Assignment 3 – Machine Learning and Big Data

Introduction

In this homework, I will be coding logistic regression.

Functions will have both unregularised and regularised implementations

With logistic regression, we can solve problems where linear regression fails.

- Because classification is better to find an answer which is a "value"
    - While logistic regression is better to find an answer which is a "class"
- And, if we try to divide the graph into 2 parts (by a threshold), model from linear regression fails
    - Because linear regression always looks to the continous values and tries to classify with continous values

How logistic regression works is like this:

We still have our model

- Differently from linear regression, our aim with model, is not to create "an accurate value"
    - Like predicting house prices… there is an actual price and our model generates an estimated price to the target price
- Our aim is to "classify. In other words, our goal is to create "an accurate seperation of classes"

Model (which generates continous values by itself) is being inputted to a sigmoid function (which creates a classification)

- Sigmoid function puts the values (which model generated) between 0 and 1

Only thing we do is to:

1. Calculating the cost of our model
2. Improving it with gradient descent algorithm

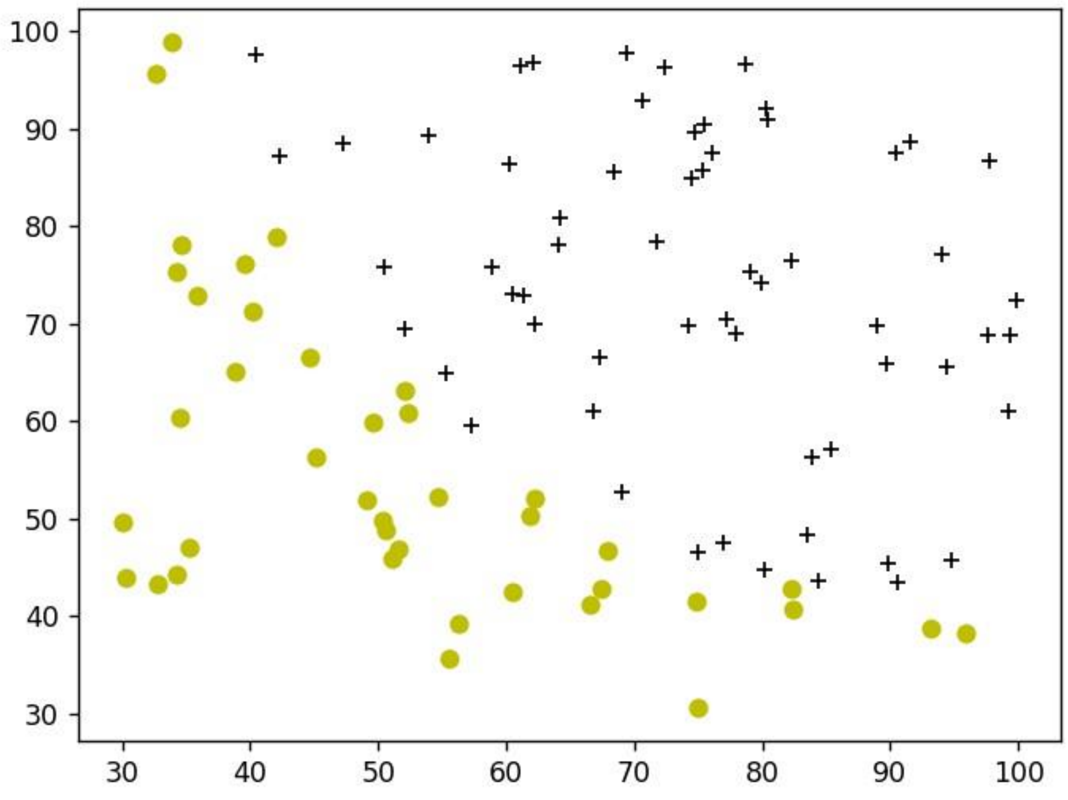
Part A: Logistic Regression

- It is for situations when we need to find a logistic regression model which is linear

Part B: Regularized Logistic Regression

- It is for situations when we need to find a logistic regression model which is not linear

Our initial data is like this:



+ represents “admitted students”

o (yellow circle) erpresent “not admitted” students

Part A

Sigmoid function

Sigmoid function is a function like below…

$$g(z) = \frac{1}{1 + e^{-z}}$$

The z input is the results vector which our model obtained

This simple function is just to put the values (which our model found) in an interval between 0 and 1

```python
def sigmoid(z):
    """
    Compute the sigmoid of z

    Args:
        z (ndarray): A scalar, numpy array of any size.

    Returns:
        g (ndarray): sigmoid(z), with the same shape as z

    """

    g = 1 / (1 + np.exp(-z))

    return g
```

Compute cost

For logistic regression, we use a formula like this below:

$$J(\mathbf{w},b) = \frac{1}{m} \sum_{i=0}^{m-1} \left[loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)}) \right]$$

where:

$$loss(f_{\mathbf{w},b}(\mathbf{x}^{(i)}), y^{(i)}) = (-y^{(i)} \log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right)$$

and $f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = g(\mathbf{w} \cdot \mathbf{x}^{(\mathbf{i})} + b)$ where function g is the sigmoid function.

The reason that this formula is different than the "linear function" is because this:

- If we use linear regression (to calculate the cost of our model) formula, we get a wiggly shape which is not convex
    - And since it is not convex, we can't reach to the global minima
        - since there are multiple minimas and we can't determine whether the minima we are at is a local minima or a global minima
            - Because, gradient descent function can only understand "whether it has reached to a minima/maxima or not"
            - And it will stop whenever it reaches a minima/maxima
- So, That is why we have a cost function like this
    - With this, we can get a convex shape for our cost function
- How cost function formula works is this:
    - We have our loss function
        - Our target class can be either 0 or 1
            - If our target is 0, left side of the subtraction becomes 0
            - If our target is 1, right side of the subtraction becomes 0
        - We calculate loss, with how far our estimation is from the target value
    - We just sum up all the loss values we got and then take their average

```python
##########################################################################
# logistic regression
#
def compute_cost(X, y, w, b, lambda_=None):
    """
    Computes the cost over all examples
    Args:
      X : (ndarray Shape (m,n)) data, m examples by n features
      y : (array_like Shape (m,)) target value
      w : (array_like Shape (n,)) Values of parameters of the model
      b : scalar Values of bias parameter of the model
      lambda_: unused placeholder
    Returns:
      total_cost: (scalar)         cost
    """

    estimation = np.dot(X, w) + b
    estimation = sigmoid(estimation)

    # print(estimation)


    loss = (np.dot(np.log(estimation), -y)) - np.dot(np.log(1-estimation), 1 - y)

    total_cost =  np.sum(loss) / len(y)


    return total_cost
```

Compute gradient

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})x_j^{(i)}$$
$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})$$

We take the derivative of our cost function, which would be later used at gradient descent to improve our model

```
def compute_gradient(X, y, w, b, lambda_=None):
    """
    Computes the gradient for logistic regression

    Args:
      X : (ndarray Shape (m,n)) variable such as house size
      y : (array_like Shape (m,1)) actual value
      w : (array_like Shape (n,1)) values of parameters of the model
      b : (scalar)                 value of parameter of the model
      lambda_: unused placeholder
    Returns
      dj_db: (scalar)                The gradient of the cost w.r.t. the parameter b.
      dj_dw: (array_like Shape (n,1)) The gradient of the cost w.r.t. the parameters w.
    """

    estimation = np.dot(X, w) + b
    estimation = sigmoid(estimation)

    dj_db = np.sum(estimation - y) / len(y)
    dj_dw = np.dot(estimation - y, X) / len(y)

    return dj_db, dj_dw
```

Gradient descent

Here, we update our model, by updating w and b

We take the derivative of our cost function

- Later, we will get closer to the minima
    - How fast and how steady we will get close is related with alpha

And then (by subtracting) we always get closer to the minima

repeat until convergence: {

$$w_j = w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \quad \text{for j = 0..n-1}$$

$$b = b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b}$$

}

```
def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function, alpha, num_iters, lambda_=None):
    """
    Performs batch gradient descent to learn theta. Updates theta by taking
    num_iters gradient steps with learning rate alpha

    Args:
      X :    (array_like Shape (m, n)
      y :    (array_like Shape (m,))
      w_in : (array_like Shape (n,))  Initial values of parameters of the model
      b_in : (scalar)                 Initial value of parameter of the model
      cost_function:                  function to compute cost
      alpha : (float)                 Learning rate
      num_iters : (int)               number of iterations to run gradient descent
      lambda_ (scalar, float)         regularization constant

    Returns:
      w : (array_like Shape (n,)) Updated values of parameters of the model after
          running gradient descent
      b : (scalar)                Updated value of parameter of the model after
          running gradient descent
      J_history : (ndarray): Shape (num_iters,) J at each iteration,
          primarily for graphing later
    """

    w = w_in
    b = b_in

    J_history = np.zeros(num_iters)

    for i in range(num_iters):
        dj_db, dj_dw = gradient_function(X=X, y=y, w=w, b=b, lambda_=lambda_)
        w = w - np.multiply(alpha, dj_dw)
        b = b - np.multiply(alpha, dj_db)

        J_history[i] = cost_function(X, y, w, b, lambda_)

    return w, b, J_history
```

Predict

This function is just for testing the accuracy of our model

- If target and result match, then for that instance, result is accurate
    - If both target and result are 0 or 1
- If they do not match, they are not accurate

```python
##########################################################################
# predict
#
def predict(X, w, b):
    """
    Predict whether the label is 0 or 1 using learned logistic
    regression parameters w and b

    Args:
    X : (ndarray Shape (m, n))
    w : (array_like Shape (n,))      Parameters of the model
    b : (scalar, float)              Parameter of the model

    Returns:
    p: (ndarray (m,1))
        The predictions for X using a threshold at 0.5
    """

    estimated_probabilities = sigmoid(np.dot(X, w) + b)

    p = (estimated_probabilities >= 0.5).astype(int)


    return p
```

Here is the result I got:

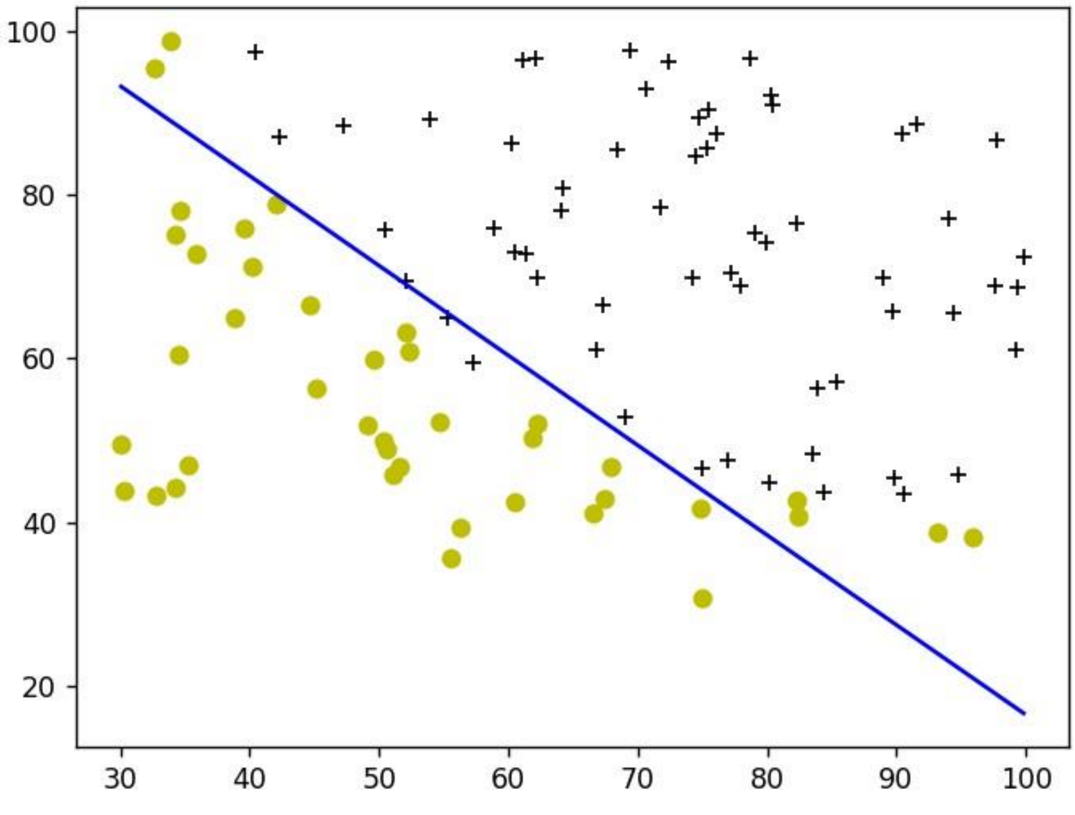
```python
X, y = load_data()

# Testing part A

w_in, b_in, J_history = gradient_descent(X=X,y=y,alpha=0.001,b_in=-8,w_in=np.zeros(2),cost_function=compute_cost_reg,gradient_function=compute_gradient_reg,num_iters=10000,lambda_=0)
print([J_history[-1]])
```

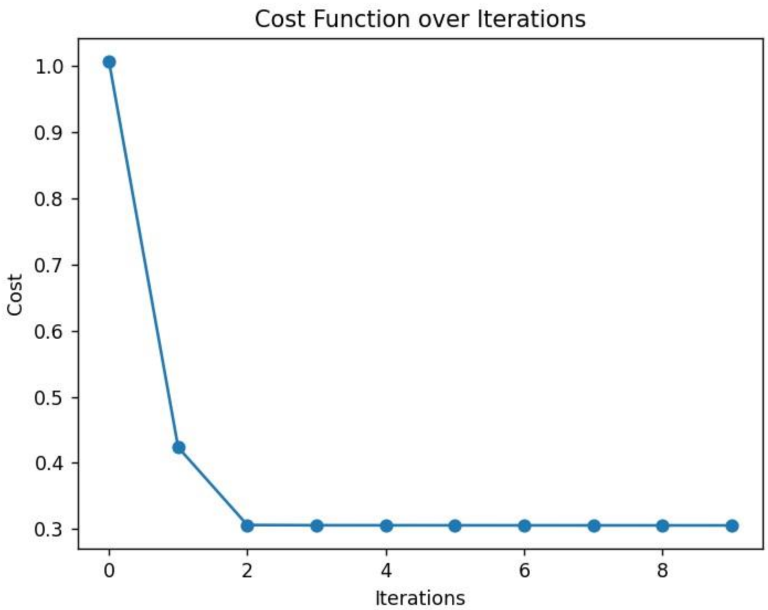
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\ardah\OneDrive\Masaüstü\p3> python -u "c:\Users\ardah\OneDrive\Masaüstü\p3\public_tests.py"
[0.30186822231814514]
PS C:\Users\ardah\OneDrive\Masaüstü\p3>
```

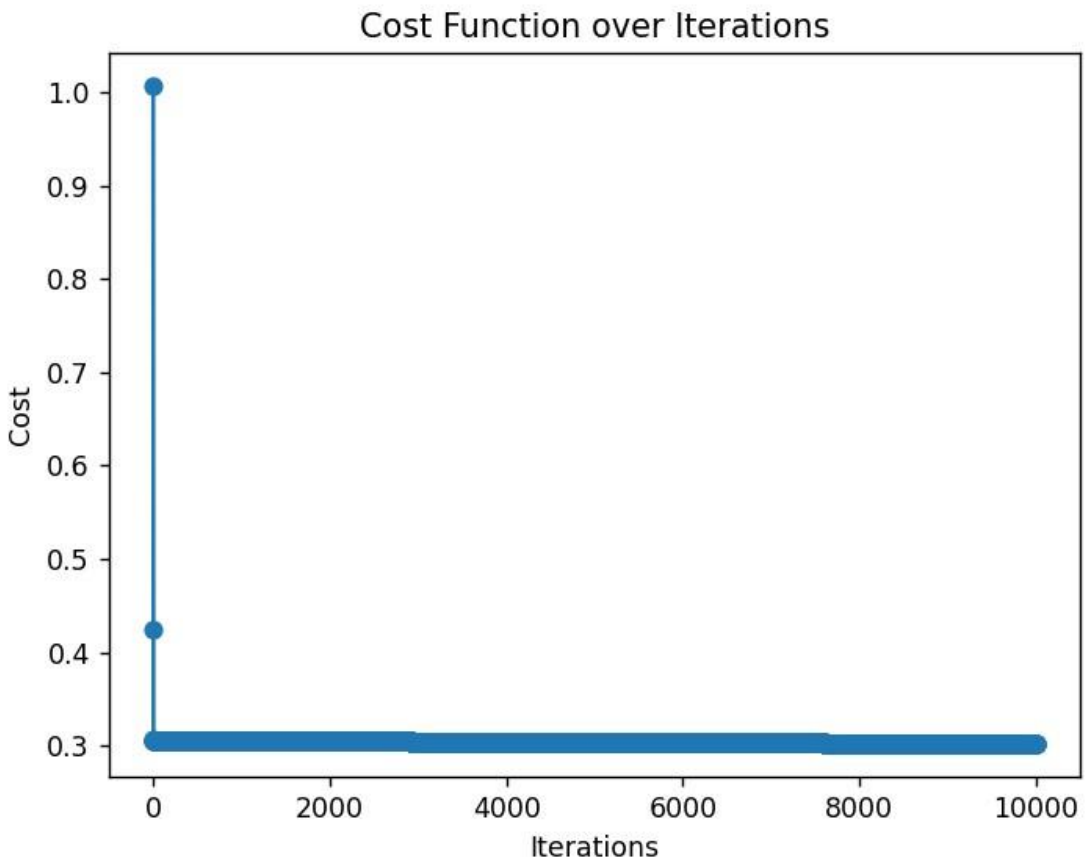


Here are my cost function graphs for part A:

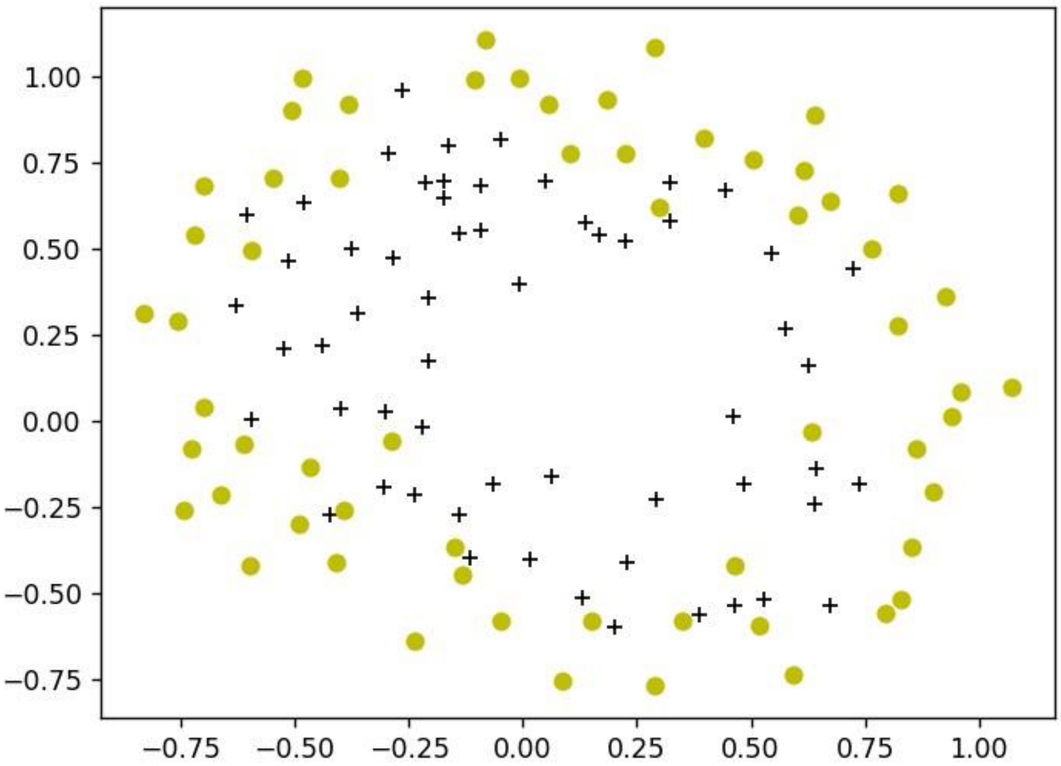
(both graphs correspond to the “cost function of model at part A”)



I had also put the cost function for 10 iterations, because this cost function was converging really quickly (in just 2 iterations haha)



Part B



+ => Accepted

o => Rejected

Feature mapping:

Because that we need to find a non-linear model, we need to do “feature mapping”

For being able to create an accurate function, we need more characteristics and each of these characteristics should have the best weight that they can have

To achieve this, we apply feature mapping

$$\text{map\_feature}(x) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \\ x_1^3 \\ \vdots \\ x_1 x_2^5 \\ x_2^6 \end{bmatrix}$$

For example, in Part B we have 2 characteristics.

But we try every possible combination (to find the best weight)

- So, we are adding more characteristics

```
# Testing part B
X = map_feature(X1=X.T[0], X2=X.T[1])
```

Compute cost for regularized logistic regression

$$J(\mathbf{w}, b) = \frac{1}{m} \sum_{i=0}^{m-1} \left[-y^{(i)} \log\left(f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - f_{\mathbf{w},b}\left(\mathbf{x}^{(i)}\right)\right) \right] + \frac{\lambda}{2m} \sum_{j=0}^{n-1} w_j^2$$

It is basically the same formula except now we use “regularization”

```python
def compute_cost_reg(X, y, w, b, lambda_=1):
    """
    Computes the cost over all examples
    Args:
      X : (array_like Shape (m,n)) data, m examples by n features
      y : (array_like Shape (m,)) target value
      w : (array_like Shape (n,)) Values of parameters of the model
      b : (array_like Shape (n,)) Values of bias parameter of the model
      lambda_ : (scalar, float)    Controls amount of regularization
    Returns:
      total_cost: (scalar)         cost
    """

    estimation = np.dot(X, w) + b
    estimation = sigmoid(estimation)

    loss = ( (np.dot(np.log(estimation), -y)) - np.dot(np.log(1-estimation), 1 - y) ) * (1 / len(y) )
    loss = np.sum(loss)
    lambda_part = np.sum((lambda_ / (2*len(y)) ) * np.square(w))

    total_cost =  loss + lambda_part

    return total_cost
```

Gradient for regularized logistic regression

Again, same thing except the only difference is just we also regularize…

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})$$

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \left(\frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} w_j \quad \text{for } j = 0...(n-1)$$

```
def compute_gradient_reg(X, y, w, b, lambda_=1):
    """
    Computes the gradient for linear regression

    Args:
      X : (ndarray Shape (m,n))   variable such as house size
      y : (ndarray Shape (m,))    actual value
      w : (ndarray Shape (n,))    values of parameters of the model
      b : (scalar)                value of parameter of the model
      lambda_ : (scalar,float)    regularization constant
    Returns
      dj_db: (scalar)             The gradient of the cost w.r.t. the parameter b.
      dj_dw: (ndarray Shape (n,)) The gradient of the cost w.r.t. the parameters w.

    """

    estimation = np.dot(X, w) + b
    estimation = sigmoid(estimation)

    # print(estimation)

    dj_db = np.sum(estimation - y) / len(y)
    dj_dw = np.dot(estimation - y, X) / len(y) + lambda_* (1 / len(y))*w

    return dj_db, dj_dw
```

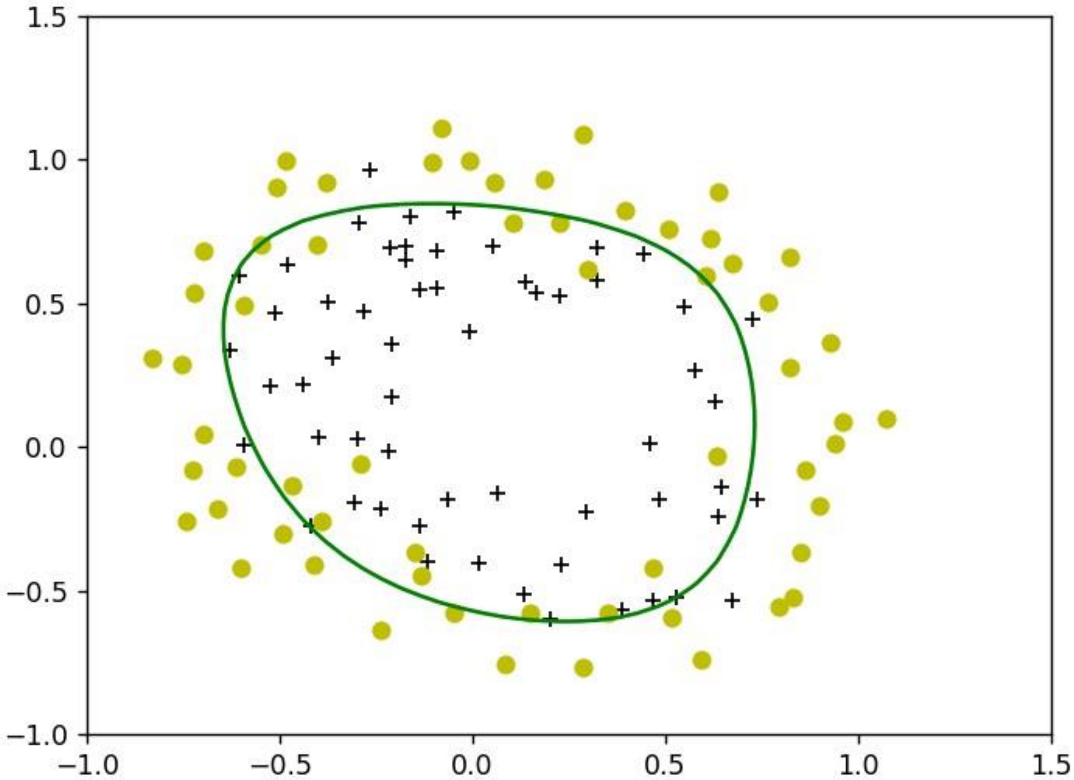
Here is the graph I got for part B:

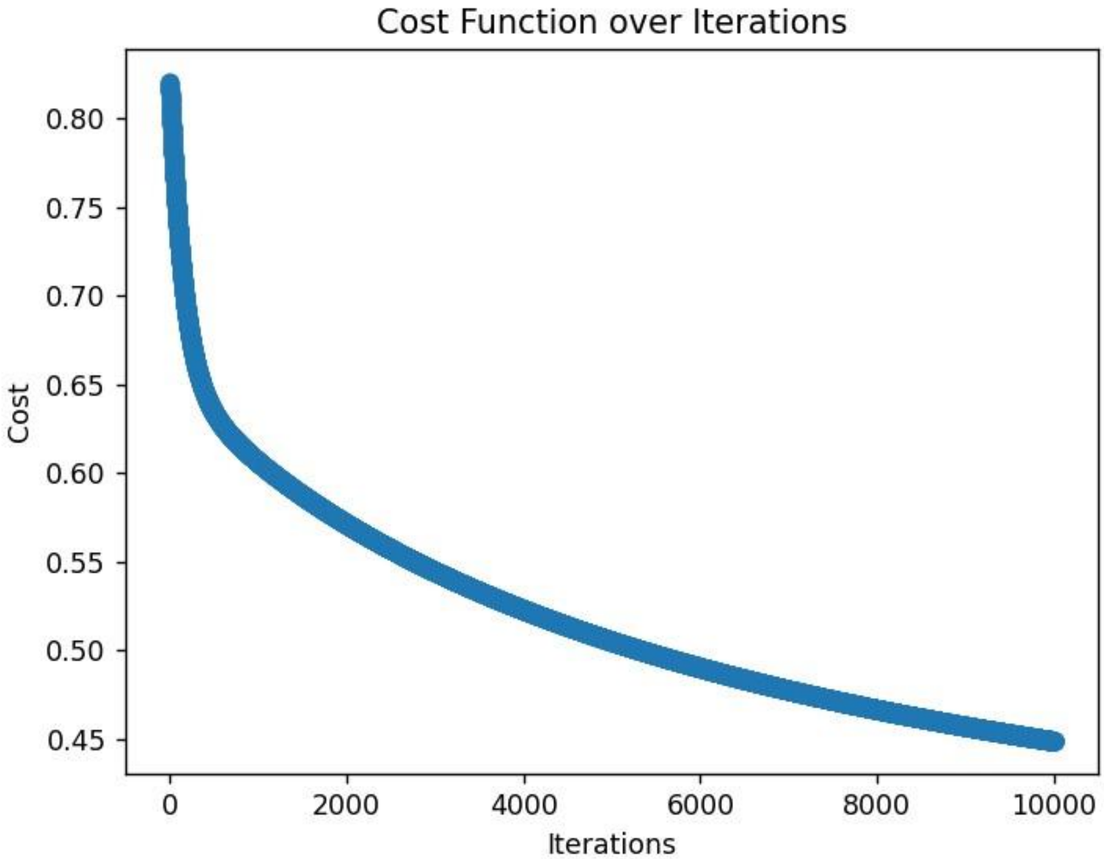
```
X = map_feature(X1=X.T[0], X2=X.T[1])

w, b, J_history = gradient_descent(X=X,y=y,alpha=0.01,b_in=1,w_in=np.zeros(27),cost_function=compute_cost_reg,gradient_function=compute_gradient_reg,num_iters=10000, lambda_=0.01)
print([J_history[-1]])
```

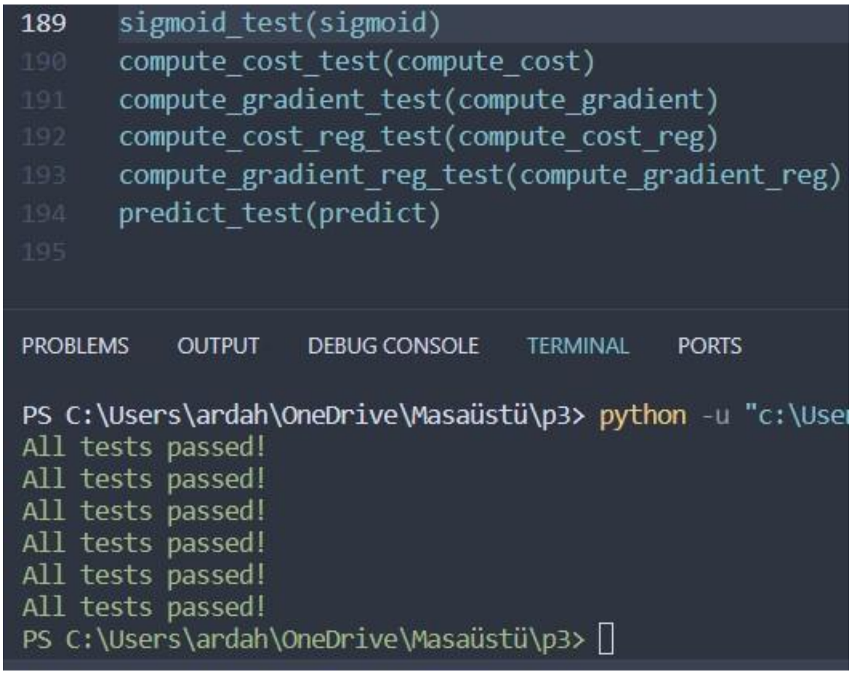
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Code

```
PS C:\Users\ardah\OneDrive\Masaüstü\p3> python -u "c:\Users\ardah\OneDrive\Masaüstü\p3\public_tests.py"
[0.44911111158137185]
```





Proof that all tests are passed:



```
sigmoid_test(sigmoid)
compute_cost_test(compute_cost)
compute_gradient_test(compute_gradient)
compute_cost_reg_test(compute_cost_reg)
compute_gradient_reg_test(compute_gradient_reg)
predict_test(predict)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\ardah\OneDrive\Masaüstü\p3> python -u "c:\User
All tests passed!
All tests passed!
All tests passed!
All tests passed!
All tests passed!
All tests passed!
PS C:\Users\ardah\OneDrive\Masaüstü\p3>
```