

Author: Son Le

Data Science at Aalto University

Project document

Fitting and Visualization of a Regression Model

1. General description

Theme: statistics

Build a program that can be used to fit a regression model to a dataset and visualize the results.

Regression means estimating the relationships between variables. Perhaps the simplest, yet very useful method for doing this is simple linear regression. It can be used when the data has two variables x, y and we can reasonably claim that they are proportional to each other, i.e.,

$$y = mx + c$$

for some m and c – finding appropriate values for them is the “fitting” part of this project. Another regression method implemented was quadratic regression, which fits a quadratic parabola to the data:

$$y = \alpha + \beta x + \gamma x^2$$

The project was completed at the advanced level. The user can load the data from a file of their choosing. The program supports CSV and JSON files. Both linear regression and quadratic regression were implemented. The program produces a plot of the original datapoints and the fitted model. The appearance of the graph, such as the color of the curve and the datapoints, can be configured by the user. The implementation was done with extendibility in mind: if the program needs to have other regression methods or support other file formats, only new classes which can extend the existing abstract classes need to be added.

2. User interface

The program is started by right-clicking on the `RegressionApp.scala` class file in Eclipse (or the corresponding action in IntelliJ IDEA) and choose “Run as Scala application” or press Shift + F11. After that, a graphical user interface will appear. An illustration of the GUI is given in Figure 1. Usage of the program is simple. First, the user opens a file from the “File” menu. Then, the user should let the program know the name of the x, y variables via the “Name of input variable/column” and “Name of output variable/column” text fields respectively (should match letter-by-letter). If the user selects a CSV file, they should also inform the program what kind of delimiter (almost always comma “,”) that the file uses. The user can get the name of the variables from the file itself. If the user wants to visualize all datapoints, they should select “Yes” in the “Visualize all data and see regression metrics?” prompt (there are restrictions on how many datapoints can be visualized, which will be explained further). If the user only cares about the fitted coefficients, then “No” should be selected, as this option reduces the runtime of the fitting process by half. Additionally, if the user wants to rescale the data to a range between -100 and 100 (with min-max scaler), then they should tick the corresponding checkbox. This option is intended to enhance the plot viewing experience of the user. However, this option results in the longest runtime of the fitting process. Performance will be further discussed below.

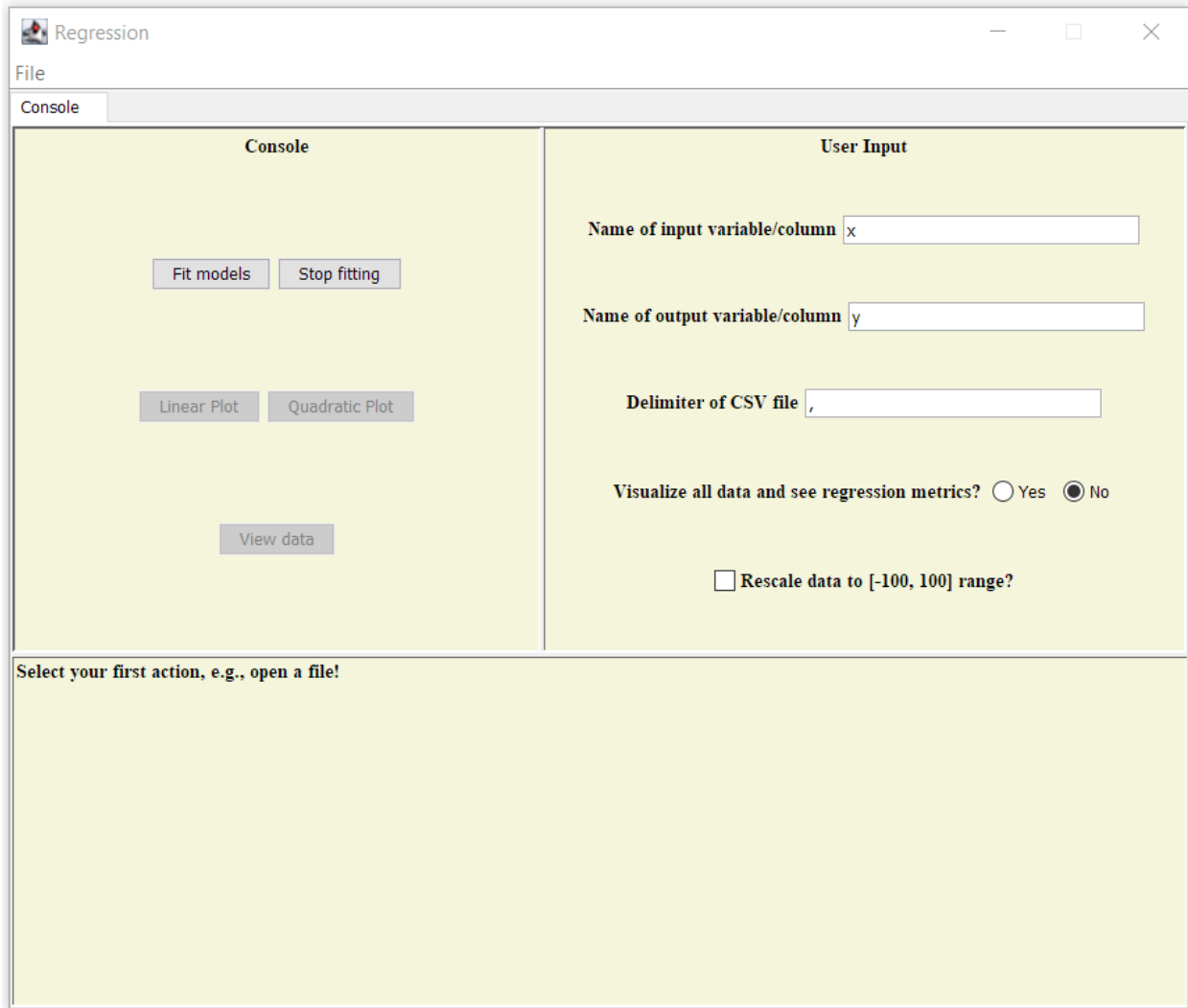


Figure 1. The GUI.

After setting the options, the user should press the “Fit models” button. This will make the program fit both linear and quadratic regression models to the data. The program will inform the user (and abort fitting) if there are problems with the file, e.g., wrong variable names and wrong CSV delimiter. The user can also choose to manually abort the fitting process by pressing the “Stop fitting” button. The greyed-out buttons should not be greyed out anymore if there were no problems in the fitting process. The functionalities of these buttons match their names. “Linear Plot” will generate a plot with the fitted linear model. “Quadratic Plot” will generate a plot with the fitted quadratic model. “View data” will generate a data table containing the datapoints (in numbers) that will be visualized. The appearance of the plots and the data table can be seen in the figure below. After pressing these buttons, new tabs will appear, and the user can switch between tabs. These new tabs can also be closed. The main tab cannot be closed. Apart from the main tab, the user can open at most 5 more tabs. The plot tabs contain buttons which allow the user to move around the plot. The user can also use the mouse to pan and zoom in/out on the plot. There are buttons which allow the user to change the color of the curve and the datapoints.

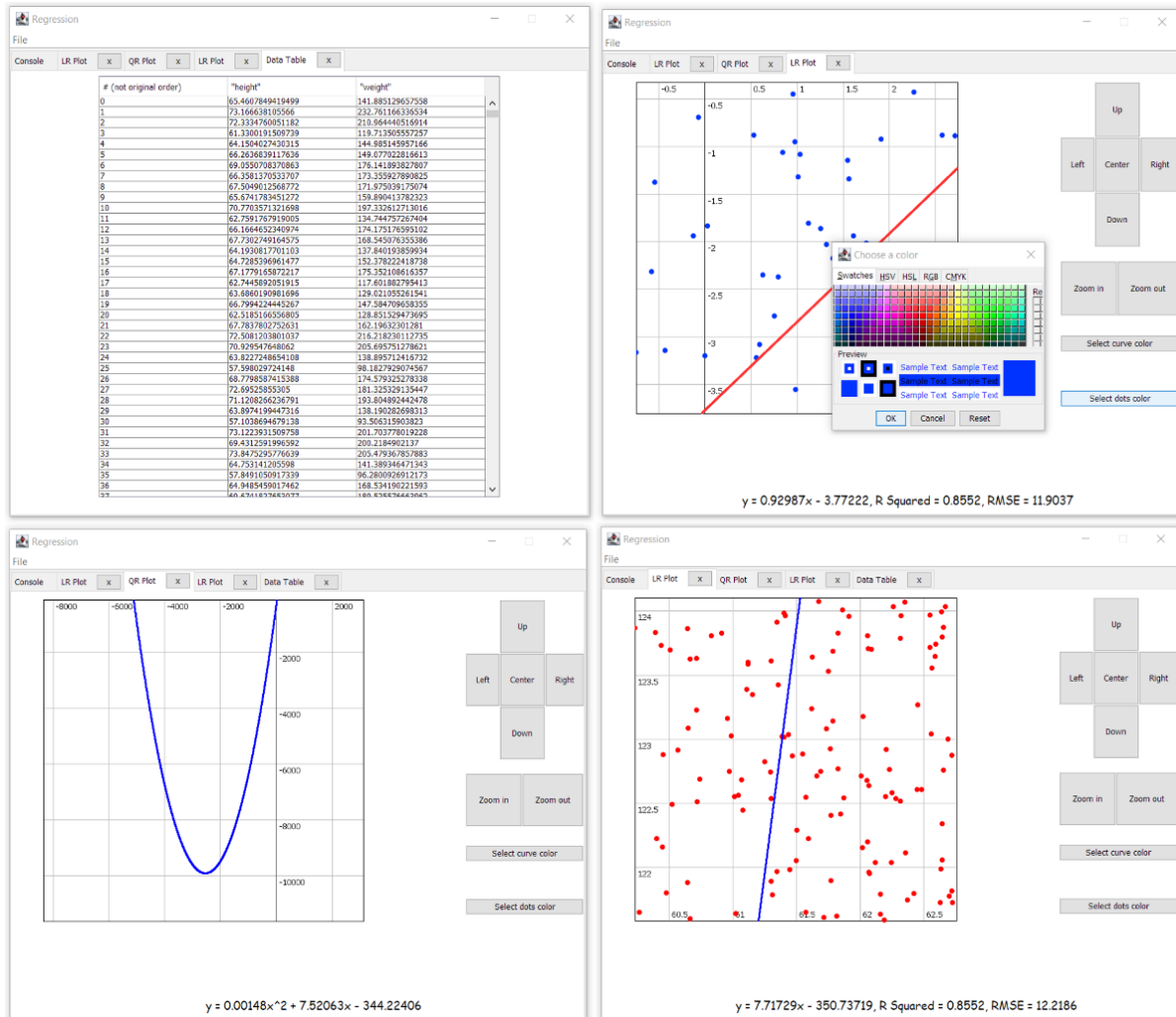


Figure 2. Some tabs of the program.

The bottom of the plot tabs shows the equation of the fitted curve and optionally some metrics such as R^2 score and root mean squared error. In the “Data Table” tabs, the user can view the datapoints that are visualized. There are three columns in the table: the index column, the input column, and the output column. The sole purpose of the index column is to inform the user the number of the datapoints that will be visualized. The order of the datapoints are not preserved from the original file because it is not necessary to do so. The user can change the width of the columns.

An important note is that there are restrictions on the number of datapoints to be visualized. Even if the user chooses “Yes” in the “Visualize all data...” prompt, the maximum datapoints that will all be visualized is 1 million (arbitrary limit). In other words, *if the file contains fewer than or equal to 1 million datapoints, all of them will be visualized* if the user chooses “Yes”. However, if there are more than 1 million datapoints, then the program will randomly load the datapoints such that only 1 percent of the original dataset will be visualized. On the other hand, if the user chooses “No”, at most 1000 datapoints will be visualized. Of course, in both scenarios, the fitted curves are visualized. These limits exist in order to avoid slowing down the plotting process because actions such as panning and zooming when, e.g., 10 million datapoints are visualized, are very slow and frustrating. The performance of the program will be further discussed below.

3. Program structure

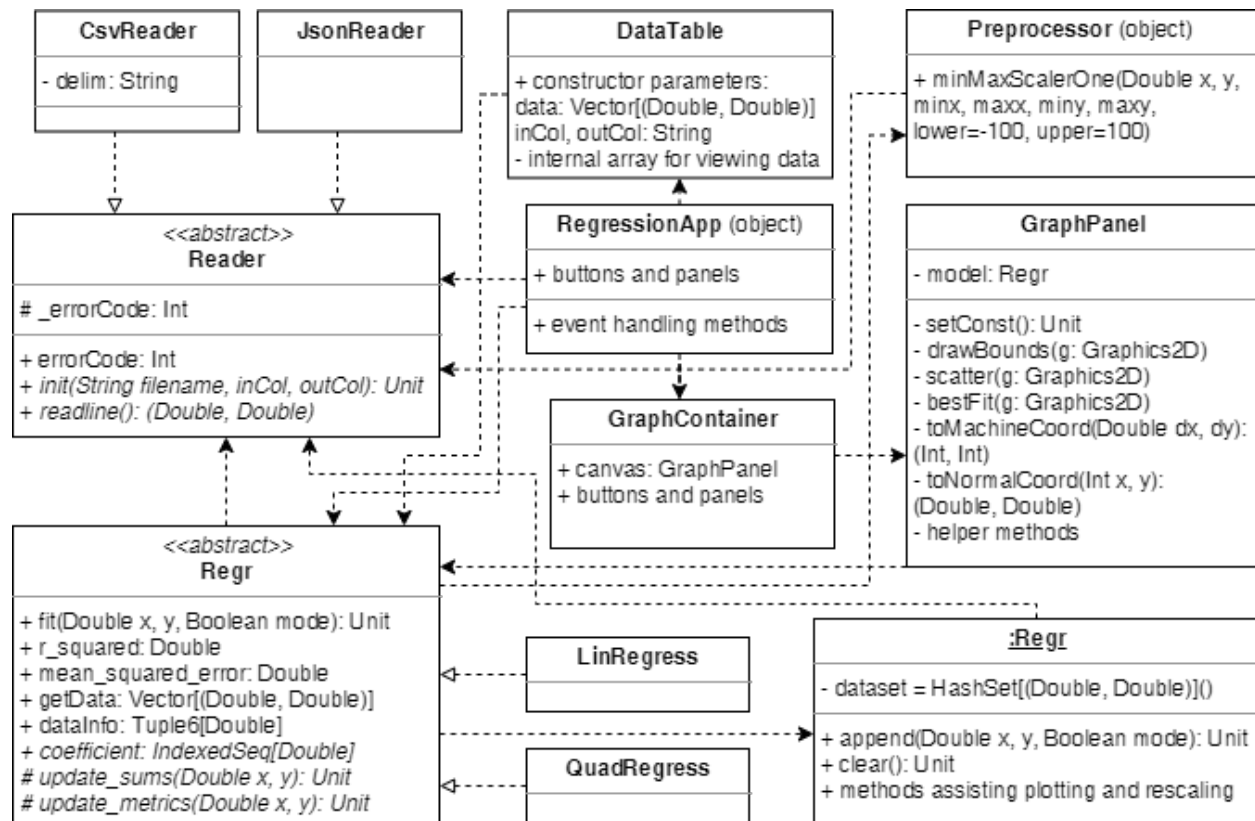


Figure 3. UML diagram of the program.

The program has 3 packages: `regression`, `regression.gui`, and `regression.io`.

The `regression` package contains the following classes: **Regr**, **LinRegress**, **QuadRegress**, and **Preprocessor** (which is a singleton object). **Regr** is an abstract class describing the concept of regression in general. It is the base class for all regression classes. The most important method of this class is the `fit` method. This method controls how the models are fitted. There are methods that returns the regression result metrics such as root mean squared error and R^2 . **Regr** has a companion object **Regr**. The companion object is mainly used for collecting data into a collection for visualization purposes. Instead of storing identical data in multiple collections in multiple instances of **Regr**, the program stores data into an internal collection of this object, reducing the memory occupied on the heap. The most important method in this class is the `append` method, which appends a datapoint into the internal collection. The object also has useful methods to help the plotting process. **LinRegress**, a subclass of **Regr**, models linear regression. Similarly, **QuadRegress** is a subclass of **Regr** and models quadratic regression. The **Preprocessor** object models cleaning the data. Currently, it rescales the data to a value between $[-100, 100]$ (if the user wants to do so) with the method `minMaxScalerOne`.

The `regression.io` package contains the following classes: **Reader**, **CsvReader**, and **JsonReader**. **Reader** is an abstract class describing the concept of reading data stored in files. It is the base class for all file reading classes. It has a private variable called `_errorCode`, which describes the potential exceptions raised during file reading. Its default value is 0, meaning there were no errors in an earlier reading process. A positive value means there was an error. **Reader** has two abstract methods which are to be implemented

by its subclasses: `init` (initialize) and `readline`. `CsvReader`, a subclass of `Reader`, models reading CSV files. Similarly, `JsonReader` is a subclass of `Reader` and models reading JSON files. They both implement the `init` and `readline` methods. The method `init` is used to initialize the reader from a filename and the names of the input and output variables. The `readline` method returns a datapoint that was read from the file. This method and the methods in the `Preprocessor` object and the `Regr` class as well as its companion object has an intimate relationship: `readline` returns a single datapoint as a `Tuple2` of `Double` values. This datapoint is the parameter of the `fit` method in `Regr` class which will then update the state of the instances of the class. The datapoint is also a parameter of the method in the `Preprocessor` singleton object which returns a rescaled version of the datapoint. This is an iterative process: the `readline` method will stop when it reaches the end of the file, when an exception is thrown, or when the required number of passes through the file is met. The algorithm will be further explained below. The `readline` method also handles missing or corrupted datapoints: it simply returns a sentinel `NaN` value, which will be ignored by other methods.

The third package, `regression.ui`, contains the following classes: the app object `RegressionApp`, `DataTable`, `GraphPanel`, and `GraphContainer`. `GraphPanel` is the class modelling a “graph canvas” on which datapoints and the fitted curve are visualized. All methods in this class are private because they are not intended to be used publicly. However, noticeable methods are coordinate transforming methods and drawing methods. In particular, the `scatter` method draws the datapoints onto the canvas. The collection in the `Regr` companion object exists solely for this method. Each time a plot is generated, i.e., an instance of `GraphContainer` is created, it receives an immutable copy of the collection. This enables the user to see the plots of different data files at the same time, as well as viewing the fitted line and the fitted curve (which are visualized on separate plots, i.e., tabs). The `bestFit` method draws a line/curve of best fit to the data based on the computed coefficients. The `GraphContainer` class is a wrapper for the `GraphPanel` canvas. It represents the plotting window, allowing the user to interact with the plot via buttons and mouse. The `DataTable` class is simply a class that represents a data table showing data in number format. This class can be used to see which datapoints are to be visualized. `RegressionApp` represents the main graphical user interface. It contains the buttons and methods that interact with the user. The GUI is a frame with tabs. Each of these tabs is an instance of either `DataTable` or `GraphContainer` (which contains an instance of `GraphPanel`). All of these classes work together to interact with the user.

This structure of classes is actually straightforward: there was not much modification comparing to the initial project plan. The only major modifications affect the implementations of the classes themselves, not the relationship between them. This will be further explained below. An alternative class structure would be for the regression classes to handle file reading themselves, eliminating the need for file reading classes. However, this is a bad solution because it not only makes the code harder to read, but it is also not a good way to modularize the application. This solution will fail if the program needs to be able to read files other than CSV and JSON, or if it needs to implement other regression methods. The code for file reading will thus be needlessly duplicated.

4. Algorithms

This project implements the basic linear regression algorithm as well as a quadratic regression algorithm. The coefficients of these models are solutions of the least squares problem. In the linear regression case, the coefficients are given by the following formulas: (Wikipedia)

$$m = \frac{s_{x,y}}{s_x^2} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}, c = \bar{y} - m\bar{x}$$

where s_x^2 and $s_{x,y}$ are variance and covariance respectively, \bar{y} and \bar{x} are the mean of the y and x variables respectively.

In the quadratic regression case, the coefficients of the parabola are given by the following formulas: (Arizona Department of Health Services):

$$y = \alpha + \beta x + \gamma x^2$$

$$\gamma = \frac{S(x^2 y) \cdot S(xx) - S(xy) \cdot S(xx^2)}{S(xx) \cdot S(x^2 x^2) - S(xx^2)^2}$$

$$\beta = \frac{S(xy) \cdot S(x^2 x^2) - S(x^2 y) \cdot S(xx^2)}{S(xx) \cdot S(x^2 x^2) - S(xx^2)^2}$$

$$\alpha = \frac{1}{n} \left(\sum_{i=1}^n y_i - \beta \sum_{i=1}^n x_i - \gamma \sum_{i=1}^n x_i^2 \right)$$

$$S(x^2 y) = \sum_{i=1}^n x_i^2 y_i - \frac{1}{n} \sum_{i=1}^n x_i^2 \sum_{i=1}^n y_i$$

$$S(xx) = \sum_{i=1}^n x_i^2 - \frac{1}{n} \left(\sum_{i=1}^n x_i \right)^2$$

$$S(xy) = \sum_{i=1}^n x_i y_i - \frac{1}{n} \sum_{i=1}^n x_i \sum_{i=1}^n y_i$$

$$S(xx^2) = \sum_{i=1}^n x_i^3 - \frac{1}{n} \sum_{i=1}^n x_i \sum_{i=1}^n x_i^2$$

$$S(x^2 x^2) = \sum_{i=1}^n x_i^4 - \frac{1}{n} \left(\sum_{i=1}^n x_i^2 \right)^2$$

In the linear regression case, in order to calculate sample variance and covariance, an iterative algorithm was used (Wikipedia). This means that the program iterates through the file and iteratively updates the means, the sums of squared difference from the mean with each single new datapoint. The formulas are:

$$dx = x - \bar{x}$$

$$dy = y - \bar{y}$$

$$\bar{x} = \bar{x} + \frac{dx}{n}, \bar{y} = \bar{y} + \frac{dy}{n}$$

$$M_{2x} = M_{2x} + dx^2 \frac{n-1}{n}, M_{2y} = M_{2y} + dy^2 \frac{n-1}{n}$$

$$n = n + 1$$

where M_{2x} is the sum of squared difference from the mean $\sum (x - \bar{x})^2$, M_{2y} is the sum of squared difference from the mean $\sum (y - \bar{y})^2$, x is the x-component of the current datapoint, y is the y-component of the

current datapoint and n is the length of the data up until the current datapoint. M_{2x} is used in the formula for the slope, and M_{2y} is used to calculate the coefficient of determination (R^2 score). Similarly, the covariance (or the sum $C = \sum_{i=1}^n (x_i - \bar{x}_n)(y_i - \bar{y}_n)$) is iteratively updated:

$$n = n + 1$$

$$dx = x - \bar{x}$$

$$\bar{x} = \bar{x} + \frac{dx}{n}, \bar{y} = \bar{y} + \frac{dy}{n}$$

$$C = C + dx \cdot (y - \bar{y})$$

The program calculates all of this in the first pass through the data file (or the first “epoch”). Therefore, after only one “epoch”, the slope and the y-intercept are ready to be calculated:

$$m = C/M_{2x}, c = \bar{y} - m\bar{x}$$

The program only iterates through the file once if the user chooses “No” in the “Visualize all data...” option. This is because the regression result metrics as well as the algorithm for choosing data to visualize (and rescaling with min-max scaler) all require at least two passes through the data. The formula for R^2 is (Wikipedia): $R^2 = \frac{SS_{reg}}{SS_{tot}}$, where $SS_{tot} = M_{2y}$, $SS_{reg} = \sum_{i=1}^n (mx_i + c - \bar{y})^2$. m and c are only available after

the first epoch. Root mean squared error formula is $\sqrt{\frac{\sum (y_i - (mx_i + c))^2}{n}}$. In addition, rescaling the data requires three epochs because in the first epoch the program calculates the minimum and maximum values of the data (since it is the min-max scaler), in the second it updates the sums, and in the third it updates the metrics. If the user does not want to rescale the data, the program takes at most two epochs to finish: the first epoch is used to update the sums, and the second epoch is used to update the metrics.

In the quadratic regression case, the sums (the sigmas in the formulas) are also iteratively calculated and are ready to be used in the formulas for the coefficients after the first epoch. Similarly, the second epoch is used to update the metrics such as root mean squared error. The idea behind the whole algorithm is similar to the idea behind Stochastic Gradient Descent, except that SGD may require many epochs while the algorithm of this program only needs one or two epochs. Perhaps everything can be done within the first epoch if normalization and regression metrics are not implemented, and if every single datapoint were stored into a collection for visualization, even if the data file is very large. However, the program was implemented such that it would try to visualize as few datapoints as possible (to reduce memory usage on the heap due to large collections). The decision to choose how many datapoints to visualize could only be made in the second epoch because the length of the data is computed in the first epoch.

The program loads data into internal collections for the sole purpose of visualization. The data collection algorithm is as follows: if the user chooses “No” in the prompt, then the program simply randomly picks at most 1000 datapoints. If they choose “Yes”, the program will get the length of the data in the first epoch and decide how many datapoints to load in the second epoch based on this length, as described above. No matter what the user chooses, the program always loads (randomly with a probability of 50 percent) at most 1000 datapoints in the first epoch. Therefore, if the user chooses “Yes”, the program will store at least 1000 datapoints if the file contains more than 1000 datapoints. For example, if the dataset contains 1000001 datapoints, the program will load about $1000 + 1\% \cdot 1000001 \approx 11000$ datapoints. It is impractical to visualize all datapoints when the dataset contains millions, even hundreds of millions datapoints. Because it is unknown how large the test files will be, the program was implemented with such restrictions, which

are reasonable. Considering files with more than 1 million datapoints, even the lowest number of randomly selected datapoints, 11000, is already visually representative of the data. Of course, one can always change the limit from 1 million to something larger, e.g., 10 million, and change the probability of loading a datapoint from 1 percent to, e.g., 10 percent. Because this program implements plotting with panning and zooming features, speed is crucial. The fewer datapoints the program has to visualize, the faster it will run, and the less annoyed the user will be. If panning and zooming were not implemented, even if millions of datapoints are visualized, the user cannot make out what is what: the dots will take up the whole viewport.

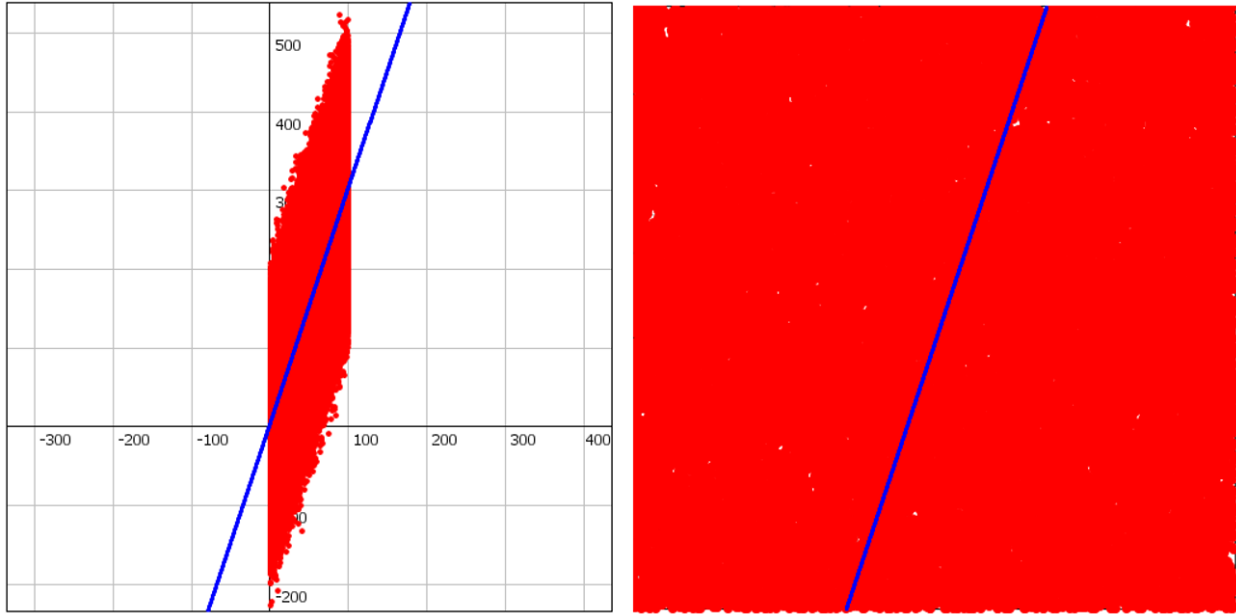


Figure 4. 5 million datapoints visualized.

Again, this restriction is only for datasets with more than 1000000 datapoints. Smaller datasets would not be a problem. The whole point of the iterative algorithm for computing regression coefficients is to avoid using very large internal collections that will throw an `OutOfMemoryError` exception and to avoid slow performance when the heap is almost full because of these large collections.

The rescaling algorithm is also iterative: the rescaling method will rescale a single datapoint. Rescaling, or normalization, is based on the following formula (Wikipedia):

$$x' = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)}$$

One may think normalization is useless because it transforms the original data into something different and rescaled data cannot visually represent the original data. However, normalization preserves the data structure, meaning if the true shape of the data is a parabola, then the normalized data will also be in the shape of a parabola. Moreover, the shape of the data may even be clearer after normalization.

To recap, the program fits a line and a parabola to the data in the following procedure:

The action the program takes in the epochs depends on whether the user wants the data to be rescaled or not. If the user ticks the checkbox, then the program will spend the entire first epoch to figure out the maximum and minimum values of the variables in the data. If not, then the program will spend this epoch iteratively calculating the sums (`update_sums`) in the formulas above. In all 3 epochs, no matter what the

user chooses, the program will load with probability 50% at most 1000 datapoints for visualization. In the second epoch, if the user chooses “Yes” in the “Visualize all...” prompt, then the program will calculate the metrics (`update_metrics`) and load data with restrictions described above, or only update the sums and randomly load 1000 datapoints if the user additionally ticks the rescaling checkbox. If the user chooses “No” but ticks the checkbox, then the program will calculate the sums as well as randomly loading 1000 datapoints. If the user both chooses “No” and does not tick the checkbox, then there is no need for a second epoch. There is only a third epoch if the user both chooses “Yes” and ticks the checkbox. In this epoch, the regression metrics are updated, and data is loaded with restrictions. The reason the decision to load the whole dataset cannot be made in the first epoch is that length of the dataset is still unknown in the first epoch. The final outcome is as follows. If the user chooses “No”, they will not see regression metrics and can only see at most 1000 datapoints visualized. If the user chooses “Yes”, they will see regression metrics and all data if the dataset has fewer than 1 million datapoints, or $1000 + 1\% \cdot n$ datapoints if the length of the dataset n is larger than 1 million. If the user ticks the checkbox, they will see the normalized data.

On the other hand, the visualization algorithm is rather simple. The only crucial step is to transform the coordinates of the datapoints to the coordinate system that the computer uses. The datapoints are drawn in a functional style: they are stored in a collection, and the `scatter` method traverses the collection with a for expression, drawing datapoints one by one. The collection is formed as described above.

The file reading algorithm is also rather simple. The program reads and parses one datapoint at a time. Reading all of them requires a loop structure. If a datapoint is corrupted (e.g., missing values in CSV files), a sentinel value (NaN) is returned, and the program will ignore that datapoint. This iterative reading algorithm was implemented so that it can work with the iterative regression algorithm. JSON reading was implemented with the `jackson-databind` external library because of its speed and efficiency.

An alternative solution to the iterative regression algorithm is to first load all data into collections and then calculate the required values in a functional style using methods on collections. Earlier versions of this program were implemented in such a way; however, a problem occurred when the length of the dataset exceeded 15 million. An `OutOfMemoryError` exception was thrown because the program generated too many collections with huge sizes. Another advantage of the iterative algorithm is that it can be run in another thread and the user can interrupt this thread after each iteration. This is not possible with the functional algorithm because it handles everything in one go, and there is no location to interrupt the process. This kind of functionality (interrupting a thread) was inspired by and only possible thanks to the source code of a project from the Programming 2 course (Kaski). Therefore, the algorithm was changed from operations on collections to iterative operations on the file itself. I also tried Stochastic Gradient Descent and its variants but could not make it work.

An alternative solution to the visualization of the datapoints problem is to simply do not load data into collections and instead iteratively visualize the datapoints straight from the file. This solution, however, disregards the normalization functionality of the program. While this solution solves the problem of not having enough heap space to store collections, it sacrifices a bit of speed because parsing a single datapoint from a file and then draw it takes more time than taking the datapoint from a collection and draw it. One can also try drawing the datapoints in another thread, but I do not think that is a good solution.

An alternative solution to file reading is to read and store everything into collections instead of returning individual datapoints. This is a bad idea because of the reasons mentioned above. In fact, early versions of this program implemented this solution, but heap space was filled up very fast when the input size got very large. When heap space is almost full, the program runs very slowly.

5. Data structures

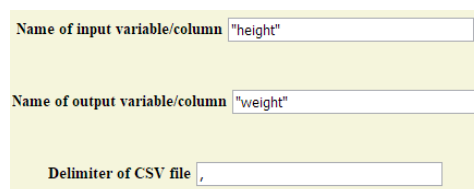
The fitting part of the program does not use collections. An exception is that the return value of the coefficient method is a `Vector` of `Double`-valued coefficients. The visualization part uses a `Vector[(Double, Double)]` converted from a mutable `HashSet[(Double, Double)]` which is a local variable of the `Regr` companion object. `HashSet` was chosen to collect and store the data for visualization because it does not allow duplicate elements. This functionality is required because the program first randomly loads 1000 datapoints into a `HashSet` (by appending to it), and then optionally load more data into the same `HashSet`, some of which may be the same as those already loaded from the 1000 batch. Also, `HashSet` was chosen instead of other kinds of mutable `Sets` because it has the best performance and there is no need to maintain the insertion order. This is why in the “Index” column of the data table tab in the program, there is a disclaimer which says the data is not in original order. `Vector` was chosen simply because it is immutable, and each instance of `GraphPanel` class should keep drawing old data even if the source of the data is changed. The user can simply press the plot buttons to generate plots of new data. In this way, the user can, e.g., compare between two datasets. The `DataTable` class uses an `Array` of `Arrays` of `java.lang.Object` elements (`java.lang.Double`) because this structure is required by the `JTable` class which was used to implement `DataTable`. Another option would have been to not use data collections at all and treat a datapoint as a `Tuple2` or variables for each component of the datapoint. This option would be a good idea if the alternative solution to the visualization of the datapoints problem were implemented. Also, another solution would be to create a case class that represents the concept of a datapoint.

6. Files and Internet access

This program only deals with CSV and JSON files (which contain the dataset) as inputs. According to Wikipedia, a comma-separated values (CSV) file is a delimited text file that uses a comma to separate values. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. The use of the comma as a field separator is the source of the name for this file format. A CSV file typically stores tabular data (numbers and text) in plain text, in which case each line will have the same number of fields. However, this program also accepts other kinds of field separators (referred to as delimiter in the program), provided that the separator is “standard” and distinguishable from everything else in the file. Also, this program requires that the first line of the file contain the names of the columns/variables, and that the user should know these names and supply them to the program (case-insensitive but needs to be correct). An example is:

```
1 "Gender","Height","Weight"
2 "Male",73.847017017515,241.893563180437
3 "Male",68.7819040458903,162.310472521300
4 "Male",74.1101053917849,212.7408555565
5 "Male",71.7309784033377,220.042470303077
6 "Male",69.8817958611153,206.349800623871
```

Figure 5. An example CSV file.



The screenshot shows a light yellow rectangular area containing three input fields. The first field is labeled "Name of input variable/column" and contains the text "height". The second field is labeled "Name of output variable/column" and contains the text "weight". The third field is labeled "Delimiter of CSV file" and contains a comma character ",".

Figure 6. Example input.

In Figure 5, the first line contains the column names. The user should know these names and supply them to the program. For example, the “input column” field can be “height” or “weight” (including the quotation marks). The same applies to the “output column” field. Because the fields are separated by commas, a comma should be keyed in to the “Delimiter of CSV file” field. The CSV file can contain any number of columns, as long as there are two columns with numeric values.

According to Wikipedia, JavaScript Object Notation (JSON) is an open standard file format, and data interchange format, that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value). This program accepts JSON files in the following format:

```
1  [
2  {" X ":17.8054769639," Y ":77.7161310136},
3  {" X ":21.1989709711," Y ":29.6075280092},
4  {" X ":7.4100548596," Y ":58.9694322167}
5  ]
6
7  // or:
8
9  [{"x":17.805,"y":77.716},{"x":7.416,"y":58.97}]
```

Figure 7. An example JSON file.

One can identify this format as an array of objects, each of which contains two fields with Double/Integer values. The names of the variables in this case are the fieldnames in the JSON file (x and y but with no quotation marks because they are a part of the JSON structure).

Attached with the source code are two Python scripts which can be used to generate CSV and JSON files for testing purposes. If one wants to create own Python scripts, one should follow the settings in the given scripts when coding the writing out to a file part. If one wants to generate the data in some other way, one should be able to generate the files according to the formats described above.

7. Testing

The program was tested manually. Unit tests are only good if the tests can be automated. This program cannot be tested in this way because the regression coefficients cannot be reliably calculated in unit tests and writing methods for calculating unit tests amounts to implementing the algorithm again. Nevertheless, there were some “unit tests” in the sense that 2 regression algorithms were implemented. The functional algorithm (the alternative solution mentioned above) was a verbatim copy of formulas and thus could not be incorrect. The iterative algorithm was tested by comparing its solutions with the functional algorithm. In addition, some datasets already fitted with a line or a parabola was also inputted, and the results were compared to the model solutions. The results of the algorithm were also compared with those computed by Python scripts. Border cases such as datasets with fewer than 3 datapoints and very large datasets were also tested. The regression algorithm passed all tests. It is worth mentioning that the iterative algorithm replaced the functional algorithm because of one of the tests in which the dataset was too large for the functional algorithm to handle.

File reading was carefully tested. Empty files, nonexistent files, broken files, large files, files with wrong formats, and other cases were manually tested. The program passed all tests.

The visualization module was also carefully tested. All known bugs and possible bugs are reported below. This part was the most difficult to test because there were a lot of border cases, and there is no room for automated testing.

8. Known bugs and missing features

Almost all of the known bugs and missing features lie within the visualization module. There is one bug with the GUI itself: occasionally, after the program is terminated, an “Exception while removing reference” is thrown. I have no idea what caused the exception (this was the entire error message), nor do I have any clue how to fix it. I tried some suggestions on StackOverFlow, but none of them worked.

One bug is that the labels on the x-axis and y-axis are far from being displayed perfectly. For example, when the scale gets very large or very small (e.g., labels with 6 digits), the strings become very long and may overlap with each other, making it difficult to read the numbers. Also, sometimes a part the label (especially the labels on the right edge of the plotting area) will be drawn out of the plotting area. In addition, sometimes the labels (two to be exact) on the right edge will be drawn on top of each other. This labelling problem exists because it is very difficult to code the rules for when to draw the labels. Another problem is with the visualization of the fitted line or the parabola. Because the line and the parabola are formed by a series of dots, the line/parabola will just become a series of dots if the line is too steep or if the parabola is too “narrow”. A possible bug is that when the zoom level is very small ($1E-7$ small) or very large ($1E+100$ large), there may be problems with the plot, such as gridlines and labels being inaccurately displayed, or the program may freeze. The program may also freeze when the user zooms in to very small zoom levels when the position is very far from the origin (coordinates like $(1E+8, -1E+8)$). This is because of division by zero or integer overflow or floating-point precision problem. I do not know what causes this problem. Be careful when zooming to very large or very small values or when one is very far away from the origin.

One (possibly) missing feature is the fact that the user cannot configure the step sizes or the endpoints on the x-axis or the y-axis. This feature is illustrated in the below figure from Desmos. This feature is missing because I could not make it work properly, so I did not include it in the final product. The reason I could not make it work is that the way the program draws is not based on the endpoints of the axes, but rather it is based on the width and the height of the viewport and the current (zoom level). This is not an ideal solution. In addition, although one of the requires of the project is that “The appearance of the graph, such as the endpoints of the axes, can be configured by the user,” I interpreted the “such as” verse of the sentence as a *suggestion*, not a requirement, for the “the appearance of the graph can be configured”. Therefore, I thought this feature was optional and did not try my best to implement it. Also, I do not think it is necessary to implement this feature because I have already implemented the zooming and panning features which, in my opinion, are much more powerful because they allow the user more freedom to visualize only the parts of the plot that they want to see. Also, zooming and panning are more than enough to complement this missing feature. Even if this feature was implemented, without panning and zooming, interpreting the plots of “big data” would be a challenge to the user because the plots would look like Figure 4.

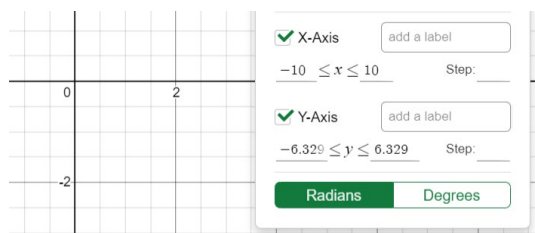


Figure 8. How the user can configure the axes in Desmos.

9. 3 best sides and 3 weaknesses

The first good thing about this program is that it is fast because it combines file reading and calculation into a single iteration instead of having a separate loading step and calculation step. The fastest use case is when the user chooses “No” and unticks the checkbox, making the program only take 1 epoch to return the regression coefficients. The user can also choose to stop the fitting process if they are impatient. This feature is only made possible because fitting happens in a different thread than the event dispatch thread. The second good thing about this program is that it is not constrained by input file size. No matter how large the input file is (even gigabytes worth of data), the program can easily iterate through the file and return correct regression coefficients, if the user is patient enough. The third good thing about this program is that the user can zoom and pan and interact with the plots using the mouse.

One drawback is that the working logic of this program is needlessly complicated (restrictions on how many datapoints to visualize) for such a straightforward topic. This is because I do not know how large the test files will be. If they are too large (hundreds of millions of datapoints large), the program will take forever to draw a plot with all of the datapoints. Unfortunately, I do not know any quick fix to this. Fortunately, the user interface is very simple, and the user should only be aware of the user interface and not be aware of why the program made the decision to visualize this number of datapoints. The third problem with this program is the zooming and panning functionalities. While these features work, they do not deliver a good enough experience compared to, e.g., Desmos or GeoGebra. For instance, it is difficult to precisely pan with the mouse. Another weakness of this program is the fact that the user is unable to configure the axes to their liking. Again, I do not really consider this to be a shortcoming because I do not like this feature and never use it in other graphing programs.

10. Deviations from the plan, realized process and schedule

The biggest deviation from the plan is that there were no unit tests that was coded. All tests were done manually, or semi-automatically (generate random data, use the same data for two different algorithms, compare results).

Another deviation from the plan is that JSON reading was implemented before CSV reading because I felt that it is more difficult to implement JSON reading. Also, the performance problem with the functional algorithm was only discovered and solved very late and required a major overhaul of the code.

Apart from those deviations, the schedule estimated in the project plan matched quite closely with reality. The algorithms were implemented first, and the GUI was implemented later. A timeline of the project is as follows: in the first two weeks, only the regression algorithms (the functional one) and JSON reading was implemented. In this stage, JSON reading was implemented with a library different from the current one. This library gave very poor performance, so it was decided to change libraries. In the next two weeks, CSV reading and a basic GUI was implemented. Graphing functionalities were also implemented during this time. Zooming and panning was implemented in the next two weeks, along with a more complete GUI and improved graphing functionalities. It was in the final days of these two weeks that the performance problem was discovered. In the remaining weeks of the timeline, the iterative algorithm was implemented and replaced the functional algorithm. File reading was also changed from loading everything to collections in one go into iteratively parsing the file and returning individual datapoints. GUI also received a big update in this phase: it became cleaner and had fewer buttons. The time spent into this project increased linearly with each passing week. I spent almost all of the week after the exam week in period 4 into this project. I think this order of progress is not well thought out because I did not spend much thought into this project

in the beginning weeks. Had I spent more time planning things out, I would not have encountered the performance problem. I would have just implemented the iterative algorithm right from the beginning.

What I learned from this project, besides the domain-specific knowledge relevant to my field of study, is how to properly plan a project and follow the plan closely. I also learned various things about programming such as performance and class structure. Overall, this project was a great learning experience.

11. Final evaluation

In summary, the good aspect of the program is the regression part. In contrast, the bad aspect lies within the visualization part. The final product is thus a simple program which can reliably fit a line and a parabola to a dataset and can somewhat decently visualize the results along with the original datapoints. In the future, the visualization aspect of the program can be improved if I spend more time studying computer graphics. The current solution methods for the two regression algorithms are good enough for the requirements of this project. Of course, better and more general algorithms exist for this kind of regression problem. For example, I could implement a general polynomial regression method instead of two separate methods. The class structure of the program facilitates making changes like this because each module is a separate entity and is barely dependent on each other. If I were to start the project again from the beginning, I would spend more time planning out and thinking clearly about my next steps in order for everything to be smooth sailing. Also, I would spend more time on the visualization module and try implementing the missing feature because visualization is clearly the more challenging aspect of this project topic.

12. References

Arizona Department of Health Services (n.d.). Quadratic Least Square Regression. <https://www.azdhs.gov/documents/preparedness/state-laboratory/lab-licensure-certification/technical-resources/calibration-training/12-quadratic-least-squares-regression-calib.pdf>

jackson-databind library. <https://github.com/FasterXML/jackson-databind/>

Java Swing Library. <https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>

Kaski, Petteri (n.d.). The “armlet” architecture. Retrieved from Round 5 of the Programming 2 course.

Scala Standard Library. <https://www.scala-lang.org/api/current/scala/index.html>

Scala Swing Library. <https://github.com/scala/scala-swing>

Wikipedia (n.d.). Algorithms for calculating variance. https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance

Wikipedia (n.d.). Coefficient of determination. https://en.wikipedia.org/wiki/Coefficient_of_determination

Wikipedia (n.d.). Comma-separated values. https://en.wikipedia.org/wiki/Comma-separated_values

Wikipedia (n.d.). Feature scaling. https://en.wikipedia.org/wiki/Feature_scaling

Wikipedia (n.d.). JSON. <https://en.wikipedia.org/wiki/JSON>

13. Appendices

The source code, the two Python scripts, and a sample CSV dataset are included in the archive file. The below figures show some example use cases of the program.

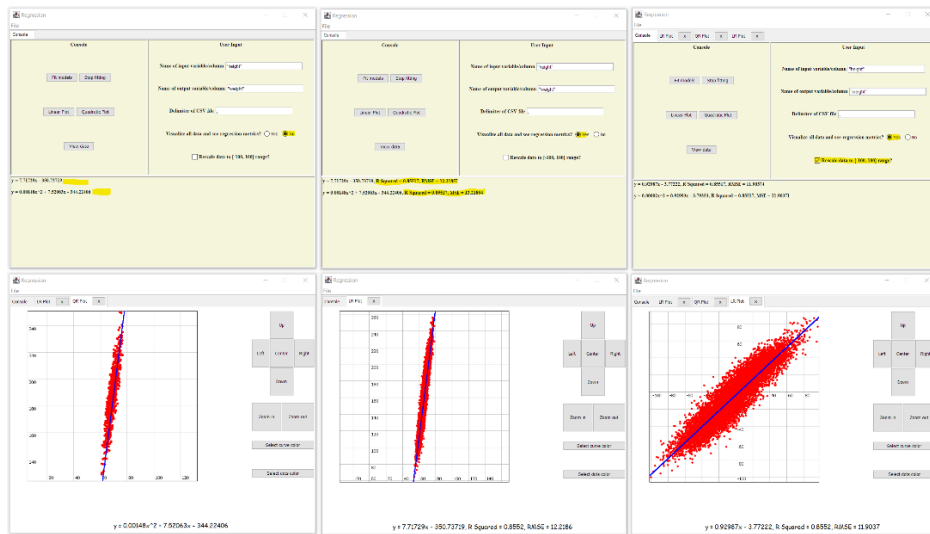


Figure 9. How different options affect the result.