Assignment_1_P2

Yang Tian 200054403 tian.yang@hss19.qmul.ac.uk

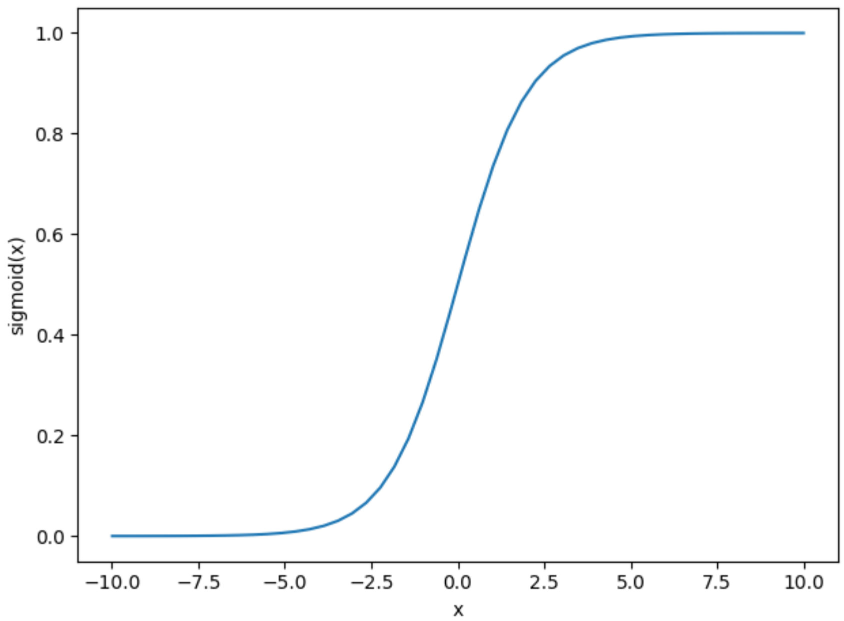
1. Logistic Regression

Task 1.

Fill out *sigmod.py*

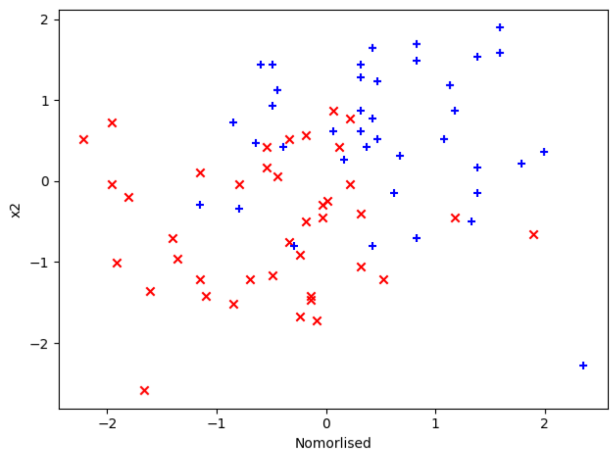
```
def sigmoid(z):

    output = 0.0
    ##########################################
    # Write your code here
    # modify this to return z passed through the sigmoid function

    ##########################################/

    output = 1.0 / (1.0 + np.exp(-z))

    return output
```

use the plot_sigmoid.py function to plot the sigmoid function

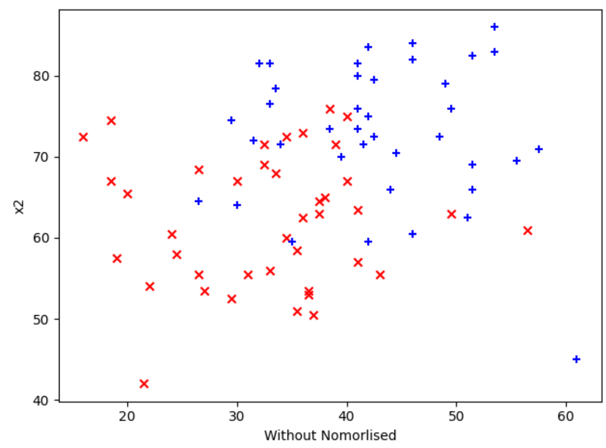


Task 2.
Plot the normalized data to see what it looks like. Plot also the data, without normalization.

Normalised



Without Normalised



1.1. Cost function and gradient for logistic regression

Task 3. Modify the calculate_hypothesis.py

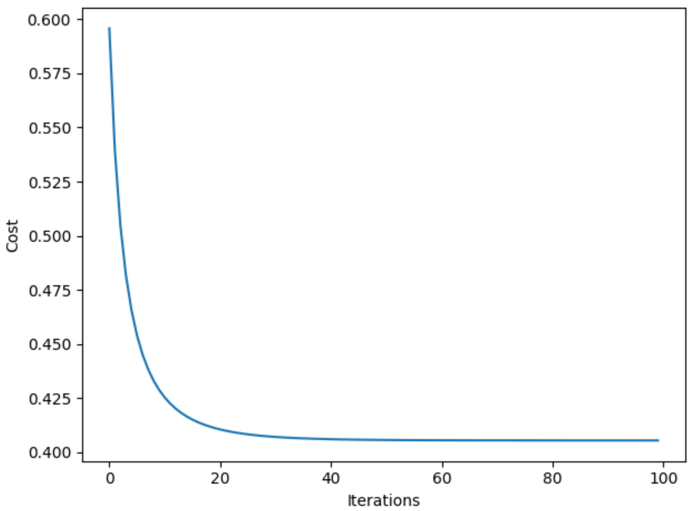
```
def calculate_hypothesis(X, theta, i):
    """
        :param X            : 2D array of our dataset
        :param theta        : 1D array of the trainable parameters
        :param i            : scalar, index of current training sample's row
    """
    hypothesis = 0.0
    #########################################
    # 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 + X[i,j] * theta[j]

    result = sigmoid(hypothesis)

    return result
```

Task 4. Modify the line “cost = 0.0” in compute_cost.py

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

    return J
```

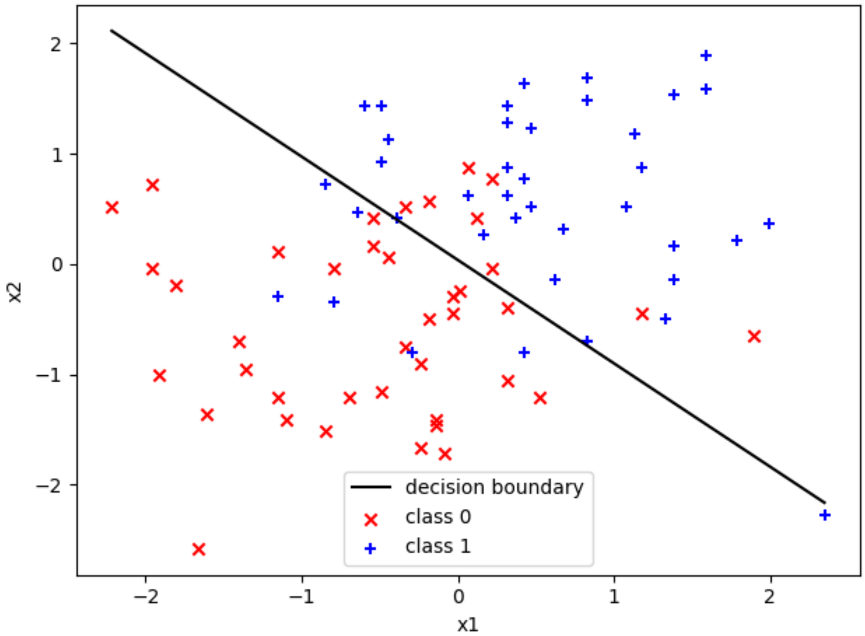
Alpha = 0.01

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



1.2. Draw the decision boundary

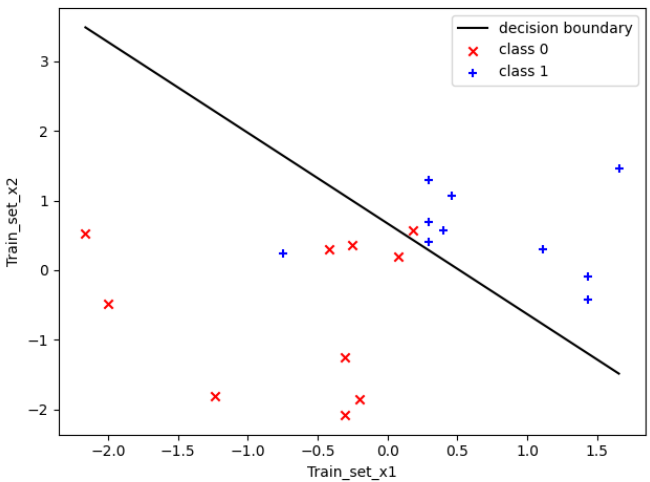
Task5. Plot the decision boundary.

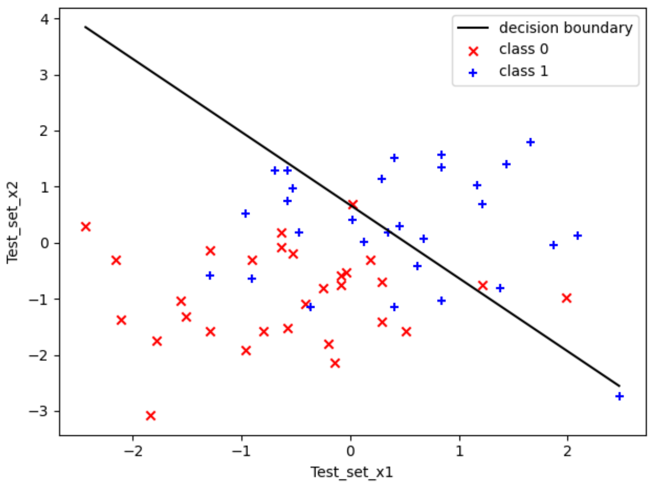


1.3. Non-linear features and overfitting

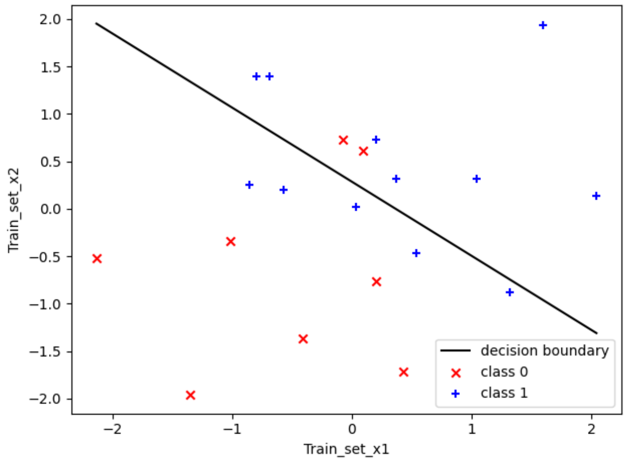
Task 6. Run the code of assgn1_ex2.py several times.

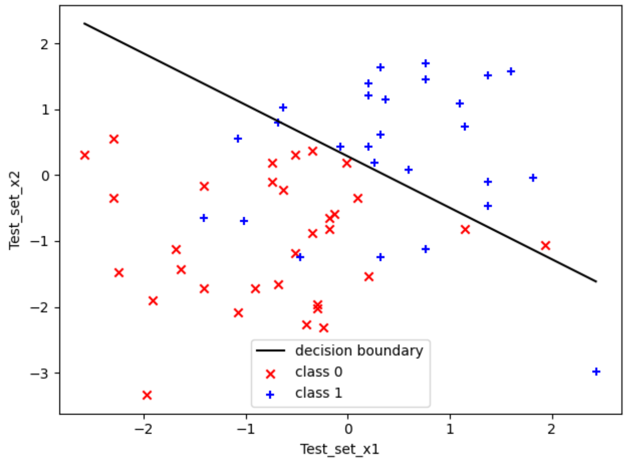
1.






```
Final training cost: 0.32273
Minimum training cost: 0.29915, on iteration #29
Final test cost: 0.69081
```

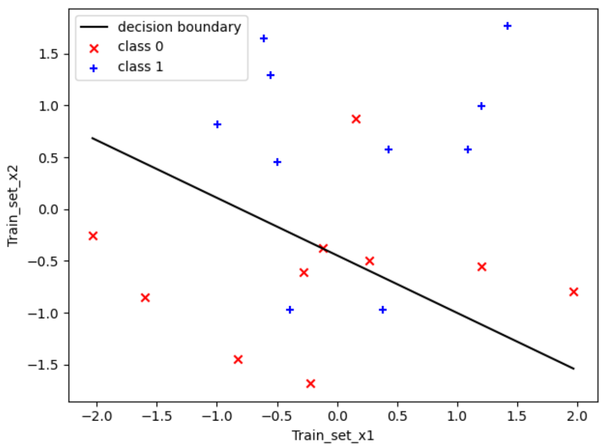


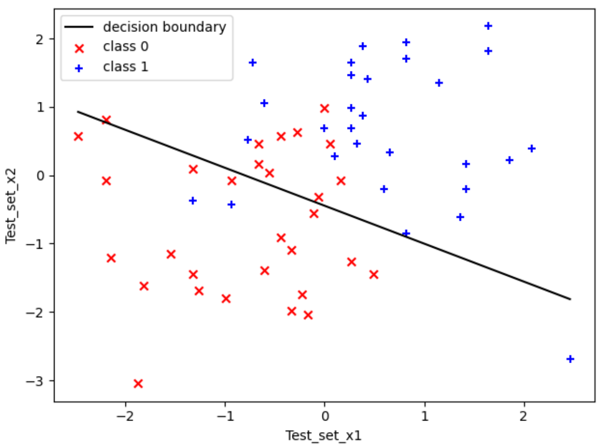



```
Final training cost: 0.53169
Minimum training cost: 0.40627, on iteration #7
Final test cost: 0.58799
```

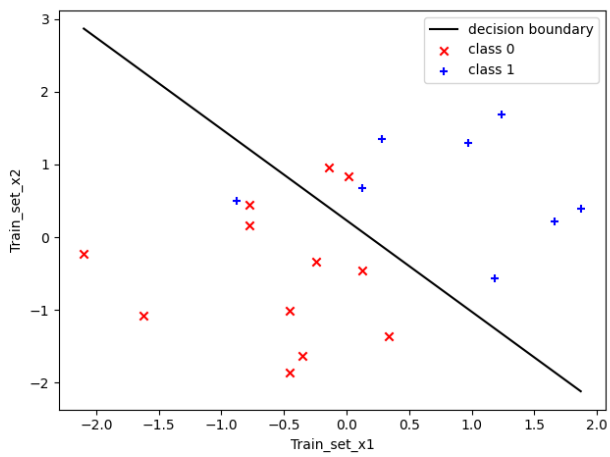
2.

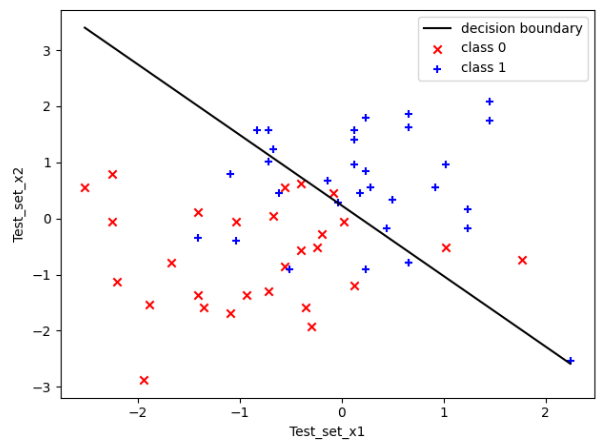
3.






```
Final training cost: 0.59119
Minimum training cost: 0.49428, on iteration #3
Final test cost: 0.43780
```

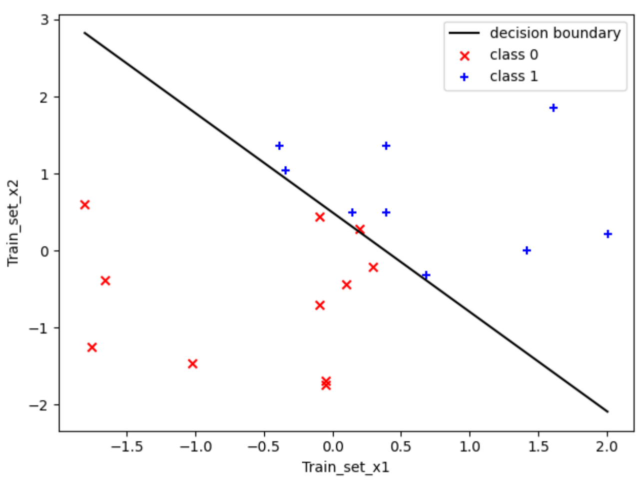


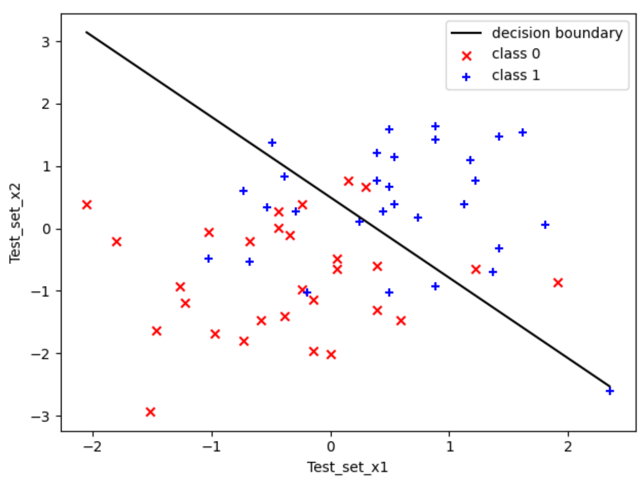


4.

```
Final training cost: 0.31082
Minimum training cost: 0.29198, on iteration #17
Final test cost: 0.53003
```

5.






```
Final training cost: 0.09693
Minimum training cost: 0.09693, on iteration #100
Final test cost: 2.16859
```

The pictures above show that there are big differences between train cost and test cost. Only in the 2nd test, the costs are similar.

use 6 parameters θ

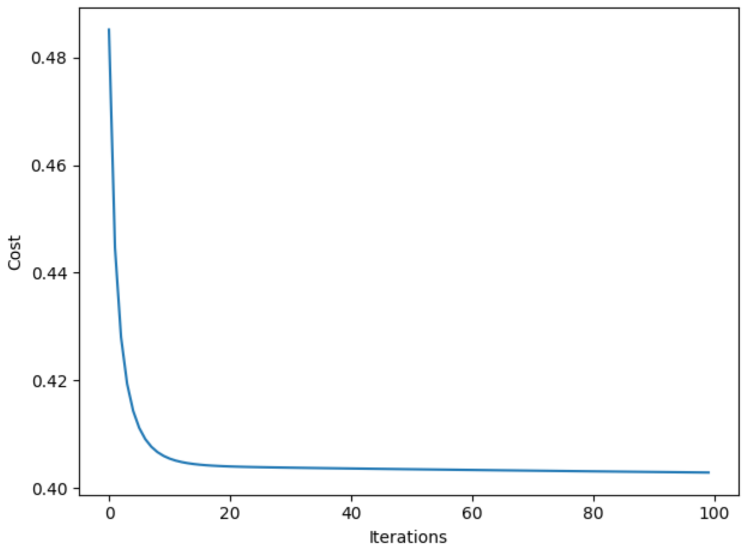
```
# Initialise trainable parameters theta
############################################
# Write your code here
############################################
theta = np.zeros((6))
```

```
rows = X.shape[0]
feature1_array = np.array([])
feature2_array = np.array([])
feature3_array = np.array([])
for i in range(rows):
    feature1_array = np.append(feature1_array,X[i,0]*X[i,1])
    feature2_array = np.append(feature2_array,X[i,0]**2)
    feature3_array = np.append(feature3_array,X[i,1]**2)
X = np.insert(X,2,feature1_array,axis=1)
X = np.insert(X,3,feature2_array,axis=1)
X = np.insert(X,4,feature3_array,axis=1)
```

Task 7.

Alpha = 0.01

```
Gradient descent finished.
Final cost: 0.40288
Minimum cost: 0.40288, on iteration #100
```



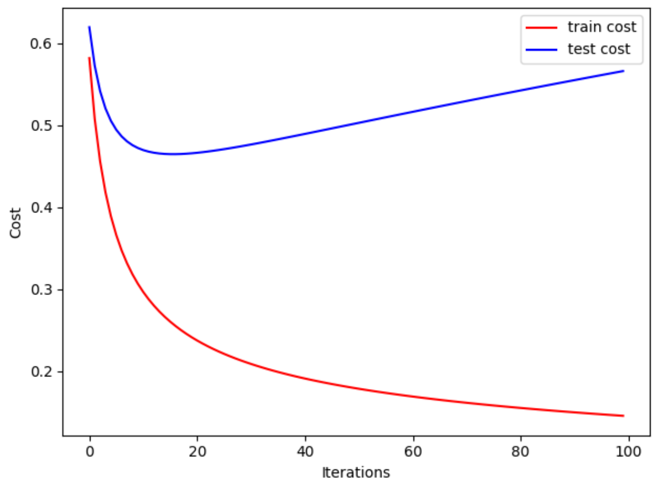
Compared to Task 4, the cost decreased with the increase of the number of features.

Task 8.

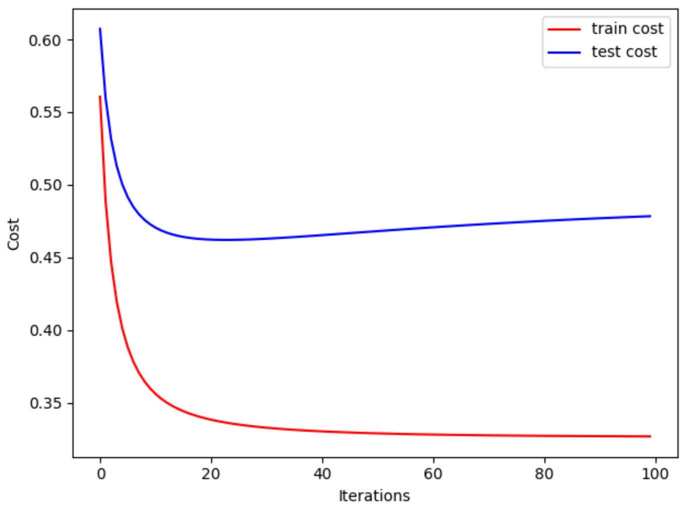
Modify the function *gradient_descent_training.py* to store the current cost for the training set and testing set

```
# append current iteration's cost to cost vector
##########################################
# Write your code here
# Store costs for both train and test set in their corresponding vectors
##########################################/
cost_vector_train = np.append(cost_vector_train, compute_cost(X_train, y_train, theta))
cost_vector_test = np.append(cost_vector_test, compute_cost(X_test, y_test, theta))
```

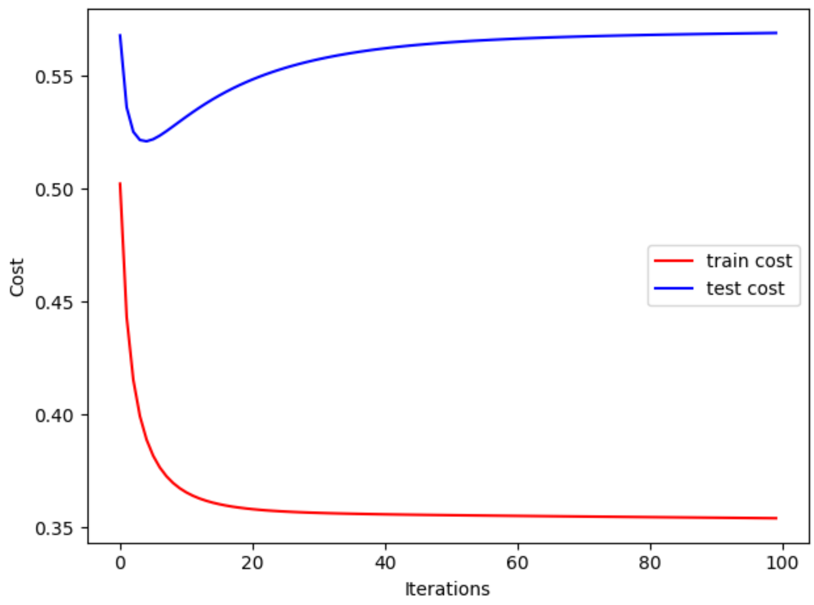
Train samples = 20 with feature features (X1,X2,X1*X2,X1^2,X2^2)



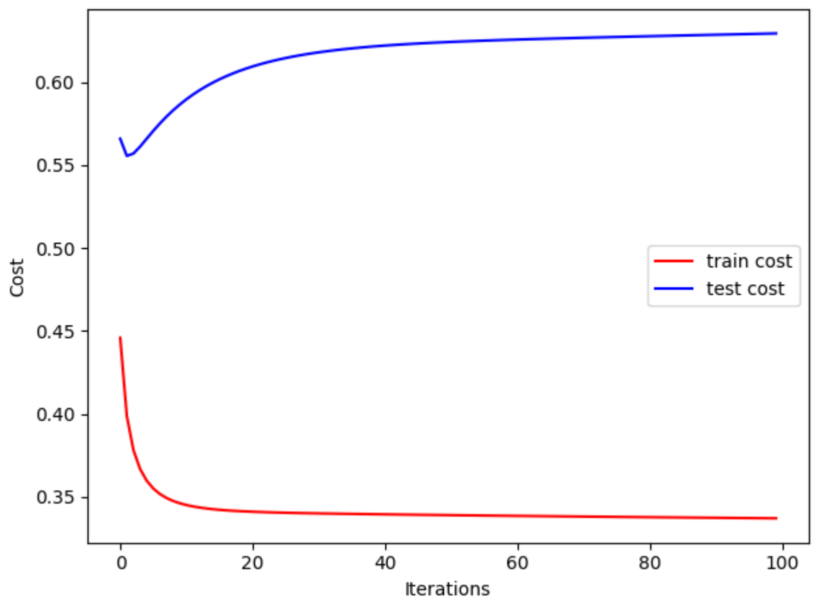
Train samples = 20 with feature features (X1,X2,X1*X2,X1^2,X2^2,X1^3,X2^3)



Train samples = 60 with feature features (X1,X2,X1*X2,X1^2,X2^2)



Train samples = 60 with feature features (X1,X2,X1*X2,X1^2,X2^2,X1^3,X2^3)

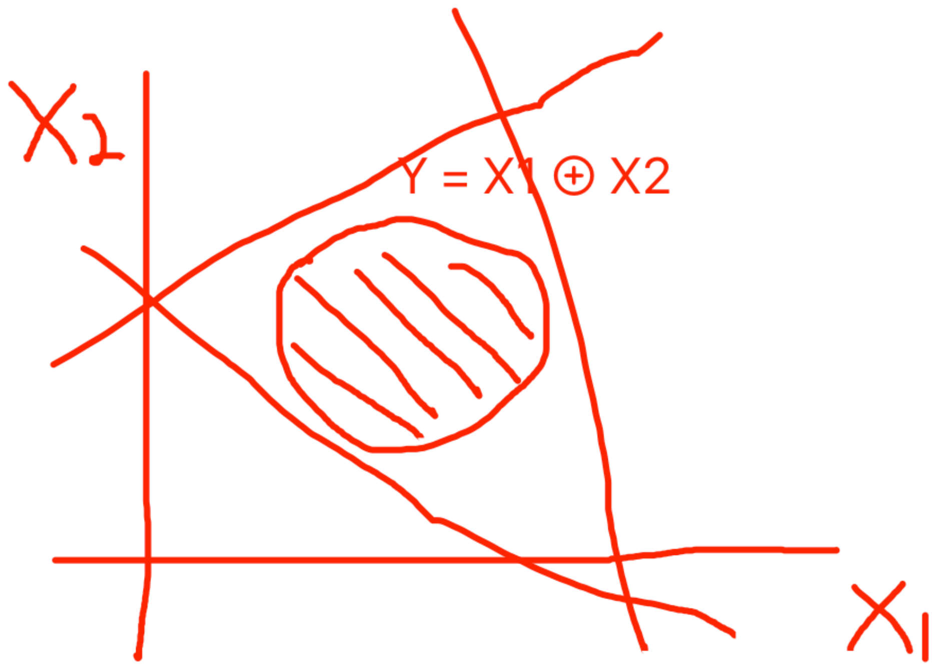


With using the same features, when there are more training samples, the overfitting comes more easily,

With using the same train samples, when there are more features, the overfitting comes more easily, Task 9. With the aid of a diagram of the decision space, explain why a logistic regression unit cannot solve the XOR classification problem.

$Y = X1 \oplus X2$

X1	X2	Y
0	0	0
1	0	1
0	1	1
1	1	0



As the diagram show, we cannot find a single lien to separate the area made from X1, X2,

So it is logistic regression cannot be used to solve the XOR classification problem.

2. Neural Network

Task 10.

Step1.

```
# 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[i] = ...
    #########################################/
    if targets.size == 1:
        delta_k = (outputs[i] - targets)*sigmoid_derivative(outputs[i])
    else:
        delta_k = (outputs[i] - targets[i])*sigmoid_derivative(outputs[i])
    output_deltas[i] = delta_k
```

Step 2.

```
# 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

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

Step 3.

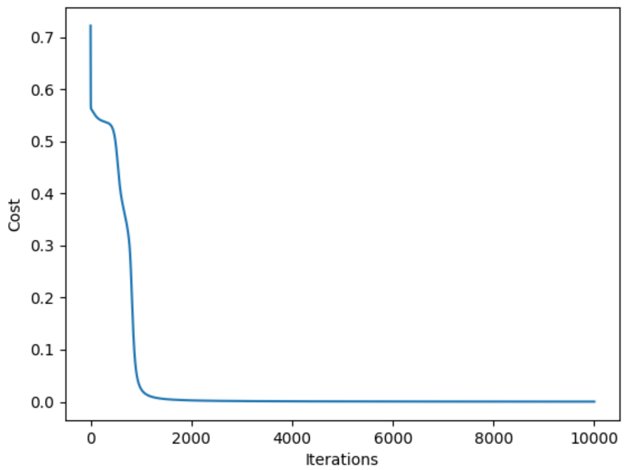
```
# Step 3. update the weights of the output layer
for i in range(len(self.y_hidden)):
    for j in range(len(output_deltas)):
        #########################################
        # Write your code here
        # update the weights of the output layer
        # self.w_out[i,j] = ...
        #########################################/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

Step 4.

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

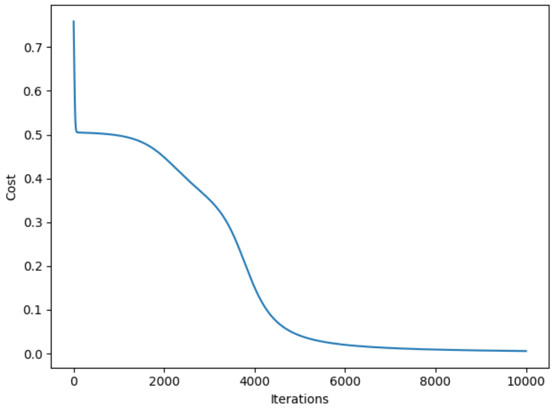
2.1. Implement backpropagation on XOR

learning rate : 1.0 iterations : 1000



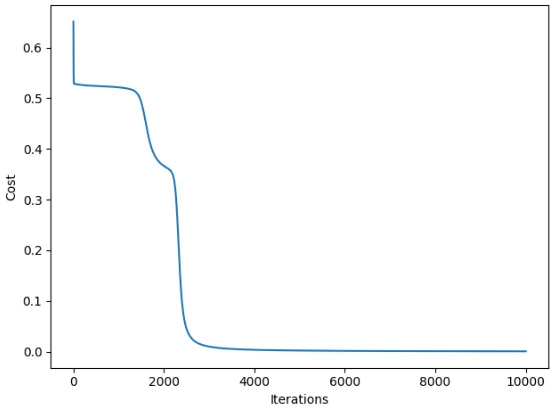

```
Iteration 09980 | Cost = 0.00029
Iteration 09990 | Cost = 0.00029
Iteration 10000 | Cost = 0.00029
Sample #01 | Target value: 0.00 | Predicted value: 0.01087
Sample #02 | Target value: 1.00 | Predicted value: 0.98865
Sample #03 | Target value: 1.00 | Predicted value: 0.98859
Sample #04 | Target value: 0.00 | Predicted value: 0.01400
Minimum cost: 0.00029, on iteration #10000
```

learning rate : 0.1 iterations : 1000



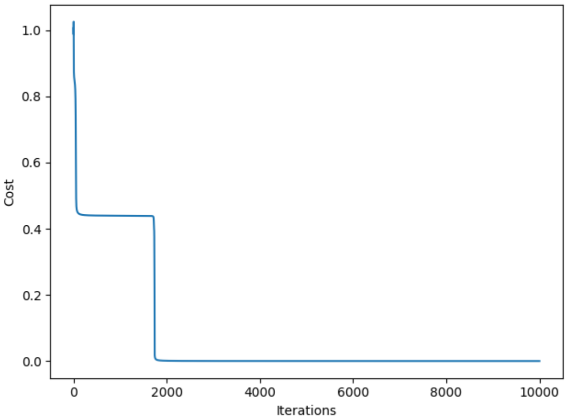

```
Iteration 09980 | Cost = 0.00571
Iteration 09990 | Cost = 0.00570
Iteration 10000 | Cost = 0.00568
Sample #01 | Target value: 0.00 | Predicted value: 0.05575
Sample #02 | Target value: 1.00 | Predicted value: 0.94890
Sample #03 | Target value: 1.00 | Predicted value: 0.94885
Sample #04 | Target value: 0.00 | Predicted value: 0.05496
Minimum cost: 0.00568, on iteration #10000
```

learning rate : 0.5 iterations : 1000




```
Iteration 09990 | Cost = 0.00070
Iteration 10000 | Cost = 0.00070
Sample #01 | Target value: 0.00 | Predicted value: 0.02044
Sample #02 | Target value: 1.00 | Predicted value: 0.98203
Sample #03 | Target value: 1.00 | Predicted value: 0.98254
Sample #04 | Target value: 0.00 | Predicted value: 0.01859
Minimum cost: 0.00070, on iteration #10000
```

learning rate : 10 iterations : 1000

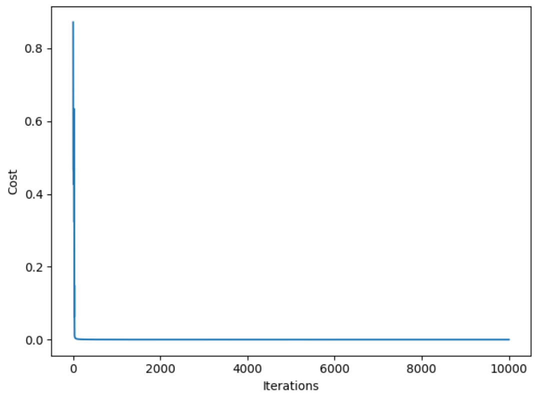



```
Iteration 09980 | Cost = 0.00003
Iteration 09990 | Cost = 0.00003
Iteration 10000 | Cost = 0.00003
Sample #01 | Target value: 0.00 | Predicted value: 0.00342
Sample #02 | Target value: 1.00 | Predicted value: 0.99644
Sample #03 | Target value: 1.00 | Predicted value: 0.99638
Sample #04 | Target value: 0.00 | Predicted value: 0.00449
Minimum cost: 0.00003, on iteration #10000
```

When we use the rate 10, we get the minimum cost

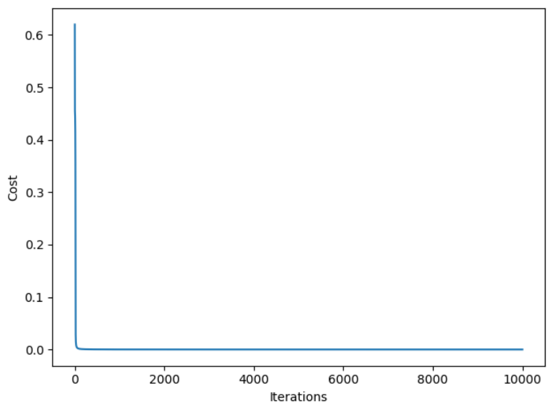
Task 11.
Change the training data in xor.m to implement a different logical function,

And, alpha = 10




```
Iteration 09980 | Cost = 0.00001
Iteration 09990 | Cost = 0.00001
Iteration 10000 | Cost = 0.00001
Sample #01 | Target value: 0.00 | Predicted value: 0.00058
Sample #02 | Target value: 0.00 | Predicted value: 0.00250
Sample #03 | Target value: 0.00 | Predicted value: 0.00250
Sample #04 | Target value: 1.00 | Predicted value: 0.99723
Minimum cost: 0.00001, on iteration #10000
```

NOR, alpha = 10




```
Iteration 09980 | Cost = 0.00001
Iteration 09990 | Cost = 0.00001
Iteration 10000 | Cost = 0.00001
Sample #01 | Target value: 1.00 | Predicted value: 0.99663
Sample #02 | Target value: 0.00 | Predicted value: 0.00176
Sample #03 | Target value: 0.00 | Predicted value: 0.00177
Sample #04 | Target value: 0.00 | Predicted value: 0.00042
Minimum cost: 0.00001, on iteration #10000
```

2.2. Implement backpropagation on Iris

Task 12.

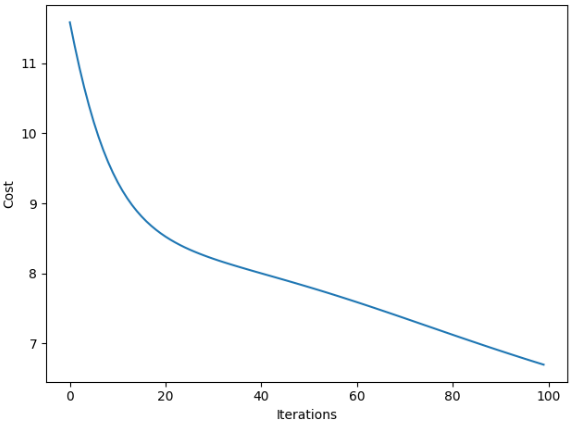
The Iris data set contains three different classes of data that we need to discriminate between. How would you accomplish this if we used a logistic regression unit? How is this scenario different, compared to the scenario of using a neural network?

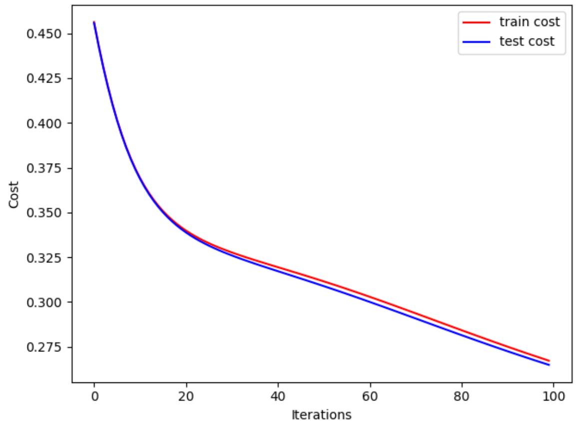
We can use 1-vs-All of Logistic regressions to solve this problem. For neural network. We can complete multi-label classification by using multiple output units in 1-of-K encoding.

.

Task 13.

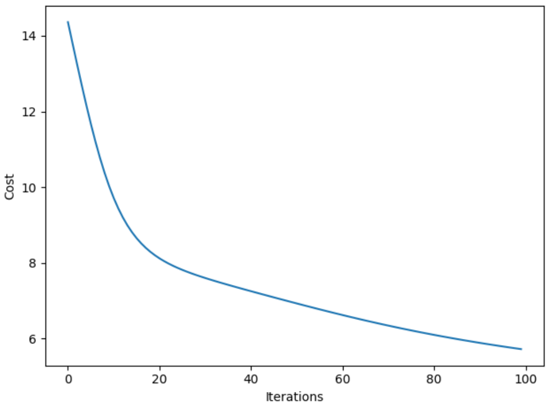
Hidden Neutron 2 alpha 0.01

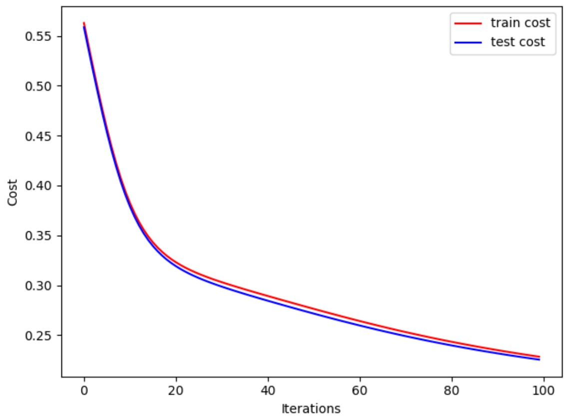




Minimum cost: 6.69489, on iteration #100

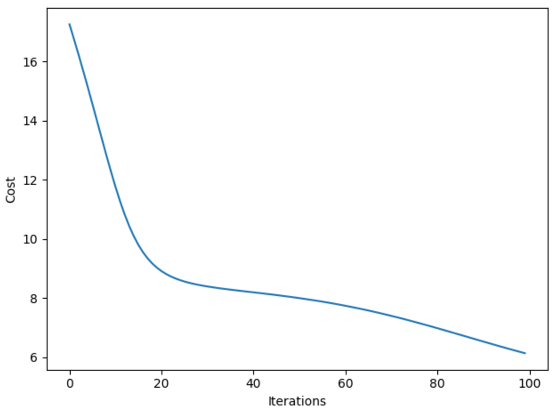
Hidden Neutron 2 alpha 0.01

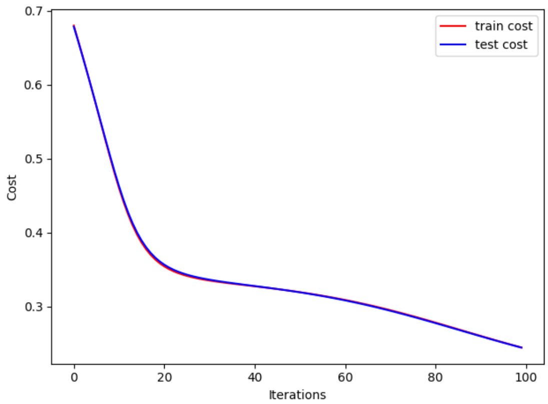




Minimum cost: 5.66102, on iteration #100

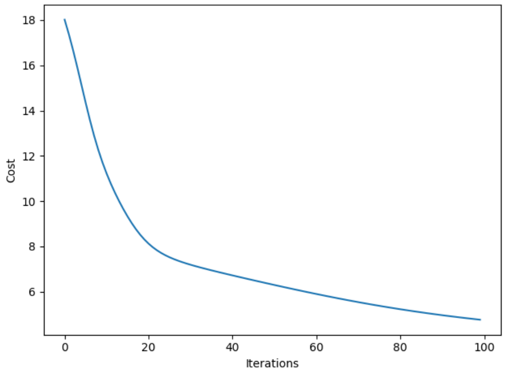
Hidden Neutron 3 alpha 0.01

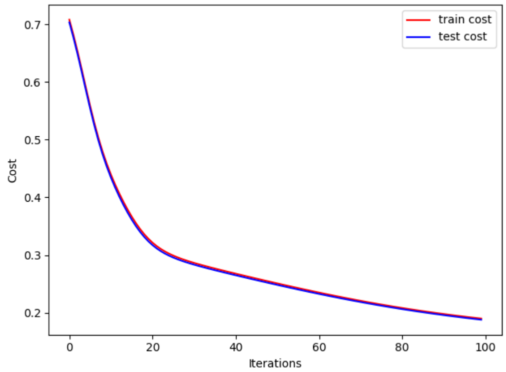




Minimum cost: 6.13993, on iteration #100

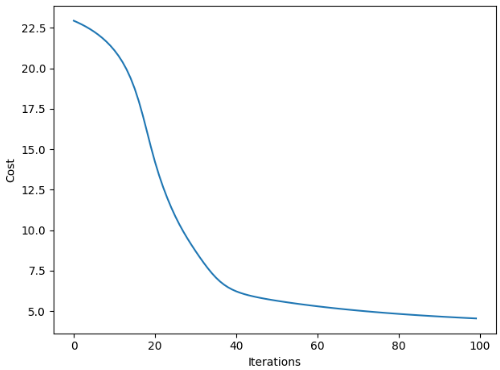
Hidden Neutron 5 alpha 0.01

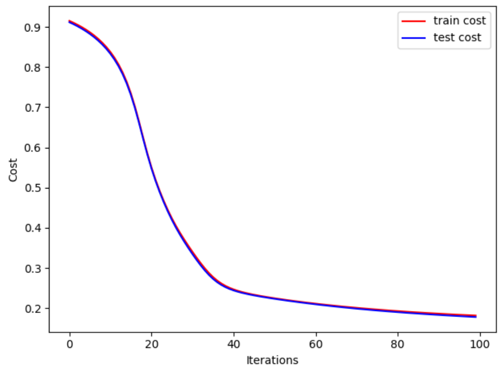




Minimum cost: 4.76552, on iteration #100

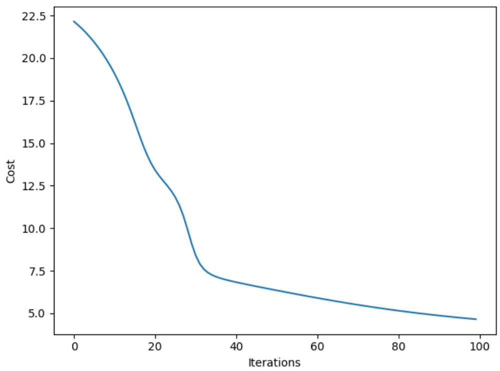
Hidden Neutron 7 alpha 0.01

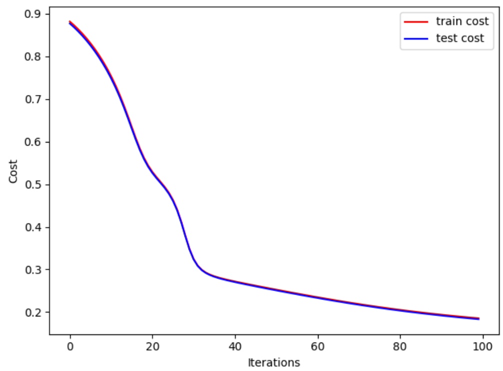




Minimum cost: 4.55354, on iteration #100

Hidden Neutron 10 alpha 0.01





Minimum cost: 4.65424, on iteration #100

According to the charts above, the best number should be 7, as the number of hidden neutron is 7 ,we get the minimum cost.