Laboratory # 2
Logistic regression

In groups of 2 develop the following exercises:

Files included in this laboratory:

1. exercise2.m - Script in Octave that will step up the exercise for you,
2. exercise2_reg.m - Script in Octave for the regularization part,
3. exercise2data1.txt - Training set for the first part of the exercise,
4. exercise2data2.txt - Training set for the second part of the exercise,
5. [*] plotData.m - Function to plot 2D classification data,
6. [*] sigmoid.m - Sigmoid Function,
7. [*] costFunction.m - Logistic Regression Cost Function,
8. [*] predict.m - Logistic Regression Prediction Function,
9. [*] costFunctionReg.m - Regularized Logistic Regression Cost.

* indicates the files you must complete

In this exercise you will build a logistic regression model to predict whether a student gets admitted into a university or not. Suppose that you are the Admission Department Coordinator of a university and you want to determine each applicant's chance of admission based on their results on two exams.

You have the 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 outline and the framework code in `exercise2.m` will guide you through this exercise.

1. Visualizing the data

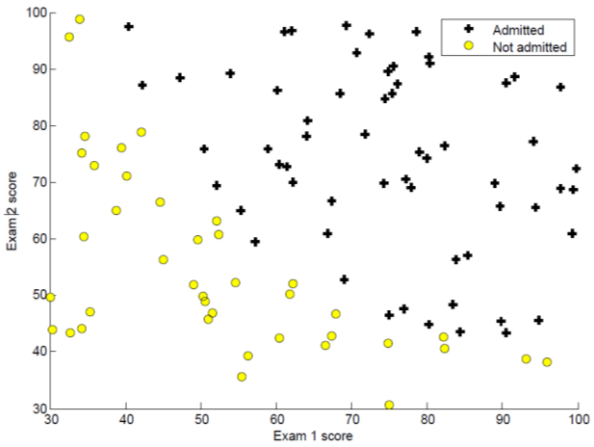


Figure 1: Scatter plot of training data

Remember, before starting to implement any learning algorithm, it is always good to visualize the data if possible. In the first part of `exercise2.m`, the code will load the data and display it on a 2-dimensional plot by calling the function `plotData`.

You will need to complete the code in `plotData` so that it displays a figure like Figure 1, where the axes are the two exam scores, and the positive and negative examples are shown with different markers.

To help you get more familiar with plotting, the `plotData.m` has been left empty, and so you can try to implement it yourself. You can refer to the code below, but I encourage you to learn what each of this commands is doing by consulting the Octave documentation.

```
% Find Indices of Positive and Negative Examples
pos = find(y==1); neg = find(y == 0);
% Plot Examples
plot(X(pos, 1), X(pos, 2), 'k+','LineWidth', 2, ...'MarkerSize',
7);
plot(X(neg,  1),  X(neg,  2),  'ko',  'MarkerFaceColor',  'y',
...'MarkerSize', 7);
```

2. Implementation

2.1 Sigmoid function

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

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

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

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

Your first step is to implement this function in `sigmoid.m` so it can be called by the rest of your program. When you are finished, try testing a few values by calling *`sigmoid(x)`* at the octave command line. 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. For a matrix, your function should perform the sigmoid function on every element.

2.2 Cost function and gradient descent

Now you will implement the cost function and gradient descent for logistic regression. Complete the code in `costFunction.m` to return the cost and gradient.

Recall that the cost function in logistic regression is:

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

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 $h_\theta(x)$.

Once you are done, `exercise2.m` will call your `costFunction` using the initial parameters of θ. You should see that the cost is about 0.693.

2.3 Learning parameters using *fminunc*

In laboratory 1 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.

This time, instead of taking gradient descent steps, you will use an Octave built-in function called *fminunc*.

Octave's *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.

You will use *fminunc* 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 *fminunc* 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 θ for the dataset (X, y)

`exercise2.m` already has code written to call *fminunc* with the corresponding arguments.

```
% Set options for fminunc
options = optimset('GradObj', 'on', 'MaxIter', 400);
% Run fminunc to obtain the optimal theta
% This function will return theta and the cost
[theta, cost] = ...
fminunc(@(t)(costFunction(t, X, y)), initial theta, options);
```

In this code excerpt, the options to be used with *fminunc* are defined first.

The `GradObj` option is set 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 function to be minimized, a "short-hand" is used for specifying functions with the `@(t)(costFunction(t, X, y))`. This creates a function, with argument *t*, which calls your `costFunction`. This allows us to wrap the `costFunction` for use with *fminunc*.

If you have completed the `costFunction` correctly, *fminunc* will converge on the right optimization parameters and return the final values of the *cost* and θ. Notice that by using *fminunc*, you don't have to write any loops yourself, or set a learning rate like you did for gradient descent. This is all done by *fminunc*: you only needed to provide a function calculating the cost and the gradient.

Once *fminunc* completes, `exercise2.m` will call your `costFunction` function using the optimal parameters of θ. You should see that the cost is about 0.203.

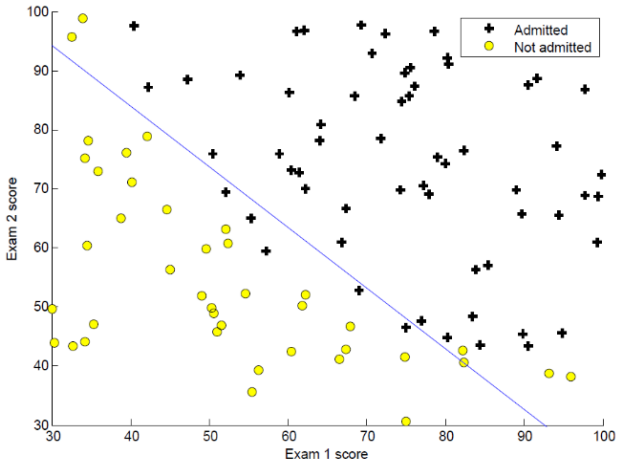


Figure 2: Training data with decision boundary

This final θ value will then be used to plot the decision boundary on the training data, resulting in a figure similar to Figure 2. I encourage you to look at the code in `plotDecisionBoundary.m` to see how to plot such a boundary using the θ values.

2.4 Evaluating logistic regression

After learning the parameters, you can use the model to predict whether a particular student will be admitted or not. 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.

Another way to evaluate the quality of the parameters we have found is to see how well the learned model predicts on our 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.

After you have completed the code in `predict.m`, the `exercise2.m` script will proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct.

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.

For this part of the exercise you will use the script, `exercise2_reg.m`.

3.1 Visualizing the data

Similar to the previous parts of this exercise, `plotData` is used to generate a graph like Figure 3, 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 3 shows that our dataset cannot be separated into positive and negative examples by a straight-line through the plot. Therefore, a straight-forward application of logistic regression will not perform well on this dataset since logistic regression will only be able to find a linear decision boundary.

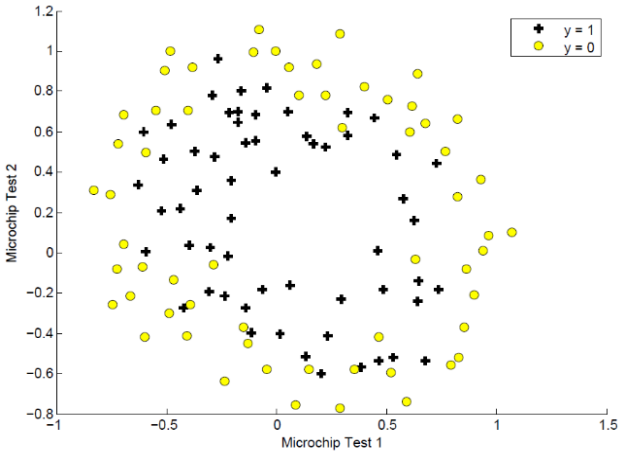


Figure 3: Plot of training data

3.2 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 x_1 and x_2 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 is also more susceptible to over fitting. 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 over fitting problem.

3.3 Cost function and gradient

Now you will implement code to compute the cost function and gradient descent 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:

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

Also, note that you should not regularize the parameter θ_0. In Octave, recall that indexing starts from 1, hence, you should not be regularizing the `theta(1)` parameter (which corresponds to θ_0) in the code. The gradient of the cost function is a vector where the j^{th} element is defined as follows:

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

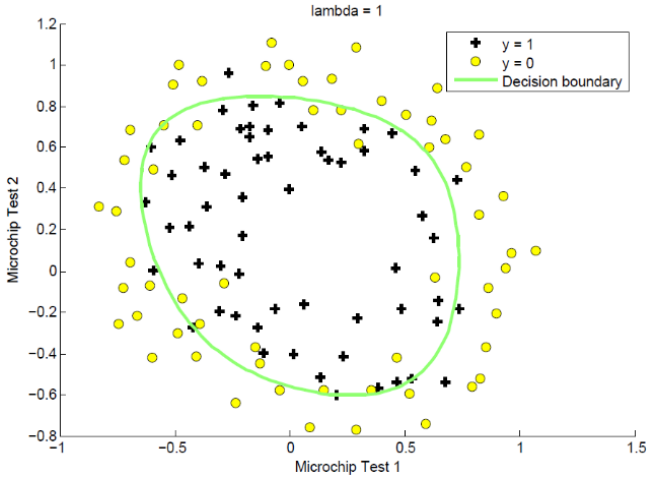
$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j \qquad \text{for } j \geq 1$$

Once you are done, `exercise2_reg.m` will call your `costFunctionReg` function using the initial value of θ (initialized to all zeros). You should see that the cost is about 0.693.

3.3.1 Learning parameters using *fminunc*

Similar to the previous parts, you will also use *fminunc* to learn the optimal parameters θ. 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 `exercise2_reg.m` to learn the parameters θ using *fminunc*.

3.4 Plotting the decision boundary



To help you visualize the model learned by this classifier, the function `plotDecisionBoundary.m` is provided, 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 drew a contour plot of where the predictions change from $y = 0$ to $y = 1$.

After learning the parameters θ, the next step in `exercise2_reg.m` will plot a decision boundary similar to Figure 4.

5. Submission and Grading

After completing the code in the necessary files, "up load" the modified code files in SIDWeb – section "Trabajos" – Laboratory 2. The following is a breakdown of how each part of this exercise is graded:

Task	Submitted File	Points
Sigmoid Function	sigmoid.m	5
Compute cost for logistic regression	costFunction.m	60
Gradient for logistic regression		
Predict Function	predict.m	5
Compute cost for regularized Logistic Regression	costFunctionReg.m	30
Gradient for regularized Logistic Regression		
Total points		**100**