Logistic Regression

Machine Learning Mini Course

**Linear Regression + Loss Function Review**

There was a lot covered in the previous class, so it would be helpful to briefly review it. Remember that we started off with the simplest kind of machine learning example: **linear regression**. Intuitively, we were finding the line of best fit for some given data points, but we were letting the machine figure it out. Our **model**, in this case, was simply two weights w and b, and to give a prediction on a value x, it would return y = wx + b. How do we find the optimal w and b?

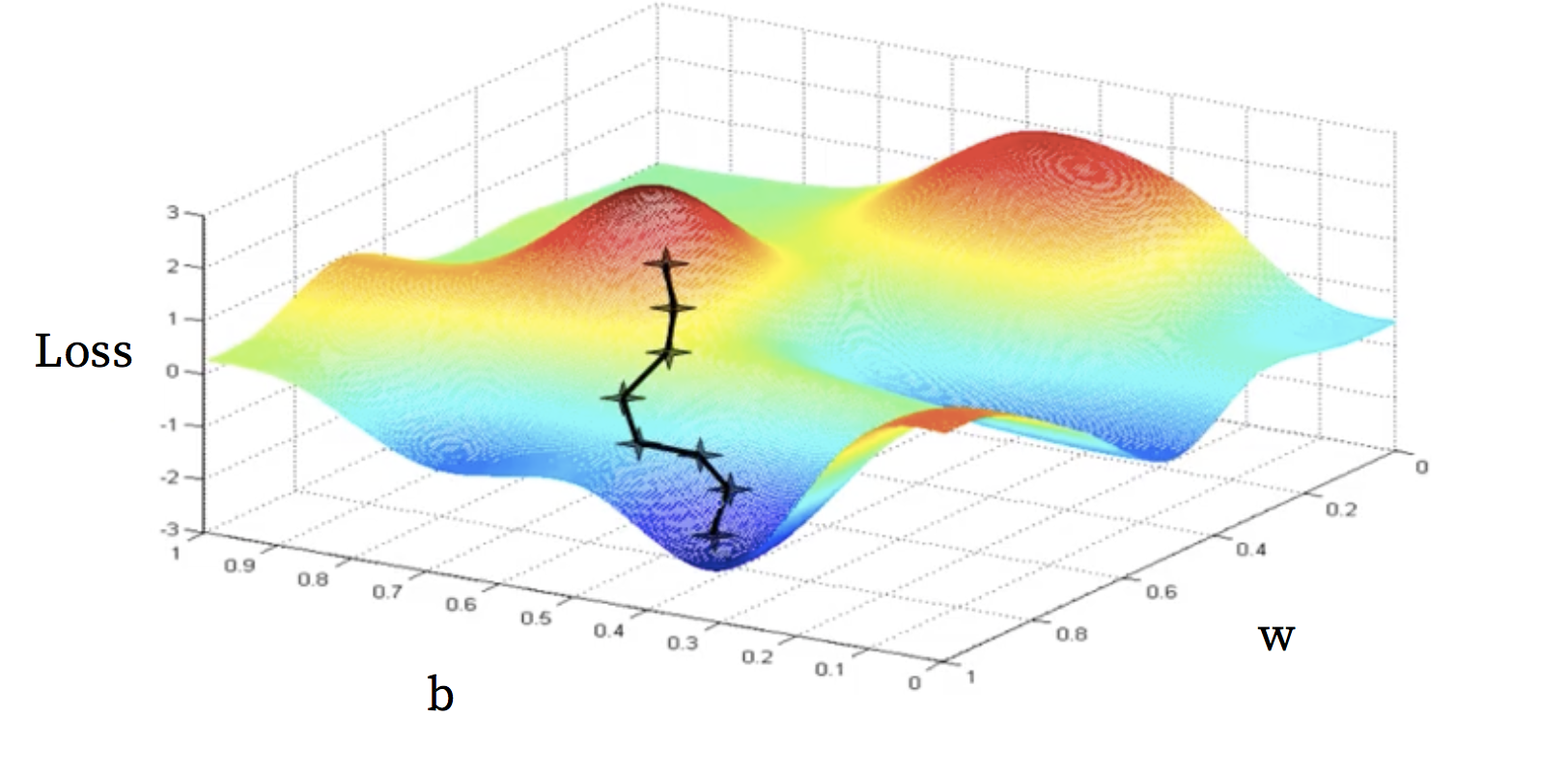
To do this, we had to define a notion of how bad our model was based on the data. This is where we introduced the **loss function**. Intuitively, you can think of it like a function that takes in a model and tells it how bad it is. Our weights are w and b, so we can let our loss function be loss(w, b). For each data point (x, y), our model is going to predict some y\_pred = wx + b, so to calculate loss, we can compare this prediction to the real value. Here are some common loss functions:

Mean Absolute Loss =

Mean Squared Error =

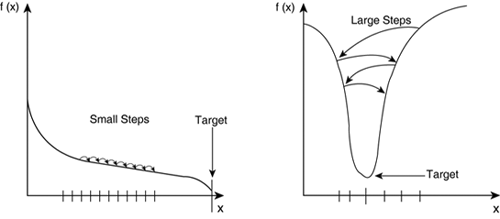
(The m here is the number of datapoints, so we’re taking the average error. This is where the “mean” in “mean squared error” comes from.)

Now that we have our loss function, we can use calculus (see last week’s optional worksheet) to optimize our model. Remember that you can think of it as going down a hill (see following picture).



**Learning Rate**

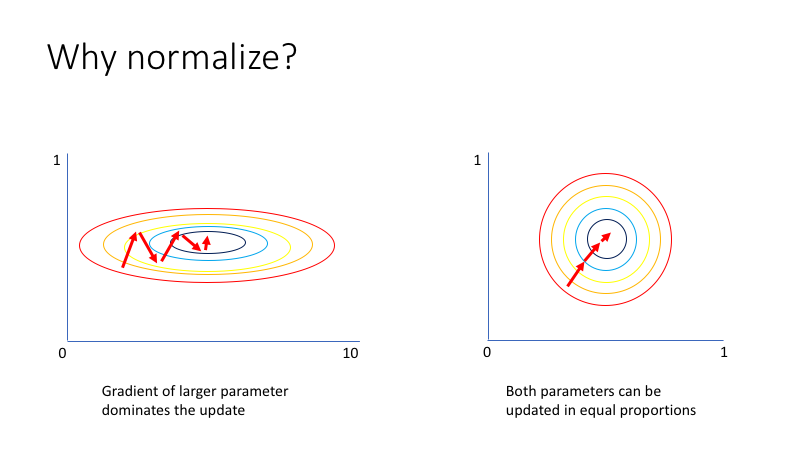
If you look at the picture closely, you’ll see that there are little +’s along the line. This is because when we are improving our model in a process called **gradient descent**, we are taking small steps towards minimizing loss. The process is more walking down a hill than rolling down it. How big are these steps then? you might ask. Well, that’s what we call **learning rate**. So far, we’ve let Keras automatically decide, but we’ll soon see how to set it ourselves.



To explain why learning rate matters, let’s go back to our hill analogy. If our learning rate is too small, then getting to the bottom of the hill is going to take a long time. If our learning rate is too big, this is like taking a big leap in the direction of the valley, but we might go to far and just end up on a higher hill. Choosing a good learning rate, therefore, will help speed up training while also making actual improvements. Unfortunately, it’s hard to tell what a good learning rate for a problem is, so you’ll have to try out a couple and plotting how well your model is doing over time to see which will give you the best results.

**Normalization**

Another way to speed up the learning process is called **normalization**. Before we explain what it is, let’s go back (yet again) to the hill analogy. Imagine we’re trying to predict housing prices and we have two features, number of bedrooms (x1) and size of house in ft2 (x2). Then our model is: y = w1x1 + w2x2 + b. But in our model’s eyes, both x1 and x2 should be weighted equally. The model will be a lot more biased to modifying w2 because that will have a bigger impact.



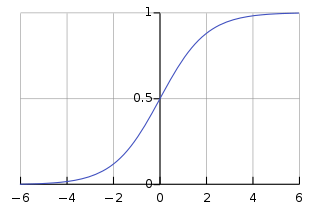
Here’s an image to visualize what I mean. Basically, in general, it’s best to keep everything in the 0 to 1 range (or -1 to 1). You don’t want variables on different orders of magnitudes.

**Regression vs Classification**

That’s enough on optimization methods for now. Onto a new type of machine learning problem! Regression takes in an input and returns a numerical value, like the quality of wine or the price of a house. **Classification**, on the other hand, takes an input and returns a “yes” or “no”, like whether this patient has cancer or whether this picture is a picture of a cat. It turns out that if you look at what’s happening under the hood, the two look very similar! In fact, **logistic regression**, the name of the method we will use for classification, has the word “regression” in it!

To see how the two are different, let’s see what happens if we try to use a linear regression model on a classification problem. Suppose we have pictures of cats and dogs and want our model to tell us whether a picture has a cat or a dog. We train our model, get some good weights, and then we feed it a data point to predict, and it gives us something like 2.35... huh. So this datapoint has a 235% chance of being a cat?

Preferably, we would want a model to give us something between 0 and 1, where 1 means the model is 100% certain that the picture is a cat and 0 means the model is certain that the picture is a dog. To turn our output into a number between 0 and 1, we will put our linear regression’s output through a **sigmoid function**.



Behold! The equation for this function is given by . In general there are many different functions which we can use. These functions are called “activation functions,” and we’ll be talking about them more in later with neural networks.

Logistic Regression then has the equation:

or

**Optimizing Logistic Regression**

So, what does our cost function look like now? Well, not too different actually! The loss function is still:

Mean Squared Error =

We just now have that ytrue is either 0 or 1, and ypred must be between 0 and 1. Specifically,

Mean Squared Error =

The more commonly used loss function for dealing with classification problems is called **cross entropy**. We’ll go over this in the codelab.

Cross Entropy Error =