Salter and Smoother Implementation with CSV Use

By: Brendan Lee

Table of Contents

The Programming Process

Surprisingly, of all the programming involved I believe this part of the project took me the longest. There was a lot of reworking code and changing things, so hopefully it's still understandable and isn't too much like spaghetti.

First of all, the function I used for this assignment was y = mx + b, so it's a simple line with one constant slope. I created the FunctionOutput class which generates the output values and builds a CSV with them. After that, I created the CSVReader class which can parse through CSVs and store the values in array lists. This is where I stopped on this section of the project initially, and then I found out we had to also add salting and smoothing (what I refer to as "smalting"). Originally I just built it into the CSVReader class, but I realized that it'd be silly to have an object that's supposed to be reading CSVs start manipulating datasets.

Therefore, I moved the salter and smoother functions to the Smalter class. The salter itself was extremely easy since it's basically just adding a random integer to each data point, but the smoother took a bit more time. After brainstorming different methods on how to do it, I decided on calculating the window of values of each point so it would be easy to just average everything inside.

Once the Smalter was ready, it was simply a matter of combining everything together into one CSV file. To accomplish this, I created the FinalCSVPrinter that would print a csv for any amount of outputs

Notes About the Program

While the program certainly works, there are a few nitpicky things that can be improved upon or changed depending on what the user needs. For starters, it only generates one CSV at a time. It was slightly tedious having to run the program multiple times in order to generate the graphs I needed. Next, the program itself isn't extremely configurable. In order to generate different FinalCSV.csv files, you'd have to manually change the code in the run method. While it isn't a huge hassle to do so, there's no built in functions that can do it with parameters.

As for the Salter itself, there's some limitations. For example, the random number generation when I created the graphs was quite sloppy. I originally used the java.util.Random class, which is quite outdated and slow. Eventually it was changed to SplittableRandom, which should be far more effective and closer to genuine randomness. Furthermore, the randomly generated salted values are different for each run. This is because I didn't implement any sort of key that can be used to generate the same numbers. As a result, each of the tests used different salted values, which can throw off observations quite a bit.

Lastly, the smoother had only one personal concern. Data values in the beginning and end of the dataset don't have enough neighbors to be averaged as much as the others. Patrick gave me the idea of just cutting off the ends of the dataset to fix this, so I implemented it in the cutOffEnds(outputs, range) method though I don't use it in the experiments.

Instructions Manual

The only file that needs to be configured to run the program properly is in Main.java. The main method has two methods being called with parameters, and all you need to do is edit those parameters to run the program.

- functionOutput.run(x, x2, increment, m, b)
- functionOutput.runSmalter(saltValue, smoothRange, smoothCount)

For the first method, the parameters specify the y = mx + b function you'd like to graph. The x and x2 specify the x range, while increment specifies how much you want the x to increment. For example, the parameters (0, 6, 1, 2, 5) would have the x values of [0, 1, 2, 3, 4, 5].

For the second method, the parameters determine how you want the salter and smoother to act. It's worth noting that if you aren't at all interested in the salting or smoothing, you can completely remove the second method and the program will still write the function's inputs and outputs to InitialCSV.csv. Using the second method as well will put the generated dataset through the salter and smoother and output the data to FinalCSV.csv.

However, if you find that you'd like to configure the FinalCSV.csv creation, you would need to edit the runSmalter method at the bottom of FunctionOutput.java. This is where FinalCSV.csv is built, and the FinalCSVPrinter class is built to handle different amounts of outputs depending on the runSmalter method. There's no actual parameters or functionality for configuring it outside of editing the code directly, but the editing process is quite easy.

As you might have guessed, the FinalCSV.csv is the golden nugget of the program. The way it's set up currently, x is the input, y1 is the output of the original function, y2 is the output that's been salted, and y3 is the output that's been salted. It's in the perfect form for being turned into a graph in excel. Note that in order to save this csv file, you must use "save as" with a different name since any file with the name "FinalCSV.csv" will be overwritten.

The Experiment

The goal of this report is to simply study the effects of the smoother with the help of a salter. In each experiment, there are four variables:

1. Input Range - Range of x values being used
2. Salt Value - Largest number possible that can be added or subtracted from each point
3. Smoother Range - Number of values on each side of the index being used for smoothing
4. Smooth Count - Number of times the smoother runs on the data

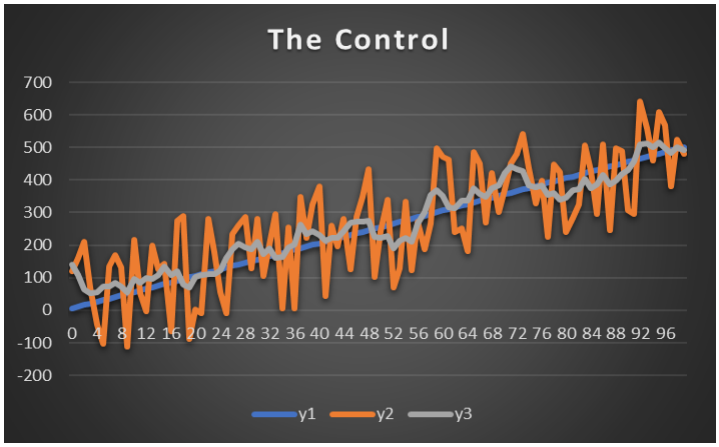
In order to observe changes, this report will be using the following default values as a sort of control. In the order of the list above, the parameters of the control will be (100, 200, 3, 1). Each test should show the effects of tweaking one of the parameters. Additionally, the equation used for the line is $y = 5x + 6$ for every graph.

The Results

For the following graphs, y1 is the original line, y2 is the salted line, and y3 is the smoothed line.

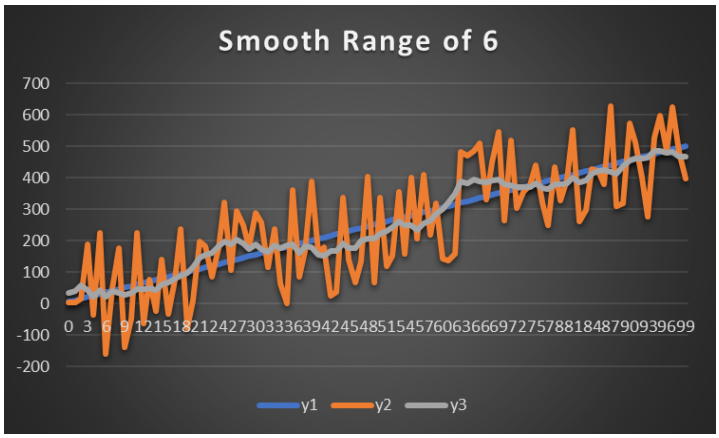
The Control: (100, 200, 3, 1)

First off, the control looks like what was expected. The smoother helps to bring the chaotic salted line back to its original state. However, the smoothed line itself doesn't actually appear very smooth in general. In fact, it almost looks like the line wasn't smooth at all and was instead just lightly salted. Additionally, you can see that in the very beginning of the graph, the smoothed value is actually higher than the salted value.



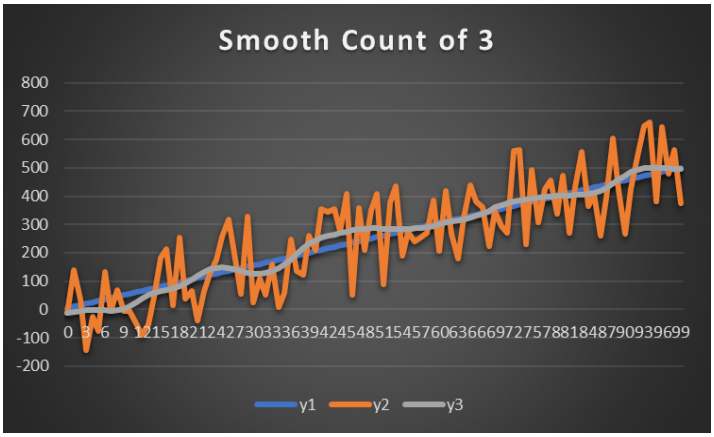
Higher Smooth Range: (100, 200, 6, 1)

A higher smooth range seems quite effective at making the smoothed line less erratic. While the values aren't necessarily brought closer to the original line, the ups and downs of the graph seem much more minor than the control.



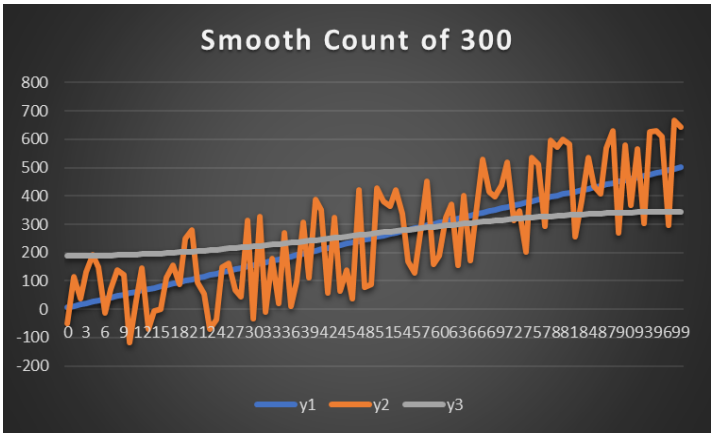
Higher Smooth Count: (100, 200, 3, 3)

With a higher smooth count, the function finally lives up to its name and produces a relatively smooth line. There's no "mountains" in the data, though there are hills that rise and fall depending on how the data was salted. Furthermore, you can see that the values in the smoothed line are very close to the original values of the function.



Much Higher Smooth Count: (100, 200, 3, 300)

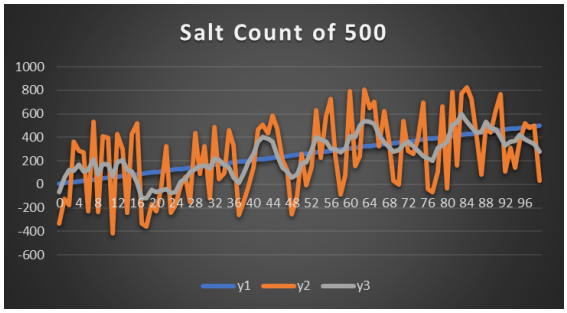
Of all the graphs, this one surprised me the most. It seems that when the smooth count is excessively high, it begins to create a straight horizontal line. It makes sense, since each value is being brought closer and closer to the mean of the dataset after each smoothing, but I initially believed that the line would more closely resemble the line of the original function. It's also worth noting that it seems smoothing has less and less of an affect as it's used. Even after three hundred smooths, the line still bears resemblance to the original function. While it may get closer and closer to a perfect horizontal line, the effects seem to lessen exponentially.

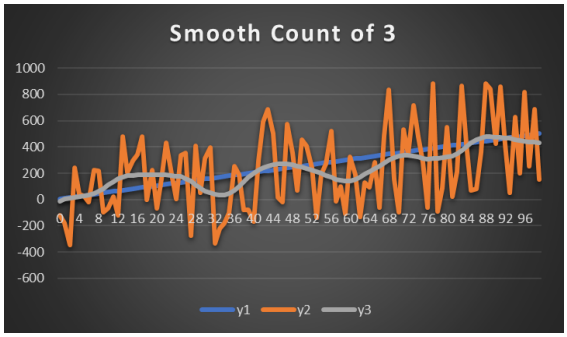


Higher Salt Value: (100, 500, 3, 1)

It seems that increasing the salt value doesn't have much of an effect on the smoother. Since the smoother is working with averages, it doesn't really matter if the salt values increase, since they'll be evened out easily and consistently.

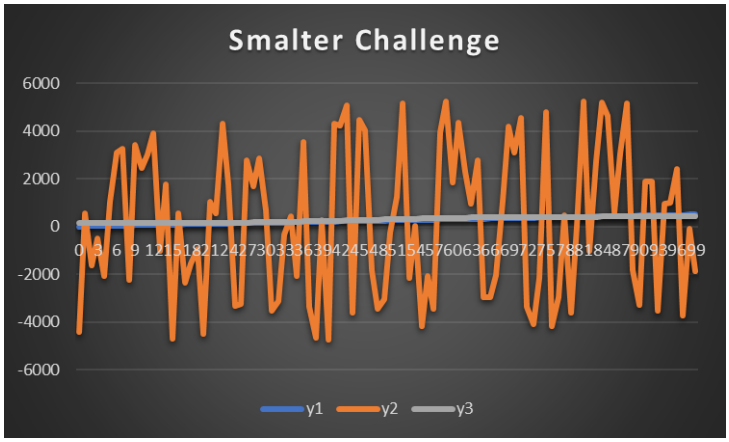
Out of curiosity, I increased the smooth count to three and noticed that the higher salt values didn't do much here either. However, it's worth Noting that the smoothed lines do appear a slight bit farther from the original line, though it isn't extremely impactful at all.



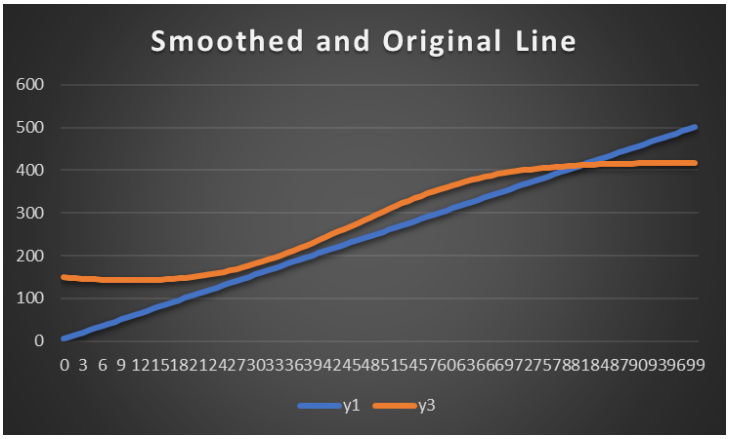


Final Test, the Smalter Challenge (100, 5000, 10, 10:

As a final test, it was decided that the smoother should be tested with a much larger salt value, five thousand to be exact. Even though the salt value was massive, a smooth range and count of ten creates a smoothed line that seems to almost perfectly match the original line. However, it's somewhat difficult to see exactly since the y scale is so high. In order to get a better look, I've provided a second graph that excludes the salted line.



It's clear that the line is extremely smooth and decently fits the original line. Still, it seems that the values in the smoothed line are noticeably higher on average than the original function. This is likely due to the x range only being one hundred versus a salt value of five thousand, though I can't test it very well since excel's graphs don't seem to enjoy when the x is one thousand or ten thousand.



Conclusion

After observing the data, I've concluded the main effect of each of the four parameters. The input range gives us much more data to work with, so random chance throws off the data less and less as we use more values. The salt value makes the smoother's job harder of course by randomizing the data, though having a larger x range helps to keep things consistent. The smoother range ultimately helps to remove frequent peaks and dips, but it doesn't bring the line much closer to the original. The smooth count, however, makes the largest difference. It brings the line very close to the original, though it will simply start to make a horizontal line if used too much since it will slowly average every single index on the line until they're almost equal.

If someone was ever to use this implementation of the salter and smoother, it's probably best they decide to use something else. Though the program works, it isn't very optimized and utilizes array lists which can be quite intensive if it's working with extremely large values. Still, it gets the job done decently well.