ECS708 Machine Learning

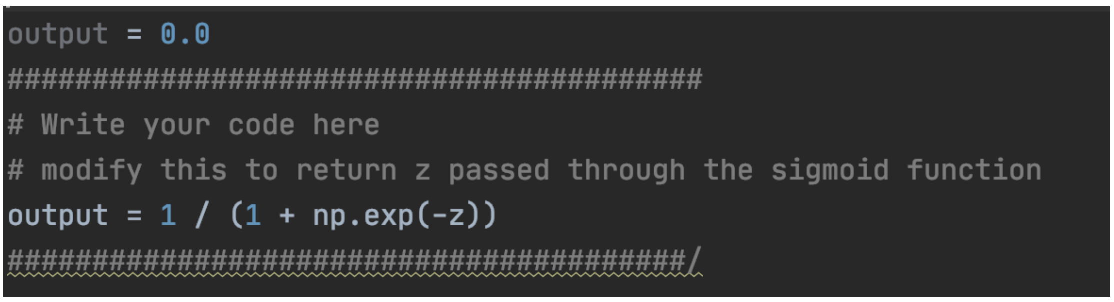
Guanzhen Wu

161189284

Assignment 1 – Part 2: Logistic Regression and Neural Networks

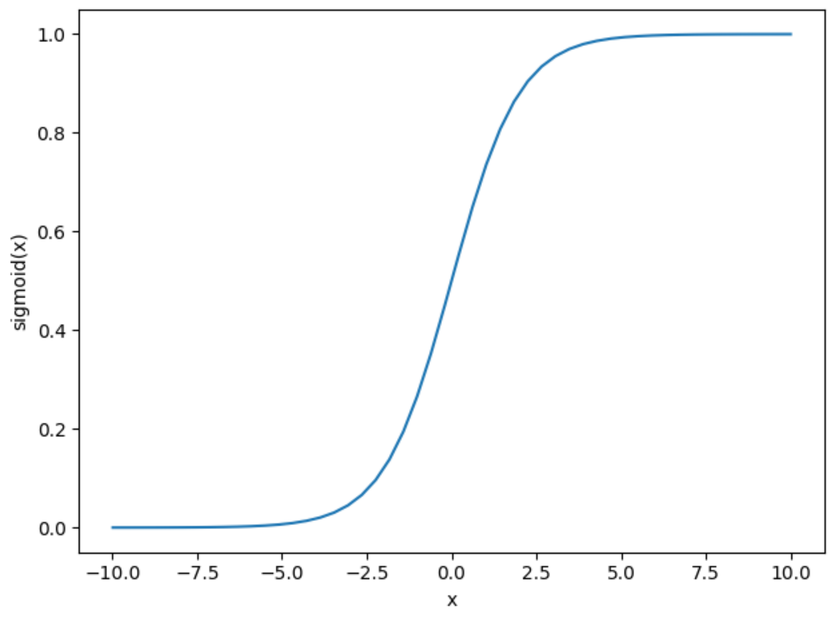
1. Logistic Regression

Task 1: Include in your report the relevant lines of code and the result of the using plot_sigmoid.py. [3 points]

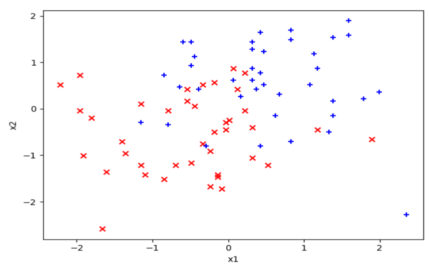


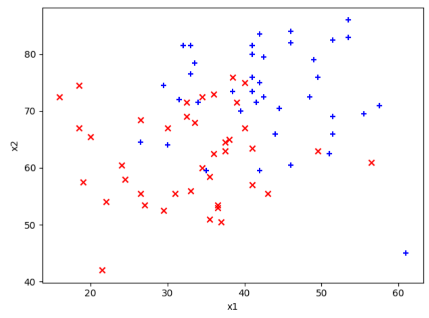
```
output = 0.0
##########################################
# Write your code here
# modify this to return z passed through the sigmoid function
output = 1 / (1 + np.exp(-z))
##########################################/
```

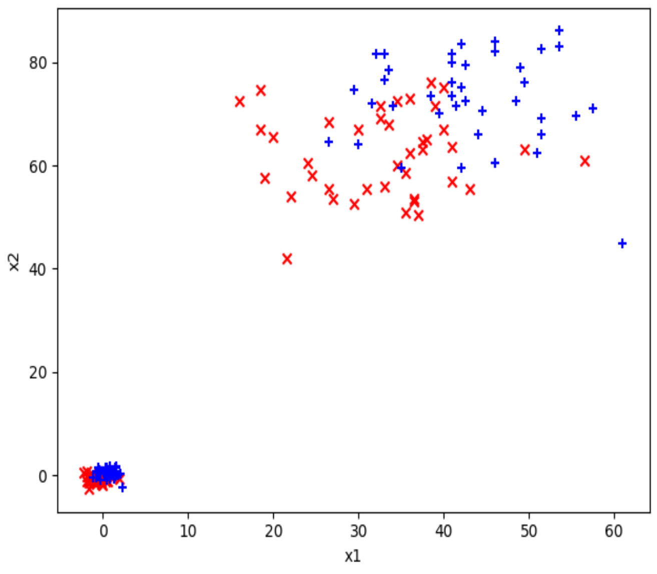
The output of the *plot_sigmoid.py* is show as below:



Task2: Plot the normalized data to see what it looks like. Plot also the data, without normalization. Enclose the plots in your report. [2 points]





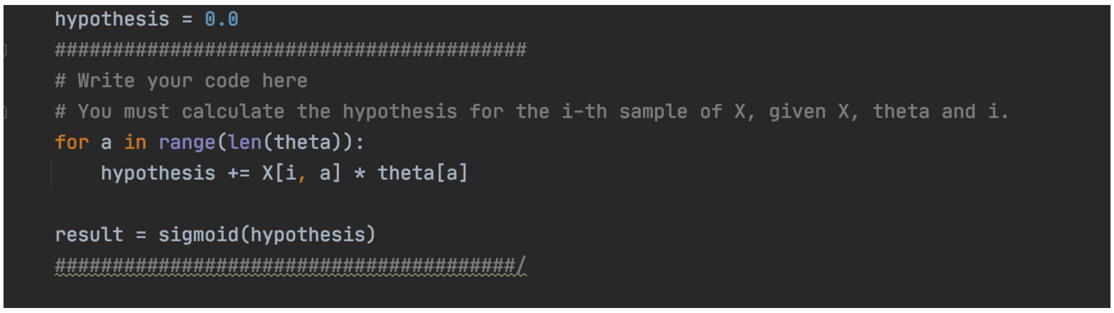


As the figure above shows that after normalization, the data separate in a narrow range which nearly between [-3,3]. The original data was separate in the range of [0, 90]. The figure on the right shows the both normalized data and original data in the same graph.

Task3. 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. [5 points]

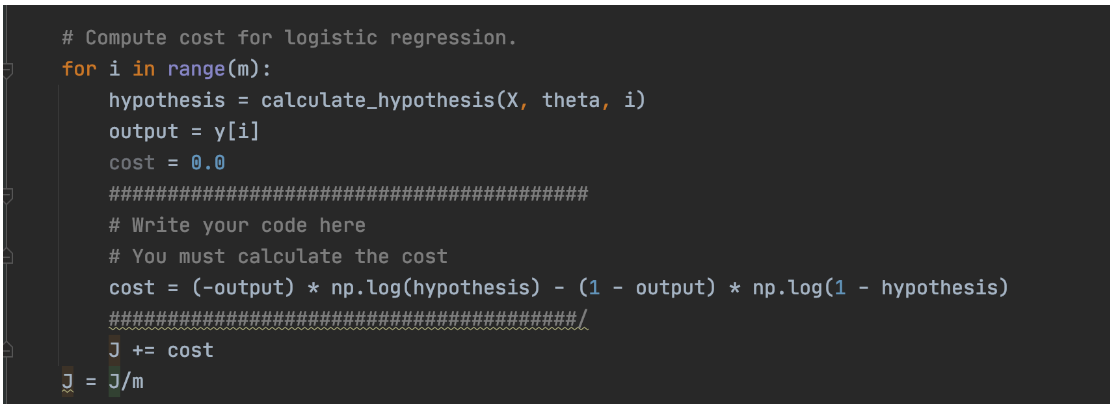
$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$



```
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 a in range(len(theta)):
    hypothesis += X[i, a] * theta[a]

result = sigmoid(hypothesis)
###########################################/
```

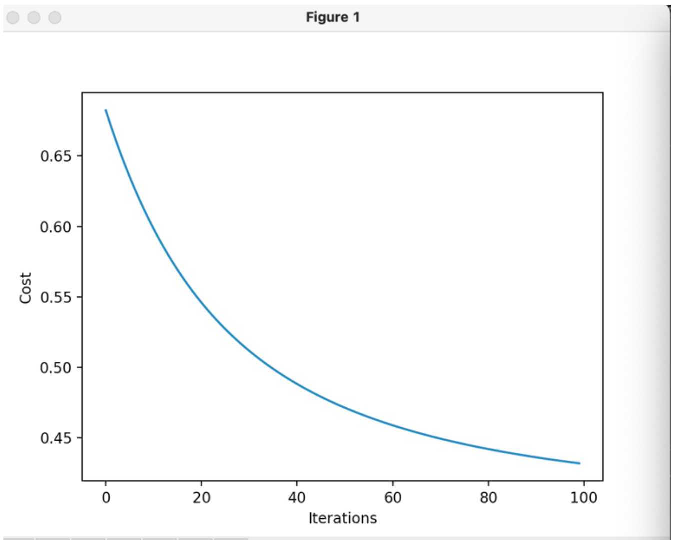
Task 4. Modify the line “cost = 0.0” in compute_cost.py so that we use our cost function:

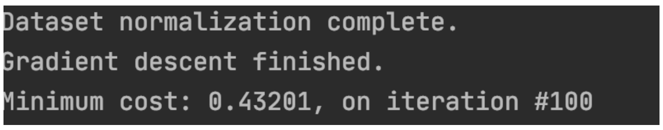
$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(-y^{(i)}log(h_\theta(x^{(i)})) - (1 - y^{(i)})log(1 - h_\theta(x^{(i)}))\right)$$



```
# 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 = (-output) * np.log(hypothesis) - (1 - output) * np.log(1 - hypothesis)
    ########################################/
    J += cost
J = J/m
```

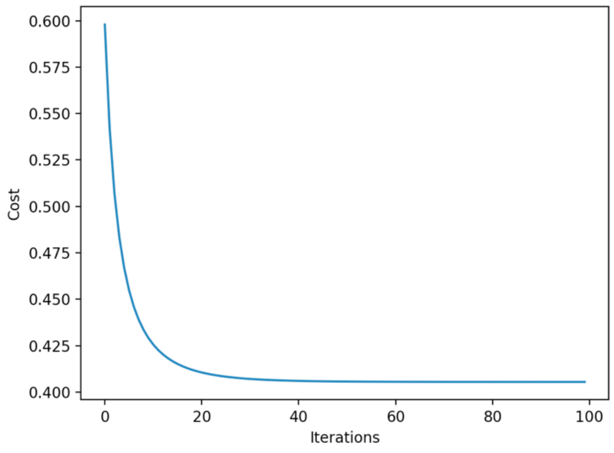
For alpha equals to 0.001, the cost figure shows as below, the final cost is 0.43201.
It can be clearly seen that the cost function has not completely converged, the learning rate is too small.

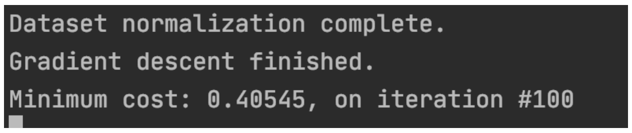




```
Dataset normalization complete.
Gradient descent finished.
Minimum cost: 0.43201, on iteration #100
```

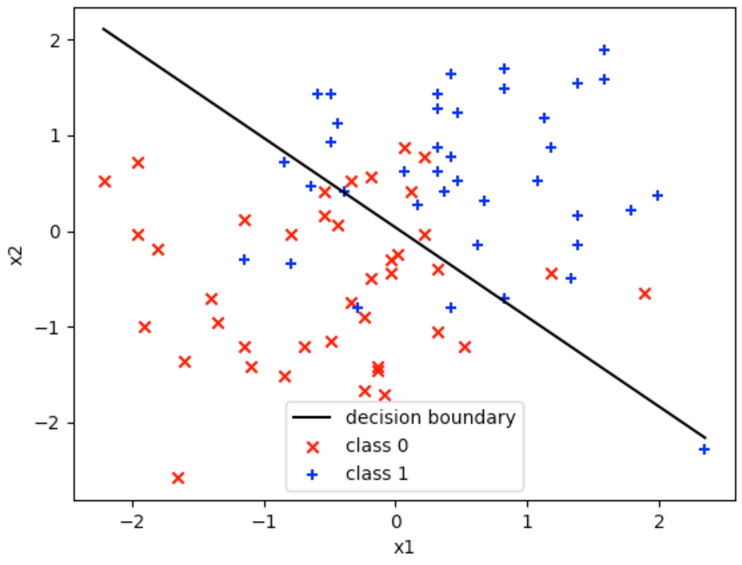
So I increase the alpha and after many times adjustment, the final alpha set to 0.01, and the cost converge to 0.40545:

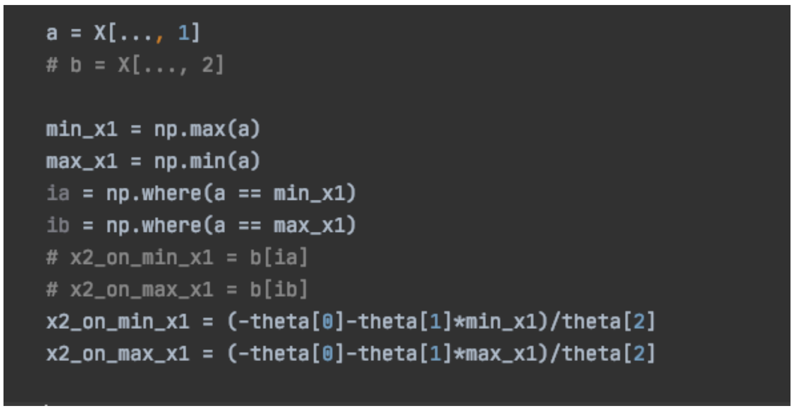




```
Dataset normalization complete.
Gradient descent finished.
Minimum cost: 0.40545, on iteration #100
```

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. [5 points]



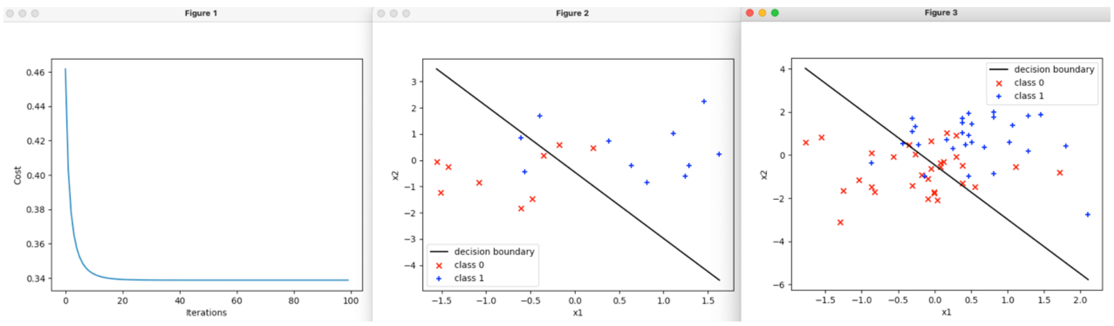


```
a = X[..., 1]
# b = X[..., 2]

min_x1 = np.max(a)
max_x1 = np.min(a)
ia = np.where(a == min_x1)
ib = np.where(a == max_x1)
# x2_on_min_x1 = b[ia]
# x2_on_max_x1 = b[ib]
x2_on_min_x1 = (-theta[0]-theta[1]*min_x1)/theta[2]
x2_on_max_x1 = (-theta[0]-theta[1]*max_x1)/theta[2]
```

With the implement code, a refers to X1 in array. Find out the max and min x1 in X1 and then use min_x1, max_x1 to compute x2.

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.

Good generalization:

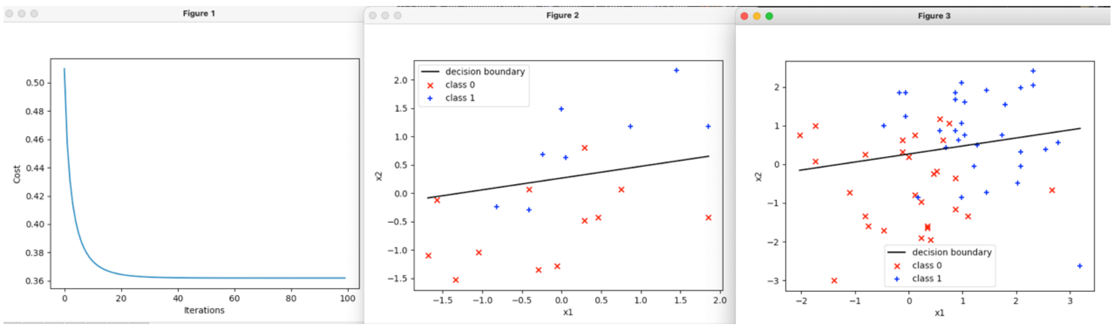


Final training cost: 0.33878

Minimum training cost: 0.33878, on iteration #100

Final test cost: 0.50238

Bad generalization:

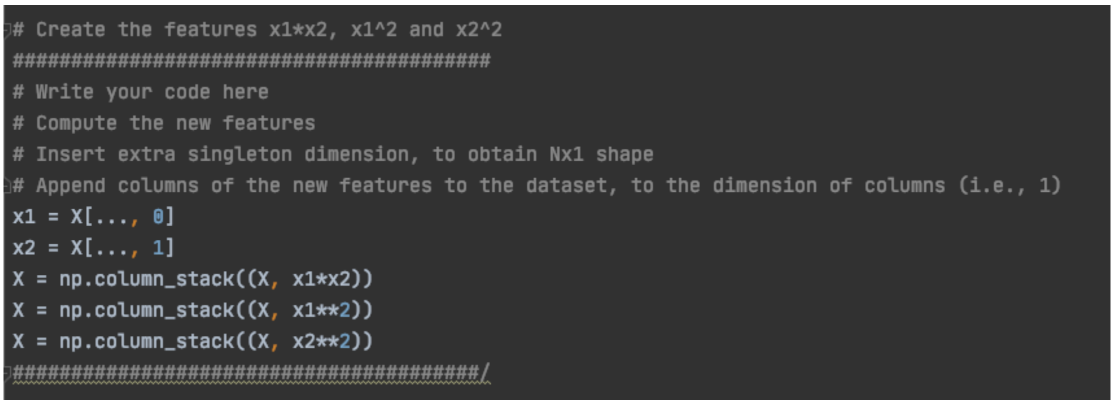


Final training cost: 0.36193

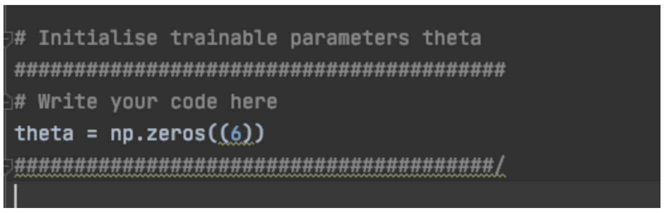
Minimum training cost: 0.36193, on iteration #100
Final test cost: 0.94596

The generalization ability of the model is good when the center points of this class in the selected training samples are close and the distribution is uniform. If too few training samples are selected, it will affect the accuracy of the classifier.

In assgn1_ex3.py, instead of using just the 2D feature vector, incorporate the following non-linear features: This results in a 5D input vector per data point, and so you must use 6 parameters θ. [2 points]

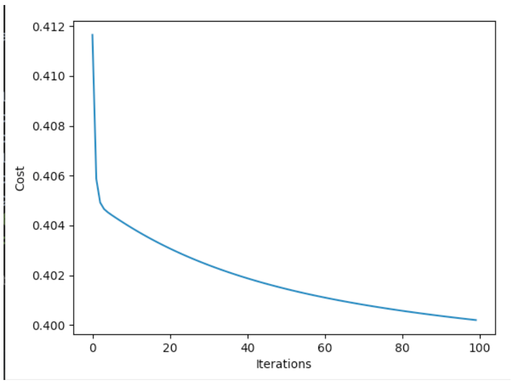


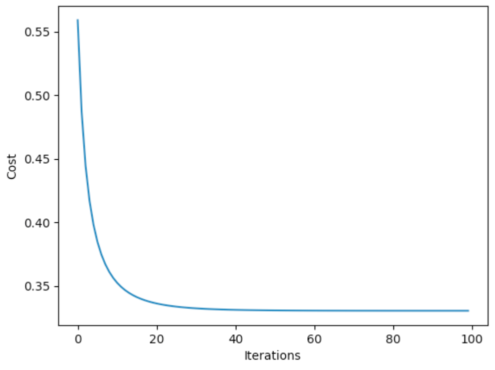
```
# Create the features x1*x2, x1^2 and x2^2
##########################################
# Write your code here
# Compute the new features
# Insert extra singleton dimension, to obtain Nx1 shape
# Append columns of the new features to the dataset, to the dimension of columns (i.e., 1)
x1 = X[..., 0]
x2 = X[..., 1]
X = np.column_stack((X, x1*x2))
X = np.column_stack((X, x1**2))
X = np.column_stack((X, x2**2))
##########################################/
```



```
# Initialise trainable parameters theta
##########################################
# Write your code here
theta = np.zeros((6))
##########################################/
```

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. [5 points]

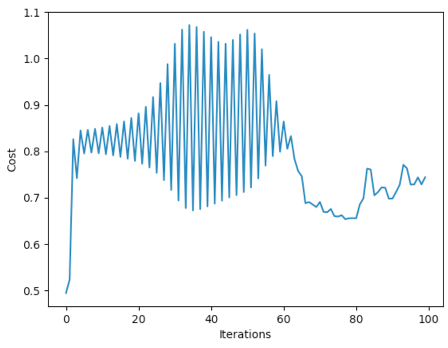




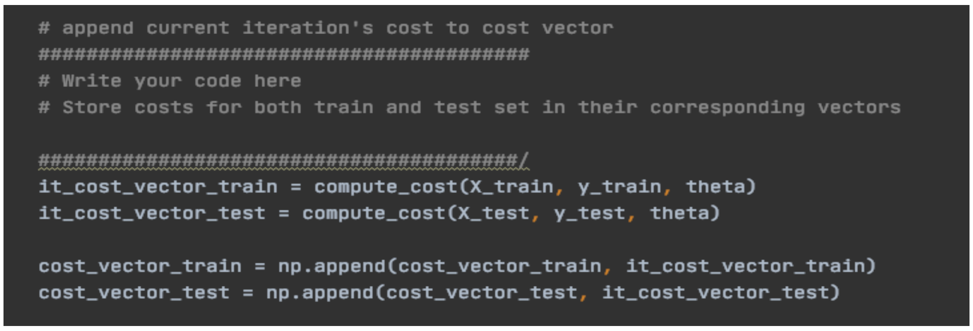
Alpha = 0.05 in task7

Alpha = 0.05 in task4

The addition of high-order terms and interaction terms will disturb the accuracy of the model. At the same time, because the value of the higher-order term is large, the cost function will begin to oscillate, as shown in the figure below when alpha = 0.4

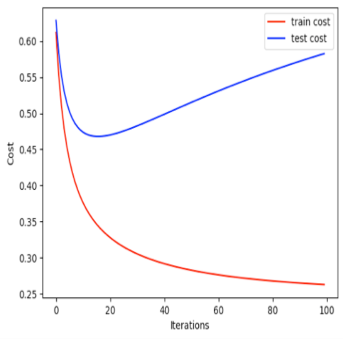


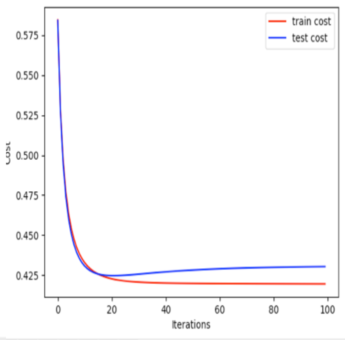
Task 8. In assgn1_ex4.py the data are split into a test set and a training set. Add your new features from the question above (assgn1_ex3.py). Modify the function gradient_descent_training.py to store the current cost for the training set and testing set. Store the cost of the training set to cost_vector_train and for the test set to cost_vector_test.

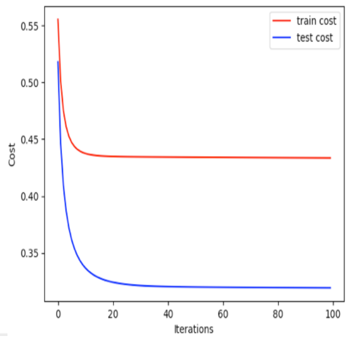


```
# append current iteration's cost to cost vector
##########################################
# Write your code here
# Store costs for both train and test set in their corresponding vectors

##########################################/
it_cost_vector_train = compute_cost(X_train, y_train, theta)
it_cost_vector_test = compute_cost(X_test, y_test, theta)

cost_vector_train = np.append(cost_vector_train, it_cost_vector_train)
cost_vector_test = np.append(cost_vector_test, it_cost_vector_test)
```

These arrays are passed to plot_cost_train_test.py, which will show the cost function of the training (in red) and test set (in blue). 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. 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? [5 points]



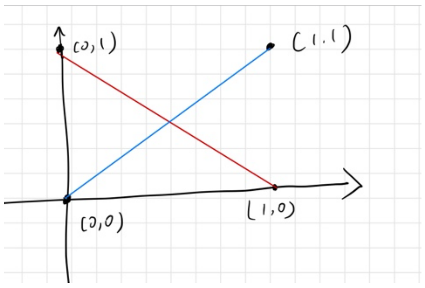




Training_sample =20 40 60

It can be seen from the above figure that when the training set becomes smaller and the test increases, the cost function of the test set will first drop and then rise, and it has not yet converged when the iteration cuts off. When train samples=60, both cost functions tend to converge and reach the optimal value.

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



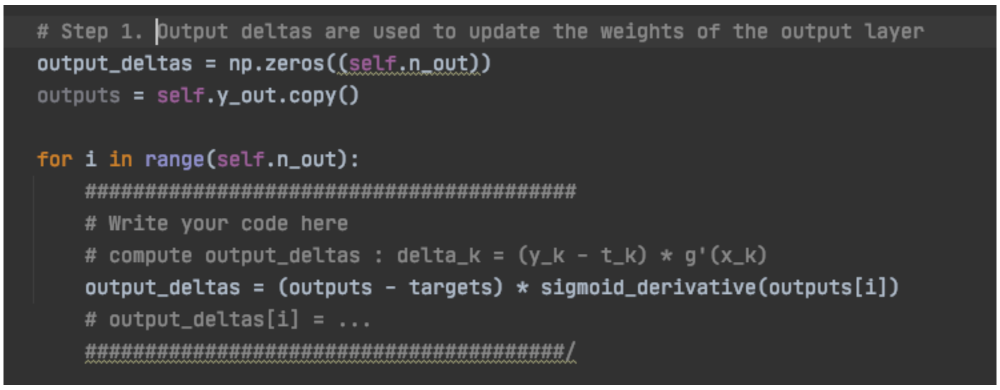
For the logistic regression unit, it cannot handle discrete and abrupt data sets to classify the two points on the diagonal into the same category.

2.Neural Network

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. Do this following the steps below: [5 points]

Step1:

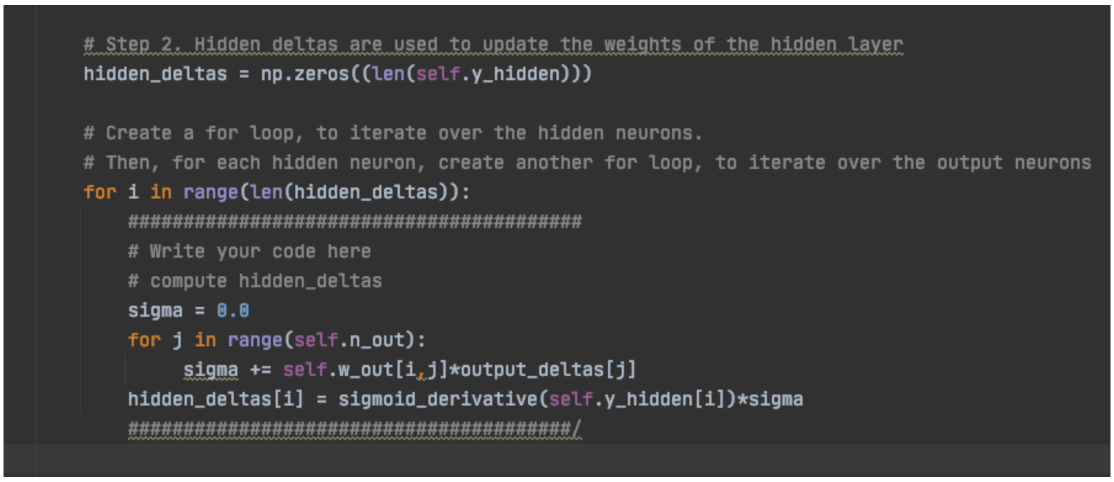
$$\delta_k = (y_k - t_k)g'(x_k)$$



```
# 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()

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 = (outputs - targets) * sigmoid_derivative(outputs[i])
    # output_deltas[i] = ...
    ##########################################/
```

Step2:

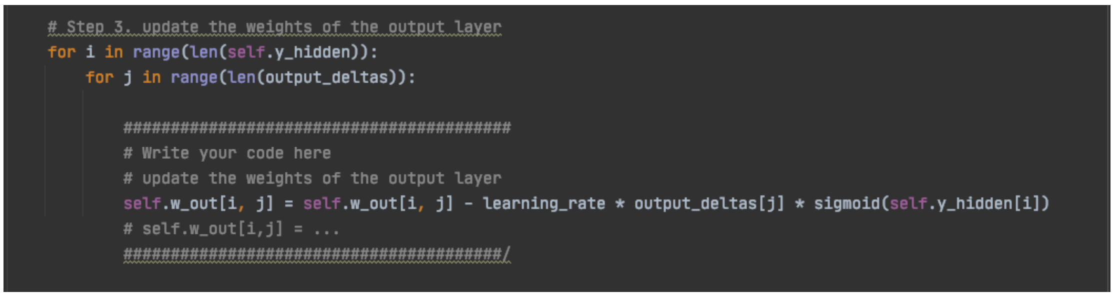
$$\delta_j = g'(x_j)\sum_k (w_{jk}\delta_j)$$



```
# 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
    sigma = 0.0
    for j in range(self.n_out):
        sigma += self.w_out[i,j]*output_deltas[j]
    hidden_deltas[i] = sigmoid_derivative(self.y_hidden[i])*sigma
    ########################################/
```

Step3:

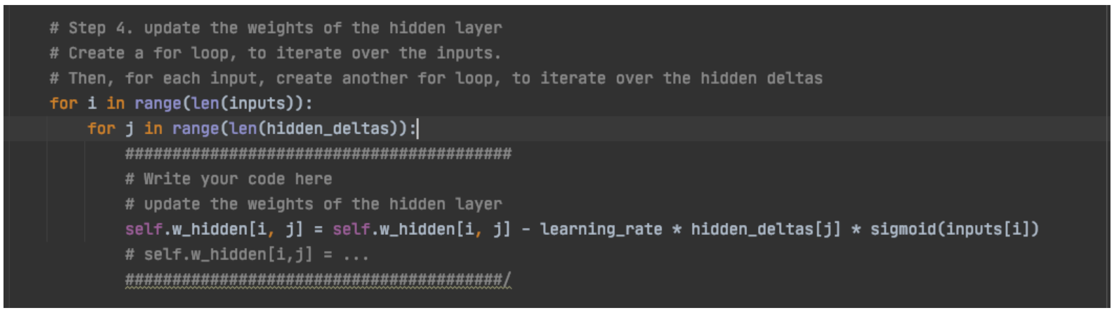
$$w_{jk} = w_{jk} - \eta\delta_k\alpha_j$$



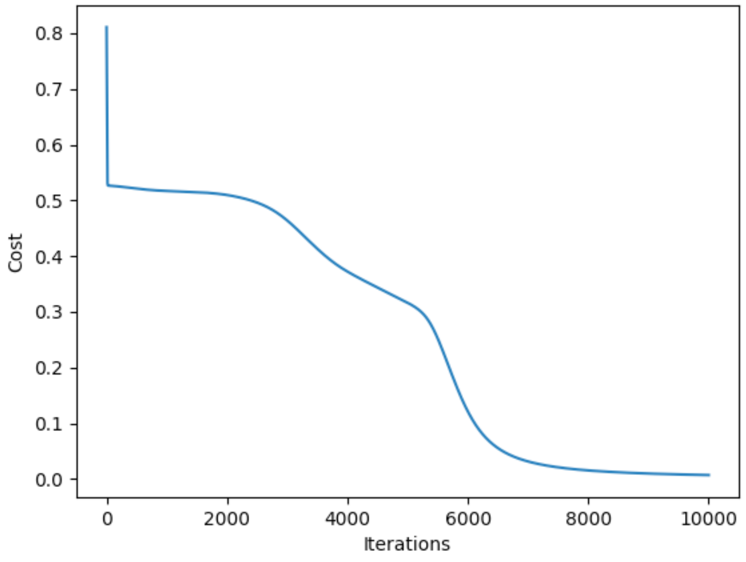
```
# Step 3. update the weights of the output layer
for i in range(len(self.y_hidden)):
    for j in range(len(output_deltas)):

        #########################################
        # Write your code here
        # update the weights of the output layer
        self.w_out[i, j] = self.w_out[i, j] - learning_rate * output_deltas[j] * sigmoid(self.y_hidden[i])
        # self.w_out[i,j] = ...
        ########################################/
```

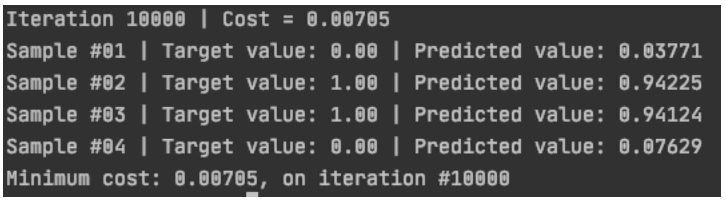
Step4:

$$w_{jk} = w_{jk} - \eta\delta_k\alpha_j$$



```
    # 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] - learning_rate * hidden_deltas[j] * sigmoid(inputs[i])
            # self.w_hidden[i,j] = ...
            #########################################/
```

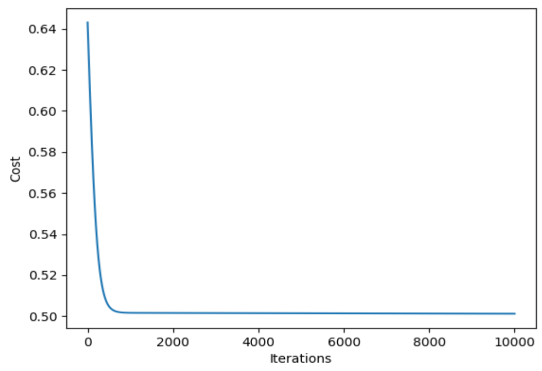


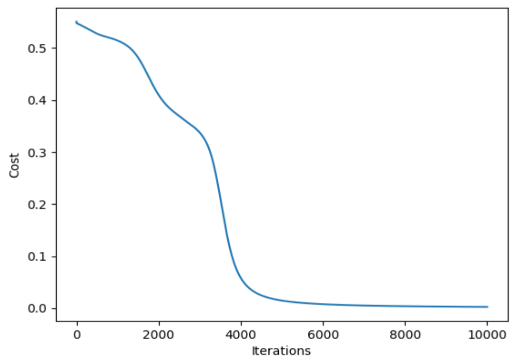


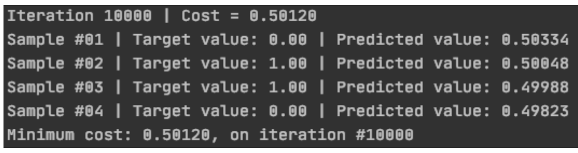
```
Iteration 10000 | Cost = 0.00705
Sample #01 | Target value: 0.00 | Predicted value: 0.03771
Sample #02 | Target value: 1.00 | Predicted value: 0.94225
Sample #03 | Target value: 1.00 | Predicted value: 0.94124
Sample #04 | Target value: 0.00 | Predicted value: 0.07629
Minimum cost: 0.00705, on iteration #10000
```

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

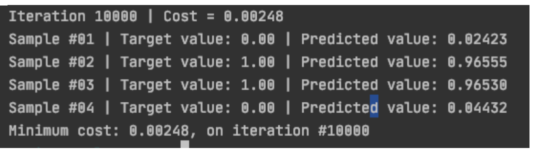
What learning rate do you find best? Include a graph of the error function in your report. Note that the backpropagation can get stuck in local optima. What are the outputs and error when it gets stuck?







```
Iteration 10000 | Cost = 0.50120
Sample #01 | Target value: 0.00 | Predicted value: 0.50334
Sample #02 | Target value: 1.00 | Predicted value: 0.50048
Sample #03 | Target value: 1.00 | Predicted value: 0.49988
Sample #04 | Target value: 0.00 | Predicted value: 0.49823
Minimum cost: 0.50120, on iteration #10000
```



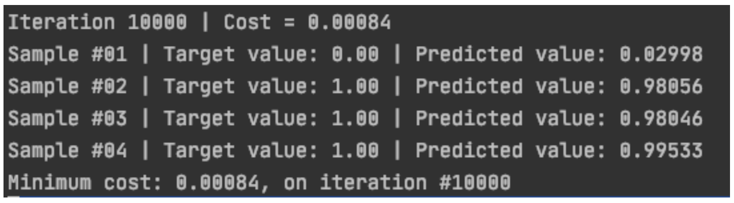
```
Iteration 10000 | Cost = 0.00248
Sample #01 | Target value: 0.00 | Predicted value: 0.02423
Sample #02 | Target value: 1.00 | Predicted value: 0.96555
Sample #03 | Target value: 1.00 | Predicted value: 0.96530
Sample #04 | Target value: 0.00 | Predicted value: 0.04432
Minimum cost: 0.00248, on iteration #10000
```

Alpha = 0.01 Alpha = 0.8

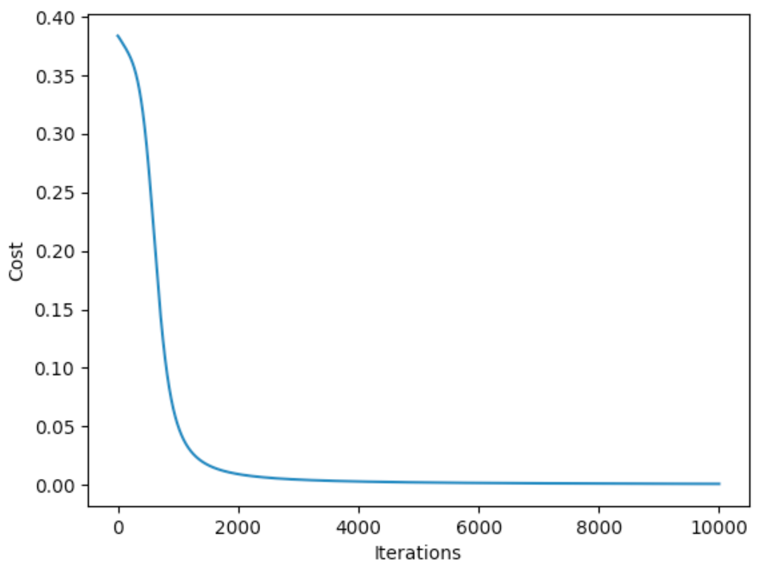
As shown above when the learning rate equal to 0.01 which is too small, the gradient descent will fall into the local optima and cannot get out. The best learning rate is 0.8

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. [5 points]

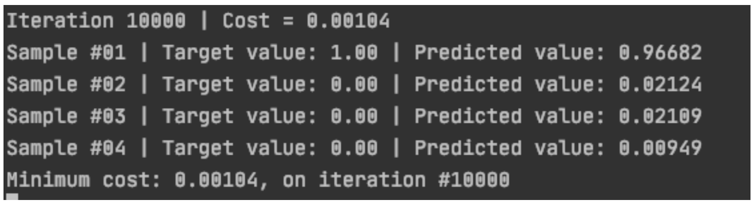
OR:



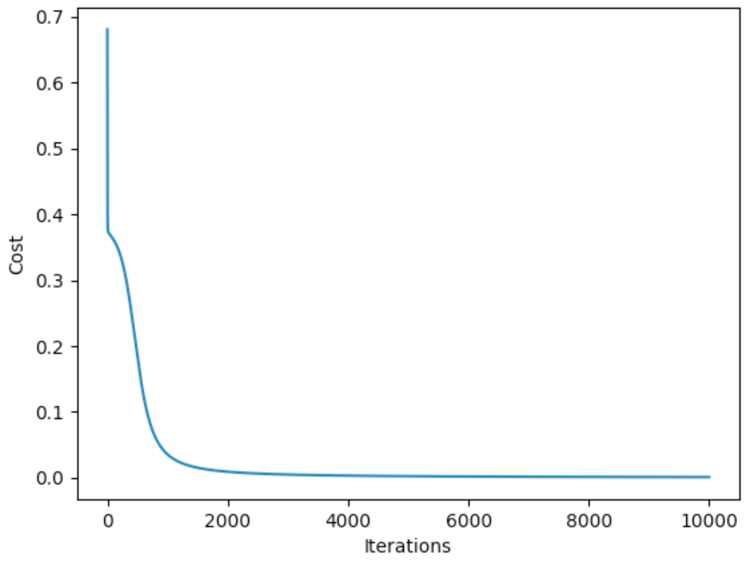
```
Iteration 10000 | Cost = 0.00084
Sample #01 | Target value: 0.00 | Predicted value: 0.02998
Sample #02 | Target value: 1.00 | Predicted value: 0.98056
Sample #03 | Target value: 1.00 | Predicted value: 0.98046
Sample #04 | Target value: 1.00 | Predicted value: 0.99533
Minimum cost: 0.00084, on iteration #10000
```



NOR:



```
Iteration 10000 | Cost = 0.00104
Sample #01 | Target value: 1.00 | Predicted value: 0.96682
Sample #02 | Target value: 0.00 | Predicted value: 0.02124
Sample #03 | Target value: 0.00 | Predicted value: 0.02109
Sample #04 | Target value: 0.00 | Predicted value: 0.00949
Minimum cost: 0.00104, on iteration #10000
```

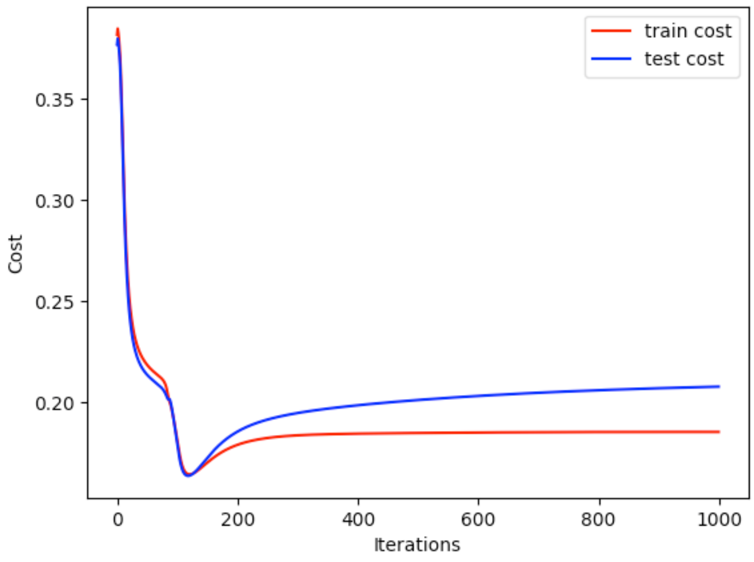


Task 12. The Iris data set contains three different classes of data that we need to discriminate between. How would you accomplish this if we used a logistic regression unit? How is this scenario different, compared to the scenario of using a neural network? [5 points]

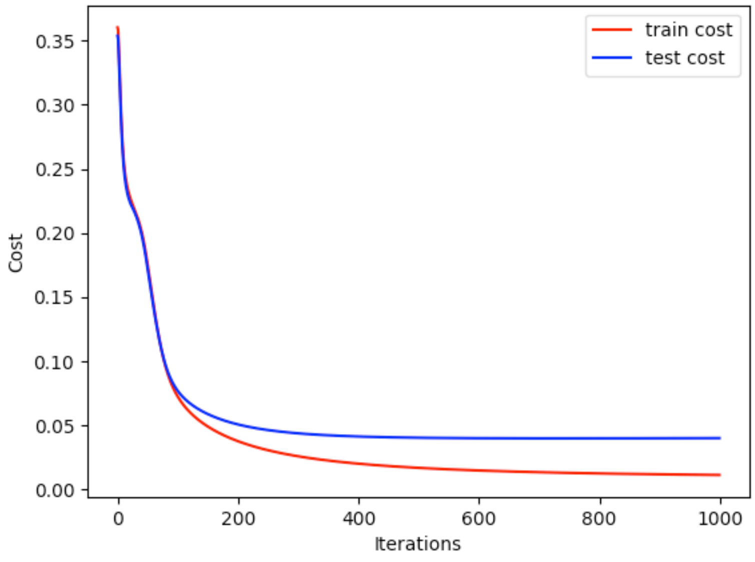
The logistic regression unit is a binary classifier, and the sigmoid function only approaches 0 or 1, so it cannot directly distinguish the three categories of data. But we can change the logistic regression to multiple logistic regression, and for each application of a logistic regression unit, the classification can be doubled. This is very similar to a neural network. For each logistic regression unit, it is equivalent to a node. The output of each node becomes tighter 0 or 1, which can add weight to one of the dimensions of the classification result.

Task 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? [5 points]

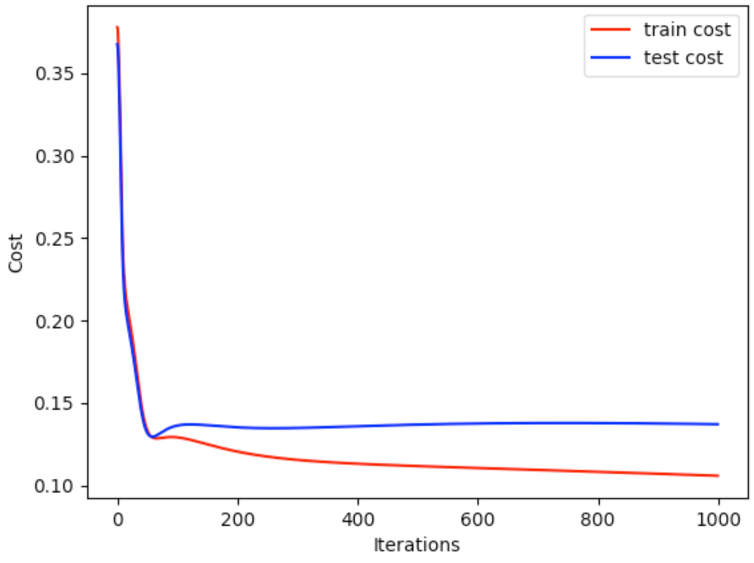
When there are too many nodes, the cost function will show an upward trend, and the prediction results will become inaccurate. After many experiments, it is found that when the hidden node is equal to 2, the effect is the best, the cost function converges and the value is the lowest. According to the result of node = 2, the generalization effect is the best, in which test result is close to the training result.


train cost
test cost
Cost
0.35
0.30
0.25
0.20
0
200
400
600
800
1000
Iterations

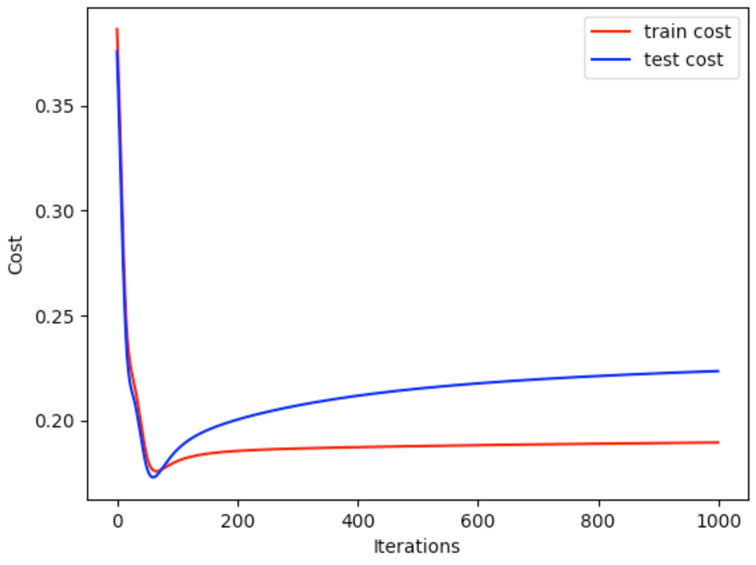
1


train cost
test cost
Cost
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
200
400
600
800
1000
Iterations

2

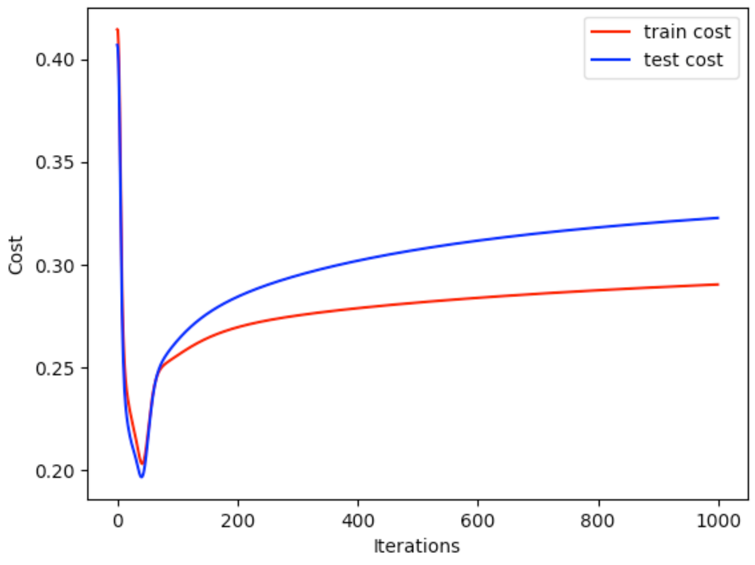

train cost
test cost
Cost
0.35
0.30
0.25
0.20
0.15
0.10
0
200
400
600
800
1000
Iterations

3



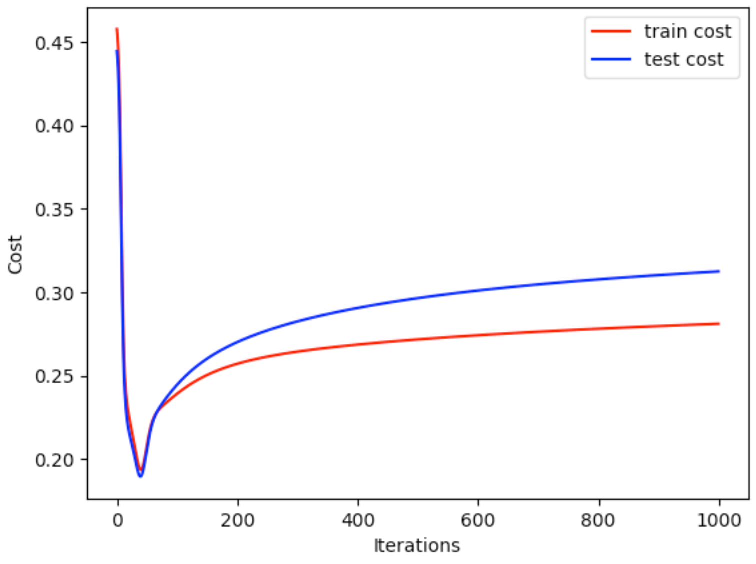
train cost
test cost
Cost
Iterations
0.35
0.30
0.25
0.20
0
200
400
600
800
1000

5



train cost
test cost
Cost
Iterations
0.40
0.35
0.30
0.25
0.20
0
200
400
600
800
1000

7



train cost
test cost
Cost
Iterations
0.45
0.40
0.35
0.30
0.25
0.20
0
200
400
600
800
1000

10