

**ASSIGNMENT 1, PART 2:  
LOGISTIC REGRESSION  
AND NEURAL  
NETWORKS.  
Machine Learning.**

Carolina Raymond Rodrigues.

Student ID: 210607000

ec21209@qmul.ac.uk

## **Index**

<b>1. Logistic Regression.....</b>	<b>4</b>
Task 1. ....	4
Task 2. ....	5
Task 3. ....	5
Task 4. ....	5
Task 5. ....	7
Task 6. ....	8
Task 7. ....	11
Task 8. ....	12
Task 9. ....	15
<b>2. Neural Network.....</b>	<b>17</b>
Task 10. ....	17
Task 11.. ....	18
Task 12. ....	20
Task 13.?. ....	20

## **Figures Index**

<b>Fig. 1</b> sigmoid function code implementation.....	<b>4</b>
<b>Fig. 2</b> Sigmoid function plot. ....	<b>4</b>
<b>Fig. 3</b> Plots with (a) and without normalised (b) data.....	<b>5</b>
<b>Fig. 4</b> calculate_hypothesis function.....	<b>5</b>
<b>Fig. 5</b> Compute cost function.....	<b>6</b>
<b>Fig. 6</b> Gradient descent function with logistic regression.....	<b>6</b>
<b>Fig. 7</b> Decision boundary calculation.....	<b>8</b>
<b>Fig. 8</b> Decision boundary plot.....	<b>8</b>
<b>Fig. 9</b> Run #1: Test set (a), train set (b) and cost (c) plots. ....	<b>9</b>
<b>Fig. 10</b> Run # 2: Test set (a), train set (b) and cost (c) plots. ....	<b>10</b>
<b>Fig. 11</b> Run # 3: Test se (a), train set (b) and cost (c) plots. ....	<b>10</b>
<b>Fig. 12</b> ml_assgn1_1_ex3.py with 5D input vector .....	<b>11</b>
<b>Fig. 13</b> Cost of the training and test data for a six-degree polynomial. ....	<b>14</b>
<b>Fig. 14</b> Cost of the training and test data for a third-degree polynomial .....	<b>14</b>
<b>Fig. 15</b> Label 1 logistic unit .....	<b>15</b>
<b>Fig. 16</b> OR logistic unit.....	<b>15</b>
<b>Fig. 17</b> AND logistic unit.....	<b>16</b>
<b>Fig. 18</b> Label 0 logistic unit .....	<b>16</b>
<b>Fig. 19</b> Backpropagation function.....	<b>17</b>
<b>Fig. 20</b> Output of the neural network. ....	<b>18</b>
<b>Fig. 21</b> Cost function of the neural network with $\alpha = 0.3$ .....	<b>18</b>
<b>Fig. 22</b> Output when backpropagation is stuck in local minima.....	<b>18</b>

<b>Fig. 23</b> AND predicted values with the neural network.....	19
<b>Fig. 24</b> AND cost graph for neural networks. ....	19
<b>Fig. 25</b> OR predicted values with the neural network.....	19
<b>Fig. 26</b> NOR cost plot for neural networks. ....	20

## **Tables Index**

<b>Table 1</b> Prediction plot, cost graph and minimum cost and iteration for different values of alpha for logistic regression. ....	7
<b>Table 2</b> Cost plot comparison between 2D and 5D input features. ....	12
<b>Table 3</b> Cost of the training and test set for different sizes of data. ....	13
<b>Table 4</b> XOR truth table .....	15
<b>Table 5</b> irisExample.py with a different number of hidden neurons. ....	22

## 1. Logistic Regression.

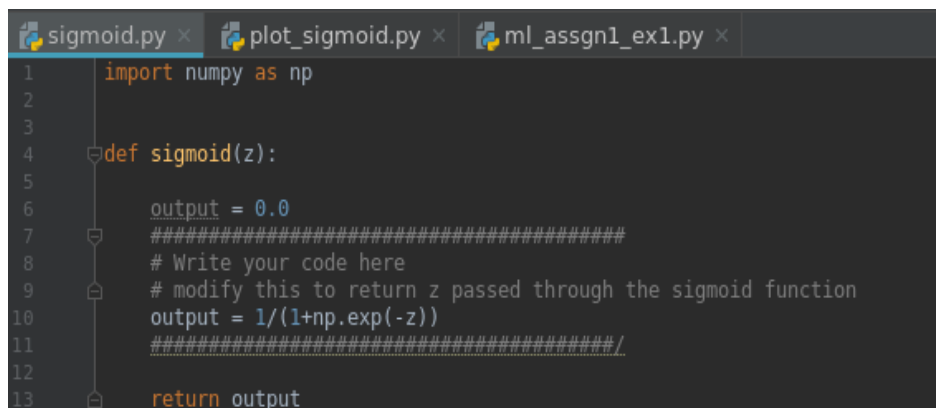
**Task 1** Fill out the sigmoid.py function. Now use the plot\_sigmoid.py function to plot the sigmoid function. Include in your report the relevant lines of code and the result of using plot\_sigmoid.py.

In Machine Learning, logistic regression is used to predict to which category the input data belongs. Since the outcome of this function is binary, it will be used in the proposed problem, where data will be classified into two classes.

Logistic regression uses the sigmoid equation (1) as the hypothesis function to predict to which class the data belongs. The sigmoid function is defined as it follows:

$$f(w) = \frac{1}{1 + e^{-w^T x}} \quad (1)$$

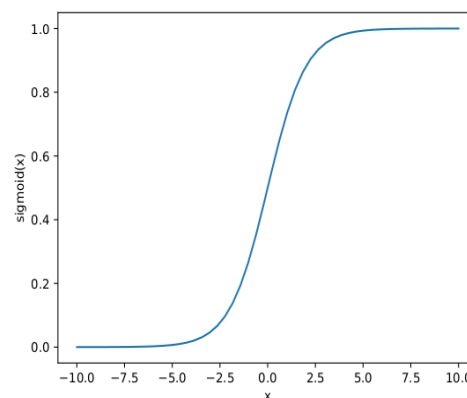
This function will be added to the code as it can be seen in Fig. 1, line 10.



```
1 import numpy as np
2
3
4 def sigmoid(z):
5
6     output = 0.0
7     #####
8     # Write your code here
9     # modify this to return z passed through the sigmoid function
10    output = 1/(1+np.exp(-z))
11    #####
12
13    return output
```

*Fig. 1 sigmoid function code implementation.*

Next, *plot\_sigmoid.py* will be run. As it can be seen in Fig. 2, the range of the sigmoid is [0,1], therefore if  $x = \infty$  and  $w = 1$ , the output of the sigmoid is 1 and consequently the data will be classified with label 1. In contrast, if  $x = -\infty$  and  $w = 1$ , the data will be categorised with label 0.



*Fig. 2 Sigmoid function plot.*

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

The goal of normalisation is to transform the input features to be on a similar scale. Doing this will allow the data to be easily optimised. As it can be seen in Fig. 3 the input features  $x_1$  and  $x_2$  are in a similar scale after being normalised.

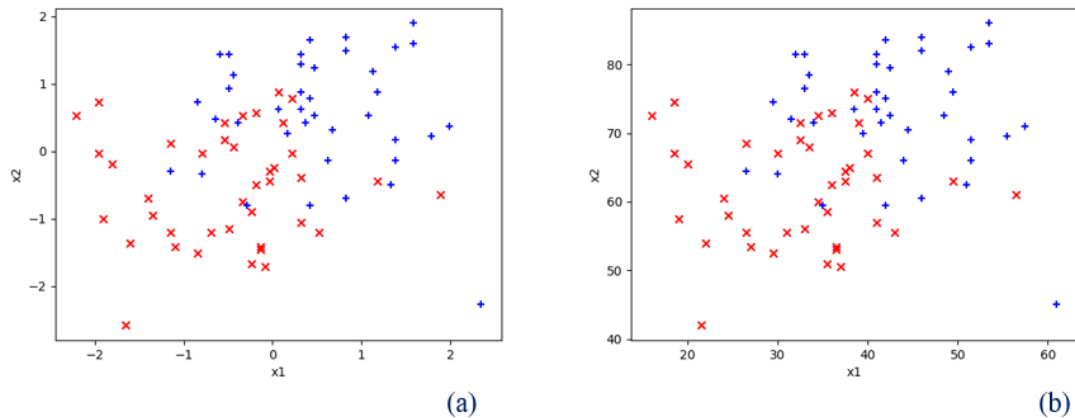


Fig. 3 Plots with (a) and without normalised (b) data.

**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.

The hypothesis function is defined as:

$$z = w^T x \quad (1)$$

Consequently, as it can be seen in Fig. 4, line 15 of `calculate_hypothesis.py` will include the previous formula.

```

sigmoid.py x calculate_hypothesis.py x compute_cost.py x ml_assgn1_ex1.py x
1  import numpy as np
2  from sigmoid import *
3
4  def calculate_hypothesis(X, theta, i):
5      """
6          :param X      : 2D array of our dataset
7          :param theta   : 1D array of the trainable parameters
8          :param i       : scalar, index of current training sample's row
9      """
10     hypothesis = 0.0
11     #####
12     # Write your code here
13     # You must calculate the hypothesis for the i-th sample of X, given X, theta and i.
14     for j in range(len(X[0])):
15         hypothesis += X[i, j]*theta[j]
16     #####
17
18     result = sigmoid(hypothesis)
19
20     return result

```

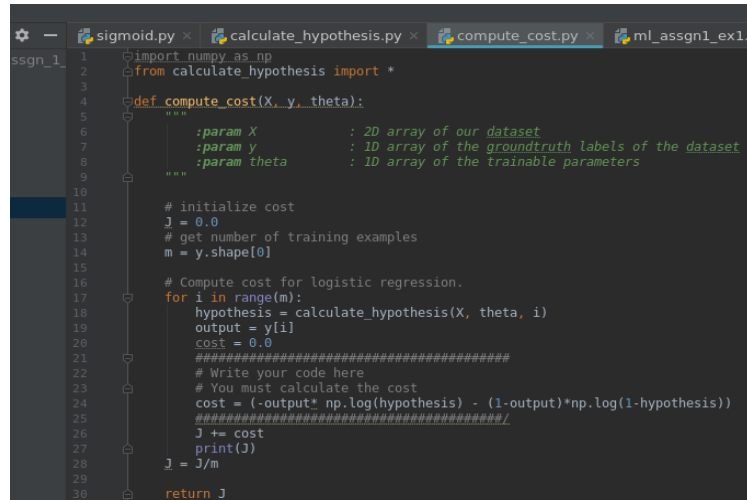
Fig. 4 `calculate_hypothesis` function.

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

The negative likelihood for logistic regression is defined as:

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

The previous formula is added to *compute\_cost.py*, as can be seen in line 24 of Fig. 5.



```

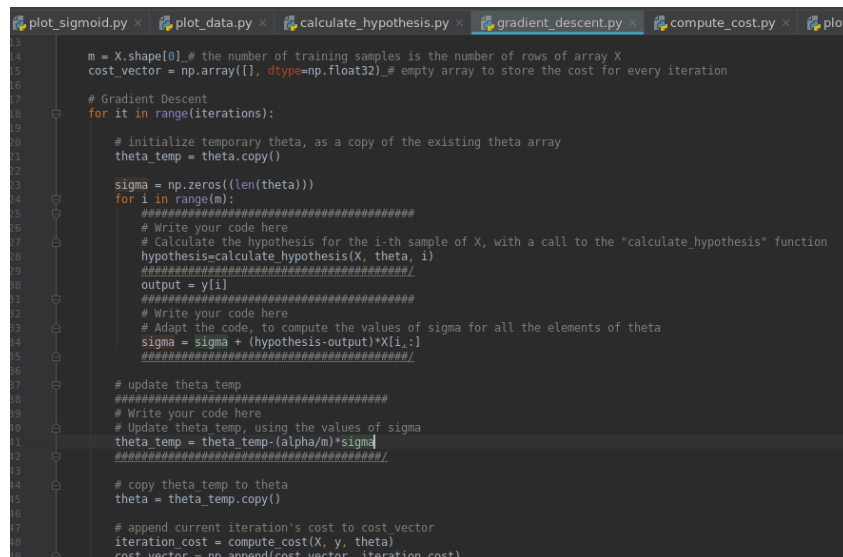
1  import numpy as np
2  from calculate_hypothesis import *
3
4  def compute_cost(X, y, theta):
5      """
6      :param X      : 2D array of our dataset
7      :param y      : 1D array of the groundtruth labels of the dataset
8      :param theta   : 1D array of the trainable parameters
9      """
10
11     # initialize cost
12     J = 0.0
13     # get number of training examples
14     m = y.shape[0]
15
16     # Compute cost for logistic regression.
17     for i in range(m):
18         hypothesis = calculate_hypothesis(X, theta, i)
19         output = y[i]
20         cost = 0.0
21         #####
22         # Write your code here
23         # You must calculate the cost
24         cost = (-output * np.log(hypothesis) - (1-output) * np.log(1-hypothesis))
25         #####
26         J += cost
27         print(J)
28     J = J/m
29
30     return J

```

Fig. 5 Compute cost function

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.

Before running the code, *gradient\_descent.py* must be adapted to the changes made in Fig. 4 and 5. In the following image, it can be seen how lines 25 to 41 have added the new hypothesis and sigma calculation, and how the weights ( $\theta$ ) are updated after with logistic regression.



```

1  m = X.shape[0] # the number of training samples is the number of rows of array X
2  cost_vector = np.array([], dtype=np.float32) # empty array to store the cost for every iteration
3
4  # Gradient Descent
5  for it in range(iterations):
6
7      # initialize temporary theta, as a copy of the existing theta array
8      theta_temp = theta.copy()
9
10     sigma = np.zeros((len(theta)))
11     for i in range(m):
12         #####
13         # Write your code here
14         # Calculate the hypothesis for the i-th sample of X, with a call to the "calculate_hypothesis" function
15         hypothesis=calculate_hypothesis(X, theta, i)
16         #####
17         output = y[i]
18         #####
19         # Write your code here
20         # Adapt the code, to compute the values of sigma for all the elements of theta
21         sigma = sigma + (hypothesis-output)*X[i,:]
22         #####
23
24     # update theta temp
25     #####
26     # Write your code here
27     # Update theta temp, using the values of sigma
28     theta_temp = theta_temp - (alpha/m)*sigma
29     #####
30
31     # copy theta_temp to theta
32     theta = theta_temp.copy()
33
34     # append current iteration's cost to cost_vector
35     iteration_cost = compute_cost(X, y, theta)
36     cost_vector = np.append(cost_vector, iteration_cost)

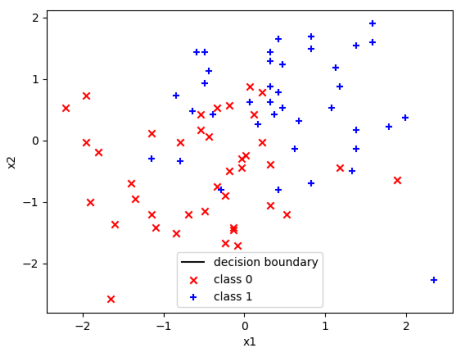
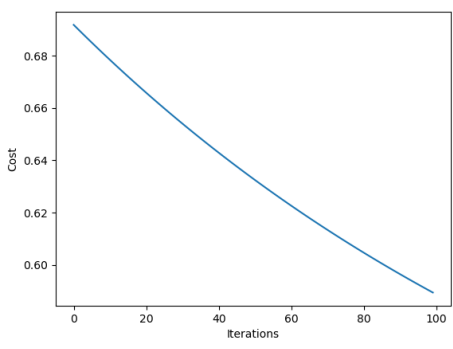
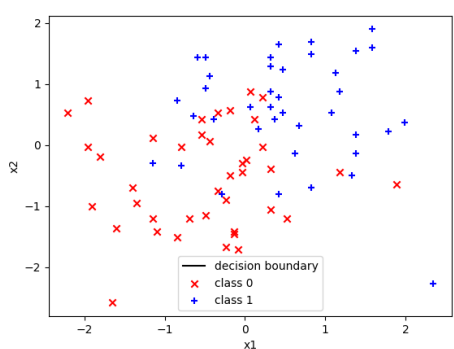
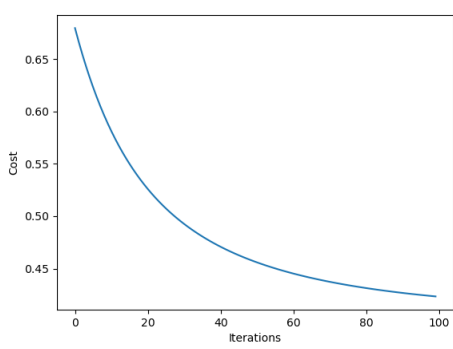
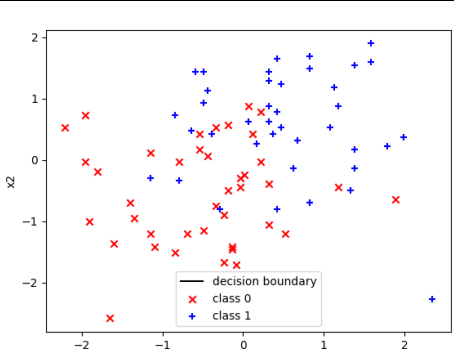
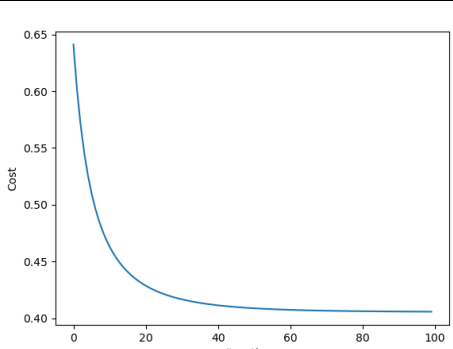
```

Fig. 6 Gradient descent function with logistic regression.

Once the bits of code have been added, it will be run. Table 1 summarises the results obtained for different tuning values of  $\alpha$ . As it can be seen in the cost plots, for  $\alpha = 0.01$  and  $\alpha = 0.1$ , the

learning rate is too small, especially for the former one. The value of the cost slowly decreases with time, which implies that it will take some time to reach convergence.

However,  $\alpha = 0.4$ , seems to be a good learning rate. For the cost graph of this value, an exponential decay can be seen, therefore it is likely that 0.4 is the right learning rate since the cost graph demonstrates that it is converging to the optimal value.

$\alpha$	Prediction	Cost	Minimum cost and iteration
0.01			C:0.5894 It: #100
0.1			C: 0.4232 It #100
0.4			C: 0.405 It #100

**Table 1** Prediction plot, cost graph and minimum cost and iteration for different values of alpha for logistic regression.

**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.

To plot the boundary line equation, we will take into account the maximum and minimum values of  $x_1$ . Once we have these values, the next step is to calculate the value of  $x_2$ . For that, equation (2) will be used:

$$\theta^T x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0 \rightarrow x_2 = -\frac{\theta_0 + \theta_1 x_1}{\theta_2} \quad (2)$$

Where  $x_1$  will be substituted by the minimum value of  $x_1$  to calculate  $x_2$  minimum. The same applies to the maximum.

In lines 13-22 of the function `calculate_hypothesis` the previous formula is applied as can be seen in Fig. 7

```

1  import numpy as np
2
3  def plot_boundary(X, theta, ax1):
4
5      min_x1 = 0.0
6      max_x1 = 0.0
7      x2_on_min_x1 = 0.0
8      x2_on_max_x1 = 0.0
9
10     #####
11     # Write your code here
12     # Re-arrange the terms in the equation of the hypothesis function.
13     X1= X[:,1]
14
15
16     min_x1 = min(X1)
17     max_x1 = max(X1)
18
19
20     x2_on_min_x1 = -(min_x1*theta[1]+theta[0])/theta[2]
21     x2_on_max_x1 = -(max_x1*theta[1]+theta[0])/theta[2]
22     #####
23

```

Fig. 7 Decision boundary calculation.

The obtained decision boundary is:

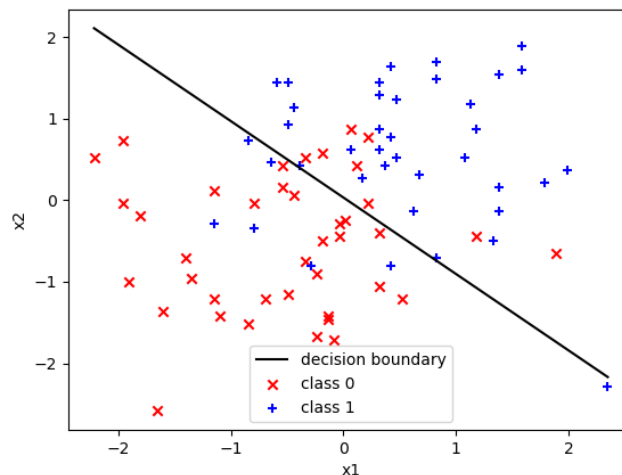


Fig. 8 Decision boundary plot.

**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 this section of the report, the data set is divided into two groups: training set (with 20 input values) and test set (with 60 data points). The main reason why the data is split is to make a proper



evaluation of our model. The training set is used to fit the model, while the test set will be used to check the performance of our model.

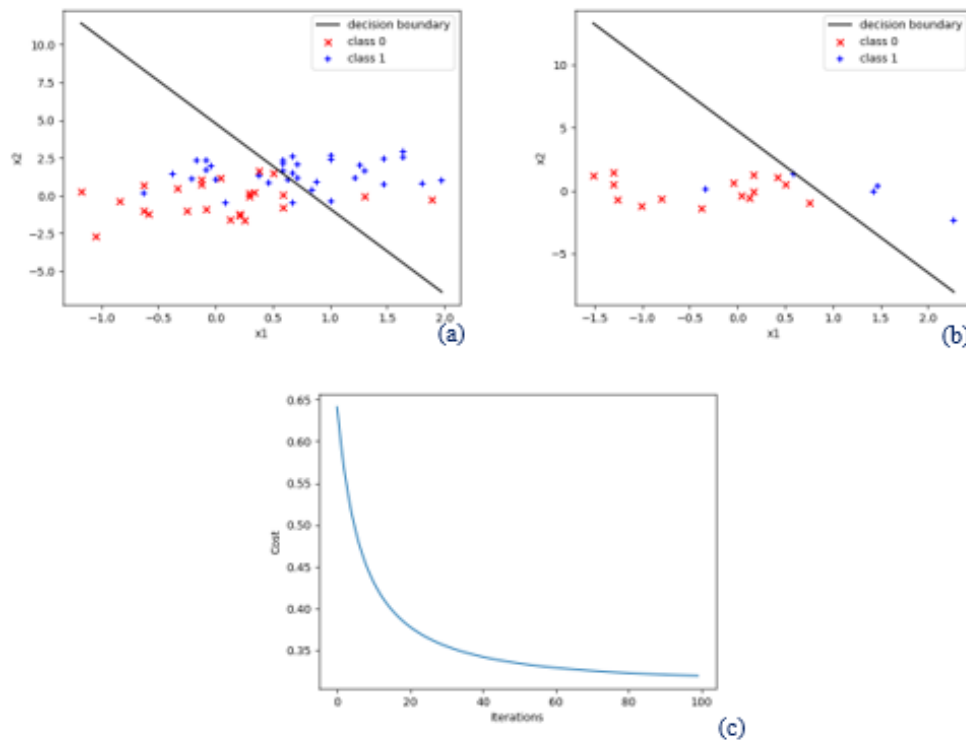
The goal of the training data is to fit a model that can generalize well all the data, this means that the new model should be able to make correct predictions on data it has not seen yet. However, two problems can rise if the data does not generalize well: over and underfitting. Both of these concepts will be explained in the following lines with some examples.

Several runs of the code have been made in this section. Some of the most significant runs are summarized below.

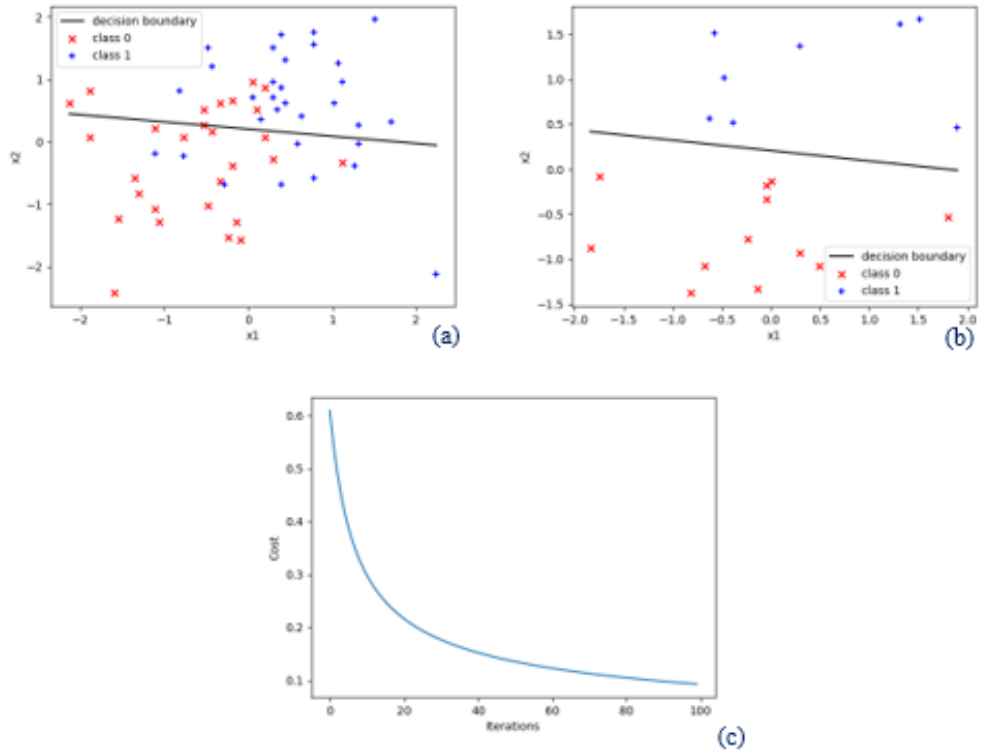
Fig. 9 is a good example of underfitting. It is noticeable that the test data is unbalanced, there are more data points for class 0 than for class 1. Consequently, the fitted model is biased towards class 0, which can be translated as a bad generalization and poor training fit.

In contrast, Fig 10 can be used as an example for overfitting, since we have a perfect train fit. However, this leads to a bad generalisation since random fluctuations in the training data are taken into account during the learning process, having, therefore, a negative impact on the fitted model.

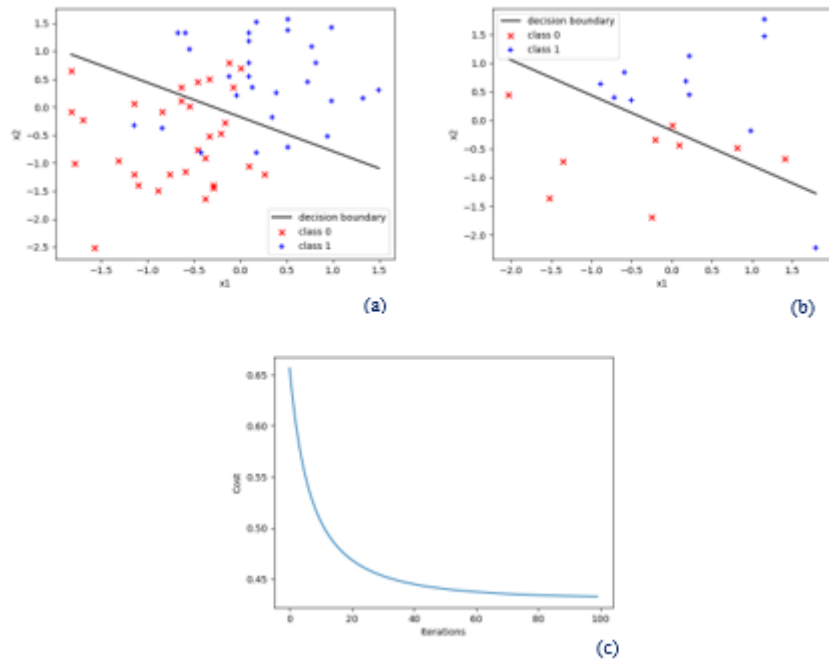
Finally, we can say that Fig. 11 is a good fit for the model. The training set is quite balanced, and the model generalizes well enough to make satisfactory estimations of the prediction data as can be seen in Fig. 11a.



*Fig. 9 Run #1: Test set (a), train set (b) and cost (c) plots.*



*Fig. 10 Run # 2: Test set (a), train set (b) and cost (c) plots.*



*Fig. 11 Run # 3: Test se (a), train set (b) and cost (c) plots.*

When it comes to cost, in the case of overfitting, the test cost is higher (0.5) than in the case of training cost (0.08). these values are reasonable since the error for the training set will be lower

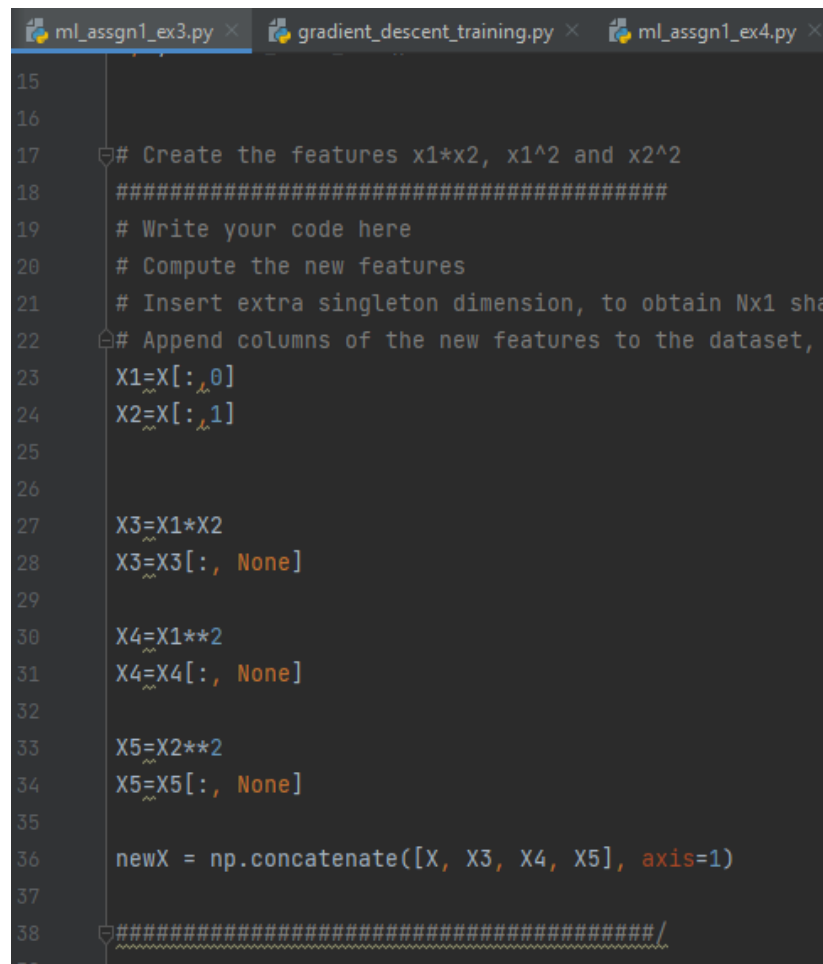
because the model fits the data almost perfectly, but a poor generalization is made resulting in higher errors in the test data.

In contrast, when the model underfits, it fails to generalize and fit the model bad enough in the training data, therefore the cost will be high in both cases (0.32 for the training set and 0.48 for the test set)

Moreover, when the model fits correctly the data, the values of the cost in both cases are similar since there is a balance and the data is well generalized.

In *assgn1\_ex3.py*, instead of using just the 2D feature vector, incorporate the following non-linear features:  $x_1, x_2, x_1^2, x_2^2$ . This results in a 5D input vector per data point, and so you must use 6 parameters  $\theta$ .

The corresponding linear features have been added to *ml\_assgn1\_ex3.py* from lines 23 to 38 as it can be seen in the following figure.

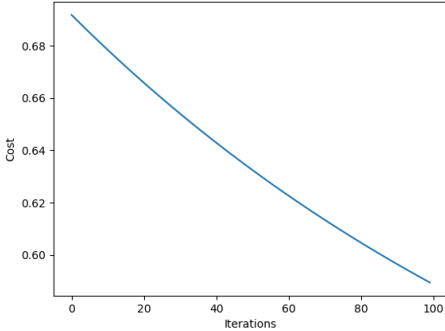
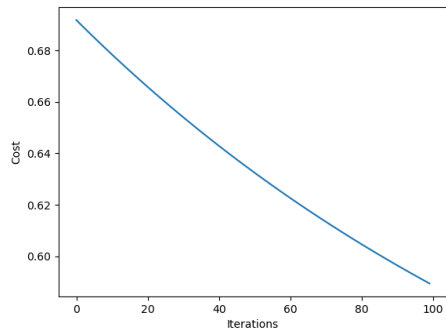
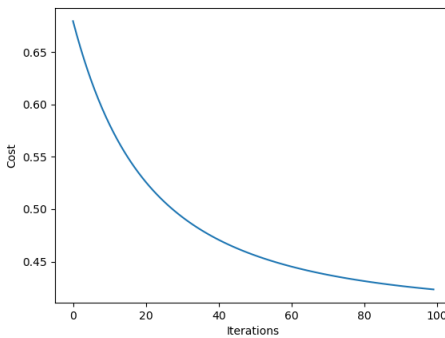
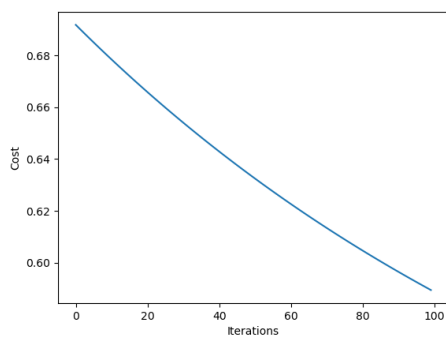
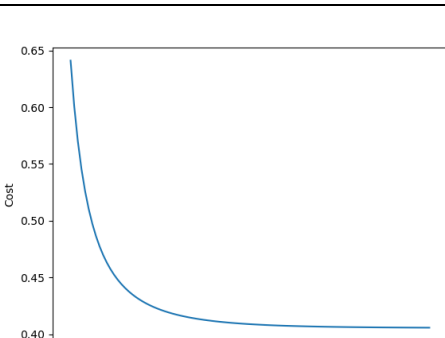
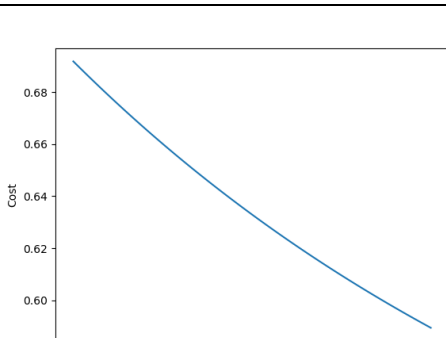


```
15
16
17 # Create the features x1*x2, x1^2 and x2^2
18 #####
19 # Write your code here
20 # Compute the new features
21 # Insert extra singleton dimension, to obtain Nx1 shape
22 # Append columns of the new features to the dataset,
23 X1=X[:,0]
24 X2=X[:,1]
25
26
27 X3=X1*X2
28 X3=X3[:, None]
29
30 X4=X1**2
31 X4=X4[:, None]
32
33 X5=X2**2
34 X5=X5[:, None]
35
36 newX = np.concatenate([X, X3, X4, X5], axis=1)
37
38 ##### /
```

Fig. 12 *ml\_assgn1\_ex3.py* with 5D input vector

**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 of what happens.

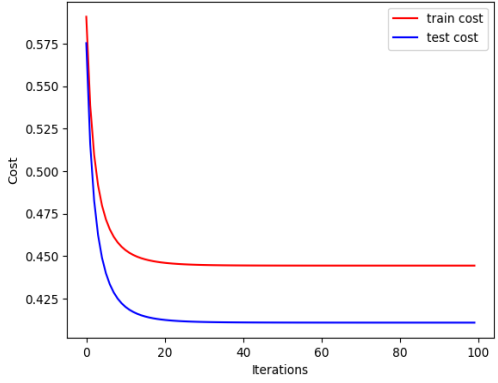
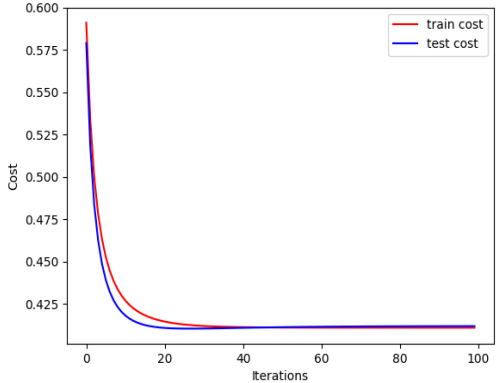
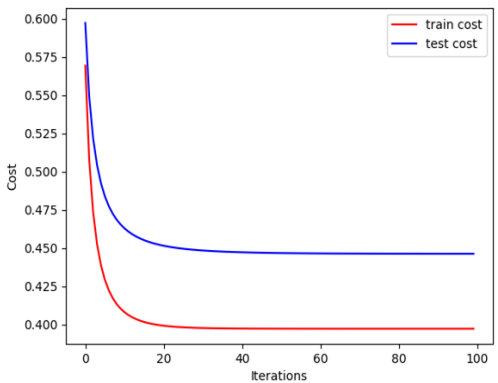
The following table is a comparison between the cost plots obtained with 2D and 5D data. As it can be seen, with a higher number of features the model fits better the data, therefore the value of the cost function will be smaller and consequently, it will converge slower to the optimal cost value (be aware that a large number of features may be a risk for overfitting)

$\alpha$	Cost 2D	Cost 5D
0.01		
0.1		
0.4		

*Table 2 Cost plot comparison between 2D and 5D input features.*

**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. 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?

The size of the training samples will have a direct effect on the performance of the model. Table 3 shows the cost of the training and test set for different training and test sizes.

Training data size	Cost of Training and Test set.
30	 <p>Line graph for training data size 30. The x-axis is 'Iterations' (0 to 100) and the y-axis is 'Cost' (0.425 to 0.575). The red line (train cost) starts at approximately 0.58 and decreases to about 0.44. The blue line (test cost) starts at approximately 0.58 and decreases to about 0.41.</p>
40	 <p>Line graph for training data size 40. The x-axis is 'Iterations' (0 to 100) and the y-axis is 'Cost' (0.425 to 0.600). The red line (train cost) starts at approximately 0.58 and decreases to about 0.42. The blue line (test cost) starts at approximately 0.58 and decreases to about 0.42.</p>
60	 <p>Line graph for training data size 60. The x-axis is 'Iterations' (0 to 100) and the y-axis is 'Cost' (0.400 to 0.600). The red line (train cost) starts at approximately 0.58 and decreases to about 0.40. The blue line (test cost) starts at approximately 0.58 and decreases to about 0.45.</p>

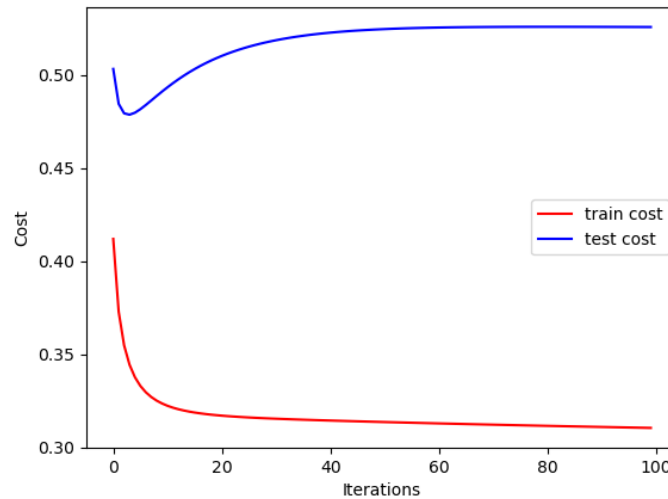
*Table 3 Cost of the training and test set for different sizes of data.*

When the training data size is 30, the model is underfitted, the sample size is too small and is not able of learning correctly.

Additionally, when the data size is 40, the value of both costs is almost the same. The training set generalizes correctly the data, hence the predictions made on the test data are appropriate.

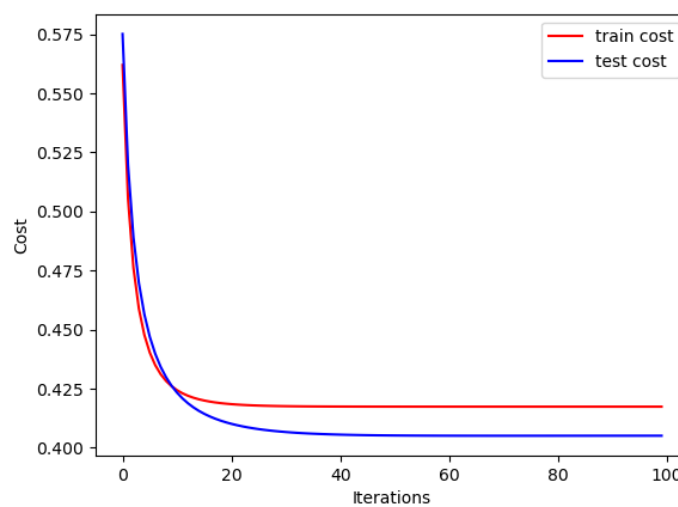
Finally, when the training set has a size of 60, it can be seen how the cost of the test data is higher than the training data. This means that the model is overfitted since it is taking into account random fluctuations while fitting the model.

On the other hand, we will analyse how adding or reducing the number of features affect the model. As it can be seen in Fig. 13, if we increase the polynomial degree to 6, the model overfits. However sometimes, when adding features, the model can fit even better than before. In these cases, the extra inputs helps to fit better the data and therefore the cost will fall, generating a generalized model that predicts better the new samples.



*Fig. 13 Cost of the training and test data for a six-degree polynomial.*

Moreover, decreasing the number of features may result in an underfitted model. As can be seen in Fig. 14, there is a poor generalization and therefore the predictions made by the model do not correspond to the ground truth values.



*Fig. 14 Cost of the training and test data for a third-degree polynomial*

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

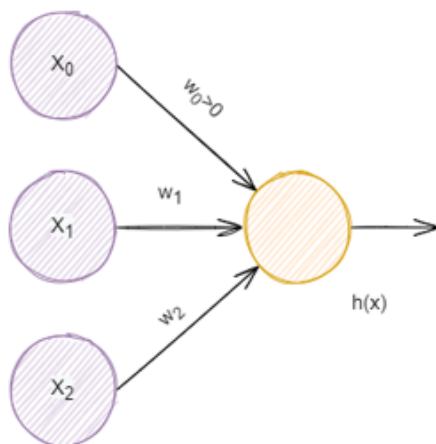
The truth table of the XOR logic function is given in the table given below:

$X_1$	$X_2$	Output
0	0	0
0	1	1
1	0	1
1	1	0

*Table 4 XOR truth table*

In this section, we will prove with the help of four diagrams why logistic regression does not solve the XOR classification problem.

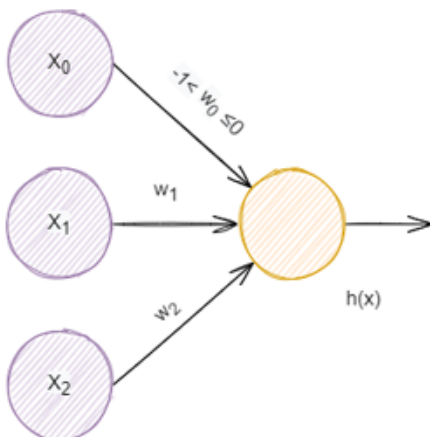
When the value of the weight of the bias is higher than 1, the output will always be 1. Therefore the data will always be categorized with the label 1



$X_1$	$X_2$	Output
0	0	1
0	1	1
1	0	1
1	1	1

*Fig. 15 Label 1 logistic unit*

Additionally, if the weight of the bias is  $-1 < w_0 \leq 0$ , the resulting logic gate is an OR as it is observable in the following figures.



$X_1$	$X_2$	Output
0	0	0
0	1	1
1	0	1
1	1	1

*Fig. 16 OR logistic unit.*

Moreover, if the weight is in the range of  $(-2, -1]$ , we will be obtaining an AND logistic unit as is proved below.

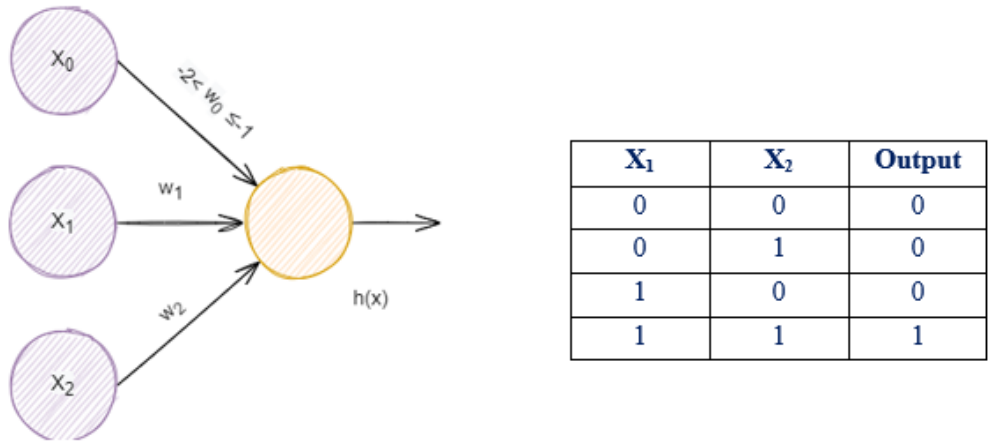


Fig. 17 AND logistic unit

Finally, if the weight of the bias is lower than -2, the output of the unit will always be 0, consequently, all the data will be categorized with label 0. Fig 18 proves it:

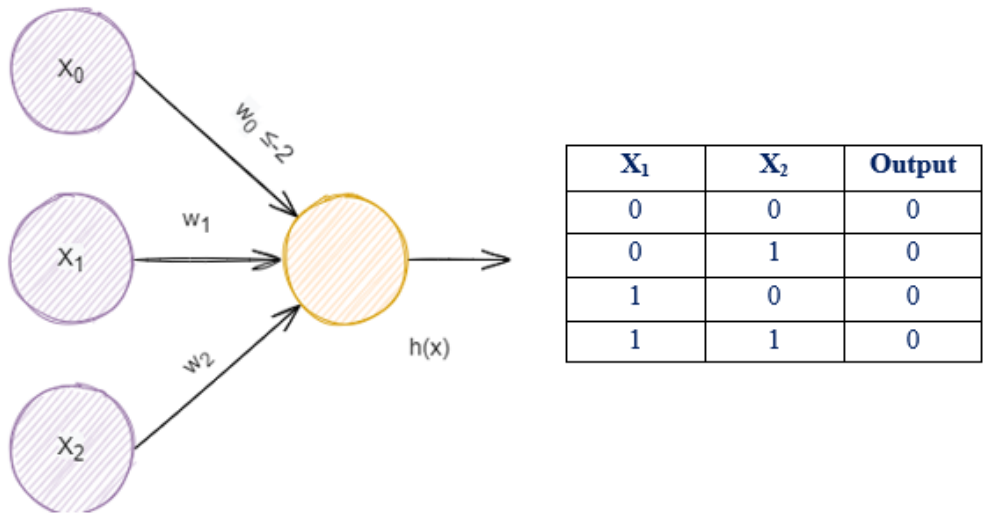


Fig. 18 Label 0 logistic unit

As it has been proved, in none of the possible cases, we can obtain an XOR with a logistic unit. We will need an additional hidden layer to obtain an XOR with logistic regression



## 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. 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?

The steps provided in the guide of the assignment have been followed. The bits of code are included in the following figure.

```
xorExample.py x irisExample.py x andExample.py x norExample.py x train_scripts.py x Ne
17 def backward_pass(self, inputs, targets, learning_rate):
18     # We will backpropagate the error and perform gradient descent on the network weights
19
20     # We compute the error between predictions and targets
21     J = 0.5 * np.sum(np.power(self.y_out - targets, 2))
22
23     # append term that was multiplied with the hidden layer's bias
24     inputs = np.append(1, inputs)
25
26     # Step 1. Output deltas are used to update the weights of the output layer
27     output_deltas = np.zeros((self.n_out))
28     outputs = self.y_out.copy()
29
30     if(self.n_out==1):
31         targets=np.array([targets])
32
33     for i in range(self.n_out):
34         #####
35         # Write your code here
36         # compute output deltas : delta_k = (y_k - t_k) * g'(x_k)
37         delta_k=(outputs[i]-targets[i])*sigmoid_derivative(outputs[i])
38         output_deltas[i]=delta_k
39     #####/
40
41     # Step 2. Hidden deltas are used to update the weights of the hidden layer
42     hidden_deltas = np.zeros((len(self.y_hidden)))
43
44     # Create a for loop, to iterate over the hidden neurons.
45     # Then, for each hidden neuron, create another for loop, to iterate over the output neurons
46     for i in range(len(hidden_deltas)):
47         #####
48         # Write your code here
49         # compute hidden deltas
50
51         #...
52         #...
53         #...
54         hidden_deltas[i] = ...
55         delta_j=0
56         for j in range(len(output_deltas)):
57             delta_j += output_deltas[j]*self.w_out[i,j]
58
59         hidden_deltas[i]= delta_j* sigmoid_derivative(self.y_hidden[i])
60
61     # Step 3. update the weights of the output layer
62     for i in range(len(self.y_hidden)):
63         for j in range(len(output_deltas)):
64             #####
65             # Write your code here
66             # update the weights of the output layer
67
68             # self.w_out[i,j] = ...
69             weight = self.w_out[i,j] - learning_rate*self.y_hidden[i]*output_deltas[j]
70             self.w_out[i,j] = weight
71             #####/
72
73     # we will remove the bias that was appended to the hidden neurons, as there is no
74     # connection to it from the hidden layer
75     # hence, we also have to keep only the corresponding deltas
76     hidden_deltas = hidden_deltas[1:]
77
78     # Step 4. update the weights of the hidden layer
79     # Create a for loop, to iterate over the inputs.
80     # Then, for each input, create another for loop, to iterate over the hidden deltas
81     for i in range(len(inputs)):
82         for j in range(len(hidden_deltas)):
83             #####
84             # Write your code here
85             # update the weights of the hidden layer
86             weight = self.w_hidden[i,j] - learning_rate*inputs[i]*hidden_deltas[j]
87             self.w_hidden[i,j] = weight
88             #####/
89
90     return J
```

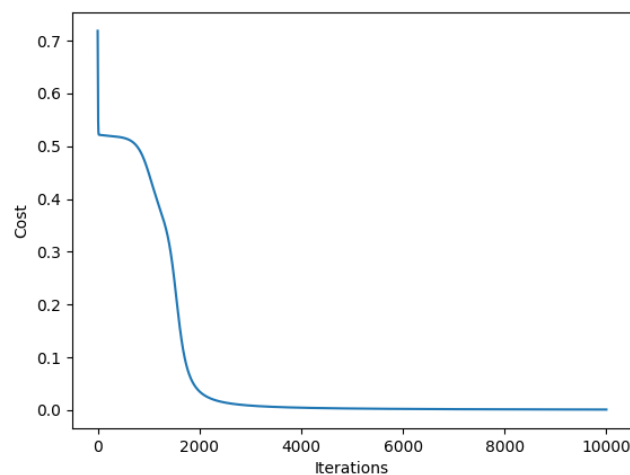
Fig. 19 Backpropagation function.

The best learning rate is found at  $\alpha = 0.3$ , as can be seen in the following figure, for this value of  $\alpha$ , the neural networks make a good prediction of the data concerning the target value.

```
Iteration 10000 | Cost = 0.00114
Sample #01 | Target value: 0.00 | Predicted value: 0.02614
Sample #02 | Target value: 1.00 | Predicted value: 0.97724
Sample #03 | Target value: 1.00 | Predicted value: 0.97720
Sample #04 | Target value: 0.00 | Predicted value: 0.02373
Minimum cost: 0.00114, on iteration #10000
```

*Fig. 20 Output of the neural network.*

Moreover, as can be seen in Fig. 21, the cost function decreases as the number of iterations increases. Since the predictions obtained are correct, it is likely that the cost has converged to global minima.



*Fig. 21 Cost function of the neural network with  $\alpha = 0.3$*

However, neural networks are not guaranteed to find the global minima since the cost is not convex. Iterative methods will be needed to train the data correctly. One example of getting stuck in a local minimum is the following, where all the outputs are correct except one:

```
Iteration 10000 | Cost = 0.51553
Sample #01 | Target value: 0.00 | Predicted value: 0.00257
Sample #02 | Target value: 1.00 | Predicted value: 0.85707
Sample #03 | Target value: 1.00 | Predicted value: 0.85698
Sample #04 | Target value: 0.00 | Predicted value: 0.85708
/homes/crr01/Desktop/ML/Assignment_1_Part_2/assgn_1_part_2/2_neu
plt.show()
Minimum cost: 0.44363, on iteration #9999
Process finished with exit code 0
```

*Fig. 22 Output when backpropagation is stuck in local minima.*

**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.

To implement an AND logic function, the target values must be changed to [0,0,0,1]. Once this is done, we obtain the following results:

```

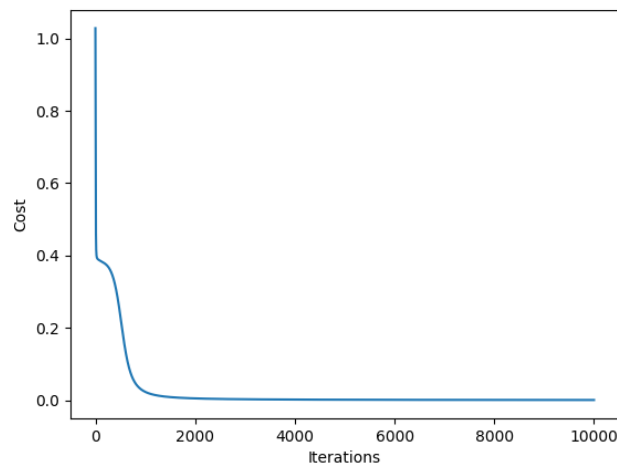
Sample #01 | Target value: 0.00 | Predicted value: 0.00283
Sample #02 | Target value: 0.00 | Predicted value: 0.01683
Sample #03 | Target value: 0.00 | Predicted value: 0.01682
Sample #04 | Target value: 1.00 | Predicted value: 0.98025

```

*Fig. 23 AND predicted values with the neural network.*

It can be seen; the neural networks succeed and train the data well enough to obtain the desired values.

The cost function for AND corresponds to Fig. 24, where it can be seen that the cost is approximately 0, which means that with the optimal values of theta, the error in the predictions will be quite low.



*Fig. 24 AND cost graph for neural networks.*

In the case of the NOR logic function, similar steps will be applied. This time the target values must be changed to [1,0,0,0]. The results obtained once the code is run are:

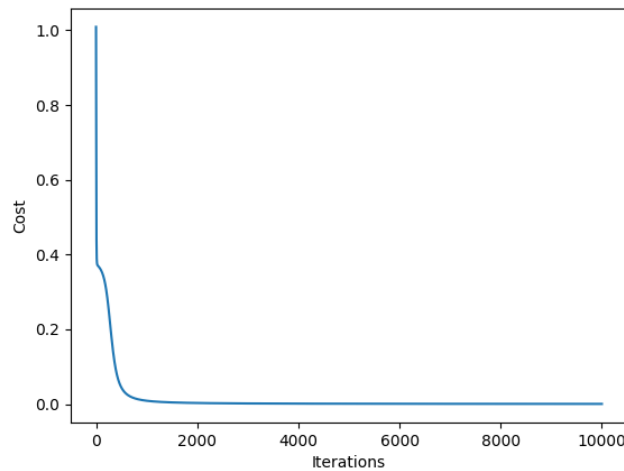
```

Sample #01 | Target value: 1.00 | Predicted value: 0.97529
Sample #02 | Target value: 0.00 | Predicted value: 0.01372
Sample #03 | Target value: 0.00 | Predicted value: 0.01389
Sample #04 | Target value: 0.00 | Predicted value: 0.00607

```

*Fig. 25 OR predicted values with the neural network.*

The prediction matches the target values, which implies that a neural network with 2 hidden layers can be used as a NOR logic function. Besides, the cost also decreases as time goes by, reaching an error value that is too low.



*Fig. 26 NOR cost plot for neural networks.*

**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?

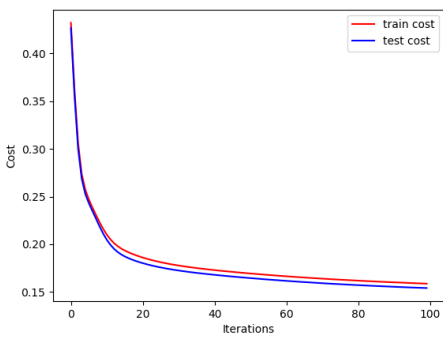
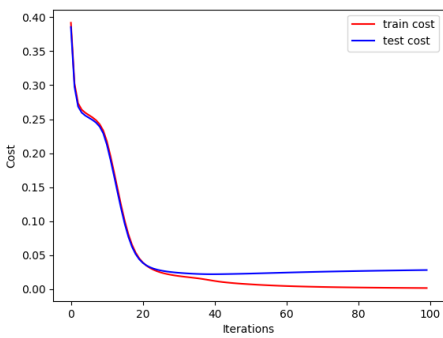
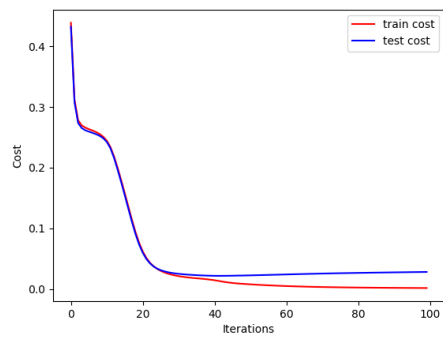
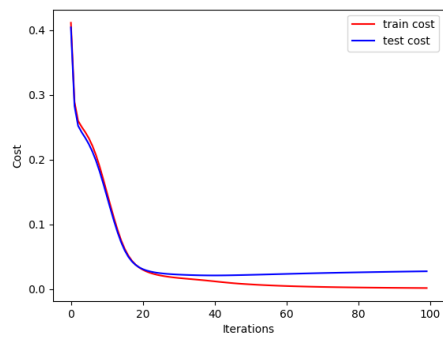
If we use a logistic regression unit, we can apply the 1-vs-All solution, which will use a logistic unit per class (3 in total), where the labels will indicate if the sample belongs or not to that particular class. Hence, a two-class problem is solved, where class  $n$  is separated from the others. Bear in mind, that this process is expensive and there are some ambiguous regions.

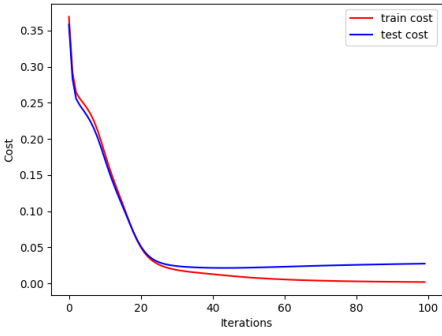
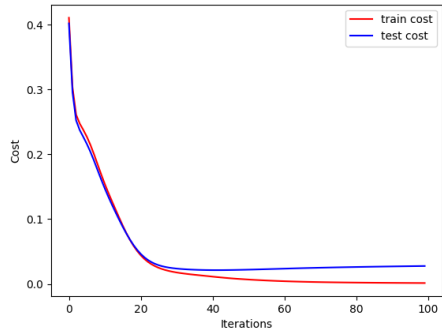
Another option is to apply SoftMax, given a sample, it will tell the probability of data to belong to a class. The category that obtains the highest values will be the predicted output of the model.

The main difference concerning the neural networks is that there is only one output for the logistic regression unit. Consequently, three logic units must be used in total. Nevertheless, neural networks can have multiple outputs for multiclass labels.

**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?

The resulting plots have been included in the following table. As it can be seen, 1 hidden neuron might be the best option, in the rest of the cases, the data seems to be overfitted. Probability during the training process with a higher number of neurons, the data is fitted perfectly, failing to generalize well and to make bad predictions during the new sample test.

N° hidden neurons	Training vs Test cost Plot.																								
1	 <p>This plot shows the training and test costs for a neural network with 1 hidden neuron over 100 iterations. The y-axis represents the cost, ranging from 0.15 to 0.40. The x-axis represents the number of iterations, ranging from 0 to 100. Both the training cost (red line) and the test cost (blue line) start at approximately 0.42 at iteration 0. They both decrease rapidly, with the training cost reaching about 0.18 and the test cost reaching about 0.19 by iteration 100. The two lines are very close to each other throughout the training process.</p> <table><tr><th>Iterations</th><th>train cost</th><th>test cost</th></tr><tr><td>0</td><td>0.42</td><td>0.42</td></tr><tr><td>10</td><td>0.20</td><td>0.20</td></tr><tr><td>20</td><td>0.18</td><td>0.19</td></tr><tr><td>40</td><td>0.17</td><td>0.18</td></tr><tr><td>60</td><td>0.16</td><td>0.17</td></tr><tr><td>80</td><td>0.16</td><td>0.17</td></tr><tr><td>100</td><td>0.16</td><td>0.17</td></tr></table>	Iterations	train cost	test cost	0	0.42	0.42	10	0.20	0.20	20	0.18	0.19	40	0.17	0.18	60	0.16	0.17	80	0.16	0.17	100	0.16	0.17
Iterations	train cost	test cost																							
0	0.42	0.42																							
10	0.20	0.20																							
20	0.18	0.19																							
40	0.17	0.18																							
60	0.16	0.17																							
80	0.16	0.17																							
100	0.16	0.17																							
2	 <p>This plot shows the training and test costs for a neural network with 2 hidden neurons over 100 iterations. The y-axis represents the cost, ranging from 0.00 to 0.40. The x-axis represents the number of iterations, ranging from 0 to 100. Both the training cost (red line) and the test cost (blue line) start at approximately 0.38 at iteration 0. They both decrease rapidly, with the training cost reaching about 0.01 and the test cost reaching about 0.03 by iteration 100. The test cost is slightly higher than the training cost after iteration 20.</p> <table><tr><th>Iterations</th><th>train cost</th><th>test cost</th></tr><tr><td>0</td><td>0.38</td><td>0.38</td></tr><tr><td>10</td><td>0.25</td><td>0.25</td></tr><tr><td>20</td><td>0.05</td><td>0.05</td></tr><tr><td>40</td><td>0.02</td><td>0.03</td></tr><tr><td>60</td><td>0.01</td><td>0.03</td></tr><tr><td>80</td><td>0.01</td><td>0.03</td></tr><tr><td>100</td><td>0.01</td><td>0.03</td></tr></table>	Iterations	train cost	test cost	0	0.38	0.38	10	0.25	0.25	20	0.05	0.05	40	0.02	0.03	60	0.01	0.03	80	0.01	0.03	100	0.01	0.03
Iterations	train cost	test cost																							
0	0.38	0.38																							
10	0.25	0.25																							
20	0.05	0.05																							
40	0.02	0.03																							
60	0.01	0.03																							
80	0.01	0.03																							
100	0.01	0.03																							
3	 <p>This plot shows the training and test costs for a neural network with 3 hidden neurons over 100 iterations. The y-axis represents the cost, ranging from 0.0 to 0.4. The x-axis represents the number of iterations, ranging from 0 to 100. Both the training cost (red line) and the test cost (blue line) start at approximately 0.42 at iteration 0. They both decrease rapidly, with the training cost reaching about 0.01 and the test cost reaching about 0.03 by iteration 100. The test cost is slightly higher than the training cost after iteration 20.</p> <table><tr><th>Iterations</th><th>train cost</th><th>test cost</th></tr><tr><td>0</td><td>0.42</td><td>0.42</td></tr><tr><td>10</td><td>0.25</td><td>0.25</td></tr><tr><td>20</td><td>0.05</td><td>0.05</td></tr><tr><td>40</td><td>0.02</td><td>0.03</td></tr><tr><td>60</td><td>0.01</td><td>0.03</td></tr><tr><td>80</td><td>0.01</td><td>0.03</td></tr><tr><td>100</td><td>0.01</td><td>0.03</td></tr></table>	Iterations	train cost	test cost	0	0.42	0.42	10	0.25	0.25	20	0.05	0.05	40	0.02	0.03	60	0.01	0.03	80	0.01	0.03	100	0.01	0.03
Iterations	train cost	test cost																							
0	0.42	0.42																							
10	0.25	0.25																							
20	0.05	0.05																							
40	0.02	0.03																							
60	0.01	0.03																							
80	0.01	0.03																							
100	0.01	0.03																							
5	 <p>This plot shows the training and test costs for a neural network with 5 hidden neurons over 100 iterations. The y-axis represents the cost, ranging from 0.0 to 0.4. The x-axis represents the number of iterations, ranging from 0 to 100. Both the training cost (red line) and the test cost (blue line) start at approximately 0.42 at iteration 0. They both decrease rapidly, with the training cost reaching about 0.01 and the test cost reaching about 0.03 by iteration 100. The test cost is slightly higher than the training cost after iteration 20.</p> <table><tr><th>Iterations</th><th>train cost</th><th>test cost</th></tr><tr><td>0</td><td>0.42</td><td>0.42</td></tr><tr><td>10</td><td>0.25</td><td>0.25</td></tr><tr><td>20</td><td>0.05</td><td>0.05</td></tr><tr><td>40</td><td>0.02</td><td>0.03</td></tr><tr><td>60</td><td>0.01</td><td>0.03</td></tr><tr><td>80</td><td>0.01</td><td>0.03</td></tr><tr><td>100</td><td>0.01</td><td>0.03</td></tr></table>	Iterations	train cost	test cost	0	0.42	0.42	10	0.25	0.25	20	0.05	0.05	40	0.02	0.03	60	0.01	0.03	80	0.01	0.03	100	0.01	0.03
Iterations	train cost	test cost																							
0	0.42	0.42																							
10	0.25	0.25																							
20	0.05	0.05																							
40	0.02	0.03																							
60	0.01	0.03																							
80	0.01	0.03																							
100	0.01	0.03																							

7	
10	

**Table 5** *irisExample.py* with a different number of hidden neurons.