Batch Gradient Descent, Stochastic Gradient Descent and Maximum Likelihood Estimation using Python

Abdulellah Abualshour
King Abdullah University of Science and Technology
Visual Computing Center
abdulellah.abualshour@kaust.edu.sa

Summary

Regression is an interesting technique of estimating the values among variables. In this experiment, I implement and test three algorithms of regression (batch gradient descent, stochastic gradient descent, and maximum likelihood estimation) using the gaussian function as a basis function to fit random noisy data I generate that represent the cosine function (plus noise).

1. Introduction

The experiment is divided into three tasks, each task dedicated to an algorithm for regression.

1.1. Language

I chose to use python is a programming language. However, I did not use any existing libraries for estimating or fitting regression models. All the math was done using the **numpy** library only. The **sklearn** library was used to split the data into a training and test set.

1.2. Code structure

The code submitted consists of three methods, each method corresponding to an algorithm that I implemented: BGD(), SGD() and MLE(). To test each of the functions, call them in the main conditional at the bottom of the code file.

Please note that a different random dataset was generated and used for each of batch gradient descent, stochastic gradient descent and maximum likelihood estimation algorithms.

2. Task One: Batch Gradient Descent

Batch gradient descent is an algorithm for optimization that allows us to find the minimum/maximum of a function by summing up over all the samples we have and update our parameters (weights) [1]:

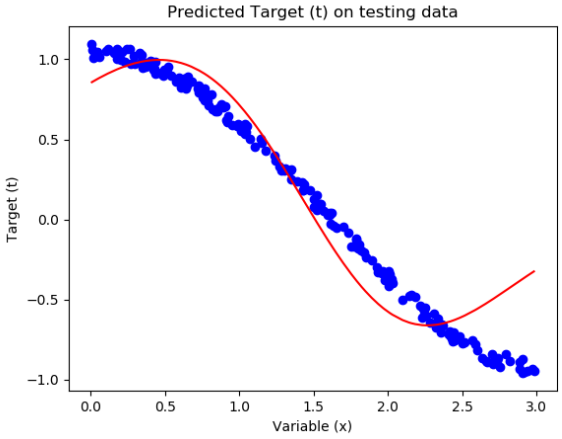
$$\mathbf{w} = \mathbf{w} - \alpha \nabla_w J(\mathbf{w})$$

The weights here are represented as a vector **w**. The learning rate, which how big we want our jump to be in our descent is represented by α, and $J(w)$ refers to the cost function (error function) that we want to minimize. We want the gradient of that with respect to *w*. For the sake of completion, the cost function is the following:

$$J(w) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, \mathbf{w}) - t\}^2$$

In the cost function above, we have a total of *N* samples, and we square the difference between our prediction $y(x_n,\mathbf{w})$ and our real value *t*.

We use the gradient of this function in our gradient descent algorithms, thus obtaining and learning the weights **w**. After applying batch gradient descent with the gaussian function as the basis on a synthetic dataset that was generated using the cosine algorithm (with noise added), I have obtained the following graph:

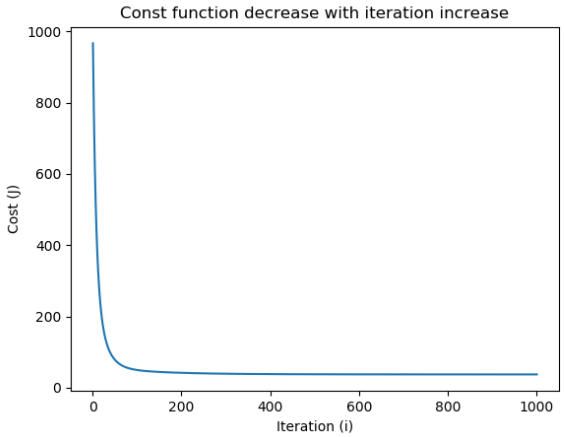


This graph represents the fitted line that was trained on 80% of the data generated (800 points) over 1000 iterations with a learning rate of 0.1 in order to learn the weight vector, and the weights learned were used to fit this line to the test set that represents 20% of the data and shown on the graph. The weights obtained are as follows:

w_0	**0.5600160878291915**
w_1	**0.8387175360173834**
w_2	**-0.8945401969027631**

A safe and true assumption to make during training is that when the iteration number increases, the cost (error) function decreases. This is intuitive due to the fact that we are learning and updating the weights at each iteration,

getting more accurate and closer to the true target value. The following graph shows this:

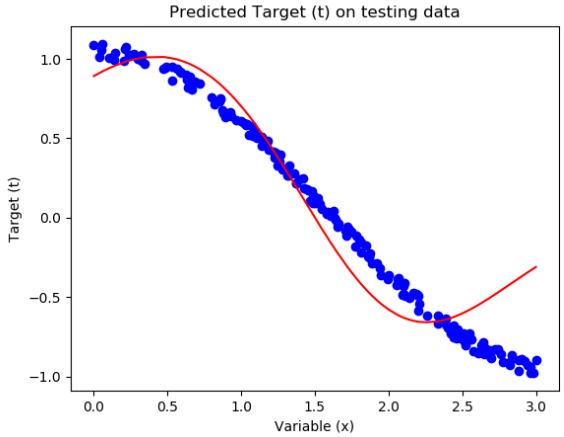


We notice that the graph is smooth, since we sum over all the samples every time. This will be important for discussion when compared to stochastic gradient descent. Calculating the Root-Mean-Square error (RMS) yields the following value:

$$\boldsymbol{RMS} = 0.07617236735307292$$

3. Task Two: Stochastic Gradient Descent

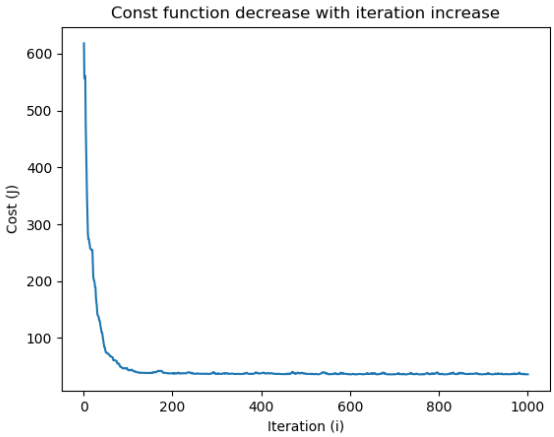
The only difference between batch gradient descent and stochastic gradient descent is that, in stochastic gradient descent, we do not sum over all the points in each iteration. Instead, we randomly choose a point to be used at each iteration [2] when we want to make a prediction and update the weights. Applying SGD yields the following nice graph:



We observe a very similar graph, since we're using the same gaussian basis function. The weights obtained are:

w_0	**0.6136568593629471**
w_1	**0.8022259994534873**
w_2	**-0.8846340759095672**

One interesting thing to observe has how the cost decrease figure is plotted:



We can clearly see how the line is jittered. This is due to randomness in our choice of points at each iteration. Sometimes we choose a bad point randomly, and therefore we do not get a better prediction. But over time, we get closer and closer to the target value regardless. Calculating the Root-Mean-Square error (RMS) yields the following value:

$$\boldsymbol{RMS} = 0.08903546008888479$$

4. Task Three: Maximum Likelihood Estimation

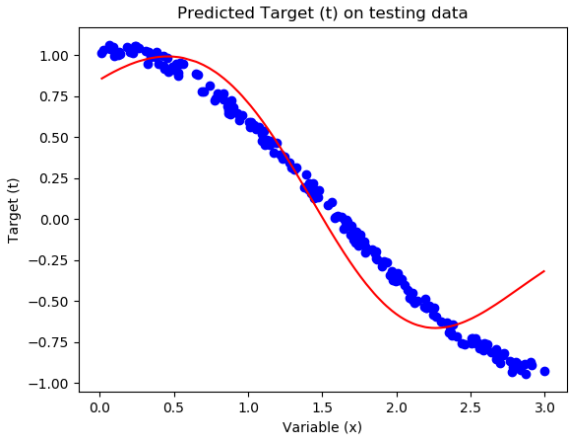
In maximum likelihood estimation, we have a different approach of calculating and learning the weights that involves more linear algebra. We need construct the following matrix [2]:

$$\phi = \begin{bmatrix} p_0(x_1) & \cdots & p_n(x_1) \\ \vdots & \ddots & \vdots \\ p_0(x_n) & \cdots & p_n(x_n) \end{bmatrix}$$

This matrix corresponds to the basis functions applied to all the points in the data. Each column corresponds to on weight value in the vector **w**, and each row corresponds to a point in the data. Once all the basis functions represented by *p*s are calculated, we take this matrix and plug it into the following equation to get our final values of **w**:

$$\mathbf{w_{ML}} = (\mathbf{\Phi}^T \mathbf{\Phi})^{-1} \mathbf{\Phi}^T \mathbf{t}$$

Implementing this yields the following plot whenfitted over the test set:



We can se a nice fit to the graph. The obtained coefficients are as follows:

w_0	**0.5582245956473079**
w_1	**0.8345758279127222**
w_2	**-0.8969190415053565**

Calculating the Root-Mean-Square error (RMS) yields the following value:

$$\boldsymbol{RMS} = 0.07928460601014468$$

5. Conclusion and Comparison

We can see that all the three algorithms yield similar results using the gaussian basis function. The only difference that I have noticed in my implementation is that MLE finishes faster than both SGD and BGD, and SGD finishes faster than BGD due to the number of calculations needed for the algorithms to terminate. In addition, the numpy library in python is optimized for linear algebra and this can contribute to MLE good performance compared to the other two algorithms.

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. 2007

[2] Zhang, Xiangliang. "Linear Regression." *Google Docs*, Google, 2019, https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxrYXVzdDIyOW1hY2hpbmVsZWFybmluZ3xneDozMjBmOTYwNjQ2MWI1YTIw.