Gradient Descent

January 23, 2020

1 Fitting Models with Gradient Descent

Gradient descent and its cousins are the core of many machine learning methods. Building a predictive model boils down to searching for some version of the model that best fits the data. But what do we mean by a "version" of the model and "fit"? And, once we make those concepts concrete, how do we find that best model? Let's dig in.

1.1 Motivation: Linear Regression

1.1.1 Data and setup

Let's say we observe n patients in a hospital, each of whom has measurements of their pre-treatment blood pressure, heart rate, etc. and also their corresponding blood pressure after taking a drug. For a new patient who walks in the door, given their current vital signs, can we predict their post-treatment blood pressure?

The data for this kind of problem is most often formulated as vectors and matrices. We usually say that the ith patient's pre-treatment values are $x_i = [x_{i1}, x_{i2} \dots x_{ip}]$, so x_{i1} is patient i's pre-treatment blood pressure, x_{i2} is their heart rate, etc. We also usually call the post-treatment blood pressure measurement for each patient y_i. So our goal is to predict y_i from x_i for a new patient who walks in the door.

Now we can stack up all the measurements for all the patients like this:

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix} = \begin{bmatrix} -3.2 & 28 & \cdots & 66 \\ 8 & 22 & \cdots & 6.8 \\ \vdots & & \ddots & \vdots \\ 0.54 & 10 & \cdots & 13 \end{bmatrix} \qquad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 3.3 \\ 21.2 \\ \vdots \\ -9.97 \end{bmatrix}$$

We used pre-treatment vitals and post-treatment blood pressure as the predictors (or features) and outcome (or target) in our example, but this is the way we usually think of the data, no matter what the predictors and outcome are.

1.1.2 Predicting

There are an infinite number of strategies we could use to try and guess what the relationship between y_i and x_i might be. We're going to presume that there is some relationship $y_i = f(x_i)$, and we want to figure out what f is. You can think of the "real" f as the function that the gods use to decide what a patient's blood pressure will be after treatment, given that the pre-treatment vitals were x_i.

Since we don't know what f is, we're going to try and guess. To make our job easier, we'll assume f is a linear function of the form $x_i\beta + \alpha$. Our problem now reduces to guessing what the right α and β are.

WARNING: Saying f is linear (or of some other form) when it's really not is called model misspecification. We never know for sure the form that f has, so in reality our models are almost always misspecified. If we only use the model for prediction, misspecification doesn't invalidate the approach, it just means that the predictions won't be as accurate as they otherwise could be. On the other hand, if the model is used for inference (i.e. to estimate the effect of a variable on the outcome, with p-values, confidence intervals, etc.), then misspecification often invalidates the result. Be aware of this distinction.

Let's play god and simulate some data where a linear relationship actually does hold between x_i and y_i (and let's use neat unicode symbols in our code!):

```
In [335]: import numpy as np

          def to_array():
              if type() in (float, int, np.float64):
                  return np.array([]) # handle the case where user passes in a single number
              if type() is list:
                  return np.array() # handle the case where user passes in a list
              return

          def simulate_linear(n, =1, =0, =1):
               = to_array()
              p = len() # number of predictors we want
              X = np.random.rand(n,p)
               = np.random.normal(scale=, size=n) # random noise
              y =  + X@ +
              return X, y
```

```
In [4]: X,y = simulate_linear(100, =1, =[1,-3,12])
```

Ok, now let's go back to being mere mortals. All we see are the data X and y. We don't know for sure that the relationship is linear, and much less what α and β were. But we're going to take a leap of faith and pretend it's linear. The worst that happens is our predictions won't be very good.

Now all we need to do is find some values $\beta = [\beta_1, \ldots \beta_p]^T$ and α so that $y_i \approx x_i\beta + \alpha$ for all observed y_i and x_i in the data. That is, we need to find the values that make the model best "fit" the data. Once we have those best values, we can feed the vitals of a new patient x_i into the formula $x_i\beta + \alpha$ to guess what their blood pressure y_i will be. We often call the estimate $\hat{y}_i = x_i\beta + \alpha$. This whole process is called linear regression, which many of you are likely familiar with.

We can set up a little `LinearModel` class in python that encapsulates what we've said so far.

```
In [5]: import numpy as np

        class LinearModel():
            def __init__(self, =None, =None):
                self. =  # these are the "parameters" of the model
                self. = to_array()
```

```
            def predict(self, X):
                return self. + X @ self. # returns y
```

And we can test it out:

```
In [6]: model = LinearModel(=1, =[1.1,-33,9]) # make a model with these coefficients
        print(f'y = {model.predict(X[0:5,:])}')
        print(f'y = {y[0:5]}')
```

```
y = [-13.16301189  -9.76725628 -10.93047664 -24.68828898   6.56525583]
y = [10.96199259  9.91667858  2.5389317  -0.79802221 13.06470167]
```

Exercise:

Make a scatterplot of the $\hat{y}_i$ values (x-axis) against the corresponding y_i values (y-axis) that are calculated in this code. You can read this plot as "when we predict $\hat{y}$, the real values are usually around y". What do you see in the plot? What would signify a "good" model? Change the values of α and β and see how the plot changes.

Obviously the predictions look very far off since we picked the values of α and β at random

The question is: how do we find β and α so that $y_i \approx x_i\beta + \alpha$?

1.1.3 Defining a Loss

The first thing to do is quantitatively define what we mean by "$\approx$". We need a rule that tells us how good our approximation $\hat{y}_i$ is if the real blood pressure is y_i. We call that rule a *loss function* (sometimes also *cost* or *objective*). The loss tells us how much we "lose" by approximating y_i with whatever value $\hat{y}_i$ is. Ultimately that depends on what's important in the problem at hand, but often we default to using the squared-error loss function: $(y_i - \hat{y}_i)^2$ The intuition is simple: all we're doing is subtracting the estimate from the real value and squaring the result to make positive and negative deviations equal in penalty. So estimates that are numerically far from the truth are penalized higher, regardless of which direction they are off.

To calculate the loss over the whole dataset (instead of just one individual i), we average the losses across each individual: $\frac{1}{n}\sum_i (y_i - \hat{y}_i)^2$. This is called *mean-squared error* (MSE) loss. In python:

```
In [7]: def mse(y, y):
            return np.mean(np.square(y-y))
```

Now we have a concrete notion of "$\approx$": the lower the loss when using the estimate $\hat{y}$, the better the approximation. So what we need to do is find the estimate that minimizes the loss.

If we plug in $\hat{y}_i = x_i\beta + \alpha$, what we're looking for are values of α and β that minimize the loss $\frac{1}{n}\sum_i (y_i - (x_i\beta + \alpha))^2$. So although we introduced the loss as a function of the truth y_i and the estimate $\hat{y}_i$, we can look at it instead as a function of the parameters α and β: $L(\alpha, \beta) = \frac{1}{n}\sum_i (y_i - (x_i\beta + \alpha))^2$. The parameters are the only thing we can actually control to change the loss since x_i and y_i are just constants that are determined by the observed data.

So how do we find the α and β that minimize the loss?