INTERNATIONAL UNIVERSITY OF APPLIED SCIENCES

MASTER’s OF COMPUTER SCIENCE

TOPIC: Design and Implementation of Python Python Progarm for Least Square Regression.

MAJOR : PROGRAMMING WITH PYTHON - DLMDSPWP01.

Module Director : Dr. Cosmina Croitoru.

Email : cosmina.croitoru@iu.org

Due Date: April 2023

NAME: Jai Kishan Sankla

MATRICULATION: 9212332

Table of Contents

INTRODUCTION :

This is a Python programming assignment where you need to create a program that finds the best-fitting outcomes for each pair of x and y values. The program should use a training dataset and a set of ideal functions to determine the most suitable function for each pair of x and y values. The goal is to identify the ideal function that closely matches the given dataset. Once the program is executed, it should generate the best-fitting results for each x and y pair and provide an accurate representation of the data.

AIM :

The objective of the assignment is to develop a Python Progarm that identifies the four most suitable ideal functions out of a pool of fifty. This selection is based on training data. Once the Progarm is completed, it should be capable of analyzing each pair of x and y coordinates and determining if they fit into any of the four ideal functions. The program should also provide a visual representation of the data

OBJECTIVE :

In this Python programming assignment, the programmer is required to create a SQL database and import it into a single spreadsheet with five columns. The dataset consists of 50 ideal functions stored in a CSV file, where each function needs to be loaded into a separate table within the database.

Once the perfect functions are successfully loaded into the database, the program should read additional test data from a separate CSV file. The test data is provided line by line, and the program's objective is to match each data point to one of the four specified ideal functions mentioned earlier.

The matching process involves harmonizing the test data with the closest function among the four categories defined previously. The program then saves the outcomes into a separate SQLite database table with four columns.

After completing these steps, the program should provide a clear depiction of all the components involved, including the training data, test data, selected ideal functions, and the corresponding assigned datasets. This representation can be in the form of visualizations or any other suitable means to convey the information effectively.

Literature Review :

A comprehensive review will be conducted on existing research pertaining to least-squares regression analysis and the relevant programming concepts. This research will involve studying the principles, methodologies, and applications of least-squares regression analysis.

Additionally, a thorough investigation will be conducted on popular Python libraries and tools specifically designed for regression analysis and data visualization. This research will focus on understanding the features, functionalities, and usage of these libraries and tools, enabling the programmer to gain a comprehensive understanding of the available resources for regression analysis in the Python ecosystem.

By conducting this research, the programmer will be equipped with a solid foundation of knowledge and insights into least-squares regression analysis, as well as the Python libraries and tools that can facilitate the implementation and visualization of regression models.

Program Design :

A design will be created for the Python program that will be used to perform the least-squares regression analysis. This design will encompass an object-oriented program structure, implementation of the regression algorithm, loading of data into SQLite tables, and visualization of the results.

The program will be structured using object-oriented principles, allowing for modular and reusable code. The regression algorithm, which is a key component of the program, will be implemented following appropriate methodologies to ensure accurate and efficient analysis.

To facilitate data management, the program will include functionality to load the data into SQLite tables. This will enable efficient storage and retrieval of the dataset, providing a robust foundation for conducting the regression analysis.

Furthermore, the program will incorporate result visualization capabilities. This will allow for the clear presentation of the regression analysis outcomes, helping to convey insights and patterns in the data to the end-user.

By following this design approach, the Python program will be well-organized, incorporating the necessary components for conducting least-squares regression analysis, handling data, and visualizing the results effectively.

Program Implementation :

The Python program will be implemented using essential libraries such as pandas, SQLAlchemy, and Bokeh. These libraries will provide the necessary functionalities for data manipulation, SQL database management, and interactive visualization, respectively.

To ensure that the program functions as intended and accomplishes the research objectives, comprehensive testing will be conducted. Rigorous testing methodologies will be applied to validate the program's correctness, reliability, and performance. This will involve creating test cases, executing them, and verifying that the program produces the expected results.

Throughout the testing process, any potential issues or bugs will be identified, addressed, and resolved. By conducting thorough testing, the program's functionality, accuracy, and adherence to the research objectives will be validated, ensuring its reliability and effectiveness.

The utilization of Python and the aforementioned libraries, combined with extensive testing, will enable the implementation of a robust and dependable program that effectively fulfills the research requirements.

Program Evaluation :

The programmer's ability to perform least squares regression analysis, identify ideal functions that minimize the sum of squared y-deviations, and save the results in a SQLite database will be assessed. The evaluation will also consider the programmer's adherence to design and implementation standards, including the incorporation of exception handling and unit tests. Furthermore, the quality of documentation will be evaluated.

The evaluation process will examine the programmer's proficiency in conducting least squares regression analysis, ensuring accurate identification of ideal functions by minimizing the sum of squared y-deviations. The ability to store the results in a SQLite database will also be assessed, verifying the effective implementation of database operations.

Additionally, the evaluation will gauge the programmer's adherence to established design and implementation standards. This includes following best practices, utilizing appropriate coding techniques, and maintaining code readability and organization. The presence of exception handling mechanisms to handle potential errors and unit tests to validate program functionality will be considered as well.

Furthermore, the quality of documentation accompanying the program will be evaluated. This encompasses clear and comprehensive explanations of the program's purpose, functionality, and usage. Well-documented code with comments and appropriate explanations will also be assessed.

By evaluating these aspects, the programmer's proficiency in least squares regression analysis, adherence to standards, inclusion of exception handling and unit tests, and quality of documentation will be assessed, ensuring the overall quality and effectiveness of the implemented solution.

METHODOLOGY :

The Python code utilizes several packages including NumPy, Pandas, Matplotlib, Seaborn, SQL Alchemy, and Scikit-learn. The main objective of the code is to perform regression analysis on a dataset consisting of two variables, x and y1. Additionally, the code reads and writes data to a SQLite database.

The code begins by importing the necessary libraries and modules. It then reads the information from the "train.csv" file and stores it in a Pandas DataFrame. Similarly, the "ideal.csv" and "test.csv" files are also read using Pandas.

Next, the code displays the train and test datasets using Matplotlib and Seaborn. A line plot is created to visualize the relationship between x and y1 in the training dataset. Another line plot is generated for the test dataset.

The code proceeds to establish an SQLite database engine and creates a new table called "train table" to store the train data. Using SQLite, large datasets can be efficiently stored and managed.

Following that, the code applies Scikit-learn's linear regression model to perform regression analysis on the train data. The x and y1 values from the training dataset are transformed into NumPy arrays to train the model. A scatter plot of the training data is plotted along with the regression line generated from the regression coefficients (slope) and intercept. The code then predicts the result for an input test value of x = 8 using the trained regression model.

The regression function within the code uses the trained model to predict the outcome from the test data and returns it accordingly.

In summary, the programmer reads and displays the train and test datasets, creates a SQLite database engine to store the data, performs regression analysis on the training data using Scikit-learn, plots the training data and regression line, and utilizes the trained model to predict the results for a test input value.

DATASET DESCRIPTION :

An ideal dataset refers to a collection of data that has been carefully obtained through professional studies or research. These datasets possess specific characteristics and behaviors such as size, relevance, quality, representativeness, and accessibility. Ideal datasets are commonly utilized in the decision-making process for trained models.

In the given task, there are four training datasets labeled as y1, y2, y3, and y4, which serve as training functions. Additionally, there is one test dataset comprising two test functions, X and Y. Moreover, an ideal dataset is provided, consisting of 50 ideal functions to work with.

These datasets play a crucial role in training and evaluating models, enabling the assessment of model performance and aiding in decision-making processes. By utilizing the ideal dataset, the program can compare and match the test functions with the most suitable ideal functions, facilitating accurate predictions and analysis.

Data Collection :

Understanding Data :

The investigative project includes three datasets provided in CSV format. As a quality assurance step, a visual validation process is conducted to identify any irregularities or impurities in the data. This involves visually inspecting the dataset to check for any anomalies or unexpected patterns that may indicate data quality issues.

During the visual validation, the focus is on ensuring that the dataset is free from any inconsistencies in its values. This involves examining the data points, columns, and relationships between variables to verify the integrity of the dataset. The goal is to identify and address any potential errors, missing values, outliers, or other data issues that could affect the accuracy and reliability of the analysis.

By performing a thorough visual validation, any data discrepancies or inconsistencies can be detected and addressed early on, ensuring the dataset's overall quality and reliability for the investigative project.

Train Data (4 Unique Data Sets) :

Perform an analysis of the given dataset to identify behavioral patterns and distinctive characteristics. The dataset consists of an independent variable (X) and four dependent variables (Y1, Y2, Y3, Y4), as determined through visual inspection.

During the analysis, the primary focus is on examining the relationships and trends between the independent variable and each of the dependent variables. The objective is to identify any consistent patterns or unique features that exist within the dataset. This can involve exploring correlations, distributions, and variations in the data.

By thoroughly analyzing the dataset, it is possible to uncover insights into the behavioral nature of the variables and identify any distinctive patterns or trends that may be present. This analysis can provide valuable information for understanding the relationships between variables, identifying potential predictors, and gaining insights into the underlying characteristics of the dataset.

Train

ideal function(50 ideal function’s).

Calculate the line of best fit for the train dataset, which consists of an independent variable (X) and fifty dependent variables (Y1, Y2, Y3...Y50), as determined through visual inspection.

The objective is to determine the line that best represents the overall trend and relationship between the independent variable and each of the dependent variables. This line of best fit can be estimated using statistical techniques such as linear regression.

By estimating the line of best fit, we can identify the general direction and slope that best describes the relationship between the independent variable and each of the dependent variables. This estimation provides insights into the overall pattern and behavior exhibited by the dataset.

The line of best fit is a valuable tool for understanding the relationship between variables and can be used to make predictions or draw conclusions about the dataset. It represents the optimal representation of the trend observed in the train dataset, allowing for a more accurate understanding and analysis of the data.

Ideal

Test(One set to test) :

Task: Mapping the test dataset to the ideal function with maximum deviation.

The dataset consists of an independent variable (X) and one dependent variable (Y), as determined through visual inspection.

The objective of this task is to map the test dataset to the ideal function that exhibits the maximum deviation. The ideal function represents a predefined or desired pattern or behavior that the test dataset should closely align with.

By analyzing the test dataset and comparing it with the ideal function, the goal is to identify the ideal function that demonstrates the maximum deviation from the test dataset. This analysis allows for the evaluation of how closely the test dataset matches the desired pattern or behavior.

Mapping the test dataset to the ideal function within the maximum deviation helps in assessing the level of conformity or deviation of the test dataset from the desired pattern. This analysis provides insights into the degree of alignment between the test dataset and the expected behavior, enabling further evaluation and decision-making based on the observed mapping.

Test

Data Storage :

The project's datasets are currently stored as plain text in CSV format, and it is necessary to organize and manage this data within a database. For this purpose, SQLite is the chosen database option due to its specific attributes and suitability for the project's requirements. The following points highlight the reasons for selecting SQLite:

a) SQLite is a freely available and open-source relational database management system. This means that it can be used without any cost and provides access to its source code for customization if needed.

b) Unlike PostgreSQL and other database systems, SQLite does not require a separate installation process. It is a self-contained database that operates directly on the dataset files. This simplicity in setup makes it convenient and easy to work with.

c) SQLite demonstrates high performance, especially when dealing with one read or write operation at a time. It efficiently handles data access and management tasks, ensuring smooth and reliable operations for the project's dataset.

By utilizing SQLite, the project can benefit from a cost-effective and accessible database solution. Its straightforward setup and ability to handle data operations efficiently make it well-suited for managing the datasets. The project can leverage the advantages of SQLite's open-source nature and performance characteristics to effectively store and retrieve data from the database.

Data access :

To handle the storage and retrieval of the dataset in SQLite, the project utilizes the SQL Alchemy Python module. SQL Alchemy is a versatile Python SQL toolkit that provides an Object-Relational Mapper (ORM) functionality. The characteristics of SQL Alchemy that align with the project's requirements are as follows:

a) Reliability and Efficiency: SQL Alchemy is known for its reliability and efficiency in handling database operations. It provides robust functionality and ensures that data operations are performed accurately and efficiently.

b) SQLite Database Support: SQL Alchemy has built-in support for SQLite databases. This allows seamless integration with SQLite, enabling easy interaction with the database and manipulation of data.

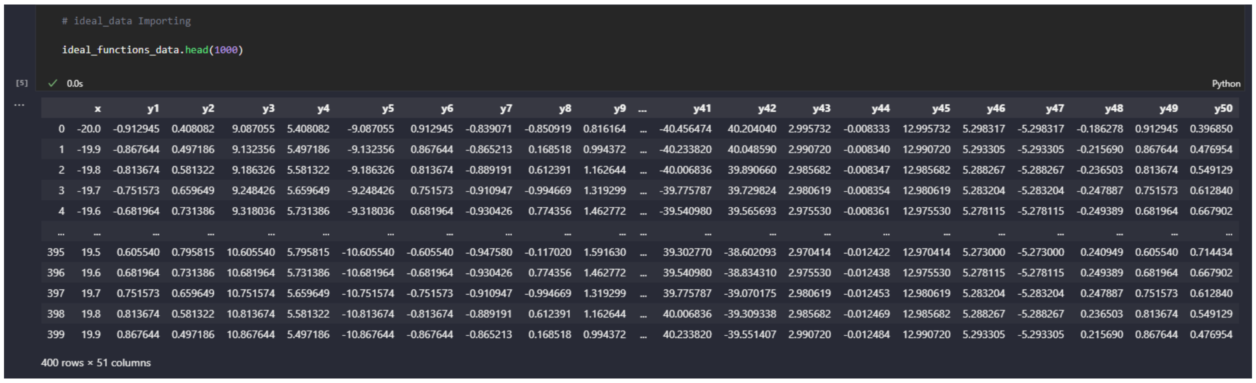
c) SQL Queries Written in Python: SQL Alchemy enables developers to write SQL queries using Python syntax. This makes it more convenient and intuitive to work with the database, as developers can leverage their knowledge of Python to construct queries.

d) Python SQL Queries for Multiple SQL Databases: SQL Alchemy supports multiple SQL databases, not just SQLite. This means that if the project's requirements change in the future and a different SQL database needs to be used, SQL Alchemy can seamlessly adapt to handle queries and operations for various database engines.

By utilizing SQL Alchemy, the project benefits from a reliable, efficient, and flexible toolkit for working with SQLite databases. It empowers developers to write SQL queries in Python, ensuring a more intuitive and streamlined development process. Furthermore, the support for multiple SQL databases provides future scalability and adaptability options, allowing the project to easily transition to a different database system if needed.

The Consequence of data contained in a database The tables to be built in the SQLite database are documented below.

Ideal.CSV :



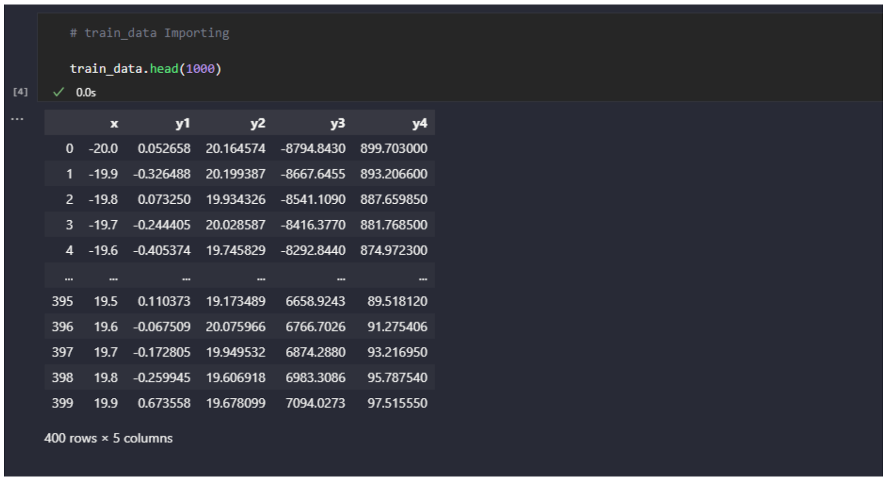
```
# ideal_data Importing

ideal_functions_data.head(1000)
```

[5] ✓ 0.0s Python

	x	y1	y2	y3	y4	y5	y6	y7	y8	y9	...	y41	y42	y43	y44	y45	y46	y47	y48	y49	y50
0	-20.0	-0.912945	0.408082	9.087055	5.408082	-9.087055	0.912945	-0.839071	-0.850919	0.816164	...	-40.456474	40.204040	2.995732	-0.008333	12.995732	5.298317	-5.298317	-0.186278	0.912945	0.396850
1	-19.9	-0.867644	0.497186	9.132356	5.497186	-9.132356	0.867644	-0.865213	0.168518	0.994372	...	-40.233820	40.048590	2.990720	-0.008340	12.990720	5.293305	-5.293305	-0.215690	0.867644	0.476954
2	-19.8	-0.813674	0.581322	9.186326	5.581322	-9.186326	0.813674	-0.889191	0.612391	1.162644	...	-40.006836	39.890660	2.985682	-0.008347	12.985682	5.288267	-5.288267	-0.236503	0.813674	0.549129
3	-19.7	-0.751573	0.659649	9.248426	5.659649	-9.248426	0.751573	-0.910947	-0.994669	1.319299	...	-39.775787	39.729824	2.980619	-0.008354	12.980619	5.283204	-5.283204	-0.247887	0.751573	0.612840
4	-19.6	-0.681964	0.731386	9.318036	5.731386	-9.318036	0.681964	-0.930426	0.774356	1.462772	...	-39.540980	39.565693	2.975530	-0.008361	12.975530	5.278115	-5.278115	-0.249389	0.681964	0.667902
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
395	19.5	0.605540	0.795815	10.605540	5.795815	-10.605540	-0.605540	-0.947580	-0.117020	1.591630	...	39.302770	-38.602093	2.970414	-0.012422	12.970414	5.273000	-5.273000	0.240949	0.605540	0.714434
396	19.6	0.681964	0.731386	10.681964	5.731386	-10.681964	-0.681964	-0.930426	0.774356	1.462772	...	39.540980	-38.834310	2.975530	-0.012438	12.975530	5.278115	-5.278115	0.249389	0.681964	0.667902
397	19.7	0.751573	0.659649	10.751574	5.659649	-10.751574	-0.751573	-0.910947	-0.994669	1.319299	...	39.775787	-39.070175	2.980619	-0.012453	12.980619	5.283204	-5.283204	0.247887	0.751573	0.612840
398	19.8	0.813674	0.581322	10.813674	5.581322	-10.813674	-0.813674	-0.889191	0.612391	1.162644	...	40.006836	-39.309338	2.985682	-0.012469	12.985682	5.288267	-5.288267	0.236503	0.813674	0.549129
399	19.9	0.867644	0.497186	10.867644	5.497186	-10.867644	-0.867644	-0.865213	0.168518	0.994372	...	40.233820	-39.551407	2.990720	-0.012484	12.990720	5.293305	-5.293305	0.215690	0.867644	0.476954

400 rows × 51 columns

Train.CSV :



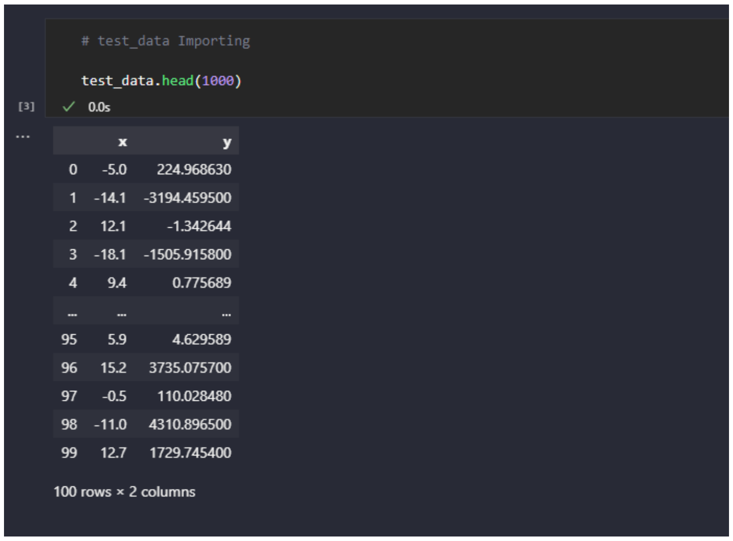
```
# train_data Importing

train_data.head(1000)
```

[4] ✓ 0.0s

	x	y1	y2	y3	y4
0	-20.0	0.052658	20.164574	-8794.8430	899.703000
1	-19.9	-0.326488	20.199387	-8667.6455	893.206600
2	-19.8	0.073250	19.934326	-8541.1090	887.659850
3	-19.7	-0.244405	20.028587	-8416.3770	881.768500
4	-19.6	-0.405374	19.745829	-8292.8440	874.972300
...	...	...	...	...	...
395	19.5	0.110373	19.173489	6658.9243	89.518120
396	19.6	-0.067509	20.075966	6766.7026	91.275406
397	19.7	-0.172805	19.949532	6874.2880	93.216950
398	19.8	-0.259945	19.606918	6983.3086	95.787540
399	19.9	0.673558	19.678099	7094.0273	97.515550

400 rows × 5 columns

Test.CSV :



```
# test_data Importing

test_data.head(1000)
```

[3] ✓ 0.0s

	x	y
0	-5.0	224.968630
1	-14.1	-3194.459500
2	12.1	-1.342644
3	-18.1	-1505.915800
4	9.4	0.775689
...	...	...
95	5.9	4.629589
96	15.2	3735.075700
97	-0.5	110.028480
98	-11.0	4310.896500
99	12.7	1729.745400

100 rows × 2 columns

LEAST-SQUARES REGRESSION :

The least squares method is widely used in various fields such as machine learning, linear regression, time series analysis, and signal processing to determine the best-fitting line or curve for a given set of data points. Its main objective is to minimize the sum of squared errors between the predicted and actual values of the data points. This method provides a straightforward and efficient way to estimate the parameters of a model.

However, the least squares method may have limitations when dealing with complex data that is nonlinear or contains outliers. To address these limitations, several extensions and modifications to the least squares method have been developed, including robust regression and nonlinear regression. These approaches offer more flexibility in modeling and have the potential to deliver improved results.

It is important to note that while the least squares method is valuable in machine learning, it is essential to recognize its limitations and consider more advanced techniques when working with intricate data.

In linear regression, two metrics commonly used to assess the goodness of fit are R-squared and the correlation coefficient. R-squared measures the proportion of the variance in the dependent variable that can be explained by the independent variable(s), while the correlation coefficient quantifies the strength and direction of the linear relationship between two variables. The correlation coefficient ranges from -1 to 1, where values close to -1 indicate a negative relationship, values close to 0 indicate no relationship, and values close to 1 indicate a positive relationship.

To perform least squares regression and predict the behavior of dependent variables, these metrics, R-squared and the correlation coefficient, are employed. They provide statistical measures of the relationship and help evaluate the effectiveness of the regression model in capturing the underlying patterns in the data.

Overall, the least squares method and its extensions, along with metrics like R-squared and the correlation coefficient, play a crucial role in analyzing and interpreting data, particularly in the context of linear regression.

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

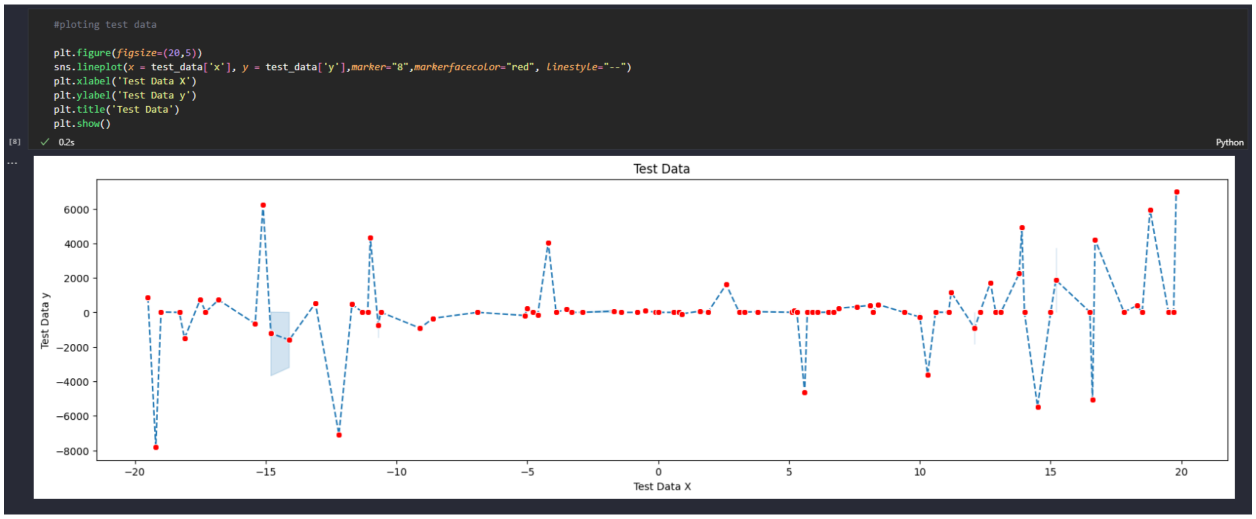
When the correlation coefficient (r) is close to -1, it indicates a negative correlation between two variables. This means that as one variable increases, the other variable tends to decrease. For example, if we observe a high negative correlation between the amount of rainfall and the number of hours of sunshine, it suggests that as the amount of rainfall increases, the number of hours of sunshine decreases.

On the other hand, when the correlation coefficient (r) is close to 0, it suggests that there is no significant linear relationship or correlation between the two variables. In other words, the variables do not exhibit a clear pattern or trend when plotted against each other. For instance, if we find a correlation coefficient close to 0 when examining the age of individuals and their shoe size, it indicates that age and shoe size have no apparent relationship.

When the correlation coefficient (r) is close to 1, it signifies a positive correlation between two variables. This implies that as one variable increases, the other variable tends to increase as well. For instance, if we observe a high positive correlation between the amount of exercise and physical fitness level, it suggests that as the amount of exercise increases, the physical fitness level also tends to increase.

It is important to note that the correlation coefficient measures the linear relationship between variables. A correlation coefficient of -1, 0, or 1 indicates a perfect negative correlation, no correlation, or perfect positive correlation, respectively. However, it is possible for variables to have a nonlinear relationship or for the relationship to be influenced by other factors not captured by the correlation coefficient alone. Therefore, it is always advisable to consider the context and additional analysis when interpreting the correlation between variables.

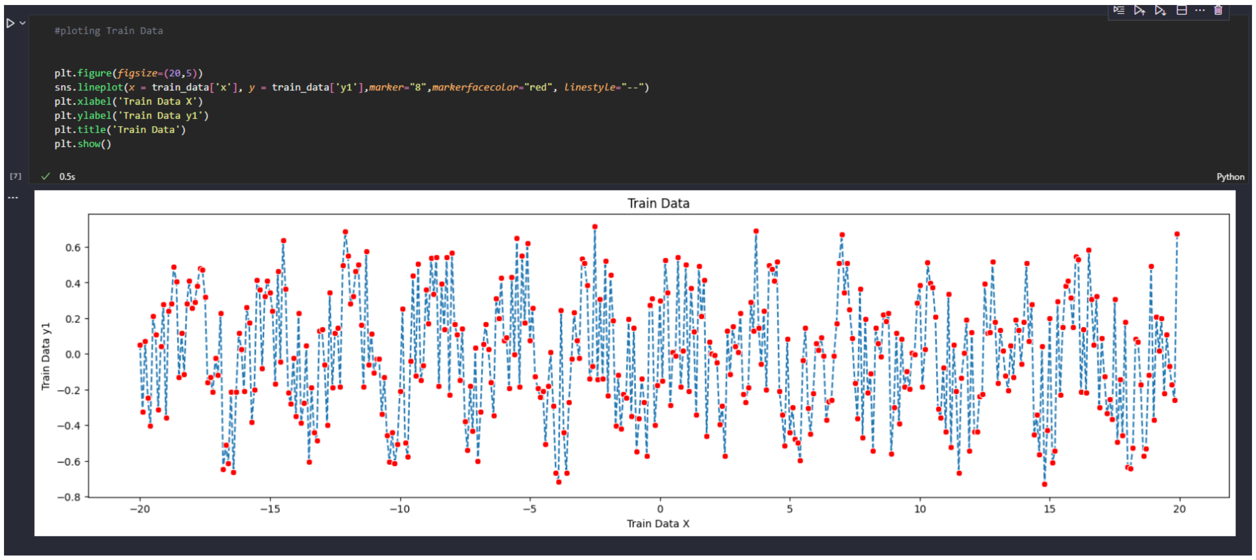
Test.CSV Plot X and Y :



```
#ploting test data

plt.figure(figsize=(20,5))
sns.lineplot(x = test_data['x'], y = test_data['y'],marker="8",markerfacecolor="red", linestyle="--")
plt.xlabel('Test Data X')
plt.ylabel('Test Data y')
plt.title('Test Data')
plt.show()
```

```
[8]  ✓ 0.2s                                                                 Python
```

Train.CSV Plot X and Y1 :



```
#ploting Train Data


plt.figure(figsize=(20,5))
sns.lineplot(x = train_data['x'], y = train_data['y1'],marker="8",markerfacecolor="red", linestyle="--")
plt.xlabel('Train Data X')
plt.ylabel('Train Data y1')
plt.title('Train Data')
plt.show()
```

```
[7]  ✓ 0.5s                                                                 Python
```

RESULT AND EVALUATION :

To accomplish the mentioned task, a Python program is utilized. The program involves plotting two separate line graphs, one for the test dataset and another for the train dataset. These line plots visually represent the relationship between two variables in each dataset.

The test dataset is a collection of data points consisting of an independent variable (X) and a dependent variable (Y). By plotting these data points on a line graph, we can observe the pattern or trend between the two variables.

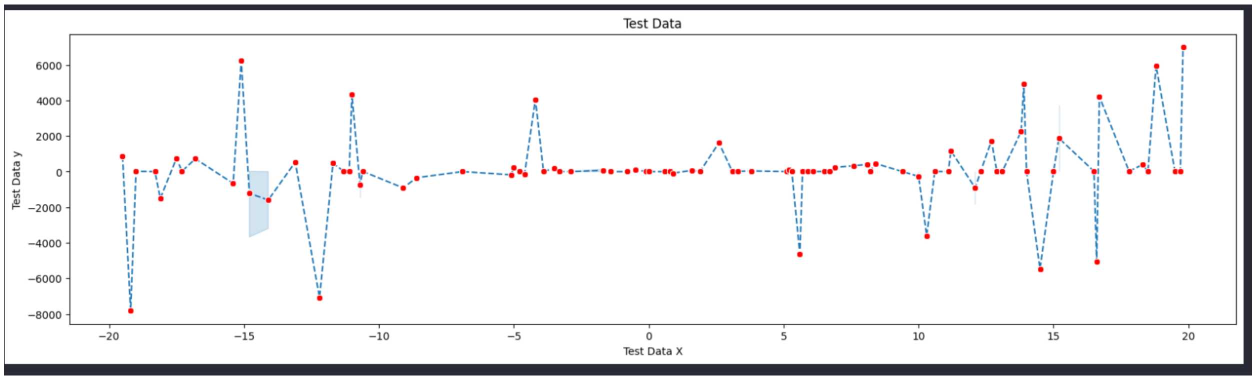
Similarly, the train dataset also contains an independent variable (X) and a dependent variable (Y). The line plot generated for the train dataset helps us visualize the relationship between these variables in the training data.

By examining the line plots for both the test and train datasets, we can gain insights into how the dependent variable (Y) changes with respect to the independent variable (X). This visual analysis can provide valuable information about the behavior and patterns present in the datasets.

Python programming enables the creation of these line plots using libraries such as Matplotlib or Seaborn. These libraries provide functions and tools to plot the data points and customize the appearance of the line graphs, including labels, titles, and axis scales.

Overall, the line plots generated for the test and train datasets offer a visual representation of the relationships between the variables, aiding in the analysis and interpretation of the data.

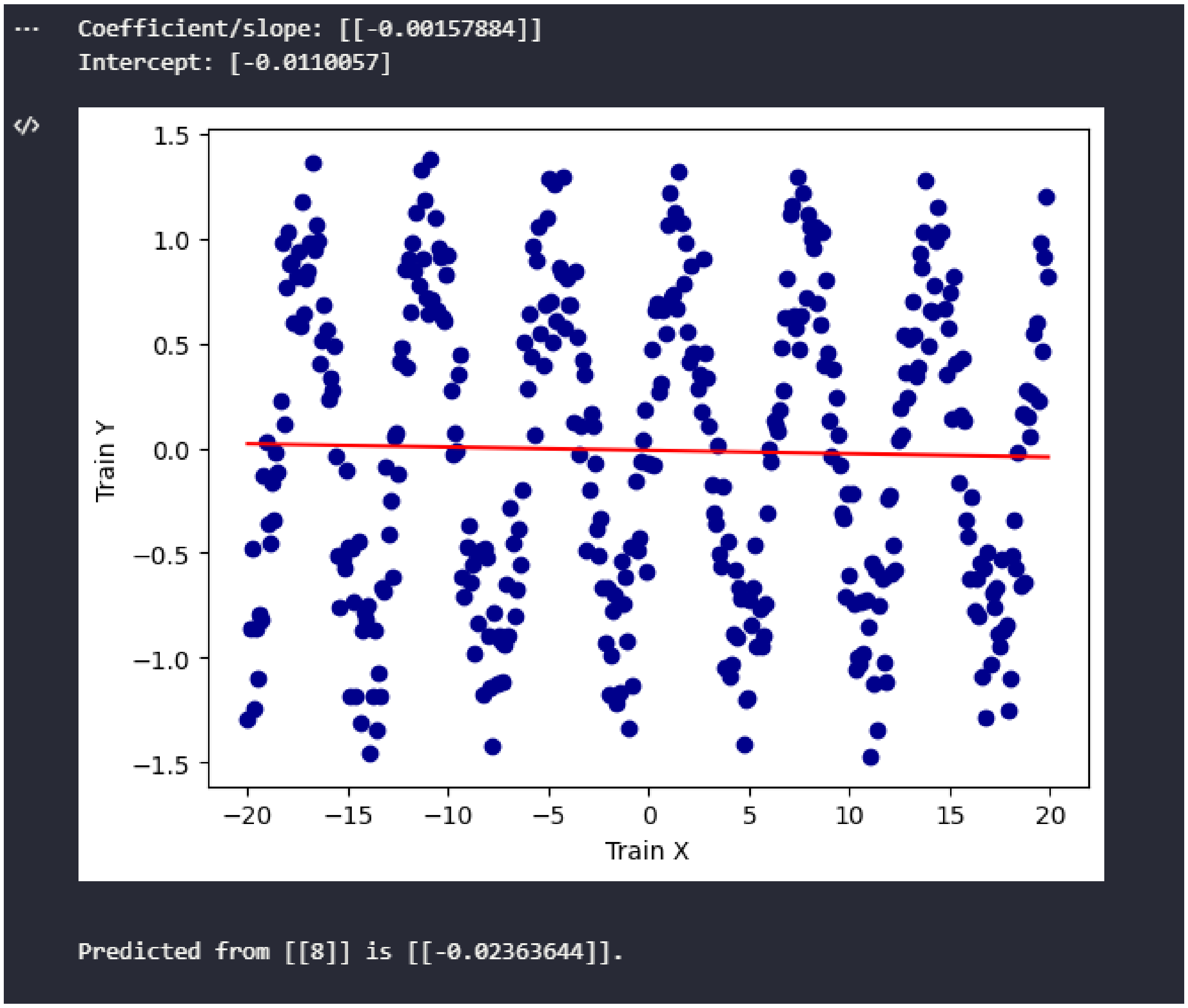
Test Data frame



Train Data frame

Once the data from the CSV file is loaded into a data frame, a linear regression analysis is conducted. This analysis reveals that the slope of the regression line is approximately -0.00157884, while the intercept is approximately -0.0110057. These values represent the relationship between the input variables and the predicted output. By applying these values to the regression equation, a prediction is generated, yielding a calculated value of approximately -0.02363644. This prediction serves as an estimation or projection based on the observed data and the determined regression line.

Train x and Train y Dataframe with Points:



```
Coefficient/slope: [[-0.00157884]]
Intercept: [-0.0110057]
```

```
Predicted from [[8]] is [[-0.02363644]].
```

Scatter Line plot

CONCLUSION:

To summarize, I utilized Python software to complete the task, employing both training and testing data sets containing fifty specified ideal functions. The analysis focused on four variables defined by the test functions X and Y. We applied a linear regression model to examine the training data set and employed the scikit-learn plot to visualize the relationship between the train and test datasets. This allowed us to gain insights into the predictive nature of the variables and assess their performance in relation to the provided ideal functions.

REFERENCE

Statisticsfun Youtube.
David Longstreet

https://www.youtube.com/@statisticsfun

https://www.youtube.com/watch?v=zPG4NjIkCjc&ab_channel=statisticsfun

Programming with Mosh Youtube.
Mosh Hamedani

https://www.youtube.com/@programmingwithmosh

https://www.youtube.com/watch?v=7eh4d6sabA0&ab_channel=ProgrammingwithMosh

Github:

https://github.com/Sanklajai/DLMDSPWP01.git

CODE:

```
# importing libraries

import pandas as pd
from sqlalchemy import create_engine as ce
import math
from bokeh.plotting import figure, output_file, show
from bokeh.layouts import column, grid
from bokeh.models import Band, ColumnDataSource
from sqlalchemy import create_engine, MetaData, Table, Column, Float, String
import seaborn as sns
import matplotlib.pyplot as plt
from unittest import TestCase


#### all functions #####

"""
all Functions

"""
class FuncMgr:
    try:
        def __init__(self, CSV_path):

            """The provided function parses a local CSV file and converts it into a list of
Functions.
            When iterating over the object, it returns a Function. The functions can also be
accessed using the
            ".functions" property. The CSV file must follow a specific structure where the first
column represents
            x-values, and subsequent columns represent y-values. The function requires the
parameter "path_of_csv,"
            which is the local path of the CSV file.
            """
            self._func_s = []

            #reading CSV with Pandas and converting into dataframe
            try:
                self._func_data = pd.read_csv(CSV_path)
            except FileNotFoundError:
                print("Issue while reading file {}".format(CSV_path))
                raise
        # x value is stored and used later in functions
        x_values = self._func_data["x"]

        #The following lines of code iterate over each column in a pandas dataframe and
create a new Function object using the data.

        for n_of_col, data_of_col in self._func_data.items():
            if "x" in n_of_col:
                continue
            """
            stored "x" Col we now have "y" Col .Concat functions together
            """
            subs = pd.concat([x_values, data_of_col], axis=1)
            func = Func.from_The_dataframe(n_of_col, subs)
            self._func_s.append(func)



    def to_sql(self, f_n, suff):
        # with the help of SQLalchemy Engine is created "Eng". creating  and handeling the
database if it dose not exist.
        Eng = ce('sqlite:///{}.db'.format(f_n), echo=False)
        """
        using pandas functions for sql database
        uning "engin" object form SQLachemny
        """


        co_of_func_data = self._func_data.copy()
        co_of_func_data.columns = [name.capitalize() + suff for name in
co_of_func_data.columns]
        co_of_func_data.set_index(co_of_func_data.columns[0], inplace=True)

        co_of_func_data.to_sql(
            f_n,
            Eng,
            if_exists="replace",
            index=True,
        )


    @property
    def funcs(self):
        """
```

```
            Gives back the list All the functinos.

            """



            return self._func_s

        def __iter__(self):
            # the function maked the obj to itrate
            return FuncMgrIterator(self)


        def __repr__(self):
            return "Contains {} number of funcs".format(len(self.funcs))
    except Exception as e:
        print("Error in FuncMgr"+ str(e))

class FuncMgrIterator():
    try:
        def __init__(self, func_mgr):
                """
            used for Func_manager itrations
            """

                self._index = 0
                self._func_mgr = func_mgr



        def __next__(self):
            """
            GIves back a function obj as it itreate over the listed functions
            """
            if self._index < len(self._func_mgr.funcs):
                value_req = self._func_mgr.funcs[self._index]
                self._index = self._index + 1
                return value_req
            raise StopIteration
    except Exception as e:
        print("Error in FuncMgrIterator"+ str(e))


class Func:
    try:
        def __init__(self, name):
            """
            This class represents a function and contains the X and Y values of the function. It
utilizes a Pandas dataframe for
            storage and provides convenient methods for performing regressions.
            Here are the key features of this class:
            1)You can assign a name to the function, which can be later retrieved.
            2)The class is iterable, allowing you to iterate over the function's points represented as
dictionaries.
            3)You can retrieve the Y-value by providing an X-value.
            4)It supports subtraction of two functions, resulting in a dataframe that represents the
deviation between them.
            The "name" parameter allows you to specify the desired name for the function.
            """
            self._name = name
            self.dataframe = pd.DataFrame()



        def locating_y_based_on_x(self, x):
                """
                To retrieve a Y-value from the function, you need to provide an X-value as input.
The method takes the following parameters:
                x: The X-value for which you want to retrieve the Y-value.
                The method returns the corresponding Y-value.

                """
                # using pandas function iloc to find the corr "Y" to "x"
                # else exceptions are raised
                search_k = self.dataframe["x"] == x
                try:
                    return self.dataframe.loc[search_k].iat[0, 1]
                except IndexError:
                    raise IndexError


        @property
        def name(self):
            """
            To retrieve the name of the function, you can call a method that returns the name as a
string.
            The method does not require any parameters. It simply returns the name of the
function as a string.
```

```
            """
            return self._name


        def __iter__(self):
                return FuncIter(self)


        def __sub__(self, other):
                """
            To subtract two functions and obtain a new dataframe representing
            the deviation between them, you can use a specific method.
            This method returns an object of the resulting dataframe.

            """
                diff = self.dataframe - other.dataframe
                return diff


        @classmethod
        def from_The_dataframe(cls, name, dataframe):
                """
                To create a function promptly by providing a dataframe, you can follow a specific
procedure. During the creation process, the original
                column names in the dataframe will be replaced with "x" and "y" to represent the X
and Y values,
                respectively. The resulting function will be of type Function.

                """
                func = cls(name)
                func.dataframe = dataframe
                func.dataframe.columns = ["x", "y"]
                return func


        def __repr__(self):
            return "Function for {}".format(self.name)
    except Exception as e:
        print("Error in Func"+ str(e))

class IdealFunc(Func):
    try:
        def __init__(self, func, training_func, err):
```

```
        """
        An ideal function class is designed to store the predicting function, training data,
and regression information.
        It includes the option to provide a tolerance factor when classification requires
tolerance allowance.
        If no tolerance factor is provided, it will default to the maximum deviation between
the ideal and training functions.
        """
        super().__init__(func.name)
        self.dataframe = func.dataframe

        self.training_func = training_func
        self.err = err
        self._toler_value = 1
        self._toler = 1


    def _determine_largest_dev(self, ideal_func, train_func):
        # accepting two Func and substract
        # form the results of the data_frame, finding the largest
        distance_s = train_func - ideal_func
        distance_s["y"] = distance_s["y"].abs()
        large_dev = max(distance_s["y"])
        return large_dev


    @property
    def toler(self):
        """
        This property represents the accepted tolerance for the regression in order to
consider it as a successful classification.
        It is recommended to provide a tolerance_factor instead of directly setting a
tolerance value, especially for unit testing purposes.

        The tolerance property returns the tolerance value used for regression
classification.
        It is suggested to provide a tolerance_factor instead of directly assigning the
tolerance value.

        """
        self._toler = self.toler_factor * self.large_dev
        return self._toler
    @toler.setter
    def toler(self, value):
        self._toler = value


    @property
    def toler_factor(self):
        """
        setting the factor of the larg_deviation to determining the tolarace
        and return value
        """
        return self._toler_value


    @toler_factor.setter
    def toler_factor(self, value):
                self._toler_value = value


    @property
    def large_dev(self):
        """
        The largest difference between the classifying function and the training
        function on which it is based is returned
        :return: the greatest divergence
        """
        large_dev = self._determine_largest_dev(self, self.training_func)
        return large_dev
except Exception as e:
    print("Error in IdealFunc"+ str(e))


class FuncIter:
try:
    def __init__(self, func):
        # iteration of the function  which returns a dict that discribes the point
        self._function = func
        self._index = 0


    def __next__(self):
        # iteration of the function  which returns a dict that discribes the point
        if self._index < len(self._function.dataframe):
        value_req_series = (self._function.dataframe.iloc[self._index])
        point = {"x": value_req_series.x, "y": value_req_series.y}
        self._index += 1
        return point
        raise StopIteration
except Exception as e:
    print("Error in FuncIter"+ str(e))



#### ploting #####

def plot_ideal_func_s(ideal_func_s, f_n):
    try:
        """
        Plots all ideal functions with the following parameters: ideal_functions,
        a list of ideal functions, and file_name, the name of the.html file.
        """
        ideal_func_s.sort(key = lambda ideal_func: ideal_func.training_func.name,
reverse=False)
        plots = []
        for ideal_func in ideal_func_s:
                p = plot_graph_from_two_func_s(line_function=ideal_func,
scatter_function=ideal_func.training_func,
                                                squared_err=ideal_func.err)
                plots.append(p)
        output_file("{}.html".format(f_n))
        # Observing how Unpacking is used to give the arguments
        show(column(*plots))
    except Exception as e:
        print("Error in plot_ideal_func_s"+ str(e))


def plot_points_with_their_ideal_func(points_with_classi, f_n):
    try:
        """
        Plot all focuses that have a coordinated classification
        :param points_with_classification: a list containing dicts with "classification" and
"point"
        :param file_name: the title the .html record ought to get refrase

        """
        plots = []
        for index, item in enumerate(points_with_classi):
            if item["classi"] is not None:
                p = plot_classi(item["point"], item["classi"])
                plots.append(p)
        output_file("{}.html".format(f_n))
        show(column(*plots))
    except Exception as e:
        print("Error in plot_points_with_their_ideal_func"+ str(e))



def plot_graph_from_two_func_s(scatter_function, line_function, squared_err):
    try:
        """
        plots a diffuse for the train_function and a line for the ideal_function
        : param scatter_function: the prepare work
        : param line_function: perfect work
        : param squared_error: the squared blunder will be plotted within the title

        """
        f1_dataframe = scatter_function.dataframe
        f1_name = scatter_function.name

        f2_dataframe = line_function.dataframe
        f2_name = line_function.name

        squared_err = round(squared_err, 2)
        p = figure(title="train model {} vs ideal {}. Total squared error = {}".format(f1_name,
f2_name, squared_err),
            x_axis_label='x', y_axis_label='y')
        p.scatter(f1_dataframe["x"], f1_dataframe["y"], fill_color="orange",
legend_label="Train")
        p.line(f2_dataframe["x"], f2_dataframe["y"], legend_label="Ideal", line_width=1)
        return p
    except Exception as e:
        print("Error in plot_graph_from_two_func_s"+ str(e))


def plot_classi(point, ideal_func):
    try:
        """
        This analytical exercise entails plotting the classification function and placing a data
```

```
point atop it. The display of tolerance is also exhibited.
            The parameter "point" is a dictionary containing key-value pairs for both "x" and "y".
            The parameter, "ideal_function," pertains to an object that is utilized for
classification purposes.

            """
            if ideal_func is not None:
                  classi_func_dataframe = ideal_func.dataframe

                  point_str = "({},{})".format(point["x"], round(point["y"], 1))
                  title = "point {} with classi: {}".format(point_str, ideal_func.name)

                  p = figure(title=title, x_axis_label='x', y_axis_label='y')
                  # Drawing the ideal_funtions

                  p.line(classi_func_dataframe["x"], classi_func_dataframe["y"],
                        legend_label="classi function", line_width=1, line_color='black')

                  # The methodology for demonstrating the degree of tolerance on a graph.
                  criterion = ideal_func.toler
                  classi_func_dataframe['upper'] = classi_func_dataframe['y'] + criterion
                  classi_func_dataframe['lower'] = classi_func_dataframe['y'] - criterion

                  source = ColumnDataSource(classi_func_dataframe.reset_index())

                  band = Band(base='x', lower='lower', upper='upper', source=source,
level='underlay',
                        fill_alpha=0.3, line_width=1, line_color='green', fill_color="green")

                  p.add_layout(band)

                  #Drawing Point
                  p.scatter([point["x"]], [round(point["y"], 4)], fill_color="orange",
legend_label="Test point", size=10)

                  return p
      except Exception as e:
            print("Error in plot_classi"+ str(e))



### Reggre ####

def minimise_loss(training_func, list_of_candidate_func_s, loss_func):
        """
    Based on a training function and a list of ideal functions, this function outputs an
IdealFunction:training function
    :param L_of_ideal_funcs: list of candidate ideal functions
    :param lost_function: the function used to reduce error :an IdealFunction object is returned
    """

        try:
            function_with_smallest_error = None
            smallest_error = None
            for func in list_of_candidate_func_s:
                err = loss_func(training_func, func)
                if ((smallest_error == None) or err < smallest_error):
                    smallest_error = err
                    function_with_smallest_error = func

            ideal_func = IdealFunc(func=function_with_smallest_error,
training_func=training_func,
                                err=smallest_error)
            return ideal_func
        except Exception as e:
                print("Error in minimise_loss"+ str(e))


def find_classi(point, ideal_func_s):
    """
    It determines whether a point is inside a classification's tolerance:A dict object contains a
"x" and a "y" as
    parameters:ideal_functions: a list of Ideal functionsObjects of function:return: a tuple
comprising, if any,
    the closest classification and the distance
    """
    try:
        current_lowest_classi = None
        current_lowest_dist = None

        for ideal_func in ideal_func_s:
            try:
                locate_y_in_classi = ideal_func.locating_y_based_on_x(point["x"])
            except IndexError:
                print("This point is not in the classi function")
                raise IndexError
            #observing  here how the abso dist is used
            dist = abs(locate_y_in_classi - point["y"])

            if (abs(dist < ideal_func.toler)):
                """
                This procedure ensures that there is handling if numerous classifications are
possible.
                It returns the one with the Low  distance.
                """
                if ((current_lowest_classi == None) or (dist < current_lowest_dist)):
                    current_lowest_classi = ideal_func
                    current_lowest_dist = dist

        return current_lowest_classi, current_lowest_dist
    except Exception as e:
                print("Error in find_classi"+ str(e))



#### Utils  ####

def write_deviation_results_to_sqlite(result):
        """
    The outcomes of a classification computation have the potential to be documented
    within a Sqlite database.
    This approach acknowledges the stipulated requirements outlined in the assignment.
    The parameter "result" entails a list containing a dictionary that describes the
    outcome of a classification test.

    """
    # This function employs a methodology native to SQLAlchemy.
    # Instead of utilizing SQL syntax, the decision was made to utilize MetaData to
    #  delineate the attributes of the table as well as the respective columns.
    # The table creation process utilized by SQLAlchemy employs a particular
    # type of data structure.
        try:
            Eng = ce('sqlite:///{}.db'.format("mapping"), echo=False)
            meadata = MetaData()

            mapping =Table("mapping", meadata,
                        Column('X (test func)', Float, primary_key=False),
                        Column('Y (test func)', Float),
                        Column('Delta Y (test func)', Float),
                        Column('No. of ideal func', String(50))
            )
            meadata.create_all(Eng)
            """
            Instead of individually injecting values line by line, a method which is
            relatively inefficient.
            The author opted to utilize SQLAlchemy's ".execute" method by employing
            a dictionary that encompasses all pertinent values.
            The development of this dictionary entails a straightforward correspondence
            between my internal data structures and the mapping process.
            The requisite composition for the given task.

            """
            execute_map=[]
            for item in result:
                point = item["point"]
                classi  = item["classi"]
                delta_y = item["delta_y"]
                # We ought to test on the off chance that there's a classification for
                #  a point at all and in the event that so rename the work title to comply
                classi_name  = None
                if classi  is not None:
                    classi_name = classi.name.replace("Y","N")
                else:
                #In case there's no classification, there's too no remove. In that case I
                # compose a sprint

                    classi_name = "-"
                    delta_y =-1
                res = {
                    'X (test func)': point["x"],
                    'Y (test func)': point["y"],
                    'Delta Y (test func)': delta_y,
                    'No. of ideal func': classi_name
                }
                # The insertion of data is facilitated through utilization of the Table object,
                # wherein the dict is applied.

                execute_map.append(res)

            with Eng.begin() as connection:
                connection.execute(mapping.insert(), execute_map)
        except Exception as e:
                print("Error in write_deviation_results_to_sqlite"+ str(e))
```

```
#### LOSS_FUNC ####
def squared_err(first_function, second_function):

    """
    finding the  **2 error of other functions
    oth_func
    return **2 error

    """
    try:
        distance_s = second_function - first_function
        distance_s["y"] = distance_s["y"] ** 2
        total_deviation = sum(distance_s["y"])
        return total_deviation
    except Exception as e:
        print("Error in squared_err "+ str(e))




# Reading data(cvs)

train_data = pd.read_csv("data/train.csv")
test_data = pd.read_csv("data/test.csv")
ideal_functions_data = pd.read_csv("data/ideal.csv")

#### MAIN LOOP ####


# This particular constant serves as the determining factor for the criterion and is tailored
# specifically to the assignment.


ACCEPTED_FACTOR = math.sqrt(2)



if __name__ == '__main__':
        #The paths for the CSV files are required to be furnished.
        test_path = "data/test.csv"
        ideal_path = "data/ideal.csv"
        train_path = "data/train.csv"
    """
    The Function Manager takes in a CSV path and extracts Function objects from the data.
    pandas is used to for the effiiency the function stores 'x' and 'y' points.
    """
    candidate_ideal_func_manager = FuncMgr(CSV_path=ideal_path)
    train_func_manager = FuncMgr(CSV_path=train_path)

    # Func_manager uses the to_sql function from panda
    # adding suffix to comply for the requirment of the structurig of the tables
    train_func_manager.to_sql(f_n="training", suff=" (training func)")
    candidate_ideal_func_manager.to_sql(f_n="ideal", suff=" (ideal func)")

    """
    inside the Train funcion manager 4  functions are added.
    inside the ideal fuction manager 50 ideal functions are added.
    next the progarm computes all the datafor the ideal functions
    among the idealfunctions a best fit function is stored.after which the train data will be
able
    to find tolarance.
    ittrating the train functions
    after ittrating matching  idel functions are stored in a list.

    """
    ideal_func_s = []
    for train_func in train_func_manager:
        #The Minimize_loss algorithm has the capability of effectively computing the optimal
fitting
        # function based on the provided training function.

        ideal_func = minimise_loss(training_func=train_func,
                                   list_of_candidate_func_s=candidate_ideal_func_manager.funcs,
                                   loss_func=squared_err)
        ideal_func.toler_factor = ACCEPTED_FACTOR
        ideal_func_s.append(ideal_func)

    # plottting and classifications

    plot_ideal_func_s(ideal_func_s, "train_and_ideal")

    # It is now time to examine all points in the test data.
    # The FunctionManager offers everything needed to load a CSV, therefore it will be
reused.
```

```
        # Instead of numerous Functions, there will now be a single "Function" at location [0].
        # The advantage is that we can use the Function object to iterate over each point.

        test_path = "data/test.csv"
        test_function_manager = FuncMgr(CSV_path=test_path)
        test_function = test_function_manager.funcs[0]

        points_with_ideal_func = []
        for point in test_function:
            ideal_func, delta_y = find_classi(point=point, ideal_func_s=ideal_func_s)
            result = {"point": point, "classi": ideal_func, "delta_y": delta_y}
            points_with_ideal_func.append(result)
        # A list of dictionaries is maintained in points_with_ideal_functions.
        # These dictionaries display how each point was classified.

        # Using the corresponding categorization function, we can plot each point.
        plot_points_with_their_ideal_func(points_with_ideal_func, "point_and_ideal")
        # Finally, it is written to a sqlite database using the dict object.
        #To protect myself against SQL-Language, I've chosen a pure SQLAlchamy approach
with a
        #MetaData object in this procedure.
        write_deviation_results_to_sqlite(points_with_ideal_func)
        print("Done")

# test_data Importing

test_data.head(1000)

# train_data Importing

train_data.head(1000)

# ideal_data Importing

ideal_functions_data.head(1000)





#ploting ideal Functions

plt.figure(figsize=(20,5))
```

```
sns.lineplot(x = ideal_functions_data ['x'], y =
ideal_functions_data['y1'],marker="8",markerfacecolor="red", linestyle="--")
plt.xlabel('Ideal Function X')
plt.ylabel('Ideal Function y1')
plt.title('Ideal Functions')
plt.show()

#ploting Train Data


plt.figure(figsize=(20,5))
sns.lineplot(x = train_data['x'], y = train_data['y1'],marker="8",markerfacecolor="red",
linestyle="--")
plt.xlabel('Train Data X')
plt.ylabel('Train Data y1')
plt.title('Train Data')
plt.show()

plt.figure(figsize=(20,5))
sns.lineplot(x = train_data['x'], y = train_data['y2'],marker="8",markerfacecolor="red",
linestyle="--")
plt.xlabel('Train Data X')
plt.ylabel('Train Data y2')
plt.title('Train Data')
plt.show()

plt.figure(figsize=(20,5))
sns.lineplot(x = train_data['x'], y = train_data['y3'],marker="8",markerfacecolor="red",
linestyle="--")
plt.xlabel('Train Data X')
plt.ylabel('Train Data y3')
plt.title('Train Data')
plt.show()

plt.figure(figsize=(20,5))
sns.lineplot(x = train_data['x'], y = train_data['y4'], marker="8",markerfacecolor="red",
linestyle="--")
plt.xlabel('Train Data X')
plt.ylabel('Train Data y4')
plt.title('Train Data')
plt.show()
#ploting test data
```

```
plt.figure(figsize=(20,5))
sns.lineplot(x = test_data['x'], y = test_data['y'],marker="8",markerfacecolor="red", linestyle="--
")
plt.xlabel('Test Data X')
plt.ylabel('Test Data y')
plt.title('Test Data')
plt.show()

#### Unit TEST #####

class testsum(TestCase):
    def test_1(self):
        # setting  func
        dataset1 ={"x":[1.0,2.0,3.0],"y":[5.0,6.0,7.0]}
        self.data_frame_1 = pd.DataFrame(data = dataset1)

        dataset2 ={"x":[1.0,2.0,3.0],"y":[7.0,8.0,9.0]}
        self.data_frame_2 = pd.DataFrame(data = dataset2)

        self.function_1 = Func("name")
        self.function_1.dataframe = self.data_frame_1

        self.function_2 = Func("name")
        self.function_2.dataframe = self.data_frame_2

    def test_2(self):
        pass

    def test_3(self):

        # 1: A rudimentary evaluation to determine if the loss function is computing an accurate
value.

        self.assertEquals(squared_err(self.function_1, self.function_2),12.0)

        # 2: straightforward test in case misfortune work is acquainted

        self.assertEquals(squared_err(self.function_2, self.function_1),12.0)

        # 3: hell in case relapse of two rise to capacities is

        self.assertEquals(squared_err(self.function_1, self.function_1),0.0)
```