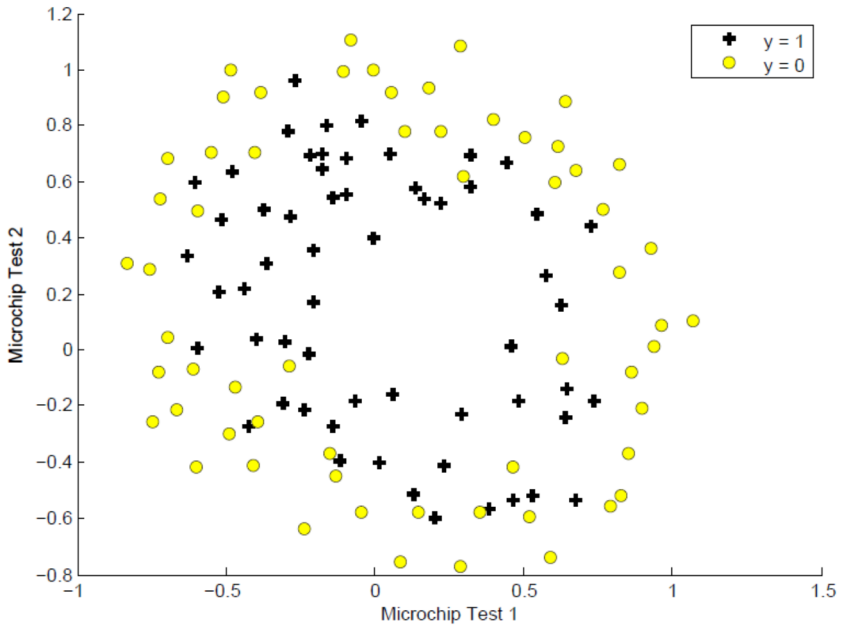
Homework 04

Regularized Logistic Regression

In this 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. You will use **hw4.m** to complete this portion of the exercise.

Part 1: Visualizing the data (0 points)

Before starting to implement any learning algorithm, it is always good to visualize the data if possible. In the first part of hw3.m, the code will load the data and display it on a 2-dimensional plot by calling the function **plotData**. It displays a figure like the following one, 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.



This figure 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.

Homework 04

Part 2: Feature mapping (0 points)

One way to fit the data better is to create more features from each data point. In the provided function **mapFeature.m**, we will map the features into all polynomial terms of x1 and x2 up to the sixth power.

$$\text{mapFeature}(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \\ x_1^3 \\ \vdots \\ x_1 x_2^5 \\ x_2^6 \end{bmatrix}$$

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.

Part 3: Cost Function and Gradient (50 points)

Now you will implement code to compute the cost function and gradient for regularized logistic regression. Complete the code in **costFunctionReg.m** to return the cost and gradient. Recall that the regularized cost function in logistic regression is

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

Note that you should not regularize the parameter theta0. In Octave, recall that indexing starts from 1, hence, you should not be regularizing the theta(1) parameter (which corresponds to theta0) in the code. The gradient of the cost function is a vector where the jth element is defined as follows:

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

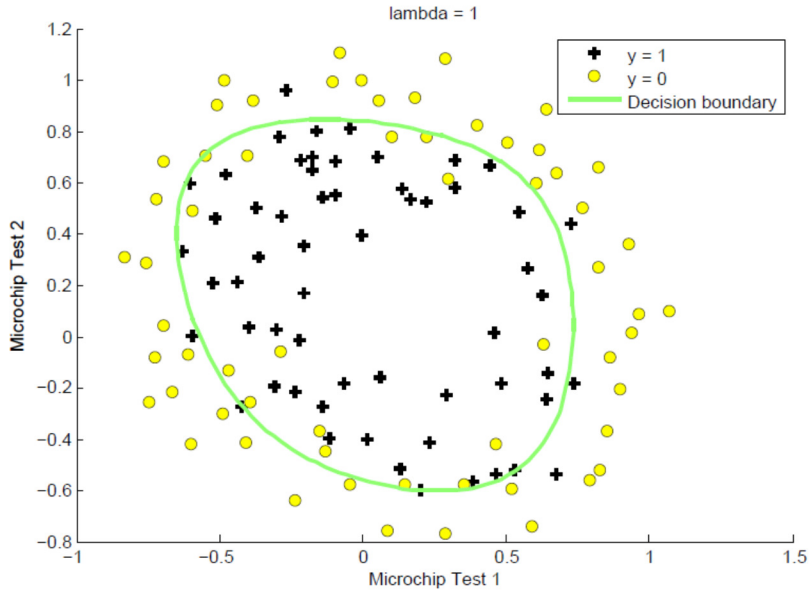
$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j \qquad \text{for } j \geq 1$$

Once you are done, **hw4.m** will call your **costFunctionReg** function using the initial value of theta (initialized to all zeros). You should see that the cost is about **0.693**.

Part 4: Learning parameters using fminunc (0 points)

Similar to the previous homework, you will use **fminunc** to learn the optimal parameters theta. If you have completed the cost and gradient for regularized logistic regression (**costFunctionReg.m**) correctly, you should be able to step through the next part of **hw4.m** to learn the parameters theta using **fminunc**.

To help you visualize the model learned by this classifier, we have provided the function **plotDecisionBoundary.m** which plots the (non-linear) decision boundary that separates the positive and negative examples. In **plotDecisionBoundary.m**, we plot the non-linear decision boundary 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. After learning the parameters theta, the next step in **hw4.m** will plot a decision boundary similar to the following one.



Submission:

To submit, turn in the following files on Canvas:

- **costFunctionReg.m**