Exploratory Data Analysis using Python

Author: Ajay Chandra A S
ajay.chandra-a-s@iu-study.org

Matriculation number: 321149868

Module: Programming with python (DLMDSPWP01)

Date: 31-03-2023
Place: Bengaluru, India

Table of Contents

1. Abstract

In this investigative report, the first objective is to write a python program as per the given text. Next objective is to find an area of interest that is worth exploring where we can develop a question, perform in-depth research, and publish the results as a research paper keeping in mind that the research topic should stem from the text and datasets of the written assignment.

2. Introduction

The written assignment is broken down in two main sections as follows:

- Development of a Python program to create and populate a SQLite database with various datasets. The data analysis is performed on them by following the below steps:
    - A Python-program is written that uses training data to choose the four ideal functions which are the best fit out of the fifty provided, this is done according to the minimum sum of squared deviations.
    - Each of the test points is mapped to the ideal functions in the following way. First, maximum deviation between each pair of training and ideal function is determined. The deviation times sqrt(2) is the maximum deviation a test point may have to be mapped to a particular ideal function. Subsequently, the deviation between each test point and ideal functions is calculated and checked if the criterion is fulfilled for one or more than one of the ideal function and finally the data is visualized logically.
- A suitable topic is derived that is logically connected with the program and the dataset. The chosen topic is investigative/research oriented in nature.

The Python program is sensibly object-oriented and include at least one inheritance. It includes standard and user-defined exception handling. To achieve the required functionalities, the program use libraries such as Pandas, Bokeh, sqlalchemy, and others for data manipulation and visualization. Additionally, the program is documented in its entirety and include docstrings. Unit tests is written for all useful elements to ensure program correctness. GitHub is used to handle version control and to push to project to the cloud. Overall, the program is designed with logical reasoning and well-organized to meet the given criteria.

My research question: “How does the choice of statistical methods for mapping ideal functions affect the final results in data analysis?”

3. Exploratory Data Analysis

Exploratory data analysis (EDA) is a statistical method used to examine and describe the main features of a dataset. This approach often involves the use of visual techniques such as statistical graphics to summarize and gain insight into the data (“Exploratory Data Analysis,” 2023).

Our project includes, three sets of data “train”, “test” and “ideal” in a .csv file format. The data is summarized and analysed to compute the correlations, check for irregularities etc.,

3.1 Data collection

The data is already provided for the project, there is no need to collect new data for exploratory data analysis. Instead, we will directly start analysing the provided data and use various statistical techniques to summarize and gain insights into the data. This can help in understanding the data better and making informed decisions based on the findings.

3.2 Data cleansing

Data cleansing, also known as data cleaning, is the process of identifying and correcting (or removing) inaccurate or corrupt records from a database. It involves determining which parts of the data are incomplete, incorrect, inaccurate, or irrelevant, and then replacing, changing, or deleting the dirty or coarse data (“Data Cleansing,” 2023).

Outliers: A data item or object that differ considerably from the other (so-called normal) items is referred to as an outlier. Measurement or execution mistakes may be the root cause of them. Outlier mining is the process of analysing data to find outliers (“Detect and Remove the Outliers Using Python,” 2021).

The most popular and reliable method for identifying outliers in the research community is the interquartile range approach given by:

$$\mathrm{IQR} = \mathrm{Quartile3} - \mathrm{Quartile1}$$

To define the outlier base value is defined above and below datasets normal range namely Upper and Lower bounds, define the upper and the lower bound (1.5*IQR value is considered)

$$\mathrm{upper} = \mathrm{Q3} + 1.5 * \mathrm{IQR};\ \mathrm{lower} = \mathrm{Q1} - 1.5 * \mathrm{IQR}$$

Based on the upper and lower bounds, we detect the values that lie outside the bounds and take appropriate actions based on the project at hand.

Duplicate data: Duplicate values refer to the occurrence of the same data point more than once in a dataset (“Data Deduplication,” 2023). It is important to take care of duplicate values as they can skew analysis results and lead to incorrect conclusions. To address this, techniques such as dropping duplicates or aggregating them through grouping can be used to ensure the integrity of the data.

Missing data: In statistics, the absence of a data value for a variable in an observation is known as missing data, or missing values (“Missing Data,” 2023). The conclusions that can be derived from the data can be significantly impacted by missing data, hence it is important to handle the missing data appropriately.

3.3 Data visualization

Data and information visualization is an area of study that involves creating visual representations of data and information. This field uses graphics to present complex information in a way that is easy to understand. It is especially useful when working with large amounts of data as it allows for efficient communication of information (“Data and Information Visualization,” 2023).

Scatter plot: It is used to display the relationship between two variables. It is a graph that plots points along two axes, with each axis representing one of the variables being measured. Scatter plots are often used to identify patterns, trends, and correlations between variables. They are useful in fields such as statistics, data analysis, and machine learning, where understanding the relationship between variables is important for making informed decisions (“Scatter Plot,” 2023).

Bar chart: A bar chart is a type of chart used to represent categorical data, where rectangular bars are used to indicate the value of each category. The height or length of the bars is proportional to the corresponding value. Bar charts can be displayed vertically or horizontally, and when presented vertically, they are sometimes referred to as column charts (“Bar Chart,” 2023).

Box plot: A box plot is a descriptive statistical method used to display the distribution, variability, and skewness of numerical data. It shows quartiles of the data through a box, with whiskers extending from the box indicating variability beyond the quartiles. The plot may also display outliers as individual points beyond the whiskers. Unlike other statistical methods, box plots do not assume any specific distribution of the data (“Box Plot,” 2023).

3.4 Training dataset

Based on the visual observation of the dataset, we know it has an independent variable ‘x’, and 4 dependent variables ‘y1’, ‘y2’, ‘y3’, ‘y4’ and consists of 400 records. The data description of the dependant variables are as follows:

	y1	**y2**	**y3**	**y4**
Mean	0.065942659	- 0.032593237	1.945832733	4.227447817
Median	0.035778266	- 0.075776383	2.23816265	4.57409575
Sample Variance	2.107005311	1.090552668	3.430803362	3.387049813
Range	4.9434569	3.7873881	32.4800968	32.684095
Minimum	-2.4628475	- 1.8919262	- 29.130646	- 26.938665
Maximum	2.4806094	1.8954619	3.3494508	5.74543
First Quartile	- 1.350924875	- 0.955948365	1.584582025	3.848527575
Third Quartile	1.481902175	0.881776118	2.765394425	4.9506881
Count	400	400	400	400

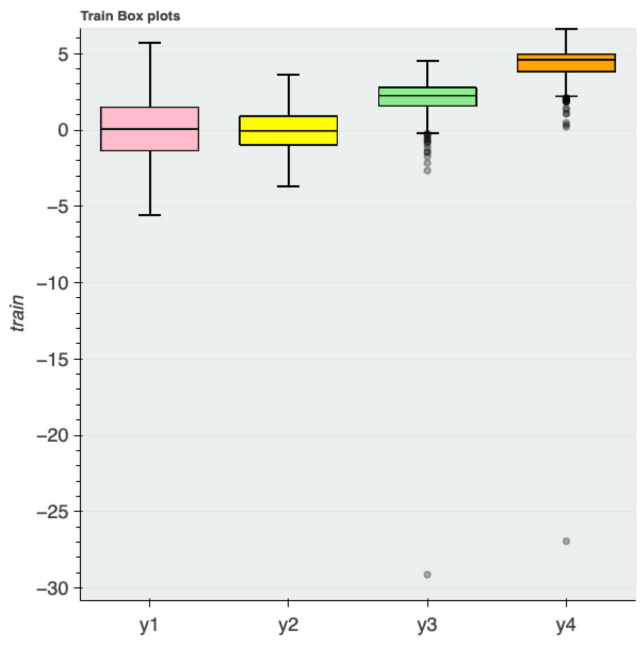
The training dataset contains no duplicate values and hence there is no need to handle or remove them.

The training dataset contains no null values and hence there is no need to handle them.

	> Upper bound	**< Lower bound**	**Total**
y1	0	0	0
y2	0	0	0
y3	0	16	16
y4	0	19	19

The above table provides the number of the values that is crossing the upper and lower bounds calculated based on IQR for all the functions in the given training dataset.

Removing/handling outliers is not feasible for this project as all values in ‘y’ columns are required for the subsequent steps involved in the process.



We can deduce the following insights from the above muti box plot for the training functions. To avoid repetition, we are excluding some of the information that is already provided in the data description table above. The boxes representing y3 and y4 are consolidated, indicating that they have lower variability and are more consistent compared to the wider boxes representing y1 and y2. After analysing the individual box plots, we can draw the following inferences. The box plots for y1 and y2 do not have points outside of the whiskers, implying that their minimum and maximum values are within the whiskers. In contrast, points can be observed outside of the whiskers in the box plots for y3 and y4, which denotes that the minimum and maximum values are outside the whiskers. Points lying outside of the whiskers are considered outliers, and we can observe that y3 and y4 have outliers in the lower end of the data.

3.5 Ideal dataset

Based on the visual observation of the dataset, we know it consists of an independent variable ‘x’, and 50 dependent variables ‘y1’, ‘y2’, ‘y3’,…..’y50’.

3.6 Test dataset

Based on the visual observation of the dataset, we know it consists of an independent variable ‘x’, and a dependent variable ‘y’ and consists of 100 records. The data description of the dependant variable ‘y’ is as follows:

Mean	-1.200604257
Median	1.05325105
Sample Variance	55.18743406
Range	34.435364
Minimum	-28.044102
Maximum	6.391262
Q1	-1.242774775
Q3	2.748816275
Count	100

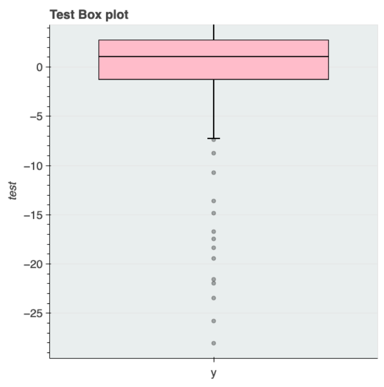
We have identified 10 duplicate values in the ‘x’ variable of the test function, and it is important to handle them appropriately to ensure the accuracy and integrity of our data. Removing duplicates from a dataset using mean value imputation is a very common approach in data pre-processing and cleaning (“Imputation (Statistics),” 2023). Therefore, we are replacing the ‘y’ row with the mean value of its respective ‘x’ row. Finally, we will remove the duplicate ‘x’ rows. We are opting to maintain a unique 'x' value for each 'y' value in this project as it will aid us in associating the optimal ideal functions with the test points during a subsequent stage of mapping.

The test dataset contains no null values and hence there is no need to handle them.

	> **Upper bound**	< **Lower bound**	**Total**
y	0	14	14

The above table provides the number of the values that is crossing the upper and lower bounds calculated based on IQR for all the functions in the given test dataset.

Removing/handling outliers is not feasible for this project as all values in ‘y’ column are required for the subsequent steps involved in the process.



We can deduce the following insights from the above box plot for the test function. To avoid repetition we are excluding some of the information that is already provided in the data description table above. The box is consolidated, indicating that they have lower variability and are more consistent. Some points can be observed outside of the whiskers of the plot, which denotes that the minimum and maximum values are outside the whiskers. Points lying outside of the whiskers are considered outliers, and we can observe that we have outliers at the lower end of the data.

3.7 Data storage and retrieval

The chosen database system for the project is SQLite, which will be managed through SQLAlchemy Python module. SQLAlchemy is an SQL toolkit and Object-relational Mapper known for its stability and high performance, making it ideal for handling large volumes of data efficiently ("SQLAlchemy," 2023). It supports SQLite database, ensuring seamless integration with the database. SQLAlchemy's flexibility allows for pythonic SQL queries to be written, making it easy to work with the data. It also allows for pythonic SQL queries for different SQL databases, providing greater flexibility and portability for the project ("SQLite," 2023).

3.8 Database schema

Table 1: The training data's database table:

x	y1 (training func)	y2 (training func)	y3 (training func)	y4 (training func)
x1	y11	y21	y31	y41
…	…	…	…	…
xn	y1n	y2n	y3n	y4n

Table 2: The ideal functions database table:

x	y1 (ideal func)	y2 (ideal func)	…	ym (ideal func)	…	y50 (ideal func)
x1	y11	y21	…	ym1	…	y50_1
…	…	…	…	…	…	…
xn	y1n	y2n	…	ymn	…	y50_n

Table 3: The database table of the test-data, with mapping and y-deviation

x (test func)	y (test func)	Delta Y (test func)	No. of ideal func
x1	y11	y21	n1
…	…	…	…
xn	y1n	y2n	y3n

3.9 Regression analysis

Regression analysis is a statistical modelling technique used to estimate the relationships between a dependent variable and one or more independent variables. The primary method of regression analysis is linear regression, which involves finding the line that best fits the data according to a particular mathematical criterion. This process is used to understand how changes in the independent variables affect the dependent variable ("Regression Analysis," 2023).

Since this project involves a single dependent variable and one independent variable, linear regression analysis can be a suitable method for analysing the data.

The formula for linear regression is as follows:

$$y = \beta_0 + \beta_1 x$$

where,

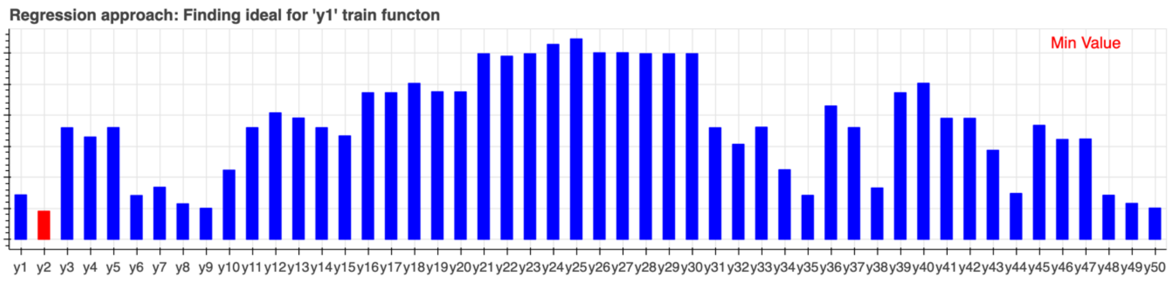
'y' is the dependent variable,

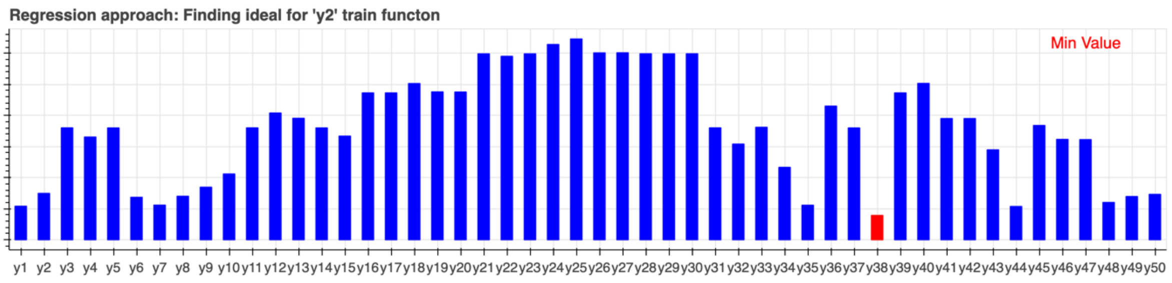
'x' is the independent variable,

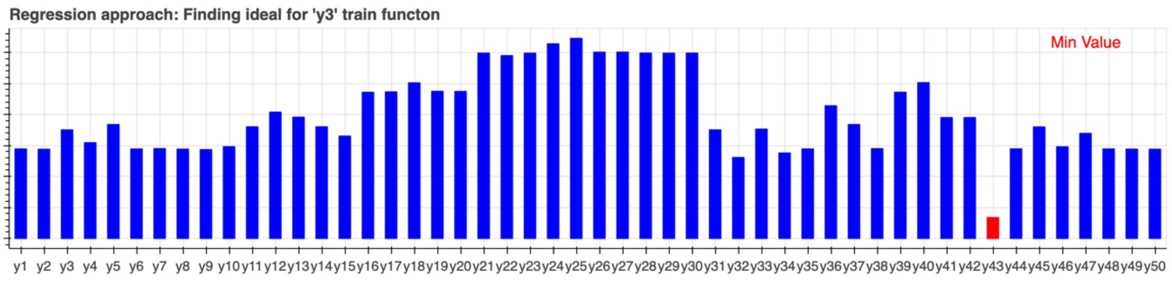
$'\beta_0'$ is the intercept, and

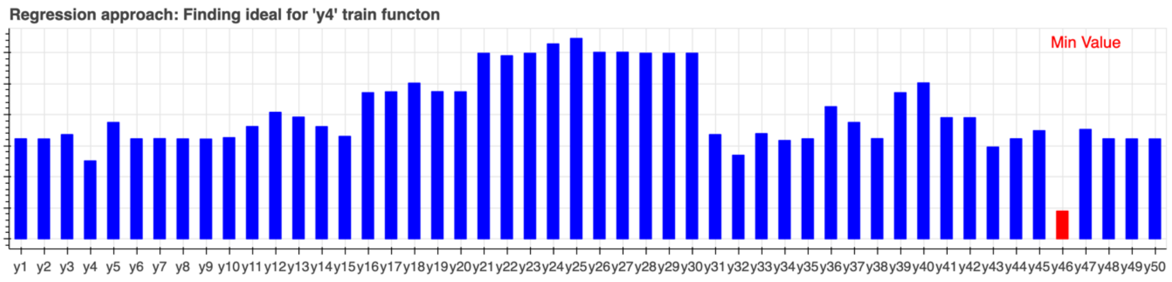
$'\beta_1'$ is the slope

3.9.1 Bar charts showing results of regression-based approach to determine ideal functions



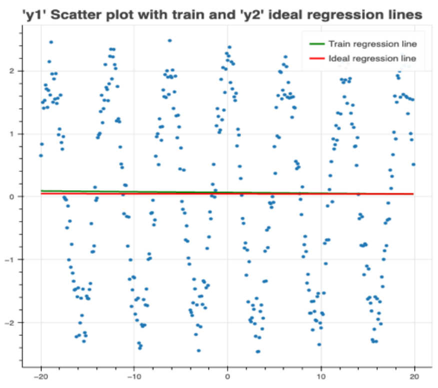


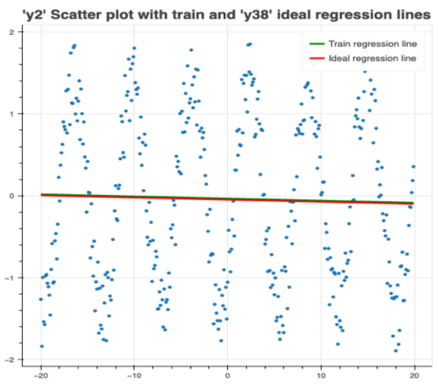


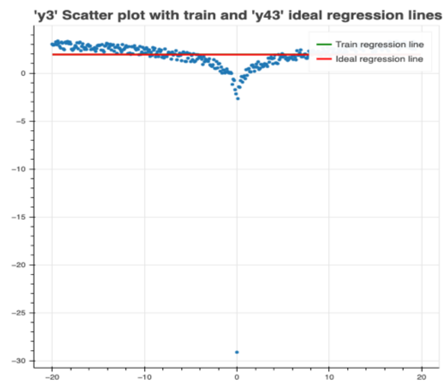


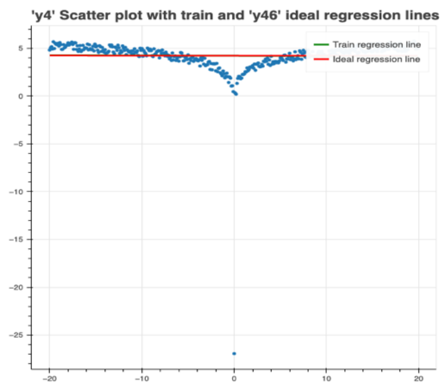
The bar charts presented above was generated through a process that involved finding regression equations for each train function and 50 ideal functions. The absolute difference was then calculated between the 'y' values of each row in the regression line of the train and ideal functions. The sum of all these differences was calculated and plotted on the y-axis, which is in logarithmic scale due to the wide range of values. The red bar in the chart represents the ideal function for the corresponding train function, as it has the least value when compared to the other values.

3.9.2 Scatter plot of training function displaying regression lines for train and ideal functions in regression-based approach to determine ideal functions









The above plots were constructed by first creating an (x, y) scatter plot for the train function. Next, a regression line was plotted for the train function along with the ideal function that was chosen based on a previous step. We can observe that the ideal functions chosen for each train function are overlapping or very close to each other. Some of the advantages of the overlapping property of the functions are, It can Improve the accuracy of the predictions and minimize the possibility of erroneous assumptions due to flawed data (Tibshirani & Friedman, n.d.), the model's capability to forecast the results of new data points that were not part of the initial training dataset will improve (Jordan & Mitchell, 2015).

3.10 Correlation analysis

Correlation in statistics refers to the statistical relationship between two random variables, often indicating the extent to which the two variables are linearly related. This relationship can be observed in various phenomena, such as the correlation between the price of a product and the quantity consumers are willing to purchase. Correlations are significant because they can indicate a predictive relationship that can be utilized in practical applications ("Correlation," 2022).

As this project involves examining the relationship between a single dependent variable and one independent variable, correlation analysis may be an appropriate method for analysing the data.

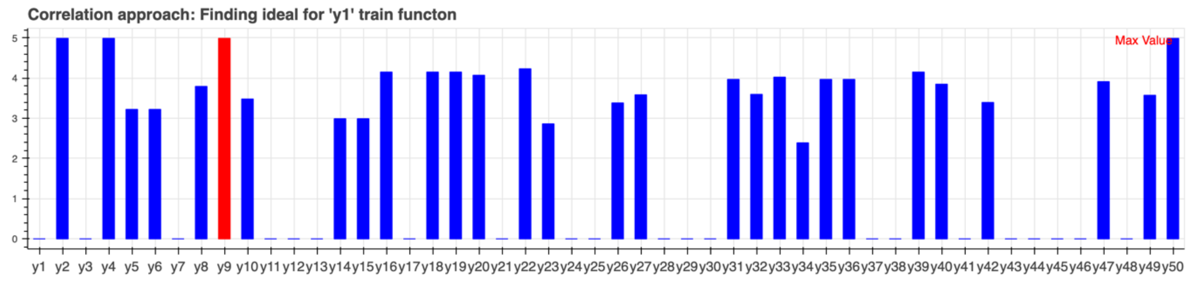
The formula for correlation is as follows:

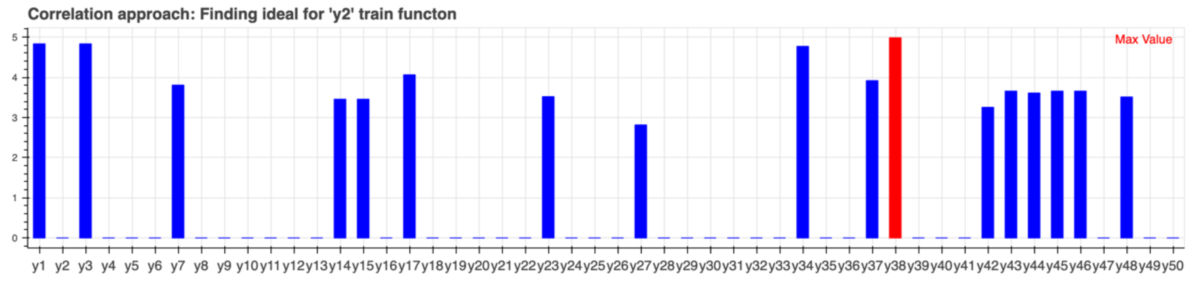
$$p(x, y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

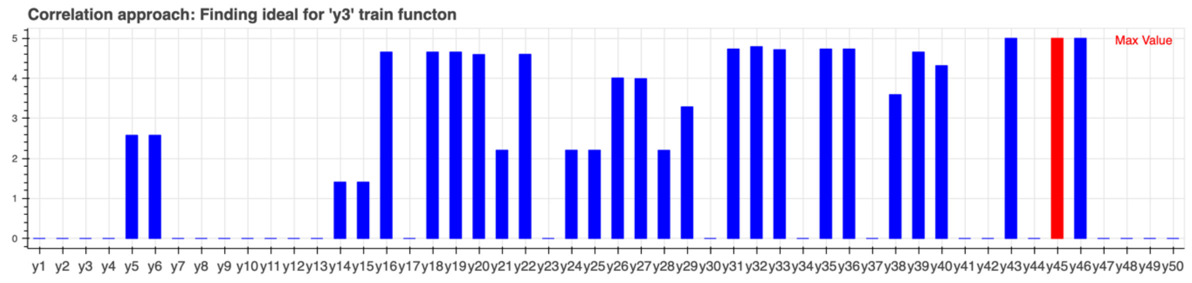
After calculating the correlation coefficient between two variables using the formula, the resulting value can be interpreted in the following way: if the coefficient is close to -1, there is a negative correlation between the variables; if it is near 0, there is no correlation; and if it is close to 1, there is a positive correlation.

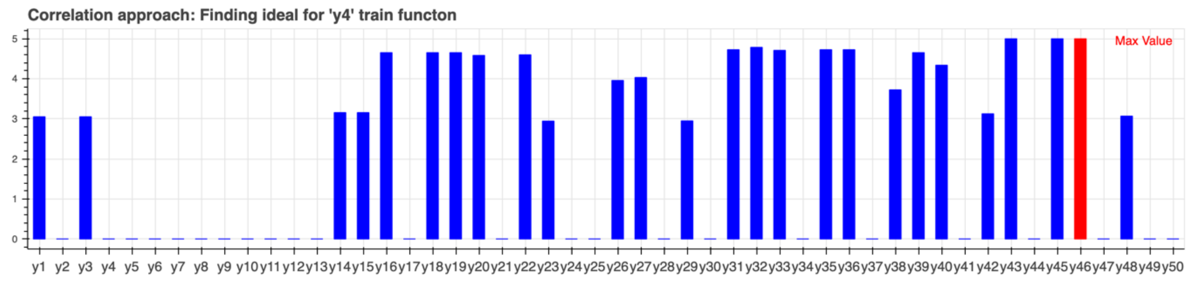
Some of the advantages of this approach are, selecting an ideal function based on correlation ensures that the evaluation is unbiased and founded on factual information (Field, 2013), with absence of bias it is easier to understand the behaviour of the ideal function and the training function (Warner, 2013), Correlation is a straightforward and easy-to-calculate measure that does not require extensive computational resources.

3.10.1 Bar charts showing results of correlation-based approach to determine ideal functions



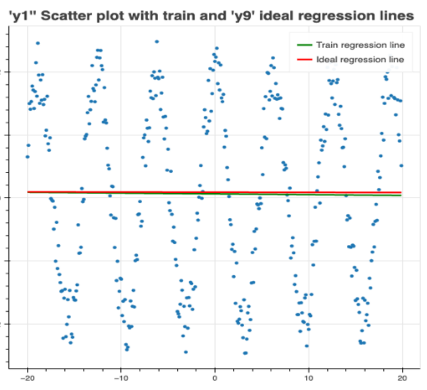


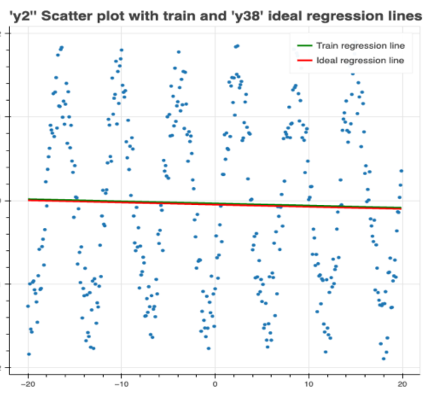


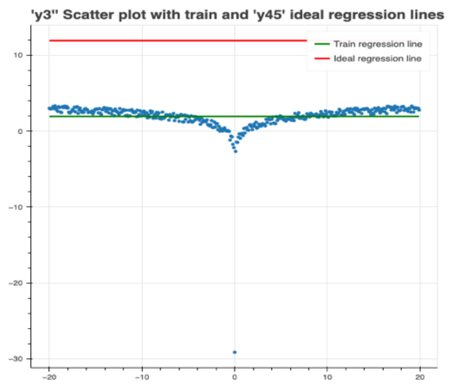


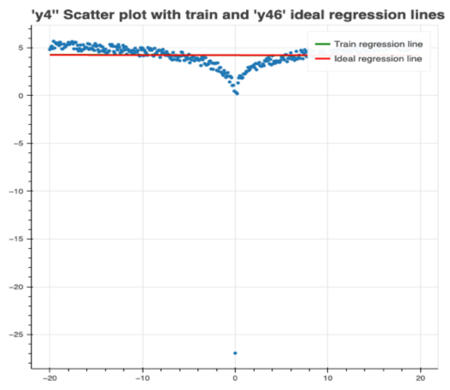
The bar charts shown above were created using the following process: first, the correlation between each train function and all ideal functions was computed. Then, the ideal function with the highest correlation for each train function was selected and displayed in red on the chart as its respective ideal function. If the same ideal function matched with multiple train functions, the ideal function was assigned to the first train function, and for subsequent matches with the same ideal function, the second highest correlated ideal function was assigned. The priority of the train functions was based on the column names in the "train.csv" file. Additionally, negative correlation bars were not represented and are left empty in the chart as they indicate a negative relationship between the train function and the ideal function. In such cases, there is no ideal function that can be recommended as a best fit, so it's appropriate to leave those bars empty to avoid misleading the viewer.

3.10.2 Scatter plot of training function displaying regression lines for train and ideal functions in correlation-based approach to determine ideal functions









The above plots were constructed by first creating an (x, y) scatter plot for the train function. Next, a regression line was plotted for the train function along with the ideal function that was chosen based on a previous step. The ideal functions ‘y9’, ‘y38’ and ‘y46’ chosen for the train functions are overlapping or very close to each other. However, we can observe that the ideal function ‘y45’ has a considerable gap between the regression lines.

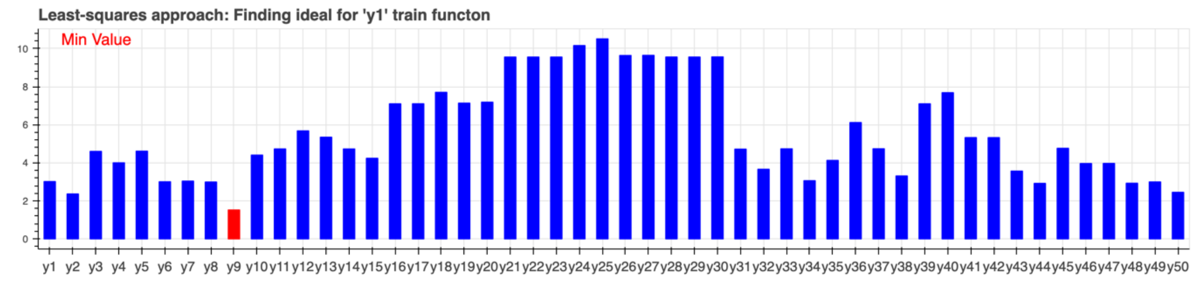
3.11 Least Squares analysis

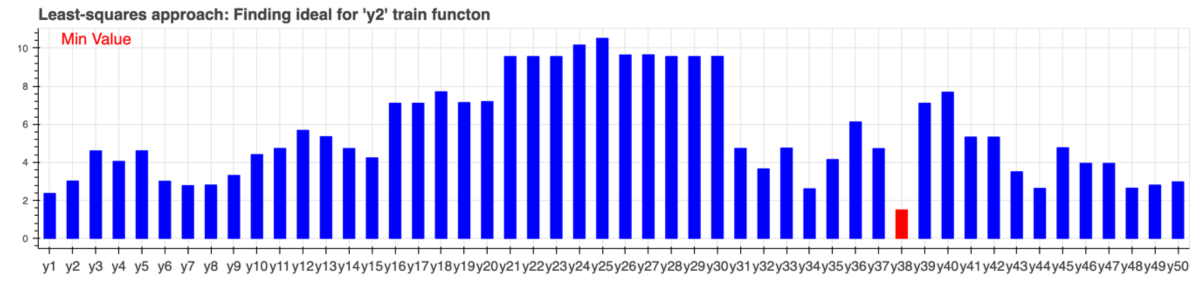
In regression analysis, the least squares method is a commonly used approach to find an approximation of the solution. This is achieved by minimizing the sum of the squares of the differences between the observed values and the predicted values obtained from each equation in the analysis (“Least Squares,” 2023).

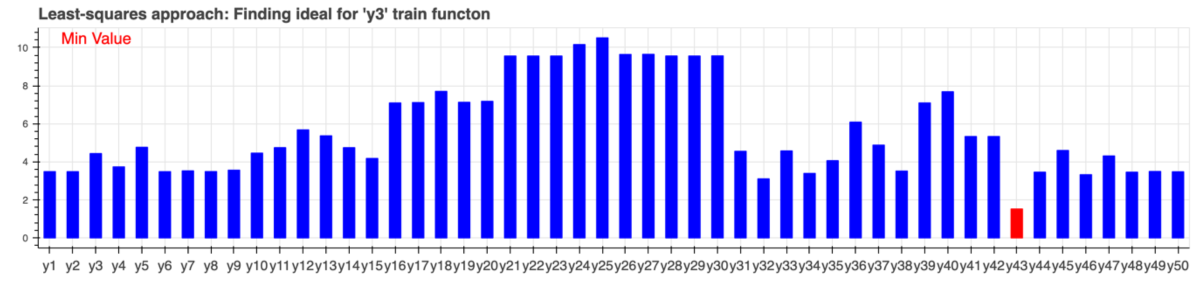
To comply with the project's requirement of identifying ideal functions using the least squares method, we have included this approach as part of our analysis.

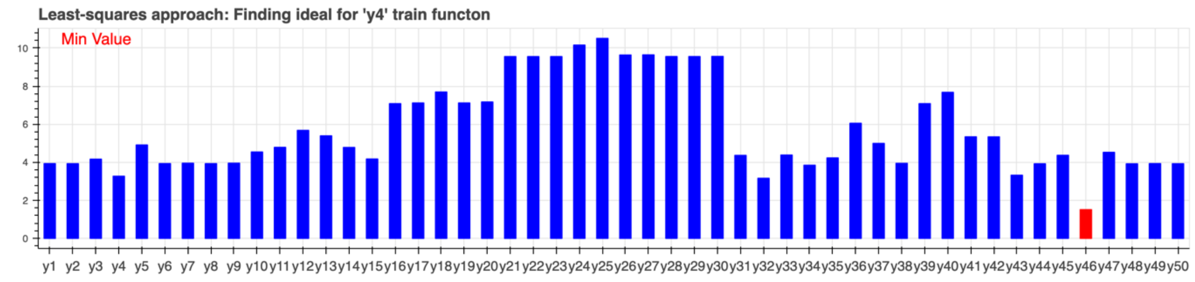
Some of the advantages of this approach are, it is less affected by extreme values or errors in the data because it works by minimizing the sum of the squared differences between the predicted values and the actual values (V, n.d.), simple and intuitive approach that is easy to understand and apply (Sohil et al., 2022).

3.11.1 Bar charts showing results of least-squares based approach to determine ideal functions



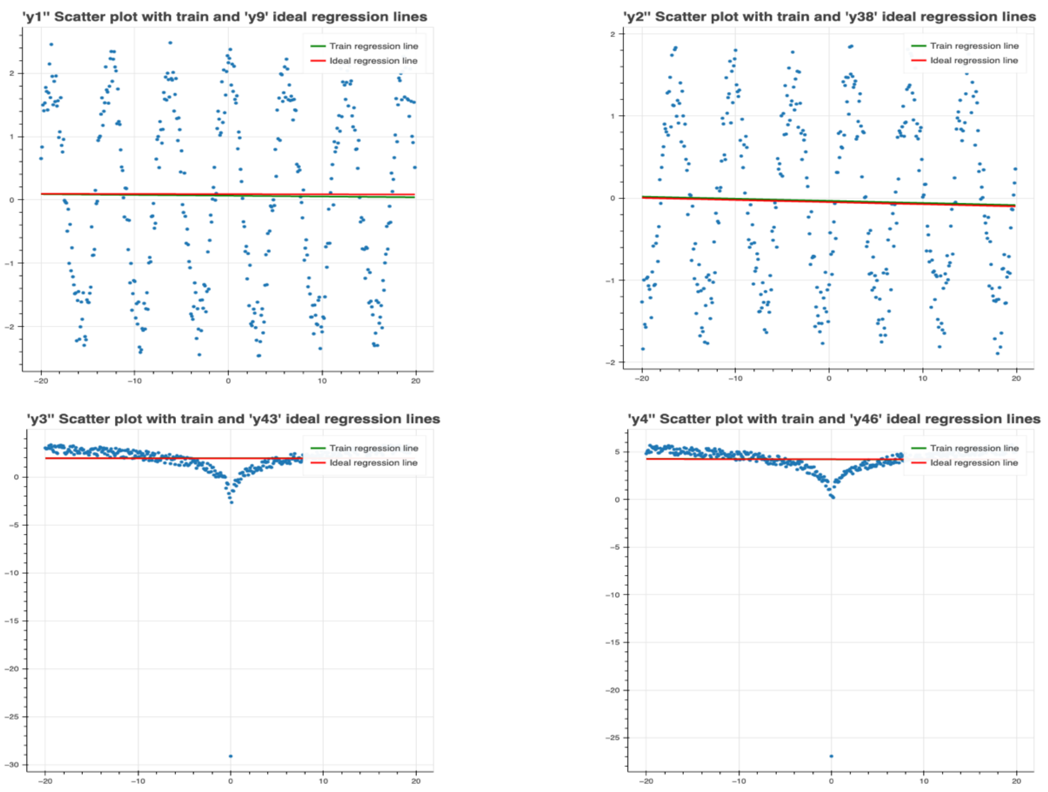






The above bar charts were generated using a particular method. Initially, the squared difference was calculated between each row of the train function and every available ideal function. The differences calculated were then added up for each train function and the corresponding ideal function. Next, each train function was matched to the ideal function with the minimum value, which was determined based on the sum of differences. The bar in red represents the ideal function that was matched to the respective train function.

3.11.2 Scatter plot of training function displaying regression lines for train and ideal functions in least-squares based approach to determine ideal functions

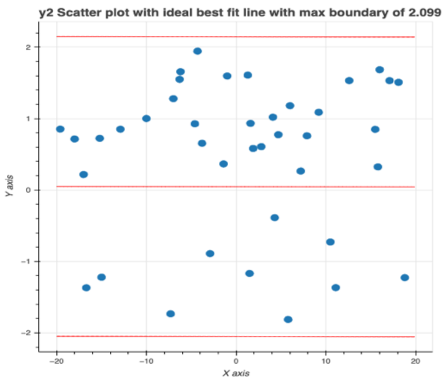


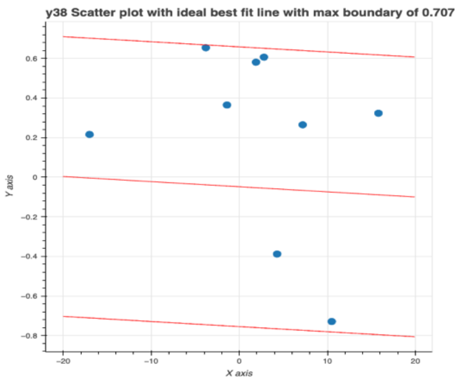
The above plots were constructed by first creating an (x, y) scatter plot for the train function. Next, a regression line was plotted for the train function along with the ideal function that was chosen based on a previous step. We can observe that the ideal functions chosen for each train function are overlapping or very close to each other. Some of the advantages of the overlapping property of the functions are, It can Improve the accuracy of the predictions and minimize the possibility of erroneous assumptions due to flawed data (Tibshirani & Friedman, n.d.), the model's capability to forecast the results of new data points that were not part of the initial training dataset will improve (Jordan & Mitchell, 2015).

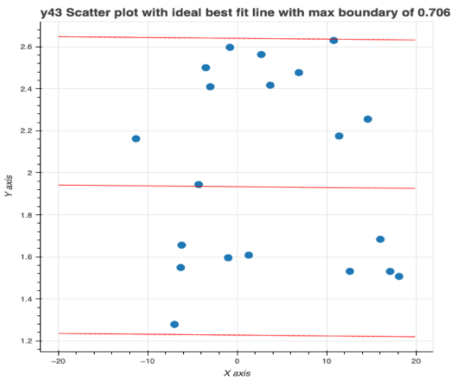
3.12 Results of test Point Mapping

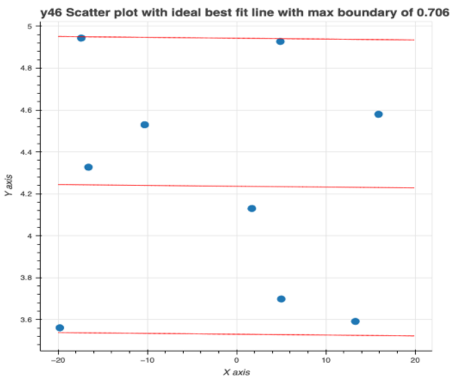
The results presented below are generated using a specific process. Firstly, the absolute difference between the test points and regression values for all four ideal functions was calculated. Then, the obtained differences were compared to the maximum deviation value for the corresponding ideal function. The maximum deviation value was calculated by finding the difference between the absolute values of the training and its respective ideal function for all rows of y values, and identifying the row with the maximum value, which was then multiplied by the square root of 2. Test points matching the criteria were then plotted for each ideal function along with the regression line of ideal function, and upper and lower bounds were plotted based on the max deviation value.

3.12.1 Results showing test data mapped within maximum deviation for regression-based approach to determine ideal functions

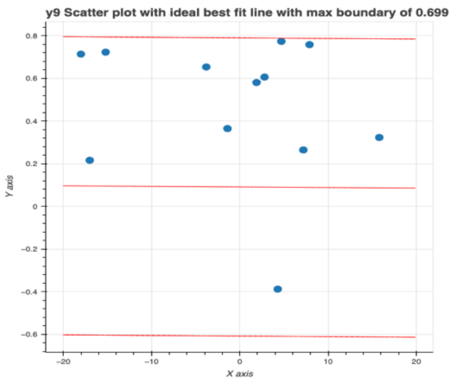


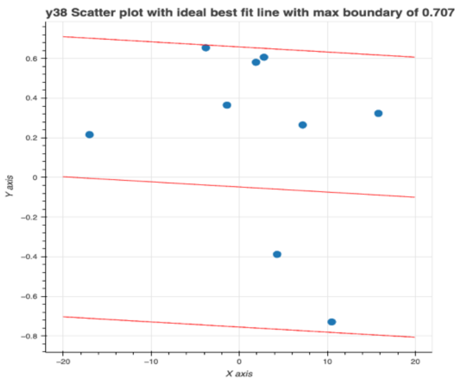


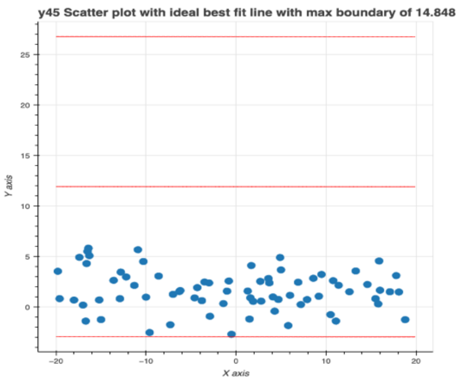


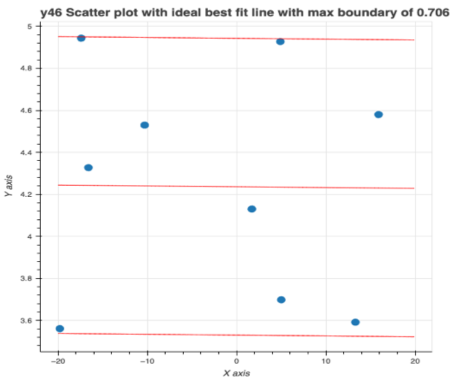


3.12.2 Results showing test data mapped within maximum deviation for correlation-based approach to determine ideal functions

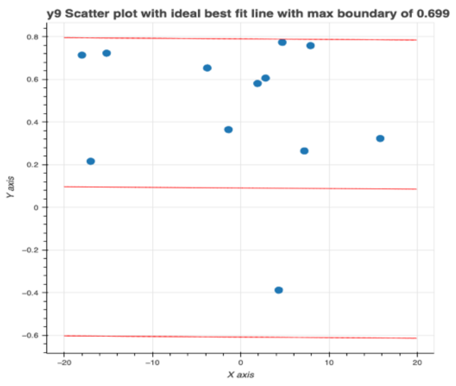


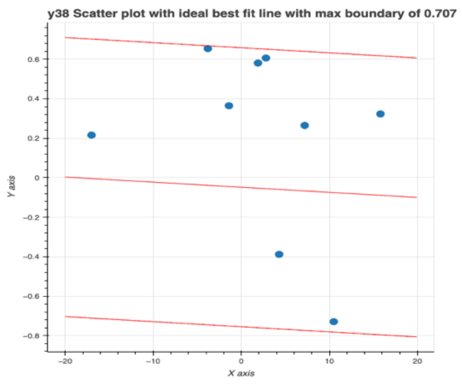


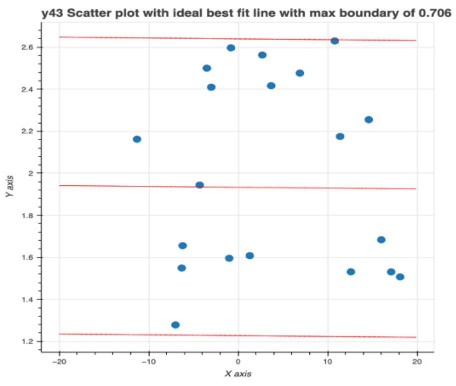


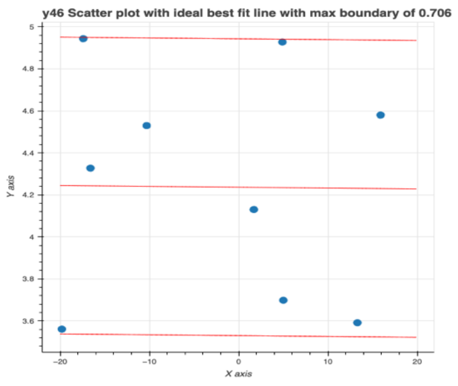


3.12.3 Results showing test data mapped within maximum deviation for least-squares approach to determine ideal functions





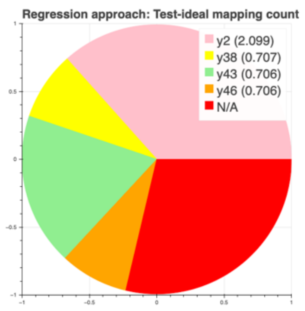


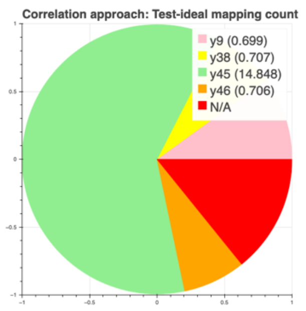


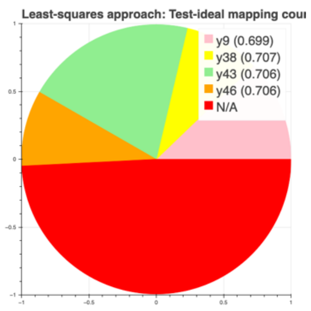
3.13 Results summary

Regression method		**Correlation method**		**Least squares method**	
Mapping type and max dev	*Count*	*Mapping type and max dev*	*Count*	*Mapping type and max dev*	*Count*
Y2 (2.099)	40	Y9 (0.699)	12	Y9 (0.699)	12
Y38 (0.707)	9	Y38 (0.707)	9	Y38 (0.707)	9
Y43 (0.706)	20	Y45 (14.848)	73	Y43 (0.706)	20
Y46 (0.706)	9	Y46 (0.706)	9	Y46 (0.706)	9
N/A	31	N/A	17	N/A	48
Multiple mappings	19	Multiple mappings	22	Multiple mappings	8
Total Mappings	109		120		98

The table presents the outcome of the final test-ideal mapping for the three approaches employed to select the ideal functions. It displays the number of test points that were mapped to their respective ideal functions based on the maximum deviation value, as well as the number of multiple mappings that occurred, indicating one test point mapped to multiple ideal functions. Additionally, it presents the count of test points that did not map to any ideal functions, denoted as N/A.







The presented pie charts provide a visual representation of the mapping of test points to their respective ideal functions along with the unmapped points. The chart's legend includes the name of the ideal function and its maximum deviation value. A crucial inference that can be drawn from this chart is that an increase in maximum deviation leads to an increase in the number of mapped test points, resulting in fewer unmapped points. This increase is due to the higher coverage provided to the test points. Based on these findings, two alternative models can be proposed: one with a lower maximum deviation value, resulting in less scattered points and coverage, and the other with a higher maximum deviation value, resulting in more scattered points and coverage.

4. Conclusion

The initial step in this project was to conduct Exploratory Data Analysis by developing a Python code that utilizes statistical graphics to examine and summarize the main features of the dataset, to gain insights into the data.

Throughout the research, I was intrigued by the central inquiry at hand: "How does the choice of statistical methods for mapping ideal functions affect the final results in data analysis?" and hence I proceeded to work on the question.

During the analysis, various statistical methods and techniques, such as "Regression", "Correlation", and "Least squares" were utilized. The results of each approach were evaluated by presenting tables and visualizations that compares the number of test points mapped to ideal functions.

Finally, based on the findings of our analysis, we have put forward two potential models that can be employed depending on the specific requirements of the situation.

The primary emphasis of the project was on the approach and evaluation. The source code created for this project adhered to the criteria outlined in the module guidelines.

5. Additional Task

Assuming that I have a successfully created project on the Version Control System Git and have a Branch called develop where all operations of the developer team are combined, the following Git-commands are necessary to clone the branch and develop on my local PC:

1. git clone <repository_url> -b develop
2. git checkout -b <new_branch_name>

Assuming that I have added a new function, the following Git-commands are necessary to introduce this project to the team's develop Branch:

1. git add <file_name>
2. git commit -m "added new function"
3. git push origin <new_branch_name>
4. Create a Pull-request for review and merge to develop branch.

Once my contribution is reviewed by one or several members of my team and approved, it will be merged into the develop Branch.

6. References

Bar chart. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Bar_chart&oldid=1139941425

Box plot. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Box_plot&oldid=1139942868

Correlation. (2022). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Correlation&oldid=1123244609

Data and information visualization. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Data_and_information_visualization&oldid=1146301693

Data cleansing. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Data_cleansing&oldid=1145894011

Data deduplication. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Data_deduplication&oldid=1142188126

Detect and Remove the Outliers using Python. (2021, February 23). *GeeksforGeeks*. https://www.geeksforgeeks.org/detect-and-remove-the-outliers-using-python/

Exploratory data analysis. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Exploratory_data_analysis&oldid=1145451809

Imputation (statistics). (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Imputation_(statistics)&oldid=1140258627

Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, *349*(6245), 255–260. https://doi.org/10.1126/science.aaa8415

Least squares. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Least_squares&oldid=1139898766

Missing data. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Missing_data&oldid=1143683400

Regression analysis. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Regression_analysis&oldid=1145196383

Scatter plot. (2023). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Scatter_plot&oldid=1139942287

Sohil, F., Sohali, M. U., & Shabbir, J. (2022). An introduction to statistical learning with applications in R: By Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, New York, Springer Science and Business Media, 2013, $41.98, eISBN: 978-1-4614-7137-7. *Statistical Theory and Related Fields*, *6*(1), 87–87. https://doi.org/10.1080/24754269.2021.1980261

Tibshirani, S., & Friedman, H. (n.d.). *Valerie and Patrick Hastie*.

V, N. (n.d.). *Applied Linear Regression Third Edition*. Retrieved March 24, 2023, from https://www.academia.edu/32085924/Applied_Linear_Regression_Third_Edition

7. Appendix A

Important notes (readme):

- Kindly ensure that the database path and file path are correctly specified in the 'db_access.py' file prior to executing the code.
- There exist three distinct approaches to finding ideal functions for the training function, each with its own implementation of the 'train_ideal_mapping()' function. The standard approach, utilizing the least squares method, is presented in Appendix A. To implement the alternative methods, either the 'regression' or 'correlation' approach should be copied one at a time from the 'Additional functions' section of Appendix A and substituted for the existing 'train_ideal_mapping()' function. The remaining code should function correctly thereafter.
- The 'data_analysis.py' file contains 2 functions namely 'appendix_b_standard_execution()' and 'appendix_b_prior_execution()' which makes use of 14 functions from the 'appendix_b.py' file. While some of these functions serve to tally and describe the data, the majority are employed to generate supplementary graphs for the assignment. It should be noted that the function automatically opens multiple browser tabs to display these graphs.
- The 'test_calc.py' file contains functions and unit tests, however, it has not been integrated into the main class. This file may be run independently using the following command: 'python -m unittest test_calc.py'.

Main code:

db_access.py

```
# External Imports
import sqlalchemy
from sqlalchemy import create_engine, MetaData, Table, Column, Integer, String, text, insert, Float
import pandas as pd

db_path= "main_code.db"
file_path= "/Users/ajaychandraas/Desktop/Datasets written assignment-231-4 2/"
"""
The above code sets the db_path variable to the path of the SQLite database that will be created,
and the file_path variable to the path of the directory that contains the CSV files.
"""

class data_access():
    """
            This class contains methods for creating tables in a SQLite database, retrieving data from these tables, and
        connecting to the database
    """

    def __init__(self):
       """
              This is the constructor method of the data_access class. It calls the engine_connect() method to connect to
        the database.
       """
       self.engine_connect()

    def engine_connect(self):
        """
        This function creates a connection to the SQLite database specified by db_path.
        If the connection is successful, it returns the engine and connection objects.
        """
        try:
            engine= create_engine(f'sqlite:///{db_path}', echo=True)
            conn = engine.connect()
            return engine, conn
        except Exception as _error:
            print(f"Error: Database Engine creating unsuccessfull: {_error}")

    def create_train_table(self):
        """
        This function creates a table in the database named training_functions,
        which is populated with data from the train.csv file located in the file_path directory.
        """
        try:
            eng_conn= self.engine_connect()
            path= file_path + "train.csv"
            df = pd.read_csv(path)
            create_table= df.to_sql("training_functions", eng_conn[0])
        except Exception as _error:
            print(f"Error: training_functions table creation was unsuccessfull: {_error}")

    def create_ideal_table(self):
        """
        This function creates a table in the database named ideal_functions,
        which is populated with data from the ideal.csv file located in the file_path directory.
        """
        try:
            eng_conn= self.engine_connect()
            path= file_path + "ideal.csv"
            df= pd.read_csv(path)
            create_table= df.to_sql("ideal_functions", eng_conn[0])
        except Exception as _error:
            print(f"Error: ideal_functions table creation was unsuccessfull: {_error}")

    def create_test_table(self):
        try:
            eng_conn= self.engine_connect()[0]
            conn_db= self.engine_connect()[1]
            """
            reading the test functions data from csv file, sorting the 'x' column values in ascending order
            removing the 'x' duplicate values by taking the mean 'y' values
            """
            path= file_path + "test.csv"
            df= pd.read_csv(path).sort_values(by='x').groupby('x', as_index=False).mean()
            """
            creating the schema for the test table
            """
            meta= MetaData()
            testTable= Table(
                        'test_functions',meta,
                        Column('index',Integer,primary_key=True),
                        Column('x',Float),
                        Column('y',Float),
                        Column('Delta Y',Float),
                        Column('No. of ideal func',String)
            )
            meta.create_all(eng_conn)
            """
            inserting the x and y columns line by line to the test table
            """
            for rec in range(0,len(df)):
                c1= df.iloc[rec][0]
                c2= df.iloc[rec][1]
                ins= testTable.insert().values(x=c1, y=c2)
                result= conn_db.execute(ins)
        except Exception as _error:
            print(f"Error: test_functions table creation was unsuccessfull: {_error}")

    def get_train_table(self):
        """
            This function reads the table "training_functions" from SQL database and returns a Pandas DataFrame using
        SQLAlchemy
        """
        conn_db= self.engine_connect()[1]
        get_train= pd.read_sql_table("training_functions", conn_db)
        return get_train

    def get_ideal_table(self):
        """
        This function reads the table "ideal_functions" from SQL database and returns a Pandas DataFrame using SQLAlchemy
        """
        conn_db= self.engine_connect()[1]
        get_ideal= pd.read_sql_table("ideal_functions", conn_db)
        return get_ideal

    def get_test_table(self):
        """
        This function reads the table "test_functions" from SQL database and returns a Pandas DataFrame using SQLAlchemy
        """
        conn_db= self.engine_connect()[1]
        get_test= pd.read_sql_table("test_functions", conn_db)
        return get_test
```

data_analysis.py

```
# External imports
import math
import pandas as pd
import numpy as np
from scipy.stats import linregress
import sqlalchemy
from sqlalchemy import create_engine, MetaData, insert, update, select
from bokeh.plotting import figure, show
# Internal imports
from db_access import data_access
from appendix_b import appendixB

class data_analysis_class():
    """
    This class contains the following methods:
        - map train and ideal functions
        - finding the maximum deviation of train and ideal functions
        - computing the absolute difference between test and ideal functions
        - mapping the ideal functions to the test points
        - plotting the mapped test points
        - executing the functions in Appendix B of the assignment present in appendix_b.py
    """

    def __init__(self):
        """
        This is the constructor method of the data_analysis_class and is used to to
        create and properly initialize objects of the required classes, making those objects ready to use.
        """
        self.db_accs_Instance= data_access()
        self.db_accs_Instance.create_train_table()
        self.db_accs_Instance.create_ideal_table()
        self.db_accs_Instance.create_test_table()
        self.train_ideal_mapping()
        self.train_ideal_maxDeviation()
        self.test_ideal_abs_diff()
        self.apxB_inst= appendixB()

    def train_ideal_mapping(self):
        """
        Code for Least-Squares Approach
                This function matches each train function with every ideal function, computes the difference between the
        squared 'y' values and sums up the values.
        Next, it identifies the column that returns the minimum value and it assigns that ideal function to the current
        training function.
        It repeats the process for all the training function and returns a dictionary which matches each training with its
        ideal function.
        """
        final_dict={}
        # The for loop iterates over each column name in training functions, excluding the 'x' and 'index' columns
        train_col_name= self.db_accs_Instance.get_train_table().columns.tolist()
        for iter_train in train_col_name:
            if(iter_train!= 'x' and iter_train!= 'index'):
                train_column= self.db_accs_Instance.get_train_table().loc[:,iter_train].values
                train_dict={}
                # The for loop iterates over each column name in ideal functions, excluding the 'x' and 'index' columns
                ideal_col_name= self.db_accs_Instance.get_ideal_table().columns.tolist()
                for iter_ideal in ideal_col_name:
                    if(iter_ideal!= 'x' and iter_ideal!= 'index'):
                        ideal_column= self.db_accs_Instance.get_ideal_table().loc[:,iter_ideal].values
                        # Computing the difference between train and ideal and summing up the values
                        diff= sum((train_column-ideal_column)**2)
                        train_dict[iter_ideal]=diff
                        min_value= min(train_dict.values())
                        # Finding the ideal function that has the minimum value
                        min_keys = [key for key in train_dict if train_dict[key]==min_value]
                        final_dict[iter_train]=min_keys[0]
        # Returning a dictionary with the training functions along with their respective ideal functions
        return final_dict

    def train_ideal_maxDeviation(self):
        """
                This function finds the maximum deviation between the training and ideal data for each function in the
        mapping by calculating their absolute difference multiplied by the square root of 2.
        It returns the maximum deviation value for each function in a dictionary.
        """
        train_dict={}
        # The for loop iterates over each column name in training functions, excluding the 'x' and 'index' columns
        train_col_name= self.db_accs_Instance.get_train_table().columns.tolist()
        for iter_train in train_col_name:
            if(iter_train!= 'x' and iter_train!= 'index'):
                ideal_fns= self.train_ideal_mapping()
                for iter_ideal in ideal_fns:
                    if(iter_ideal==iter_train):
                        train_column= self.db_accs_Instance.get_train_table().loc[:,iter_train].values
                        ideal_column= self.db_accs_Instance.get_ideal_table().loc[:,ideal_fns.get(iter_ideal)].values
                        # Computing the absolute difference between train and ideal function, multiplying it with sqrt(2)
and returning the maximum value
                        diff= max(abs(train_column-ideal_column)) * math.sqrt(2)
                        train_dict[ideal_fns.get(iter_ideal)]=diff
        # Returning a dictionary containing ideal functions along with their respective max deviation values
        return train_dict

    def test_ideal_abs_diff(self):
        """
                This function returns the absolute difference of 'y' values between test function and the 4 selected ideal
        functions by matching 'x' their values.
        """
        # Initialize mainList with x-values from the test table
        mainList= []
        testTable_x_values= self.db_accs_Instance.get_test_table().loc[:,'x'].values.tolist()
        mainList.append(testTable_x_values)
        # Iterate over ideal mapping functions and calculate absolute differences
        train_ideal_mapping_fn= self.train_ideal_mapping()
        for value in train_ideal_mapping_fn.values():
            # Get x and y values for the ideal table and calculate estimated y-values using linear regression
            ideal_x_values= self.db_accs_Instance.get_ideal_table().loc[:,'x'].values.tolist();
            ideal_x_df= pd.DataFrame(ideal_x_values)
            ideal_column= self.db_accs_Instance.get_ideal_table().loc[:,value].values;
            ideal_y_df= pd.DataFrame(ideal_column)
            slope, intercept, r_value, p_value, std_err= linregress(ideal_x_values, ideal_column)
            x_arr_values= np.array(ideal_x_values)
            y_arr_values= np.array(ideal_column)
            est_y= (slope * x_arr_values) + intercept
            # Create a DataFrame with the estimated y-values and the x-values from the ideal table
            est_y_df= pd.DataFrame(est_y)
            y_dataframe= pd.concat([ideal_x_df, est_y_df],axis=1); y_dataframe.columns=['x','y']
            # Merge the DataFrame with the estimated y-values with the test table and calculate the absolute difference
            df1= y_dataframe; df2= self.db_accs_Instance.get_test_table()
            merged_df= pd.merge(df1, df2, on='x'); merged_df.columns=['x','y_ideal','index','y_test','dummy1','dummy2']
            diff= abs(merged_df['y_test']- merged_df['y_ideal']); final_diff= pd.DataFrame(diff); final_diff.columns=['y']
            final_df= pd.concat([merged_df['x'], final_diff],axis=1); final_df.columns=['x','y_diff1']
            # Append the absolute difference values to mainList
            y_values= final_diff.loc[:,'y'].values.tolist()
            mainList.append(y_values)
        # Create the final table with column names
        final_table= pd.DataFrame(mainList).transpose()
        ideal_value_head= train_ideal_mapping_fn.values()
        ideal_value_list= []
        ideal_value_list.insert(0, 'x')
        for ideal_rec in ideal_value_head:
            ideal_value_list.append(ideal_rec)
        final_table.columns= ideal_value_list
        return final_table

    def map_ideal_to_test(self):
        """
                This function compares the values computed by test_ideal_abs_diff() function with the maximum deviation
        values computed in train_ideal_maxDeviation() function.
        If the result is less than or equal to max dev value then that corresponding ideal function and its respective
        difference value is updated in the test_functions table.
        """
        # The following code extracts x and y values from a test table using a database access class instance. It then
creates an empty list called "data_list" to hold data that will be inserted into the test table.
        x_values= self.db_accs_Instance.get_test_table().loc[:,'x'].values.tolist()
        y_values= self.db_accs_Instance.get_test_table().loc[:,'y'].values.tolist()
        data_list= []
        # The following code establishs a database connection and metadata information for the database.
        eng_conn= self.db_accs_Instance.engine_connect()
        meta = MetaData(bind=eng_conn[0])
        MetaData.reflect(meta)
        conn= eng_conn[1]
        # The following code creates a list by extracting the names of the ideal functions from the "train_ideal_mapping"
function.
        ideal_list= []
        train_ideal_mapping_fn= self.train_ideal_mapping()
        ideal_fn_names= train_ideal_mapping_fn.values()
        for id_rec in ideal_fn_names:
            ideal_list.append(id_rec)
        # The following code loops through each ideal function and performs calculations to map the ideal function to the
test function based on the criteria
        for ideal_fn_rec in ideal_list:
            train_ideal_max_dev_fn= self.train_ideal_maxDeviation()
            y_max= train_ideal_max_dev_fn.get(ideal_fn_rec)
            test_ideal_abs_diff_fn= self.test_ideal_abs_diff()
            y_diff= test_ideal_abs_diff_fn.loc[:,ideal_fn_rec].values.tolist()
            for x_rec, y_rec, y_val_rec in zip(x_values, y_diff, y_values):
                bulk_dict= {}
                # The following code checks if the value meets the max deviation criteria
                if(y_rec <= y_max):
                    # The following code checks if 'x' matches with more than 1 ideal function using select query
                    test_table= meta.tables['test_functions']
                    sel= sqlalchemy.select(test_table).where(test_table.c.x == x_rec)
                    check_x= conn.execute(sel).fetchall()[0][3]
                    if(check_x == None):
                        # The following code updates 'Delta Y' and 'No. of ideal func' values
                        u = update(test_table)
                        u = u.values({ "Delta Y": y_rec, "No. of ideal func": ideal_fn_rec})
                        u= u.where(test_table.c.x == x_rec)
                        conn= eng_conn[1]
                        result= conn.execute(u)
                    if(check_x != None ):
                        check_ideal_y= conn.execute(sel).fetchall()[0][3]
                        if(check_ideal_y != y_rec):
                            # Adding items to dictionary and then to the list to create data object
                            bulk_dict["x"]=x_rec
                            bulk_dict["y"]=y_val_rec
                            bulk_dict["Delta Y"]=y_rec
                            bulk_dict["No. of ideal func"]=ideal_fn_rec
                            data_list.append(bulk_dict)
        # The following code inserts the data in "data_list" into the test table.
        ins= test_table.insert().values(data_list)
        conn= eng_conn[1]
        res= conn.execute(ins)
        # The following code orders the data in the test table based on 'x' and also updates the values of unmatched
"Delta Y" and "No. of ideal func" column
        ord_list= []
        test_table= meta.tables['test_functions']
        sel= sqlalchemy.select(test_table).order_by(test_table.c.x.asc())
        fetch= conn.execute(sel).fetchall()
        for rec in fetch:
            ord_list.append(rec)
        # Using a counter to match the where clause
        count= 1
        for ord_rec in ord_list:
            if(ord_rec[4] != None):
                u = update(test_table)
                u = u.values({ "x":ord_rec[1], "y":ord_rec[2],
                            "Delta Y": ord_rec[3], "No. of ideal func": ord_rec[4]}).where(test_table.c.index == count)
                conn= eng_conn[1]
                result= conn.execute(u)
                count+= 1
            else:
                u = update(test_table)
                u = u.values({ "x":ord_rec[1], "y":ord_rec[2],
                            "Delta Y": 0, "No. of ideal func": "N/A"}).where(test_table.c.index == count)
                conn= eng_conn[1]
                result= conn.execute(u)
                count+= 1

    def mapping_test_points(self):
        """
        Generates scatter plots of test data points and their ideal best fit lines with max deviation boundaries.
        """
        # The following code connect to the database and retrieve the necessary tables and data
        eng_conn= self.db_accs_Instance.engine_connect()
        meta = MetaData(bind=eng_conn[0])
        MetaData.reflect(meta)
        conn= eng_conn[1]
        test_table= meta.tables['test_functions']
        # Create a list of ideal function names.
        ideal_list= []
        train_ideal_mapping_fn= self.train_ideal_mapping()
        ideal_fn_names= train_ideal_mapping_fn.values()
        for id_rec in ideal_fn_names:
            ideal_list.append(id_rec)
        # Generate scatter plots for each train function.
        for ideal_rec in ideal_list:
            # Retrieve the test data for the current ideal function.
            sel= sqlalchemy.select(test_table).where(test_table.c['No. of ideal func'] == ideal_rec)
            sel_y= conn.execute(sel).fetchall()
            xlst= []; ylst= []
            for rec in sel_y:
                xlst.append(rec[1])
                ylst.append(rec[2])
            # Generate the scatter plot with the ideal best fit line and max deviation boundaries.
            train_ideal_maxDev_fn= self.train_ideal_maxDeviation()
            max_dev_val= round(train_ideal_maxDev_fn[ideal_rec],3)
            title= f"{ideal_rec} Scatter plot with ideal best fit line with max boundary of {max_dev_val}"
            p= figure(title= title, x_axis_label='X axis', y_axis_label='Y axis')
            p.title.text_font_size = '13pt'
            p.scatter(xlst, ylst, size= 10)
            # Add the ideal best fit line to the plot.
            ideal_list= [ideal_rec]
            for plot_rec in ideal_list:
                ideal_x= self.db_accs_Instance.get_ideal_table().loc[:,'x'].values
                ideal_y= self.db_accs_Instance.get_ideal_table().loc[:,plot_rec].values
                slope, intercept, r_value, p_value, std_err= linregress(ideal_x, ideal_y)
                xs = np.linspace(np.min(ideal_x), np.max(ideal_x), 100)
                ys = slope * xs + intercept
                p.line(xs, ys, color='red')
                # Add the maximum deviation as upper and lower bounds to the scatter plot
                x_values= np.array(ideal_x)
                est_y= (slope * x_values) + intercept
                pos_dev_est= est_y + train_ideal_maxDev_fn[plot_rec]
                neg_dev_est= est_y - train_ideal_maxDev_fn[plot_rec]
                p.line(ideal_x, pos_dev_est, color='red')
                p.line(ideal_x, neg_dev_est, color='red')
            # Show the plot
            try:
                show(p)
            except Exception as _error:
                print(f"Error: Unable to generate the graph: {_error}")
    def appendix_b_prior_execution(self):
        """
        This function calls various functions of an instance of AppendixB class.
                This function is responsible for generating different graphs and visualizations as well as counting and
        describing certain parameters.
        """
        self.apxB_inst.train_test_desc()
        self.apxB_inst.train_outliers()
        self.apxB_inst.test_outliers()
        self.apxB_inst.train_box_plots()
        self.apxB_inst.test_box_plot()
        self.apxB_inst.find_dup_train()
        self.apxB_inst.find_dup_test()
        self.apxB_inst.missing_values()
        # Append the output of the functions to a dictionary and print (to make it easier to check values in the Terminal)
        object_dict= {
                      "Train & Test Desc":self.apxB_inst.train_test_desc(),
                      "Train Outliers": self.apxB_inst.train_outliers(),
                      "Test Outliers": self.apxB_inst.test_outliers(),
                      "Train Duplicates": self.apxB_inst.find_dup_train(),
                      "Test Duplicates": self.apxB_inst.find_dup_test(),
                      "Train & Test Missing values": self.apxB_inst.missing_values()
                        }
        print(object_dict)

    def appendix_b_standard_execution(self):
        self.apxB_inst.bestfit_scatter_plots()
        self.apxB_inst.bar_chart_regression()
        self.apxB_inst.bar_chart_correlation()
        self.apxB_inst.bar_chart_leastSquares()
        self.apxB_inst.test_ideal_mapping_count()
        self.apxB_inst.counter_pie_chart()

#start of main code
if __name__ == "__main__":
    """
    This code block is the entry point of the script/application.
    It creates an instance of the data_analysis_class and calls its map_ideal_to_test, mapping_test_points,
    and appendix_b_fns methods to execute the main functionality of the script.
    """
    data_analysis= data_analysis_class()
    data_analysis.map_ideal_to_test()
    data_analysis.mapping_test_points()
    data_analysis.appendix_b_standard_execution()
    data_analysis.appendix_b_prior_execution()
```

8. Additional functions

Regression and Correlation approaches to find ideal function

```
def train_ideal_mapping(self):
        """
        Code for Regression Approach
        This function computes regression values of training and ideal function.
                Next, it matches each train function with every ideal function, computes the absolute difference between
        the best fit values and sums up the values.
        Subsequently, it identifies the column that returns the minimum value and it assigns that ideal function to the
        current training function.
        It repeats the process for all the training function and returns a dictionary which matches each training with its
        ideal function.
        """
        main_dict= {}
        train_x_values= self.db_accs_Instance.get_train_table().loc[:,'x'].values;
        # The for loop iterates over each column name in training functions, excluding the 'x' and 'index' columns
        train_col_name= self.db_accs_Instance.get_train_table().columns.tolist()
        for train_rec in train_col_name:
            secondary_dict= {}
            if(train_rec!= 'x' and train_rec!= 'index'):
                train_column= self.db_accs_Instance.get_train_table().loc[:,train_rec].values;
                # Computing regression values of train functions
                slope1, intercept1, r_value1, p_value1, std_err1= linregress(train_x_values, train_column)
                train_x_arr_values= np.array(train_x_values)
                train_y_arr_values= np.array(train_column)
                train_est_y= (slope1 * train_x_arr_values) + intercept1
                train_est_y_df= pd.DataFrame(train_est_y)
                ideal_x_values= self.db_accs_Instance.get_ideal_table().loc[:,'x'].values;
                ideal_x_df= pd.DataFrame(ideal_x_values)
                # The for loop iterates over each column name in ideal functions, excluding the 'x' and 'index' columns
                ideal_col_name= self.db_accs_Instance.get_ideal_table().columns.tolist()
                for ideal_rec in ideal_col_name:
                    if(ideal_rec!= 'x' and ideal_rec!= 'index'):
                        ideal_column= self.db_accs_Instance.get_ideal_table().loc[:,ideal_rec].values;
                        # Computing regression values of ideal functions
                        slope2, intercept2, r_value2, p_value2, std_err2= linregress(ideal_x_values, ideal_column)
                        ideal_x_arr_values= np.array(ideal_x_values)
                        ideal_y_arr_values= np.array(ideal_column)
                        ideal_est_y= (slope2 * ideal_x_arr_values) + intercept2
                        ideal_est_y_df= pd.DataFrame(ideal_est_y)
                        # Finding the absolute difference between train and ideal best fit values and summing the values
                        diff= sum(abs(train_est_y- ideal_est_y))
                        secondary_dict[ideal_rec]= diff
                # Finding the ideal function that has the minimum value
                min_value= min(secondary_dict.values())
                min_keys = [key for key in secondary_dict if secondary_dict[key]==min_value]
                main_dict[train_rec]=min_keys[0]
        # Returning a dictionary with the training functions along with their respective ideal functions
        return main_dict

    def train_ideal_mapping(self):
        """
        Code for Correlation Approach
                This function matches each train function with every ideal function, computes the correlation between the
        'y' values.
        Next, it identifies the column that returns the maximum value and it assigns that ideal function to the current
        training function.
        It repeats the process for all the training function and returns a dictionary which matches each training with its
        ideal function.
        """
        final_dict={}
        # The for loop iterates over each column name in training functions, excluding the 'x' and 'index' columns
        train_index= self.db_accs_Instance.get_train_table().columns.tolist()
        for iter_train in train_index:
            if(iter_train!= 'x' and iter_train!= 'index'):
                train_column= self.db_accs_Instance.get_train_table().loc[:,iter_train].values
                train_df= pd.DataFrame(train_column, columns = ['y'])
                train_dict={}
                # The for loop iterates over each column name in ideal functions, excluding the 'x' and 'index' columns
                ideal_index_list= self.db_accs_Instance.get_ideal_table().columns.tolist()
                for iter_ideal in ideal_index_list:
                    if(iter_ideal!= 'x' and iter_ideal!= 'index'):
                        ideal_column= self.db_accs_Instance.get_ideal_table().loc[:,iter_ideal].values
                        ideal_df= pd.DataFrame(ideal_column, columns = ['y'])
                        # Computing the correlation between the 'y' values of train and ideal function
                        calc_correl= train_df['y'].corr(ideal_df['y'])
                        train_dict[iter_ideal]=calc_correl
                # Finding the ideal function that has the maximum value
                max_value= max(train_dict.values())
                max_keys = [key for key in train_dict if train_dict[key]==max_value]
                # Checking if a ideal function is already assigned to the train. If yes, assign the ideal function to the
training function that has the second maximum correaltion between them.
                if(max_keys[0] in final_dict.values()):
                    values_list = list(train_dict.values())
                    values_list.sort(reverse=True)
                    second_largest_value = values_list[1]
                    for key, value in train_dict.items():
                        if value == second_largest_value:
                            final_dict[iter_train]=key
                else:
                    final_dict[iter_train]=max_keys[0]
        # Returning a dictionary with the training functions along with their respective ideal functions
        return final_dict
```

test_calc.py

```
# External imports
import unittest
# Internal imports
from data_analysis import data_analysis_class

class TestCalc(unittest.TestCase):
    """
    This class defines the unit tests for the methods in the data_analysis_class.
    """

    def testing_train_ideal_mapping_fn(self):
        """
        Tests the train_ideal_mapping method in the data_analysis_class.
        It checks that the method returns a non-None object and that the length of the returned object is 4.
        """
        obj= data_analysis_class()
        self.assertIsNotNone(obj.train_ideal_mapping())
        self.assertEqual(len(obj.train_ideal_mapping()), 4)


    def testing_train_ideal_maxDeviation_fn(self):
        """
        Tests the train_ideal_maxDeviation method in the data_analysis_class.
        It checks that the method returns a non-None object and that the length of the returned object is 4.
        """
        obj= data_analysis_class()
        self.assertIsNotNone(obj.train_ideal_maxDeviation())
        self.assertEqual(len(obj.train_ideal_maxDeviation()), 4)

    def testing_test_ideal_abs_diff_fn(self):
        """
        Tests the test_ideal_abs_diff method in the data_analysis_class.
        It checks that the method returns a non-None object.
        """
        obj= data_analysis_class()
        self.assertIsNotNone(obj.test_ideal_abs_diff())
```

9. Appendix B

appendix_b.py

```
# External imports
from bokeh.plotting import figure, show
from bokeh.models import ColumnDataSource, Label, Whisker
from bokeh.transform import factor_cmap
from scipy.stats import linregress
import numpy as np
import pandas as pd
import math
from collections import Counter
# Internal imports
from db_access import data_access
from db_access import file_path

class appendixB():
    """
    This class contains all the functions that are part of the appendix B of the assignment
    """
    def __init__(self):
        """
        Initializes the class instance and data frames using the "data_access" class.
        """
        self.db_access_class_instance_2= data_access()
        self.train_df= pd.DataFrame(self.db_access_class_instance_2.get_train_table())
        self.ideal_df= pd.DataFrame(self.db_access_class_instance_2.get_ideal_table())
        self.test_df= pd.DataFrame(self.db_access_class_instance_2.get_test_table())
        self.test_table_read_2()

    def test_table_read_2(self):
        """
        Reading the test functions data from csv file
        """
        path= file_path + "test.csv"
        test_read= pd.read_csv(path).sort_values(by='x').groupby('x', as_index=False).mean()
        test_df= pd.DataFrame(test_read)
        test_read_raw= pd.read_csv(path)
        return test_read, test_df, test_read_raw

    def train_test_desc(self):
        """
        Prints descriptive statistics of the train and test tables using the "describe" function.
        """
        tr_desc= self.db_access_class_instance_2.get_train_table().describe(); print(tr_desc)
        tst_desc= self.test_table_read_2()[0].describe(); print(tst_desc)

    def train_outliers(self):
        """
                Calculates the upper and lower bounds for each column of the train table and counts the number of outliers
        in each column.
        """
        outier_dict= {}
        col_name= self.train_df.columns.tolist()
        for rec in col_name:
            bounds_dict= {}
            if(rec!= 'x' and rec!= 'index'):
                y_values= self.train_df.loc[:,rec].values.tolist()
                # Finding IQR
                Q1 = np.percentile(y_values, 25, interpolation = 'midpoint')
                Q3 = np.percentile(y_values, 75, interpolation = 'midpoint')
                IQR = Q3 - Q1
                # Finding upper and lower bounds
                uppr_bound= Q3+1.5*IQR
                lowr_bound= Q1-1.5*IQR
                upprcount= 0
                lwrcount= 0
                # Counting the outliers in each column
                for y_rec in y_values:
                    if(y_rec> uppr_bound):
                        upprcount+= 1
                    elif(y_rec< lowr_bound):
                        lwrcount+= 1
                    else:
                        pass
                bounds_dict['upper bound']= upprcount
                bounds_dict['lower bound']= lwrcount
                outier_dict[rec]= bounds_dict
        return(outier_dict)
        print(outier_dict)

    def test_outliers(self):
        """
        Calculates the upper and lower bounds of the test table and counts the number of outliers.
        """
        outier_dict= {}
        bounds_dict= {}
        y_values= self.test_table_read_2()[1].loc[:,'y'].values.tolist()
        # Finding IQR
        Q1 = np.percentile(y_values, 25, interpolation = 'midpoint')
        Q3 = np.percentile(y_values, 75, interpolation = 'midpoint')
        IQR = Q3 - Q1
        # Finding upper and lower bounds
        uppr_bound= Q3+1.5*IQR
        lowr_bound= Q1-1.5*IQR
        upprcount= 0
        lwrcount= 0
        for y_rec in y_values:

            if(y_rec> uppr_bound):
                upprcount+= 1
            elif(y_rec< lowr_bound):
                lwrcount+= 1
            else:
                pass
            bounds_dict['upper bound']= upprcount
            bounds_dict['lower bound']= lwrcount
            outier_dict['y']= bounds_dict
        return(outier_dict)
        print(outier_dict)

    def train_box_plots(self):
        """
        Generate box plots for each column in the training data.
        """
        train_list= self.train_df.columns.tolist()
        const_dict= {}; const_name= []; const_y= []; data_dict= {}
        q1_list= []; q2_list= []; q3_list= []
        for rec1 in train_list:
            if(rec1!= 'x' and rec1!= 'index'):
                name_lst= [rec1]*len(self.train_df)
                y_lst= self.train_df[rec1].tolist()
                const_name.extend(name_lst)
                const_y.extend(y_lst)
        const_dict['Train']= const_name
        const_dict['y']= const_y
        df= pd.DataFrame(const_dict)

        trains = df.Train.unique()
        qs = df.groupby("Train").y.quantile([0.25, 0.5, 0.75])
        qs = qs.unstack().reset_index()
        qs.columns = ["Train", "q1", "q2", "q3"]
        df = pd.merge(df, qs, on="Train", how="left")
        # compute IQR outlier bounds
        iqr = df.q3 - df.q1
        df["upper"] = df.q3 + 1.5*iqr
        df["lower"] = df.q1 - 1.5*iqr
        source = ColumnDataSource(df)
        p = figure(x_range=trains, tools="", toolbar_location=None,
                title="Train Box plots",
                background_fill_color="#eaefef", y_axis_label="train")
        # outlier range
        whisker = Whisker(base="Train", upper="upper", lower="lower", source=source)
        whisker.upper_head.size = whisker.lower_head.size = 20
        p.add_layout(whisker)
        # quantile boxes
        color1 = ["pink", "yellow", "lightgreen","orange", "red"]
        cmap = factor_cmap("Train", color1, trains)
        p.vbar("Train", 0.7, "q2", "q3", source=source, color=cmap, line_color="black")
        p.vbar("Train", 0.7, "q1", "q2", source=source, color=cmap, line_color="black")
        # outliers
        outliers = df[~df.y.between(df.lower, df.upper)]
        p.scatter("Train", "y", source=outliers, size=6, color="black", alpha=0.3)
        p.xgrid.grid_line_color = None
        p.axis.major_label_text_font_size="18px"
        p.axis.axis_label_text_font_size="18px"
        try:
            show(p)
        except Exception as _error:
            print(f"Error: Unable to generate the graph: {_error}")

    def test_box_plot(self):
        """
        Generate box plots for each column in the test data.
        """
        const_dict= {}
        name_lst= ['y']*len(self.test_table_read_2()[1])
        y_lst= self.test_table_read_2()[1]['y'].tolist()
        const_dict['Test']= name_lst
        const_dict['y']= y_lst
        df= pd.DataFrame(const_dict)
        tests = df.Test.unique()
        qs = df.groupby("Test").y.quantile([0.25, 0.5, 0.75])
        qs = qs.unstack().reset_index()

        qs.columns = ["Test", "q1", "q2", "q3"]
        df = pd.merge(df, qs, on="Test", how="left")
        # compute IQR outlier bounds
        iqr = df.q3 - df.q1
        df["upper"] = df.q3 + 1.5*iqr
        df["lower"] = df.q1 - 1.5*iqr

        source = ColumnDataSource(df)
        p = figure(x_range=tests, tools="", toolbar_location=None,
                    title="Test Box plot",
                    background_fill_color="#eaefef", y_axis_label="test")
        p.title.text_font_size = '15pt'
        # outlier range
        whisker = Whisker(base="Test", upper="upper", lower="lower", source=source)
        whisker.upper_head.size = whisker.lower_head.size = 20
        p.add_layout(whisker)
        # quantile boxes
        color1 = ["pink", "yellow", "lightgreen","orange", "red"]
        cmap = factor_cmap("Test", color1, tests)
        p.vbar("Test", 0.7, "q2", "q3", source=source, color=cmap, line_color="black")
        p.vbar("Test", 0.7, "q1", "q2", source=source, color=cmap, line_color="black")
        # outliers
        outliers = df[~df.y.between(df.lower, df.upper)]
        p.scatter("Test", "y", source=outliers, size=6, color="black", alpha=0.3)
        p.xgrid.grid_line_color = None
        p.axis.major_label_text_font_size="18px"
        p.axis.axis_label_text_font_size="18px"
        try:
            show(p)
        except Exception as _error:
            print(f"Error: Unable to generate the graph: {_error}")

    def find_dup_train(self):
        """
        Checking for duplicates values in training data
        """
        dup_dict= {}
        col_name= self.train_df.columns.tolist()
        for rec in col_name:
            if(rec != 'index'):
                total_items= len(self.train_df[rec])
                uniq_items= len(self.train_df[rec].unique())
                dup_items= total_items- uniq_items
                dup_dict[rec]= dup_items
        return(dup_dict)
        print(dup_dict)

    def find_dup_test(self):
        """
        Checking for duplicates values in test data
        """
        dup_dict= {}
        col_name= self.test_table_read_2()[1].columns.tolist()
        for rec in col_name:
            if(rec!= 'index' and rec!= 'Delta Y' and rec!= 'No. of ideal func'):
                total_items= len(self.test_table_read_2()[2][rec])
                uniq_items= len(self.test_table_read_2()[2][rec].unique())
                dup_items= total_items- uniq_items
                dup_dict[rec]= dup_items
        return(dup_dict)
        print(dup_dict)


    def missing_values(self):
        """
        Checking for missing values in train and test data
        """
        train_num_missing = self.db_access_class_instance_2.get_train_table().isnull().sum()
        return(train_num_missing)
        print(train_num_missing)
        test_num_missing =self.test_table_read_2()[0].isnull().sum()
        return(train_num_missing)
        print(test_num_missing)

    def bestfit_scatter_plots(self):
        """
        [All approaches]
                Generate scatter plot of training functions with regression lines for specified 'y' columns in the
        training and ideal dataframes
        """
        method_dict= {"Regression":['y2','y38','y43','y46'],
                      "Correlation":['y9','y38','y45','y46'],
                      "Least-Squares":['y9','y38','y43','y46']}
        ideal_list=[]
        train_list= ['y1','y2','y3','y4']
        for key, value in method_dict.items():
            if(key== "Regression"):
                ideal_list= value
            elif(key== "Correlation"):
                ideal_list= value
            else:
                ideal_list= value
             # Get x and y values from training dataframe
            x_values_train= self.train_df.loc[:,'x'].values.tolist()
            for train_rec, ideal_rec in zip(train_list, ideal_list):
                y_values= self.train_df.loc[:,train_rec].values.tolist()
                p= figure(title= f"{key} Approach: Fitting {train_rec} and {ideal_rec} regression lines")
                p.title.text_font_size = '15pt'
                p.scatter(x_values_train, y_values)
                # Calculate regression line for training data
                train_x= self.train_df.loc[:,'x'].values
                train_y= self.train_df.loc[:,train_rec].values
                slope1, intercept1, r_value1, p_value1, std_err1= linregress(train_x, train_y)
                xs1 = np.linspace(np.min(train_x), np.max(train_x))
                ys1 = slope1 * xs1 + intercept1
                p.line(xs1, ys1, color='green', line_width=2.5, legend_label='Train regression line')
                 # Calculate regression line for ideal data
                ideal_x= self.ideal_df.loc[:,'x'].values
                ideal_y= self.ideal_df.loc[:,ideal_rec].values
                slope2, intercept2, r_value2, p_value2, std_err2= linregress(ideal_x, ideal_y)
                xs2 = np.linspace(np.min(ideal_x), np.max(ideal_x))
                ys2 = slope2 * xs2 + intercept2
                p.line(xs2, ys2, color='red', line_width=2.5, legend_label='Ideal regression line')
                # Show the plot
                try:
                    show(p)
                except Exception as _error:
                    print(f"Error: Unable to generate the graph: {_error}")

    def bar_chart_regression(self):
        """
        [Regression approach]
```

```
            Generates a bar chart displaying the sum of the absolute deviation of regression values for every ideal
        function and a given train function.
        """
        main_dict= {}
        # The for loop iterates over each column name in training functions, excluding the 'x' and 'index' columns
        train_x_values= self.db_access_class_instance_2.get_train_table().loc[:,'x'].values
        train_col_name= self.db_access_class_instance_2.get_train_table().columns.tolist()
        for train_rec in train_col_name:
            if(train_rec!= 'x' and train_rec!= 'index'):
                secondary_dict= {}
                train_column= self.db_access_class_instance_2.get_train_table().loc[:,train_rec].values
                # Computing regression values of train functions
                slope1, intercept1, r_value1, p_value1, std_err1= linregress(train_x_values, train_column)
                train_x_arr_values= np.array(train_x_values)
                train_y_arr_values= np.array(train_column)
                train_est_y= (slope1 * train_x_arr_values) + intercept1
                train_est_y_df= pd.DataFrame(train_est_y)
                # The for loop iterates over each column name in ideal functions, excluding the 'x' and 'index' columns
                ideal_x_values= self.db_access_class_instance_2.get_ideal_table().loc[:,'x'].values;
                ideal_x_df= pd.DataFrame(ideal_x_values)
                ideal_col_name= self.db_access_class_instance_2.get_ideal_table().columns.tolist()
                for ideal_rec in ideal_col_name:
                    if(ideal_rec!= 'x' and ideal_rec!= 'index'):
                        ideal_column= self.db_access_class_instance_2.get_ideal_table().loc[:,ideal_rec].values;
                        # Computing regression values of ideal functions
                        slope2, intercept2, r_value2, p_value2, std_err2= linregress(ideal_x_values, ideal_column)
                        ideal_x_arr_values= np.array(ideal_x_values)
                        ideal_y_arr_values= np.array(ideal_column)
                        ideal_est_y= (slope2 * ideal_x_arr_values) + intercept2
                        ideal_est_y_df= pd.DataFrame(ideal_est_y)
                        # Finding the absolute difference between train and ideal best fit values and summing the values
                        diff= sum(abs(train_est_y- ideal_est_y))
                        secondary_dict[ideal_rec]= diff
                # Finding the ideal function that has the minimum value
                min_value= min(secondary_dict.values())
                min_keys = [key for key in secondary_dict if secondary_dict[key]==min_value]
                # Computing the values and properties for the bar chart
                key_list= list(secondary_dict.keys())
                value_list = np.log10(list(secondary_dict.values()))
                title= f"Regression approach: Finding ideal for '{train_rec}' train functon"
                p = figure(title= title, x_range=key_list, plot_width=1300, plot_height=300)
                p.yaxis.major_label_text_font_size = '8pt'; p.xaxis.major_label_text_font_size = '11pt'
                p.title.text_font_size = '13pt'
                index = key_list.index(min_keys[0])
                colors = ['blue' if i != index else 'red' for i in range(len(value_list))]
                p.vbar(x= key_list, top= value_list, width=0.5, color=colors)
                label = Label(x=1, y=10, text='Min Value', text_font_size='13pt', text_color='red')
                p.add_layout(label)
                label.x = 45; label.y = 6; label.y_offset = 1
                # Show the plot
                try:
                    show(p)
                except Exception as _error:
                    print(f"Error: Unable to generate the graph: {_error}")

    def bar_chart_correlation(self):
        """
        [Correlation approach]
        Generates a bar chart displaying correlation values for every ideal function and a given train function.
        """
        ideal_dict= {'y1': 'y9', 'y2': 'y38', 'y3': 'y45', 'y4': 'y46'}
        # The for loop iterates over each column name in training functions, excluding the 'x' and 'index' columns
        train_col_name= self.db_access_class_instance_2.get_train_table().columns.tolist()
        for train_rec in train_col_name:
            if(train_rec!= 'x' and train_rec!= 'index'):
                secondary_dict= {}
                train_column= self.db_access_class_instance_2.get_train_table().loc[:,train_rec].values;
                train_df= pd.DataFrame(train_column, columns = ['y'])
                # The for loop iterates over each column name in ideal functions, excluding the 'x' and 'index' columns
                ideal_col_name= self.db_access_class_instance_2.get_ideal_table().columns.tolist()
                for ideal_rec in ideal_col_name:
                    if(ideal_rec!= 'x' and ideal_rec!= 'index'):
                        ideal_column= self.db_access_class_instance_2.get_ideal_table().loc[:,ideal_rec].values
                        ideal_df= pd.DataFrame(ideal_column, columns = ['y'])
                        # Finding the correlation between train and ideal best fit values
                        calc_correl= train_df['y'].corr(ideal_df['y'])
                        secondary_dict[ideal_rec]=calc_correl
                max_value= max(secondary_dict.values())
                max_keys = [key for key in secondary_dict if secondary_dict[key]==max_value]
                key_list= list(secondary_dict.keys())
                value_list1= list(secondary_dict.values())
                value_list2= [i * 100000 for i in value_list1]
                replace_neg = [max(0, i) for i in value_list2]
                place_zero = [i if i != 0 else 1 for i in replace_neg]
                new_value_list = np.log10(place_zero).tolist()
                # Computing the values and properties for the bar chart
                title= f"Correlation approach: Finding ideal for '{train_rec}' train functon"
                p = figure(title= title, x_range=key_list, plot_width=1300, plot_height=300)
                p.yaxis.major_label_text_font_size = '8pt'; p.xaxis.major_label_text_font_size = '11pt'
                p.title.text_font_size = '13pt'
                index = key_list.index(max_keys[0])
                colors = ['blue' if i != ideal_dict[train_rec] else 'red' for i in key_list]
                p.vbar(x= key_list, top= new_value_list, width=0.5, color=colors)
                label = Label(x=1, y=10, text='Max Value', text_font_size='10pt', text_color='red')
                p.add_layout(label)
                label.x = 47; label.y = 4.75; label.y_offset = 1
                # Show the plot
                try:
                    show(p)
                except Exception as _error:
                    print(f"Error: Unable to generate the graph: {_error}")


    # Bar chart to show ideal and y values [Least-squares approach] executed for individual train function
    def bar_chart_leastSquares(self):
        """
        [Least-squares approach]
        Generates a bar chart displaying the sum of the squared deviation of ideal function and a given train function.
        """
        main_dict= {}
        # The for loop iterates over each column name in training functions, excluding the 'x' and 'index' columns
        train_x_values= self.db_access_class_instance_2.get_train_table().loc[:,'x'].values
        train_col_name= self.db_access_class_instance_2.get_train_table().columns.tolist()
        for train_rec in train_col_name:
            if(train_rec!= 'x' and train_rec!= 'index'):
                secondary_dict= {}
                train_column= self.db_access_class_instance_2.get_train_table().loc[:,train_rec].values
                ideal_x_values= self.db_access_class_instance_2.get_ideal_table().loc[:,'x'].values;
                ideal_x_df= pd.DataFrame(ideal_x_values)
                # The for loop iterates over each column name in ideal functions, excluding the 'x' and 'index' columns
                ideal_col_name= self.db_access_class_instance_2.get_ideal_table().columns.tolist()
                for ideal_rec in ideal_col_name:
                    if(ideal_rec!= 'x' and ideal_rec!= 'index'):
                        ideal_column= self.db_access_class_instance_2.get_ideal_table().loc[:,ideal_rec].values
                        # Finding the squared difference between train and ideal functions and summing the values
                        diff= sum((train_column- ideal_column)**2)
                        secondary_dict[ideal_rec]= diff
                min_value= min(secondary_dict.values())
                min_keys = [key for key in secondary_dict if secondary_dict[key]==min_value]
                key_list= list(secondary_dict.keys())
                value_list = np.log10(list(secondary_dict.values()))
                # Computing the values and properties for the bar chart
                title= f"Least-squares approach: Finding ideal for '{train_rec}' train functon"
                p = figure(title= title, x_range=key_list, plot_width=1300, plot_height=300)
                p.yaxis.major_label_text_font_size = '8pt'; p.xaxis.major_label_text_font_size = '11pt'
                p.title.text_font_size = '13pt'
                index = key_list.index(min_keys[0])
                colors = ['blue' if i != index else 'red' for i in range(len(value_list))]
                p.vbar(x= key_list, top= value_list, width=0.5, color=colors)
                label = Label(x=1, y=10, text='Min Value', text_font_size='13pt', text_color='red')
                p.add_layout(label)
                # Show the plot
                try:
                    show(p)
                except Exception as _error:
                    print(f"Error: Unable to generate the graph: {_error}")

    # Test-ideal mapping counter
    def test_ideal_mapping_count(self):
        """
        This function checks for the count of test points that have been mapped multiple times
        """
        mapped_ideal_val= self.db_access_class_instance_2.get_test_table().loc[:,'No. of ideal func'].values
        result_dict = dict(Counter(mapped_ideal_val))
        mapped_ideal_x_val= self.db_access_class_instance_2.get_test_table().loc[:,'x'].values
        x_counter= dict(Counter(mapped_ideal_x_val))
        rep_count= 0
        for rec in x_counter.values():
            if(rec > 1):
                rep_count+= 1
        result_dict['multiple mappings']= rep_count
        print(result_dict)
```

```
# Pie chart that shows the results of all the approaches by comparing test-ideal mapping
def counter_pie_chart(self):
    """
        This function generates pie charts for different methods to showcase the portion of test-ideal mapping
    count in the method.
    The function takes no arguments, and the values are hardcoded in the function itself.
    """
    # Define fields, percentages, and title for each method
    reg_fields= ["y2 (2.099)", "y38 (0.707)", "y43 (0.706)", "y46 (0.706)", "N/A"];
    reg_percentages= [36.6, 8.2, 18.3, 8.2, 28.7]; reg_title= "Regression approach"
    corr_fields= ["y9 (0.699)", "y38 (0.707)", "y45 (14.848)", "y46 (0.706)", "N/A"];
    corr_percentages= [10, 7.5, 60.8, 7.5, 14.2]; corr_title= "Correlation approach"
    ls_fields= ["y9 (0.699)", "y38 (0.707)", "y43 (0.706)", "y46 (0.706)", "N/A"];
    ls_percentages= [12.2, 9.1, 20.4, 9.1, 49.2]; ls_title= "Least-squares approach"

    method_dict= ["regr", "corr", "ls"]
    field_var= [];pct_var= []; title_var="";
    # Loop through each method, set the fields, percentages, and title accordingly, and generate the pie chart
    for rec in method_dict:
        if(rec== "regr"):
            field_var= reg_fields; pct_var= reg_percentages; title_var= reg_title
        elif(rec== "corr"):
            field_var= corr_fields; pct_var= corr_percentages; title_var= corr_title
        else:
            field_var= ls_fields; pct_var= ls_percentages; title_var= ls_title
        # Instantiate the figure object
        graph1 = figure(title = f"{title_var}: Test-ideal mapping count")
        graph1.title.text_font_size = "18pt"
        # Define fields for investment
        fields = field_var
        # Define percentage weightage of the sectors
        percentages = pct_var
        # Convert percentage into radians
        radians1 = [math.radians((percent / 100) * 360) for percent in percentages]
        # Generating the start angle values
        start_angle = [math.radians(0)]  ; prev = start_angle[0]
        for k in radians1[:-1]:
            start_angle.append(k + prev)
            prev = k + prev
        # generating the end angle values
        end_angle = start_angle[1:] + [math.radians(0)]
        # Set the center of the pie chart
        x = 0 ;y = 0
        # Set the radius of the glyphs
        radius = 1
        # now, generate the color of the wedges
        color1 = ["pink", "yellow", "lightgreen","orange", "red"]
        # Plot the graph
        for k in range(len(fields)):
            graph1.wedge(x, y, radius,
                        start_angle = start_angle[k],
                        end_angle = end_angle[k],
                        color = color1[k],
                        legend_label = fields[k])
        graph1.legend.label_text_font_size = '19pt'
        # Display the graph
        try:
            show(graph1)
        except Exception as _error:
            print(f"Error: Unable to generate the graph: {_error}")
```