Cubic Regression with Stochastic, Batch, and Mini-Batch Gradient Descent

Hong Yuan Tan (9384652)

University of California Santa Barbara

Abstract

In this paper, we attempt to find the best cubic polynomial to fit a given noisy data set shaped like a cubic. This is achieved through the use of gradient descent to minimize the mean squared error of a random starting model. Stochastic, mini-batch, and batch gradient descent are used and their efficiencies are compared through runtime and accuracy.

Introduction

One use of polynomial regression, or in this case - cubic regression, is providing answers for the unknown with great accuracy. For example, polynomial regression is used in death rate prediction for natural disasters. We can give a program a number of known variables about a future natural disaster and it will return an accurately estimated death rate. This allows governments and first-aid to prepare according to reduce the number of casualties. This paper address the ways to properly build such a program. More specifically, a cubic function that approximates a given data set shaped like a cubic. To begin, we have to build our own data set, though in the real world, we would be given such data set. This is done through the use of randomly generated coefficients for a cubic polynomial, a function to evaluate it, and the use of numpy.random.normal to generate noise. The inputs are also randomly generated and we can choose how many. To predict the best fitting cubic polynomial, we need one to begin, so a randomly generated cubic polynomial is chosen. The cost function or error of this polynomial is evaluated through the mean squared error method, which takes the average of the sum of the squared of our predicted value subtracted by its real value. The goal is to make this error as small as possible, and this is done through gradient descent. We take the derivative of the cost function with respect to each coefficient. Then the gradient tells us the direction of steepest ascent, i.e., direction of fastest increase in error for the coefficients. We want the direction of fastest decrease in error, so the negative of the gradient is used. Now that we have the direction, we must decide how fast we want to move in that direction, or the learning rate. We multiply these two values and add it to the current corresponding coefficients to attain our new coefficients which will give a lower error score. Many iterations of this, will give a cubic polynomial with very low error. Garvey's article was instrumental in the design of the program and the understanding of gradient descent.[2]

Mean Squared Error

The reason for using mean squared error (MSE)

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \overline{y_i})^2$$

as the cost function is that it gives a good representation of the distance between the predicted values and the real values from the data set. This is because MSE takes the square of the actual distance between the two values. The square is taken to ensure no negative values. This way, error is not lowered because a predicted value is really far above the cubic regression line. We want the distance between the two points to be as low as possible, which means that the mean squared error should be as low as possible. The square ensures that a low mean squared error implies a low distance between the predicted and real values. Our program stops when it reaches a certain number of iterations or the desired error.

Gradient Descent

Gradient descent is used to find the coefficients which best minimize the mean squared error. As stated above, the negative of the derivative of MSE with respect each coefficient times the learning rate is added to the current coefficient.

$$a_{\text{new}} = a - \left[\frac{2}{n}\sum_{i=1}^{n} x_i^3(y_i - (ax_i^3 + bx_i^2 + cx_i + d))\right] \times \text{learning rate.}$$

$$b_{\text{new}} = b - \left[\frac{2}{n}\sum_{i=1}^{n} x_i^2(y_i - (ax_i^3 + bx_i^2 + cx_i + d))\right] \times \text{learning rate.}$$

$$c_{\text{new}} = c - \left[\frac{2}{n}\sum_{i=1}^{n} x_i(y_i - (ax_i^3 + bx_i^2 + cx_i + d))\right] \times \text{learning rate.}$$

$$d_{\text{new}} = d - \left[\frac{2}{n}\sum_{i=1}^{n} (y_i - (ax_i^3 + bx_i^2 + cx_i + d))\right] \times \text{learning rate.}$$

The new cubic polynomial with these new coefficients can be thought of as taking a step towards the best fitting cubic. The size of the step is determined by the learning rate. Too large of a step will cause the program to step over the best fitting cubic, and too small of a step will make the program take longer than necessary. With large data sets, the run time will be exponentially longer, so having a good learning rate is very important. In our case, the best learning rate was found through trial and error.

How do we know that using gradient descent on mean squared error is valid? In other words, how do we know that the best fitting cubic generated from gradient descent will have the lowest mean squared error? Breaking down the MSE into its polynomials shows that the MSE polynomial is degree 2 with respect to each coefficient.

$$\begin{aligned}
\text{MSE} &= \frac{1}{n}\sum_{i=1}^{n}(y_i - \overline{y_i})^2 = \frac{1}{n}\sum_{i=1}^{n}\left[(a_1x_i^3 + b_1x_i^2 + c_1x_i + d_1) - (a_2x_i^3 + b_2x_i^2 + c_2x_i + d_2)\right]^2 \\
&= \frac{1}{n}\sum_{i=1}^{n}(ax_i^3 + bx_i^2 + cx_i + d)^2 \\
&= \frac{1}{n}\sum_{i=1}^{n}(a^2x_i^6 + \cdots + b^2x_i^4 + \cdots + c^2x_i^2 + \cdots + d^2).
\end{aligned}$$

The summation of degree 2 polynomials is still degree 2. This means that MSE must have local minimum(s) or/and local maximum(s) with respect to each coefficient. This is because the the derivative of a degree 2 polynomial is a polynomial of degree 1, and such a polynomial will always cross the x-axis once. Since we know that the least squares method is a general convex optimization problem[1, (1.3)], i.e.,

$$f(a,b,c,d) = \sum_{i=1}^{n}(y_i - (ax_i^3 + bx_i^2 + cx_i + d))^2 = \sum_{i=1}^{n}(y_i - \overline{y_i})^2$$

is convex. Then it follows that the mean squared error function is convex, too. Explicitly, we have that αf is convex for any $\alpha \geq 0$ and $\frac{1}{n} \geq 0$.[1, (3.2.1)] Then we know that MSE restricted to each coefficient, $f|_a : \mathbb{R} \to \mathbb{R}$, is also convex.[3, Theorem 3] So $f|_a$ is a quadratic function which is convex. Using the fact that

A twice differentiable function of one variable is convex on an interval if and only if its second derivative is non-negative in that interval,[4]

$$\frac{\partial^2 f|_a}{\partial a^2} = 2\sum_{i=1}^{n} x_i^6, \ \frac{\partial^2 f|_b}{\partial b^2} = 2\sum_{i=1}^{n} x_i^4, \ \frac{\partial^2 f|_c}{\partial c^2} = 2\sum_{i=1}^{n} x_i^2, \ \frac{\partial^2 f|_d}{\partial d^2} = 2n.$$

we see that $f|_a$ has a global minimum. This is because $f|_a'$ is increasing, so $f|_a$ must have a local minimum. Since $\mathbb{R}$ is a convex set, we can use

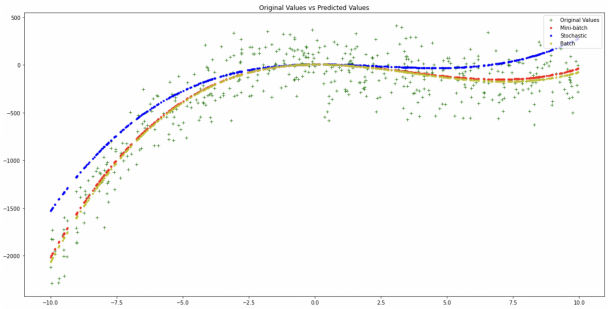
Proposition 1. Let X be a convex set. If f is convex, then any local minimum of f in X is also a global minimum,[5]

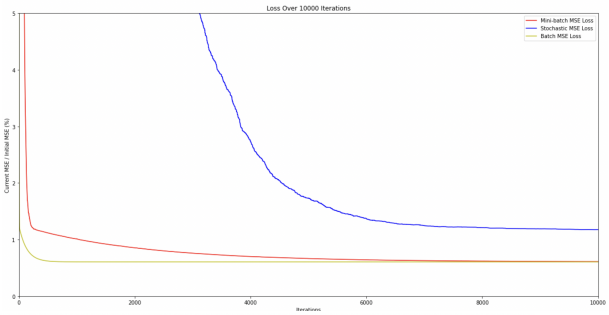
to show that this local minimum is a global minimum. It is also the case for $f|_b, f|_c, f|_d$. This means that gradient descent ensures that each coefficient goes to the respective coefficient that gives the lowest MSE. In other words, there is a global minimum for $\frac{1}{n}f(a,b,c,d)$ and gradient descent goes to this minimum with a sufficient number of iterations.

Results

We can see that gradient descent gives an accurate model of the data set. With more iterations, we can get within any desired error. Our best fitting cubic polynomial is in the middle of the data set. We see that stochastic gradient descent is the fastest to compute but least accurate, whereas batch gradient descent is the most accurate but has the longest runtime. Mini-batch is the best of both worlds with a runtime close to stochastic and an accuracy score close to batch. The most surprising thing is that the MSE does not converge to zero. This makes sense because there are too many data points, so no matter how well the graph fits, there will always be some error with some data point. This is why we measure the accuracy of each gradient descent as a percentage of the final MSE divided by the initial MSE. Even with few iterations of gradient descent, the error percentage becomes very low. For batch gradient descent, we see that it does not take more than a few hundred iterations to less than 1%, but it does not get any lower even with more iterations. Mini-batch converges to a low percentage with slightly more iterations than batch and Stochastic takes the most iterations. The program caps the number of iteration to $10,000$, but we can see

from the loss graph that this is in fact the case.






```
Starting Coefficients: (-6, -6, 2, 5)
Final Mini-batch Coefficients: (0.9675673370496258, -10.290046864059113, 2.059686092916862, 4.926654835059193)
Final Stochastic Coefficients: (0.8837992698579624, -6.281771424704408, 2.0950259929932145, 4.995028127977811)
Final Batch Coefficients: (0.9778595628424818, -10.728201853827642, 1.5664997287349178, 4.9028902748105185)
Real Coefficients: [1, -10, -2, -7]

Mini-batch MSE Loss: 40134.504507154394 | Percentage of Initial MSE Loss: 0.6112%
Stochastic MSE Loss: 77240.72322918671 | Percentage of Initial MSE Loss: 1.17628%
Batch MSE Loss: 39747.439994377964 | Percentage of Initial MSE Loss: 0.6053%

Mini-batch Run Time: 1.1390588283538818 sec
Stochastic Run Time: 0.8181002140045166 sec
Batch Run Time: 3.8109371662139893 sec

Mini-batch Iterations: 10000
Stochastic Iterations: 10000
Batch Iterations: 10000
```

In the results, we see that the coefficients bounce around the global minimum once they're close. In order to fix this, we need a learning rate that decreases as MSE gets closer to the minimum. Our code uses a fixed learning rate, so we are stuck bouncing. This is easily seen in batch gradient descent, where the MSE plateaus after a few hundred iterations.

The run time for stochastic is the fastest, as it should be, since it takes only one random sample of predicted outputs and its respective real value to calculate the new coefficients. Batch should take the longest as it takes all samples. This is reflective of the actual run times of each type of gradient descent. The surprising thing here is that mini-batch has such a low run time with a percentage error so close to batch's. This shows that mini-batch gradient descent is the most efficient method for our problem.

We see that our final model cubic polynomial fits well with the data. This implies that it is a sufficient function to use for predicting the y-value of any x-value. In the case of death rate prediction, our model would give a good estimation of the death rate when given the known variables of a future natural disaster.

REFERENCES

1. Boyd, Stephen. Vandenberghe, Lieven; *Convex Optimization*; Cambridge University Press 2004; pg. 7, 79;

2. Garvey, Mark; *Polynomial Regression - Gradient Descent from Scratch*; Towards Data Science; November 30, 2021;

3. Gu, Quanquan; *SYS 6003: Optimization - Lecture 7*; Fall 2016; pg. 3-4;

4. Kumar, D. Nagesh; *Convexity and Concavity of Functions of One and Two Variables*; pg. 2;

5. Thomas, Garrett; *Convexity*; pg. 3;