Machine Learning Programming Exercise 2 : Logistic Regression

adapted to Python language from Coursera/Andrew Ng

25 septembre 2020

1 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.

2 Files included in this exercise

— **ex2.py** - Python script that will help step you through the exercise
— **ex2_reg.py** - Python script for the later parts of the exercise
— **ex2data1.txt** - Dataset for linear regression with one variable
— **ex2data2.txt** - Dataset for linear regression with multiple variables
— **plotDecisionBoundary.txt** - Plots the data points X and y into a new figure with the decision boundary defined by theta
⋆ **plotData.py** - function to display the dataset
⋆ **sigmoid.py** -Sigmoid Function
⋆ **gradientDescent.py** - Function to run gradient descent
⋆ **costFunction.py** - Logistic Regression Cost Function
⋆ **gradientFunction.py** - Logistic Regression Gradient Function
⋆ **costFunctionReg.py** - Regularized Logistic Regression Cost variables
⋆ **gradientFunctionReg.py** - Regularized Logistic Regression Gradient Function
⋆ **predict.py** - Logistic Regression Prediction Function

[⋆] indicates files you will need to complete

Throughout the exercise, you will be using the scripts **ex2.py** and **ex2_reg.py**. 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.

3 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 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.py** will guide you through the exercise.

3.1 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.py**, 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.

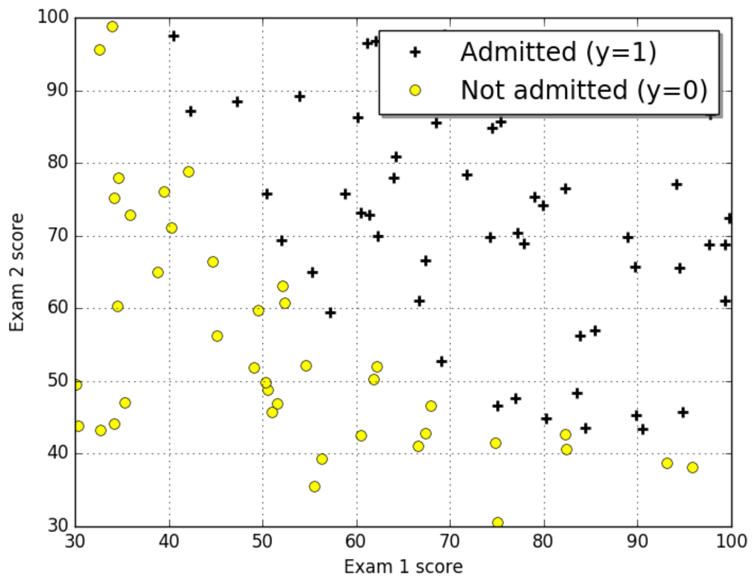

FIGURE 1 – Scatter plot of training data

To help you get more familiar with plotting, we have left **plotData.py** empty so you can try to implement it yourself. However, this is an optional 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 Python documentation.

```
pos = X[(y==1).flatten(),:]
neg = X[(y==0).flatten(),:]

plt.plot(pos[:,0], pos[:,1], '+', markersize=7,
 markeredgecolor='black', markeredgewidth=2)
plt.plot(neg[:,0], neg[:,1], 'o', markersize=7,
 markeredgecolor='black', markerfacecolor='yellow')

plt.legend(['Admitted (y=1)', 'Not admitted (y=0)'], loc='upper right',
shadow=True, fontsize='x-large', numpoints=1)
plt.grid()
plt.xlabel('Exam 1 score')
plt.ylabel('Exam 2 score')
```

3.2 Implementation

3.2.1 Warmup exercise : sigmoid function

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

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

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.py** 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 python 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.

3.2.2 Cost function and gradient

Now you will implement the cost function and gradient for logistic regression. Complete the code in **costFunction.py** and **gradientFunction.py** to return the cost and gradient. Recall that the cost function in logistic regression is

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

and the gradient of the cost is a vector θ where the j^{th} element (for $j = 0, 1, \ldots, n$) is defined as follows :

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

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, **ex2.py** will call your **costFunction** using the initial parameters of θ. You should see that the cost is about 0.693.

3.2.3 Learning parameters using fmin_tnc

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 python built-in function called **fmin_tnc**.

Python's **fmin_tnc** is an optimization solver that finds the minimum of an unconstrained [1] function. For logistic regression, you want to optimize the cost function $J(\theta)$ with parameters θ.

Concretely, you are going to use **fmin_tnc** 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 **fmin_tnc** the following inputs :

— A function that, when given the training set and a particular θ, computes the logistic regression cost for the dataset (X, y)

— 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 gradient with respect to θ for the dataset (X, y)

In **ex2.py**, we already have code written to call **fmin_tnc** with the correct arguments.

```
theta = opt.fmin_tnc(costFunction, initial_theta, gradientFunction,
                     args=(X, y))
```

If you have completed the costFunction correctly, **fmin_tnc** will converge on the right optimization parameters and return the final values of the cost and θ. Notice that by using **fmin_tnc**, 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 **fmin_tnc** : you only needed to provide a function calculating the cost and the gradient.

Once **fmin_tnc** completes, **ex2.py** 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.py** to see how to plot such a boundary using the θ values.

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

3.2.4 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 0.774.

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.py**. The predict function will produce 1 or 0 predictions given a dataset and a learned parameter vector θ.

After you have completed the code **predict.py**, the **ex2.py** script will proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct.

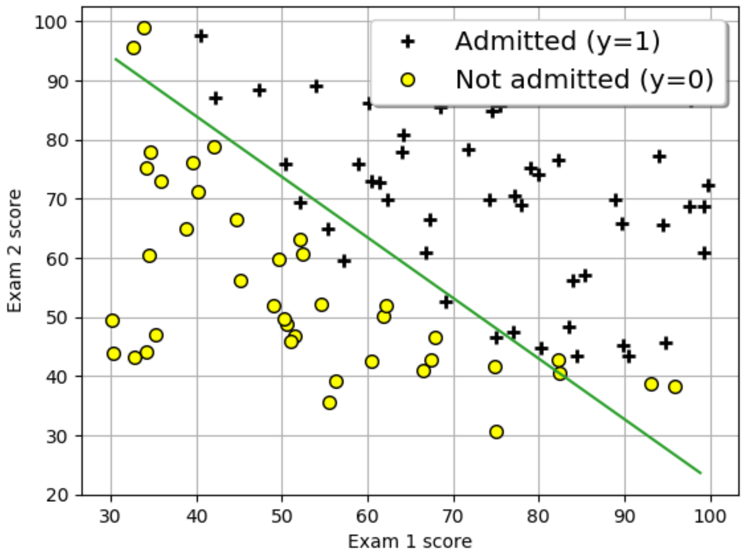

FIGURE 2 – Training data with decision boundary.

4 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.

You will use another script, **ex2_reg.py** to complete this portion of the exercise.

4.1 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.

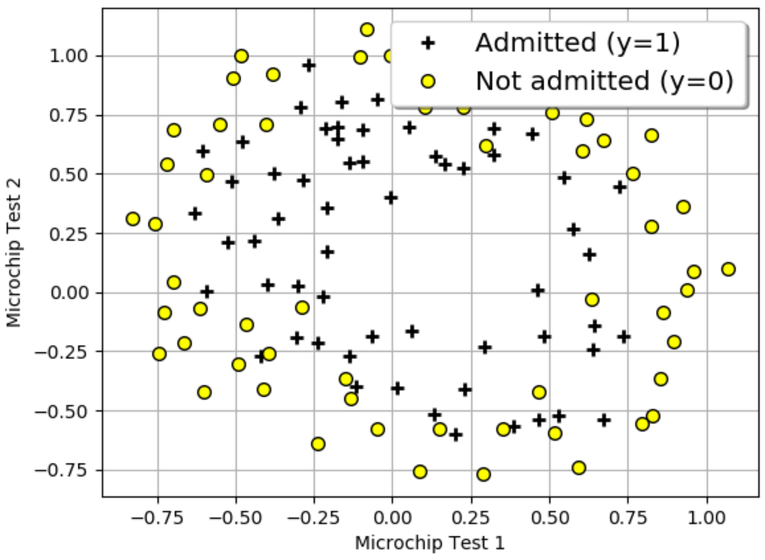

FIGURE 3 – Plot of training data.

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.

4.2 Feature mapping

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

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

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.

4.3 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.py** and **gradientFunctionReg.py** to return the cost and gradient.

Recall that the regularized cost function in logistic regression is

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

Note that you should not regularize the parameter θ_0 ; thus, the final summation above is for $j = 1$ to n, not $j = 0$ to n. The gradient of the cost function is a vector where the j-th element is defined as follows :

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_0^{(i)} \qquad \text{pour } j = 0$$

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

Once you are done, **ex2_reg.py** 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.

4.3.1 Learning parameters using fmin_tnc

Similar to the previous parts, you will use **fmin_tnc** to learn the optimal parameters θ. If you have completed the cost and gradient for regularized logistic regression (**costFunctionReg.py** and **gradientFunctionReg**) correctly, you should be able to step through the next part of **ex2_reg.py** to learn the parameters θ using **fmin_tnc**.

4.4 Plotting the decision boundary

To help you visualize the model learned by this classifier, we have provided the function **plotDecisionBoundary.py** which plots the (non-linear) decision boundary that separates the positive and negative examples. In **plotDecisionBoundary.py**, 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 θ, the next step in **ex_reg.py** will plot a decision boundary similar to Figure 5.

4.5 Impact of λ

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

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 4). 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. However, 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).

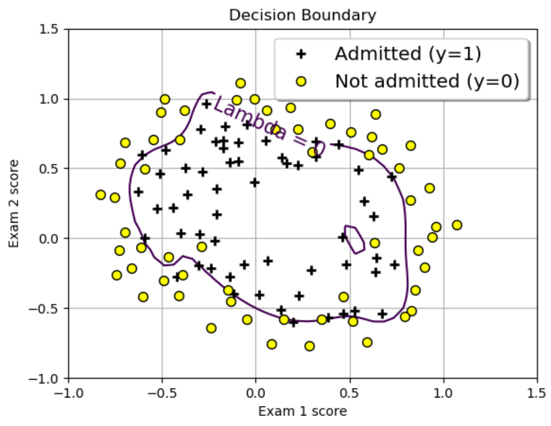

FIGURE 4 – No regularization (Overfitting) ($\lambda = 0$)

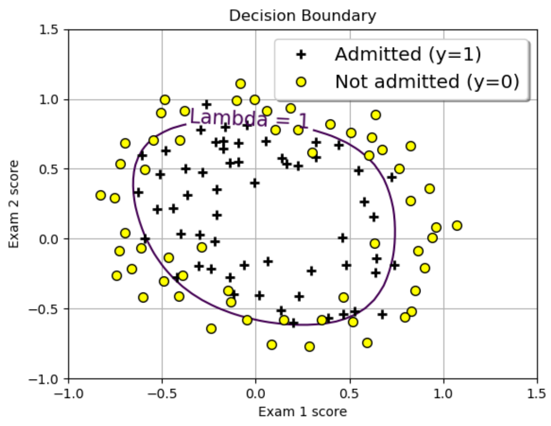

FIGURE 5 – Training data with decision boundary ($\lambda = 1$)

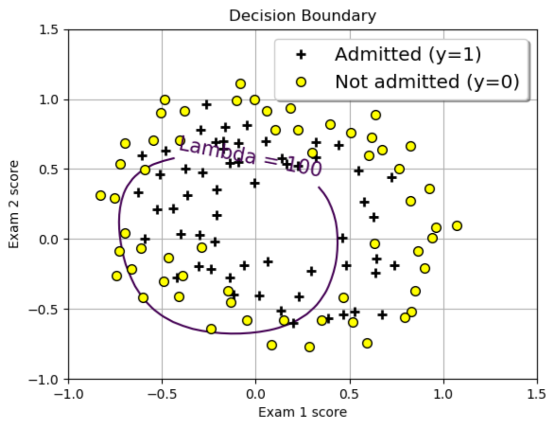

FIGURE 6 – Too much regularization (Underfitting) ($\lambda = 100$)

4.6 Some questions, you have to answer...

Le codage n'est qu'un prétexte pour comprendre les différents concepts du machine learning. **Ici aucune équation...Exprimez avec vos mots les concepts pour les comprendre.**

— Identifiez les différences d'approche entre le TP1 et le TP2 ? et les différences de résolution du problème ?
— A quoi sert le principe de régularisation ?
— Décrivez le problème du sur-apprentissage et comment on y répond dans ce TP.
— Quelle est l'influence de la valeur de λ ?