UNIVERSIDADE DE AVEIRO
DEPARTAMENTO DE ELECTRÓNICA TELECOMUNICAÇÕES E INFORMÀTICA

Machine Learning (2018/19) – Lab work 3

Objectives: Implement unregularized and regularized logistic regression and get to see it works on data.

First, you need to download the starter code to the directory where you wish to complete the exercise.

Part 1 Logistic regression

Files included in this exercise:
ex2.m - Octave/MATLAB script that steps you through the exercise (the main program)
ex2data1.txt - Training dataset
plotDecisionBoundary.m - Function to plot classifier's decision boundary
plotData.m - Function to plot data (you need to finish this function)
costFunction.m - Logistic Regression Cost Function (you need to finish this function)
predict.m - Logistic Regression Prediction Function (you need to finish this function)

In this part of the lab work, 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 on the scores from those two exams.

1.1. Load and plot the Data

The file *ex2data1.txt* contains the dataset for our logistic regression problem. The first and the second columns are the scores from the exams, the third column indicates if the student was admitted (1) or not admitted (0). Load the data into variables X (first & second columns) and y (third column). Complete the *plotData.m* function to create a scatter plot of data. The axes are the two exam scores, and the positive and negative examples are shown with different markers, Fig.1

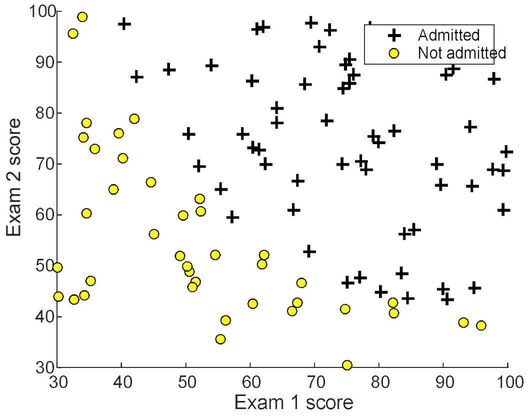

Fig. 1

1.2. Cost function and gradient

Now you will complete the code in *costFunction.m* to return the cost function and gradient for logistic regression. Recall that the logistic regression hypothesis is defined as:

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

where *g(.)* is the sigmoid function:

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

The cost function in logistic regression is

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

Matlab/Octave implementation : *J=sum(-y.*log(h)-(1-y).*log(1-h))/m; (.* - element-wise multiplication).*

and 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:

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

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(x).* After the main script calls *costFunction* with the initial parameters of θ, you should see that the cost is about 0.693.

1.3. Learning parameters using *fminunc*

In the previous lab work, you found the optimal parameters of a linear regression model by implementing gradient descent. You wrote a cost function and calculated its gradient, then took a gradient descent step accordingly. Now you will use an Octave/MATLAB built-in function called *fminunc*. *fminunc* is an optimization solver that finds the minimum of an unconstrained function. For logistic regression, you want to optimize the cost function $J(\theta)$ with parameters θ. *fminunc* will find the best parameters for the logistic regression cost function, given a dataset (X and y). *fminunc* needs the following inputs:

- The initial values of the parameters we are trying to optimize.
- A function that, when given the training set and a particular θ, computes the logistic regression cost and gradient with respect to the dataset (X, y).

In the main program, we already have code written to call *fminunc* with the correct arguments. We first defined the options to be used with *fminunc*. Specifically, we set the GradObj option to on, which tells *fminunc* that our function returns both the cost and the gradient. This allows *fminunc* to use the gradient when minimizing the function. Furthermore, we set the MaxIter option to 400, so that *fminunc* will run for at most 400 steps before it terminates. To specify the actual function we are minimizing, we use a short-hand for specifying functions with the *@(t) (costFunction(t, X, y))* . This creates a function, with argument t, which calls *costFunction*. Notice that by using *fminunc*, you did not have to write any loops yourself, or set a learning rate like you did for gradient descent. This is done by *fminunc*: you only needed to provide a function calculating the cost and the gradient.

Once *fminunc* completes, the main script will call *costFunction* function using the optimal parameters of θ. You should see that the cost is about 0.203. The final values of θ will then be used to plot the decision boundary on the training data (function *plotDecisionBoundary.m*), you should get a figure similar to Fig.2.

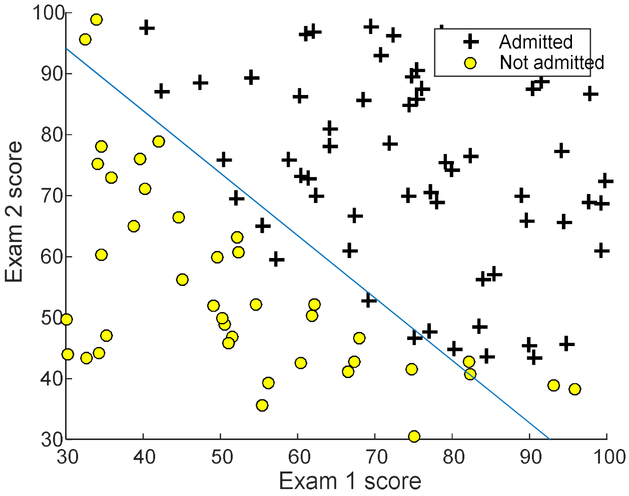

Fig.2

$$z = \theta^T x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0 \Rightarrow \text{decision boundary}$$
$$if \quad z > 0 \Rightarrow g(z) > 0.5 \Rightarrow y = 1$$
$$if \quad z < 0 \Rightarrow g(z) < 0.5 \Rightarrow y = 0$$

1.4 Evaluating logistic regression

For a student with an Exam1 score of 45 and an Exam2 score of 85, use the learned model to compute what is the admission probability of this student. The answer is around 77% probability (0.77).

1.5 Accuracy on training data

Evaluate how well the learned model predicts on the training set. Your task is to complete the code in *predict.m*. The predict function will produce 1 or 0 predictions given a dataset and a learned parameter vector θ. The training accuracy of the classifier by computing the percentage of examples it got correct is 89%.

Part 2 Regularized logistic regression

Files included in this exercise:

ex2_reg.m - Octave/MATLAB script that steps you through the exercise (the main program)
ex2data2.txt - Training dataset
plotDecisionBoundary.m - Function to plot classifier's decision boundary
mapFeature.m - Function to generate polynomial features
predict.m - Logistic Regression Prediction Function (you finished this function in part 1)
plotData.m - Function to plot 2D classification data (you finished this function in part 1)
costFunctionReg.m - Regularized Logistic Regression Cost (**you need to complete this function**)

In this part, you will implement regularized logistic regression to predict whether microchips from a fabrication plant pass 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 another script, *ex2_reg.m* to complete this exercise.

2.1 Load and plot the data

Load the data into Matlab/Octave and use the same function *plotData,* you created in part 1, to get the data (Fig.3). Now the axes are the two test scores, and the positive (y = 1, accepted) and negative (y = 0, rejected) examples are shown with different markers. The plot shows that the dataset cannot be separated into positive and negative examples by a straight-line. 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.

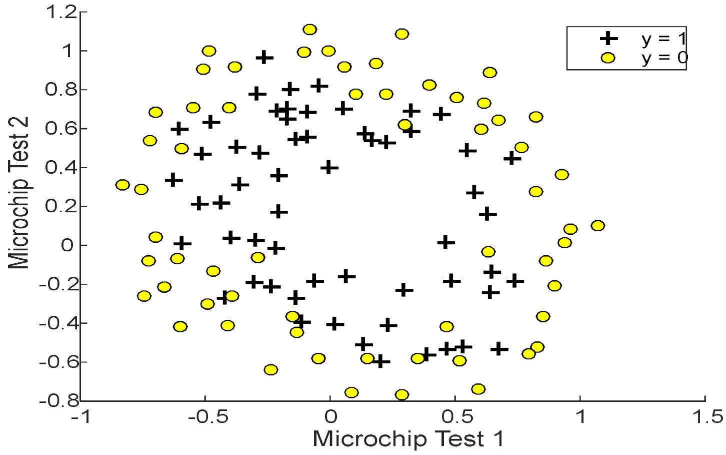

Fig. 3

2.2 Feature mapping

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, the 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 is also more susceptible to over fitting. Now, you will implement regularized logistic regression to fit the data and see how regularization can help combat the overfitting problem.

2.3 Cost function and gradient

Complete the code in *costFunctionReg.m* to return the cost function and gradient for regularized logistic regression (consult the lecture). Note that you should not regularize the parameter θ_0. In Octave/MATLAB, recall that indexing starts from 1, hence, you should not regularize theta(1) parameter (which corresponds to θ_0) in the code. Once you are done, call *costFunctionReg* function using the initial value of θ (initialized to all zeros). You should see that the cost is about 0.693.

Regularized cost function:

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

Matlab/Octave implementation :
*J=sum(-y.*log(h)-(1-y).*log(1-h))/m + theta(2:end)'*theta(2:end)*lambda/(2*m);*

Cost function gradient for θ_0:

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

Regularized cost function gradient for all other thetas:

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

Matlab/Octave implementation :
grad=(1/m)(h-y)'*X+ (lambda/m).*theta';*
grad(1)=(1/m)(h-y)'*X(:,1);*

2.4 Learning parameters using *fminunc*

Similar to the previous part, after you have completed the cost and gradient for regularized logistic regression (*costFunctionReg.m*) correctly, you will use *fminunc* to learn the optimal parameters θ.

2.5 Plotting the decision boundary

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.

2.6 Choice of the regularization parameter lambda λ

Try out different regularization parameters for the dataset to understand how regularization prevents overfitting. For example λ = (0, 1, 10, 100). With small λ, you should find that the classifier gets almost every training example correct, but draws a very complicated boundary, thus overfitting the data. This is not a good decision boundary: for example, it predicts that a test point (not included in the train data) at x = (-0-25; 1.5) is accepted (y = 1), which is an incorrect decision. With larger λ, you should see a plot that shows a simpler decision boundary which still separates the positives and negatives fairly well. However, if λ is set to a very high value, you will not get a good fit and the decision boundary will not follow the data so well, thus under-fitting the data. See Fig. 4.

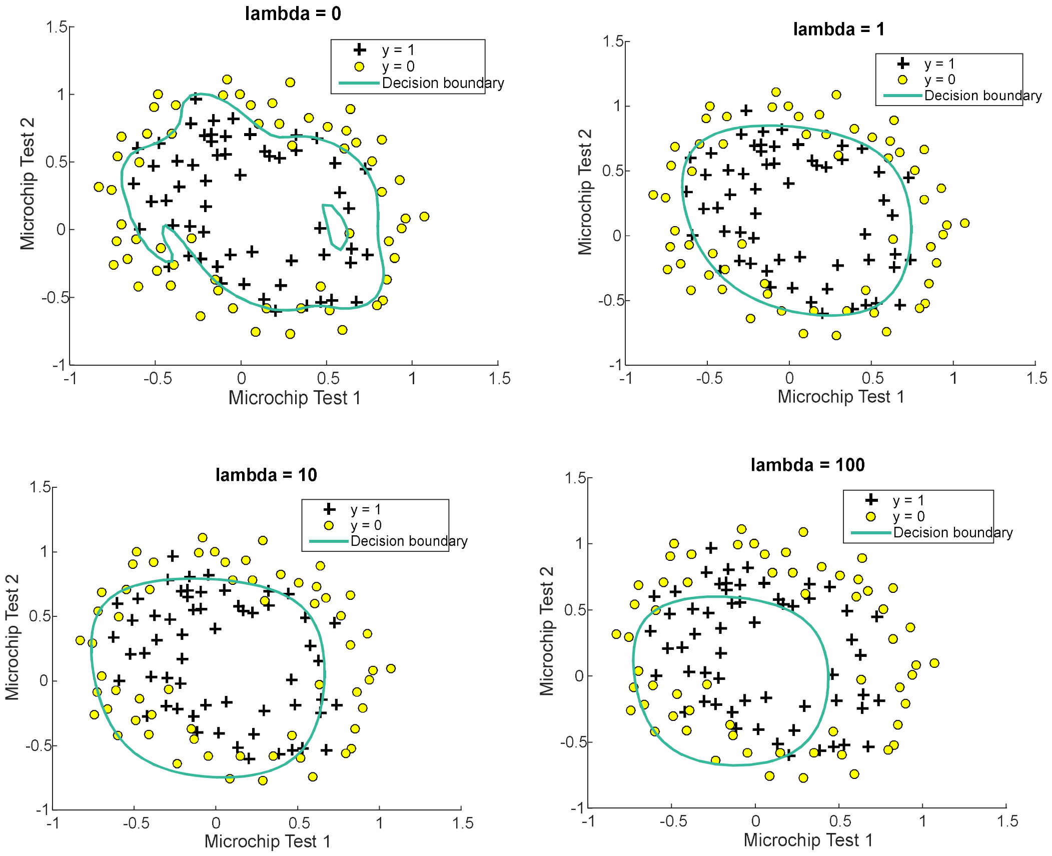

Fig. 4 Accuracy on training data: 87% (λ=0) | 83% (λ=1) | 74% (λ=10) | 61% (λ=100)