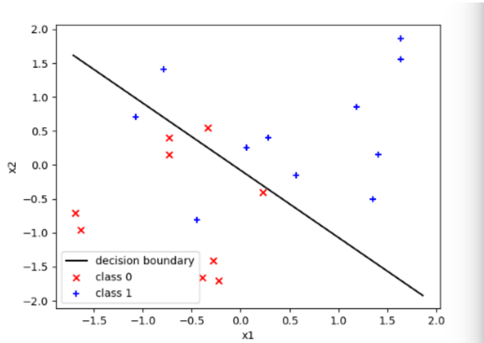
For example, if $h_\theta = \theta_1 x_1 + \theta_2 x_2$ the boundary is where $\theta_1 x_1 + \theta_2 x_2 = 0$

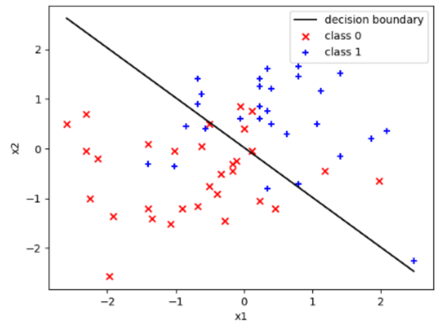
Rearrange the equation in terms of x_2 and in the plot function set y1 equal to x_2 when x_1 is at the minimum in the data set and set y2 equal to x_2 when x_1 is at its maximum in the data set. Uncomment the relevant plot function in lab2_lr_ex1.m and include the graph in your report.



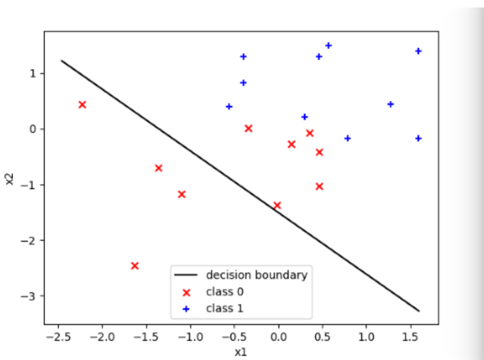
Task 6. Run the code in lab2_lr_ex2.m several times.

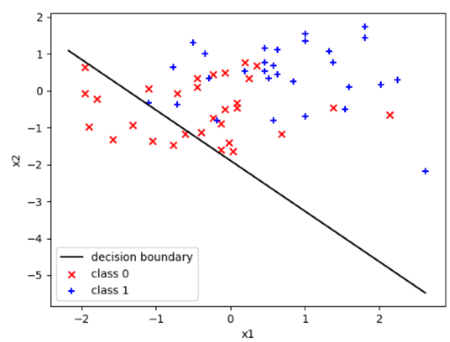
What is the general difference between the training and test error? When does the training set generalize well? Demonstrate two splits with good and bad generalisation and put both graphs in your report.






```
Dataset normalization complete.
Dataset normalization complete.
Gradient descent finished.
Final training cost: 0.35370
Minimum training cost: 0.35370, on iteration #100
Final test cost: 0.45516
```



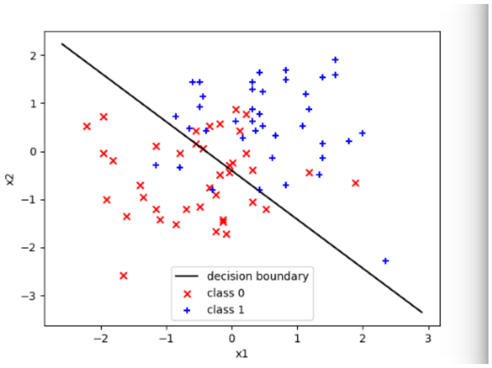


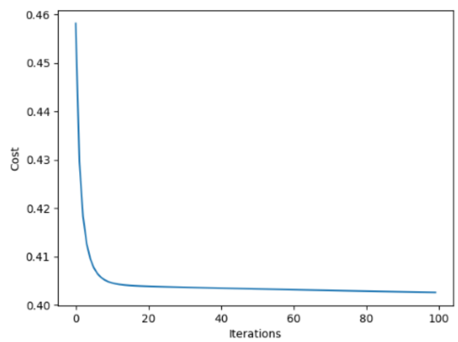

```
Dataset normalization complete.
Dataset normalization complete.
Gradient descent finished.
Final training cost: 0.15269
Minimum training cost: 0.15269, on iteration #100
Final test cost: 0.69570
```

The first graph is an example of good generalization when the training and test cost are comparable and do not have much difference whereas in the second graph the training and test cost have a very big difference. In the first graph the data points are well generated and spread over and across whereas in second graph we see a very small amount of training data and sparsely distributed this is the main reason of discrepancy seen above.

In lab2_lr_ex3a.m, instead of using just the 2D feature vector, incorporate the following non-linear features: $x_1 * x_2$ x_1^2 and x_2^2. This results in a 5D input vector per data point, and so you must use 6 parameters θ.

Task 7. Run logistic regression on this dataset. How does the error compare to using the original features (i.e. the error found in Task 4)? Include in your report the error and an explanation on what happens.





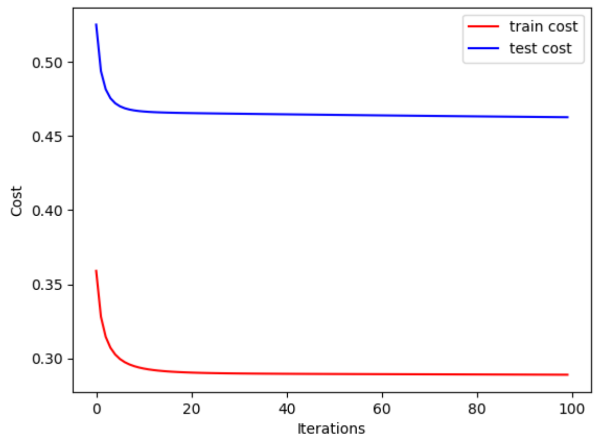

```
Dataset normalization complete.
Gradient descent finished.
Final cost: 0.40261
Minimum cost: 0.40261, on iteration #100
```

We see a reduction in cost when we increase the number of features, as previously seen (in task 4) the cost has stagnated at 0.40450 for alpha 1 but when we increased the number of features the cost has decreased.

Task 8. In lab2_lr_ex3b.m the data is split into a test set and a training set. Add your new features from the question above. Modify the function gradient_descent_training() to store the current cost for the training set and testing set. Store the cost of the training set to cost_array_training and for the test set to cost_array_test

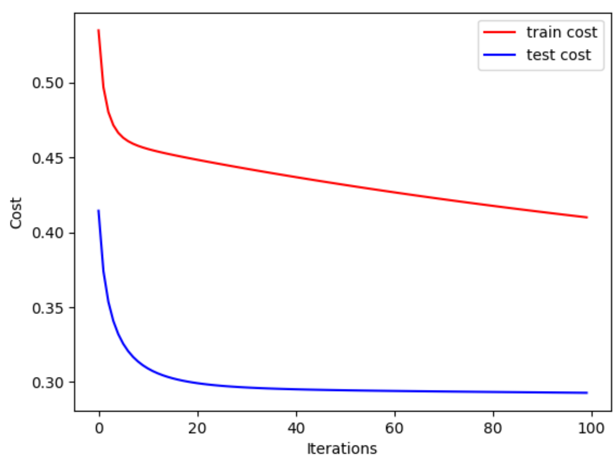
These arrays are passed to plotdata2(), which will show the cost function of the training (in blue) and test set (in red). Experiment with different sizes of training and test set (remember that the total data size is 80) and show the effect of using sets of different sizes by saving the graphs and putting them in your report. Add extra features (e.g. a third order polynomial) and analyse the effect. What happens when the cost function of the training set goes down but that of the test set goes up?

Features = X1, X2, X1^2, X2^2, X1*X2, test-train split = 30-70



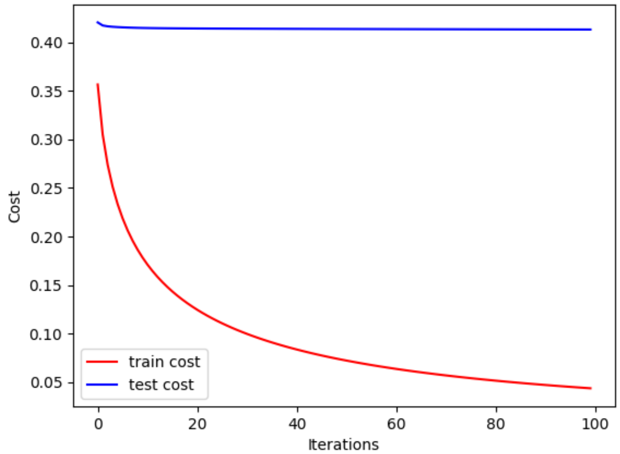

```
Gradient descent finished.
Final train cost: 0.28895
Minimum train cost: 0.28895, on iteration #100
Final test cost: 0.46263
Minimum test cost: 0.46263, on iteration #100
---------------------
```

Features = X1, X2, X1^2, X2^2, X1*X2, test-train split = 20-80




```
Gradient descent finished.
Final train cost: 0.41004
Minimum train cost: 0.41004, on iteration #100
Final test cost: 0.29286
Minimum test cost: 0.29286, on iteration #100
---------
```

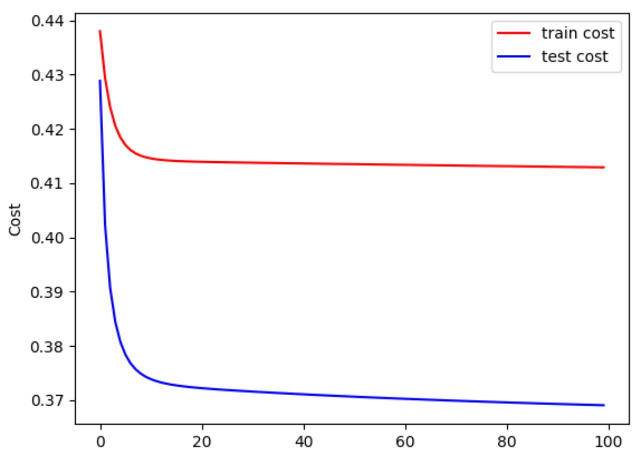
Features = X1, X2, X1^2, X2^2, X1*X2, test-train split = 10-90




```
Gradient descent finished.
Final train cost: 0.04365
Minimum train cost: 0.04365, on iteration #100
Final test cost: 0.41330
Minimum test cost: 0.41330, on iteration #100
```

From above we see that if we provide adequate amount of data for training then test cost is less(test train split 20-80), no over fitting but as we increase the data in training we see overfitting and it generalizes less on test data(test train split 10-90). In test train split 10-90, we see train cost function goes down and test function is very high, this is due to the fact that model fails to generalize on test data and model is overfitting on train data.

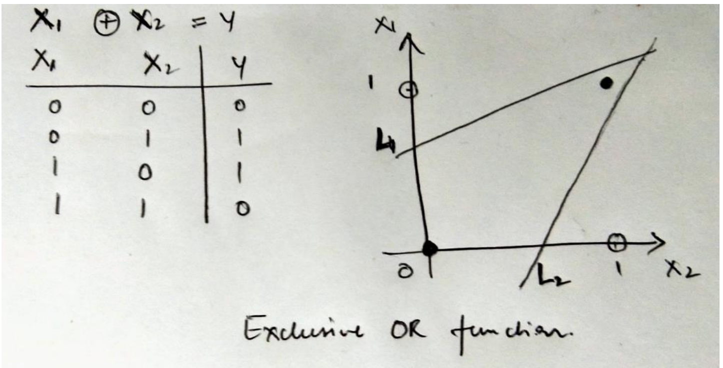
Features = X1, X2, X1^2, X2^2, X1*X2, **X1^3, X2^3**, test-train split = 20-80




```
Gradient descent finished.
Final train cost: 0.41292
Minimum train cost: 0.41292, on iteration #100
Final test cost: 0.36907
Minimum test cost: 0.36907, on iteration #100
```

We see that the test and train cost has decreased as we increase the number of features as relatively sufficient amount of data is given and good generalization can be seen on test data as well.

Task 9. With the aid of a diagram of the decision space, explain why a logistic regression unit cannot solve the XOR classification problem.



The Logistic regression cannot find a single separable line for XOR classification problem, here we have two lines L1 and L2 trying to separate dots (when y=0) and circles (when y =1) but in this decision tree we cannot find single line which can separate these points into right classes.

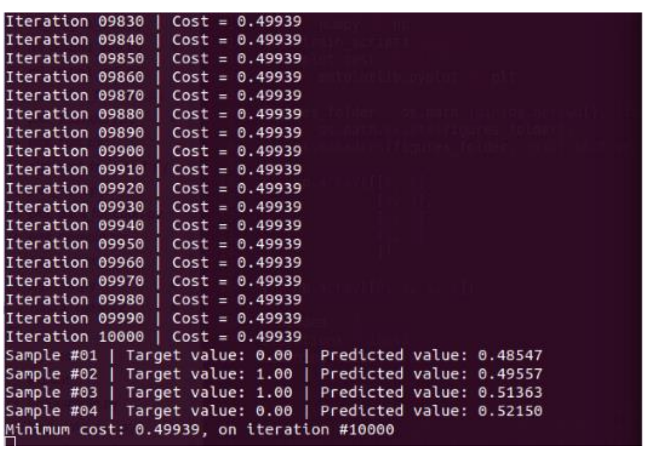
Task 10. Implement backpropagation. Although XOR only has one output, this should support outputs of any size.

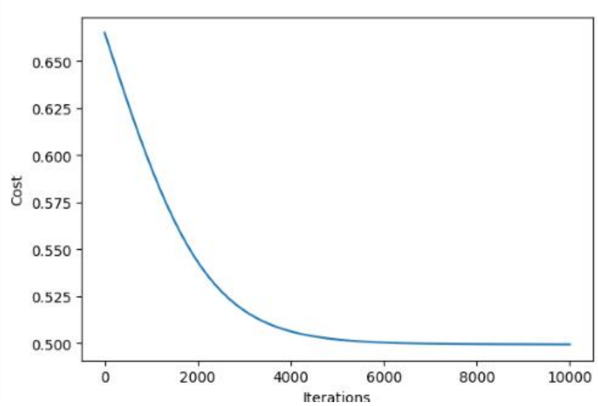
```
def backward_pass(self, inputs, targets, learning_rate):
    # We will backpropagate the error and perform gradient descent on the network weights
    # We compute the error between predictions and targets
    J = 0.5 * np.sum( np.power(self.y_out - targets, 2) )

    # append term that was multiplied with the hidden layer's bias
    inputs = np.append(1, inputs)

    # Step 1. Output deltas are used to update the weights of the output layer
    # print(type(targets))
    # sys.exit(0)
    output_deltas = np.zeros((self.n_out))
    outputs = self.y_out.copy()
    for i in range(self.n_out):
        # compute output_deltas : delta_k = (y_k - t_k) * g'(x_k)
        # output_deltas[i] = ...
        if type(targets) is np.int64:
            output_deltas[i] = (outputs[i] - targets) * sigmoid_derivative(outputs[i])
        else:
            if len(targets) > 1:
                output_deltas[i] = (outputs[i] - targets[i]) * sigmoid_derivative(outputs[i])

    # Step 2. Hidden deltas are used to update the weights of the hidden layer

    hidden_deltas = np.zeros((len(self.y_hidden)))

    # Create a for loop, to iterate over the hidden neurons.
    # Then, for each hidden neuron, create another for loop, to iterate over the output neurons
    for j in range(len(hidden_deltas)):
```

```
        hidden_error_weight = 0.0
        for k in range(self.n_out):
            hidden_error_weight += self.w_out[j,k]* output_deltas[k]
        hidden_deltas[j] = sigmoid_derivative(self.y_hidden[j]) * hidden_error_weight

    # Step 3. update the weights of the output layer

    for i in range(len(self.y_hidden)):
        for j in range(len(output_deltas)):
            # update the weights of the output layer
            self.w_out[i,j] = self.w_out[i,j] - (learning_rate * output_deltas[j] *
self.y_hidden[i]) #sigmoid(self.y_hidden[i]))

    # we will remove the bias that was appended to the hidden neurons, as there is no
    # connection to it from the hidden layer
    # hence, we also have to keep only the corresponding deltas
    hidden_deltas = hidden_deltas[1:]

    # Step 4. update the weights of the hidden layer
    # Create a for loop, to iterate over the inputs.
    # Then, for each input, create another for loop, to iterate over the hidden deltas

    for i in range(len(inputs)):
        for j in range(len(hidden_deltas)):
            # update the weights of the hidden layer
            # self.w_hidden[i,j] = self.w_hidden[i,j] - (learning_rate * hidden_deltas[j] *
sigmoid_derivative(inputs[i]))
            self.w_hidden[i,j] = self.w_hidden[i,j] - (learning_rate * hidden_deltas[j] * inputs[i])
#sigmoid(inputs[i]))
    return J
```

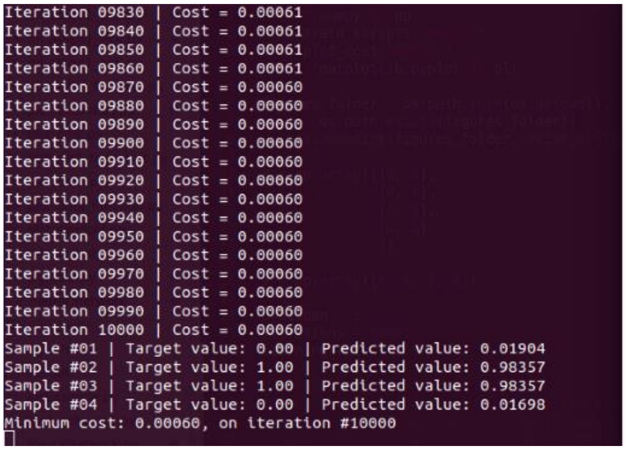
Your task is to implement backpropagation and then run the file with different learning rates (loading from xorExample.py).

Alpha = 0.001

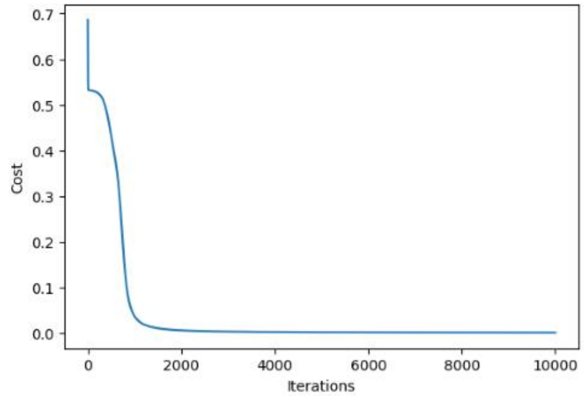




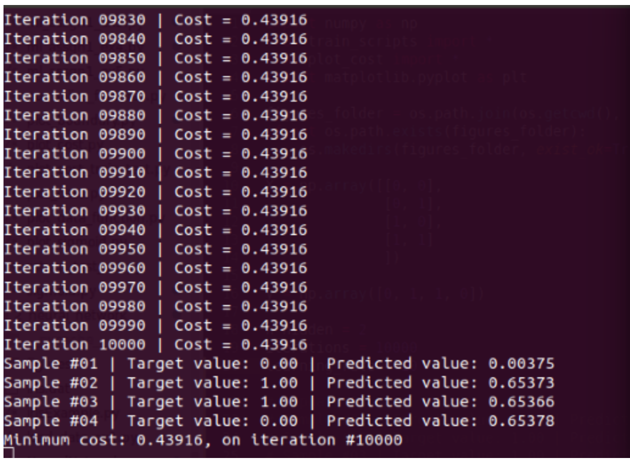
Alpha = 0.5



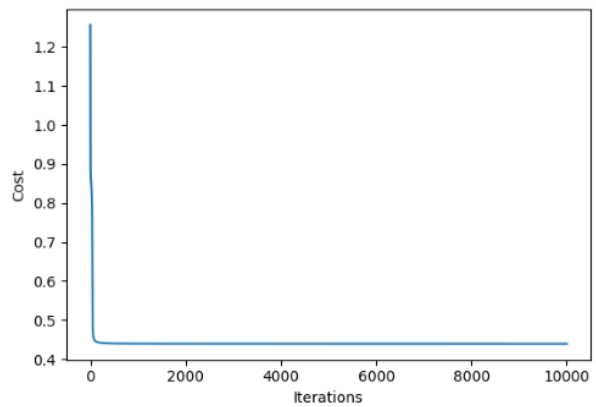
```
Iteration 09830 | Cost = 0.00061
Iteration 09840 | Cost = 0.00061
Iteration 09850 | Cost = 0.00061
Iteration 09860 | Cost = 0.00061
Iteration 09870 | Cost = 0.00060
Iteration 09880 | Cost = 0.00060
Iteration 09890 | Cost = 0.00060
Iteration 09900 | Cost = 0.00060
Iteration 09910 | Cost = 0.00060
Iteration 09920 | Cost = 0.00060
Iteration 09930 | Cost = 0.00060
Iteration 09940 | Cost = 0.00060
Iteration 09950 | Cost = 0.00060
Iteration 09960 | Cost = 0.00060
Iteration 09970 | Cost = 0.00060
Iteration 09980 | Cost = 0.00060
Iteration 09990 | Cost = 0.00060
Iteration 10000 | Cost = 0.00060
Sample #01 | Target value: 0.00 | Predicted value: 0.01904
Sample #02 | Target value: 1.00 | Predicted value: 0.98357
Sample #03 | Target value: 1.00 | Predicted value: 0.98357
Sample #04 | Target value: 0.00 | Predicted value: 0.01698
Minimum cost: 0.00060, on iteration #10000
```



Alpha = 10



```
Iteration 09830 | Cost = 0.43916
Iteration 09840 | Cost = 0.43916
Iteration 09850 | Cost = 0.43916
Iteration 09860 | Cost = 0.43916
Iteration 09870 | Cost = 0.43916
Iteration 09880 | Cost = 0.43916
Iteration 09890 | Cost = 0.43916
Iteration 09900 | Cost = 0.43916
Iteration 09910 | Cost = 0.43916
Iteration 09920 | Cost = 0.43916
Iteration 09930 | Cost = 0.43916
Iteration 09940 | Cost = 0.43916
Iteration 09950 | Cost = 0.43916
Iteration 09960 | Cost = 0.43916
Iteration 09970 | Cost = 0.43916
Iteration 09980 | Cost = 0.43916
Iteration 09990 | Cost = 0.43916
Iteration 10000 | Cost = 0.43916
Sample #01 | Target value: 0.00 | Predicted value: 0.00375
Sample #02 | Target value: 1.00 | Predicted value: 0.65373
Sample #03 | Target value: 1.00 | Predicted value: 0.65366
Sample #04 | Target value: 0.00 | Predicted value: 0.65378
Minimum cost: 0.43916, on iteration #10000
```



What learning rate do you find best? Include a graph of the error function in your report. Note that the backpropagation can get stuck in local optima. What are the outputs and error when it gets stuck?

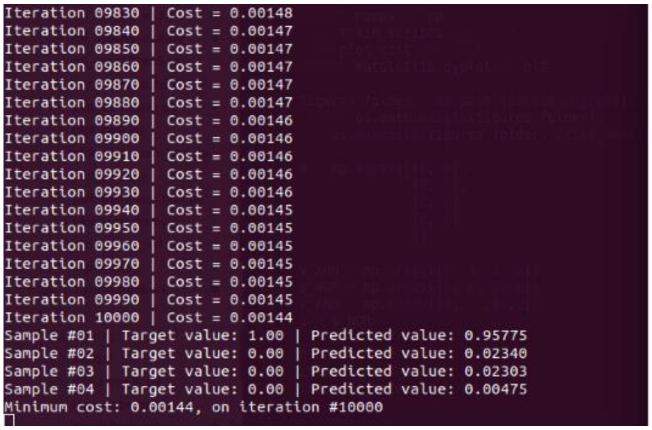
When alpha is 0.5 we get minimum cost of around 0.00060

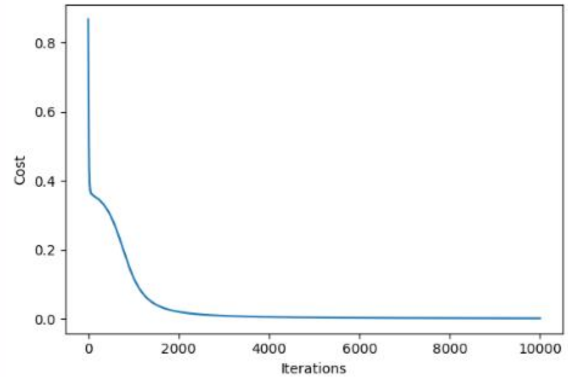
When learning rate is very small (0.001) then we see backpropagation can get stuck in local optima.

When the learning rate is big (10) then it fails gradient descent never reaches local optima as it overshoots.

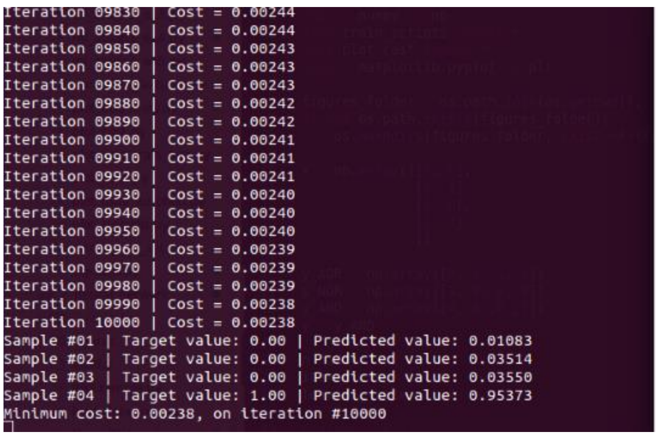
Task 11. Change the training data in xor.py to implement a different logical function, such as NOR or AND. Plot the error function of a successful trial

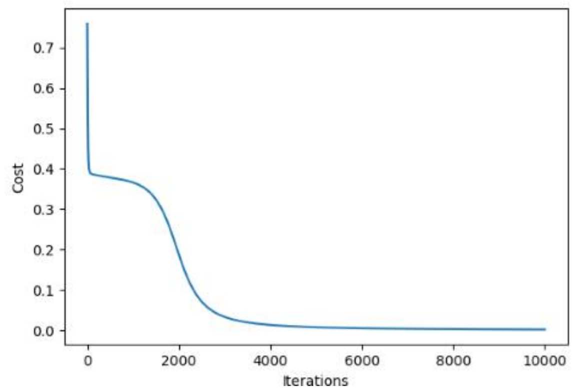
For NOR, alpha = 0.1





For AND, alpha = 0.1





Task 12. The Iris data set contains three different classes of data that we need to discriminate between. How would accomplish this if we used a logistic regression unit? How is it different using a neural network?

Multi-Label Classification is the generalization of logistic regression, here we can use One-vs-all technique for each of the n classes in the training dataset. In other words, we train for one class and treat other classes as opposite this means we apply logistic regression n times for n classes.

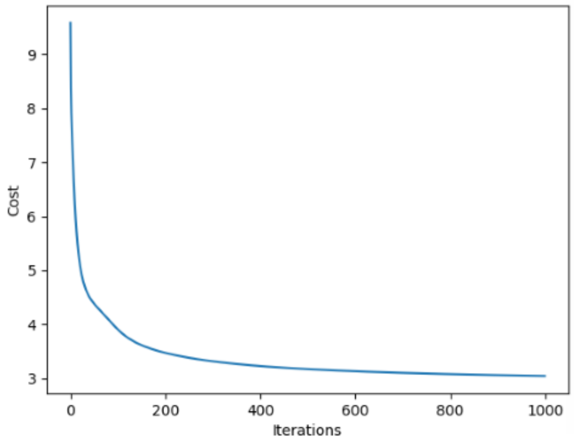
In neural network we define y-label classes and create network based on simple non-linear functions as linear problems can be solved using logistic regression.

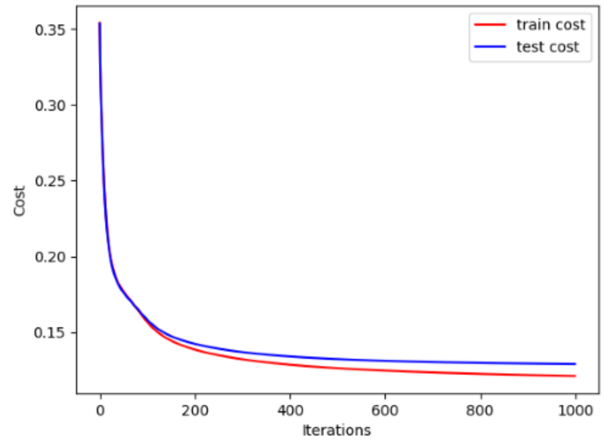
Task 13. Run iris.py using the following number of hidden neurons: 1, 2, 3, 5, 7, 10. The program will plot the costs of the training set (blue) and test set (red) at each iteration. What are the differences for each number of hidden neurons? Which number do you think is the best to use? How well do you think that we have generalized?

As we increase the number of neurons, the cost or the lost function decreases. With a constant learning rate of 0.1 and 1000 iterations we see the cost decreases sharply initially but then stabilises as we increase the number of hidden neurons.

With hidden neuron = 2 and alpha value = 0.1 and iteration =1000 the model has an admissible cost around 0.04 with good generalisation.

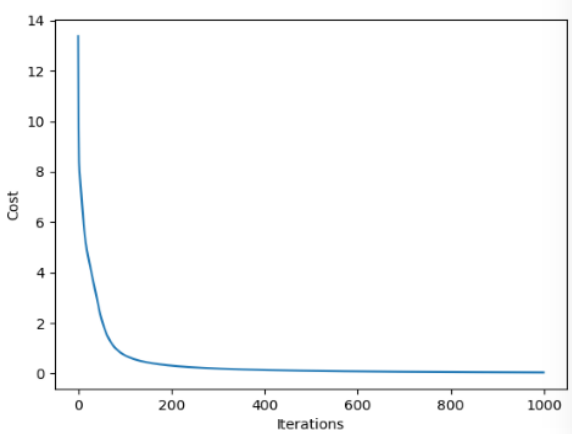
```
Alpha = 0.1 hidden neuron =1, iterations = 1000
```

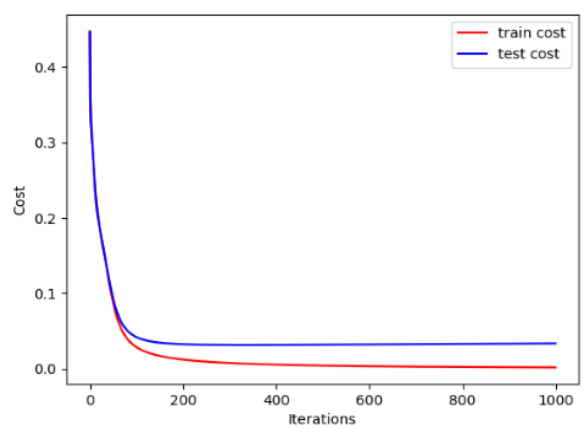





```
Testing on Iris dataset...
Sample #01 | Target value: 0.00 | Predicted value: 0.00000
Sample #02 | Target value: 0.00 | Predicted value: 0.00000
Sample #03 | Target value: 0.00 | Predicted value: 0.00000
Sample #04 | Target value: 0.00 | Predicted value: 0.00000
Sample #05 | Target value: 0.00 | Predicted value: 0.00000
Sample #26 | Target value: 1.00 | Predicted value: 1.00000
Sample #27 | Target value: 1.00 | Predicted value: 1.00000
Sample #28 | Target value: 1.00 | Predicted value: 1.00000
Sample #29 | Target value: 1.00 | Predicted value: 1.00000
Sample #30 | Target value: 1.00 | Predicted value: 1.00000
Sample #51 | Target value: 2.00 | Predicted value: 2.00000
Sample #52 | Target value: 2.00 | Predicted value: 2.00000
Sample #53 | Target value: 2.00 | Predicted value: 2.00000
Sample #54 | Target value: 2.00 | Predicted value: 2.00000
Sample #55 | Target value: 2.00 | Predicted value: 2.00000
Minimum cost: 3.04657, on iteration #1000
```

```
Alpha = 0.1 hidden neuron =2, iterations = 1000
```

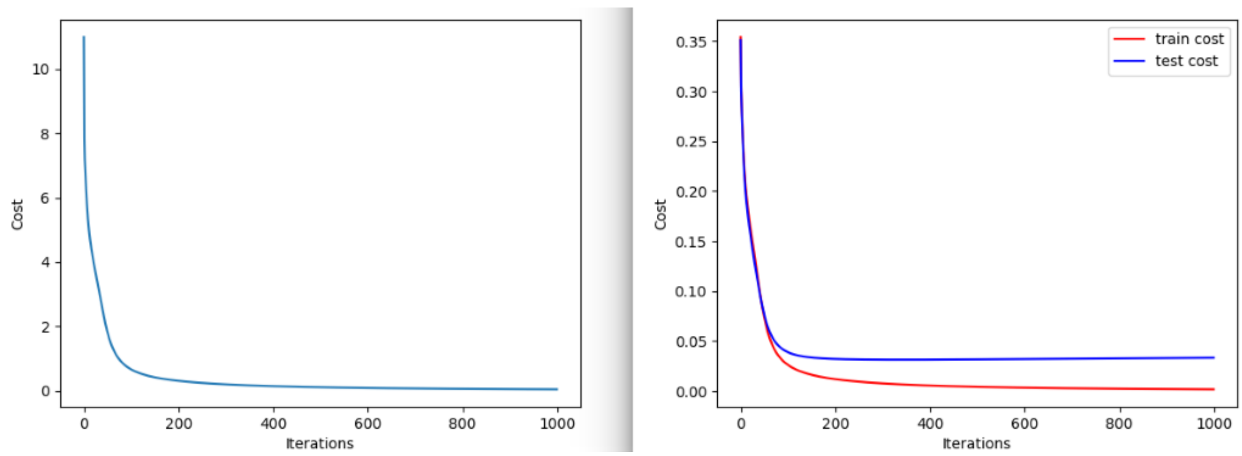





```
Testing on Iris dataset...
Sample #01 | Target value: 0.00 | Predicted value: 0.00000
```

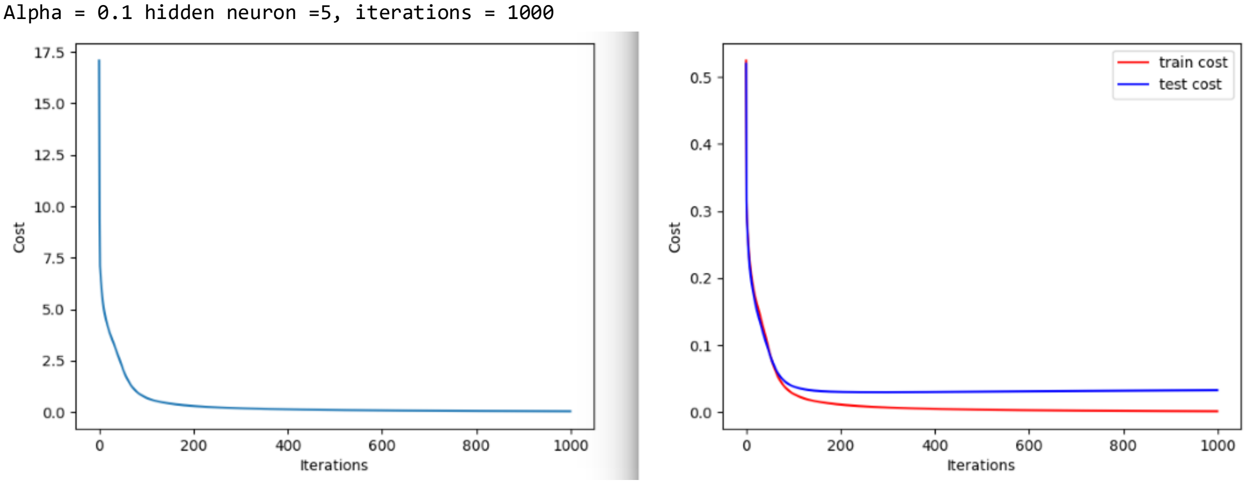
```
Sample #02 | Target value: 0.00 | Predicted value: 0.00000
Sample #03 | Target value: 0.00 | Predicted value: 0.00000
Sample #04 | Target value: 0.00 | Predicted value: 0.00000
Sample #05 | Target value: 0.00 | Predicted value: 0.00000
Sample #26 | Target value: 1.00 | Predicted value: 1.00000
Sample #27 | Target value: 1.00 | Predicted value: 1.00000
Sample #28 | Target value: 1.00 | Predicted value: 1.00000
Sample #29 | Target value: 1.00 | Predicted value: 1.00000
Sample #30 | Target value: 1.00 | Predicted value: 1.00000
Sample #51 | Target value: 2.00 | Predicted value: 2.00000
Sample #52 | Target value: 2.00 | Predicted value: 2.00000
Sample #53 | Target value: 2.00 | Predicted value: 2.00000
Sample #54 | Target value: 2.00 | Predicted value: 2.00000
Sample #55 | Target value: 2.00 | Predicted value: 2.00000
Minimum cost: 0.04330, on iteration #1000
```

```
Alpha = 0.1 hidden neuron =3, iterations = 1000
```



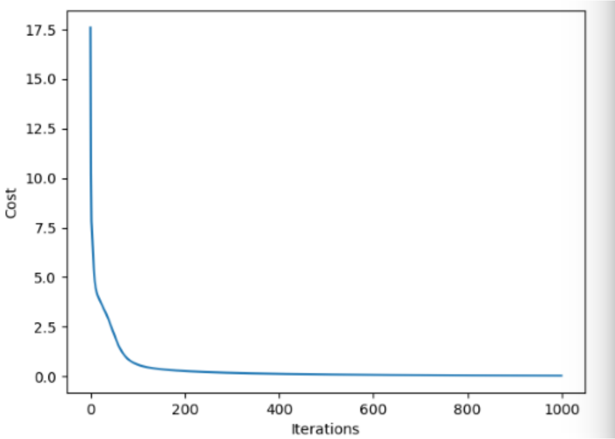

```
Minimum cost: 0.04637, on iteration #1000
```

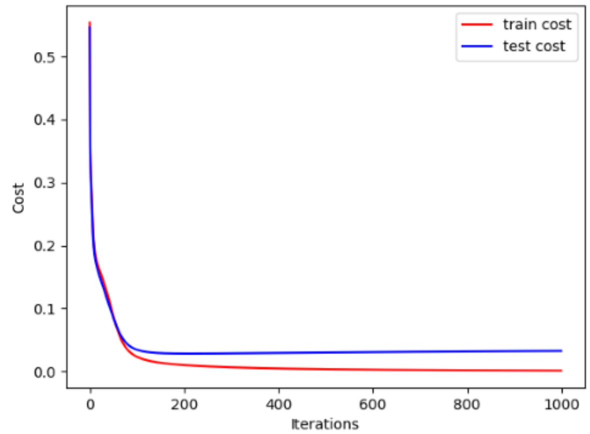
```
Alpha = 0.1 hidden neuron =5, iterations = 1000
```



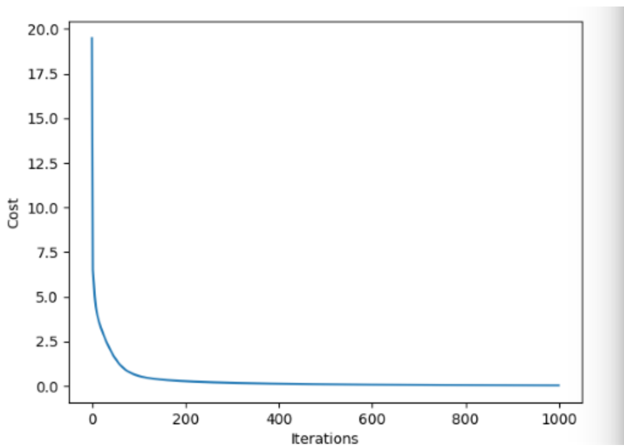

```
Minimum cost: 0.04075, on iteration #1000
```

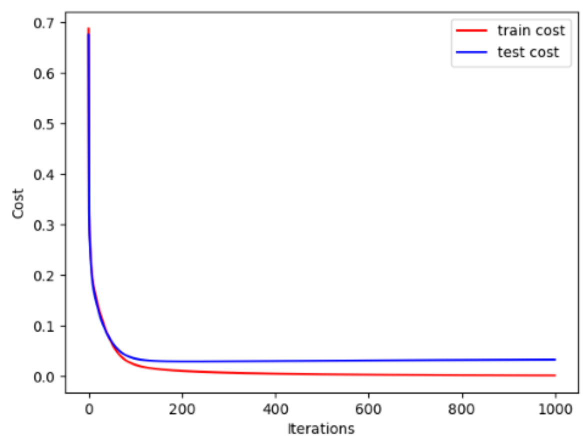
```
Alpha = 0.1 hidden neuron =7, iterations = 1000
```






```
Minimum cost: 0.03838, on iteration #1000

Alpha = 0.1 hidden neuron =10, iterations = 1000
```






```
Minimum cost: 0.03861, on iteration #1000
```