Machine Learning

Assignment1

Part2

Spyridon Roumpis

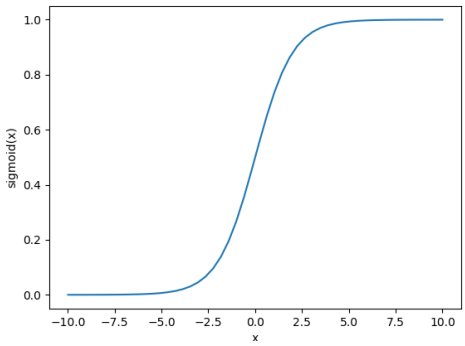
181004877

1. Logistic Regression

Task1

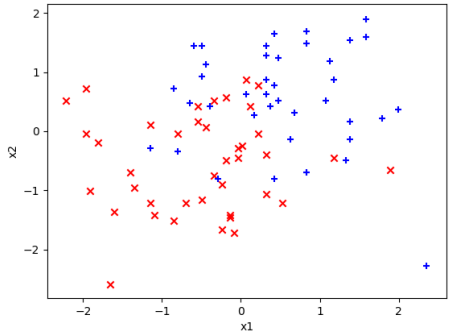
In this task we had to modify the sigmoid.py function and then to plot it. The logistic/sigmoid function is used to form the hypothesis in logistic regression in the binary classification problem.

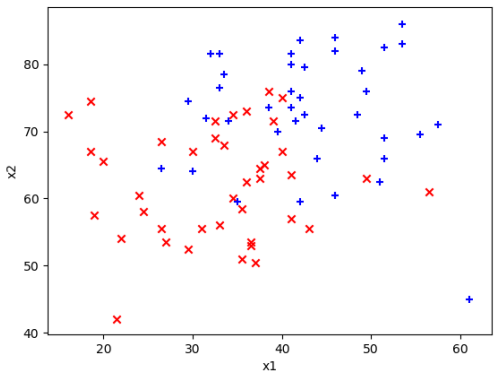
```
def sigmoid(z):

    output = 0.0
    # modify this to return z passed through the sigmoid function
    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.





On the left hand size it can be seen the normalized data while on the right the data without the normalization.

1.1. Cost function and gradient for logistic regression

Task 3

Modify the calculate_hypothesis.py function so that for a given dataset, theta and training example it returns the hypothesis. 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

    z=np.dot(X[i,:],theta)
    hypothesis = sigmoid(z)

    return hypothesis
```

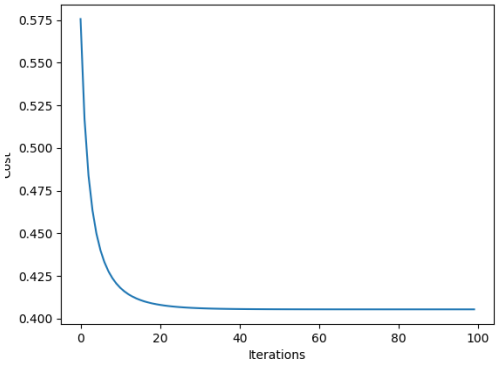
Task 4

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

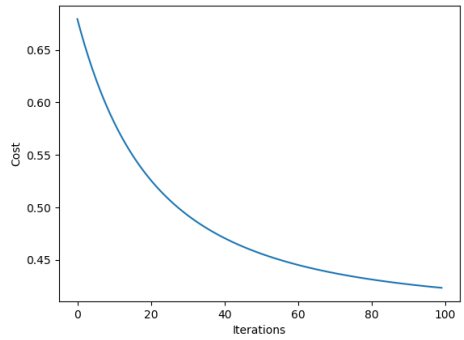
$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(-y^{(i)}log(h_\theta(x^{(i)})) - (1-y^{(i)})log(1-h_\theta(x^{(i)}))\right)$$

```
def compute_cost(X, y, theta):

    # 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

        # You must calculate the cost
        cost = (-output * np.log(hypothesis)) - ((1-output)*np.log(1-hypothesis))

        J += cost
    J = J/m

    return J
```

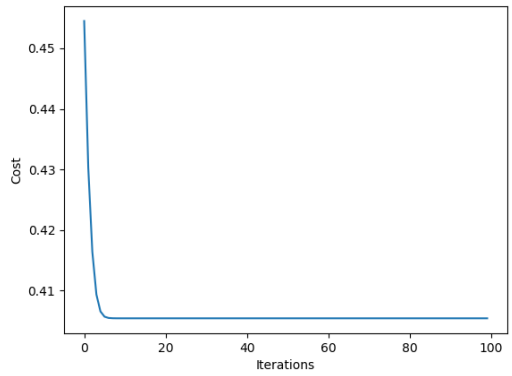
For a=1 → Minimum cost: 0.40545, on iteration #100



For a=0.1 → Minimum cost: 0.42342, on iteration #100



For a=10 → Minimum cost: 0.40545, on iteration #33

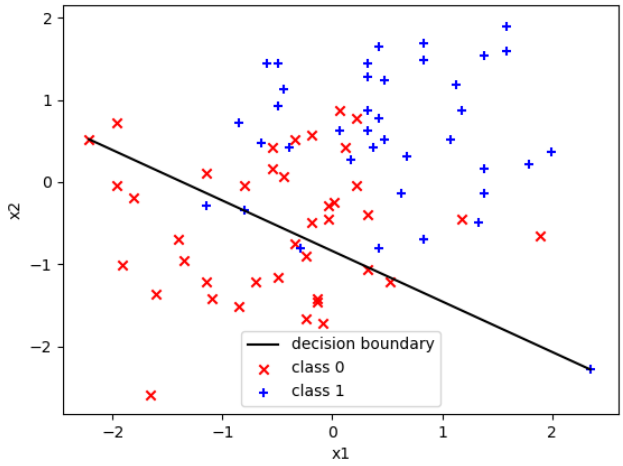


It can be seen that for a= 10 we get the minimum cost from the 33 iteration, so we will pick it as the optimal.

1.2. Draw the decision boundary

Task 5.

Plot the decision boundary. To plot the line of the boundary, we'll need two points of (x1,x2). Given a known value for x1, you can find the value of x2. Rearrange the equation in terms of x2 to do that. We used the minimum and maximum values of x1 as the known values, so that the boundary line that we'll plot, will span across the whole axis of x1. For these values of x1, compute the values of x2.



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
for i in range(len(X)):
    if X[i,1] < min_x1:
        min_x1 = X[i,1]
        x2_on_min_x1 = X[i,2]
    if X[i,1] > max_x1:
        max_x1 = X[i,1]
        x2_on_max_x1 = X[i,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()
```

1.3. Non-linear features and overfitting

We don't always have access to the full dataset. In assgn1_ex2.py, the dataset has been split into a training set of 20 samples, and the rest samples have been assigned to the test set. This split has been done using the function return_test_set.py. Gradient descent is run on the training data (this means that the parameters are learned using only the training set and not the test set). After theta has been calculated, compute_cost() is called on both datasets (training and test set) and the error is printed, as well as graphs of the data and boundaries shown.

Task 6

In this task we have to run the code of assgn1_ex2.py several times. In every execution, the data are shuffled randomly, so we'll see different results. We will report the costs found over the multiple runs and we will answer the below questions. 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.

- First Iteration

Final training cost: 0.20343

Minimum training cost: 0.20343, on iteration #100

Final test cost: 0.54229

- Second Iteration

Final training cost: 0.29777

Minimum training cost: 0.29777, on iteration #100

Final test cost: 0.66736

- Third Iteration

Final training cost: 0.33528

Minimum training cost: 0.33528, on iteration #100

Final test cost: 0.51008

- Fourth Iteration

Final training cost: 0.35202

Minimum training cost: 0.35202, on iteration #100

Final test cost: 0.48116

- Fifth Iteration

Final training cost: 0.40305

Minimum training cost: 0.40305, on iteration #100

Final test cost: 0.53014

- Sixth Iteration

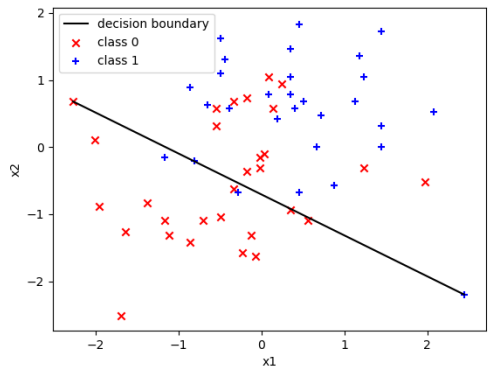
Final training cost: 0.48268

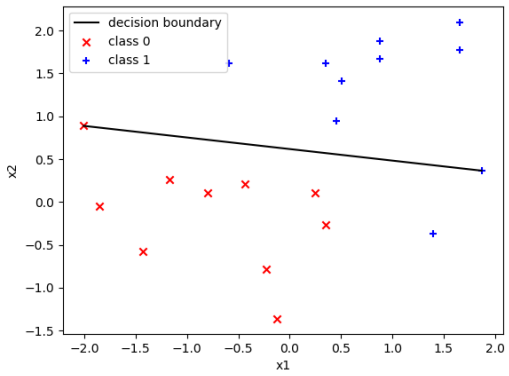
Minimum training cost: 0.48268, on iteration #100

Final test cost: 0.40919

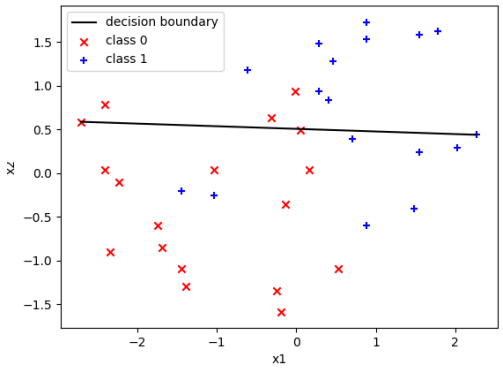
The general trend is that Test Cost tends to be always bigger than the training cost.

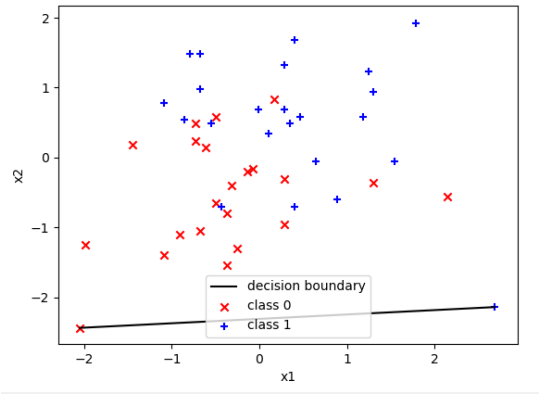
Good Split





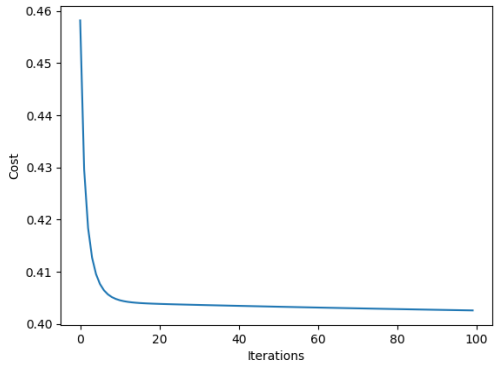
Bad Split





Task 7

Run logistic regression on this dataset, assgn1_ex3.py.



Final cost: 0.40261

Minimum cost: 0.40261, on iteration #100

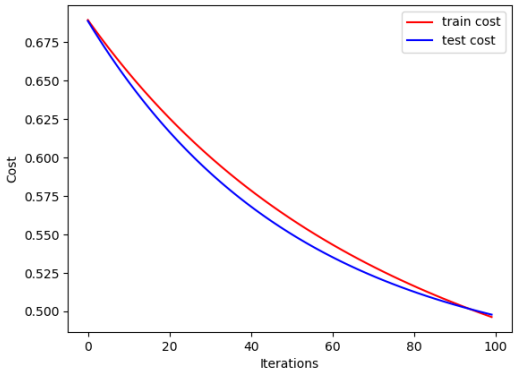
In general, because we are using a 5d vector instead of a 2d, as we did in the previous task, our classifier performs better, this is also what we expected as we have more information to ‘ feed’ .

Task 8

In assgn1_ex4.py the data are split into a test set and a training set. We will add our new features from the question above (assgn1_ex3.py). Then we will modify the function gradient_descent_training.py to store the current cost for the training set and testing set and we will store the cost of the training set to cost_vector_train and for the test set to 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). We will experiment with different sizes of training and test set and will show the effect of using sets of different sizes by the graphs.

- Train samples = 50



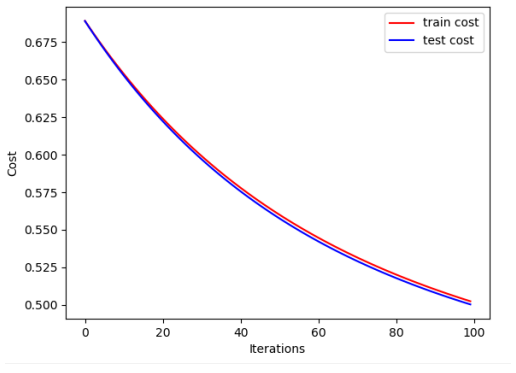
Final train cost: 0.49635

Minimum train cost: 0.49635, on iteration #100

Final test cost: 0.49793

Minimum test cost: 0.49793, on iteration #100

- Train samples = 30



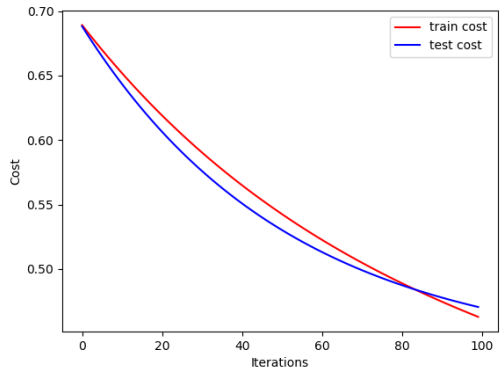
Final train cost: 0.50235

Minimum train cost: 0.50235, on iteration #100

Final test cost: 0.50025

Minimum test cost: 0.50025, on iteration #100

- Train samples = 20



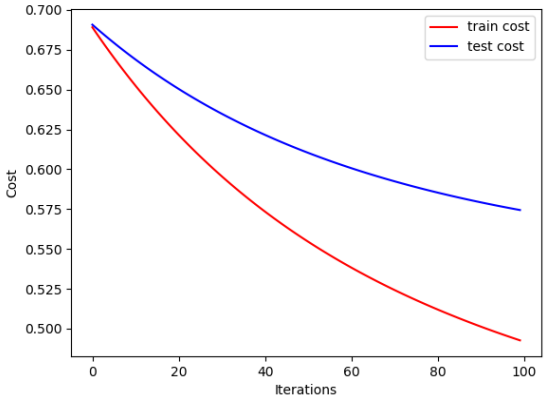
Final train cost: 0.46297

Minimum train cost: 0.46297, on iteration #100

Final test cost: 0.47061

Minimum test cost: 0.47061, on iteration #100

- Train samples = 70



Final train cost: 0.49266

Minimum train cost: 0.49266, on iteration #100

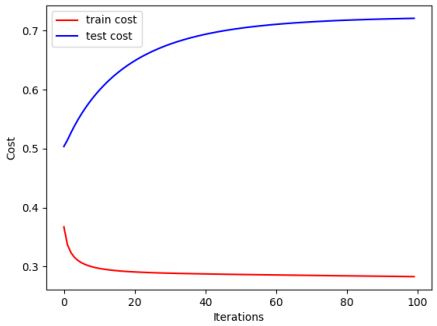
Final test cost: 0.57445

Minimum test cost: 0.57445, on iteration #100

It can be seen from the graphs that the best split is 30 training samples and 50 test samples.

In assgn1_ex5.py, we added extra features (e.g. both a second-order and a third-order polynomial).

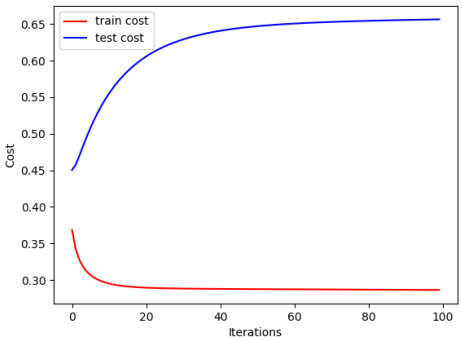
- Train samples = 20



Final train cost: 0.28310 , Minimum train cost: 0.28310, on iteration #100

Final test cost: 0.72075, Minimum test cost: 0.50365, on iteration #1

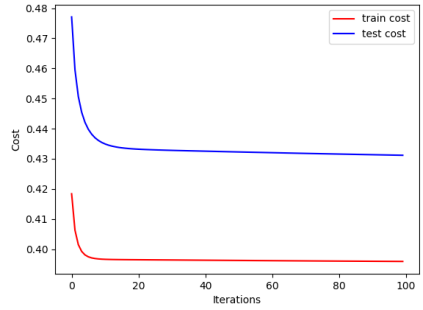
- Train samples = 30



Final train cost: 0.28640, Minimum train cost: 0.28640, on iteration #100

Final test cost: 0.65638, Minimum test cost: 0.45038, on iteration #1

- Train samples = 70

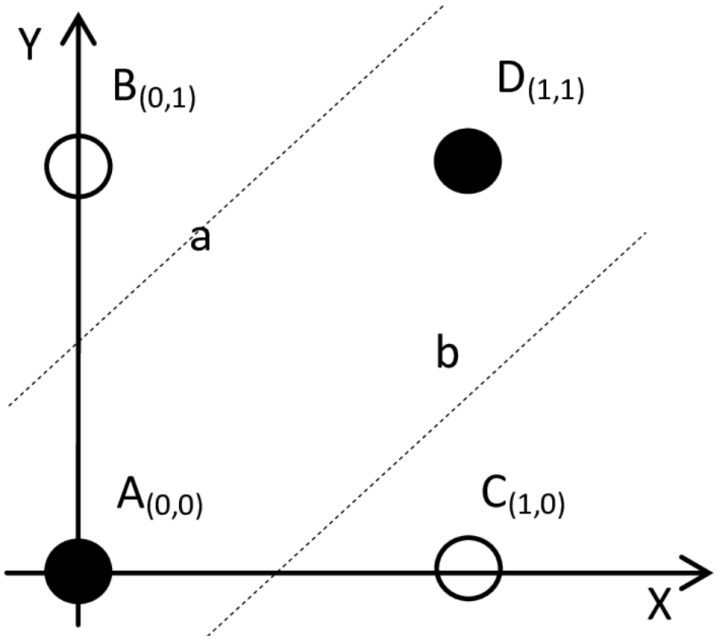


Final train cost: 0.39594, Minimum train cost: 0.39594, on iteration #100

Final test cost: 0.43118, Minimum test cost: 0.43118, on iteration #100

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 main reason is that we cannot draw a straight line, boundary, to separate the points (0,0),(1,1) from the points (0,1),(1,0). Without being able to have a boundary to separate our data set we cannot implement logistic regression, or even if we could and get a random boundary our classifier would behave very badly.

2. Neural Network

Task 10

Implementing backpropagation's code, by filling the backward_pass() function, found in NeuralNetwork.py.

```
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
    output_deltas = np.zeros((self.n_out))
    outputs = self.y_out.copy()
    #targets = self.targets.copy()

    for i in range(self.n_out):
      # Write your code here
      if np.isscalar(targets) == True:
        output_deltas[i] = (outputs[i]-targets)*sigmoid_derivative(outputs[i])
      else:
        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 i in range(len(hidden_deltas)):
      ########################################
```

```
    # Write your code here
    # compute hidden_deltas
    s=0
    for j in range(self.n_out):

        s+=(self.w_out)[i,j]*output_deltas[j]

    hidden_deltas[i]=sigmoid_derivative(self.y_hidden[i])*s
```

 # 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]

# 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:]
```

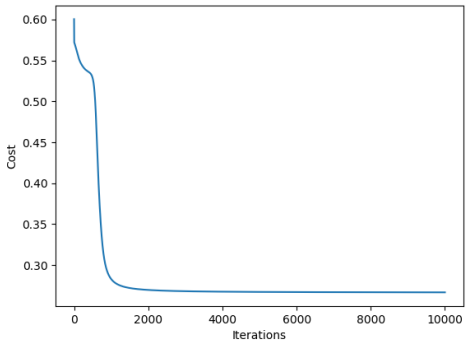
 # Step 4. update the weights of the hidden layer

```
# 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)):
        # 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]*inputs[i]

return J
```

Now that the backpropagation process has been implemented, we will run some experiments for different values of the learning rate.

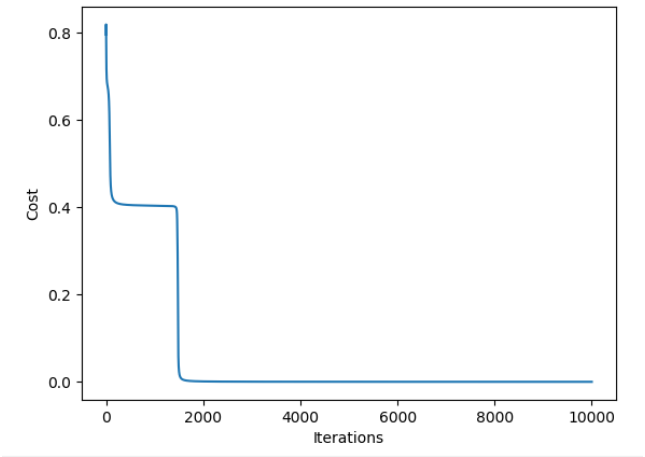
- **a=1**

Sample #01 | Target value: 0.00 | Predicted value: 0.01169
Sample #02 | Target value: 1.00 | Predicted value: 0.48360
Sample #03 | Target value: 1.00 | Predicted value: 0.98800
Sample #04 | Target value: 0.00 | Predicted value: 0.48410
Minimum cost: 0.26677, on iteration #10000



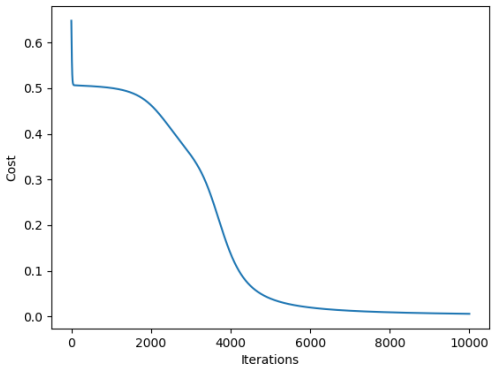
- **a=5**

Sample #01 | Target value: 0.00 | Predicted value: 0.00476
Sample #02 | Target value: 1.00 | Predicted value: 0.99499
Sample #03 | Target value: 1.00 | Predicted value: 0.99495
Sample #04 | Target value: 0.00 | Predicted value: 0.00628
Minimum cost: 0.00006, on iteration #10000



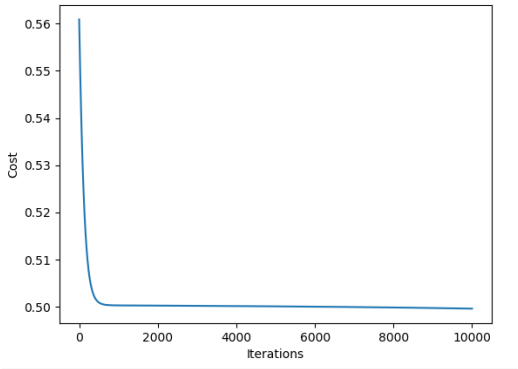
- **a=0.1**

Sample #01 | Target value: 0.00 | Predicted value: 0.05577
Sample #02 | Target value: 1.00 | Predicted value: 0.94909
Sample #03 | Target value: 1.00 | Predicted value: 0.94892
Sample #04 | Target value: 0.00 | Predicted value: 0.05466
Minimum cost: 0.00566, on iteration #10000



- **a=0.01**

Sample #01 | Target value: 0.00 | Predicted value: 0.49218
Sample #02 | Target value: 1.00 | Predicted value: 0.50290
Sample #03 | Target value: 1.00 | Predicted value: 0.50033
Sample #04 | Target value: 0.00 | Predicted value: 0.50895
Minimum cost: 0.49965, on iteration #10000



It can be seen from the above graphs that the smaller the learning rate is the worse our predictions are. The best learning rate is for a=5.

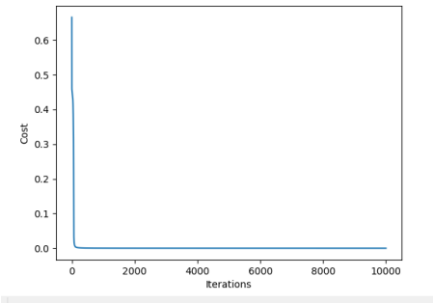
For a=0.01 our function does not get stuck in a local optima and the predictions are really bad. In the other cases, we see that our error function gets stuck in a local optimum. The best case is where the a=5. The metrics for these can be seen above.

Task 11

In this task we will change the training data in xor.m to implement a different logical function, AND.

- **a=5**

Sample #01 | Target value: 0.00 | Predicted value: 0.00015
Sample #02 | Target value: 0.00 | Predicted value: 0.00266
Sample #03 | Target value: 0.00 | Predicted value: 0.00267
Sample #04 | Target value: 1.00 | Predicted value: 0.99613
Minimum cost: 0.00001, on iteration #10000



2.2. Implement backpropagation on Iris

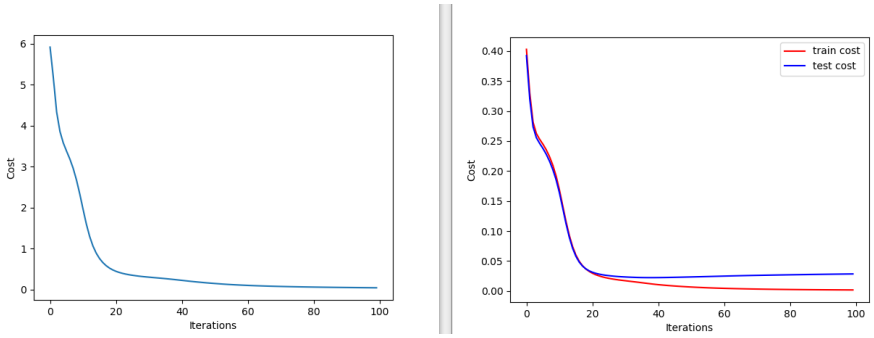
Task 12

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?

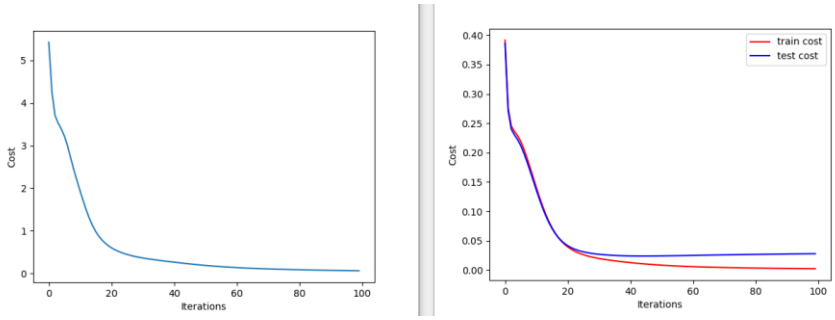
The main difference between our logistic classifier and the neural network is that our classifier is binary, and the hypothesis was implemented with the sigmoint function. That means, it can only have two values as a result, that means that our data will either belong to class 1 or to class 2. For the iris dataset that's not a good solution, as the iris dataset has three different classes that needs to be classified. The neural network can have as many inputs and outputs as it needs. Finally if I had to use logistic regression on the iris dataset I would use a multi-class logistic classifier like maxent and not a binary one.

Task 13

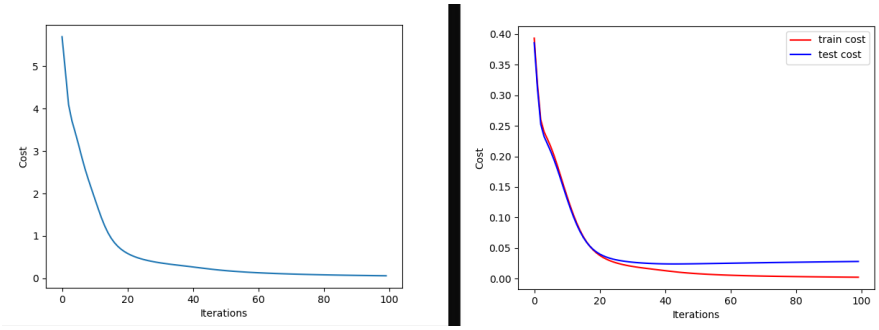
- hidden neurons: 1 → Minimum cost: 0.04466, on iteration #100



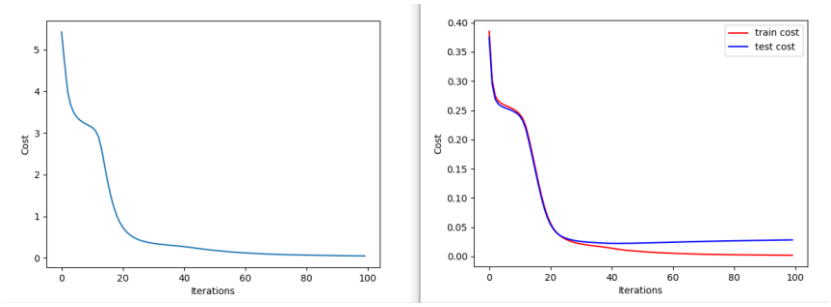
- hidden neurons: 2 → Minimum cost: 0.06178 on iteration #100



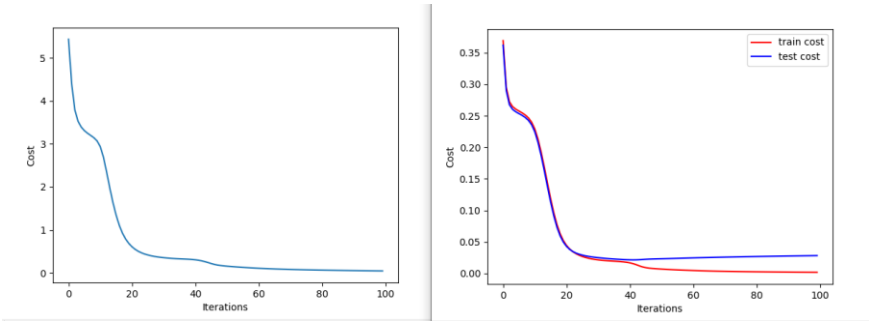
- hidden neurons: 3 → Minimum cost: 0.06066 on iteration #100



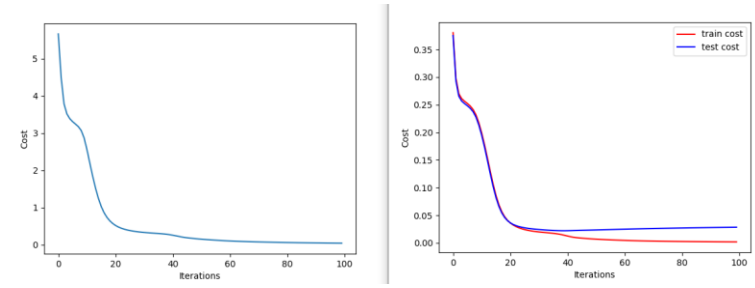
- hidden neurons: 5 → Minimum cost: 0.04791 on iteration #100



- hidden neurons: 7 → Minimum cost: 0.04627 on iteration #100



- hidden neurons: 3 → Minimum cost: 0.04543 on iteration #100



Deciding the number of neurons in the hidden layers is a very important part and must be carefully considered. Using too few neurons in the hidden layers will result in underfitting while Using too many neurons in the hidden layers may result in overfitting. Also using a large number of neurons in the hidden layers can increase the time it takes to train the network.

Another fact to consider on which number of neurons in the hidden layer to use is the cost function, we see that for 5 neurons we get a really good result compared to the number of neurons less than five. For the number of neurons that are bigger than five the cost function doesn't change much and because of the overfitting phenomenon we don't want to overdo it and choose a very high number of neurons, so I think 5 is the best solution at this point.