William Tang and Jason Dinh
Professor Jean-François Brière
Intro to Programming
25 April 2019

Preliminary Report

Term Project: Data Analysis/Machine Learning
Topic: Wine Classification

Introduction

Classification is a broad and important topic in machine learning. A program that can classify things into different categories autonomously has a variety of uses (spam filters, disease diagnostics, etc.) and provides a solution for sorting through large amounts of data. As an introductory project into machine learning and classification, we wanted to see how we could teach a program to recognize different types of wine based on their characteristics, such as the alcohol content, color intensity, and hue. After finding the appropriate data, we build a model from scratch to see if it would correctly predict between two sorts of wine at a time. Validation is done simply by comparing the prediction output of our model with the actual answer from the data.

Model

To classify things in supervised learning, we will be using **logistic regression**. Logistic regression takes the data as an input (in our case, characteristics of the wine) and outputs the appropriate category for that element (type of wine). To simplify things, we deal with **binary logistic regression**; our program will categorize between two different things at a time only.

So the goal in logistic regression is to put things of one type in one category, and things of another type in the other category. It uses a **sigmoid function** to return a probability value between 0 and 1. Based on some threshold that we decide, that value will be rounded to a 0 or a 1 - these represent the categories, and the program can now make the prediction.

The sigmoid function is what allows us to do this. It maps any real value into another value in the range (0,1). It is given by:

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

Now, where do the characteristics of the wine come in? We'll make up some equation that takes a linear combination of the characteristics, plus a constant. The equation that takes the linear sum of all the feature variables (that are multiplied by some coefficients beta) is given as follows :

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

Where the x variables are the numerical values of the characteristics of the wine. In order to get the "prediction", we'll run the value of this sum through the sigmoid function.

$$h(x) = \frac{1}{1+e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2)}}$$

This is called the **hypothesis function**. Now we have something that, given a set of input parameters (characteristics of wine), will spit out a number between 0 and 1 and make a prediction on the type. **Our goal will be to find the best $\beta_0, \beta_1, \beta_2, ...$ parameters that will give the most accurate predictions.**

To achieve this, we need another function in our model: something that will calculate the error between the prediction the program gives and the actual answer. In logistic regression, the mathematical equation of the error J is given as:

$$J = cost(h(x), y) = -\frac{1}{m}\sum_{i=1}^{m} y_i log(h(x_i)) + (1 - y_i) log(1 - h(x_i))$$

Again, the *x* variables represent the numerical values of the characteristics of the wine, and the *y* variables indicate the type of wine (turned into 0 or 1). *m* is the total number of data points. We call this the **cost function**. In order to find the best parameters that give the most accurate predictions, we will have to minimize the error between predictions and real answers, which in turn means minimizing the the function that represents the error - the cost function.

In the end, **logistic regression will be interested in finding the $\beta_0, \beta_1, \beta_2, ...$ parameters that should give the smallest value for the cost function**. We describe the method used to achieve this in the following section.

Method

We would now like to know how to find the $\beta_0, \beta_1, \beta_2, ...$ parameters that minimize the cost function, given a set of characteristics of the wine. If we can do that, the predictions of our program should be fairly accurate.

The numerical method we use here is called **gradient descent**. Recall that the gradient of a function is just the vector whose components are the partial derivatives of the function in question, with respect to the components (x_1, x_2, ..., x_n) of the function. We may compare gradient descent method with Euler's method: it calculated the slope, and *then took a step in the direction of that slope* to estimate the actual curve. The same idea applies here. If we have the slope (the partial derivative), then we can to take a step (the size of which we define) in the direction of the slope - as we saw in Euler's method, small steps ensure that we follow that actual curve. Also, in our case we are always planning on going downwards. The goal is to take steps until we reach the bottom of the curve - the bottom of our cost function.

Now, the slope changes as we compute a new partial derivative at the updated point. The derivative at a minimum should be zero, which means the difference between the updated point and the previous one should be zero at that point. **This means that if we are continuously updating our point (a,b), at some point it will stop decreasing.** In other words, we found the minimum. All we have to do is start at a random point, and go down from there. For a convex function, which is what we are dealing with, it does not matter where you start since there will be only one minimum. For a non-convex function, it becomes more problematic…. As such, in our project we made the assumption that the function to minimize is smooth and has only one global minimum (which should be the case, either way).

The gradient descent method is as follows:

$$\beta_0 = \beta_0 - \alpha\frac{\delta J}{\delta\beta_0}$$

$$\beta_1 = \ldots$$

$$\ldots$$

As seen above, each coefficient beta is brought into a continuous loop. We update all of them at the same time. *alpha* is the **learning rate*;*** it's how big of step you want to take each time. Finally, for any $\beta_j x_j$ parameter in the hypothesis function

$$h(x) \ = \ \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}}$$

the partial derivative of **the cost function** with respect to β_j will be given by

$$\frac{\delta J}{\delta\beta_j} = \frac{1}{m}\sum_{i=1}^{m}(h(x_i) \ - \ y_i)x_{j\ i}$$

Afterwords, the difference between the betas are calculated. If this is below the tolerance level, the loop is terminated and the results are displayed. Another method is just to iterate for a set number of times; if we know that the cost function does not change much after a certain point, this way could be faster.

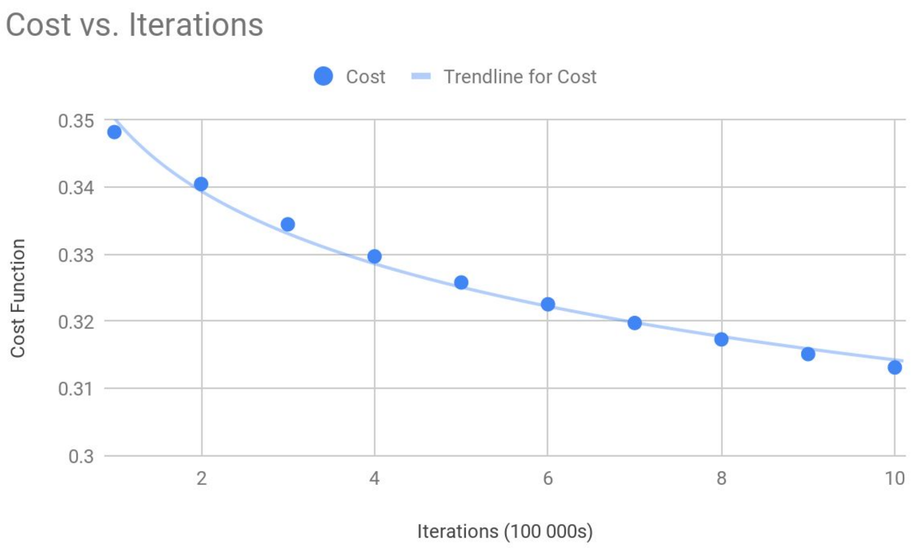
One thing that was also included in our model was a normalization of the data. This is done so that our logistic regression will work smoothly (it also converges faster usually). We use a **standardization to z-scores** for our dataset.

Test Case Presentation

We built a working model in java to test this logistic regression. It was done without any extra libraries we could've used, so it's relatively slow.

We obtained a dataset with three types of wine - that is, cultivated from three different regions in Italy - that had 13 different numerical characteristics, ranging from alcohol level to hue. The total number of data points was 178. Our project deals with a binary logistic regression, so in our test case, we coded a program that would differentiate between wine "type 1" and wine "type 2", denoted as 0 and 1 respectively. Each time the program was run, the dataset was put into arrays, and then randomly separated into training and test sets (80/20 split, standardized to z-scores). We then initialized the beta array[1] and performed a gradient descent. The alpha parameter was at 0.00001 and we ran it for 1 million iterations. After the optimal **weights** (betas) were calculated, we set the prediction threshold at 0.5: any prediction above this value would round to a 1, and anything below 0.

[1] You could start at any small value for the betas, but in later tests we found a specific set of points that would allow the program to converge faster and produce better results.



This is an example run of our program (Training set accuracy 88%, Test set 84%) that tests wine 1 vs. wine 2; the cost function seems to consistently decrease as the beta coefficients are updated. The cost function does not seem to decrease significantly past 1 million iterations (and in some cases even increased). What this graph tells us is that the error between the predictions the program will give and the actual answers is being reduced, which is a good thing. Notice that the cost function does not decrease by a large amount; at the end of the loop it had decreased to about 0.31 from 0.35. We are still working on this issue. In any case, this did not seem to affect the results: if you had initialized the betas and instantly tried to make it predict the wine type, its performance would be abysmal (worse than just guessing 50/50), whereas the trained model was much better.

Which brings us to the next part, validation. After our prototype was built, we had to test it and see if it correctly categorizes the wines. This was as simple as comparing the prediction of the trained model with the actual answers, one by one. At the end, we divided the total number of good answers by the total number of tries. The results are promising so far. From our repeated tests, the accuracy of this test case ranges anywhere from 80 to 90 percent on the training set. More importantly, similar results are observed on the test set - about **80 percent accuracy on average**.

A very small alpha leads to slow convergence. **One run of this test program took about 3-4 minutes**, which is quite long. However, we noted that if we changed the alpha to higher numbers, results could become unpredictable. At times, the cost function would decrease at a very large rate and the model would get an accuracy in the high 90s; in other instances, it would immediately diverge and the resulting accuracy would be in the 40s. Even with small alpha, the cost function does not always decrease. Further testing is needed.

In summary, we managed to implement a working logistic regression model in java, but there is still some tweaking to do and issues to fix. We should also look into other ways of evaluating the performance of our logistic regression, such as the effectiveness of each feature variable.

References

D., Dua and C., Graff. “Wine Data Set.” *UCI Machine Learning Repository*, Irvine, University of California, School of Information and Computer Science, http://archive.ics.uci.edu/ml

Molnar, Christoph. “4.2 Logistic Regression.” 12 April 2019, https://christophm.github.io/interpretable-ml-book/logistic.html

“The Cost Function in Logistic Regression.” *Internal Pointers*, 31 March 2018, https://www.internalpointers.com/post/cost-function-logistic-regression