Exercise 2 - Logistic Regression

Azka NA

October 26, 2020

1 Introduction

This is the guide for Andrew Ng's Machine Learning course programming assignment done in Python, adapted from the original guide written for Octave or MATLAB.

In this exercise, you will implement logistic regression and get to see it work on data. Before starting on this programming exercise, we strongly recommend watching the video lectures and completing the review questions for the associated topics

For Programming Exercise 2: Logistic Regression, you will need to download the following files:

`exercise2-2.ipynb` - Jupyter notebook containing the script for part 1
`exercise2-1.ipynb` - Jupyter notebook containing the script for part 2
`ex2data1.txt` - Dataset for part 1 Logistic Regression
`ex2data2.txt` - Dataset for part 2 Regularized Logistic Regression

---

2 Logistic Regression

In this part of the exercise, you will build a logistic regression model to predict whether a student gets admitted into a university.

Suppose 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. You have historical data from previous applicants that you can use 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. This guide and the framework code in `exercise2-1.ipynb` will guide you through the exercise.

2.1 Visualizing the data

Before starting to implement any learning algorithm, it is always good to visualize the data if possible. In the first part of `exercise2-1.ipynb`, the code will load the data and display it on a 2-dimensional plot.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
dataset = pd.read_csv('ex2data1.txt', names=['exam1', 'exam2', '
 admitted'])
dataset.head()
```

Next, we create a scatter plot of the data with the following code.

```
X = dataset.iloc[:,0:2]
y = dataset.iloc[:,2]
pos = y==1
neg = y==0
sns.set()
plt.scatter(X[pos]['exam1'], X[pos]['exam2'], marker='+')
plt.scatter(X[neg]['exam1'], X[neg]['exam2'])
plt.xlabel('Exam 1 score')
plt.ylabel('Exam 2 score')
plt.legend(['Admitted', 'Not admitted'])
plt.show()
```

The end result should look like Figure 1 where the axes are the two exam scores, and the positive and negative examples are shown with different markers.

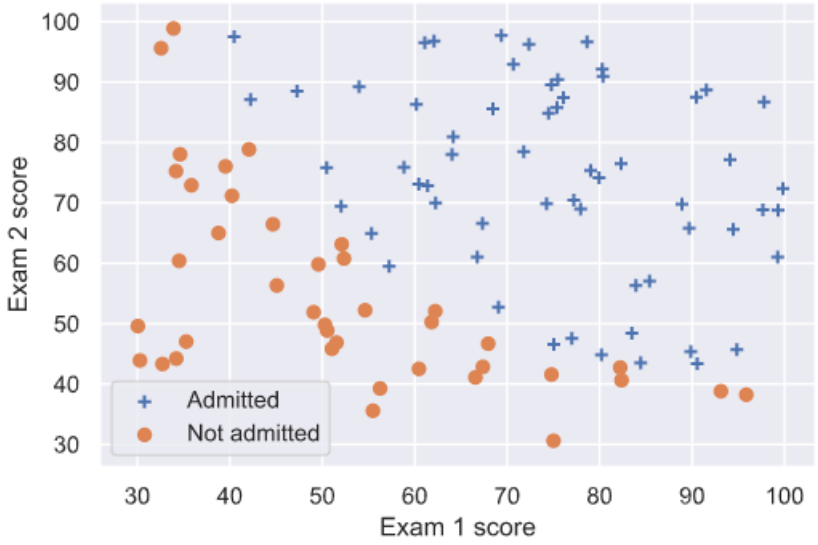

Figure 1: Scatter plot of training data

2.2 Implementation

2.2.1 Initializing parameters

Start by making the parameter array. Remember that the arrays should be rank-2 arrays (have a shape of (m,1)) rather than rank-1 arrays (have a shape of (m,)). For multiple variables problem, remember to scale the feature, that is, the X. We will scale the X and store them in X_scaled to be used for `gradientDescent`.

```
m, n = X.shape
ones = np.ones((m,1))

X_scaled = (X - np.mean(X))/np.std(X)

X = np.hstack((ones, X)) # (X, y)) will be used for optimization
 using scipy.optimize's minimize
theta = np.zeros((n+1,1))
y = y[:,np.newaxis]

# for optimization using gradientDescent, we should use scaled X
X_scaled = np.hstack((ones, X_scaled)) # (X_scaled, y) will be used
 for optimization gradientDescent
```

2.2.2 Sigmoid function

Before you start with the actual cost function, recall that the logistic regression hypothesis is defined as:

$$h_\theta(x) = g(\theta^T x), \tag{1}$$

where function g is the sigmoid function. The sigmoid function is defined as:

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

For large positive values of x, the sigmoid should be close to 1, while for large negative values, the sigmoid should be close to 0. Evaluating `sigmoid(0)` should give you exactly 0.5. Your code should also work with vectors and matrices.

```
def sigmoid(x):
  return 1/(1+np.exp(-x))
```

2.2.3 Cost function and gradient

Now you will implement the cost function and gradient for logistic regression. Recall that the cost function in logistic regression is

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}log(h_\theta(x^{(i)})) - (1-y^{(i)}log(1-h_\theta(x^{(i)})))\right] \tag{3}$$

which can be written as `costFunction`:

```
def costFunction(theta, X, y):
  m = len(y)
  z = np.dot(X, theta)
  h = sigmoid(z)
  term1 = -y*np.log(h)
  term2 = (1-y)*np.log(1-h)
  cost = np.sum(term1 - term2)/m
  return cost
```

The gradient of the cost is a vector of the same length as θ where the j^{th} element (for $j = 0, 1, ..., n$) is defined as follows:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \tag{4}$$

which can be written as `grad`:

```
def grad(theta, X, y):
m = len(y)
h = sigmoid(np.dot(X, theta))
grad = np.dot(X.T, h-y)/m
return grad
```

Note that while this gradient looks identical to the linear regression gradient, the formula is actually different because linear and logistic regression have different definitions of $h_\theta(x)$. Using the initial parameters of θ, you should see that the cost is about `0.693`.

There are two ways to optimize the learning parameters: using gradient descent and `minimize` from Scipy's Optimize module. The `costFunction` and `grad` function will be used in both technique.

2.3 Learning parameters

2.3.1 Learning parameters using `gradientDescent`

The code for `gradientDescent` is based on the following equation:

$$\theta_j := \theta_j - \alpha\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \tag{5}$$

The `gradientDescent` will return the value of optimized θ and the $J(\theta)$ after each iteration.

```
def gradientDescent(X, y, theta_initial, alpha, iterations):
''' check "2 Linear regression with one variable"
page 36
'''
m = len(y)
theta = theta_initial.copy()
J_val = []
for i in range(iterations):
    grad2 = grad(theta,X,y)
    cost = costFunction(theta,X,y)
    theta -= alpha*grad2
    J_val.append(cost)
return theta, J_val
```

To determine the α value, we can try several values and plot the computed cost after each iteration.

```
iterations = 400
alpha = [0.01, 0.1, 1]
theta_optimized1 = 0
for i in alpha:
    theta_optimized1, J_value =  gradientDescent(X_scaled, y, theta,
  alpha=i, iterations=iterations)
    plt.plot(np.arange(1,iterations+1), J_value, label='alpha = '+
 str(i))
print(f'Optimized theta from gradient descent: {theta_optimized1}')
```

From Figure 2, we will choose `alpha` of 1 where the cost (J_θ) is decreasing more rapidly. The optimized θ is: `1.65947664`, `3.8670477`, and `3.60347302`.

When we compute the cost using the optimized θ, we will get a lower value of `0.2036`.

2.3.2 Learning parameters using `minimize`

In section 2.3.1 you write a cost function and calculate its gradient, then took a gradient descent step accordingly. This time, instead of taking gradient descent steps, you will use a function in Python's `optimize`[1] module which is the equivalent of `fminunc` in Octave/MATLAB.

Octave/MATLAB's `fminunc` is an optimization solver that finds the minimum of an unconstrained[2] function. The `minimize`[3] from Scipy's library also can do unconstrained

[1]More details on Scipy optimize module here.

[2]Constraints in optimization often refer to constraints on the parameters, for example, constrains that bound the possible values θ can take (e.g., $\theta \leq 1$). Logistic regression does not have such constraints since θ is allowed to take any real value.

[3]More details on Scipy minimize here.

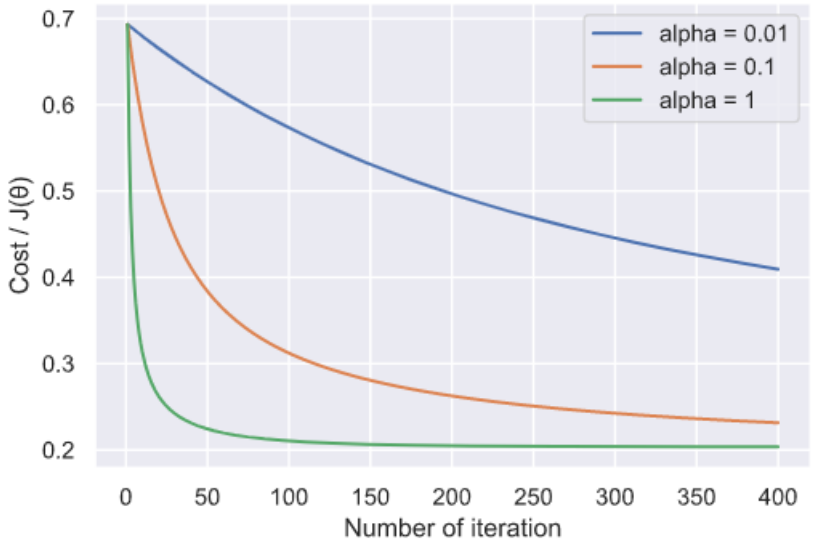

Figure 2: Cost for each iteration for different α

minimization of a function and here we will use the Newton-CG method. For logistic regression, you want to optimize the cost function $J(\theta)$ with parameters θ.

Using `minimize` we do not need to write any loops ourselves, or set the initial learning rate value like we do for `alpha` in `gradientDescent`. You are going to use `minimize` to find the best parameters θ for the logistic regression cost function, given a fixed dataset (of X and y values). You will pass to `minimize` the following inputs:

1. The method we are using, `Newton-CG`
2. The initial values of the parameters we are trying to optimize.
3. Objective funtion to be minimized
4. Gradient of the function that we minimize
5. Extra arguments passed to two functions in point 3 and 4

```
from scipy.optimize import minimize
minimizef = minimize(fun=costFunction, x0=theta.flatten(), jac=grad,
  args=(X, y.flatten()), method='Newton-CG')
theta_optimized2 = minimizef.x
```

The optimized θ is: `-25.16155593`, `0.20623349`, and `0.2014734`.

2.3.3 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 of 85, you should expect to see an admission probability of `0.776`.

```
examsval = [[45, 85]]
m = len(examsval)
ones = np.ones((m,1))
x = np.array(examsval)
x = np.hstack((ones, x)).reshape(m,3)
pred = sigmoid(np.dot(x, theta_optimized2))
print(pred)
```

Another way to evaluate the quality of the parameters we have found is to see how well learned model predicts on our training set. In logistic regression, the threshold is at 0.5, thus when we put the value from training dataset and the optimized θ to equation 1 and 2, all the values more than or equal to 0.5 will be classified as 'Admitted', or $y = 1$.

```
pred1 = [sigmoid(np.dot(X_scaled, theta_optimized1.flatten())) >=
 0.5]
print(f'Train accuracy using gradientDescent: {np.mean(pred1 == y.
 flatten()) * 100:.4f}%')
pred2 = [sigmoid(np.dot(X, theta_optimized2.flatten())) >= 0.5]
print(f'Train accuracy using minimize: {np.mean(pred2 == y.flatten()
 ) * 100:.4f}%')
```

Both method give accuracy of `89.0000`%.

2.4 Plotting the decision boundary

The decision boundary in this example is linear and the result of optimization using Scipy's `minimize` can be plotted as follows:

```
pos = y[:,0] ==1
neg = y[:,0] ==0
plt.scatter(X[:,1][pos], X[:,2][pos], marker='+', label='Admitted')
plt.scatter(X[:,1][neg], X[:,2][neg], label='Not admitted')

x2_value= np.array([np.min(X[:,1]),np.max(X[:,1])])
y2_value=-(theta_optimized2[0] + theta_optimized2[1]*x2_value)/
 theta_optimized2[2]
plt.plot(x2_value,y2_value, 'g', label='scipy "minimize" optimized')
```

You can plot the decision boundary obtained from `gradientDescent` with the same code on `X_scaled`, which will result Figure 4.

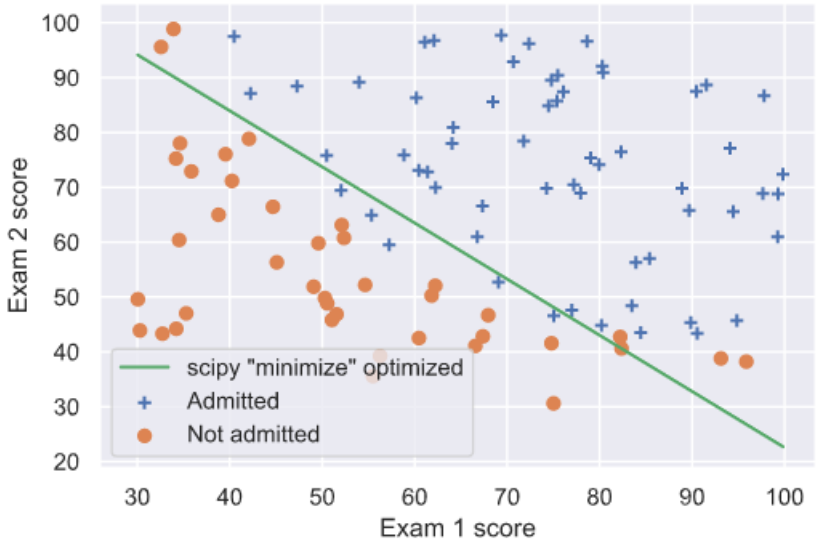

Figure 3: Training data with decision boundary from Scipy's `minimize`

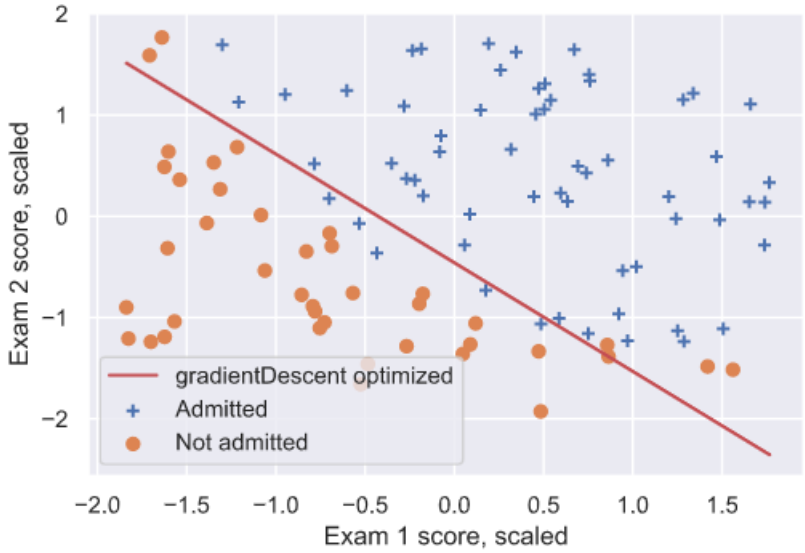

Figure 4: Training data with decision boundary from `gradientDescent`

3 Regularized logistic regression

In this part of the exercise, you will implement regularized logistic regression to predict whether microchips from a fabrication plant passes quality assurance (QA). During QA, each microchip goes through various tests to ensure it is functioning correctly.

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. This guide and the framework code in `exercise2-2.ipynb` will guide you through the exercise.

3.1 Visualizing the data

Similar to the previous parts of this exercise, we plot the data to generate a figure like Figure 5, where the axes are the two test scores, and the positive ($y = 1$, accepted) and negative ($y = 0$, rejected) examples are shown with different markers.

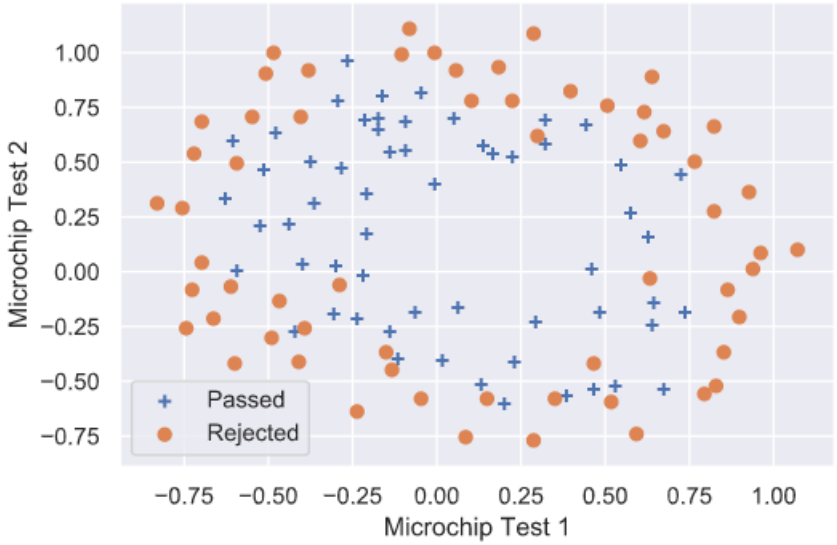

Figure 5: Plot of training data

Figure 5 shows that our dataset cannot be separated into positive and negative examples by a straight-line through the plot. Therefore, a straightforward application of logistic regression will not perform well on this dataset since logistic regression will only be able to find a linear decision boundary.

3.2 Feature mapping

One way to fit the data better is to create more features from each data point. In the `mapFeature`, we will map the features into all polynomial terms of x_1 and x_2 up to the sixth power.

$$\mathbf{m}apFeature(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \\ x_1^3 \\ \vdots \\ x_1 x_2^5 \\ x_2^6 \end{bmatrix} \tag{6}$$

As a result of this mapping, our vector of two features (the scores on two QA tests) 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.

While the feature mapping allows us to build a more expressive classifier, it also more susceptible to overfitting. In the next parts of the exercise, you will implement regularized logistic regression to fit the data and also see for yourself how regularization can help combat the overfitting problem.

3.3 Cost function and gradient

Now you will implement code to compute the cost function and gradient for regularized logistic regression. Make a function to compute the cost (`costFunctionReg`) and gradient (`grad`).

Recall that the regularized cost function in logistic regression is

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left[-y^{(i)} log(h_\theta(x^{(i)})) - (1 - y^{(i)} log(1 - h_\theta(x^{(i)}))) \right] + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2 \tag{7}$$

```
def costFunctionReg(theta, X, y, lmbd):
m = len(y)
z = np.dot(X, theta)
term1 = -y*np.log(sigmoid(z))
term2 = (1-y)*np.log(1-sigmoid(z))
J = (1/m)*np.sum(term1 - term2) + lmbd/(2*m)*np.sum(theta**2)
return J
```

The gradient of the cost function is a vector where the j^{th} element is defined as follows:

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \qquad \text{for} j = 0 \tag{8}$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j \qquad \text{for} j \geq 0 \tag{9}$$

```
def grad(theta, X, y, lmbd):
m = len(y)
h = sigmoid(np.dot(X,theta))
grad = (1/m)*np.dot(X.T, h-y)
return grad + (lmbd/m)*theta
```

Using `lmbd`(λ) of 1, the cost is about `0.693`.

3.4 Learning parameters

3.4.1 Learning parameters using `gradientDescent`

The code for `gradientDescent` is based on the following equation:

$$\theta_j := \theta_j - \alpha\left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j\right) \tag{10}$$

The `gradientDescent` wiill return the value of optimized θ and the $J(\theta)$ after each iteration.

```
def gradientDescent(X, y, theta_initial, alpha, iterations, lmbd):
m = len(y)
theta = theta_initial.copy()
J_val = []
for i in range(iterations):
    grad2 = grad(theta, X, y, lmbd)
    cost = costFunctionReg(theta, X, y, lmbd)
    theta -= alpha*grad2
    J_val.append(cost)
return theta, J_val
```

We can determine the α and λ value through iterations and plot the computed cost after each iteration. For example, here we try `alpha` value of `0.01`, `0.03`, `0.1`, `0.3`, and `1`. Using similar code to search for λ value we wil get Figure 7.

```
iterations = 800
alpha = [0.01, 0.03, 0.1, 0.3, 1]
theta_optimized1 = 0
for i in alpha:
```

```
    theta_optimized1, J_value =  gradientDescent(X_mapped, y, theta,
  alpha=i, iterations=iterations, lmbd=1)
    plt.plot(np.arange(1,iterations+1), J_value, label='alpha = '+
str(i))
```

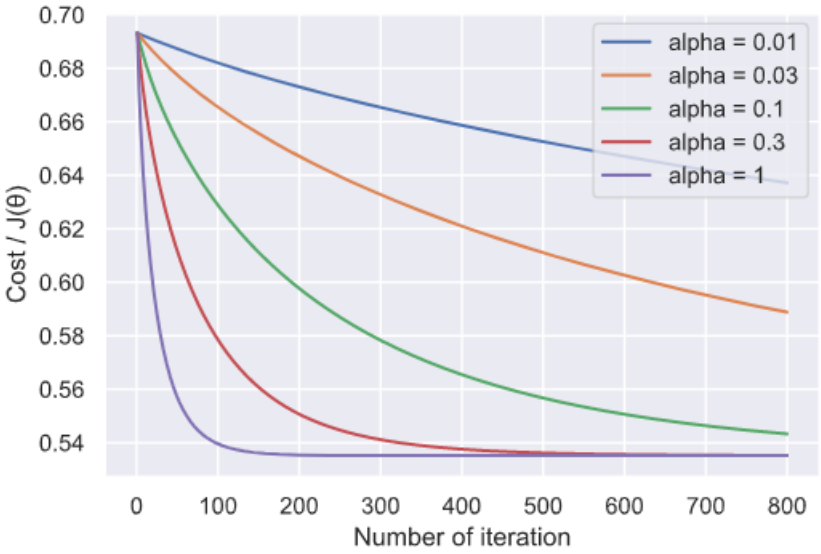

Figure 6: Cost for each iteration for different α

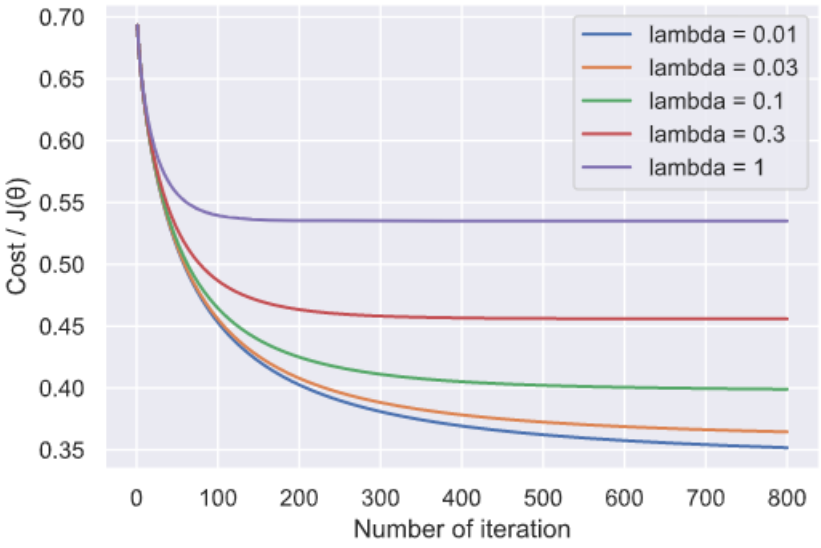

Figure 7: Cost for each iteration for different λ

From Figure 6, $\alpha = 1$ gives the steepest decrease of the cost. From Figure 7,

we will choose `lmbd` of 0.03 where the cost (J_θ) is decreasing more rapidly. The `gradientDescent` returns the optimized θ value which has 28 elements.

Using the optimized `alpha` and `lmbd`, we can compute the optimized θ and plot the cost function as follows:

```
theta_optimized1, J_value =  gradientDescent(X_mapped, y, theta,
 alpha=1, iterations=iterations, lmbd=0.03)
plt.plot(np.arange(1,iterations+1), J_value)
```

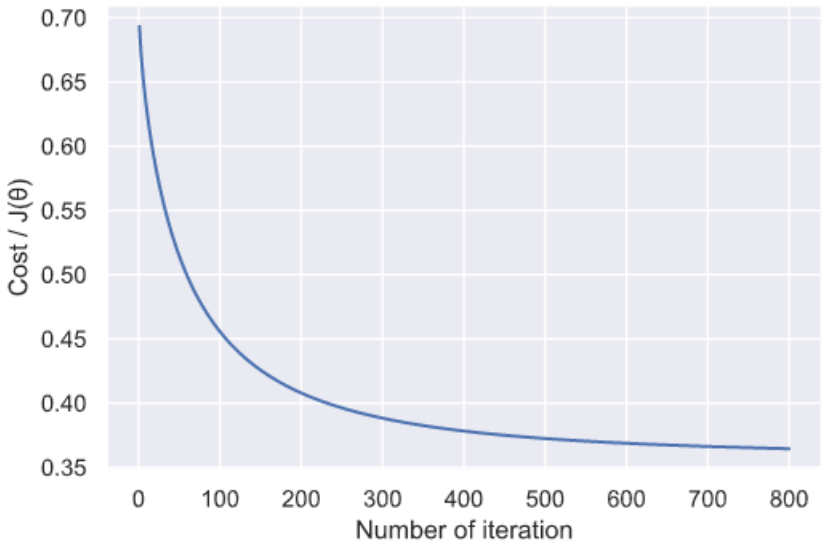

Figure 8: Cost for each iteration

The cost is computed using `costFunctionReg`, `theta_optimized1` and $\lambda = 0.03$, which gives value of `0.3644891`, lower than when the θ was 0, which was `0.693` in Section 3.3.

```
J_optimized1 = costFunctionReg(theta_optimized1, X_mapped, y, lmbd
 =0.03)
```

3.4.2 Learning parameters using `minimize`

Much like optimizing θ in Section 2.3.2, you will use `minimize` function from Scipy's `optimize` module. The only difference with the code in Section 2.3.2 is we need the value of λ for the regularization parameter.

```
from scipy.optimize import minimize
lmbd = 0.03
temp = minimize(fun = costFunctionReg,
                    x0 = theta.flatten(),jac = grad,
                    args = (X_mapped, y.flatten(), lmbd),
                    method='Newton-CG')
theta_optimized2 = temp.x
```

The cost is computed using `costFunctionReg`, `theta_optimized2` and $\lambda = 0.03$, which gives value of `0.356175`, lower than when the θ was 0, which was `0.693`in Section 3.3.

3.5 Plotting the decision boundary

3.5.1 Comparing `gradientDescent` and `minimize`

To visualize the model learned by this classifier, we plot the (non-linear) decision boundary that separates the positive and negative examples by computing the classifier's predictions on an evenly spaced grid and then and drew a contour plot of where the predictions change from y = 0 to y = 1. We write a function called `mapFeaturePlot` to help map the features.

```
def mapFeaturePlot(X1, X2, degree):
out = np.ones(1)
for i in range(1,degree+1):
    for j in range(i+1):
        feature = (X1**(i-j) * X2**j)
        out = np.hstack((out,feature))
return out
```

The code below is plotting the decision boundary with θ obtained from `gradientDescent` (`theta_optimized1`) and from `minimize` (`theta_optimized2`). The result is in Figure 9.

```
u_vals = np.linspace(-1,1.5,50)
v_vals= np.linspace(-1,1.5,50)
z1=np.zeros((len(u_vals),len(v_vals)))
z2=np.zeros((len(u_vals),len(v_vals)))

for i in range(len(u_vals)):
    for j in range(len(v_vals)):
        z1[i,j] = np.dot(mapFeaturePlot(u_vals[i],v_vals[j],6),
 theta_optimized1)

for i in range(len(u_vals)):
    for j in range(len(v_vals)):
        z2[i,j] = np.dot(mapFeaturePlot(u_vals[i],v_vals[j],6),
 theta_optimized2.reshape(len(theta_optimized2),1))

plt.contour(u_vals,v_vals,z1.T,0, colors='g', label='gradientDescent
 ')
plt.contour(u_vals,v_vals,z2.T,0, colors='r', label='minimize')

plt.xlabel('Microchip Test 1')
plt.ylabel('Microchip Test 2')
```

The green line is the boundary line of `gradientDescent` method, while the red line is the boundary line of `minimize`. As seen in Figure 9, the two methods have similar

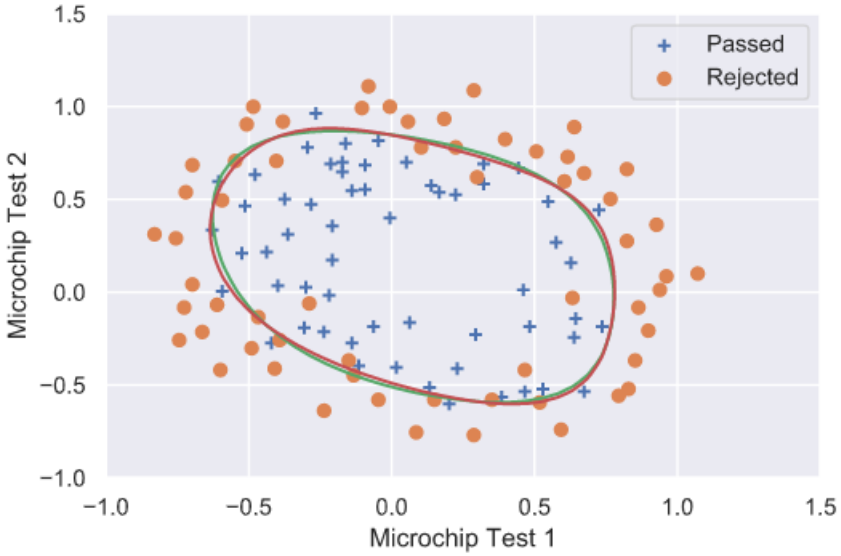

Figure 9: Training data with decision boundary ($\lambda = 1$)

boundary line. We can evaluate our methods by calculating the training accuracy as we did in Section 2.3.3. Below is the code to check the training accuracy:

```
pred1 = [sigmoid(np.dot(X_mapped, theta_optimized1.flatten())) >=
  0.5]
print(f'Train accuracy using gradientDescent: {np.mean(pred1 == y.
  flatten()) * 100:.4f}%')
pred2 = [sigmoid(np.dot(X_mapped, theta_optimized2)) >= 0.5]
print(f'Train accuracy using fmin_tnc: {np.mean(pred2 == y.flatten()
  ) * 100:.4f}%')
```

`minimize` method has a slightly higher training accuracy, `84.7458`%, compared to that of `gradientDescent` which is only `83.8983`%.

We can try out different regularization parameters (λ) for the dataset to understand how regularization prevents overfitting. Notice the changes in the decision boundary as you vary λ. With a small λ, you should find that the classifier gets almost every training example correct, but draws a very complicated boundary, thus overfitting the data (Figure 10).

With a larger λ, you should see a plot that shows an simpler decision boundary which still separates the positives and negatives fairly well. However, if λ is set to too high a value, you will not get a good fit and the decision boundary will not follow the data so well, thus underfitting the data (Figure 11).

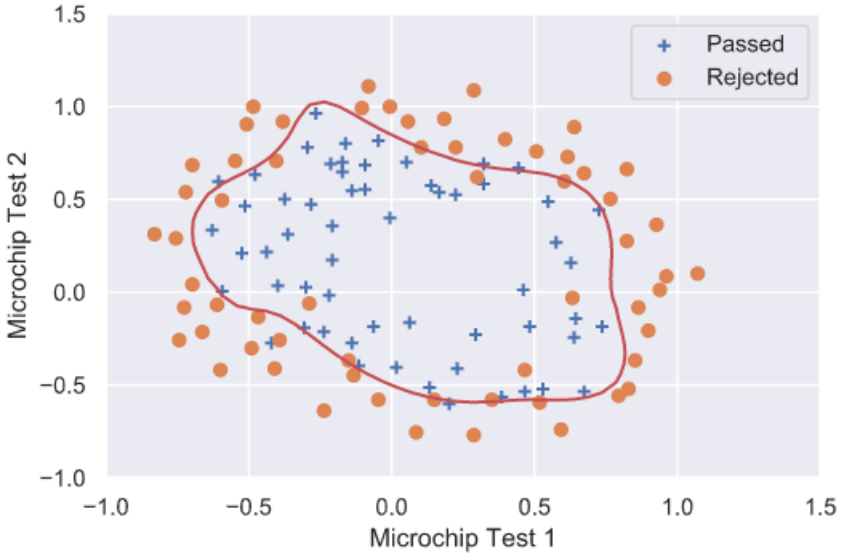

Figure 10: No regularization (Overfitting) ($\lambda = 0$)

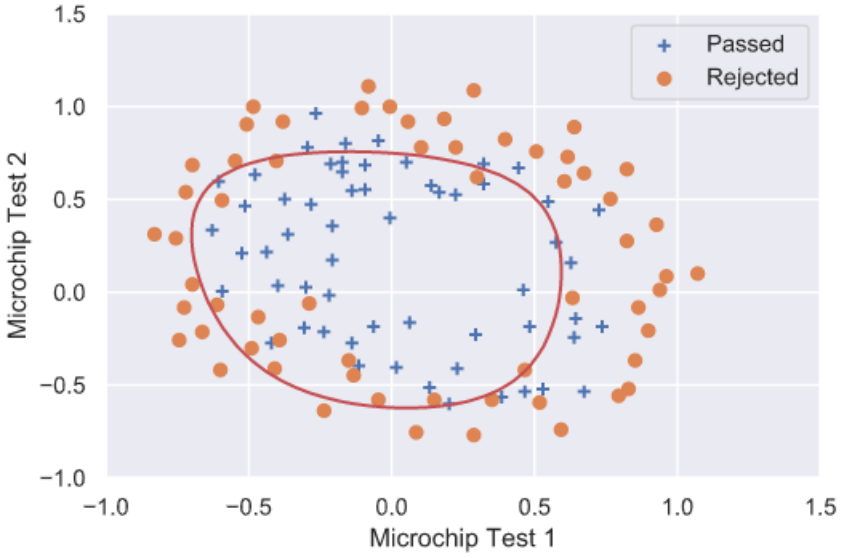

Figure 11: Too much regularization (Underfitting) ($\lambda = 100$)