LINEAR REGRESSION

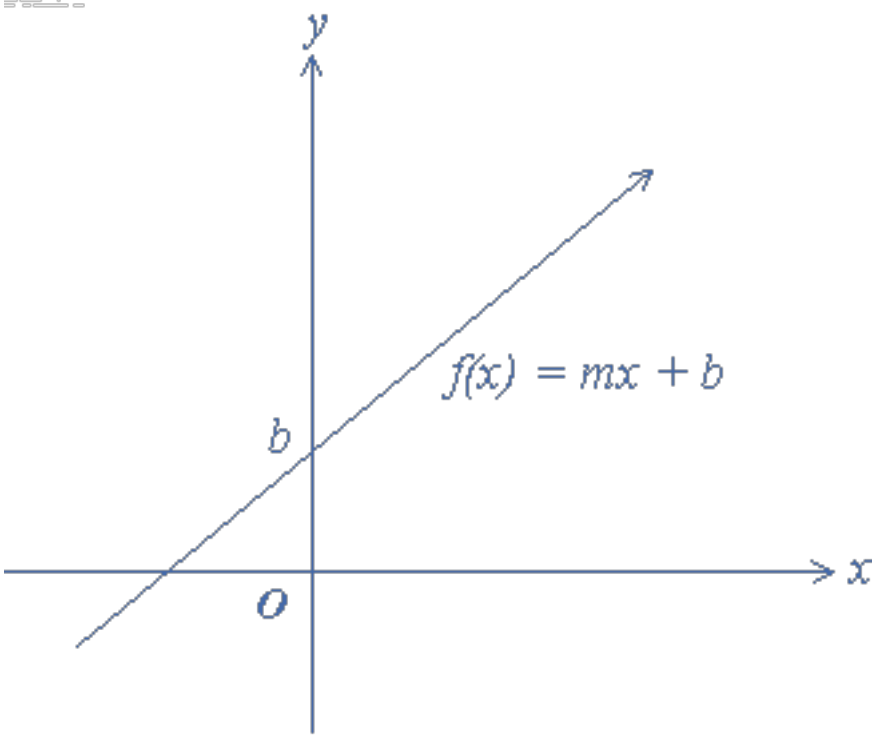
Linear regression is one of the simplest machine learning techniques you can use. It is often useful as a baseline relative to more powerful techniques. To start, we will look at a simple 1-D case. Like all regressions, we wish to map some input X to some input Y.ie.

Y = f(X)

With linear regression:

Y = aX + b
Or we can say:
h(X) = aX + b Where “h” is our “hypothesis”.

You may recall from your high school studies that this is just the equation for a straight line.



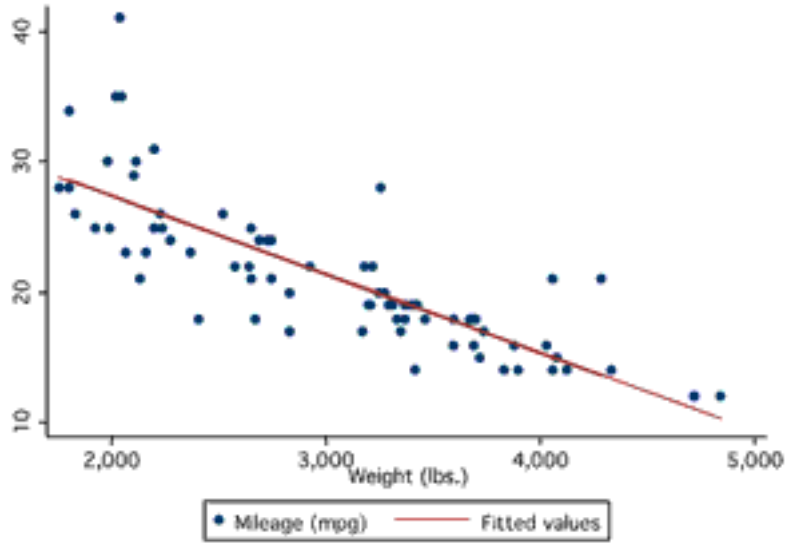
When X is 1-D, or when “Y has one explanatory variable”, we call this “simple linear regression”. When we use linear regression,

we are using it to model linear relationships, or what we think may be linear relationships.

As with all supervised machine learning problems, we are given labeled data points:

(X1, Y1), (X2, Y2), (X3, Y3), …, (Xn, Yn)

And we will try to *fit* the line (aX + b) as best we can to these data points.



This means we have to *optimize* the parameters “a” and “b”.
How do we do this?

We will define an *error function* and then find the “a” and “b” that will make the error as small as possible. You will see that many regression problems work this way. What is our error function? We could use the *difference* between the predicted Y and the actual Y like so:

$$\sum_{i=1}^{n} y_i - h(x_i)$$

$$h(x) = ax + b$$

But if we had equal amounts of errors where Y was bigger than the prediction, and where Y was smaller than the prediction, then the errors would cancel out, even though the *absolute difference* in errors is large. Typically in machine learning, the *squared error* is a good place to start.

$$J = \sum_{i=1}^{m} (y_i - h(x_i))^2$$

Now, whether or not the difference in the actual and predicted output is positive or negative, its contribution to the total error is still positive. We call this sum the "sum of squared errors". Recall that we want to minimize it. Recall from calculus that to minimize something, you want to take its derivative.

$$\frac{\partial J}{\partial a} = 0, \quad \frac{\partial J}{\partial b} = 0$$

Because there are two parameters, we have to take the derivatives both with respect to a and with respect to b, set them to 0, and solve for a and b. Luckily, because the error function is a quadratic it increases as (a,b) get further and further away from the minimum. As an exercise I will let you calculate the derivatives.

You will get 2 equations (the derivatives) and 2 unknowns (a, b). From high school math you should know how to solve this by rearranging the terms.

Note that these equations can be solved *analytically*. Meaning you can just plug and chug the values of your inputs and get the final value of a and b by blindly using a formula.

Note that this method is also called "ordinary least squares".

Measuring the error (R-squared)

To determine how well our model fits the data, we need a measure called the "R-square". Note that in classification problems, we can simply use the "classification rate", which is the number of correctly classified inputs divided by the total number of inputs. With the real-valued outputs we have in regression, this is not possible.

Here are the equations we use to predict the R-square.

$$SS_{total} = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

$$SS_{residual} = \sum_{i=1}^{n} (y_i - h(x_i))^2$$

$$R^2 = 1 - \frac{SS_{residual}}{SS_{total}}$$

$SS_{(residual)}$ is the sum of squared error between the actual and predicted output. This is the same as the error we were trying to minimize before! $SS_{(total)}$ is the sum of squared error between each sample output and the mean of all the sample outputs, i.e. What

the residual error would be if we just predicted the average output every time.

So the R-square then, is just *how much better* our model is compared to predicting the mean each time. If we just predicted the mean each time, the R-square would be 1-1=0. If our model is perfect, then the R-square would be 1-0=1.
Something to think about: If our model performs *worse* than predicting the mean each time, what would be the R-square value?

Limitations of Linear Regression

- It only models linear equations. You can model higher order polynomials (link to later post) but the model is still linear in its parameters.
- It is sensitive to outliers. Meaning if we have one data point very far away from all the others, it could "pull" the regression line in its direction, away from all the other data points, just to minimize the error.

Today we will continue our discussion of linear regression by extending the ideas from simple linear regression to multiple linear regression.

Recall that in simple linear regression, the input is 1-D. In multiple linear regression, the input is N-dimensional (any number of dimensions). The output is still just a scalar (1-D).
So now our input data looks like this:

(X1, Y1), (X2, Y2), …, (Xm, Ym)

Where X is a vector and Y is a scalar. But now instead of our hypothesis, h(), looking like this:

h(X) = aX + b
It looks like this:

$$h(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_2 x_2 + \ldots + \beta_n x_n$$

Where each subscripted x is a scalar.

$beta_0$ is also known as the “bias term”.

Another, more compact way of writing this is:

$$h(x) = \beta^T x$$

Where beta and x are vectors. When we transpose the first vector this is also called a “dot product” or “inner product”. In this representation, we introduce a dummy variable x0 = 1, so that beta and x both contain the same number of elements (n+1).
To solve for the beta vector, we do the same thing we did for simple linear regression: define an error function (we’ll use sum of squared error again), and take the derivative of J with respect to each parameter (beta0, beta1, …) and set them to 0 to solve for each beta.

$$J = \sum_{i=1}^{m} (y_i - h(x_i))^2, \quad \frac{\partial J}{\partial \beta_0} = 0, \quad \frac{\partial J}{\partial \beta_1} = 0, \quad \frac{\partial J}{\partial \beta_2} = 0, \quad \ldots$$

This is a lot more tedious than in the 1-D case, but I would suggest as an exercise attempting at least the 2-D case. As before, there is a “closed form” solution for beta:

$$\beta = (\sum_{i=1}^{m} x_i x_i^T)^{-1} (\sum_{i=1}^{m} x_i y_i)$$

Here, each (Xi, Yi) is a “sample” from the data. Notice that in the first term we transpose the second Xi. This is an “outer product” and the result is an (n+1) x (n+1) vector. The superscript -1 denotes a matrix inverse.

An even more compact form of this equation arises when we consider all the samples of X together in an m x (n+1) matrix, and all the samples of Y together in an m x 1 matrix:

$$\beta = (X^T X)^{-1} X^T y$$

As in the 1-D case, we use the R-square to measure how well the model fits the actual data (the formula is exactly the same).

Linear programming solves optimization problems whereby you have a linear combination of inputs **x**,

c(1)x(1) + c(2)x(2) + c(3)x(3) + … + c(D)x(D)
that you want to maximize or minimize, subject to constraints of the form:

a(1,1)x(1) + a(1,2)x(2) + … + a(1,D)x(D) <= b(1)
a(2,1)x(1) + a(2,2)x(2) + … + a(2,D)x(D) <= b(2)
…
where each A(i,j) is an entry of a matrix.

In compact form:

A**x** <= **b**

So if **x** is a vector of length D, A is a matrix of size N * D, where N is the number of constraints.

Note that “greater than or equal to” (>=) and “equality” (==) and “non-equality” (!=) can be converted into “less than or equal to” (<=) constraints.

Most linear programming packages allow constraints of any form, however.
2 popular algorithms for solving LP problems (that you don’t need to understand in order to understand this tutorial) are the “simplex method” and the “interior point” method. Using an LP library will hide these details from us.

Setting up the linear regression objective function

Suppose you are given a set of points on or near some line:

S = {(0,1), (1,2), (2,3), (3,4), (10,12), …}

And you would like to find a line of the form:

ax + by + c = 0

That best fits the given set of points.
I already showed how to solve this problem by minimizing the mean squared error here.

What if we want to minimize the absolute (rather than squared) error? Or in another sense, minimize the *maximum absolute error*, i.e.
min (max |ax(i) + by(i) + c|)

For all x(i), y(i) in S.

Intuitively, you can think about it this way:

ax + by + c = 0 is the line.
If, given a point (x, y), we get:
ax + by + c > 0
Or
ax + by + c < 0

We are not on the line. Thus, we want all of our (x, y) pairs to make |ax + by + c| as close to 0 as possible, thus, we want to minimize it.

The next question is, how do we turn this into "standard form" for an LP solver?:

maximize: $c^T x$
subject to: $Ax \leq b$

The Solution

Create a variable, "z", such that:
z = max |ax(i) + by(i) + c| for all x(i),y(i) in S

Then our constraints become:

|ax(i) + by(i) + c| <= z
Or:
ax(i) + by(i) + c <= z and ax(i) + by(i) + c>= -z
The objective function is just "z".

Note that our variables in **x** are NOT (x, y, z) but (a, b, c, z). x and y simply represent the different data points. The variables are the parameters of the line.

Practical considerations

Note that for a line there are really two parameters, not 3, since a line can be represented by:

$y = mx + p$

So if your best fit line is y = 2x, you might get strange answers like:

a = 2000, b = -1000, c = 0
a = 20000, b = -10000, c = 0

Because all of these are valid solutions that lead to the same line. You can thus write your constraints as:

$|mx - y + p| <= z$

And solve for the variables (m, p, z).