General Sir John Kotelawala Defense University

Department of Electrical, Electronics & Telecommunication Engineering

Machine Learning

ET 4103

Assignment – 02

Regularized Logistic Regression

Index No	: D/ENG/22/0120/ET
Name	: M. A. E. Wijesuriya
Intake	: 39
Submission Date	: 20/06/2025

Q1. Utilize the given Jupyter notebook[1] for Regularized Logistic Regression. Comment on the code and the output of the program, explaining utilized Machine Learning concepts where necessary

The following code is a python program that demonstrates Regularized Logistic Regression. Logistic Regression is a type of statistical model used to classify data into binary outcomes. It is a supervised learning algorithm that used a sigmoid function to generate probability values for a set of linear inputs. This probability value is then used to classify the data into one of two classes.

Regularized Logistic Regression utilizes a regularization parameter (lambda) that is used to prevent overfitting, by adding a penalty term to the cost function.

Code with Explanation:

(text in *italics*, along with any graphs or tables, are the output of the preceding code segment)

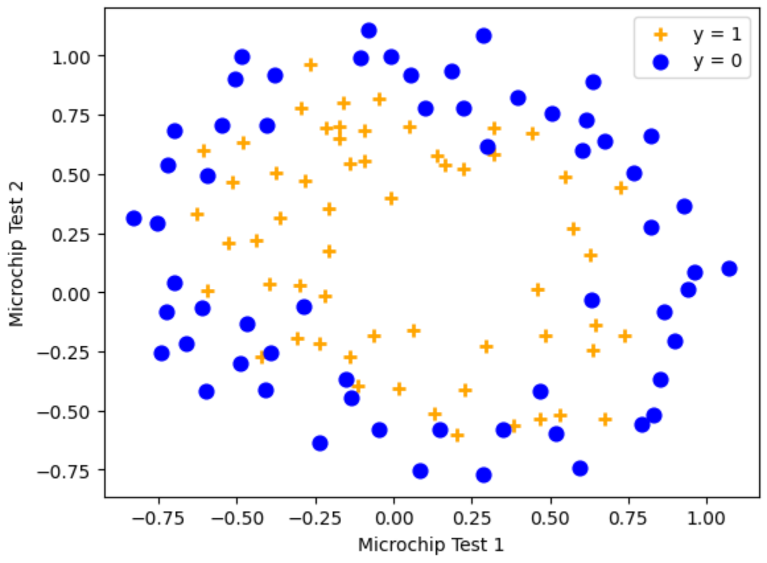
```
from google.colab import drive
drive.mount('/content/drive') # Grants Colab access to Google Drive in
order to retrieve the data files
%cd "/content/drive/MyDrive/ML_files"
```

```
/content/drive/MyDrive/ML_files
```

```
# Importing Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


# Retrieving the data file
data_path = 'ex2data2.txt'
data = pd.read_csv(data_path, header=None, names = ["x1","x2","y"])

data.head() # Prints the first 5 rows of the data in a table
```

	x1	x2	y
0	0.051267	0.69956	1
1	-0.092742	0.68494	1
2	-0.213710	0.69225	1
3	-0.375000	0.50219	1
4	-0.513250	0.46564	1

```
# Generates a scatter plot of the data with negative data marked with
blue dots, and positive data marked with yellow crosses
def plotData(data, label_x, label_y, label_pos, label_neg, axes=None):
    # Get indexes for class 0 and class 1
    neg = data['y'] == 0
    pos = data['y'] == 1

    # If no specific axes object has been passed, get the current axes.
    if axes == None:
        axes = plt.gca()
    axes.scatter(data[pos]['x1'], data[pos]['x2'], marker='+',
c='orange', s=60, linewidth=2, label=label_pos)
    axes.scatter(data[neg]['x1'], data[neg]['x2'], c='blue', s=60,
label=label_neg)
    axes.set_xlabel(label_x)
    axes.set_ylabel(label_y)
    axes.legend(frameon= True, fancybox = True);
```



```
n = data.shape[1]-1
x = data[data.columns[0:n]]

y = data[data.columns[n:n+1]]
# convert to np.array
X = x.values
y = y.values

## Feature mapping
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(6)
XX = poly.fit_transform(X)
print(X.shape, XX.shape) # Shows the change in the array X before and
after transformation
```

```
(118, 2) (118, 28)
```

```
## Regularized cost function
m = y.shape[0]

# Defines the Sigmoid function
def sigmoid(z):
    return(1 / (1 + np.exp(-z)))

# Defines the cost function
def Cost(theta, reg, XX,y):
    h = sigmoid(XX.dot(theta))
    J = -1*(1/m)*(np.log(h).T.dot(y)+np.log(1-h).T.dot(1-y)) +
(reg/(2*m))*np.sum(np.square(theta[1:]))
    if np.isnan(J[0]):
        return(np.inf)
    return (J[0])

# testing with reg =1
theta_initial = np.zeros(XX.shape[1])
Cost(theta_initial,1,XX,y)
```

```
np.float64(0.6931471805599454)
```

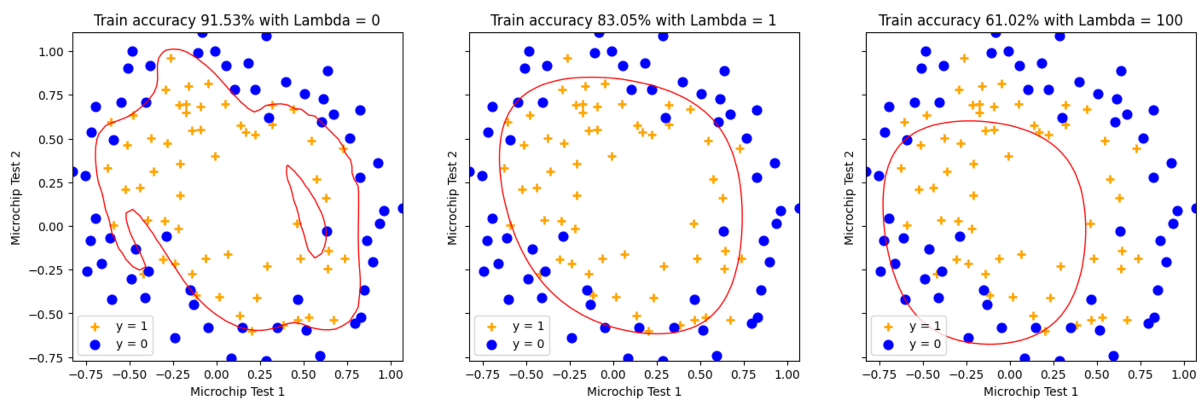
```
theta_initial.shape # Gives the shape of the intial theta value
```

```
(28,)
```

```
## Partial derivative (with regularization)

def gradientReg(theta, reg, *args):

    h = sigmoid(XX.dot(theta.reshape(-1,1)))

    grad = (1/m)*XX.T.dot(h-y) +
(reg/m)*np.r_[[[0]],theta[1:].reshape(-1,1)]

    return(grad.flatten())

### Optimization (using the minimize algorithm)

from scipy.optimize import minimize
# reg = 0
res = minimize(Cost, theta_initial, args=(0, XX, y), method=None,
jac=gradientReg, options={'maxiter':3000})

res.x.shape # Gives the shape of the optimal theta values
```

```
(28,)
```

```
res.x # Prints the values of the optimal theta for minimizing the cost
function

array([   35.10191603,    44.11916362,    69.27189416,  -344.27909705,
        -198.23463043,  -184.22842064,  -295.8204313 ,  -621.73277966,
        -510.8493901 ,  -328.31189673,  1094.70042861,  1269.58591712,
        1757.74920592,   900.9379677 ,   436.58887509,   471.12033517,
        1236.23866847,  1822.82041592,  1929.66786448,  1131.05336056,
         463.79937972, -1142.11743445, -2020.95915332, -3463.39994523,
       -3484.51083578, -3252.2679351 , -1546.00965736,  -510.41277032])


# Binary classifier prediction function
def predict(theta, X, threshold=0.5):
    p = sigmoid(X.dot(theta.T)) >= threshold
    return(p.astype('int'))


accuracy = 100*sum(predict(res.x, XX) == y.ravel())/y.size
accuracy  # for C = reg = lambda = 0

np.float64(61.016949152542374)

# Effect of lambda
lambda_set = [0,1,100]
fig, axes = plt.subplots(1,len(lambda_set), sharey = True,
figsize=(17,5))

# Decision boundaries
# Lambda = 0 : No regularization (overfitting)
# Lambda = 1 : Looks about right
# Lambda = 100 : Too much regularization --> high bias (underfitting)

x1_min, x1_max = data['x1'].min(), data['x1'].max(),
x2_min, x2_max = data['x2'].min(), data['x2'].max(),
xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max), np.linspace(x2_min,
x2_max))
B0 = np.linspace(x1_min,x1_max)
B1 = np.linspace(x2_min,x2_max)
Z = np.zeros((B0.size,B1.size))

for i, C in enumerate(lambda_set):
    # Optimize costFunctionReg
    res = minimize(Cost, theta_initial, args=(C, XX, y), method=None,
jac=gradientReg, options={'maxiter':3000})

    def h(x1,x2):
        a=poly.fit_transform(np.c_[x1.ravel(),x2.ravel()])
        return sigmoid(a.dot(res.x))
    # Scatter plot of X,y
    plotData(data, 'Microchip Test 1', 'Microchip Test 2', 'y = 1', 'y
= 0', axes.flatten()[i])

    # Contour plot
    for p in range (B0.size):
        for q in range(B1.size):
            Z[p,q] = h(xx1[p,q],xx2[p,q])

    axes.flatten()[i].contour(xx1, xx2, Z, [0.5], linewidths=1,
colors='red');

    # Accuracy
    accuracy = 100*sum(predict(res.x, XX) == y.ravel())/y.size
    axes.flatten()[i].set_title('Train accuracy {}% with Lambda =
{}'.format(np.round(accuracy, decimals=2), C))
```



The code above shows Regularized Logistic Regression being done on a common dataset of Microchips, that aims to classify the microchips into two classes, being acceptable or defective.

First, the data is expanded into a polynomial of degree 6 using the PolynomialFeatures function, allowing the classifier to understand more complex, non-linear boundaries within the dataset. The Sigmoid function, along with the cost function are then defined using mathematical theory, without utilizing inbuilt libraries. This allows us to see the actual calculations peformed on the data in order to create the prediction model.

The GradientReg function calculates the regularized gradient of the regression, which is then used by the minimize function in order to caculate the optimized theta values. Finally those values are fed into the predict function, which is a binary classifier that classifies data into defective or acceptable based on if they are greater than or less than the threshold value of 0.5

The final graph plots scatter plots for 3 different values of lambda, showing how low values for lambda lead to overfitting (with a complex boundary and poor generalization), while high values of lambda lead to underfitting (with a simple boundary and high bias).

Overfitting causes the model to be highly accurate on the training data, but not accurate on other real world data. Underfitting, on the other hand, creates an extremely general model that is very conservative in its estimates.

This highlights the importance of regularization in order to control the trade-off between bias and variance in logistic regression.

Code Source:

[1] https://colab.research.google.com/drive/1O_22CzqXuaJHEO7NZe2q-FgHcGhI9eEM