Tutorial Four: Linear Regression

Imad Pasha
Chris Agostino

February 25, 2015

1 Introduction

When looking at the results of experiments, it is critically important to be able to fit curves to scattered data points. We will demonstrate that doing so in python is relatively simple, but the theory behind how it works is a bit more involved. For the upper division lab, you may be expected to create best fit lines from scratch, so we are going to briefly go over that here.

2 Linear Least Squares

There are different ways of fitting curves to scattered points. The preferred method is known as Linear Least Squares. Note, we can fit any order polynomial, not just lines, using this method. The "linear" part refers to how the distance between the data point and the line is measured, as we describe momentarily.

The method of LLS fits a line to your data that minimizes the squared distances between all the points and the line. The reason for choosing the squared distances is that some points will lie below your line, but distances are positive. By squaring we allow for points below the line to also be a "positive" distance away from the line. The formula for generating a LLS fit outputs the constants of the equation $(a_0 + a_1x + a_2x^2 + ...)$ for as many orders as you require. For the linear case, then, LLS outputs a slope and a y-intercept. The formula requires linear algebra, which some of you may not have taken, and looks like this:

$$\begin{pmatrix} N & \sum x & \sum x^2 & \cdots & \sum x^m \\ \sum x & \sum x^2 & \sum x^3 & \cdots & \sum x^{m+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x^m & \sum x^{m+1} & \sum x^{m+2} & \cdots & \sum x^{2m} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} \sum y_i \\ \sum xy_i \\ \vdots \\ \sum x^{n-1}y_i \end{pmatrix} \tag{1}$$

This may look a little scary, but its not hard to implement. N is the number of data points you are trying to fit to. To enter the x sums, simply take your x array (such as an array of centroids), and run np.sum on them (squared, cubed, etc as required). The y_i are the y values of the data points, which can be multiplied by the x arrays within the np.sum function.

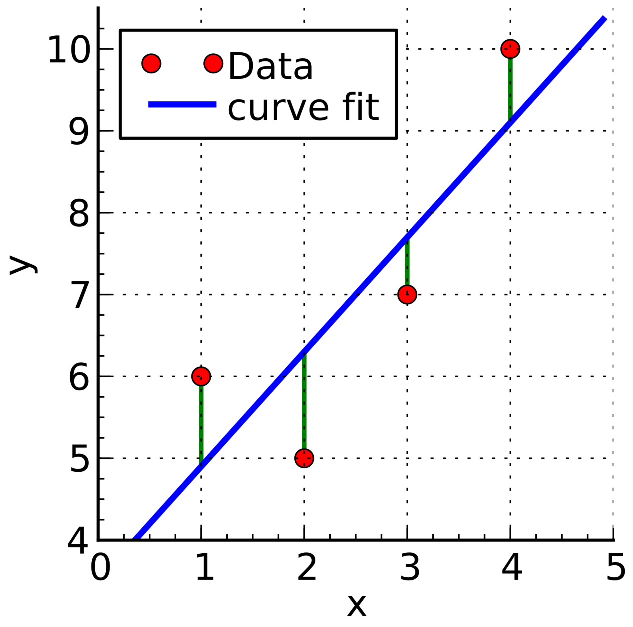

Figure 1: A linear fit. The "linear" in LLS comes from the fact that the residuals- the distances between the points and your fit line- are measured in straight lines. There are multiple types- you could measure vertically, as shown, or perpendicular to the line, or horizontal. It makes little difference.

3 Fitting a straight line

Equation 1 shows us how to fit any order polynomial to a set of data (based on how large you make the array). We are going to practice simply fitting an order 1 polynomial (a straight line) to some data. In this case, the LLS formula simplifies to

$$\begin{pmatrix} N & \sum x_i \\ \sum x_i & \sum x_i^2 \end{pmatrix} \begin{pmatrix} cfit \\ mfit \end{pmatrix} = \begin{pmatrix} \sum y_i \\ \sum x_i y_i \end{pmatrix}. \tag{2}$$

3.1 Writing the script

There is a data file located on the tutorials page. It has two columns, one for x and one for y. Load it, (transpose it), and create separate variables for x and y. You can plot it if you want to see what the data looks like(use '+' or 'o' in your plot argument to get discrete points). In order to get cfit and mfit (the intercept and slope) out of equation 2, we need to put our various arrays into matrices and then perform some basic matrix algebra to solve for the matrix containing cfit.

To construct the matrices we can use numpy matrices or numpy arrays (they really work the same until you start doing more advanced matrix operations). We will stick with arrays. You will need to use what we learned about multidimentional arrays. N is just the length of your x or y array.

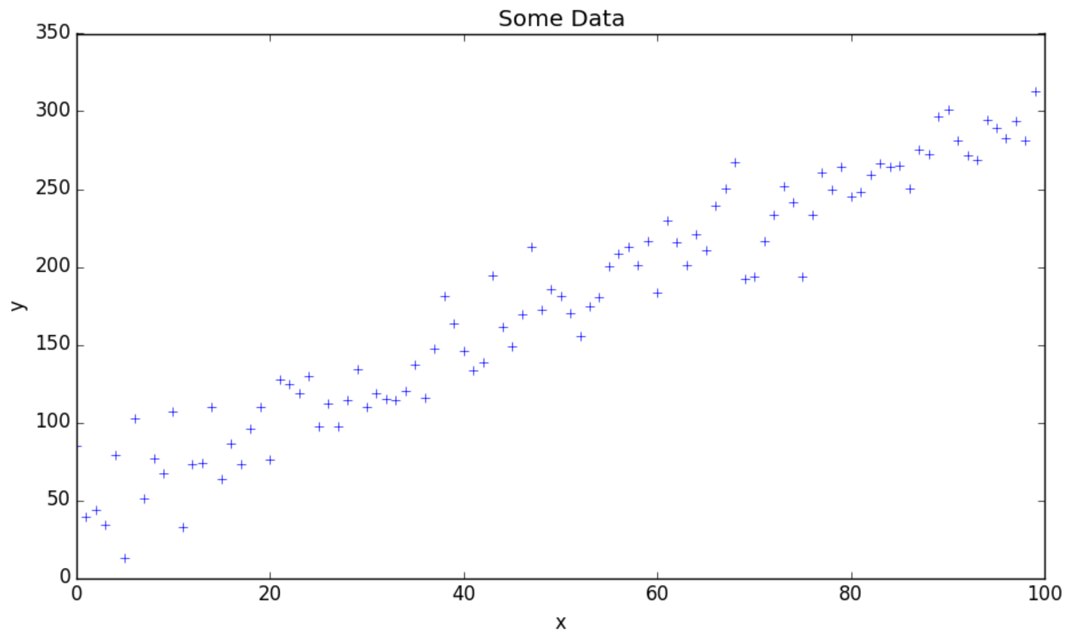

Figure 2: The data you will be fitting

If you need help, this is the syntax for creating the 3rd matrix in equation 2:

matrix_three = np.array([[np.sum(x_i*y_i)] , [np.sum(y_i)]])

Once you have youre three arrays made, (and you can test print them to make sure they have the right dimensions), you will want to solve for cfit/mfit. To get it by itself, we will multiply both sides by the *inverse* of matrix 1, which can be computed by setting matrix_inv (or whatever you want to call it) equal to np.linalg.inv(matrix_one). We can now set a variable (such as 'output') equal to np.dot(matrix_inv, matrix_three). If you index this final variable properly, you should get out a best fit for the slope and intercept. For reference, the numbers you should get are approx. 2.5 for the slope and 50 for the intercept.

3.2 Plotting the overfit

As a last step, create a best fit line to run through the data.

1. Create an x array (np.arange(101))
2. Set a variable y = mfit*x + cfit
3. Make sure the data is also being plotted, if it isnt, plot it using plt.plot(xdata,ydata, 'bo')
4. Then simply type plt.plot(x,y, 'r', label='best fit')
5. Type plt.legend()

6. Type plt.show()

4 Numpy Polyfit

If all of that seemed horrible to you, the good news is you will never have to do it again, with the sole exception of once during the upper division lab. Now that you understand how fitting a best fit line works, we can use a built in numpy function to do the same thing. The function is np.polyfit, and it can fit any degree polynomial to your data. The format is

output = np.polyfit(data, degree)

where data is the y array of your data and degree is the degree of polynomial you want (1 being a straight line, 2 being quadratic, etc). The variable output will store the constants, in ascending order, of the polynomial you fit (and can be indexed to pull them out individually).

1. Add code to your document that will calculate a quadratic fit for the data. Plot it over your data and your linear best fit, in a different color from both.

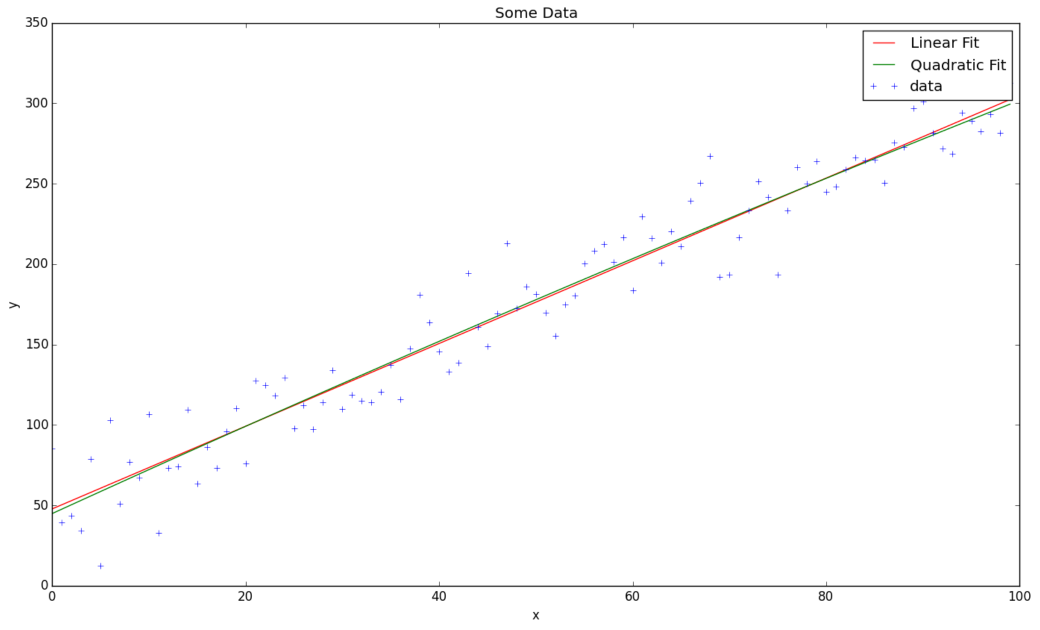

Figure 3: The final image, with both a linear and quadratic fit overlaid.

5 Examining Residuals

So we have now fit both a line and a quadratic to our set of data. The question is, which is a better fit? Should we do a cubic instead? Is more always better?

The answer to the final question is no. Ultimately we can always find some ridiculous polynomial that will cycle all over the place but pass through every single point, but this is not a good representation of our data. Of course, if we know that our data should be linear (maybe we are measuring Hooke's law), or we know it should be quadratic (measuring a gravitational acceleration), then we can just pick the appropriate polynomial. But what if we don't know? (What if that is in fact what we are trying to find out?). The way we can quantitatively check how "good" a fit is to our data is to examine the *residuals*. The residual can be seen in figure 1 as green lines- it is the distance between each point and the fit line (the distance that the fit minimized). The question we want to ask is, does a linear fit produce the same residuals as a quadratic? (and so forth). Let's find out.

5.1 Plotting residuals

After typing plt.figure() in your code, you are ready to plot the residuals from your two fits. It's very simple. Your linear best fit line, call it yfit, is equal to $a_0 + a_1 x$, where the constants come from your fit (either matrix or numpy). You can make an array of residuals by subtracting the datapoints y from the fit line (resid = yfit-y). This will make a new array with the distance between each data point and the fit line.

1. Create residual arrays for both your linear and quadratic fits.

2. Plot the two on the same figure, and see if there's a significant difference.

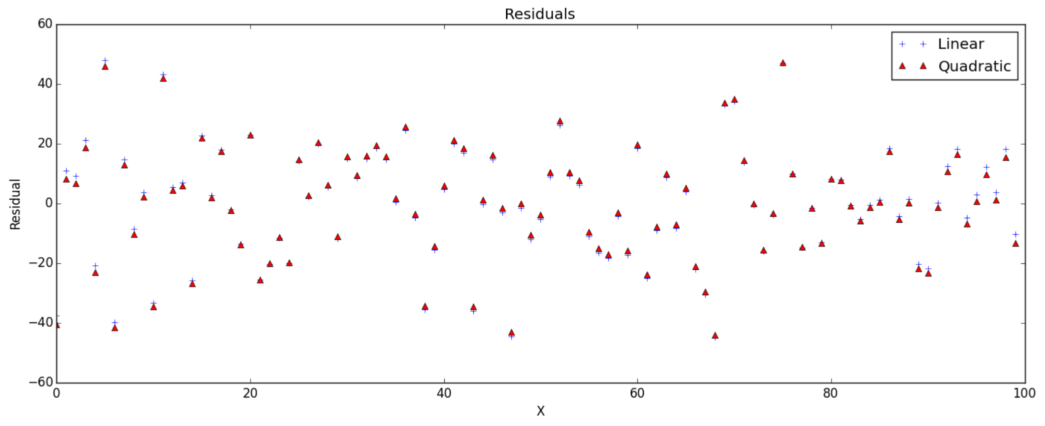

Figure 4: The residuals for the linear and quadratic fits. They are quite large here, because the data was very spread out. But notice how there is little difference between the linear and quadratic fits- the residuals dont improve by using quadratic. This suggest that our data was in fact inherently linear (which was indeed the case).

If you are trying to fit only several points, you might find that the lowest order fits produce residuals that look very suspiciously functional- this is BAD! Seeing that pattern means you want to move to a higher order, to see if it disappears. Here is an example from one of the Astro 120 labs (figure 5).

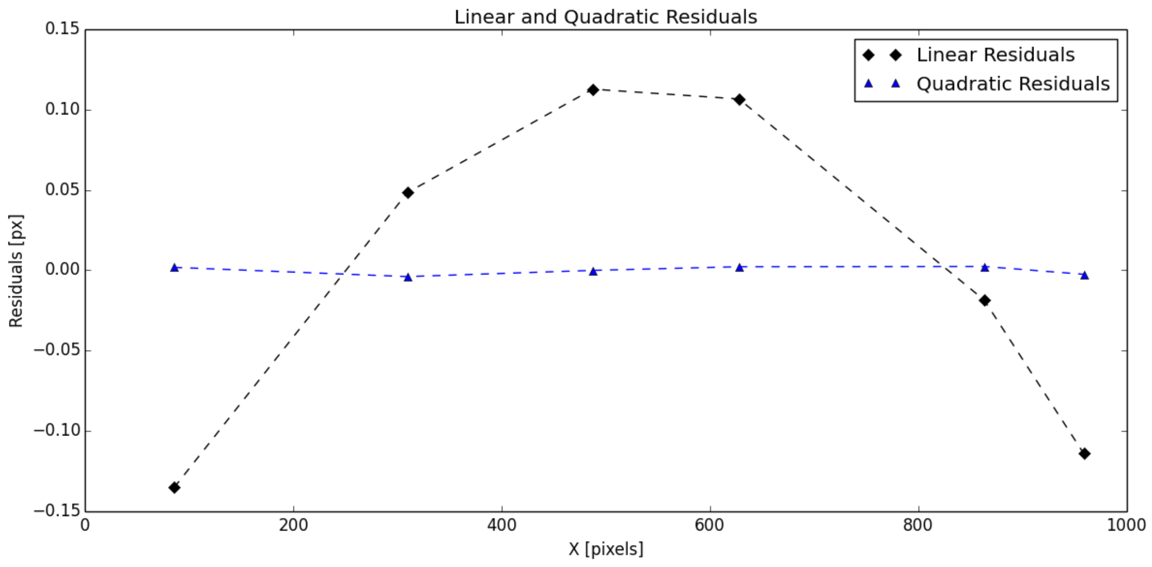

Figure 5: The curved, functional appearance of the linear fit means that a higher order fit is needed. The quadratic fit, on the other hand, has much smaller residuals that oscillate much more randomly about 0.

6 Homework

The homework for this week is to finish this lab. If you need help, please feel free to come to office hours! What you will turn in is a zip of (1) your code (commented!), (2) an image of your data with the overlaid fits (see fig. 3), and (3) an image of your residuals (like in fig. 4). Practice making your plots pretty with legends, titles, axes labels, etc.

7 OPTIONAL Homework

If you have masochistic tendencies, look here!!
Last week we graphed and centroided a spectrum of Neon. The reason we needed to know how to do this was in fact to perform a linear least squares fit between the centroids and some known Neon emmision lines, in order to calibrate the spectrometer. Basically, the spectra we looked at were plotted against pixel, whereas we wanted to know them in terms of wavelength, so we need a conversion, a scal factor that can convert pixels into nanometers. If you plot our centroids we found against known Neon values you will find that it looks almost linear. If you fit a line to those points, the slope of your line represents the conversion you are looking for. If you want, you can use online resources to find the real Neon lines to match against your centroids, fit a line to them (check your residuals, you might need quadratic), and take the constants as your wavelength calibration. IF you do this, congratulations, you've just completed the second Astro 120 lab (don't tell the instructor you have though!) To get you started, the really high peak in the Neon you plotted was at around 585 nm. I would suggest, instead of trying to fit all your centroids, just pick four or five that you can match and do your fit with those.