APL 405 Assigment-3

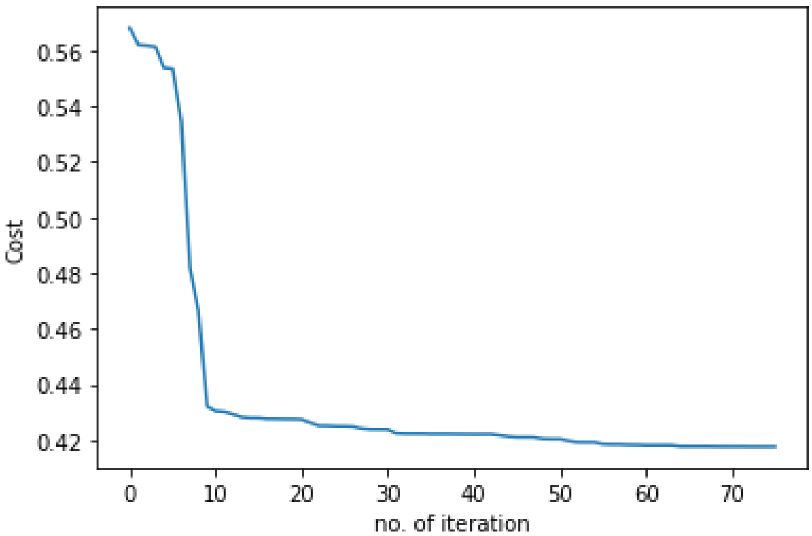
Nihal Pushkar

1.2 Mean of the Normalized 'MaxTemp'

```
[186] X,y,mean = lr().data_clean(df_train)
      mean

      0.4647423708243247
```

1.4 Plotting cost as function of iteration

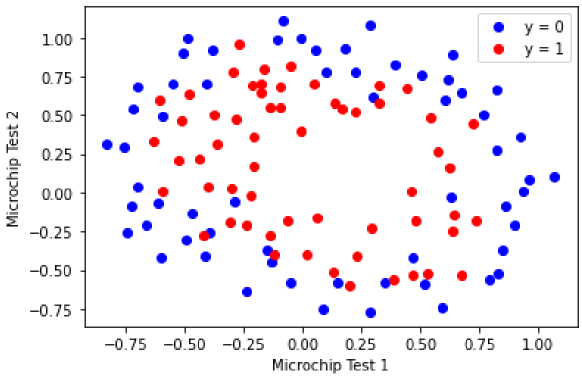
```
def minCostFun(self, w_ini, X_train, y_train, iters):
  # iters - Maximum no. of iterations; X_train - Numpy array
  lambda_ = 0.1  # Regularization parameter
  X_train = np.vstack([np.ones(len(X_train)),X_train.transpose()]).transpose()     # Add '1' for bias term

  def f(w):
    return costing().costFunctionReg(w,X_train,y_train,iters)[0]

  li = []
  i = 1
  def callbackF(xi):
    global i
    li.append(f(xi))

  res = optimize.minimize(f,w_ini, method = 'TNC',callback = callbackF,options = {'maxiter' : iters})
  w_opt = res.x   # Optimized weights rounded off to 3 decimal places
  p = costing().predict(w_opt,X_train)
  ones = np.ones(len(p))
  e = ones.transpose()@abs(p-y_train)/len(X_train)
  acrcy = (1-e)*100  # Training set accuracy (in %) rounded off to 3 decimal places
  return w_opt, acrcy,li
```



2.0 Load Data

```
# Load Data
datanl = np.loadtxt('/content/drive/MyDrive/GitHub/APL405-1/Week_03_Logistic_Regression/nonLinearClass.txt',delimiter = ',')
x1 = datanl[:,1]
x0 = datanl[:,0]
y = datanl[:,2]
```

2.1 Visualization of the Data



2.2 Feature mapping

2.2 Feature mapping

From the plot, it is clear that the dataset can not be separated into positive and negative example by using a straight line through the plot.

- One way to fit the data better is to create more features from each data point. We can define a function `mapFeature` in order to map the features into all polynomial terms of x_1 and x_2 up to the n^{th} power.

$$\text{mapFeature}(x) = \begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1x_2 & x_2^2 & x_1^3 & \dots & x_1x_2^{n-1} & x_2^n \end{bmatrix}^T$$

- As a result of this mapping, the **vector of two features (the scores on two QA tests)** will be transformed into a multi-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.

```
def mapFeature(X1, X2, degree):
  out = []
  for i in range(degree+1):
    for j in range(i+1):
      t = (X1**j)*(X2**(i-j))
      out.append(t)
  return np.array(out)
```

```
# Note that mapFeature should also add a column of ones, so the intercept/bias term is handled
degree = 6
m = len(x0)
X = []
for i  in range(m):
  X.append(mapFeature(x0[i],x1[i],degree))
X = np.array(X)
print(X.shape)
```

```
(118, 28)
```

2.3 Cost function and Gradient

2.3 Cost function and gradient

Now the code to compute the cost function and gradient for regularized logistic regression will be implemented in the function `costFunctionReg` below to return the cost and gradient.

Recalling that the regularized cost function in logistic regression is

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

Note that one should not regularize the parameters w_0. The gradient of the cost function is a vector where the j^{th} element is defined as follows:

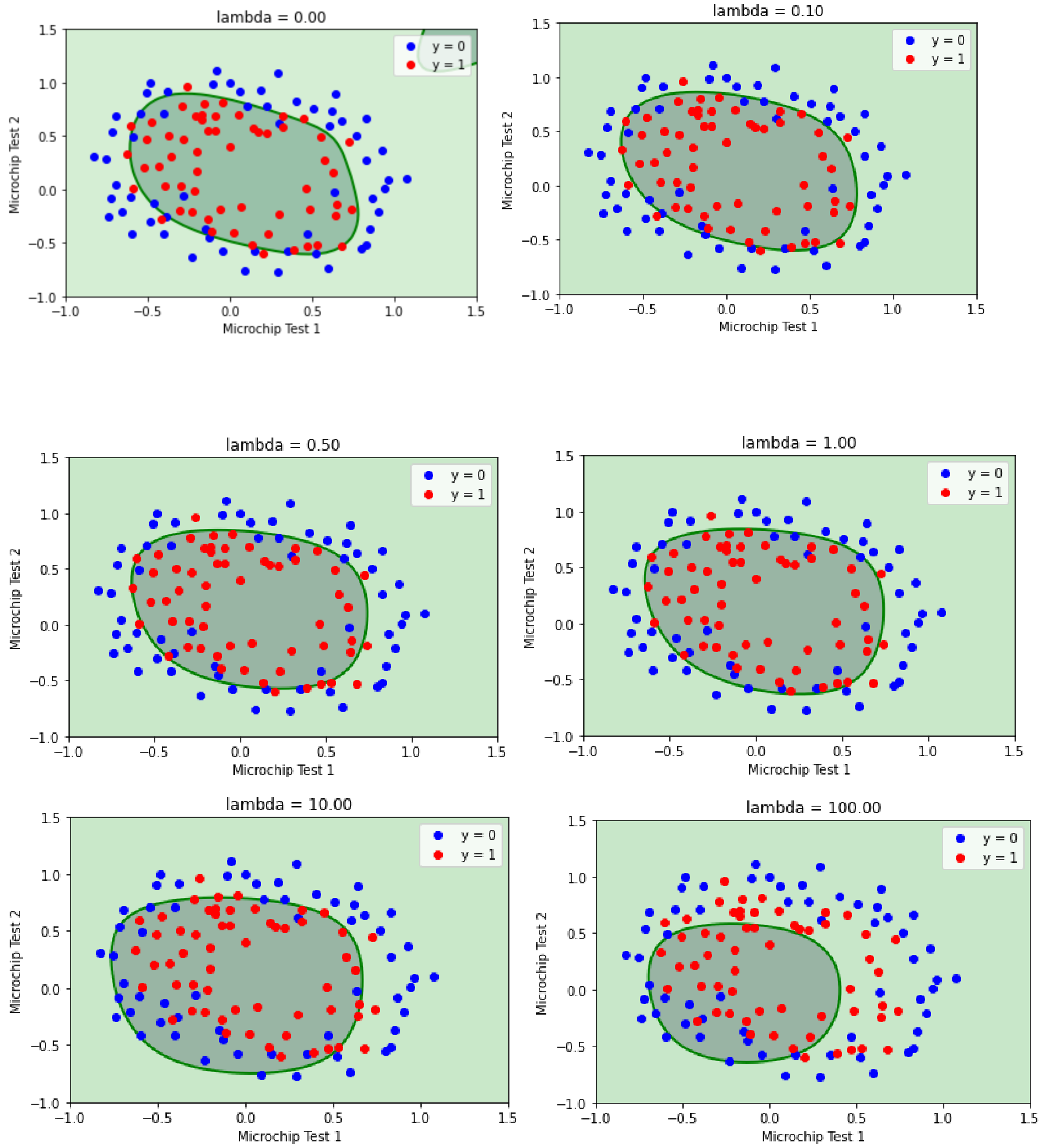
$$\frac{\partial J(w)}{\partial w_0} = \frac{1}{m}\sum_{i=1}^{m}\left(h_w\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)} \qquad \text{for } j = 0$$

$$\frac{\partial J(w)}{\partial w_j} = \left(\frac{1}{m}\sum_{i=1}^{m}\left(h_w\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}\right) + \frac{\lambda}{m}w_j \qquad \text{for } j \geq 1$$

```
[ ] def sigmoid(z):
      g = 1/(1+np.exp(-1*z))
      return g
```

```
def costFunctionReg(w, X, y, lambda_):
    # ============================================================
    p = len(w)    # no. of parameters, order(X) = m*p
    m = len(X)    # no. of training examples
    h = sigmoid(X@w)  # (m*p)*(p*1) = (m*1)
    sum1 = y.transpose()@np.log(h) + (1-y).transpose()@np.log(1-h)
    w2 = w.copy()
    w2[0] = 0
    sum2 = (lambda_/(2*m))*(w2.transpose()@w2)
    J = (-1*sum1/m) + sum2       # Cost 'J' should be a scalar
    grad = (1/m)*((X.transpose())@(h-y) + lambda_*w2)
    # ============================================================
    return J, grad
```

2.4 Plots with variation of the regularization term: λ = 0, 0.1, 0.5, 1, 10, 100



For **λ = 0** we observe that the model tries to get every positive training set. But while doing so the curves become sharp. Case of overfitting

Whereas for **λ = 0.1** we observe the curve to be smooth enough and being covering positive training points.

For **λ = 0.5, 1** curve is quite smooth but misses more points as compared to the case of **λ = 0.1**, but still defines the decision boundary good enough.

Whereas for **λ = 10, 100** the weights were so suppressed to reduce the cost that it couldn't even define the decision boundary properly.