**Basic Linear Regressions in Python**

[programming](http://jmduke.com/tags/programming)

Tue, Mar 11, 2014

Linear regressions are a great tool for any level of data exploration: chances are, if you’re looking to investigate the relationship between two variables, somewhere along the line you’re going to want to conjure a regression. So how do you accomplish that in Python?

First, let’s grab some test data from [John Burkardt](http://people.sc.fsu.edu/~jburkardt/) at FSU: specifically, some [toy housing data](http://people.sc.fsu.edu/~jburkardt/datasets/regression/x27.txt) which contains – amongst other things – the area of the site in thousands of square feet and the final selling price. We’ll investigate a pretty simple hypothesis: as the area of the site increases, the selling price increases as well.

Creating the regression itself is pretty simple if you go the route of NumPy:

import pandas as pd

import numpy as np

COLUMN\_SEPARATOR = ' '

housing\_data = pd.DataFrame.from\_csv('housing.csv', sep=COLUMN\_SEPARATOR, header=None)

AREA\_INDEX = 4

SELLING\_PRICE\_INDEX = 13

x = housing\_data[AREA\_INDEX]

y = housing\_data[SELLING\_PRICE\_INDEX]

regression = np.polyfit(x, y, 1)

regression prints as [ 3.87739108 13.10587299], which is a list of the coefficients in descending power. That is, this is the same as the following equation:

y = 3.87739108 \* x + 13.10587299.

So, our hypothesis turns out to be sane! For every thousand square feet in the housing site, our selling price increases by approximately $380 dollars.

And, since we’re feeling fancy, let’s graph this lil’ guy! I decided to go with [Bokeh](http://jmduke.com/posts/basic-linear-regressions-in-python/bokeh.pydata.org), which is very much still a work in progress on binding matplotlib-esque interfaces onto D3 – it doesn’t have much embedding functionality, but it’s easy to get off the ground and looks pretty dang neat.

from bokeh.plotting import \*

# We need to generate actual values for the regression line.

r\_x, r\_y = zip(\*((i, i\*regression[0] + regression[1]) for i in range(15)))

output\_file("regression.html")

line(r\_x, r\_y, color="red")

# Specify that the two graphs should be on the same plot.

hold(True)

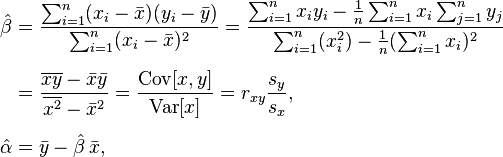
scatter(x, y, marker="square", color="blue")

show()

Plots

And that is fine and dandy. However, we’re left with the cold hard truth: using NumPy is lame when you can implement the algorithm yourself.

In fact, creating a basic one-dimensional regression takes less than a dozen lines of Python! Specifically, we’re going to be co-opting this [simple linear regression equation](http://en.wikipedia.org/wiki/Simple_linear_regression) from Wikipedia:



(This is a little different in practice than the standard linear regression equation, which involves pre-computing the mean and subtracting it from values as we multiply and sum them. The trade-off here is a slight performance increase and a slighter readability increase in exchange for theoretic precision issues due to floating point arithmetic. That being said, this is totally cool for the purposes of didactics!)

Anyhow, this equation becomes:

def basic\_linear\_regression(x, y):

# Basic computations to save a little time.

length = len(x)

sum\_x = sum(x)

sum\_y = sum(y)

# Σx^2, and Σxy respectively.

sum\_x\_squared = sum(map(lambda a: a \* a, x))

sum\_of\_products = sum([x[i] \* y[i] for i in range(length)])

# Magic formulae!

a = (sum\_of\_products - (sum\_x \* sum\_y) / length) / (sum\_x\_squared - ((sum\_x \*\* 2) / length))

b = (sum\_y - a \* sum\_x) / length

return a, b

And if we call our new method using the same arrays as before, we get:

(3.8773910821388129, 13.105872988455717)

Which is accurate within eight degrees of freedom.

 Yay!