towards
data science

Follow 570K Followers

Linear Regression Using Gradient Descent in 10 Lines of Code

Joseph J. Bautista Sep 7, 2017 · 6 min read

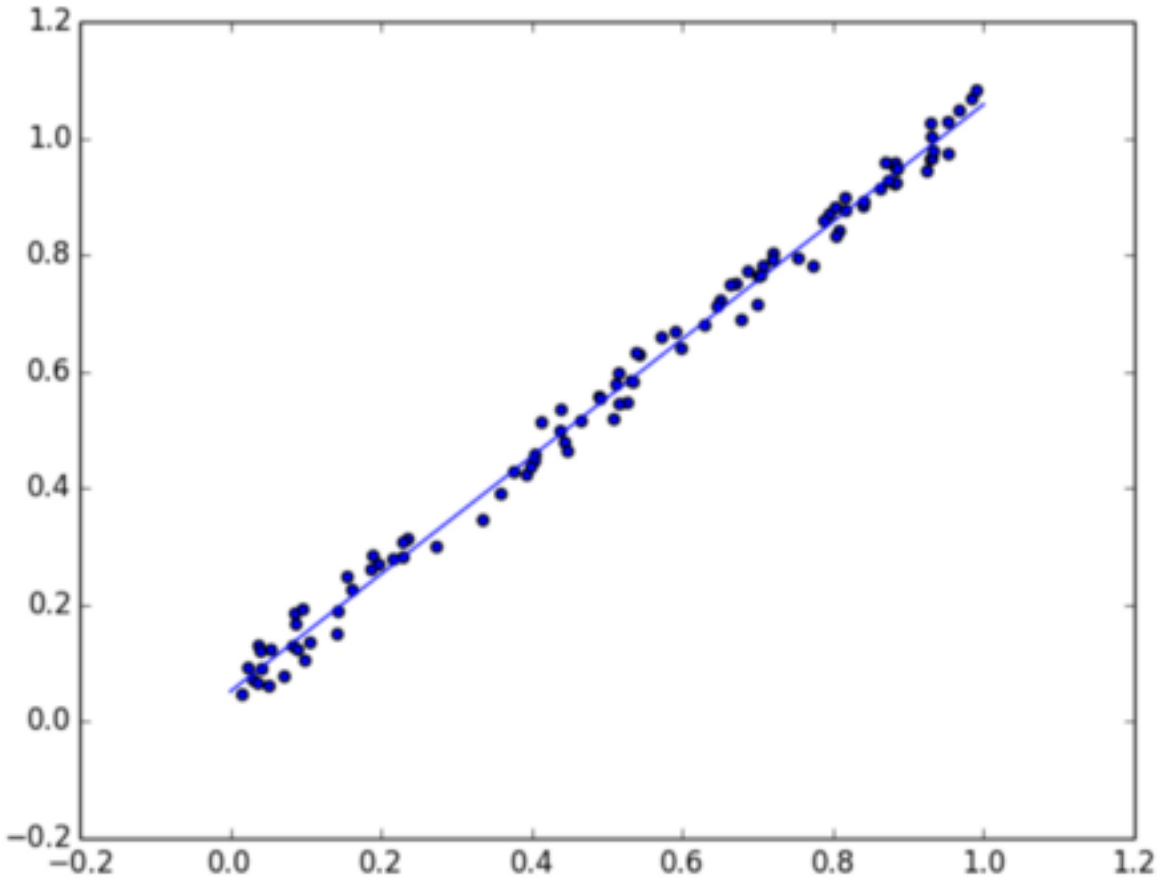
My goal is to eventually write out articles like this for other optimization techniques. Let's start with gradient descent. Note: This isn't a comprehensive guide as I skim through a lot of things.

In our journey to study machine learning and artificial intelligence, it is important to know the basics before going deeper. The basics are the building blocks needed in order to understand more complex forms of architectures. For instance, you wouldn't want to study ordinary differential equations without first understanding what, how, and why a derivative works. The same can be said with machine learning. An understanding of linear regression by using gradient descent as our optimization technique will help us understand more complex models in the future.

I remember one time explaining to a group of data scientists the random forest classification model I created for this company. I tried making an analogy by using logistic regression since I assumed that my fellow data scientists in the room would know about it. A whole lot of them said they weren't familiar with it. I got shocked since we were talking about a more advanced method here yet they didn't even know what a logistic regression model was. Don't be like that.

Linear Regression, Costs, and Gradient Descent

the bias (to move the line up and down the graph), x is the explanatory variable, and y is the output. We use linear regression if we think there's a linear relationship. For example, let's say that the x-axis below is *study_time* while the y_axis is *test_score*.



Source: https://i.imgur.com/oZ6CBpi.png

A straight line best describes this relationship because as a student studies more, his or her test scores increase. It wouldn't make sense to use an exponential, sinusoidal, or logarithmic function to model this relationship. Linear models offer us simplicity; linear relationships are easy to understand and interpret. Note that relationships in the real world aren't always linear, so instead we use more advanced methods like neural networks, which are universal function approximators. More on neural networks in the future.

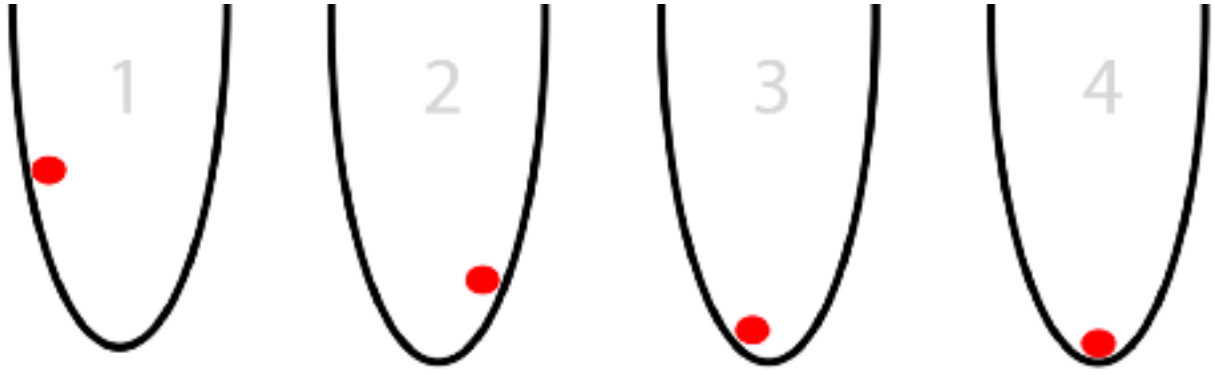
cost. A high cost value means it's expensive — our approximation is far from describing the real relationship. On the other hand, a low cost value means it's cheap — our approximation is close to describing the relationship.

For linear regressions we use a cost function known as the mean squared error or MSE. We take the squared value of our real data points minus the approximated values. Our approximated values can be calculated using the current *m* and *b* values we have: *y_approx = m_current*x + b_current*. After that, we add all those values up and divide them by the number of data points we have, effectively just taking the average. Now you see why it's called the mean squared error.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

Source: https://i.stack.imgur.com/19Cmk.gif

You're probably thinking right now, "that sounds splendid and all but how can we uncover *m* and *b*?? Brute force??" Brute force isn't helpful. A more efficient way is gradient descent. Imagine trying to find the lowest point blindfolded as can be seen below. What you would do is to check left and right and then feel which one brings you to a lower point. You do this every step of the way until checking left and right both brings you to a higher point.



The math behind it isn't as complicated as it looks. What we're doing here is applying partial derivatives with respect to both *m* and *b* to the cost function to point us to the lowest point. If you remember your math, a derivative of zero means you are at either a local minima or maxima. Which means that the closer we get to zero, **the better**. When we reach close to, if not, zero with our derivatives, we also inevitably get the lowest value for our cost function.

$$\frac{\partial}{\partial \mathrm{m}} = \frac{2}{N}\sum_{i=1}^{N} - x_i(y_i - (mx_i + b))$$

$$\frac{\partial}{\partial \mathrm{b}} = \frac{2}{N}\sum_{i=1}^{N} -(y_i - (mx_i + b))$$

Source: https://spin.atomicobject.com/wp-content/uploads/linear_regression_gradient1.png

The process of finding the optimal values for *m* and *b* is to then minimize our derivatives. Training a machine learning algorithm or a neural network really is just the process of minimizing the cost function.

Programming It

Here I'll be using Python to code our linear regression model. I use Python because it's my go-to language for doing data science. Furthermore, Python is great for both experts and beginners. The language was made for readability, so whether you've been programming for years or have just been programming for a day, it's still fairly easy to navigate through someone else's code.

```
    N = float(len(y))
    for i in range(epochs):
        y_current = (m_current * X) + b_current
        cost = sum([data**2 for data in (y-y_current)]) / N
        m_gradient = -(2/N) * sum(X * (y - y_current))
        b_gradient = -(2/N) * sum(y - y_current)
        m_current = m_current - (learning_rate * m_gradient)
        b_current = b_current - (learning_rate * b_gradient)
    return m_current, b_current, cost
```

X and *y* are our input parameters. On the other hand, *m_current* and *b_current* are our slope and bias terms respectively, both of which will be updated as we try to find the best numbers so that the equation we get best fits our data. Here *epochs* refer to the number of times we train our model to find the best slope and bias for our model to fit the data. Finally, *learning_rate* here refers to the speed of convergence, meaning how fast gradient descent finds the best parameters.

To further understand the *learning_rate*, let's go back to our example of finding the lowest point blindfolded. A big *learning_rate* would mean that the steps we take are too big and that you might miss the lowest point entirely. However, too small of a *learning_rate* means that we will take *a long time* to reach the bottom. Try to strike a balance between the two.

The important bit is the for-loop:

```
for i in range(epochs):
    y_current = (m_current * X) + b_current
    cost = sum([data**2 for data in (y-y_current)]) / N
    m_gradient = -(2/N) * sum(X * (y - y_current))
    b_gradient = -(2/N) * sum(y - y_current)
    m_current = m_current - (learning_rate * m_gradient)
    b_current = b_current - (learning_rate * b_gradient)
```

We iterate 1000 times because 1000 sounds good. Really depends on how much you want to iterate. Hyperparameter fine-tuning is an ongoing research right now, so you might want to check that out!

respective gradient value. The learning rate will determine the speed we will reach convergence. After 1000 iterations, we return *m_current*, *b_current*, and the *cost*.

Congratulations! That's the first step in your machine learning and artificial intelligence journey. Get an intuitive feel for how gradient descent works because this is actually used in more advanced models also. The goal here is to learn the basics, and you my friend have just taken the first step. Now off you go and learn more!

Machine Learning Data Science