MACHINE LEARNING ASSIGNMENT - 1 [Part -2]

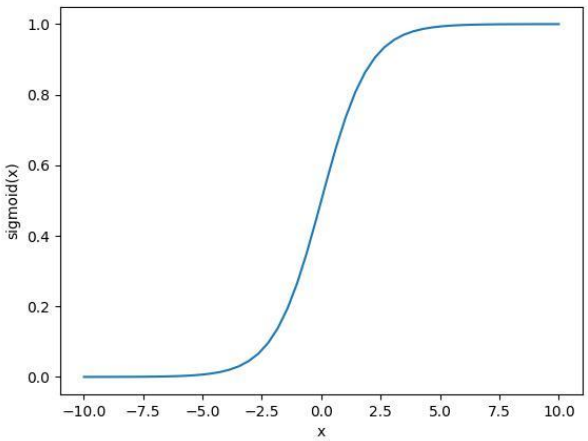
Task 1

Fill out the sigmoid.py function. Now use the plot_sigmoid.py function to plot the sigmoid function. Include in your report the relevant lines of code and the result of the using plot_sigmoid.py.

sigmoid.py

```
from math import e
def sigmoid(z):
output = 1 / (1 + e**(-z))

return output
```

Output of plot_sigmoid.py

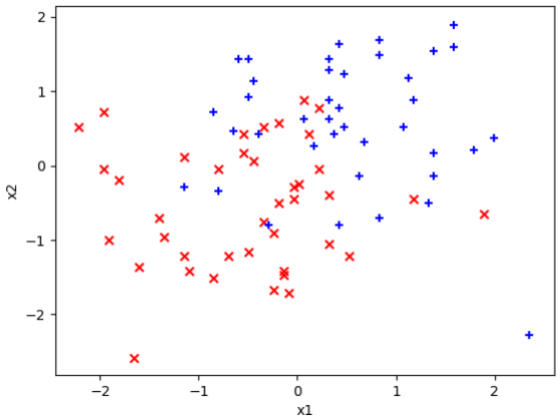


The above lines of code were included in sigmoid.py and output of running plot_sigmoid.py is shown in illustration.

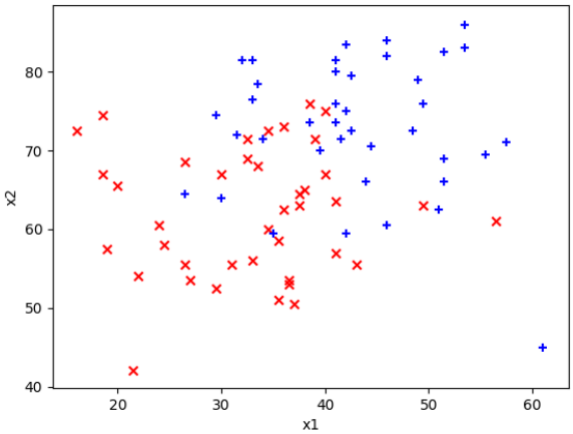
Task 2

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

Normalized Data points



Without Normalization



In case of data being normalized, the x1 and x2 attributes scaling is very well decreased.

In contrary, when the data is not normalized, x1 and x2 attributes has a wide range of values without proper scaling as illustrated.

Task 3

Modify the calculate_hypothesis.py function so that for a given dataset, theta and training example it returns the hypothesis. For example, for the dataset X=[[1,10,20],[1,20,30]] and for Theta = [0.5,0.6,0.7], the call to the function calculate_hypothesis(X,theta,0) will return: sigmoid(1 * 0.5 + 10 * 0.6 + 20 * 0.7) The function should be able to handle datasets of any size. Enclose in your report the relevant lines of code.

```
def calculate_hypothesis(X, theta, i):
    """
        :param X            : 2D array of our dataset
        :param theta        : 1D array of the trainable parameters
        :param i            : scalar, index of current training sample's row
    """
    hypothesis = 0.0
    ########################################
    # Write your code here
    # You must calculate the hypothesis for the i-th sample of X, given X, theta
and i.
    for j in range(len(theta)):
        hypothesis = hypothesis + theta[j] * X[i, j]
    ########################################/
    result = sigmoid(hypothesis)

    return result
```

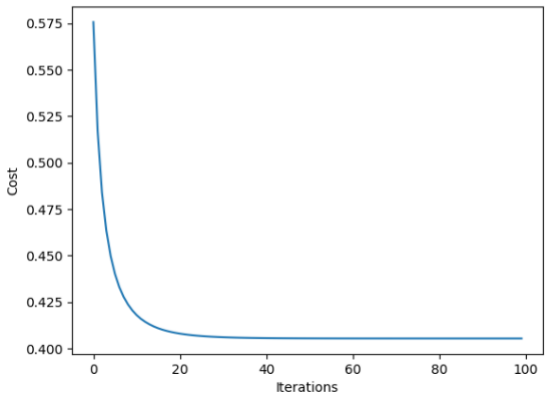
The above code can handle datasets of any size ('i' is the reference to the tuple in a dataset) and can handle any number of variables (ie. attributes) with the help of variable 'j'.

Task 4.

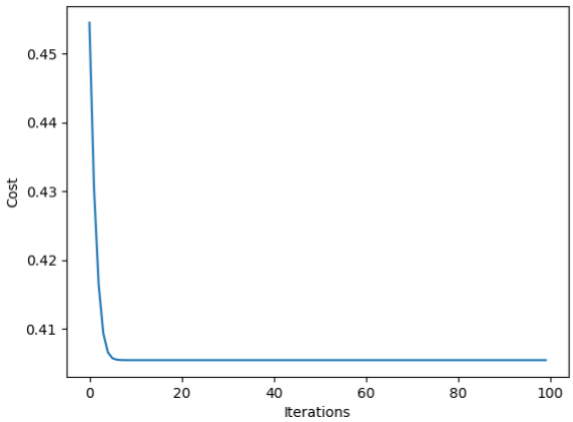
Modify the line "cost = 0.0" in compute_cost.py so that we can use our cost function. To calculate a logarithm you can use np.log(x). Now run the file assgn1_ex1.py. Tune the learning rate, if necessary. What is the final cost found by the gradient descent algorithm? In your report include the modified code and the cost plot.

```
def compute_cost(X, y, theta):
    """
        :param X            : 2D array of our dataset
        :param y            : 1D array of the groundtruth labels of the dataset
        :param theta        : 1D array of the trainable parameters
    """

    # initialize cost
    J = 0.0
    # get number of training examples
    m = y.shape[0]

    # Compute cost for logistic regression.
```

```
    for i in range(m):
        hypothesis = calculate_hypothesis(X, theta, i)
        output = y[i]
        cost = 0.0
        ########################################
        # Write your code here
        # You must calculate the cost
        cost = (-1 * y[i] * np.log(hypothesis))-((1 - y[i]) * np.log(1-
               hypothesis))

        ########################################/
        J += cost
    J = J/m

    return J
```



Cost graph when alpha = 1



Cost graph when alpha = 10

- When alpha = 1 the final cost found by the gradient descent is 0.40545.
- When alpha = 10 the final cost found by the gradient descent is 0.40545.

-→ alpha = 10 is considered better because we get the minimum cost at iteration 37 whereas for alpha = 1 we get the minimum cost at iteration 100.

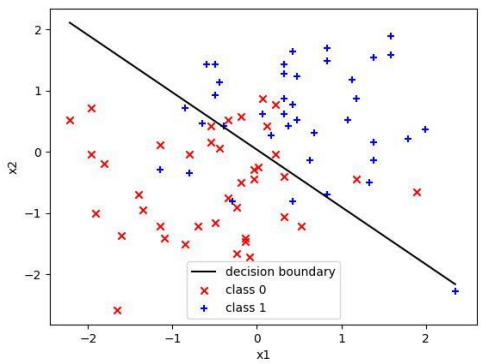
Illustrations shows the output of running assgn1_ex1.py for various values of alpha.

Task 5

Plot the decision boundary. This corresponds to the line where $\theta^T x = 0$, which is the boundary line's equation. To plot the line of the boundary, you'll need two points of (x_1 ,x_2). Given a known value for x_1 , you can find the value of x_2 . Rearrange the equation in terms of x_2 to do that. Use the minimum and maximum values of x_1 as the known values, so that the boundary line that you'll plot, will span across the whole axis of x_1 . For these values of x_1 , compute the values of x_2 . Use the relevant plot_boundary function in assgn1_ex1.py and include the graph in your report.

```
def plot_boundary(X, theta, ax1):

    min_x1 = 0.0
    max_x1 = 0.0
    x2_on_min_x1 = 0.0
    x2_on_max_x1 = 0.0

    ##########################################
    # Write your code here
    # Re-arrange the terms in the equation of the hypothesis function, and solve
with respect to x2, to find its values on given values of x1
    m = X.shape[0]
    for i in range(m):
       if min_x1 > X[i, 1]:
          min_x1 = X[i, 1]

       if max_x1 < X[i, 1]:
          max_x1 = X[i, 1]


    ########################################/
    x2_on_min_x1 = ((-1 * theta[0]) - (theta[1] * min_x1))/theta[2]
    x2_on_max_x1 = ((-1 * theta[0]) - (theta[1] * max_x1))/theta[2]
    x_array = np.array([min_x1, max_x1])
    y_array = np.array([x2_on_min_x1, x2_on_max_x1])
    ax1.plot(x_array, y_array, c='black', label='decision boundary')

    # add legend to the subplot
    ax1.legend()
```

Graph with decision boundary



The graph depicts the plotted boundary line using plot boundary function.

Task 6

Run the code of assgn1_ex2.py several times. In every execution, the data are shuffled randomly, so you'll see different results. Report the costs found over the multiple runs. What is the general difference between the training and test cost? When does the training set generalize well? Demonstrate two splits with good and bad generalisation and put both graphs in your report.
In assgn1_ex3.py, instead of using just the 2D feature vector, incorporate non-linear features.

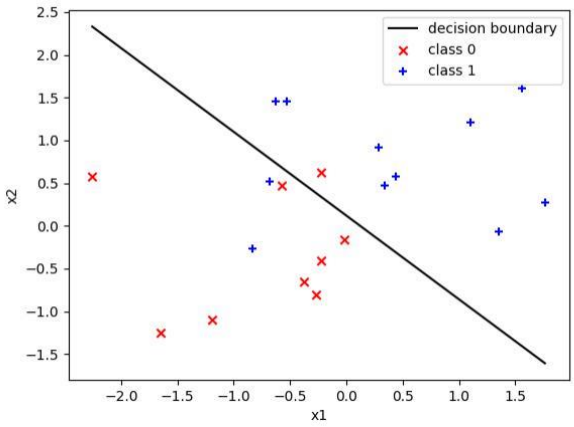
a)
Dataset normalization complete.
Gradient descent finished.
Final training cost: 0.40925
Minimum training cost: 0.40925, on iteration #100
Final test cost: 0.41689

Dataset normalization complete.
Gradient descent finished.
Final training cost: 0.37944
Minimum training cost: 0.37944, on iteration #100
Final test cost: 0.43008

Dataset normalization complete.
Gradient descent finished.
Final training cost: 0.40714
Minimum training cost: 0.40714, on iteration #100
Final test cost: 0.47164

Running assgn1_ex2.py multiple times gives different test and training costs at each run, this is because of the random shuffling of data at every execution.
Gradient descent is obtained using the training data and the weights obtained are applied again on training data, the error that we get now with respect to true value is training error and when the weights obtained are applied on test data which gives the test error.
Training set generalizes well when there is a higher degree polynomial order which can fit the training data well.

Good split [train = 60, test = 20]



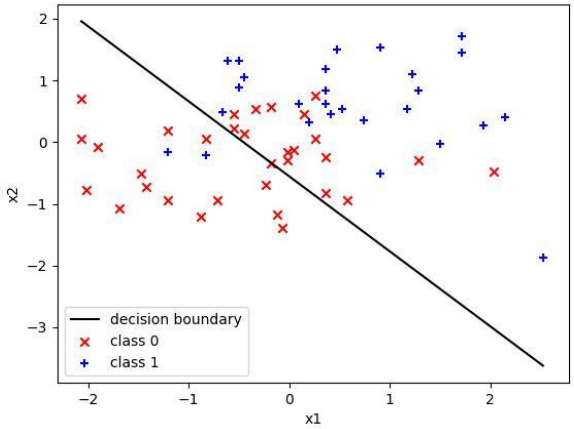
Final training cost: 0.42357
Minimum training cost: 0.42357, on iteration #100
Final test cost: 0.35708

Given graph shows the test dataset classification

Bad split [train = 20, test = 60]

Final training cost: 0.31227
Minimum training cost: 0.31227, on iteration #100
Final test cost: 0.56725

Given graph shows the test dataset classification.

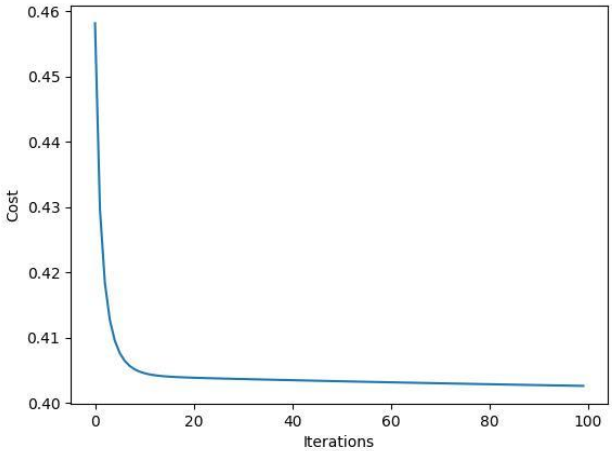


b) Incorporating non linear features.

```
# This loads our data
X, y = load_data_ex1()

# Create the features x1*x2, x1^2 and x2^2
##########################################
# Write your code here
# Compute the new features
# Insert extra singleton dimension, to obtain Nx1 shape
# Append columns of the new features to the dataset, to the dimension of columns
(i.e., 1)
m = X.shape[0]
a1 = [None] * m
a2 = [None] * m
a3 = [None] * m
for i in range(m):
   a1[i] = X[i, 0] * X[i, 1]
   a2[i] = X[i, 0] * X[i, 0]
   a3[i] = X[i, 1] * X[i, 1]

##########################################/
a1 = np.asarray(a1)
a1 = a1.reshape(X.shape[0],1)
a2 = np.asarray(a2)
a2 = a2.reshape(X.shape[0],1)
a3 = np.asarray(a3)
a3 = a3.reshape(X.shape[0],1)
X = np.append(X, a1, axis=1)
X = np.append(X, a2, axis=1)
X = np.append(X, a3, axis=1)
print(X)
```

Task 7

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



Cost graph for non-linear features with alpha = 1

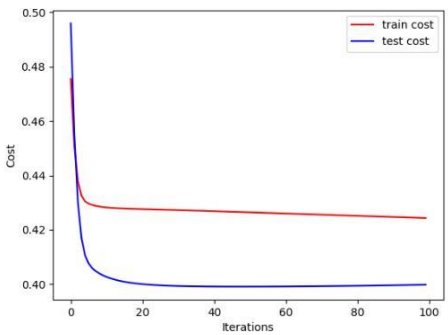
The minimum cost for nonlinear features is 0.40261 for alpha = 1.
The minimum cost for linear features is 0.40545 for alpha = 1.

Addition of nonlinear parameters results in better fitting. But the anomaly is least able to find the cost involved.

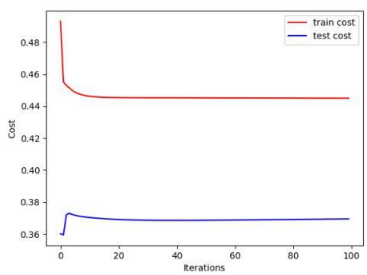
Task 8.

a)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. b)In the file assgn1_ex5.py, add extra features (e.g. both a second-order and 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?

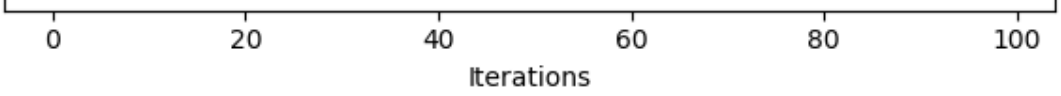
a)

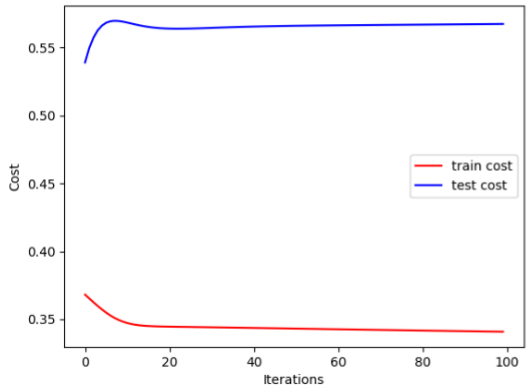


Train = 20 Test = 60

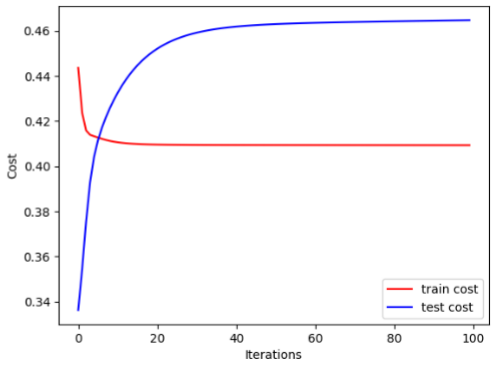


Train = 40 Test = 40





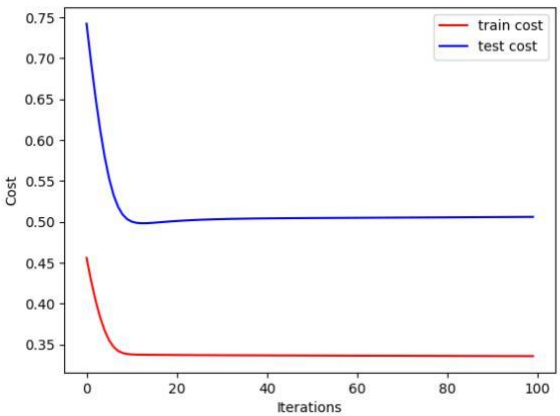
Train = 50 Test = 30



Train = 60 Test = 20

As training set size is increasing, the test cost is also increasing because of overfitting of data.

b)

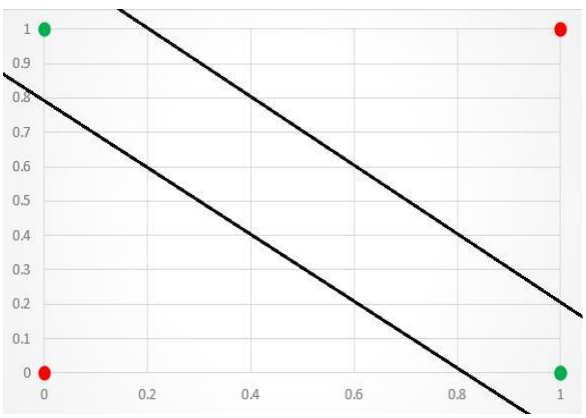


For 2nd and 3rd order polynomial

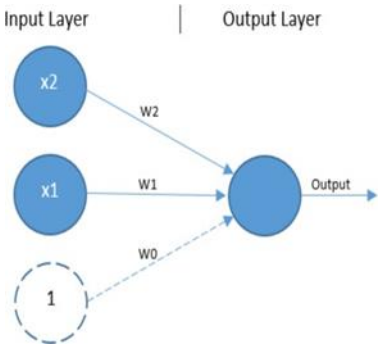
Figure shows the cost graph for training set of size 40 and test set of size 40 for a polynomial of order 2 and 3. When the polynomial order increases, the training dataset is memorized and trained very well which leads to increase in test dataset.

Task 9

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



Decision space



It is s not quite possible to draw a straight line to separate the points (0,0) (1,1) from the points (0,1) (1,0) The reason is because the classes in XOR are not linearly separable. XOR Classification problem is not linearly separable. When x1 and x2 is given as input to the perceptron, the output will be sigmoid(w0+w1x1+w2x2) and the problem given must be linearly separable to get the correct output and hence a logistic regression unit cannot solve XOR Problem as the illustration indicates that XOR is not linearly separable.Logistic regression tries to find a linear separator. Try drawing a single line with 1s on oneside and 0s on the other. It can't be done. It is true that Logistic regression (LR) finds a linear decision boundary, and therefore in a straightforward implementation cannot nail XOR. However, logistics regression can get 100% accuracy on the XOR problem by introducing a simple transformation to the feature space. Add the feature $x1 \times x2 x1 \times x2$ to the data and you will get perfect accuracy. Adding this nonlinear feature allows LR to learn a decision boundary that is linear in the features, but not in the original dataspace, and that's all you need for 100% accuracy on XOR.

Task 10

Implement backpropagation's code, by filling the backward_pass() function, found in NeuralNetwork.py. Although XOR has only one output, your implementation should support outputs of any size.

NeuralNetwork.py

```
#       # Step 1. Output deltas are used to update the weights of the output
layer
output_deltas = np.zeros((self.n_out))
outputs = self.y_out.copy()
#print(len(self.y_out))
#print((self.n_out))
#print(len(outputs))
#print((targets))
#print(len(output_deltas))
for i in range(self.n_out):
    ########################################
    # Write your code here
    # compute output_deltas : delta_k = (y_k - t_k) * g'(x_k)
    output_deltas[i] = (outputs[i] - targets) * sigmoid_derivative(outputs[i])
    # output_deltas[i] = ...
    ########################################/
```

Step 1. Output deltas are used to update the weights of the output

```
# 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 i in range(len(hidden_deltas)):
    ########################################
    # Write your code here
    # compute hidden_deltas
    for j in range(len(output_deltas)):
        hidden_deltas[i] = hidden_deltas[i] + output_deltas[j] * self.w_out[i, j]
    hidden_deltas[i] = hidden_deltas[i] * sigmoid_derivative(self.y_hidden[i])
    #...
    #...
    #...
    #hidden_deltas[i] = ...

    ########################################/
```

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

```
# Step 3. update the weights of the output layer
for i in range(len(self.y_hidden)):
    for j in range(len(output_deltas)):
        ########################################
        # Write your code here
        # update the weights of the output layer
        self.w_out[i, j] = self.w_out[i, j] - (n * output_deltas[j] *
self.y_hidden[i])
        # self.w_out[i,j] = ...
        ########################################/

# 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 3. update the weights of the output layer

```
# 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)):
        ########################################
        # Write your code here
        # update the weights of the hidden layer
        self.w_hidden[i, j] = self.w_hidden[i, j] - (n * hidden_deltas[j] *
inputs[i])
        # self.w_hidden[i,j] = ...
        ########################################/
```

 # Step 4. update the weights of the hidden layer

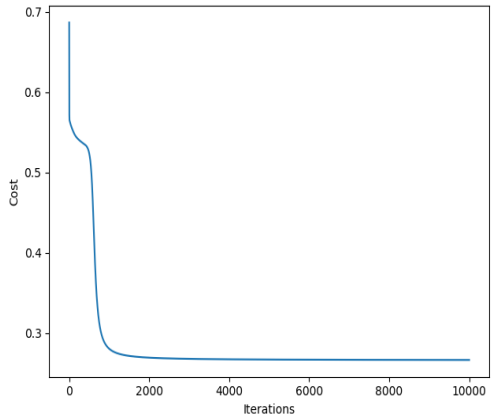
Task 11

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

NOR Gate:

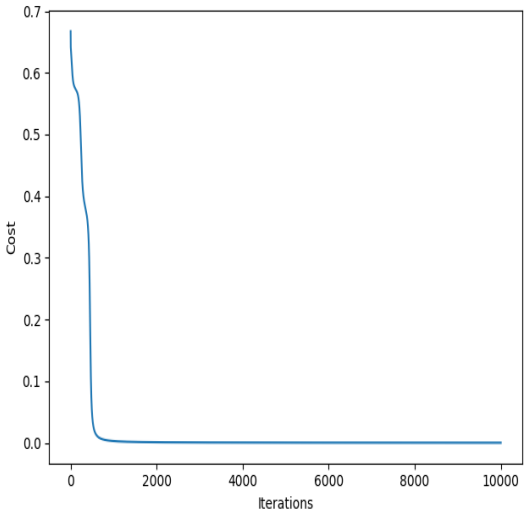
Learning rate =1

Sample #01 | Target value: 0.00 | Predicted value: 0.01066
Sample #02 | Target value: 1.00 | Predicted value: 0.48370
Sample #03 | Target value: 1.00 | Predicted value: 0.98674
Sample #04 | Target value: 0.00 | Predicted value: 0.48420
Minimum cost: 0.26677, on iteration #10000



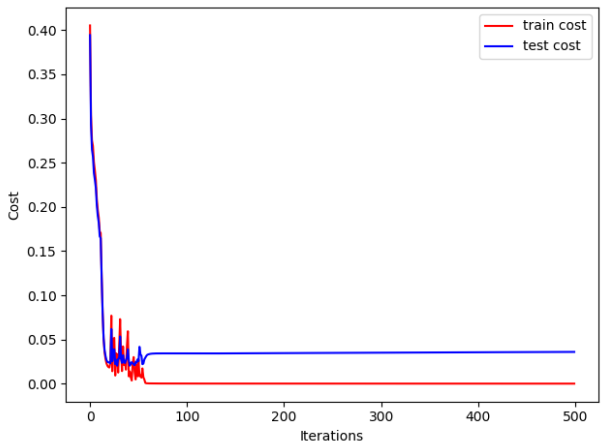
Learning rate = 2

Sample #01 | Target value: 0.00 | Predicted value: 0.00746
Sample #02 | Target value: 1.00 | Predicted value: 0.99226
Sample #03 | Target value: 1.00 | Predicted value: 0.99223
Sample #04 | Target value: 0.00 | Predicted value: 0.00950
Minimum cost: 0.00013, on iteration #10000



Learning rate = 3

Iteration 00470 | Cost = 0.00067
Iteration 00480 | Cost = 0.00065
Iteration 00490 | Cost = 0.00064
Iteration 00500 | Cost = 0.00063
Minimum cost: 0.00063, on iteration #500



Learning rate 2 can be considered because it gives the least error rate.

Task 12

The Iris data set contains three different classes of data that we need to discriminate between. How would you accomplish this if we used a logistic regression unit? How is this scenario different, compared to the scenario of using a neural network?

As per definition and practice, logistics regression would misclassify because of its algorithmic nature which cannot perform linearly separable task.

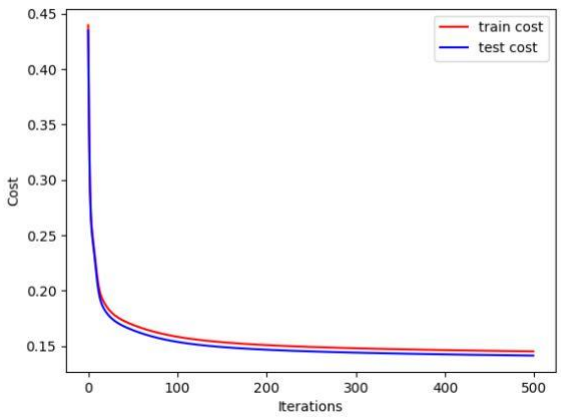
Neural classification is non linear and should perform much better with this data set.

Moreover logistics regression does not consider specific parameters like location .

13)Run irisExample.py using the following number of hidden neurons: 1, 2, 3, 5, 7, 10. The program will plot the costs of the training set (red) and test set (blue) 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?

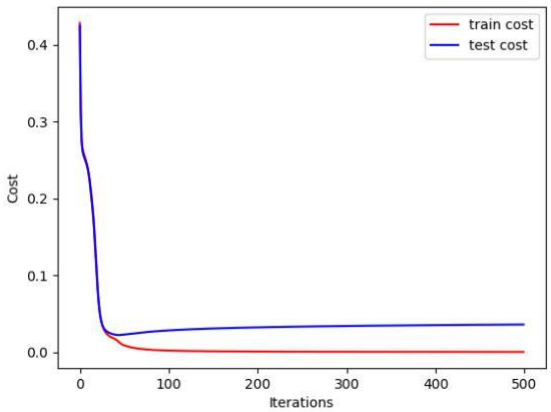
Hidden Neurons = 1

Minimum cost: 2.76482, on iteration #500



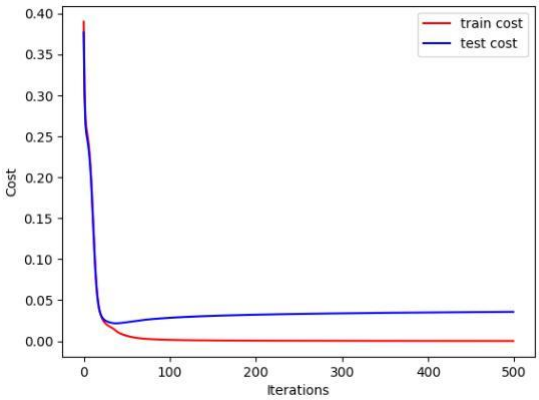
Hiddens Neurons = 2

Minimum cost: 0.00556, on iteration #500



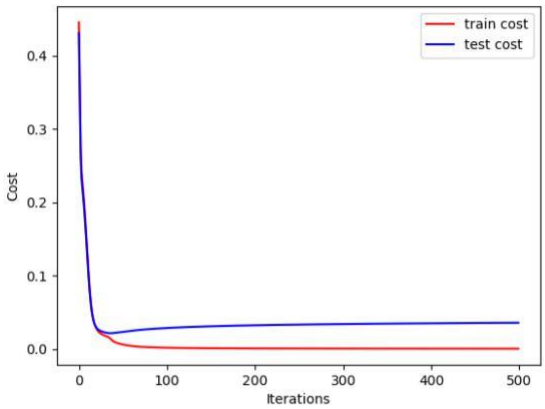
Hidden Neurons = 3

Minimum cost: 0.00460, on iteration #500



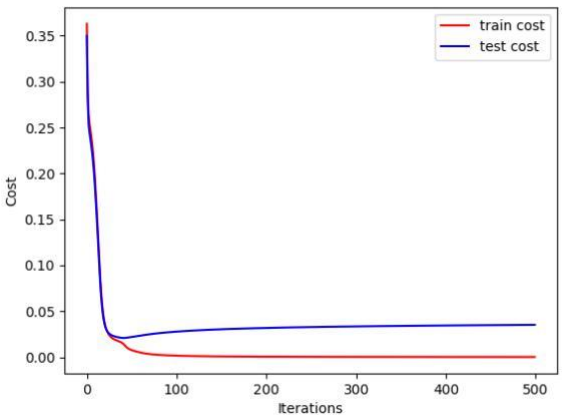
Hidden Neurons = 5

Minimum cost: 0.00432, on iteration #500



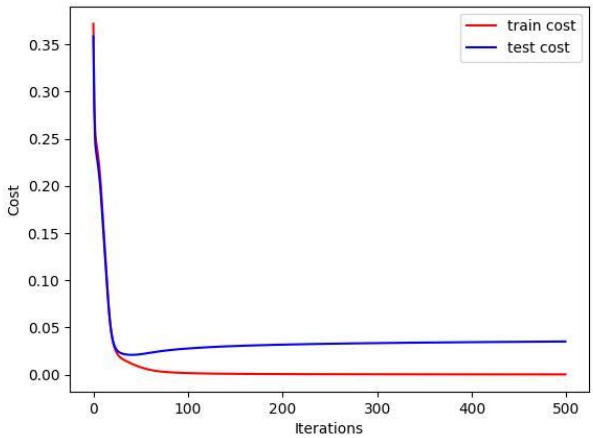
Hidden Neurons = 7

Minimum cost: 0.00426, on iteration #500



Hidden Neurons = 10

Minimum cost: 0.00350, on iteration #500



When number of hidden neurons is 10, the program gives the least test error and hence 10 hidden neurons is considered to be the best.