Assignment report

Course: Programming with Python

Author: Maurice ten Koppel

Date: 3. September 2020

Table of contents

1. Introduction

Within the body of this document I will provide an in-depth explanation of the structure and rationale behind a Python program written to solve the assignment for course “Programming with Python (DLMDSPWP01). In the appendix, the full source can be found. A public git-repository is also provided at: https://github.com/mtenkoppel/assignment-programming-with-python

I will zoom in on 10 key aspects that explain to the reader in further detail the critical aspects of the software written.

This is followed up with a conclusion with retrospection on the limitations and improvements that could be made to the software.

2. The assignment and its algorithms

The assignment can be interpreted into two parts. In the following they are described in text.

2.1 Part 1: determining the best fitting ideal function

In part 1 the goal is to find for a *training function* the best fitting *ideal function* amongst 50 candidates. There are a total of 4 different training functions and for each, the ideal function has to be found. A function is a collection of x- and y- coordinates and is provided for in a .csv file. Within the 50 candidates, the ideal function which has the lowest squared error towards the training function, is defined as the *ideal function.* This in essence is a variation of the “Mean squared error” and is a popular loss function towards models are optimised (Kerzel, 2020).

The Squared Error has a couple of properties

1. Because the deviation is squared we always have a positive result
2. Large deviations have a strong impact

I now describe in written words how the algorithm which I implemented for part 1 works. The implementation itself can be found in chapter 5.

For each training function, go over each point and calculate the deviation of the y-value towards a candidate ideal function. Square the deviation and sum all up, this value is defined as Squared Error. Do this for all candidate functions and the function which has the lowest Squared Error, is the *ideal function*.

The result is thus 4 ideal functions.

2.2 Part 2: classifying points

In part 2, a collection of points is provided for as .csv file, and for each point it needs to be determined if it can be assigned towards one of the 4 ideal functions. In addition, if a match is made the deviation has to be computed.

Again, in words how the assigning of points in the implementation works. Implementation details can be found in chapter 6.

For each point within the test data collection, compute the **absolute linear deviation (ald)** to each of the ideal functions. Determine for each deviation if it is within lower than the tolerance. **If multiple fit, pick the classification with the lowest ald**. The computation of the tolerance is given within the assignment, and equals to the *largest deviation* between training- and ideal function multiplied with sqrt(2).

2.3 Storing of data using SQLite

In addition to the computation, data has to be written towards a SQLite database. Three databases have to be generated.

1. A database which mirrors the training data set
2. A database which mirrors the candidate ideal functions data set
3. A database which stores the classification towards the ideal functions together with the deviation

Ad 3: The assignment did not provide any detail towards what should be saved if no classification can be made. Furthermore, if no classification can be made the deviation cannot be provided for either. The program writes in the case of no classification “-” within the “No of ideal func” column and “-1” within the “Delta Y (test func)” column.

2.4 Other requirements

Within the assignment, there are further requirements that impact the design of the program significantly. Foremost, it has to demonstrate an object-oriented design and use packages such as Panda, Bokeh and SQLAlchamy. As conclusion, the point is clearly not to simply calculate and solve the assignment but to demonstrate knowledge of Python and popular packages for data science purposes.

3. Software design choices

Given the requirements and the algorithms I made some choices on how to design the software. My goal is to make these explicit so that the reader can better understand the result.

1. **Object-oriented design**. Pandas is Python package that can read csv files and amongst others do quick computations with the data (cf. pandas). To solve the assignment, few clever lines with help of the Pandas packages would suffice. Instead though, the purpose of the assignment is to demonstrate object oriented design. This heavily impacted the design.
2. **High level readability**: My attempt is to provide high reliability on the highest level of code to clearly communicate the sequence of steps. As a consequence, it can get quite deep until the "real" algorithm is doing its work. This might feel atypical for python *(unpythonic)* as one of the style guidelines is to keep structures flat (cf. Peters).
3. **Heavy documentation within the code**: Doc strings and comments are long and descriptive. As consequence, it might negatively impact the readability of the code itself.
4. **No focus on performance**: I am willing to accept a mediocre performance for the assignment as the datasets are small. As an example, it includes no threading or other optimisation techniques.

4. Overview of software architecture

The object oriented design is foremost based on the insight that the software has to do a lot of comparisons between x,y-functions. Class *Function* is the fundamental unit for the software and contains a collection of x- and y-coordinates, a name and has some convenience methods. Functions are mass-created by reading three different .csv files. Class *FunctionManager* is in charge of this and makes all Functions available. Functionmanager also contains a method to write data to a SQLite database.

This means all data is easily accessible and different regression methods can be applied. Collected within module *regression* are different functions to do so. Function *minimise_loss* accepts a train Function, a list of candidate ideal Functions as well as a loss function. Loss functions are collected within module *lossfunction* and for now only contains function *squared_error.* This design allows for a flexible exchange of loss functions while keeping the generic procedure of finding an ideal x,y-function with the lowest error seperate.

Method *minimise_loss* returns an instance of class *IdealFunction* which inherits from *Function.* In other words, after a candidate ideal x,y-function has been identified as the closest fit, it is casted to an *IdealFunction*. It stores the original training data and computes a property *tolerance* which is needed to compute the matching of points in a next step.

Finally, module *regression* contains a function *find_classification.* This function takes a point, a list of IdealFunctions. Its purpose is to make a classification and if so calculate the deviation.

In between all this computation different plots are created. Functions are stored within module *plotting*. Lastly, module *utils* contains functions for writing results to a sqlite database. Below you find a visual representation of the software design.

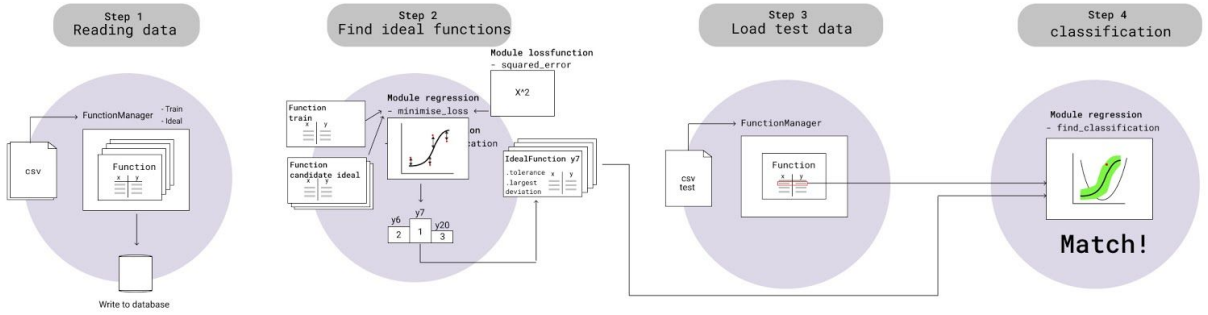

Image 1: a schematic overview of the software architecture. For simplification purposes not all detail is included.

4.1 Class Function (module function)

The backbone of the class is a Panda dataframe in which all coordinates are saved. In essence, a dataframe is a representation of data - similar like a spreadsheet - that allows for easy retrieval and computation. The class makes it easy to find a y-value given an x-value. Furthermore, the class implements Pythons *iterable* protocol (cf. Iterator protocol) which makes it convenient to go over all points within a function. Lastly, the class implements *__sub__* which means two functions can simply be subtracted in order to find the linear distance as dataframe.

4.2 Class FunctionManager (module function)

Given a .csv file this class creates Functions and makes them retrievable. Reading the .csv is handled by Panda's *.read_csv* function. It is important to note that the .csv must follow a specific structure in which the first column represents the x-coordinates and the following columns y-coordinates for different functions. The amount of columns is not important. This class also implements the *iterable* protocol and allows for simple iteration of all functions. It includes a method to immediately write its data to a SQLite database, in a structure which is compliant to the assignment rules. Writing the database is done using Pandas *.to_sql* function.

The result of these class Function and FunctionManager is a convenient way to load functions and to write them to a database (see figure 2).

```
ideal_path = "data/ideal.csv"
train_path = "data/train.csv"

# The FunctionManager accepts a path to a csv and parses Function objects from the data.
# A Function is stores X and Y points of a function. It uses Pandas to do this efficiently.
candidate_ideal_function_manager = FunctionManager(path_of_csv=ideal_path)
train_function_manager = FunctionManager(path_of_csv=train_path)

# A FunctionManager uses the .to_sql function from Pandas
# The suffix is added to comply to the requirement of the structure of the table
train_function_manager.to_sql(file_name="training", suffix=" (training func)")
candidate_ideal_function_manager.to_sql(file_name="ideal", suffix=" (ideal func)")
```

Figure 2: FunctionManager is used to create Functions and write data to a database

4.3 Class IdealFunction (module function)

IdealFunction inherits from *Function* and is used to cast a generic candidate Function to an IdealFunction. In addition to Function it has methods and properties that make classification a breeze: it computes the largest deviation, can handle a custom tolerance factor (√2 in the case of the assignment, default value is 1) and transforms this to a property *tolerance.* Because it inherits from Function also this class is iterable.

4.4 Function minimise_loss (module regression)

The procedure to find a Function that minimises an error is generic and is available within module *regression* as *minimise_loss*. This function accepts an instance of Function that represents the training data and a list of Function instances that represent potential candidates a fit. It also accepts a loss function as argument. This is used to compute the error and find the Function with lowest error. This function returns an instance of *IdealFunction*.

4.5 Function squared_error (module lossfunction)

Within this module different loss functions are stored. For now it only contains *squared_error*. Any loss function accepts two functions and the total summation of error is returned. Figure 3 shows how *minimise_loss* in conjunction with the loss function works.

```
ideal_functions = []
for train_function in train_function_manager:
    # minimise_loss is able to compute the best fitting function given loss function square_error
    ideal_function = minimise_loss(train_function, candidate_ideal_function_manager.functions, squared_error)
    ideal_function.tolerance_factor = ACCEPTED_FACTOR
    ideal_functions.append(ideal_function)

```

Figure 3: A simple iteration over a function manager combined with the creation of IdealFunction objects.

4.6 Function find_classification (module regression)

Function *find_classification* accepts a point and a list of instances of *IdealFunction*. The algorithm implemented will be described chapter 6. Figure 4 shows how the high level code looks like.

```
47      test_path = "data/test.csv"
48      test_function_manager = FunctionManager(path_of_csv=test_path)
49      test_function = test_function_manager.functions[0]
50
51      points_with_ideal_function = []
52      for point in test_function:
53          ideal_function, delta_y = find_classification(point=point, ideal_functions=ideal_functions)
54          result = {"point": point, "classification": ideal_function, "delta_y": delta_y}
55          points_with_ideal_function.append(result)
56
```

Figure 4: Observe (1) how the test data is loaded and (2) how a simple iteration over each point is able to compute a classification towards an ideal function.

4.7 Conclusion on design

In a few high-level lines of code the whole assignment can be solved. The design is built around the unit of a Function, which can easily be generated with the FunctionManager. A design with different modules has been chosen to maintain a clear separation of concerns. For instance, rather than nesting the loss functions within the minimising algorithm, these are available externally and can be passed as argument.

5. Pandas and float conversion inaccuracy

A Pandas Dataframe can be seen as a spreadsheet. More technically put, a DataFrame is a collection of Pandas Series. A Series is within the spreadsheet analogy a column but with a significant difference: its entries must adhere to an agreed data type such as Integers or Floats. Different types are available and Pandas allows for casting between different types. Pandas also has a convenient method to read from a .csv file and convert its data to a dataframe. This also means that Pandas must make an inference to the type of data within the .csv file. One has to be careful with this *automagical* (it happens without the users awareness using heuristics) conversion. In the case of the assignment in which the data contains many decimals, Pandas casted all values to floats. Float is a built-in primitive numerical type of Python and has 64 bit precision (Python Software Foundation (I), n.d.). This sounds impressive but it also means representation errors can occur (Python Software Foundation (II), n.d.). As example 0.1 is represented by 0.1000000000000000055511151231257827021181583404541015625. This behaviour has two consequences in the scope of this assignment:

1. There is a difference in the data from the .csv and inserted by Pandas. See figure 5 for an example.

2. When writing unit tests for float computation, it can be the case that simple math does not seem to make sense. As we will see later on the float accuracy plays a role. Luckily, Python provides an *AssertAlmostEqual* function that can be used for these cases.
3. The calculation of the delta-y property of the classification algorithm has a tiny error

Python also provides a Decimal type that can maintain an exact representation of decimal values. This is the ideal option to take if accuracy is of the most importance. Unfortunately, this cannot simply be used by Pandas. A workaround, in which a Series is of type Object has to be maintained, has to be implemented. (Beepscore, 2018). These workarounds need to be maintained and might not stand the test of time. I therefore decided against this path as I felt that the accuracy is high enough and would not lead to different classification results. The only impact it has, is on the calculation of the deviation when performing the classification.

x	y1	y2	y3	y4
-20.0	-1.2484317	7999.5684	800.00616	-7599.578
-19.9	-0.43193787	7880.8604	792.258	-7484.833
-19.8	-0.39401698	7762.471	784.1513	-7370.614
-19.7	-1.3280147	7645.832	775.8123	-7257.3843
-19.6	-0.6485785	7529.4395	768.28564	-7145.592
-19.5	-0.48129416	7414.965	760.1956	-7034.754
-19.4	-0.9362377	7300.9116	752.33026	-6925.4233
-19.3	-0.5324824	7188.9507	745.1888	-6816.993
-19.2	-0.9737219	7077.861	737.7582	-6709.5996
-19.1	-0.9816949	6968.294	729.2189	-6602.876
-19.0	-1.1095492	6859.1772	721.94476	-6498.494
-18.9	-1.1523664	6751.265	714.1916	-6394.0366
-18.8	-1.412118	6644.9688	707.3265	-6290.816
-18.7	-1.4121236	6539.4053	698.9919	-6189.7173
-18.6	-0.895603	6434.88	692.0502	-6088.6665

	x	y1	y2	y3	y4
0	-20.00000	-1.24843	7999.56840	800.00616	-7599.57800
1	-19.90000	-0.43193	7880.86040	792.25800	-7484.83300
2	-19.80000	-0.39402	7762.47100	784.15130	-7370.61400
3	-19.70000	-1.32801	7645.83200	775.81230	-7257.38430
4	-19.60000	-0.64858	7529.43950	768.28564	-7145.59200
5	-19.50000	-0.48129	7414.96500	760.19560	-7034.75400
6	-19.40000	-0.93624	7300.91160	752.33026	-6925.42330
7	-19.30000	-0.53248	7188.95070	745.18880	-6816.99300
8	-19.20000	-0.97372	7077.86100	737.75820	-6709.59960
9	-19.10000	-0.98169	6968.29400	729.21890	-6602.87600
10	-19.00000	-1.10955	6859.17720	721.94476	-6498.49400
11	-18.90000	-1.15237	6751.26500	714.19160	-6394.03660
12	-18.80000	-1.41212	6644.96880	707.32650	-6290.81600
13	-18.70000	-1.41212	6539.40530	698.99190	-6189.71730
14	-18.60000	-0.89560	6434.88000	692.05020	-6088.66650
15	-18.50000	-1.14458	6331.92970	684.16490	-5989.79830
16	-18.40000	-1.33950	6229.16000	676.78720	-5891.04900
17	-18.30000	-0.61373	6128.83250	670.22520	-5793.11800
18	-18.20000	-0.73187	6028.56800	662.73553	-5696.90770
19	-18.10000	-0.82542	5930.22950	655.42460	-5602.34030

Figure 5: train_data.csv (left) and the dataframe loaded by Pandas (right). In the red circle is an example where there is a representation mismatch. Also notice, how these do not happen for all values.

6. Implementing Least-Square

The objective of the least-squared algorithm is to find a function that minimises the deviation towards the test data. The function that determines the deviation is often referred to as the loss function. In the case of the assignment, this is defined as squared distance but other metrics can be used as well. The result of the loss function is called the error.

In order to compute the error, module *lossfunction* is of assistance. Within it, function *squared_error* accepts two Functions and calculates the error.

In the first step of the implementation, both Functions are subtracted. This is possible because class Function implements *__sub__*. Whenever this is executed, dataframes of the respected instances of Function are subtracted. This means that underneath is an instruction to Pandas. The

returned result is a dataframe. In the second step, the resulting deviation is squared. This is a rather simple operation as Pandas provide very easy manipulation of data: we can just take a column and square it. Pandas translates this instruction by applying it to each value in the column. Finally, the sum of all deviations is computed and returned using Python's native *sum* function.

This has to be done for each candidate ideal function. The iteration constantly checks if the new error is the smallest of all and as such the ideal function can be identified.

The computation itself has two values that are of interest. Foremost, the ideal function identified but also the error is important. I also have to consider that in a next step the largest deviation between the train and ideal function is important. Under these considerations, I decided to return an instance of class *IdealFunction* that stores exactly all these properties. Conveniently, IdealFunction inherits from Function so properties like name come for free.

6.1 Unit testing least squared

The unit test for the squared error loss function covers three cases. The first case, it is checked if a given function 1 and function 2 output the expected result. In the next case, I test if the operation is associative by reversing the arguments. Finally, I test if the error on two equal Functions is 0.

The unit tests for *minimise_loss* function checks if it is able to pick the correct candidate given 3 functions. Moreover, it is checked if the error is computed correctly and the correct training function is stored.

7 Implementing the classification algorithm

In the next part of the assignment, x,y-points have to be classified towards the 4 beforehand defined ideal functions. This can be achieved in a few simple steps:

For loading the test points from the .csv the *FunctionManager* can be used. Normally, the *FunctionManager* loads multiple *Functions* but in this dataset, there is only a single function present. To be correct, the dataset does not contain a real function but it is a collection of unrelated points. However, since iteration over points is already part of a Function, this approach provides all functionality needed. It fits the Python philosophy which is considered a "Duck type" language which states that an object may be used for other purposes it was designed for until it breaks (on an interesting note "duck" comes from: "*If it walks like a duck and it quacks like a duck, then it must be a duck"*) (Geeksforgeeks, 2019).

Within module *regression* function *find_classification* is doing the actual computation. In order to make the computation the function accepts a description of a point as dictionary, a list of instances of *IdealFunction*. It is important to note that the criterion to see if a point is close enough to count as

a classification, is stored within property *tolerance.* Tolerance is a unique attribute of an IdealFunction and is computed on instantiating.

An iteration over all *IdealFunctions* retrieves the matching y-value of the IdealFunction based on the x-value provided by the point. The **absolute** distance between both values is computed. This distance is now compared to the accepted tolerance. Since there can be multiple fits, the algorithm considers the best fit for the ideal function with the lowest error. The *find_classification* returns a tuple with the matching IdealFunction and distance. If no classification can be made *None* is returned for both values.

To conclude: A *FunctionManager* loads the points. A simple iteration over each point uses *find_classification* and all results are stored. These results are in a next step plotted and written to an SQLite database.

7.1 Unit testing find_classification

For unit testing this function a set up of two IdealFunctions is created. With five different points, 5 cases are covered:

1. Simple happy case where there is a clear match with a positive deviation
2. Simple happy case where there is no match
3. Simple happy case where there is a clear match with a negative deviation
4. Whenever a point cannot be located on the Function of the IdealFunction an IndexError exception is expected
5. Tricky happy case in which a point has a match towards multiple IdealFunctions. The expected result is the IdealFunction with lowest deviation

8 Plotting using Bokeh

Visualisation can help understand if the computations made by an algorithm make sense. Different packages to do so are available for Python. Matplotlib (https://matplotlib.org/) is a well known library. If however, the goal is to have charts that offer functionality such as zooming, panning or creating dashboards fewer libraries are available. Bokeh is such a library (Bokeh, n.d.). As it is also recommended by the course, I simply decided to use it and gain first hand experience.

The amount of charts and functionality that Bokeh offers is tremendous. It goes beyond the scope of this report to paint a picture of how Bokeh works. Instead, I will focus on the basic instructions needed to draw the graphs needed for this assignment.

8.1 Plotting IdealFunctions

What I wanted to achieve, is to output a graph that shows the ideal function as line chart with a scatter plot of the training data on top. Moreover, I wanted to output all 4 ideal functions in a single “view”. This would allow me to quickly inspect if the results make sense.

In order to combine graphs, Bokeh offers a layout function. Different types of layouts are preconfigured and examples are column based or grid based. As I wanted to inspect the data carefully, I wanted the graphs to be as big as possible. As such a column based layout is a good choice. The *column* function within Bokeh accepts multiple *figure* objects as arguments and makes these available for outputting. The output can be consumed using *show* (immediately opens the browser) and *output_file* to save the result as local .html file. Both are implemented. The only step left is the actual creation of *figure* objects. The figure itself should be seen as having two parts. The meta level which contains for instance titles and other type of visual elements. And there is the data level which actually plots data. On creating an instance of *Figure,* meta information has to be provided. The created object contains methods for plotting data. I used *.scatter* and *.line.* Luckily, these functions simply accept a Pandas dataframe, as well as some visual properties the plot should have. Since Function objects have all required information, this is a straightforward step. On running the software the output is generated. For the readers convenience these are shown below (figure 6):

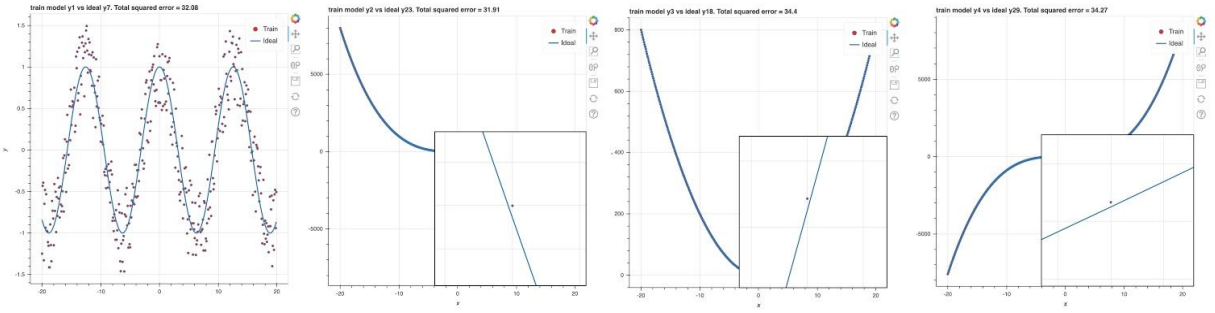

Figure 6: From left to right is train function y1, y2, y3 and y4 as scatter and the ideal function as line. Whereas the deviation in y1 is as visible from a 100% view, this is not the case for the other functions. However, on zooming in on the individual point, it is clear that there is in fact a deviation (shown within the bottom right corner). Also note that this row-visualisation is not the column layout that is generated from the software.

8.2 Plotting classifications

In order to visually check the result of a classification, a couple of elements need to be drawn. (1) the ideal function itself as line (2) the accepted tolerance around the line as surface (3) the point itself. Expected is that the point is within the surface of the tolerance.

Plotting this graph is similar like plotting IdealFunctions: a line and scatter has to be drawn. The challenge in this graph, is to display the tolerance. Luckily Bokeh has a built-in ability to draw this by using a *Band* object. This is an advanced topic and the documentation provided an example I followed. But in essence, the object expects an upper and lower bound and Bokeh takes care of the coloring.

Below some excerpts of the results (see figure 7, figure 8)

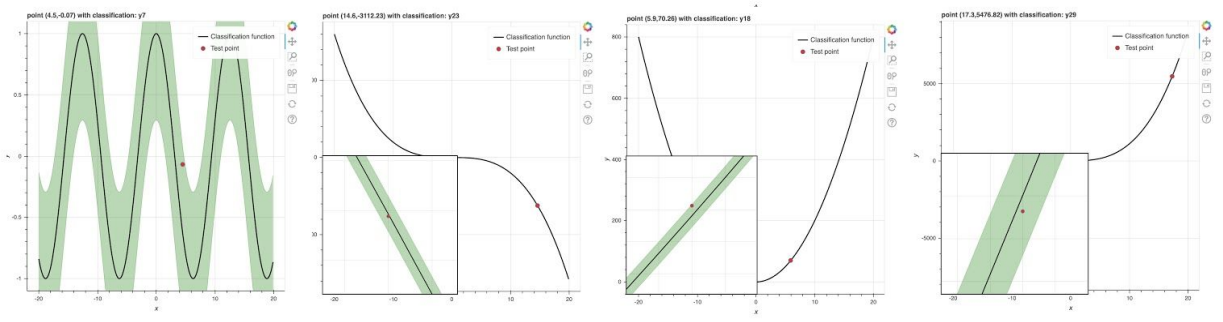

Figure 7: Classification graphs that show (1) the black line as ideal function (2) in green the tolerance of classification (3) the point as red dot. Observe that for 3 out 4 ideal functions, the deviation and band are in the 100% view not visible. When zooming in it becomes clear there is no mistake and there is indeed a small deviation.

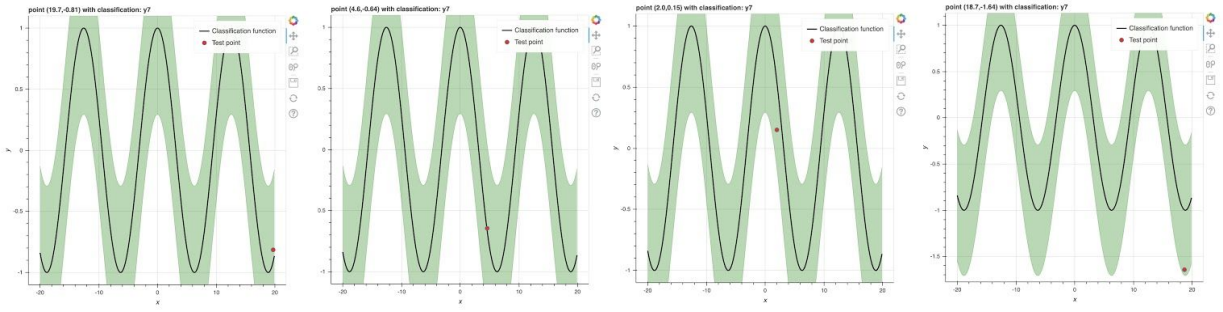

Figure 8: Shown are 4 points that classify towards ideal function y7. These examples show that the band is a useful attribute to immediately spot the correctness of the computation.

9. Writing to SQLite

The software uses two different techniques of writing data to a SQLite database. Both techniques take advantage of SQLAlchemy. Technique 1: using Pandas *to_sql*. This method is available for a dataframe. It however, cannot connect or write itself data and needs an SQLAlchemy engine. This procedure is straightforward. The downside of this approach, is that it only works if the dataframe is to a large extend in structure equal to the desired structure of the database. For writing ideal.csv and train.csv to a database this was the case indeed.

In technique 2, I exclusively used SQLAlchemy to write data. SQLAlchemy uses a concept of Meta Data to create and write to a database without using SQL syntax. To make use of this, one has to describe the table semantically using objects that SQLAlchemy provides. I chose this approach as I am not very familiar with SQL. The code for this procedure can be found in module *utils*.

10. Unit testing

For the unit tests I focussed exclusively on making sure the “algorithms” are well tested. This means that class *Function*, *IdealFunction*, the *regression* and *loss* modules have high test coverage. The description of the test cases for Least-Squared and the classification can be found in Section 5 and Section 6. Overall, the test coverage is 85% over all lines of code. Table 1 shows that the regression and loss function have full coverage.

Class / Module	Coverage
Module regression	100%
Module lossfunction	100%
Module function	59%
Module utils	0%

Table 1: The algorithmic functions have full test coverage.

11. Exception handling

The software does not make high use of exception handling. For production software more work has to be done. I have focussed the effort on providing exceptions that are important for the algorithms to work. These include for instance IndexError exceptions whenever a point cannot be located. This is also covered in the unit tests.

12. Running the software

Download or git from my personal github:

https://github.com/mtenkoppel/assignment-programming-with-python

You may execute main.py without providing any argument. The dependencies are listed in table 2. Table 3 shows which files are generated after execution. Should the script not work, all files are also saved into a directory called output. You may run the unit tests by executing the following command: *python -m unittest* within the main directory.

Dependency	**Version**
Pandas	1.1.0
SQLAlchemy	1.3.19
Bokeh	2.2.1

Table 2: list of required libraries that need to be installed

File	**What**
training.db	All training functions as sqlite database
ideal.db	All ideal functions as sqlite database
mapping.db	Result of point classification test in which the ideal function and its delta is computed
train_and_ideal.html	View the train data as scatter and the best fitting ideal function as curve
points_and_ideal.html	View for those point with a matching ideal function the distance between them in a figure

Table 3: The file names and a description of the content.

13. Conclusion

The design of the software is modular and flexible. These positive aspects come at the price of speed. In initial experimentation, it took me perhaps 2 days to see some basic charts and compute the result. Of course, the final result is now unit tested and covers more functionality and can be extended easily. If the aim would be to develop this software further this is the right approach. If the aim would be, to compute the result and continue in understanding the implications, this is too costly.

The author does not have a computer engineering background and it took quite some iterations to see what the best design would be. Overall, it took me 6-8 days to complete the software. This is of course not the amount that I took studying the python language. I have made myself familiar with Python beforehand.

In retrospect I would make the following improvements:

1. I am not happy having two techniques for writing to a database. It allowed me to learn but is not the best approach to have maintainable software. An improvement would be to write a simple high-level api on top of SQLalchemy.
2. When the classifications are computed, the distance between point and line is computed within the method of classifying itself. In retrospect this should have been a loss function of some kind, which is passed as argument. It would be consistent to how the the ideal function is computed and provides better maintainability.
3. As UX-Designer I know that merely outputting graphs does not make sure that information gets across. Actually, Bokeh as well as its competitor Plot.ly provide extensive functionality to have interactive dashboards. This would allow the viewer to have higher interactivity, experiment and see more information on what is happening. One would also chunk the page so that 30 graphs are not visible on a single page.

I would appreciate the readers feedback on the design and code. Thank you.

References

Beepscore (2018). Using Pandas with Python Decimal for accurate currency arithmetic. Retrieved from https://beepscore.com/website/2018/10/12/using-pandas-with-python-decimal.html

Bokeh (n.d). Documentation. Retrieved from https://docs.bokeh.org/en/latest/index.html#

Geeksforgeeks (2019). Duck Typing in Python. Retrieved from https://www.geeksforgeeks.org/duck-typing-in-python/

Kerzel, U. (2020). Course Book Advanced Mathematics (Version 001-2020-0513). Internationale Hochschule GmbH

Pandas (2020, September 3). Python Data Analysis. Retrieved from https://pandas.pydata.org/

Peters. T. (2020, September 3). Pep 20 -- The Zen of Python. Retrieved from https://www.python.org/dev/peps/pep-0020/

Python Software Foundation (I), (n.d). 15. Floating Point Arithmetic: Issues and Limitations. Retrieved from https://docs.python.org/3/tutorial/floatingpoint.html

Python Software Foundation (II) (n.d.). Iterator Protocol. Retrieved from https://docs.python.org/3/c-api/iter.html

Python Software Foundation (III) (n.d). Built-in Types. Retrieved from https://docs.python.org/3/library/stdtypes.html

Appendix

In the next pages the source code can be retrieved. The first part contains the code itself and in the second part the unit tests. I recommend downloading the source from

https://github.com/mtenkoppel/assignment-programming-with-python

If this does not work you may also contact me under tenkoppel@gmail.com

File - /Users/Maurice/Documents/pycharm/main.py

```python
import math
from function import FunctionManager
from regression import minimise_loss, find_classification
from lossfunction import squared_error
from plotting import plot_ideal_functions, plot_points_with_their_ideal_function
from utils import write_deviation_results_to_sqlite

# This constant is the factor for the criterion. It is specific to the assignment
ACCEPTED_FACTOR = math.sqrt(2)

if __name__ == '__main__':
    # Provide paths for csv files
    ideal_path = "data/ideal.csv"
    train_path = "data/train.csv"

    # The FunctionManager accepts a path to a csv and parses Function objects from the data.
    # A Function  stores X and Y points of a function. It uses Pandas to do this efficiently.
    candidate_ideal_function_manager = FunctionManager(path_of_csv=ideal_path)
    train_function_manager = FunctionManager(path_of_csv=train_path)

    # A FunctionManager uses the .to_sql function from Pandas
    # The suffix is added to comply to the requirement of the structure of the table
    train_function_manager.to_sql(file_name="training", suffix=" (training func)")
    candidate_ideal_function_manager.to_sql(file_name="ideal", suffix=" (ideal func)")

    # As Recap:
    # Within train_function_manager 4 functions are stored.
    # Withing ideal_function_manager 50 functions are stored.
    # In the next step we can use this data to compute an IdealFunction.
    # An IdealFunction amongst others stores best fitting function, the train data and is able to compute the tolerance.
    # All we now need to do is iterate over all train_functions
    # Matching ideal functions are stored in a list.
    ideal_functions = []
    for train_function in train_function_manager:
        # minimise_loss is able to compute the best fitting function given the train function
        ideal_function = minimise_loss(training_function=train_function,
                                       list_of_candidate_functions=candidate_ideal_function_manager.functions,
                                       loss_function=squared_error)
        ideal_function.tolerance_factor = ACCEPTED_FACTOR
        ideal_functions.append(ideal_function)

    # We can use the classification to do some plotting
    plot_ideal_functions(ideal_functions, "train_and_ideal")

    # Now it is time to look at all points within the test data
    # The FunctionManager provides all the necessary to load a CSV, so it will be reused.
    # Instead of multiple Functions like before, it will now contain a single "Function" at location [0]
    # The benefit is that we can iterate over each point with the Function object
    test_path = "data/test.csv"
    test_function_manager = FunctionManager(path_of_csv=test_path)
    test_function = test_function_manager.functions[0]

    points_with_ideal_function = []
    for point in test_function:
        ideal_function, delta_y = find_classification(point=point, ideal_functions=
```

```
ideal_functions)
        result = {"point": point, "classification": ideal_function, "delta_y": delta_y}
        points_with_ideal_function.append(result)

    # Recap: within points_with_ideal_functions a list of dictionaries is stored.
    # These dictionaries represent the classification result of each point.

    # We can plot all the points with the corresponding classification function
    plot_points_with_their_ideal_function(points_with_ideal_function, "point_and_ideal")

    # Finally the dict object is used to write it to a sqlite
    # In this method a pure SQLAlchamy approach has been choosen with a MetaData object to save myself from SQL-Language
    write_deviation_results_to_sqlite(points_with_ideal_function)
    print("following files created:")
    print("training.db: All training functions as sqlite database")
    print("ideal.db: All ideal functions as sqlite database")
    print("mapping.db: Result of point test in which the ideal function and its delta is computed")
    print("train_and_ideal.html: View the train data as scatter and the best fitting ideal function as curve")
    print("points_and_ideal.html: View for those point with a matching ideal function the distance between them in a figure")

    print("Author: Maurice ten Koppel")
    print("Date: 01. September 2020")
    print("Script completed successfully")
```

File - /Users/Maurice/Documents/pycharm/utils.py

```python
from sqlalchemy import create_engine, Table, Column, String, Float, MetaData

def write_deviation_results_to_sqlite(result):
    """
    Can write results of a classification computation towards a sqllite db
    It takes into consideration the requirements given in the assignment
    :param result: a list with a dict describing the result of a classification test
    """
    # In this function we use a native SQLAlchemy approach
    # Rather than using SQL syntax, I decided to use MetaData to describe the table and the columns
    # This data structure is used by SQLAlchemy to create the table
    engine = create_engine('sqlite:///{}.db'.format("mapping"), echo=False)
    metadata = MetaData(engine)

    mapping = Table('mapping', metadata,
                    Column('X (test func)', Float, primary_key=False),
                    Column('Y (test func)', Float),
                    Column('Delta Y (test func)', Float),
                    Column('No. of ideal func', String(50))
                    )

    metadata.create_all()

    # Rather than injecting the values line by line (which is slow)
    # I decided to use SQLAlchemy's .execute using a dict contain all the values
    # The creation of this dict is a simple mapping between the my internal data structures and
    # the structure which is required for the assignment

    execute_map = []
    for item in result:
        point = item["point"]
        classification = item["classification"]
        delta_y = item["delta_y"]

        # We need to test if there is a classification for a point at all and if so rename the function name to comply
        classification_name = None
        if classification is not None:
            classification_name = classification.name.replace("y", "N")
        else:
            # If there is no classification, there is also no distance. In that case I write a dash
            classification_name = "-"
            delta_y = -1

        execute_map.append(
            {"X (test func)": point["x"], "Y (test func)": point["y"], "Delta Y (test func)": delta_y,
             "No. of ideal func": classification_name})

    # using the Table object, the dict is used to insert the data
    i = mapping.insert()
    i.execute(execute_map)
```

File - /Users/Maurice/Documents/pycharm/function.py

```
1  import pandas as pd
2  from sqlalchemy import create_engine
3
4  class FunctionManager:
5
6      def __init__(self, path_of_csv):
7          """
8          Parses a local .csv into a list of Functions. On iterating the object, it
   returns a Function.
9          The functions can also be retrieved with the .functions property
10         The csv needs a specific structure in which the first column represents x-
   values and following columns represent y-values
11         :param path_of_csv: local path of the csv
12         """
13         self._functions = []
14
15         #The csv is being red by the Panda module and turned into a dataframe
16         try:
17             self._function_data = pd.read_csv(path_of_csv)
18         except FileNotFoundError:
19             print("Issue while reading file {}".format(path_of_csv))
20             raise
21
22         #The x values are stored and later on fed into each Function
23         x_values = self._function_data["x"]
24
25         #The next lines iterate over each column within panda dataframe and create a
   new Function object from the data
26         for name_of_column, data_of_column in self._function_data.iteritems():
27             if "x" in name_of_column:
28                 continue
29             # We already stored the x column, we now have the y colum. We can stick
   them together with the concat function
30             subset = pd.concat([x_values, data_of_column], axis=1)
31             function = Function.from_dataframe(name_of_column, subset)
32             self._functions.append(function)
33
34
35     def to_sql(self, file_name, suffix):
36         """
37         Writes the data to a local sqlite db using pandas to.sql() method
38         If the file already exists, it will be replaced
39         :param file_name: the name the db gets
40         :param suffix: to comply to the assignment the headers require a specific
   suffix to the original column name
41         """
42         #Using SQLalchemy an "engine" is created. It handles the creation of the db
   for us if it is not existent
43         engine = create_engine('sqlite:///{}.db'.format(file_name), echo=False)
44
45         # Instead of writing an own implementation and possibly create bugs,
46         # I decided to use functionality from Pandas to write to an sql db.
47         # It only needs the "engine" object from sqlalchemy
48         # Some special care has to be taken to furfill the requirements from the
   assignment on the naming of the columns
49         # In the next lines, the names of the functions are slightly modified to
   comply
50         copy_of_function_data = self._function_data.copy()
51         copy_of_function_data.columns = [name.capitalize() + suffix for name in
   copy_of_function_data.columns]
52         copy_of_function_data.set_index(copy_of_function_data.columns[0], inplace=
   True)
53
```

```
54          copy_of_function_data.to_sql(
55              file_name,
56              engine,
57              if_exists="replace",
58              index=True,
59          )
60
61      @property
62      def functions(self):
63          """
64          Returns a list with all the functions. The user can also just iterate over
    the object itself.
65          :rtype: object
66          """
67          return self._functions
68
69      def __iter__(self):
70          # this makes the object iterable
71          return FunctionManagerIterator(self)
72
73      def __repr__(self):
74          return "Contains {} number of functions".format(len(self.functions))
75
76
77  class FunctionManagerIterator():
78
79      def __init__(self, function_manager):
80          """
81          Used for the iteration of a FunctionManager
82          :param function_manager:
83          """
84          #This simple class which handles the iteration over a FunctionManager
85          self._index = 0
86          self._function_manager = function_manager
87
88      def __next__(self):
89          """
90          returns a function object as it iterates over the list of functions
91          :rtype: function
92          """
93          if self._index < len(self._function_manager.functions):
94              value_requested = self._function_manager.functions[self._index]
95              self._index = self._index + 1
96              return value_requested
97          raise StopIteration
98
99
100 class Function:
101
102     def __init__(self, name):
103         """
104         Contains the X and Y values of a function. Underneath it uses a Panda
    dataframe.
105         It has some convenient methods that makes calculating regressions easy.
106         1) you can give it a name that can be retrieved later
107         2) it is iterable and returns a point represented as dict
108         3) you can retrieve a Y-Value by providing an X-Value
109         4) you can subtract two functions and get a resulting dataframe with the
    deviation
110         :param name: the name the function should have
111         """
112         self._name = name
113         self.dataframe = pd.DataFrame()
```

```
114
115     def locate_y_based_on_x(self, x):
116         """
117         retrieves a Y-Value
118         :param x: the X-Value
119         :return: the Y-Value
120         """
121         # use panda iloc function to find the x and return the corresponding y
122         # If it is not found, an exception is raised
123         search_key = self.dataframe["x"] == x
124         try:
125             return self.dataframe.loc[search_key].iat[0, 1]
126         except IndexError:
127             raise IndexError
128
129
130     @property
131     def name(self):
132         """
133         The name of the function
134         :return: name as str
135         """
136         return self._name
137
138     def __iter__(self):
139         return FunctionIterator(self)
140
141     def __sub__(self, other):
142         """
143         Substracts two functions and returns a new dataframe
144         :rtype: object
145         """
146         diff = self.dataframe - other.dataframe
147         return diff
148
149     @classmethod
150     def from_dataframe(cls, name, dataframe):
151         """
152         Immediately create a function by providing a dataframe.
153         On creation the original column names are overwritten to "x" and "y"
154         :rtype: a Function
155         """
156         function = cls(name)
157         function.dataframe = dataframe
158         function.dataframe.columns = ["x", "y"]
159         return function
160
161     def __repr__(self):
162         return "Function for {}".format(self.name)
163
164 class IdealFunction(Function):
165     def __init__(self, function, training_function, error):
166         """
167         An ideal function stores the predicting function, training data and the
    regression.
168         Make sure to provide a tolerance_factor if for classification purpose
    tolerance is allowed
169         Otherwise it will default to the maximum deviation between ideal and train
    function
170         :param function: the ideal function
171         :param training_function: the training data the classifying data is based
    upon
172         :param squared_error: the beforehand calculated regression
```

```
173         """
174         super().__init__(function.name)
175         self.dataframe = function.dataframe
176 
177         self.training_function = training_function
178         self.error = error
179         self._tolerance_value = 1
180         self._tolerance = 1
181 
182     def _determine_largest_deviation(self, ideal_function, train_function):
183         # Accepts an two functions and substracts them
184         # From the resulting dataframe, it finds the one which is largest
185         distances = train_function - ideal_function
186         distances["y"] = distances["y"].abs()
187         largest_deviation = max(distances["y"])
188         return largest_deviation
189 
190     @property
191     def tolerance(self):
192         """
193         This property describes the accepted tolerance towards the regression in
    order to still count as classification.
194         Although you can set a tolerance directly (good for unit testing) this is
    not recommended. Instead provide
195         a tolerance_factor
196         :return: the tolerance
197         """
198         self._tolerance = self.tolerance_factor * self.largest_deviation
199         return self._tolerance
200 
201     @tolerance.setter
202     def tolerance(self, value):
203 
204         self._tolerance = value
205 
206     @property
207     def tolerance_factor(self):
208         """
209         Set the factor of the largest_deviation to determine the tolerance
210         :return:
211         """
212         return self._tolerance_value
213 
214     @tolerance_factor.setter
215     def tolerance_factor(self, value):
216         self._tolerance_value = value
217 
218     @property
219     def largest_deviation(self):
220         """
221         Retrieves the largest deviation between classifying function and the
    training function it is based upon
222         :return: the largest deviation
223         """
224         largest_deviation = self._determine_largest_deviation(self, self.
    training_function)
225         return largest_deviation
226 
227 
228 class FunctionIterator:
229 
230     def __init__(self, function):
231         #On iterating over a function it returns a dict that describes the point
```

```
232         self._function = function
233         self._index = 0
234 
235     def __next__(self):
236         # On iterating over a function it returns a dict that describes the point
237         if self._index < len(self._function.dataframe):
238             value_requested_series = (self._function.dataframe.iloc[self._index])
239             point = {"x": value_requested_series.x, "y": value_requested_series.y}
240             self._index += 1
241             return point
242         raise StopIteration
243 
```

File - /Users/Maurice/Documents/pycharm/plotting.py

```python
from bokeh.plotting import figure, output_file, show
from bokeh.layouts import column, grid
from bokeh.models import Band, ColumnDataSource


def plot_ideal_functions(ideal_functions, file_name):
    """
    Plots all ideal functions
    :param ideal_functions: list of ideal functions
    :param file_name: the name the .html file should get
    """
    ideal_functions.sort(key=lambda ideal_function: ideal_function.training_function.name, reverse=False)
    plots = []
    for ideal_function in ideal_functions:
        p = plot_graph_from_two_functions(line_function=ideal_function, scatter_function=ideal_function.training_function,
                                          squared_error=ideal_function.error)
        plots.append(p)
    output_file("{}.html".format(file_name))
    # Observe here how unpacking is used to provide the arguments
    show(column(*plots))


def plot_points_with_their_ideal_function(points_with_classification, file_name):
    """
    Plot all points that have a matched classification
    :param points_with_classification: a list containing dicts with "classification" and "point"
    :param file_name: the name the .html file should get
    """
    plots = []
    for index, item in enumerate(points_with_classification):
        if item["classification"] is not None:
            p = plot_classification(item["point"], item["classification"])
            plots.append(p)
    output_file("{}.html".format(file_name))
    show(column(*plots))


def plot_graph_from_two_functions(scatter_function, line_function, squared_error):
    """
    plots a scatter for the train_function and a line for the ideal_function
    :param scatter_function: the train function
    :param line_function: ideal function
    :param squared_error: the squared error will be plotted in the title
    """
    f1_dataframe = scatter_function.dataframe
    f1_name = scatter_function.name

    f2_dataframe = line_function.dataframe
    f2_name = line_function.name

    squared_error = round(squared_error, 2)
    p = figure(title="train model {} vs ideal {}. Total squared error = {}".format(f1_name, f2_name, squared_error),
               x_axis_label='x', y_axis_label='y')
    p.scatter(f1_dataframe["x"], f1_dataframe["y"], fill_color="red", legend_label="Train")
    p.line(f2_dataframe["x"], f2_dataframe["y"], legend_label="Ideal", line_width=2)
    return p


```

```
59 def plot_classification(point, ideal_function):
60     """
61     plots the classification function and a point on top. It also displays the
   tolerance
62     :param point: a dict with "x" and "y"
63     :param ideal_function: a classification object
64     """
65     if ideal_function is not None:
66         classification_function_dataframe = ideal_function.dataframe
67 
68         point_str = "({},{})".format(point["x"], round(point["y"], 2))
69         title = "point {} with classification: {}".format(point_str, ideal_function.
   name)
70 
71         p = figure(title=title, x_axis_label='x', y_axis_label='y')
72 
73         # draw the ideal function
74         p.line(classification_function_dataframe["x"],
   classification_function_dataframe["y"],
75                legend_label="Classification function", line_width=2, line_color='
   black')
76 
77         # procedure to show the tolerance within the graph
78         criterion = ideal_function.tolerance
79         classification_function_dataframe['upper'] =
   classification_function_dataframe['y'] + criterion
80         classification_function_dataframe['lower'] =
   classification_function_dataframe['y'] - criterion
81 
82         source = ColumnDataSource(classification_function_dataframe.reset_index())
83 
84         band = Band(base='x', lower='lower', upper='upper', source=source, level='
   underlay',
85             fill_alpha=0.3, line_width=1, line_color='green', fill_color="green")
86 
87         p.add_layout(band)
88 
89         # draw the point
90         p.scatter([point["x"]], [round(point["y"], 4)], fill_color="red",
   legend_label="Test point", size=8)
91 
92         return p
93 
```

File - /Users/Maurice/Documents/pycharm/regression.py

```python
from function import IdealFunction

def minimise_loss(training_function, list_of_candidate_functions, loss_function):
    """
    returns an IdealFunction based on a training function and a list of ideal functions
    :param training_function: training function
    :param list_of_candidate_functions: list of candidate ideal functions
    :param loss_function: the function use to minimise the error
    :return: a IdealFunction object
    """
    function_with_smallest_error = None
    smallest_error = None
    for function in list_of_candidate_functions:
        error = loss_function(training_function, function)
        if ((smallest_error == None) or error < smallest_error):
            smallest_error = error
            function_with_smallest_error = function

    ideal_function = IdealFunction(function=function_with_smallest_error, training_function=training_function,
                           error=smallest_error)
    return ideal_function


def find_classification(point, ideal_functions):
    """
    It computes if a point is within the tolerance of a classification
    :param point: a dict object in there is an "x" and an "y"
    :param ideal_functions: a list of IdealFunction objects
    :return:a tuple containing the closest classification if any, and the distance
    """
    current_lowest_classification = None
    current_lowest_distance = None

    for ideal_function in ideal_functions:
        try:
            locate_y_in_classification = ideal_function.locate_y_based_on_x(point["x"])
        except IndexError:
            print("This point is not in the classification function")
            raise IndexError

        # Observe here how the absolute distance is used
        distance = abs(locate_y_in_classification - point["y"])

        if (abs(distance < ideal_function.tolerance)):
            # This procedure makes sure there is handling if there are multiple classifcations possible
            # It returns the one with lowest distance
            if ((current_lowest_classification == None) or (distance < current_lowest_distance)):
                current_lowest_classification = ideal_function
                current_lowest_distance = distance

    return current_lowest_classification, current_lowest_distance

```

File - /Users/Maurice/Documents/pycharm/lossfunction.py

```
1  def squared_error(first_function, second_function):
2      """
3      Calculates the squared error to another function
4      :param other_function:
5      :return: the squared error
6      """
7      distances = second_function - first_function
8      distances["y"] = distances["y"] ** 2
9      total_deviation = sum(distances["y"])
10     return total_deviation
11
12
```

File - /Users/Maurice/Documents/pycharm/tests/test_main.py

```
 1 from unittest import TestCase
 2
 3
 4 class Test(TestCase):
 5
 6     def setUp(self):
 7         pass
 8
 9
10     def tearDown(self):
11         pass
12
13
```

File - /Users/Maurice/Documents/pycharm/tests/test_function.py

```python
from unittest import TestCase
from function import Function
import pandas as pd
from pandas._testing import assert_frame_equal
from function import IdealFunction
from lossfunction import squared_error

class TestFunction(TestCase):

    def setUp(self):
        # setting up some functions to do some tests with
        data1 = {"x":[1.0,2.0,3.0],"y":[5.0,6.0,7.0]}
        self.dataframe1 = pd.DataFrame(data=data1)

        data2 = {"x":[1.0,2.0,3.0], "y":[7.0,8.0,9.0]}
        self.dataframe2 = pd.DataFrame(data=data2)

        self.function1 = Function("name")
        self.function1.dataframe = self.dataframe1

        self.function2 = Function("name")
        self.function2.dataframe = self.dataframe2


    def tearDown(self):
        pass

    def test_locate_y_based_on_x(self):
        # Case 1 + 2: simple location test
        self.assertEqual(self.function1.locate_y_based_on_x(1.0), 5.0)
        self.assertEqual(self.function1.locate_y_based_on_x(3.0), 7.0)

        # Case 2 + 3: an exception is raised if a point cannot be found, or the input is bogus
        self.assertRaises(IndexError, self.function1.locate_y_based_on_x, 12.1)
        self.assertRaises(IndexError, self.function1.locate_y_based_on_x, "bogus")


    def test___sub__(self):
        # Case 1: substracting two functions and get the intended result
        result = {"x": [0.0, 0.0, 0.0], "y": [2.0, 2.0, 2.0]}
        result_dataframe = pd.DataFrame(data=result)
        difference = self.function2 - self.function1
        assert_frame_equal(difference,result_dataframe)

    def test_name(self):
        # Case 1: from the funtions intially created, test that the name property is set
        self.assertEqual(self.function1.name,"name")

class TestIdealFunction(TestCase):

    def test_largest_deviation(self):
        # First some setting up to do some basic testing
        data1 = {"x": [1.0, 2.0, 3.0], "y": [5.0, 6.0, 7.0]}
        dataframe1 = pd.DataFrame(data=data1)
        function1 = Function("name_ideal")
        function1.dataframe = dataframe1

        data2 = {"x": [1.0, 2.0, 3.0], "y": [7.0, 8.0, 10.0]}
        dataframe2 = pd.DataFrame(data=data2)
        function2 = Function("name_train")
        function2.dataframe = dataframe2
```

```
62
63          error = squared_error(function1,function2)
64
65          ideal_function1 = IdealFunction(function1, function2, error)
66          ideal_function2 = IdealFunction(function2, function1, error)
67          ideal_function3 = IdealFunction(function1, function1, error)
68
69          # Case 1: make sure the deviation is computed
70          self.assertEqual(ideal_function1.largest_deviation, 3.0)
71          # Case 2: make sure the deviation is the same if the functions are passed in
    reverse order
72          self.assertEqual(ideal_function2.largest_deviation, 3.0)
73          # Case 3: make sure the deviation is 0 if the deviation to the same function
    is computed
74          self.assertEqual(ideal_function3.largest_deviation, 0.0)
```

File - /Users/Maurice/Documents/pycharm/tests/test_regression.py

```python
from unittest import TestCase
from regression import minimise_loss
from regression import find_classification
from lossfunction import squared_error
import pandas as pd
from function import Function
from function import IdealFunction

class TestClassification(TestCase):
    def setUp(self):
        # testing this class requires a lot of setting up
        # We basically need a couple of ideal functions
        data1 = {"x": [1.0, 2.0, 3.0], "y": [5.0, 6.0, 7.0]}
        self.dataframe1 = pd.DataFrame(data=data1)
        self.function1 = Function("name_ideal")
        self.function1.dataframe = self.dataframe1

        data2 = {"x": [1.0, 2.0, 3.0], "y": [7.0, 8.0, 10.0]}
        self.dataframe2 = pd.DataFrame(data=data2)
        self.function2 = Function("name_train")
        self.function2.dataframe = self.dataframe2

        self.error = squared_error(self.function1, self.function2)

        self.ideal_function = IdealFunction(self.function1, self.function2, self.
error)

    def tearDown(self):
        pass

    def test_name(self):
        # Case: test the name property
        self.assertEqual(self.ideal_function.name, "name_ideal")

    def test_minimise_loss(self):
        # In order to actually test minimise loss we also need to have a list of
 candidate ideal functions
        # Below is the setup
        data1 = {"x": [1.0, 2.0, 3.0], "y": [7.5, 8.5, 9.5]}
        dataframe1 = pd.DataFrame(data=data1)
        function1 = Function("ideal_1")
        function1.dataframe = dataframe1

        data2 = {"x": [1.0, 2.0, 3.0], "y": [8.0, 9.0, 10.0]}
        dataframe2 = pd.DataFrame(data=data2)
        function2 = Function("ideal_2")
        function2.dataframe = dataframe2

        data3 = {"x": [1.0, 2.0, 3.0], "y": [8.5, 9.5, 10.5]}
        dataframe3 = pd.DataFrame(data=data3)
        function3 = Function("ideal_3")
        function3.dataframe = dataframe3

        # after all this typing we can create a list
        list_of_ideal_functions = [function1,function2,function3]

        # we throw all the variables in the function
        ideal_function = minimise_loss(self.function2,list_of_ideal_functions,
squared_error)
        # case 1: check that the correct function is identigied, by checking the name
        self.assertEqual(ideal_function.name, "ideal_1")
        # case 2: check that the correct error (squared error) is computed
        self.assertEqual(ideal_function.error, 0.75)
```

```
61          # case 3: check that the training data function is correctly stored as well
    by verifying the name
62          self.assertEqual(ideal_function.training_function.name, "name_train")
63          # Case 4: check that the largest distance is computed correctly
64          self.assertEqual(ideal_function.largest_deviation, 0.5)
65
66      def test_find_classification(self):
67          # for testing the matching we need a couple of ideal functions
68          # Data for the first ideal function
69          data1 = {"x": [1.0, 2.0, 3.0, 4.0], "y": [5.0, 6.0, 7.0, 8.0]}
70          dataframe1 = pd.DataFrame(data=data1)
71          function1 = Function("name_ideal1")
72          function1.dataframe = dataframe1
73
74
75          data2 = {"x": [1.0, 2.0, 3.0, 4.0], "y": [7.0, 8.0, 10.0, 9.0]}
76          dataframe2 = pd.DataFrame(data=data2)
77          function2 = Function("name_train1")
78          function2.dataframe = dataframe2
79
80          error1 = squared_error(function1, function2)
81          ideal_function1 = IdealFunction(function1, function2, error1)
82          ideal_function1.tolerance = 2.0
83
84          # Data for the second ideal function
85          data3 = {"x": [1.0, 2.0, 3.0, 4.0], "y": [10.0, 11.0, 12.0, 9.0]}
86          dataframe3 = pd.DataFrame(data=data3)
87          function3 = Function("name_ideal2")
88          function3.dataframe = dataframe3
89
90          data4 = {"x": [1.0, 2.0, 3.0, 4.0], "y": [10.5, 11.5, 12.5, 8.5]}
91          dataframe4 = pd.DataFrame(data=data4)
92          function4 = Function("name_train2")
93          function4.dataframe = dataframe4
94          error2 = squared_error(function3,function4)
95          ideal_function2 = IdealFunction(function3, function4, error2)
96          ideal_function2.tolerance = 0.5
97
98
99          # Case 1: Simple happy case in which a clear match and positive distance has
     to be considered
100         list_of_ideal_functions = [ideal_function1, ideal_function2]
101         point1 = {"x": 1.0, "y": 6.0}
102         classification, distance = find_classification(point1,
    list_of_ideal_functions)
103         self.assertEqual(classification.name, "name_ideal1")
104         self.assertEqual(distance, 1.0)
105
106         # Case 2: Simple happy case in which no match can be identified because the
    criterion is not met
107         point2 = {"x": 1.0, "y": 18.0}
108         classification, distance = find_classification(point2,
    list_of_ideal_functions)
109         self.assertIsNone(classification)
110         self.assertIsNone(distance)
111
112         # Case 3: Simple happy case in which a clear match and negative distance has
     to be considered
113         point3 = {"x": 3.0, "y": 11.6}
114         classification, distance = find_classification(point3,
    list_of_ideal_functions)
115         self.assertEqual(classification.name, "name_ideal2")
116         self.assertAlmostEqual(distance, 0.4)
```

```
117
118         # Case 4: The provided point is not part of the sequence and no
    classification can be made (in this simple algorithm)
119         point4 = {"x": 6.0, "y": 11.6}
120         self.assertRaises(IndexError, find_classification, point4,
    list_of_ideal_functions)
121
122         # Case 5: Tricky case in which two classifiactions match but the one with
    the closest distances wins
123         point5 = {"x": 4.0, "y": 8.6}
124         classification, distance = find_classification(point5,
    list_of_ideal_functions)
125         self.assertEqual(classification.name, "name_ideal2")
126         self.assertAlmostEqual(distance, 0.4)
```

File - /Users/Maurice/Documents/pycharm/tests/test_lossfunction.py

```python
from unittest import TestCase
import pandas as pd
from function import Function
from lossfunction import squared_error

class Test(TestCase):
    def setUp(self):
        # Setting up some functions
        data1 = {"x":[1.0,2.0,3.0],"y":[5.0,6.0,7.0]}
        self.dataframe1 = pd.DataFrame(data=data1)

        data2 = {"x":[1.0,2.0,3.0], "y":[7.0,8.0,9.0]}
        self.dataframe2 = pd.DataFrame(data=data2)

        self.function1 = Function("name")
        self.function1.dataframe = self.dataframe1

        self.function2 = Function("name")
        self.function2.dataframe = self.dataframe2


    def tearDown(self):
        pass

    def test_squared_error(self):
        # case 1: simple test if loss function computes correct value
        self.assertEqual(squared_error(self.function1, self.function2), 12.0)
        # case 2: simple test if loss function is associative
        self.assertEqual(squared_error(self.function2, self.function1), 12.0)
        # case 3: check if regression of two equal functions is 0
        self.assertEqual(squared_error(self.function1, self.function1), 0.0)


```