Programming Exercise 2: Logistic Regression

Machine Learning

**Introduction**

In this exercise, you will implement logistic regression and apply it to two different datasets. Before starting on the programming exercise, we strongly recommend watching the video lectures and completing the review questions for the associated topics.

To get started with the exercise, you will need to download the starter code and unzip its contents to the directory where you wish to complete the

exercise. If needed, use the cd command in Octave to change to this directory

before starting this exercise.

You can also find instructions for installing Octave on the “Octave In- stallation” page on the course website.

**Files included in this exercise**

ex2.m - Octave script that will help step you through the exercise ex2 reg.m - Octave script for the later parts of the exercise ex2data1.txt - Training set for the first half of the exercise ex2data2.txt - Training set for the second half of the exercise submit.m - Submission script that sends your solutions to our servers mapFeature.m - Function to generate polynomial features

plotDecisionBounday.m - Function to plot classifier’s decision boundary

[*\**] plotData.m - Function to plot 2D classification data

[*\**] sigmoid.m - Sigmoid Function

[*\**] costFunction.m - Logistic Regression Cost Function

[*\**] predict.m - Logistic Regression Prediction Function

[*\**] costFunctionReg.m - Regularized Logistic Regression Cost

*\** indicates files you will need to complete

Throughout the exercise, you will be using the scripts ex2.m and ex2 reg.m. These scripts set up the dataset for the problems and make calls to functions that you will write. You do not need to modify either of them. You are only required to modify functions in other files, by following the instructions in this assignment.

**Where to get help**

The exercises in this course use Octave,1 a high-level programming language well-suited for numerical computations. If you do not have Octave installed, please refer to the installation instructons at the “Octave Installation” page on the course website.

At the Octave command line, typing help followed by a function name

displays documentation for a built-in function. For example, help plot will bring up help information for plotting. Further documentation for Octave functions can be found at the [Octave documentation pages](http://www.gnu.org/software/octave/doc/interpreter/).

We also strongly encourage using the online **Q&A Forum** to discuss exercises with other students. However, do not look at any source code written by others or share your source code with others.

If you run into network errors using the submit script, you can also use

an online form for submitting your solutions. To use this *alternative* submis- sion interface, run the submitWeb script to generate a submission file (e.g., submit ex2 part1.txt). You can then submit this file through the web submission form in the programming exercises page (go to the programming exercises page, then select the exercise you are submitting for). If you are having no problems submitting through the standard submission system us- ing the submit script, you do *not* need to use this alternative submission interface.

**1 Logistic Regression**

In this part of the exercise, 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

1 Octave is a free alternative to MATLAB. For the programming exercises, you are free to use either Octave or MATLAB.

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 the scores from those two exams. This outline

and the framework code in ex2.m will guide you through the exercise.

**1.1 Visualizing the data**

Before starting to implement any learning algorithm, it is always good to visualize the data if possible. In the first part of ex2.m, the code will load the data and display it on a 2-dimensional plot by calling the function plotData.

You will now 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.

100

90

Admitted

Not admitted

80

70

Exam 2 score

60

50

40

30

30 40 50 60 70 80 90 100

Exam 1 score

Figure 1: Scatter plot of training data

To help you get more familiar with plotting, we have left plotData.m empty so you can try to implement it yourself. However, this is an *optional (ungraded) exercise*. We also provide our implementation below so you can copy it or refer to it. If you choose to copy our example, make sure you learn what each of its 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);

**1.2 Implementation**

**1.2.1 Warmup exercise: sigmoid function**

Before you start with the actual cost function, recall that the logistic regres- sion hypothesis is defined as:

*hθ* (*x*) = *g*(*θT x*)*,*

where function *g* is the sigmoid function. The sigmoid function is defined as:

1

*g*(*z*) =

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.**

You can submit your solution for grading by typing submit at the Octave command line. The submission script will prompt you for your username and password and ask you which files you want to submit. You can obtain a sub- mission password from the website.

*You should now submit the warm up exercise.*

**1.2.2 Cost function and gradient**

Now you will implement the cost function and gradient 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* (*θ*) =

*m*

) [*−y*(*i*) log(*hθ* (*x*(*i*))) *−* (1 *− y*(*i*)) log(1 *− hθ* (*x*(*i*)))l *,*

1

*m*

*i*=1

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:

*∂J* (*θ*)

*∂θj*

*m*

= )(*hθ* (*x*(*i*)) *− y*(*i*))*x*(*i*)

*j*

1

*m*

*i*=1

Note that while this gradient looks identical to the linear regression gra- dient, the formula is actually different because linear and logistic regression have different definitions of *hθ* (*x*).

Once you are done, ex2.m will call your costFunction using the initial

parameters of *θ*. You should see that the cost is about 0.693.

*You should now submit the cost function and gradient for logistic re- gression. Make two submissions: one for the cost function and one for the gradient.*

**1.2.3 Learning parameters using** fminunc

In the previous assignment, you found the optimal parameters of a linear regression model by implementing gradent 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 unconstrained2 function. For logistic regression, you want to optimize the cost function *J* (*θ*) with parameters *θ*.

Concretely, you are going to 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:

❼ The initial values of the parameters we are trying to optimize.

2 Constraints in optimization often refer to constraints on the parameters, for example, constraints that bound the possible values *θ* can take (e.g., *θ ≤* 1). Logistic regression does not have such constraints since *θ* is allowed to take any real value.

❼ 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*)

In ex2.m, we already have code written to call fminunc with the correct 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 snippet, 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 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 did not 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, ex2.m will call your costFunction function using the optimal parameters of *θ*. You should see that the cost is about

0.203.

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. We also encourage you to look at the code in plotDecisionBoundary.m to see how to plot such a boundary using the *θ* values.

100

90

Admitted

Not admitted

80

70

Exam 2 score

60

50

40

30

30 40 50 60 70 80 90 100

Exam 1 score

Figure 2: Training data with decision boundary

**1.2.4 Evaluating logistic regression**

After learning the parameters, you can use the model to predict whether a particular student will be admitted. 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. In this

part, 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 ex2.m script will

proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct.

*You should now submit the prediction function for logistic regression.*

**2 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 assur- ance (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 portion of the

exercise.

**2.1 Visualizing the data**

Similar to the previous parts of this exercise, plotData is used to generate a figure 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.

1.2

1

y = 1 y = 0

0.8

0.6

0.4

Microchip Test 2

0.2

0

−0.2

−0.4

−0.6

−0.8

−1 −0.5 0 0.5 1 1.5

Microchip Test 1

Figure 3: Plot of training data

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.

**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 *x*1 and *x*2 up to the sixth power.

 1 

 *x*1 

 *x*2 





 2 

*x*

 1 

 

mapFeature(*x*) =  *x*1*x*2 





2

*x*





2





3

*x*





 1 

 

.

 

 *x*1*x*5 

 2 

6

*x*

2

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.

**2.3 Cost function and gradient**

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* (*θ*) =

*m*

) [*−y*(*i*) log(*hθ* (*x*(*i*))) *−* (1 *− y*(*i*)) log(1 *− hθ* (*x*(*i*)))l +

1

*m*

*i*=1

*λ* ) *θ*2*.*

2*m*

*j*

*n*

*j*=1

Note that you should not regularize the parameter *θ*0. In **Octave**, re- call 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:

*∂J* (*θ*)

*∂θ*0

*m*

= )(*hθ* (*x*(*i*)) *− y*(*i*))*x*(*i*)

*j*

1

*m*

*i*=1

for *j* = 0

*∂J* (*θ*)

*∂θj*

1 *m* \

= )(*hθ* (*x*(*i*)) *− y*(*i*))*x*(*i*) +

*j*

*m*

*i*=1

*λ*

*m θj* for *j ≥* 1

Once you are done, ex2 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.

*You should now submit the cost function and gradient for regularized lo- gistic regression. Make two submissions, one for the cost function and one for the gradient.*

**2.3.1 Learning parameters using** fminunc

Similar to the previous parts, you will 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 ex2 reg.m to learn the parameters *θ* using fminunc.

**2.4 Plotting the decision boundary**

To help you visualize the model learned by this classifier, we have pro- vided 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 com- puting 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 *θ*, the next step in ex reg.m will plot a decision boundary similar to Figure 4.

**2.5 Optional (ungraded) exercises**

In this part of the exercise, you will get to try out different regularization parameters for the dataset to understand how regularization prevents over- fitting.

Notice the changes in the decision boundary as you vary *λ*. With a small *λ*, you should find that the classifier gets almost every training example correct, but draws a very complicated boundary, thus overfitting the data (Figure 5). This is not a good decision boundary: for example, it predicts that a point at *x* = (*−*0*.*25*,* 1*.*5) is accepted (*y* = 1), which seems to be an incorrect decision given the training set.

With a larger *λ*, you should see a plot that shows an simpler decision boundary which still separates the positives and negatives fairly well. How- ever, if *λ* is set to too high a value, you will not get a good fit and the decision boundary will not follow the data so well, thus underfitting the data (Figure

6).

*You do not need to submit any solutions for these optional (ungraded)*

*exercises.*

1.2

1

0.8

lambda = 1

y = 1 y = 0

Decision boundary

0.6

0.4

Microchip Test 2

0.2

0

−0.2

−0.4

−0.6

−0.8

−1 −0.5 0 0.5 1 1.5

Microchip Test 1

Figure 4: Training data with decision boundary (*λ* = 1)

1.5

1

lambda = 0

y = 1 y = 0

Decision boundary

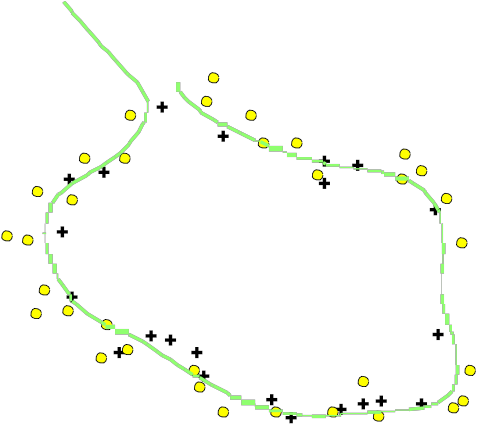
0.5

Microchip Test 2

0

−0.5

−1



−1 −0.5 0 0.5 1 1.5

Microchip Test 1

Figure 5: No regularization (Overfitting) (*λ* = 0)

1.2

1

0.8

lambda = 100

y = 1 y = 0

Decision boundary

0.6

0.4

Microchip Test 2

0.2

0

−0.2

−0.4

−0.6

−0.8

−1 −0.5 0 0.5 1 1.5

Microchip Test 1

Figure 6: Too much regularization (Underfitting) (*λ* = 100)

**Submission and Grading**

After completing various parts of the assignment, be sure to use the submit function system to submit your solutions to our servers. The following is a breakdown of how each part of this exercise is scored.

|  |  |  |
| --- | --- | --- |
| **Part** | **Submitted File** | **Points** |
| Sigmoid Function | sigmoid.m | 5 points |
| Compute cost for logistic regression  Gradient for logistic regression  Predict Function | costFunction.m  costFunction.m predict.m | 30 points  30 points  5 points |
| Compute cost for regularized LR  Gradient for regularized LR | costFunctionReg.m  costFunctionReg.m | 15 points  15 points |
| Total Points |  | 100 points |

You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration. To prevent rapid-fire guessing, the system enforces a minimum of 5 minutes between submissions.