NAME: ABDULKADIR DURAN ADAN

STUDENT ID: 5035190144

CLASS: BCS1901

MAJOR: COMPUTER SCIENCE AND TECHNOLOGY

PROGRAMMING EXERCISE 2: LOGISTIC REGRESSION

MACHINE LEARNING

1. LOGISTIC REGRESSION

In this the exercise, I will build a logistic regression model to predict whether a student gets admitted into a university. Let's say that you are the administrator of a university department and you want to determine each applicant's chance of admission based on their results on two exams so how will you solve this. In this case you can use historical data from previous applicants that as a training set for logistic regression. For each training example, you have the applicant's scores on two exams and the admissions decision. Your task is to build a classification model that estimates an applicant's probability of admission based the scores from those two exams. Here we will build the modal with help of python libraries (numpy, pandas, seaborn and matplotlib).

IMPORTING THE LIBRARIES ON OUR JUPYTER NOTEBOOK

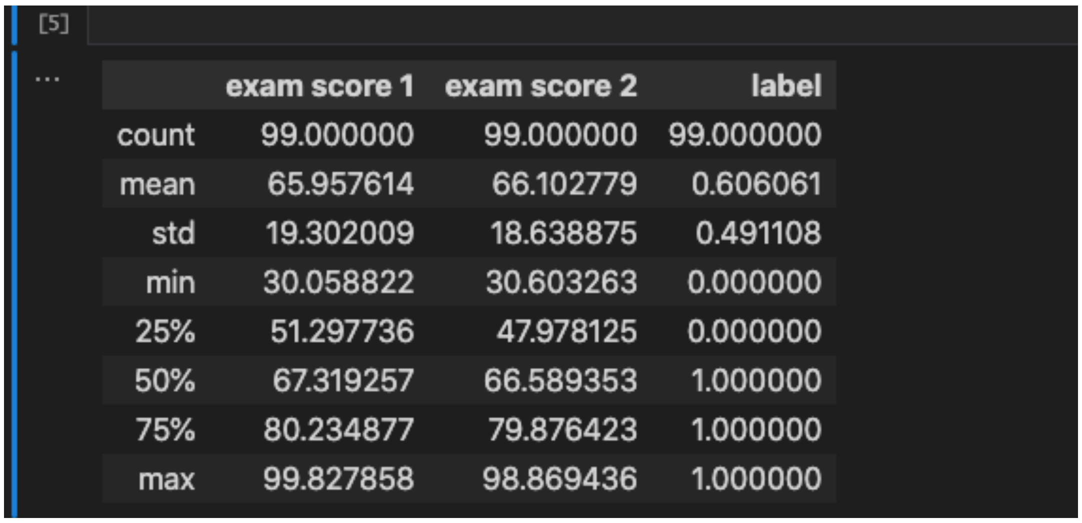
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

```
%matplotlib inline
import scipy.optimize as opt
```

RED AND DISPLAY DATA FROM YHE EXTERNAL FILE

```
df = pd.read_csv('/Users/boom/Desktop/homework4-
LogisticRegression/ex2data1.txt', sep=',')
df.columns = ['exam score 1', 'exam score 2', 'label']
df.describe()
```

OUTPUT



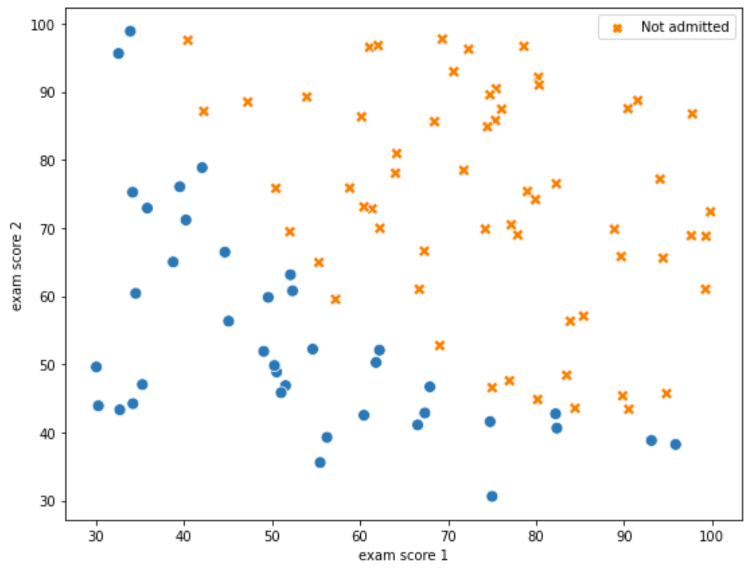
```
[5]
...
       exam score 1  exam score 2      label
count     99.000000     99.000000  99.000000
mean      65.957614     66.102779   0.606061
std       19.302009     18.638875   0.491108
min       30.058822     30.603263   0.000000
25%       51.297736     47.978125   0.000000
50%       67.319257     66.589353   1.000000
75%       80.234877     79.876423   1.000000
max       99.827858     98.869436   1.000000
```

VISUALIZING THE DATA

Before starting to implement any learning algorithm, it is always good to visualize the data if possible. In the rst part of ex2.m, the code will load the data and display it on a 2-dimensional plot by calling the function plotData. You will now complete the code in plotData so that it displays a gure like Figure 1, where the axes are the two exam scores.

```
plt.figure(figsize=(9,7))
ax = sns.scatterplot(x='exam score 1', y='exam score 2',
hue='label', data=df, style='label', s=80)
handles, labels = ax.get_legend_handles_labels()
ax.legend(handles[1:], ['Not admitted', 'Admitted'])

plt.show(ax)
```

OUTPUT



IMPLEMENTATION

SIGMOID FUNCTION

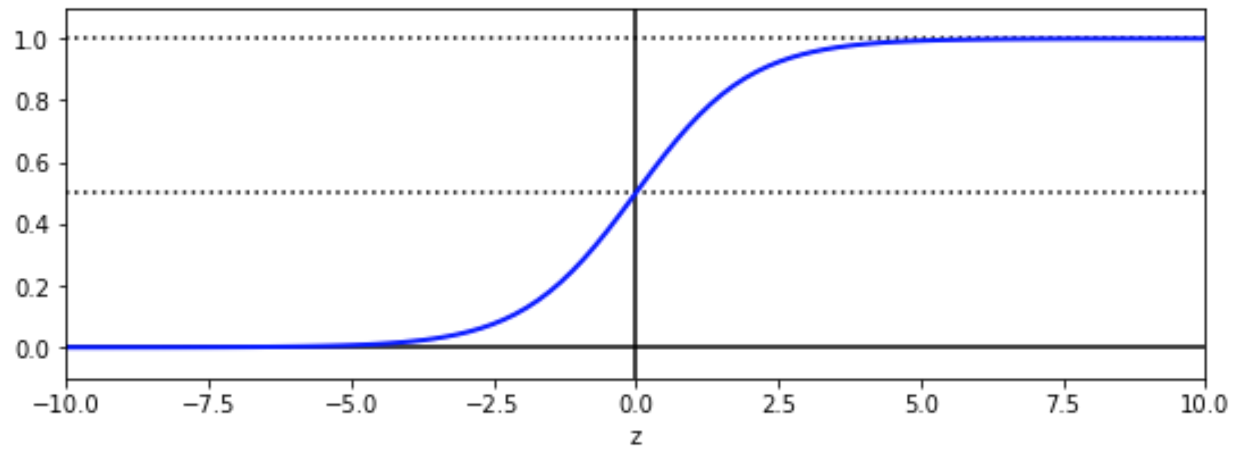
Logistic regression hypothesis:

$$h_\theta(x) = g(\theta^T x)$$

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

```
def sigmoid(z):
    z = np.array(z)
    return 1 / (1+np.exp(-z))

z = np.linspace(-10, 10, 100)
sig = sigmoid(z)
plt.figure(figsize=(9, 3))
plt.plot([-10, 10], [0, 0], "k-")
plt.plot([-10, 10], [0.5, 0.5], "k:")
plt.plot([-10, 10], [1, 1], "k:")
plt.plot([0, 0], [-1.1, 1.1], "k-")
plt.plot(z, sig, "b-", linewidth=2)
plt.xlabel("z")
plt.axis([-10, 10, -0.1, 1.1])
plt.show()
```

OUTPUT:



COST FUNCTION

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

Vectorized implementation

$$h = g(X\theta)$$

$$J(\theta) = \tfrac{1}{m}(-y^T log(h) - (1-y)^T log(1-h))$$

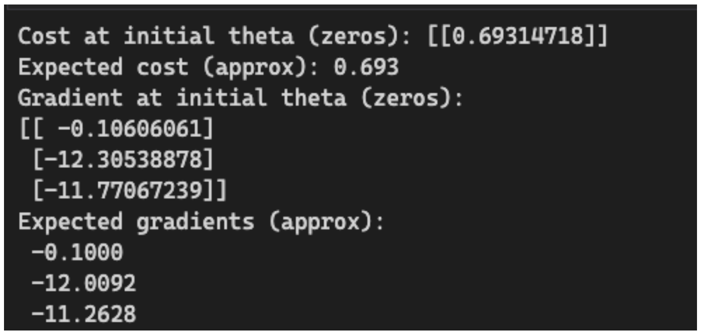
The gradient of the cost is a vector of the same length as θ where jth element (for $j=0,1,\ldots,n$) is defined as follows:

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

Vectorized: $\nabla J(\theta) = \frac{1}{m} \cdot X^T \cdot (g(X\theta) - y)$

```
def cost_function(theta, X, y):
    m = y.shape[0]
    theta = theta[:, np.newaxis]
    h = sigmoid(X.dot(theta))
    J = (1/m) * (-y.T.dot(np.log(h)) - (1-y).T.dot(np.log(1-h)))

    diff_hy = h - y
    grad = (1/m) * diff_hy.T.dot(X)

    return J, grad
m = df.shape[0]
X = np.hstack((np.ones((m,1)),df[['exam score 1', 'exam score
2']].values))
y = np.array(df.label.values).reshape(-1,1)
initial_theta = np.zeros(shape=(X.shape[1]))
cost, grad = cost_function(initial_theta, X, y)
print('Cost at initial theta (zeros):', cost)
print('Expected cost (approx): 0.693')
print('Gradient at initial theta (zeros):')
print(grad.T)
print('Expected gradients (approx):\n -0.1000\n -12.0092\n -
11.2628')
```

OUTPUT



```
Cost at initial theta (zeros): [[0.69314718]]
Expected cost (approx): 0.693
Gradient at initial theta (zeros):
[[ -0.10606061]
 [-12.30538878]
 [-11.77067239]]
Expected gradients (approx):
 -0.1000
 -12.0092
 -11.2628
```

LEARNING PARAMETERS USING AN OPTIMIZATION SOLVER

"Conjugate gradient", "BFGS", and "L-BFGS" are more sophisticated, faster ways to optimize θ that can be used instead of gradient descent.

```
def optimize_theta(X, y, initial_theta):
    opt_results = opt.minimize(cost_function, initial_theta,
args=(X, y), method='TNC',
                                jac=True, options={'maxiter':400})
    return opt_results['x'], opt_results['fun']
opt_theta, cost = optimize_theta(X, y, initial_theta)
print('Cost at theta found by fminunc:', cost)
print('Expected cost (approx): 0.203')
print('theta:\n', opt_theta.reshape(-1,1))
print('Expected theta (approx):')
print(' -25.161\n 0.206\n 0.201')
```

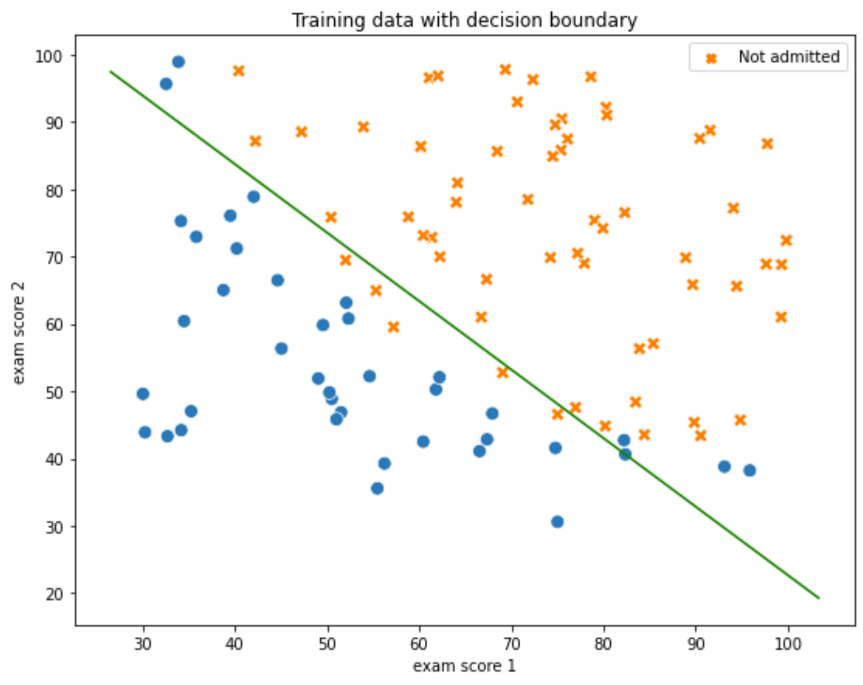
OUTPOUT

```
Cost at theta found by fminunc: [[0.2045574]]
Expected cost (approx): 0.203
theta:
 [[-24.86556581]
 [  0.20334347]
 [  0.19985042]]
Expected theta (approx):
 -25.161
 0.206
 0.201
```

DECISION BOUNDARY

```
plt.figure(figsize=(9,7))
ax = sns.scatterplot(x='exam score 1', y='exam score 2',
hue='label', data=df, style='label', s=80)
handles, labels = ax.get_legend_handles_labels()
ax.legend(handles[1:], ['Not admitted', 'Admitted'])
plt.title('Training data with decision boundary')

plot_x = np.array(ax.get_xlim())
plot_y = (-1/opt_theta[2]*(opt_theta[1]*plot_x + opt_theta[0]))
plt.plot(plot_x, plot_y, '-', c="green")
plt.show(ax)
```

OUTPUT

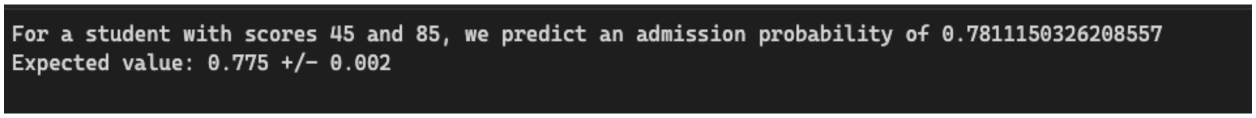


EVALUATING LOGISTIC REGRESSION

After learning the parameters, you can use the model to predict whether a particular student will be admitted. For a student with an Exam 1 score of 45 and an exam 2 score, should expect to see an admission probability of 0.776.

```
prob = sigmoid(np.array([1, 45, 85]).dot(opt_theta))
print('For a student with scores 45 and 85, we predict an
admission probability of', prob)
print('Expected value: 0.775 +/- 0.002');
def predict(X, theta):
    y_pred = [1 if sigmoid(X[i, :].dot(theta)) >= 0.5 else 0 for i
in range(0, X.shape[0])]
    return y_pred
X = np.hstack((np.ones((m,1)),df[['exam score 1', 'exam score
2']].values))

y_pred_prob = predict(X, opt_theta)
f'Train accuracy: {np.mean(y_pred_prob == df.label.values) * 100}'
```

OUTPUT



```
For a student with scores 45 and 85, we predict an admission probability of 0.7811150326208557
Expected value: 0.775 +/- 0.002
```

2. REGULARIZED LOGISTIC REGRESSION

In the second part of this exercise, we'll improve our logistic regression algorithm from part one by adding a regularization term. In a nutshell, regularization is a term in the cost function that causes the algorithm to prefer "simpler" models (in this case, models will smaller coefficients). The theory is that this helps to minimize overfitting and improve the model's ability to generalize. With that, let's get started.
Suppose you are the product manager of the factory and you have the test results for some microchips on two different tests. From these two tests, you would like to determine whether the microchips should be accepted or rejected. To help you make the decision, you have a dataset of test results on past microchips, from which you can build a logistic regression model.
let's start by visualizing the data.

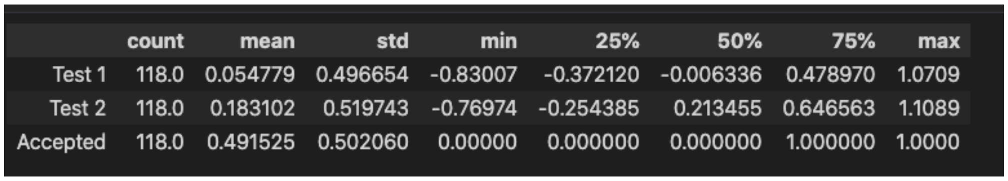
IMPORTING ALL DEPENCIES

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

READING AND DISPLAYING THE DATA FROM THE EXTERNAL FILE

```
data2 = pd.read_csv('/Users/boom/Desktop/homework4-
LogisticRegression/ex2data2.txt', sep=',', header=None,
names=['Test 1', 'Test 2', 'Accepted'] )


data2.describe().T
```

OUTPUT

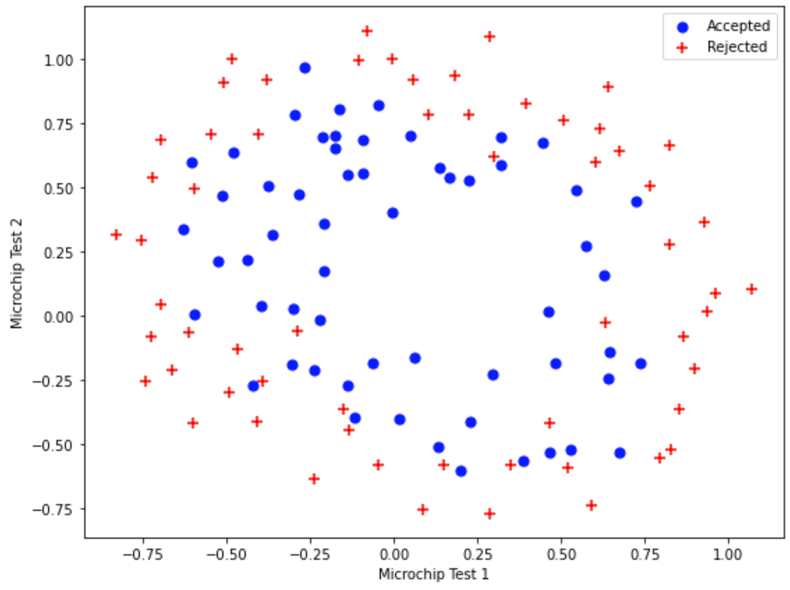


	count	mean	std	min	25%	50%	75%	max
Test 1	118.0	0.054779	0.496654	-0.83007	-0.372120	-0.006336	0.478970	1.0709
Test 2	118.0	0.183102	0.519743	-0.76974	-0.254385	0.213455	0.646563	1.1089
Accepted	118.0	0.491525	0.502060	0.00000	0.000000	0.000000	1.000000	1.0000

VISUALIZING THE DATA

```
positive = data2[data2['Accepted'].isin([1])]
negative = data2[data2['Accepted'].isin([0])]

fig, ax = plt.subplots(figsize=(9,7))
ax.scatter(positive['Test 1'], positive['Test 2'], s=50, c='b',
marker='o', label='Accepted')
ax.scatter(negative['Test 1'], negative['Test 2'], s=50, c='r',
marker='+', label='Rejected')
ax.legend()
ax.set_xlabel('Microchip Test 1')
ax.set_ylabel('Microchip Test 2')
```

OUTPUT

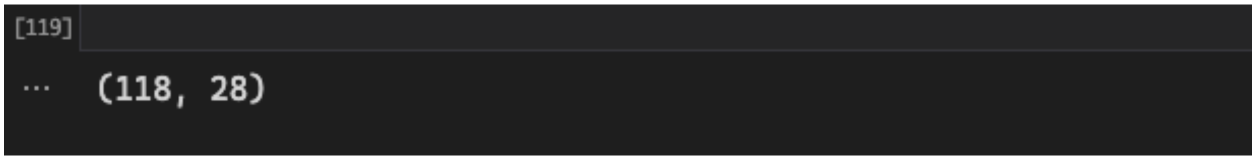


FEATURE MAPPING

One way to fit the data better is to create more features from each data point. We will map the features into all polynomial terms of x1 and x2 up to the sixth power. As a result of this mapping, our vector of two features has been transformed into a 28-dimensional vector. A logistic regression classifier trained on this higher-dimension feature vector will have a more complex decision boundary and will appear nonlinear when drawn in our 2-dimensional plot.

```
def map_feature(X1, X2, degree):
    X1 = np.array(X1).reshape(-1,1)
    X2 = np.array(X2).reshape(-1,1)

    out = np.ones((X1.shape[0], 1))
    for i in range(1, degree+1):
        for j in range(0, i+1):
            p = (X1**(i-j)) * (X2**j)
            out = np.append(out, p, axis=1)
    return out
X_p = map_feature(df2.test_1.values, df2.test_2.values, 6)
X_p.shape
```

OUTPUT



```
[119]
...   (118, 28)
```

COST FUNCTION AND GRADIENT

Regularized cost function in logistic regression:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^i log(h_\theta(x^i)) + (1-y^i)log(1-h_\theta(x^i))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Gradient:

$$\frac{\partial J(\theta)}{\partial\theta_0} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)} - y^{(i)}) \cdot x_j^{(i)} \text{ for j=0}$$

$$\frac{\partial J(\theta)}{\partial\theta_0} = (\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)} - y^{(i)}) \cdot x_j^{(i)}) + \frac{\lambda}{m}\theta_j \text{ for j}\geq 1$$

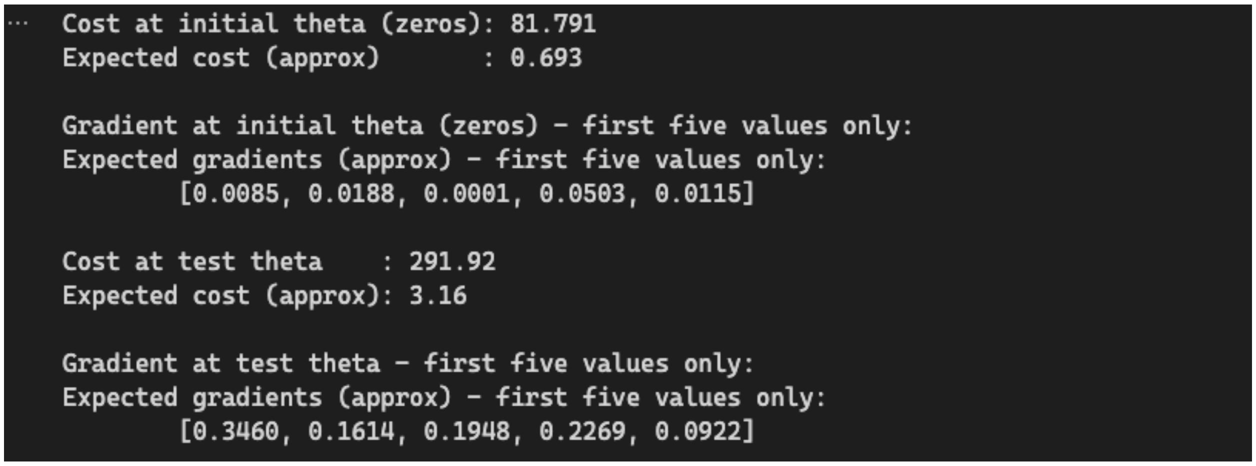
```
def costFunctionReg(theta, X, y, lambda_):

    m = y.size
    J = 0
    grad = np.zeros(theta.shape)

    J =  1/m * np.sum((-y * np.log(hyp)) + (-(1 - y) * np.log(1 - 
hyp))) + lambda_ / (2 * m) * np.sum(np.square(theta[1:]))

    grad = 1/m * np.dot(X.T, (hyp - y))

    return J, grad
```

LEARNING PARAMETERS USING FMINUNC

```
def costFunctionReg(theta, X, y, lambda_):


    m = y.size


    J = 0
    grad = np.zeros(theta.shape)

    hyp = sigmoid(np.dot(X, theta))

    J =  1/m * np.sum((-y * np.log(hyp)) + (-(1 - y) * np.log(1 - 
hyp))) + lambda_ / (2 * m) * np.sum(np.square(theta[1:]))


    grad = 1/m * np.dot(X.T, (hyp - y))

    return J, grad

initial_theta = np.zeros(X.shape[1])
```

```
lambda_ = 1

cost, grad = costFunctionReg(initial_theta, X, y, lambda_)

print('Cost at initial theta (zeros): {:.3f}'.format(cost))
print('Expected cost (approx)       : 0.693\n')
print('Gradient at initial theta (zeros) - first five values
only:')
print('Expected gradients (approx) - first five values only:')
print('\t[0.0085, 0.0188, 0.0001, 0.0503, 0.0115]\n')
test_theta = np.ones(X.shape[1])
cost, grad = costFunctionReg(test_theta, X, y, 10)
print('Cost at test theta    : {:.2f}'.format(cost))
print('Expected cost (approx): 3.16\n')

print('Gradient at test theta - first five values only:')

print('Expected gradients (approx) - first five values only:')
print('\t[0.3460, 0.1614, 0.1948, 0.2269, 0.0922]')
```

OUTPUT



```
... Cost at initial theta (zeros): 81.791
    Expected cost (approx)       : 0.693

    Gradient at initial theta (zeros) - first five values only:
    Expected gradients (approx) - first five values only:
            [0.0085, 0.0188, 0.0001, 0.0503, 0.0115]

    Cost at test theta    : 291.92
    Expected cost (approx): 3.16

    Gradient at test theta - first five values only:
    Expected gradients (approx) - first five values only:
            [0.3460, 0.1614, 0.1948, 0.2269, 0.0922]
```