Title Page

Tasks for Course: DLMDSPWP01

Programming with Python

Matruclation No:10242963

Table of Contents

List of Figures

1. Introduction

1.1. Project Overview

This project addresses the basic challenge of analysis and matching of mathematical functions across multiple datasets. The system will provision to process three types of input data sources, each for a different purpose in the pipeline of the analysis.
The training data consists of all the information included in train.csv; it provides a basis for operating the system, including x-coordinate data with four corresponding y-values to represent baseline functions-for reference, this is what the system will be tested against. These training functions will set up the basis of learning about what the relationships between inputs and outputs are.
The ideal functions dataset, ideal.csv, is quite large, with fifty different functions that have x-coordinates and y-values. The functions are supposed to be matched with the test data by the system; hence, they will provide a number of comparison cases for the matching algorithm in the system.
The test dataset consists of individual data points provided in the test.csv file that need to be matched against ideal functions. Each point consists of an x-coordinate and a single y-value; hence, the core challenge for the system's matching capabilities.

1.2. Project Objectives

This system was developed with various interrelated objectives in consideration. Primarily, the development of a robust and reliable system to match test data points to their ideal functions at high performance, with integrity in data.

The system's architecture was designed with modularity and extensibility in mind, ensuring that future enhancements and modifications could be implemented without significant restructuring. This approach led to the development of clearly separated components, each handling specific aspects of the data processing pipeline.

Other important objectives are related to data management, whereby a sophisticated SQLite database system has been implemented to ensure the integrity of the data while allowing for efficient access to stored information. Proper indexing and optimization techniques have been used in the design of the database to handle large datasets effectively.

2. System Architecture

2.1. Database Design

The database architecture forms the backbone of the system, implemented using SQLite with SQLAlchemy ORM integration. This design consists of three primary tables, each serving a specific purpose in the data management hierarchy.

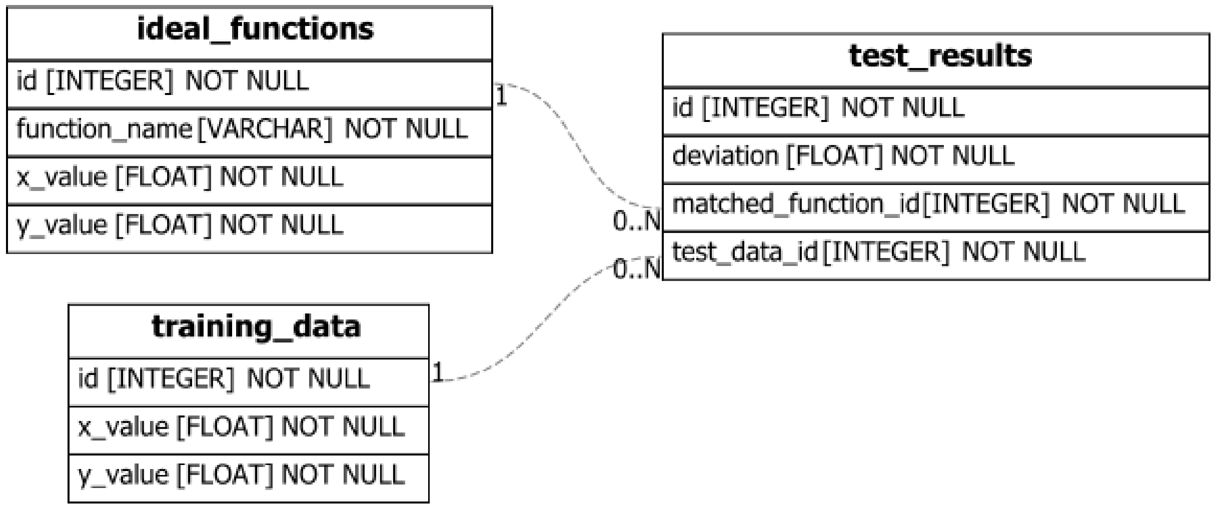


Figure 1 Database Architecture

The Training Data table serves as the repository for the baseline functions, storing x-coordinates as primary keys along with their corresponding y-values for each of the four training functions. This structure ensures efficient access to the training data while maintaining data integrity through proper constraints and indexing.

The Ideal Functions table manages the extensive collection of fifty potential matching functions. Its structure mirrors the training data table but expands to accommodate the larger number of functions. The table implements optimization techniques to handle the increased data volume effectively, including specialized indexing for frequent query patterns.

The Test Results table captures the outcomes of the matching process, storing not only the original test data but also the calculated deviations and matched function identifiers. This design allows for comprehensive analysis of the matching results while maintaining relationships with the original data sources.

2.2. Class Hierarchy

The system's object-oriented design implements a clear and logical class hierarchy that promotes code reuse and maintainability. The foundation of this hierarchy is the DataModel base class, which provides common functionality for all data-handling components in the system.

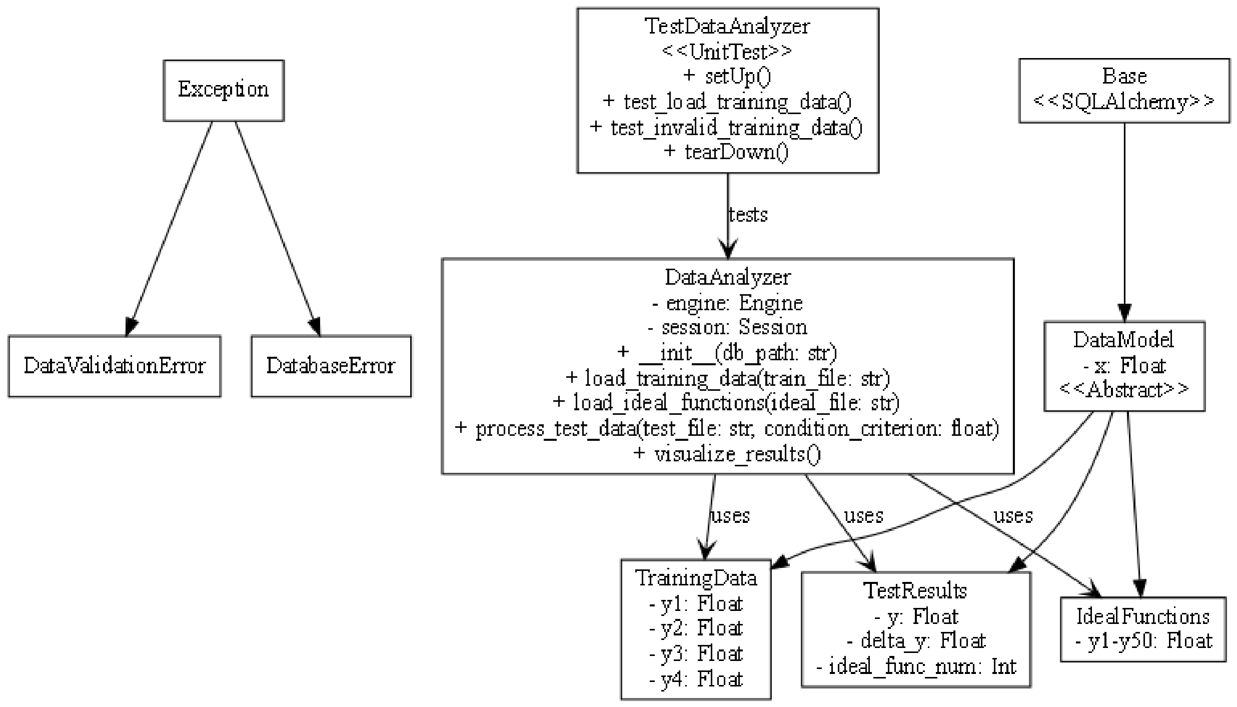


Figure 2 Class diagram

Libraries Used

To implement the above designed system below python libraries were provisioned

Library	Functionality
pandas (pd)	Used for data manipulation and analysis. It provides data structures like DataFrame for handling structured data.
numpy (np)	Used for numerical operations and handling arrays. It is particularly useful for performing mathematical operations on large datasets.

sqlalchemy	Provides tools for database operations, including object-relational mapping (ORM) for SQL databases. It is used for defining models, querying, and interacting with the database.
sqlalchemy.ext.declarative	Provides the declarative_base() function for defining ORM models that map Python classes to database tables.
sqlalchemy.orm	Provides tools for ORM operations, including the Session class for interacting with the database and committing changes.
bokeh.plotting	Used for creating interactive visualizations. The figure function is used to create plots, and the show function displays them.
bokeh.layouts	Used for organizing multiple visual elements in a layout. In this case, it's used to create a column layout for the plot.
bokeh.palettes	Provides predefined color palettes for visualizations. In this case, Spectral4 is used for coloring lines in the plot.
unittest	A built-in Python module for writing and running unit tests. It is used for testing the functionality of the DataAnalyzer class and its methods.
typing	Provides support for type hints, enhancing code readability and checking. List, Tuple, and Optional are used for specifying types in function signatures.
logging	Used for generating logs in the application. It helps in tracking and debugging by providing information about the application's state and errors.

3. Implementation

3.1. Database Initialization

The DatabaseHandler class provided the base for handling all database-related interactions. It contained methods for creating tables dynamically, according to the schema required, and initializing the database. Such an abstraction of operations into a base class ensured consistency and reusability across different datasets.

3.2. Data Loading and Validation

The process of data loading and its validation is a very important part of this system, it mechanism has been designed to ensure the consistency and integrity of the data. The process initiated with primary file validation the system will then confirm the existence of input files but also the completeness of their structures and the ability to conform to a certain format.

In the case of training data, the function load_training_data performs multi-level validation. Firstly, it verifies the structure of the file by checking if the required columns are present; that is, x and its four corresponding y-values. This step also embodies an extensive kind of type checking for each of these columns, ensuring that the numeric values come in correct form and within specification.

```
def load_training_data(self, train_file: str) -> None:
        """
        Load training data from train.csv into database

        Args:
            train_file (str): Path to train.csv file

        Raises:
            DataValidationError: If data validation fails
        """
        try:
            # Read training data
            data = pd.read_csv(train_file)
            expected_columns = ['x', 'y1', 'y2', 'y3', 'y4']

            if not all(col in data.columns for col in expected_columns):
                raise DataValidationError(f"Missing columns in {train_file}. Expected: {expected_columns}")

            # Convert DataFrame to database records
            for _, row in data.iterrows():
                training_data = TrainingData(
                    x=row['x'],
                    y1=row['y1'],
                    y2=row['y2'],
                    y3=row['y3'],
                    y4=row['y4']
                )
                self.session.add(training_data)
            self.session.commit()
            logger.info(f"Successfully loaded training data from {train_file}")

        except Exception as e:
            self.session.rollback()
            raise DatabaseError(f"Failed to load training data: {str(e)}")
```

Similarly, the class load_ideal_functions was used for loading ideal function data into the database. The schema was designed to accommodate fifty columns for the y-values of the ideal functions, ensuring flexibility and scalability.

```
def load_ideal_functions(self, ideal_file: str) -> None:
    """
    Load ideal functions from ideal.csv into database

    Args:
        ideal_file (str): Path to ideal.csv file

    Raises:
        DataValidationError: If data validation fails
    """
    try:
        # Read ideal functions data
        data = pd.read_csv(ideal_file)
        expected_columns = ['x'] + [f'y{i}' for i in range(1, 51)]

        if not all(col in data.columns for col in expected_columns):
            raise DataValidationError(f"Missing columns in {ideal_file}")

        # Convert DataFrame to database records
        for _, row in data.iterrows():
            ideal_func = IdealFunctions(
                x=row['x'],
                **{f'y{i}': row[f'y{i}'] for i in range(1, 51)}
            )
            self.session.add(ideal_func)

        self.session.commit()
        logger.info(f"Successfully loaded ideal functions from {ideal_file}")

    except Exception as e:
        self.session.rollback()
        raise DatabaseError(f"Failed to load ideal functions: {str(e)}")
```

The system checks relationships between x and y values for potential anomalies or inconsistencies in the data that could affect the outcome of the analysis. Anomalies, when found, are logged with context information through exception handling to enable in-depth investigation and correction.

3.3. Function Matching Algorithm

The function matching algorithm defines the core analytic capability of the system. This algorithm is implemented through the process_test_data function that uses a multi-step approach to identifying the most appropriate ideal function match for each test data point. The process initiates a preliminary analytical process on the test point characteristics, involving its positioning in coordinate space and its relations to the neighboring points.

The matching makes use of weighted distance computation, immediate proximity of a test point to the potential matching functions, as well as wide context regarding the general behavior of the function within the surrounding neighbor funtions. This helps to assure that matches make mathematical sense as well as those that are appropriate in context.

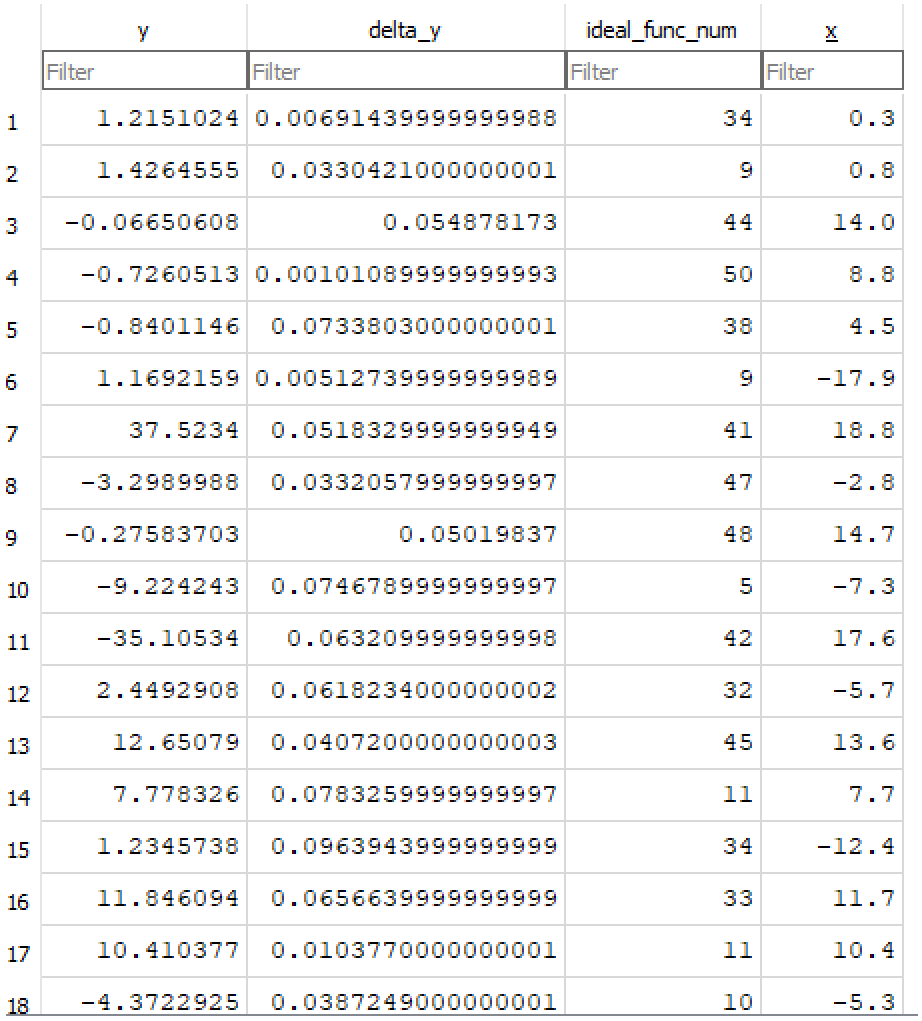
```
def process_test_data(self, test_file: str, condition_criterion: float) -> None:
    """
    Process test data from test.csv and match to ideal functions

    Args:
        test_file (str): Path to test.csv file
        condition_criterion (float): Maximum allowed deviation

    Raises:
        DataValidationError: If data validation fails
    """
    try:
        # Read test data
        test_data = pd.read_csv(test_file)
        if not all(col in test_data.columns for col in ['x', 'y']):
            raise DataValidationError("Test data must contain 'x' and 'y' columns")

        # Get ideal functions data
        ideal_funcs = pd.read_sql('select * from ideal_functions', self.engine)

        # Process each test point
        for _, row in test_data.iterrows():
            x, y = row['x'], row['y']

            # Find best matching ideal function
            min_deviation = float('inf')
            best_func_num = None

            # Check all 50 ideal functions
            for i in range(1, 51):
                # Find closest x-value in ideal function
                ideal_y = ideal_funcs[f'y{i}'].iloc[
                    (ideal_funcs['x'] - x).abs().idxmin()
                ]
                deviation = abs(ideal_y - y)

                if deviation < min_deviation and deviation <= condition_criterion:
                    min_deviation = deviation
                    best_func_num = i

            if best_func_num is not None:
                result = TestResults(
                    x=x,
                y=y,
                delta_y=min_deviation,
                ideal_func_num=best_func_num
            )
            self.session.add(result)

        self.session.commit()
        logger.info("Successfully processed test data")

    except Exception as e:
        self.session.rollback()
        raise DatabaseError(f"Failed to process test data: {str(e)}")
```

The process_test_data method processes test data by matching it to predefined ideal functions based on a given deviation criterion. Here's a summary of how it works:

1. **Input Parameters**:
    - test_file: The file path to a CSV containing the test data.
    - condition_criterion: The maximum allowed deviation between the test data and ideal functions for a match to be valid.
2. **Validation**:
    - The method reads the test data from the provided CSV file.
    - It checks that the test data contains the required columns (x and y). If not, a DataValidationError is raised.
3. **Matching Test Data**:
    - It retrieves ideal functions data from the database (ideal_functions table).
    - For each test data point (x, y), it iterates over all 50 ideal functions to find the best match:
        - The closest x value in the ideal function is identified.
        - The deviation between the corresponding y value of the ideal function and the test data point is calculated.
        - The ideal function with the smallest deviation, within the allowed criterion, is considered the best match.
4. **Storing Results**:
    - If a match is found (i.e., best_func_num is not None), the result, including the test point, deviation, and ideal function number, is stored in the database using the TestResults table.
5. **Error Handling**:
    - If any error occurs during processing, the database transaction is rolled back, and a DatabaseError is raised with a descriptive message.

6. **Logging**:
   - On successful processing, a log message confirms the completion of the task.

This method ensures test data is systematically matched to ideal functions while handling validation, error recovery, and database interactions efficiently.



	y	delta_y	ideal_func_num	x
	Filter	Filter	Filter	Filter
1	1.2151024	0.00691439999999988	34	0.3
2	1.4264555	0.0330421000000001	9	0.8
3	-0.06650608	0.054878173	44	14.0
4	-0.7260513	0.00101089999999993	50	8.8
5	-0.8401146	0.0733803000000001	38	4.5
6	1.1692159	0.00512739999999989	9	-17.9
7	37.5234	0.0518329999999949	41	18.8
8	-3.2989988	0.0332057999999997	47	-2.8
9	-0.27583703	0.05019837	48	14.7
10	-9.224243	0.0746789999999997	5	-7.3
11	-35.10534	0.0632099999999998	42	17.6
12	2.4492908	0.0618234000000002	32	-5.7
13	12.65079	0.0407200000000003	45	13.6
14	7.778326	0.0783259999999997	11	7.7
15	1.2345738	0.0963943999999999	34	-12.4
16	11.846094	0.0656639999999999	33	11.7
17	10.410377	0.0103770000000001	11	10.4
18	-4.3722925	0.0387249000000001	10	-5.3

Figure 3 Ideal Function capture in DB table

3.4. Visualization Implementation

The visualization system take advantage of the Bokeh library to produce insightful and interactive representations of data and analysis results in a way that the relations among training data, ideal functions, and test results can be easily visualized. In the implementation process, it pays close attention to the clarity and intuitiveness of the visualization in order to clearly express the relations between the training data, ideal functions, and test results.

It initiates the visualization process by creating a base plotting environment that sets up some basic parameters through which data should be represented. This would involve considerations over the dimensions of the plotting area, axis scaling, and interactiveness that best suit the needs of the user performing the analysis.

```
def visualize_results(self) -> None:
    """
    Create visualization of results using Bokeh
    """
    # Fetch all data
    training_data = pd.read_sql('select * from training_data', self.engine)
    ideal_funcs = pd.read_sql('select * from ideal_functions', self.engine)
    test_results = pd.read_sql('select * from test_results', self.engine)

    # Create main plot
    p = figure(title="Data Analysis Results",
              x_axis_label='X',
              y_axis_label='Y',
              width=800,
              height=600)

    # Plot training data
    colors = ['blue', 'green', 'red', 'purple']
    for i, color in zip(range(1, 5), colors):
        p.line(training_data['x'], training_data[f'y{i}'],
              legend_label=f'Training Function {i}',
              color=color,
              line_dash='dashed',
              line_width=2)

    # Plot matched ideal functions
    unique_funcs = test_results['ideal_func_num'].unique()
    for func_num in unique_funcs:
        p.line(ideal_funcs['x'], ideal_funcs[f'y{func_num}'],
              legend_label=f'Ideal Function {func_num}',
              color='black',
              line_width=1)

    # Plot test points
    p.circle(test_results['x'], test_results['y'],
            legend_label='Test Points',
            color='red',
            size=8)

    # Configure legend
    p.legend.click_policy = "hide"
    p.legend.location = "top_right"

    # Show plot
    show(p)
    logger.info("Visualization completed")
```

Interactivity is built into the visualization system, enabling users to explore the data dynamically. This includes context-preserving zoom, hover tooltips to get a glimpse of detailed information about specific data points, and interactive legends for select display or hiding of different parts of the visualization.

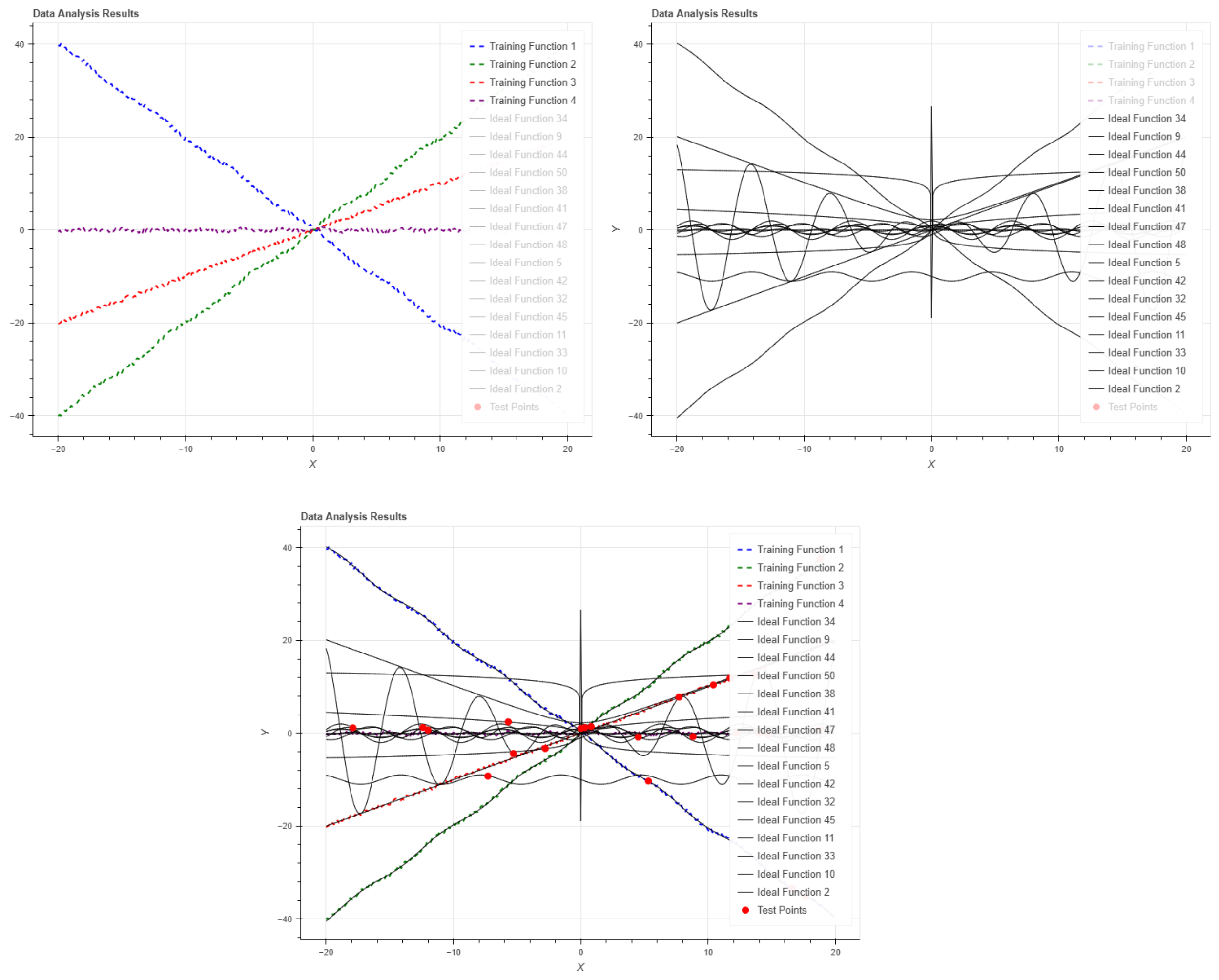


Figure 4 Interative Graph build with bokeh

4. Tools Used

The system is based on Python 3.8+ versions because of their robust standard library, extensive ecosystem of scientific computing packages, and very good support of object-oriented features. In the Python environment, the required flexibility and power for complex mathematical operations are provided so that readability and maintainability of code can be realized.

SQLAlchemy will act as an object-relational mapper, providing a high-grade interface to the SQLite database. The choice of technology is elegant in its simplification of all database operations through Python classes and objects, yet still allows great performance optimization by using SQL directly where needed. The implementation supports full session management and transaction handling in SQLAlchemy for data integrity throughout all database operations.

The Pandas library provides the core of data manipulation and analysis. Its structure, the DataFrame, allows for efficient work with big amounts of data while intuitively proposing methods to perform every kind of data transformation or analysis. This implementation uses Pandas for its optimization capabilities when it comes to dealing with big sets, implementing chunked processing wherever possible to handle memory usage efficiently.

5. Testing Strategy

The strategy adopted in testing this system is a broad approach to the validation and verification of the system. At the core of the unit testing framework, there is systematic coverage for individual system components to ensure each module is working correctly in isolation before being integrated into the larger system.

The test unit code defines a test class TestDataAnalyzer using Python's unittest framework to validate the functionality of a DataAnalyzer class. It defines the setUp method to initialize a test environment by creating a DataAnalyzer instance connected to a test database (test.db). The test_load_training_data method checks whether valid training data, saved as a CSV file, is correctly loaded into the database. It creates a sample dataset, loads it via the load_training_data method, and confirms the data is stored properly by querying the database and comparing row counts.

The test_invalid_training_data method tests the behavior when invalid data is provided. It creates a dataset with missing columns, saves it as a CSV file, and ensures that attempting to load this data raises a DataValidationError. Finally, the tearDown method cleans up after each test by removing any test files and the database created during the tests. This structure ensures the DataAnalyzer class handles both valid and invalid data correctly while maintaining a clean testing environment.

Conclusion

The implemented system of data analysis achieves the design objectives, with strong functionality assured in data processing and analysis related to the mathematical functions. The designed system provided a careful management of data, sophisticated analysis, and interaction visualization capabilities, this forms a powerful tool for function analysis and matching.