Data-Driven Computer Science Coursework Report

Grace Stangroome

I have used least squares as my linear regression. That is,

$$A = (X^T . X)^{-1} . X^T . Y$$

where X is the x values passed in and Y are the y values passed in. The x values are resized so that the matrices are the correct shape for multiplication.

This resizing works as follows. In the linear case, a column of ones is added before the column of x's. In the polynomial case, X is a matrix of: a column of ones, the x values, x^2, x^3. In unknown cases, the column of sin(x)'s appears first, and is then preceded by a column of one's.

Linear coefficients are the gradient and y-intercept, and the line is simply y = gradient * x + y-intercept, where x is a set of resized evenly spaced numbers over the set of 20 x values. The polynomial coefficients is the least squares of the resized x values, and the y values, and the line is a set of resized evenly spaced numbers over the set of 20 x values multiplied by the coefficients. Very similarly, the unknown coefficients are the least squares formula of resized sin of the x values and y values. The unknown line is then the resized sin x values multiplied by the coefficients.

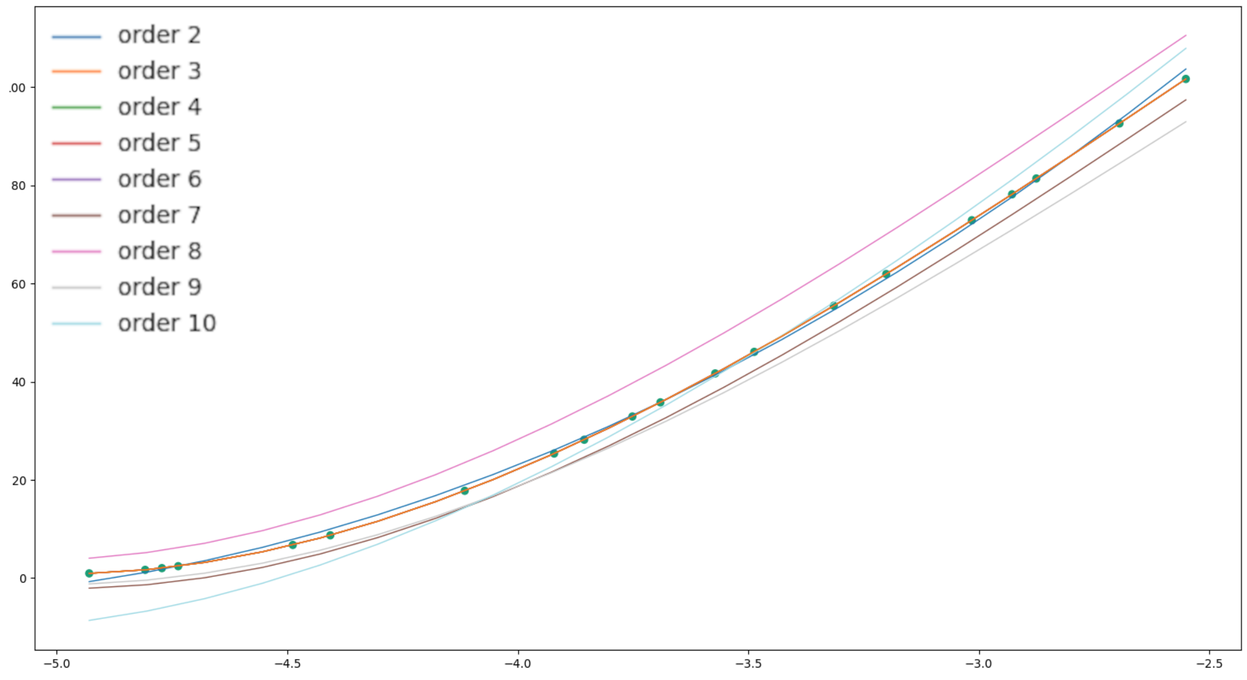

Figure 1: Each polynomial order plotted once on Basic_3. Generally, as the order of polynomial got larger, the fit got worse. Order 2 was quite good, but orders 3, 4, 5, 6 performed almost exactly identically. (They all follow the orange line in figure 4 when zoomed out that far.) However, the larger the polynomial, the higher the likelihood that over-fitting is occurring. Therefore I feel quite confident that the cubic is likely to be the true fit.

For the polynomial order, I wrote a program that split the 20 data points into 4 randomly selected testing data points and the remaining 16 were training data. These were chosen simply to have 80% of the data as training, as this is convention for best results. It then plotted every order from 2-10 on the training data points by resizing each one to the correct size for that order of polynomial and plotting the resulting line. Then, I found the squared error of each line by using the testing data and coefficients calculated from the training data. This was repeated 100 times, and then it found the all time lowest error. Then it re-plotted with that order for the entire data set, calculating the squared error through using the coefficients.

I then ran this 3 times on Basic_3 data set, which I felt was definitely polynomial due to the shape of the data. Every time, an order of 3 was reported to be the lowest error,and it also calculated the total error of the entire data set to be 4.403496031576045e-19 each time. I consider this to be a truly reliable result, which is why I opted for a cubic equation for my polynomial.

I used the same method as above to select the unknown function, between sin, cos, tan and exponential, as I felt these were good options for what the unknown function could be. I only ran this once on basic_5, and not 100 times as looking at the graph it’s indeed quite obvious that sin is the best fit. The program also calculated that sin did indeed fit with an error of 5.896893038708107e-26. I also do not think it is over-fitting, because the line is not making an extreme effort to touch all of the points.

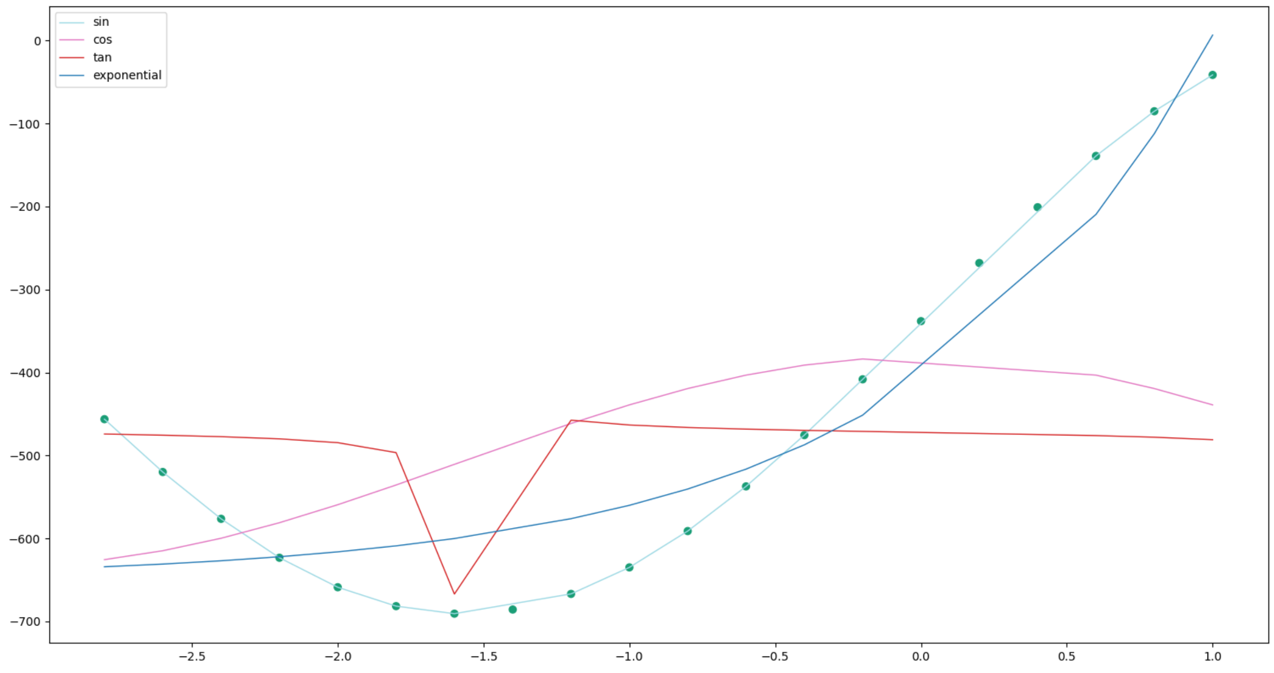

Figure 2: Unknown testing on Basic_5. Sin clearly has the best fit, but follows a nice curve not entirely tied to the points.

In order to select between linear/polynomial and unknown, I used Monte Carlo cross validation, which is the same method I sed to find which order of polynomial and unknown function to use. This works well with this data set size, whereas other forms would be too large. Monte Carlo is also impressively good at prevention against over-fitting. This is why I believe that repeating the Monte Carlo 100 times and picking the test with the lowest error makes the method significantly more foolproof.

The error itself is calculated by recreating the line with the relevant coefficients and x values, and finding the difference between each re-created value and the y values for each x value. This is then squared, and returned. For example, at the Monte Carlo stage, the coefficients are those which are calculated with the training x and y sets, and the x and y values are the testing sets.

For the total final error returned at the end of the program, the line is recalculated over the whole set of 20 x and y values. The coefficients from this are then used to calculate the error with the same set of 20 x and y values. These are calculated for each segment, and the total is the addition of all the error values from each segment of 20.

When viewing adv_3, which contains all 3 different types of function, the lines all plot well, with no signs of over-fitting. In noise data sets, the error also tends to be higher, (in the ranges of 12 – 17 over the usual 0-5) which further affirms my belief that the program probably isn't over-fitting.

However, certain sections do seem to flip-flop between polynomial and linear. There is a potential that when it does pick polynomial it is over-fitting. Perhaps a fix is to, instead of pick the lowest error out of 100, pick the function that had the lowest error most times. This would potentially make it more consistent but still may alternate between the two options. I do not believe this is significant, because it does not appear very often. In the very noisy environment of the first part of adv_3, it alternates between linear, polynomial and unknown. Unknown looks like it does overfit and this might be fixed with regularisation, but the data is very noisy and both polynomial and linear do look relatively okay.

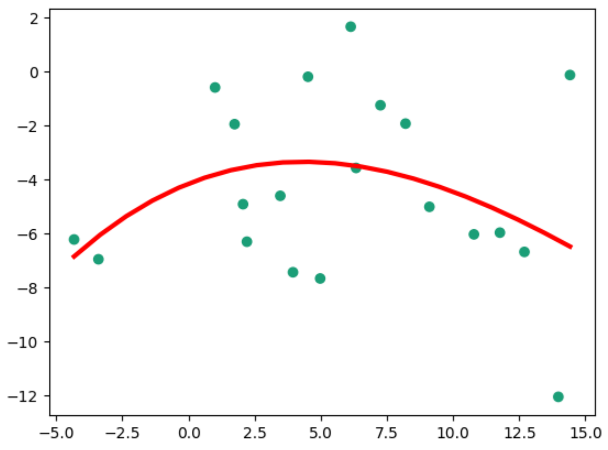

Figure 3: First Part of adv_3 with polynomial line

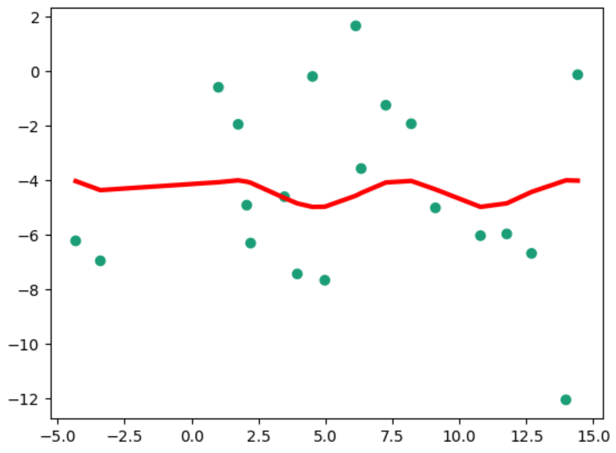

Figure 4: First section of adv_3 with unknown line

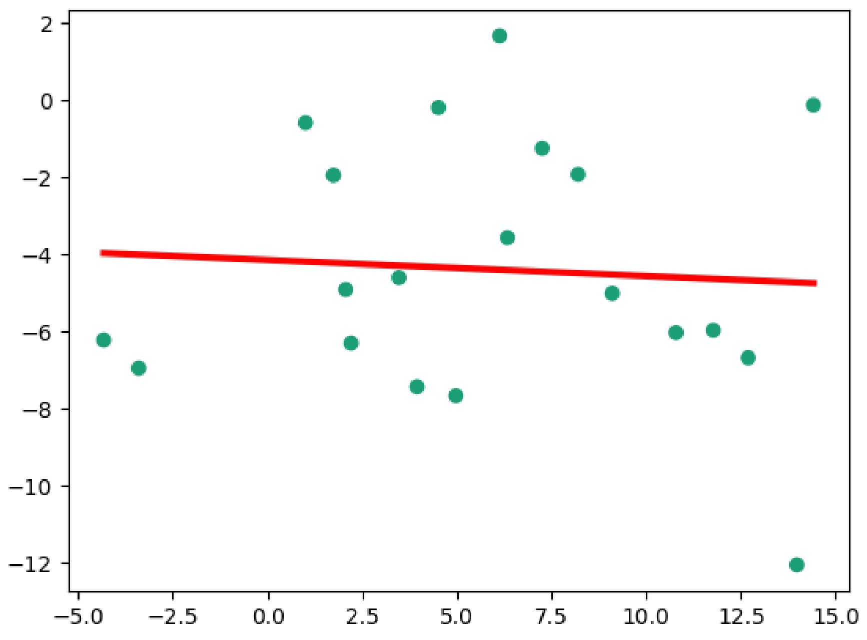

Figure 5: FIrst part of adv_3 with a linear line

Section 4 of Adv_3 is sometimes linear and sometimes polynomial. It’s hard to detect to my eyes which one is correct, and the two lines are very similar and therefore I do not think this is a large issue. Higher orders of polynomial can imitate lower orders, and linear is technically an order of 1. Generally, it works well with most cases and therefore I think it is simply working as it should.