Report on plotted, salted and smoothed data (Java Edition)

Introduction

While progressing through this project, graphs and data were collected from multiple collection methods. One being the Java method. That method is the topic of this statistical report. Three equations are presented. One being a form of a quadratic equation, the second being the distance formula in its raw form, and the third being the original absolute value formula. All three of these equations were plotted, salted and smoothed using Java without any extra libraries or extensions. The purpose of this project was to show the difference, amount of difficulty and *effort* to navigate through all the approaches performing the same action: creating a plotter, salter smoother. Since Java is a semi-lengthy, object-oriented programming language, the amount of work was medium, but that is coming from someone who has worked in Java for years. However, if someone was learning from scratch, it may take a longer amount of time. It took learning how to work the built-in Java read and write properties (ex: filereader and bufferedreader).

The first part of this project focuses on regular Java imports and code without libraries or additional properties. It utilizes a class for each action: plotting, salting and smoothing as well as multiple Java methods. Each method has its own Java doc comment, detailing its purpose, parameters and return type. This project promoted experimentation, different organizational approaches and ways to work in Java (ex: using separate methods and classes), but this organizational approach was chosen, so if anyone ever needs to use the program, they can choose as they please if they want to plot, salt or smooth data based on the multiple tester classes.

This project part shows how when doing everything manually, it may be inefficient when someone else has created a library or a program (MATLAB) to do what needs to be done. Coding everything from scratch took the lengthiest amount of time out of all three parts. Learning the libraries and MATLAB was much faster, because they had graphing software built in. MATLAB was a couple of lines of code to produce the plotted graph, salt and smooth it. Java took multiple methods, writing a file writer and reader, producing .csv files and then creating a graph in Excel off those files. However, as stated above, it was a learning process to experiment with different methods of performing the same goal.

**What is a plotter, salter and smoother?**

*Plotter*

A plotter is what it sounds like. It plots a function, and that means any function. The graph that it creates will be the raw data with no changes to it. So, for example, if you put in a sine function, it will return to you that exact function you inserted. The plotter will plot a quadratic equation, the distance formula and the absolute value formula. It produces the first .csv file of all three.

*Salter*

The salter’s purpose is to mess up the initial data. It may add or subtract values, excessively or softly and make your data spike up and down, different than its original form. Basically, it takes a random number 0 – 10000 and subtracts 100 from it. The data points are read from the plotting .csv file. Then, after that, a new file is created again with the new salted values.

*Smoother*

The smoother takes values within a range, averaging them which *fixes* the data. In this case, it is very subtle. The value chosen was 3, and the graphs are extremely salted to show the mess that salt can create. The smoothed graph looks much nicer than the salted graph, however, and does show a positive change. The smoother reads the salted data for all three functions. Then, it creates the final .csv which is full of smoothed data.

***Note:*** *The FULL code documentation for these programs can be found in the “documentation” folder. This document does summarize methods at the end.*

The why

**Why were x-values from -50 to 50 chosen?**

At first, thousands of values were chosen during this experiment. However, that led to the conclusion that too many values make the graph too big to view fully and inaccurate. The same goes for too few values, it doesn’t show a full representation of the plotting, salting and smoothing process.

**Why were three formulas chosen?**

Replication is wonderful in any sort of experiment. Doing something three times with three different datasets shows how a process works accurately and properly, proving how difficult it is to get through that process. That stands for comparing basic Java and Java libraries.

**Why was such a high salting value chosen?**

It was chosen because the lower values used did not really show the salt. To make this report showcase true salting, 10,000 was used to show the spikes in data jumps from high to low as each data point was extremely messed up. After the high value was ran through, a small and a moderate value were chosen too.

**Purpose of the Program:**

* The role of the program is to plot, salt and smooth a function. The program does this mainly in the “PlotSaltSmoothData” class. All the classes were split up for organization and personal preference. There is a salter, plotter and smoother class along with their personal testers, in case someone only wants to perform one.
* Within the first class is the plotter as well as the write to csv method. This allows the data to be plotted and then written to a .csv file to be salted. There are three functions plotted. A quadratic, the distance formula and the absolute value formula based off different parameters.
* Within the second class is the singular salting method with the write to csv and read to csv methods. The parameter for salting can be changed in its tester class, so only one method was needed. The plotting data is read from the plotter class to be salted, then a salted version is written to a new .csv in the tester class.
* Within the third class is the singular smoothing method with the write to csv and read to csv methods. The parameter for smoothing can be changed in its personal tester class, so only one method was needed. The salted data is read from the salter class to be smoothed, then a smoothed version is written to a new .csv file in the tester class.
* The program works off x and y array lists, adding and changing values as it goes.
* All the .csv files appear in the folder after the program is run. If the parameters are updated, the new data will be inserted into the .csv files, erasing the old data.
* Three functions were chosen here to test different salting and smoothing values at first. Since this was the first of three programs, it was a learning experience to create.
* The .csv files can be open in Excel to better look at the data and create graphs to visualize it. A cool plugin in VSCode is called rainbow CSV and allows the user to color-code each column in CSV files. That is a cool addition if users plan to create larger csv files or want to use it to test within this program.

The first formula being plotted was the quadratic function:

***Fig. 1.1: Graph of original Quadratic function data***

A simple quadratic equation was developed in Java and transferred into a .csv file for plotting purposes. The point range was specified from -50 to 50. This number of points was picked to get an accurate graph that shows multiple Y values. If too small of a range was chosen, it may not show enough. However, if too many were chosen, it may not show the entire spread of data during creation of the graph. The Y values range from 2401 to 0 to 2601, creating the quadratic curve.

***Fig: 1.2: Graph of Salted Quadratic function data***

Here is the salted Quadratic graph of the equation above. The bound was 10000 here. A number from 0 – 10,000 was generated for each point. 100 was subtracted from it. It created a messy display, turning the pretty quadratic into a mess. Instead of having a clean quadratic Y-value range which runs from a large number to zero to another large number, the numbers fluctuate up and down consistently throughout the Y-value outputs. That creates the sensation of a salted creation from Java’s main syntax.

***Fig. 1.3: Graph of Smoothed Quadratic data***

Java was used to generate this graph from the same methods above—just using the smoothing method. It was smoothed by 3. This looks as close to the original than any of the graphs, however, it is still spiking and does not meet the original quadratic. It’s starting to make a similar shape though. The smoothing did not make it exactly the original but managed to make it look nicer.

***Fig. 1.4: Graph of Slightly Salted Quadratic Data***

To show a different depiction, a different salted version of the quadratic function was produced. Instead of being salted by 10,000, this was salted by 100. As visible, it is clear to see that it is not as bad. It is not the original graph (slight ups and downs in the data) but it is not terribly salted. It was an experiment to see how intense salt (above) can look versus a little bit of salt.

***Fig. 1.5: Graph of Smoothed Quadratic Data***

This version of the smoothed quadratic was smoothed by 2. It looks much nicer than the slightly salted version, but it serves its purpose to show the job of smoothing a less salted quadratic. The parameters were experimented with here to show the differences.

The next formula was the distance formula:

***Fig. 2.1: Original Graph of Distance Formula***

The distance formula was coded in java next, generated into a .csv file and created into the graph above. It looks like a standard slope line. The output y-values range from -47 to 53. The line ranges from the negative values third quadrant to the first quadrant’s positive values. Because of the parameters included in the Java method (see: Documentation section of project for more), the user can insert any values for , , , . For this graph, = 3, , , . All these values being inserted produced the line above.

***Fig. 2.2: Graph of Salted Distance Data***

The distance formula presented above is then salted. Java was used to salt the formula. The same bound as the quadratic equation was used for consistency. It was 10,000. This was also subtracted by 100. For clarity and ease, they were salted using one method, and it got the job done (see: documentation. However, the method name is: “saltFormulas()”) Because of the extreme salt, the data jumps up and down in wild way.

***Fig. 2.3: Graph of Smoothed Distance Data***

The graph of the distance data smoothed is not as messy as the salted data. It spikes still, but it is fixed. The smoothed method does its job of making the data look nicer and fixing it, not perfectly, but to an extent since the range was 3.

Then the third formula was the Absolute Value formula.

***Fig. 3.1: Graph of Plotted Absolute Value formula***

Java was used to produce this graph in the same method by simulating the formula above. Excel handled creating the graph. The absolute value formula uses the same range -50 to 50 for consistency and clarity to view the entire graph. The y values are from 150 to 0 to 150 which forms the shape it does, appearing on both sides mirrored. The a-value used was 3, however, since it’s a parameter, the user using the program can change the value as they please. 😊

***Fig. 3.2: Graph of Salted Absolute Value formula***

The salted absolute value formula shows an immense amount of salt just like the first two. Java was used here from It was salted by values up to 10,000 and subtracted by 100. It shows the distinct change in shape that salting can do to data. The y-values jump around like the first two and show how much salting can ruin data. This was also salted using the one salting method which salted the first two.

***Fig. 3.3: Graph of Smoothed Absolute Value formula***

Java was used once again to produce the data for the graph above. The x-values are once again ranging from -50 to 50. The y-values still are spiking, but they are not as bad. That’s due to the smoothing of the range 3 from the smoothing method. It can be seen here that the graph looks much nicer—just not exactly like the original graph.

***Fig. 3.4: Graph of Moderately Salted Absolute Value formula***

This formula was now salted by 300. It kind of keeps its shape. The depiction above shows that it still has the V shape of an absolute value formula, however, the points are jumping up and down.

***Fig. 3.5: Graph of Moderately Smoothed Absolute Value formula***

It was smoothed by 6 here. It shows that smoothing it by that large of number kind of gives it back its original shape. It is more of an absolute value function, but it still has errors. However, the smoothing process makes it easier to identify the function.

***An overview of the methods (see: documentation for screenshots and in detail explanations!)***

The plotter class methods are below:

* plotQuadraticFormula() – the method lacks parameters, however, it does its job. It takes x-values from -50 to 50 using a for-loop and then uses those values to create the function as well as the y-values. It has the x-values to the x Array List and the equation’s y-values to the y Array List. The equation is formed using the Math.pow function to put the x to the power of 2. The rest uses basic Java math syntax like 2 \* a value and adding a value.
* plotDistanceFormula(int x1, int y1, int x2, int y2) – this method has the parameters of the values it is going to use to plot the function. The distance formula is applied to an integer variable (this includes using the inserted x and y values from the parameter), and then it is applied to the y Array List by adding both the distance values and the x values. The x values remain -50 to 50.
* plotAbsoluteValue(int a) – this one was shorter. It has the parameter of a. That value scales the graph. It still uses x-values -50 to 50 and adds them to the x Array List. The y Array List obtains the values from the equation which are the x-values multiplied by the inserted value of a.

*All three classes have a reader and writer. It will be covered once. (Plotter only has a writer; it’s the beginning.)*

The writer and reader files are as below:

* writeTofile(String filename) – the parameter here is the filename which the user would insert as a string (ex: “graph.csv” or “graph.txt”). The method uses both file writer and print writer to create and print data to a file. The for loop iterates over all the points and can print both the x and y values. There’s an IOException in case anything goes wrong.
* readTofile(string Filename) – just like the writer, the reader has a parameter for the user to write in the filename. A buffered reader and a file reader are used to correctly read a file. Each line is read. A while loop is used to make sure it is only reading lines are not empty. Both x and y values are added to the array lists. There is also an IOException here in case of errors.

The salter class method is below:

* saltFormula(int bound) – from a for loop from 0 to the size of y, it salts values. The values are salted by generating a random integer from 0 to 10000 in this case (but the bound can be chosen). That integer is then subtracted by 100. It replaces the correct data with the salted data. The salted data will look messy.

The smoother class method is as below:

* smoothData(int range) – this method has a parameter called range which is the range that will be used to smooth the values. As said before, it was 3 here, but any value can be chosen. It has an initial for loop to go through the size of the array list y, then a second for loop which closes on both sides of the range. This creates the averaging process and makes the data look nicer than the salted data.

All three classes: plotter, salter and smoother have their own tester class. That is where the numerical parameters are filled in and the file names are chosen.