The script includes two classes: `InvalidParameterError` and `DatabaseConnector`.

`InvalidParameterError` is a custom exception class to handle errors when an invalid parameter is used in a function.

`DatabaseConnector` is used for managing database connections. It includes four private attributes for storing database details, and two methods:

1. `__init__`: Initializes the class with database details.
2. `create_engine`: Creates a database engine using SQLAlchemy. If unsuccessful, it handles the exception and returns `None`.

These classes help in organizing and managing database connections effectively in Python applications.

In [ ]:
```
from sqlalchemy import create_engine
from sqlalchemy.exc import SQLAlchemyError


class InvalidParameterError(Exception):
    """
    This class is a custom exception that is used to indicate that there was an error with a parameter in a function call.

    Attributes:
        parameter (str): The name of the parameter that caused the error.
        message (str): An optional error message.
    """

    def __init__(self, parameter, message="is invalid"):
        """
        The constructor for the InvalidParameterError class.

        Parameters:
            parameter (str): The name of the parameter that caused the error.
            message (str): An optional error message.
        """
        self.parameter = parameter
        self.message = f"Parameter {parameter} {message}"
        super().__init__(self.message)
```

```python
class DatabaseConnector:
    """
    This class handles the details of connecting to a database.

    Attributes:
        __db_url (str): The URL of the database.
        __db_name (str): The name of the database.
        __db_user (str): The username to use when connecting to the database.
        __db_password (str): The password to use when connecting to the database.
    """

    def __init__(self, db_url, db_name, db_user, db_password):
        """
        The constructor for the DatabaseConnector class.

        Parameters:
            db_url (str): The URL of the database.
            db_name (str): The name of the database.
            db_user (str): The username to use when connecting to the database.
            db_password (str): The password to use when connecting to the database.

        Raises:
            InvalidParameterError: If any of the parameters are None or empty.
        """

        # Validate the input parameters
        if not db_url or not db_name or not db_user or not db_password:
            raise InvalidParameterError(
                "db_url/db_name/db_user/db_password", "cannot be None or empty"
            )

        self.__db_url = db_url
        self.__db_name = db_name
        self.__db_user = db_user
        self.__db_password = db_password

    def create_engine(self):
        """
        This method creates and returns a database engine.
```

```
        Returns:
            Engine: A SQLAlchemy Engine instance.

        Raises:
            SQLAlchemyError: If there was a problem creating the engine.
        """
        try:
            return create_engine(
                f"mysql+pymysql://{self.__db_user}:{self.__db_password}@{self.__db_url}/{self.__db_name}"
            )
        except SQLAlchemyError as e:
            print(f"Failed to create the database engine: {str(e)}")
            return None
```

The `DataHandler` class, inheriting from `DatabaseConnector`, is designed for data operations on a MySQL database.

1. `read_data(self, table_name)`: Reads data from a specified table in the database and returns a pandas DataFrame.
2. `read_csv(self, csv_file_path)`: Reads a CSV file into a pandas DataFrame. If the file doesn't exist or has incorrect formatting, it raises an exception.
3. `store_in_database(self, dataframe, table_name)`: Stores a DataFrame in a MySQL database, replacing the table if it already exists.
4. `process_training_datasets(self, file_paths)`: Reads all training datasets from given file paths and stores them in the MySQL database.

The `InvalidFilePathError` is a custom exception class used to indicate that an invalid file path was used.

In [ ]:

```
import pandas as pd
from sqlalchemy.exc import SQLAlchemyError
import os


class InvalidFilePathError(Exception):
    """
    Exception raised when an invalid file path is passed.
```

```python
    Attributes:
        filepath (str): The file path that caused the error.
        message (str): An error message.
    """

    def __init__(self, filepath, message="is invalid"):
        """
        Initialize the InvalidFilePathError instance.

        Args:
            filepath (str): The file path that caused the error.
            message (str, optional): An error message. Defaults to "is invalid".
        """
        self.filepath = filepath
        self.message = f"File path {filepath} {message}"
        super().__init__(self.message)


class DataHandler(DatabaseConnector):
    """
    Class to handle data operations on a MySQL database.
    """

    def read_data(self, table_name):
        """
        Reads the data from the specified table in the database and returns a pandas dataframe.

        Args:
            table_name (str): The name of the database table to read.

        Returns:
            DataFrame: The table data as a pandas DataFrame.
        """
        try:
            with self.create_engine().connect() as conn:
                return pd.read_sql_table(table_name, conn)
        except SQLAlchemyError as e:
            print(f"Failed to read data from the database: {str(e)}")
            return None
```

```python
    def read_csv(self, csv_file_path):
        """
        Reads in a CSV file and returns a pandas dataframe.

        Args:
            csv_file_path (str): The path to the CSV file to read.

        Returns:
            DataFrame: The CSV data as a pandas DataFrame.
        """
        if not os.path.isfile(csv_file_path):
            raise InvalidFilePathError(csv_file_path)

        try:
            return pd.read_csv(csv_file_path, index_col=0)
        except pd.errors.ParserError as e:
            print(f"Failed to read the CSV file: {str(e)}")
            return None

    def store_in_database(self, dataframe, table_name):
        """
        Stores a pandas dataframe in a MySQL database.

        Args:
            dataframe (DataFrame): The pandas DataFrame to store.
            table_name (str): The name of the database table to store the data in.
        """
        try:
            with self.create_engine().connect() as conn, conn.begin():
                dataframe.to_sql(table_name, conn, if_exists="replace")
        except SQLAlchemyError as e:
            print(f"Failed to store data in the database: {str(e)}")

    def process_training_datasets(self, file_paths):
        """
        Reads in all training datasets and stores them in the MySQL database.

        Args:
            file_paths (list[str]): A list of file paths to the training datasets.
```

```
        """
        for file_path in file_paths:
            df = self.read_csv(file_path)
            if df is not None:
                table_name = os.path.splitext(os.path.basename(file_path))[0]
                self.store_in_database(df, table_name)
```

In this code, the `InvalidTableNameError` class is a custom exception for invalid table names. It takes the invalid table name and an optional error message as input.

The `DataAnalyzer` class is a subclass of the `DatabaseConnector` class. It has a single method `analyze_dataset()`, which takes a table name as input. The method connects to the database and executes a SQL query to select all records from the specified table. It then prints the info and shape of the resulting DataFrame. If the table name is None or empty, an `InvalidTableNameError` is raised. If there's any error connecting to the database or executing the query, it catches the `SQLAlchemyError` exception and prints an error message.

In [ ]:

```
from sqlalchemy.exc import SQLAlchemyError
from sqlalchemy.sql import text
import pandas as pd


class InvalidTableNameError(Exception):
    """
    Exception raised when an invalid table name is passed.

    Attributes:
        table_name (str): The table name that caused the error.
        message (str): An error message.
    """

    def __init__(self, table_name, message="is invalid"):
        """
        Initialize the InvalidTableNameError instance.

        Args:
            table_name (str): The table name that caused the error.
            message (str, optional): An error message. Defaults to "is invalid".
        """
        self.table_name = table_name
```

```python
        self.message = f"Table name {table_name} {message}"
        super().__init__(self.message)


class DataAnalyzer(DatabaseConnector):
    """
    Class to analyze data from a MySQL database.
    """

    def analyze_dataset(self, table_name):
        """
        Loads the training data from the specified table name in the database
        and analyzes it using Pandas functions.

        Args:
            table_name (str): The name of the database table to read and analyze.

        Raises:
            InvalidTableNameError: If the table name is None or empty.
        """
        if not table_name:
            raise InvalidTableNameError(table_name, "cannot be None or empty")

        try:
            with self.create_engine().connect() as conn:
                query = text(
                    f"SELECT * FROM {self._DatabaseConnector__db_name}.{table_name}"
                )
                df = pd.read_sql(query, conn)
                print(f"Dataset ({table_name}) info:\n")
                print(df.info())
                print(f"\nDataset ({table_name}) shape:\n")
                print(df.shape)
        except SQLAlchemyError as e:
            print(f"Failed to analyze the Dataset: {str(e)}")
```

In this code, the `IdealFunctionFinder` class accepts a training dataset and a set of ideal functions (both as pandas DataFrames). It calculates the best fit function for each column of the training dataset by minimizing the sum of squared deviations from the training data to each ideal function. The best fit functions, their corresponding deviations, and the maximum deviation across all functions are stored as attributes of the instance. These can be visualized in a table using matplotlib. If the input datasets are not pandas DataFrames, an `InvalidDataFrameError` is raised.

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


class InvalidDataFrameError(Exception):
    """
    This class is a custom exception that is used to indicate that there was an error with a DataFrame in a function call.

    Attributes:
        message (str): An error message.
    """

    def __init__(self, message="Invalid DataFrame"):
        """
        Initialize the InvalidDataFrameError instance.

        Args:
            message (str, optional): An error message. Defaults to "Invalid DataFrame".
        """
        self.message = message
        super().__init__(self.message)


class IdealFunctionFinder:
    """
    This class is used to find the best fit functions for a given training dataset from a set of ideal functions.

    Attributes:
        df_train (DataFrame): The training dataset.
        df_ideal (DataFrame): The ideal functions.
    """

    def __init__(self, df_train, df_ideal):
        """
        Initialize the IdealFunctionFinder instance.
```

```
        Args:
            df_train (DataFrame): The training dataset.
            df_ideal (DataFrame): The ideal functions.

        Raises:
            InvalidDataFrameError: If df_train or df_ideal is not a pandas DataFrame.
        """
        if not isinstance(df_train, pd.DataFrame) or not isinstance(
            df_ideal, pd.DataFrame
        ):
            raise InvalidDataFrameError(
                "df_train and df_ideal must be pandas DataFrames"
            )

        self.df_train = df_train
        self.df_ideal = df_ideal

    def calculate_best_fit_functions(self):
        """
        Calculate the best fit functions for each column of the training dataset.
        The best fit function is the one that minimizes the sum of squared deviations from the training data.

        Returns:
            DataFrame: A dataframe containing the training dataset, the corresponding ideal function, and the deviation.
        """
        self.best_fit_functions = []
        deviation_list = []

        try:
            for i in range(1, len(self.df_train.columns)):
                train = self.df_train.iloc[:, i]
                ssd = np.sum(
                    (self.df_ideal.iloc[:, 1:] - train.values.reshape(-1, 1)) ** 2,
                    axis=0,
                )
                min_index = np.argmin(ssd)
                col_name = self.df_ideal.columns[
                    min_index + 1
                ]  # Add 1 to account for skipping the first column
```

```
            self.best_fit_functions.append(col_name)
            deviation_list.append(ssd[min_index])

        # Compute max deviation
        self.max_deviation = max(deviation_list)

        data = {
            "Training Dataset": self.df_train.columns[1:],
            "Ideal Function": self.best_fit_functions,
            "Deviation": deviation_list,
        }
        df_result = pd.DataFrame(data)

        return df_result
    except Exception as e:
        print(f"Failed to calculate best fit functions: {str(e)}")
        return None

def visualize_table(self, df_result):
    """
    Visualize the result of the best fit functions calculation in a table.

    Args:
        df_result (DataFrame): The result of the best fit functions calculation.

    Returns:
        Figure: A matplotlib figure showing the table.
    """
    if df_result is None:
        print("Cannot visualize table because df_result is None")
        return None

    try:
        fig, ax = plt.subplots(figsize=(8, 2))
        table = plt.table(
            cellText=df_result.values.tolist(),
            colLabels=df_result.columns,
            cellLoc="center",
            loc="center",
        )
```

```python
        table.auto_set_font_size(False)
        table.set_fontsize(10)
        table.scale(1, 1.5)
        cells = table.properties()["celld"]
        for i in range(len(df_result.columns)):
            cells[0, i].set_facecolor("#DBE2EF")  # color of header cells
            cells[0, i].set_fontsize(14)  # font size of header
        for i in range(len(df_result.values.tolist())):
            for j in range(len(df_result.columns)):
                cells[i + 1, j].set_facecolor("#F9F7F7")  # color of other cells

        ax.axis("off")
        plt.title(
            "Training Data Ideal Function Mapping \nWith Corresponding Min Deviation",
            fontsize=12,
            pad=10,
        )

        return fig
    except Exception as e:
        print(f"Failed to visualize table: {str(e)}")
        return None

def run(self):
    """
    Run the ideal function finder.

    Returns:
        tuple: A tuple containing the matplotlib figure, the maximum deviation, and the list of best fit functions.
    """
    df_result = self.calculate_best_fit_functions()
    fig = self.visualize_table(df_result)

    return fig, self.max_deviation, self.best_fit_functions
```

This code defines a class, `IdealFunctionMapping`, that extends `DatabaseConnector`. It takes in a set of test data and a set of ideal functions, and maps each test data point to the best matching ideal function(s). The best matching ideal function is the one that has the smallest absolute difference between its y-value and the y-value of the test data point. The results of the mapping process are stored in a list of dictionaries and can be written to a database. If the input datasets are not pandas DataFrames, an `InvalidDataFrameError` is raised.

In [ ]:
```
import numpy as np
import pandas as pd
from sqlalchemy.exc import SQLAlchemyError


class InvalidDataFrameError(Exception):
    """
    This class is a custom exception that is used to indicate that there was an error with a DataFrame in a function call.

    Attributes:
        message (str): An error message.
    """

    def __init__(self, message="Invalid DataFrame"):
        """
        Initialize the InvalidDataFrameError instance.

        Args:
            message (str, optional): An error message. Defaults to "Invalid DataFrame".
        """
        self.message = message
        super().__init__(self.message)


class IdealFunctionMapping(DatabaseConnector):
    """
    This class is used to map test data to the best matching ideal functions from a set of ideal functions.

    Attributes:
        df_test (DataFrame): The test dataset.
        df_ideal (DataFrame): The ideal functions.
        ideal_functions (list): The names of the ideal functions.
        cutoff (float): The cutoff for the deviation from the ideal function.
        results (list): The results of the mapping process.
    """
```

```python
    def __init__(
        self, db_password, db_name, df_test, df_ideal, ideal_functions, cutoff
    ):
        """
        Initialize the IdealFunctionMapping instance.

        Args:
            db_password (str): The password to use when connecting to the database.
            db_name (str): The name of the database.
            df_test (DataFrame): The test dataset.
            df_ideal (DataFrame): The ideal functions.
            ideal_functions (list): The names of the ideal functions.
            cutoff (float): The cutoff for the deviation from the ideal function.

        Raises:
            InvalidDataFrameError: If df_test or df_ideal is not a pandas DataFrame.
        """
        super().__init__("localhost", db_name, "root", db_password)

        if not isinstance(df_test, pd.DataFrame) or not isinstance(
            df_ideal, pd.DataFrame
        ):
            raise InvalidDataFrameError(
                "df_test and df_ideal must be pandas DataFrames"
            )

        self.df_test = df_test
        self.df_ideal = df_ideal
        self.ideal_functions = ideal_functions
        self.cutoff = cutoff
        self.results = []

    def process_data(self):
        """
        Process the test data and map each data point to the best matching ideal function(s).

        Returns:
            list: A list of dictionaries containing the results of the mapping process.
        """
        try:
```

```python
            for idx, row in self.df_test.iterrows():
                x_test = row["x"]
                y_test = row["y"]
                for func_name in self.ideal_functions:
                    if x_test in self.df_ideal["x"].values:
                        y_ideal = self.df_ideal.loc[
                            self.df_ideal["x"] == x_test, func_name
                        ].values[0]
                        deviation = np.abs(y_test - y_ideal)
                        if deviation <= self.cutoff:
                            existing_rows = [
                                r
                                for r in self.results
                                if r["x"] == x_test and r["y"] == y_test
                            ]
                            if existing_rows:
                                existing_row = existing_rows[0]
                                existing_row["Ideal Function"] += ", " + func_name
                                existing_row["Deviation"] += ", " + str(deviation)
                            else:
                                self.results.append(
                                    {
                                        "x": x_test,
                                        "y": y_test,
                                        "Ideal Function": func_name,
                                        "Deviation": str(deviation),
                                    }
                                )
            return self.results
        except Exception as e:
            print(f"Failed to process data: {str(e)}")
            return None

    def write_to_database(self):
        """
        Write the results of the mapping process to a database.

        The database connection is handled by the DatabaseConnector superclass.
        """
        if not self.results:
```

```python
            print("No results to write to database")
            return

        try:
            result_df = pd.DataFrame(self.results)
            with self.create_engine().connect() as conn, conn.begin():
                result_df.to_sql("results", con=conn, if_exists="replace", index=False)
        except SQLAlchemyError as e:
            print(f"Failed to write results to database: {str(e)}")
```

This code defines two classes, `DataProcessor` and `Visualization`. `DataProcessor` is responsible for processing data and calculating mappings and deviations. `Visualization`, which inherits from `DataProcessor`, is responsible for visualizing the training data, test data, ideal function fitting, deviations, mappings, and assignment counts. It uses methods defined in the `DataProcessor` class and adds visualization capabilities to them.

In [ ]:

```python
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
from matplotlib.lines import Line2D


class DataProcessor:
    """
    A class for processing data and calculating mappings and deviations.

    Attributes:
        df_train (DataFrame): Training dataset.
        df_test (DataFrame): Test dataset.
        df_ideal (DataFrame): Ideal dataset.
        df_result (DataFrame): Result dataset.
        best_fit_functions (list): List of best fit functions.
        cutoff (float): Cutoff value for deviation.
    """

    def __init__(
        self, df_train, df_test, df_ideal, df_result, best_fit_functions, cutoff
    ):
        self.df_train = df_train
```

```python
        self.df_test = df_test
        self.df_ideal = df_ideal
        self.df_result = df_result
        self.best_fit_functions = best_fit_functions
        self.cutoff = cutoff

    def calculate_mappings(self):
        """
        Calculate the number of ideal functions each test data point is mapped to.

        Returns:
            list: A list of the number of mappings for each test data point.
        """
        mappings = []
        for idx, row in self.df_test.iterrows():
            x_test = row["x"]
            y_test = row["y"]
            count_mapped = 0
            for func_name in self.best_fit_functions:
                filtered_df = self.df_ideal.loc[self.df_ideal["x"] == x_test, func_name]
                if not filtered_df.empty:
                    y_ideal = filtered_df.values[0]
                    deviation = np.abs(y_test - y_ideal)
                    if deviation <= self.cutoff:
                        count_mapped += 1
            mappings.append(count_mapped)
        return mappings

    def calculate_deviations(self):
        """
        Calculate the deviations for each test data point from each ideal function.

        Returns:
            DataFrame: A DataFrame of deviations.
        """
        df_deviations = pd.DataFrame()
        for func_name in self.best_fit_functions:
            deviations = []
            for idx, row in self.df_test.iterrows():
                x_test = row["x"]
```

```
                y_test = row["y"]
                filtered_df = self.df_ideal.loc[self.df_ideal["x"] == x_test, func_name]
                if not filtered_df.empty:
                    y_ideal = filtered_df.values[0]
                    deviation = np.abs(y_test - y_ideal)
                    deviations.append(deviation)
                else:
                    deviations.append(None)
            df_deviations[func_name] = deviations
        return df_deviations


class Visualization(DataProcessor):
    def __init__(
        self, df_train, df_test, df_ideal, df_result, best_fit_functions, cutoff
    ):
        super().__init__(
            df_train, df_test, df_ideal, df_result, best_fit_functions, cutoff
        )
        sns.set(style="darkgrid")

    def plot_training_data(self):
        """
        Plot the training data.
        """
        sns.set(style="darkgrid")
        palette = ["#060047", "#E90064", "#4C4B16", "#FFE15D"]
        fig, axs = plt.subplots(2, 2, figsize=(16, 8))
        axs = axs.ravel()

        for i in range(1, len(self.df_train.columns)):
            axs[i - 1].plot(
                self.df_train["x"], self.df_train.iloc[:, i], color=palette[i - 1]
            )
            axs[i - 1].set_title(f"Plot for {self.df_train.columns[i]}")
            axs[i - 1].set_xlabel("X")
            axs[i - 1].set_ylabel("Y")

        plt.tight_layout()
        plt.show()
```

```python
def plot_ideal_function_fitting(self):
    """
    Plot the ideal function fitting for the training data.
    """

    sns.set(style="darkgrid")
    palette = sns.color_palette("coolwarm", len(self.df_result))

    fig, axes = plt.subplots(2, 2, figsize=(16, 8))
    axes = axes.ravel()

    for i, row in self.df_result.iterrows():
        training_dataset = self.df_train[row["Training Dataset"]]
        ideal_function = self.df_ideal[row["Ideal Function"]]
        axes[i].plot(
            self.df_train["x"],
            training_dataset,
            color=palette[0],
            linestyle="dotted",
            linewidth=1.75,
            label=row["Training Dataset"],
        )
        axes[i].plot(
            self.df_ideal["x"],
            ideal_function,
            color=palette[3],
            linestyle="solid",
            linewidth=1.0,
            label=row["Ideal Function"],
        )

        axes[i].set_title(
            f'Ideal Function Fitting for {row["Training Dataset"]}', fontsize=12
        )
        axes[i].set_xlabel("X", fontsize=10)
        axes[i].set_ylabel("Y", fontsize=10)
        axes[i].legend(fontsize=10)

    fig.tight_layout()
```

```python
        plt.show()

    def plot_test_data(self):
        """
        Plot the test data.
        """

        sns.set(style="darkgrid")
        fig, ax = plt.subplots(figsize=(8, 4))
        ax.scatter(
            self.df_test["x"], self.df_test["y"], color="#E90068", marker="s", s=5
        )
        ax.set_title("Scatter Plot for Test Dataset")
        ax.set_xlabel("X")
        ax.set_ylabel("Y")
        ax.set_yscale("symlog")
        plt.show()

    def calculate_and_plot_deviations(self, best_fit_functions, cutoff):
        """
        Calculate the deviations for each test data point and each ideal function
        and plot them in a 2x2 grid of subplots.

        Parameters:
        df_test (DataFrame): A dataframe with 'x' and y-values for the test dataset.
        df_ideal (DataFrame): A dataframe with 'x' and y-values for each ideal function.
        ideal_functions (List[str]): List of ideal function names.
        cutoff (float): The maximum allowed deviation for a point to be mapped to an ideal function.
        """
        # Set the style of seaborn for better looking plots
        sns.set(style="darkgrid")

        # Create a 2x2 grid of subplots with a light grey background
        fig, axes = plt.subplots(2, 2, figsize=(14, 10))
        fig.patch.set_facecolor("#D3D3D3")  # type: ignore # set background to light gray
        axes = axes.ravel()

        # Define a color palette for distinguishing different functions
        palette = ["#FFE15D", "#99DBF5", "#C58940", "#D18CE0"]
```

```
        # Calculate and plot the deviations for each ideal function
        for i, func_name in enumerate(self.best_fit_functions):
            ax = axes[i]
            deviations = []
            mapped = []
            x_values = []

            for idx, row in self.df_test.iterrows():
                x_test = row["x"]
                y_test = row["y"]
                filtered_df = self.df_ideal.loc[self.df_ideal["x"] == x_test, func_name]

                if not filtered_df.empty:
                    y_ideal = filtered_df.values[0]
                    deviation = np.abs(y_test - y_ideal)
                    deviations.append(deviation)
                    mapped.append(deviation <= cutoff)
                    x_values.append(x_test)
                else:
                    deviations.append(None)
                    mapped.append(False)
                    x_values.append(x_test)

            # Create a DataFrame for this function
            df_func = pd.DataFrame(
                {"x": x_values, "deviation": deviations, "mapped": mapped}
            )

            # Plot the deviations, using different colors for mapped and not mapped points
            ax.scatter(
                df_func["x"][df_func["mapped"]],
                df_func["deviation"][df_func["mapped"]],
                color=palette[i],
                marker="*",
                s=30,
                label=f"Deviations for {func_name} (mapped)",
            )
            ax.scatter(
                df_func["x"][~df_func["mapped"]],
                df_func["deviation"][~df_func["mapped"]],
```

```python
                color="black",
                marker="+",
                s=30,
                label=f"Deviations for {func_name} (not mapped)",
            )
            ax.set_title(f"Deviations for Ideal Function: {func_name}")
            ax.set_xlabel("X")
            ax.set_ylabel("Deviation")
            ax.set_yscale("log")  # Set y-axis to logarithmic scale
            ax.legend()

        # Adjust layout for better appearance
        plt.tight_layout()
        plt.show()

    def plot_mappings(self, mapping):
        """
        Plot test data points colored by the number of ideal functions they are mapped to.

        Parameters:
        df_test (DataFrame): A dataframe with 'x' and y-values for the test dataset. It must also contain a 'mappings' column.
        """
        # Define the color palette
        palette = ["#000000", "#99DBF5", "#E90064", "#4C4B16", "#FFE15D"]
        markers = ["+", "s", "*", "D", "^"]
        labels = [
            "Not mapped",
            "Mapped to 1 ideal function",
            "Mapped to 2 ideal functions",
            "Mapped to 3 ideal functions",
            "Mapped to 4 ideal functions",
        ]

        # Set the style of seaborn for better looking plots
        sns.set(style="dark")

        # Create a new figure
        fig = plt.figure(figsize=(10, 6))
        fig.patch.set_facecolor("#D3D3D3")  # Change background color of the canvas
```

```python
        df_mapping = self.df_test.copy()
        df_mapping["mappings"] = mapping

        # Plot y-values against x-values colored by the number of mappings
        for i in range(5):
            plt.scatter(
                df_mapping["x"][df_mapping["mappings"] == i],
                df_mapping["y"][df_mapping["mappings"] == i],
                c=palette[i],
                marker=markers[i],
                s=20,
            )

        # Customize the plot
        plt.title("Scatter Plot for Test Dataset")
        plt.xlabel("X")
        plt.ylabel("Y")
        plt.yscale("symlog")  # Set y-axis to symmetrical logarithmic scale

        # Create a legend
        legend_elements = [
            Line2D(
                [0],
                [0],
                marker=markers[i],
                color="w",
                markerfacecolor=palette[i],
                markersize=10,
                label=labels[i],
            )
            for i in range(5)
        ]
        plt.legend(handles=legend_elements, loc="upper left")

        plt.show()

    def plot_deviation_histograms(self, df_deviations):
        """
        Plot histograms of deviations for each ideal function.
```

```python
        Args:
            df_deviations (DataFrame): A DataFrame of deviations.
        """

        sns.set(style="darkgrid")
        fig, axes = plt.subplots(2, 2, figsize=(14, 10))
        axes = axes.ravel()
        fig.patch.set_facecolor("#D3D3D3")

        palette = ["#060047", "#E90064", "#4C4B16", "#FFE15D"]

        for i, column in enumerate(df_deviations.columns):
            ax = axes[i]
            sns.histplot(
                df_deviations[column].dropna(), ax=ax, color=palette[i]
            )  # dropna() to remove None values
            ax.set_title(f"Deviations for {column}")
            ax.set_xlabel("Deviation")
            ax.set_ylabel("Count")

        plt.tight_layout()
        plt.show()

    def plot_deviation_kde(self, df_deviations):
        """
        Plot Kernel Density Estimation (KDE) plots of deviations for each ideal function.

        Args:
            df_deviations (DataFrame): A DataFrame of deviations.
        """

        sns.set(style="darkgrid")
        fig, axes = plt.subplots(2, 2, figsize=(14, 10))
        axes = axes.ravel()
        fig.patch.set_facecolor("#D3D3D3")

        palette = ["#060047", "#E90064", "#4C4B16", "#FFE15D"]

        for i, column in enumerate(df_deviations.columns):
            ax = axes[i]
```

```
            sns.kdeplot(
                df_deviations[column].dropna(), ax=ax, color=palette[i]
            )  # dropna() to remove None values
            ax.set_title(f"Deviations for {column}")
            ax.set_xlabel("Deviation")
            ax.set_ylabel("Density")

        plt.tight_layout()
        plt.show()

    def plot_assignment_counts(self, df_deviations, best_fit_functions, cutoff):
        """
        Plot a bar chart of assignment counts with data labels.

        Parameters:
        df_deviations (DataFrame): A dataframe with deviations for each ideal function.
        ideal_functions (List[str]): List of ideal function names.
        cutoff (float): The maximum allowed deviation for a point to be mapped to an ideal function.
        """
        assignment_counts = {func_name: 0 for func_name in best_fit_functions}

        for idx, row in df_deviations.iterrows():
            for func_name in best_fit_functions:
                if row[func_name] <= cutoff:
                    assignment_counts[func_name] += 1

        # Convert the assignment counts dictionary to a DataFrame
        df_counts = pd.DataFrame(assignment_counts, index=[0])

        plt.figure(figsize=(6, 4))
        bars = plt.bar(df_counts.columns, df_counts.values[0])

        plt.title("Count of Test Data points Assigned to Ideal Functions ")
        plt.xlabel(" best_fit_functions")
        plt.ylabel("Count")

        # Add data labels
        for bar in bars:
            yval = bar.get_height()
            plt.text(
```

```python
                bar.get_x() + bar.get_width() / 2, yval, int(yval), va="bottom"
            )  # va: vertical alignment

        plt.show()

    def run_visualization(self, mapping, df_deviations, best_fit_functions, cutoff):
        """
        Run all the visualization functions.

        Args:
            mapping (list): A list of the number of mappings for each test data point.
            df_deviations (DataFrame): A DataFrame of deviations.
            best_fit_functions (list): List of best fit functions.
            cutoff (float): Cutoff value for deviation.
        """

        self.plot_training_data()
        self.plot_ideal_function_fitting()
        self.plot_test_data()
        self.calculate_and_plot_deviations(best_fit_functions, cutoff)
        self.plot_mappings(mapping)
        self.plot_deviation_histograms(df_deviations)
        self.plot_deviation_kde(df_deviations)
        self.plot_assignment_counts(df_deviations, best_fit_functions, cutoff)
```

This script is the main execution script that uses the classes defined earlier. It performs several operations:

1. It establishes a connection to a MySQL database.
2. It processes training datasets and stores them in the database.
3. It reads training and ideal data from the database.
4. It analyzes the data in the 'train_data' and 'ideal_data' tables in the database.
5. It uses the `IdealFunctionFinder` class to find the best fit functions for the training data and calculates the maximum deviation.
6. It reads test data, processes it and writes the results back to the database using the `IdealFunctionMapping` class.
7. It visualizes various aspects of the data using the `Visualization` class.

In [ ]:

```python
from sqlalchemy import create_engine, text
import matplotlib.pyplot as plt
import seaborn as sns
```

```python
import pandas as pd
import numpy as np
from matplotlib.lines import Line2D
import os
from urllib.parse import quote_plus

if __name__ == "__main__":
    # database configurations
    db_name = "iu_py_database"
    db_url = "localhost"
    db_user = "root"
    db_password = quote_plus("********")  # encrypt the password

    # create the database connection
    engine = create_engine(f"mysql+pymysql://root:{db_password}@localhost")
    conn = engine.connect()

    # create the database
    stmt = text(f"CREATE DATABASE {db_name}")  # SQL statement to create the database
    conn.execute(stmt)  # execute the SQL statement

    # paths to the dataset files
    train_path = "train.csv"
    ideal_path = "ideal.csv"
    test_path = "test.csv"

    file_paths = [train_path, ideal_path]

    # create a DataHandler object to handle data operations
    data_handler = DataHandler(db_url, db_name, db_user, db_password)

    # process the training datasets and store them in the database
    data_handler.process_training_datasets(file_paths)

    # read the training and ideal data from the database
    df_train = data_handler.read_data("train")
    df_ideal = data_handler.read_data("ideal")

    # create a DataAnalyzer object to analyze data from the database
    analyzer = DataAnalyzer(db_url, db_name, db_user, db_password)
```

```python
    # Analyze the 'train_data' table
    analyzer.analyze_dataset("train")
    # Analyze the 'ideal_data' table
    analyzer.analyze_dataset("ideal")

    # Instantiate and run the IdealFunctionFinder class
    finder = IdealFunctionFinder(df_train, df_ideal)
    fig, max_deviation, best_fit_functions = finder.run()
    plt.show()

    # Read the test data
    df_test = pd.read_csv(test_path)

    # cutoff for the test data
    cutoff = max_deviation * np.sqrt(2)

    # Instantiate and run the IdealFunctionMapping class
    db = IdealFunctionMapping(
        db_password, db_name, df_test, df_ideal, best_fit_functions, cutoff
    )
    db.process_data()
    db.write_to_database()

    # Instantiate and run the Visualization class
    df_result = finder.calculate_best_fit_functions()
    viz = Visualization(
        df_train, df_test, df_ideal, df_result, best_fit_functions, cutoff
    )
    mapping = viz.calculate_mappings()
    df_deviations = viz.calculate_deviations()
    viz.run_visualization(mapping, df_deviations, best_fit_functions, cutoff)
```

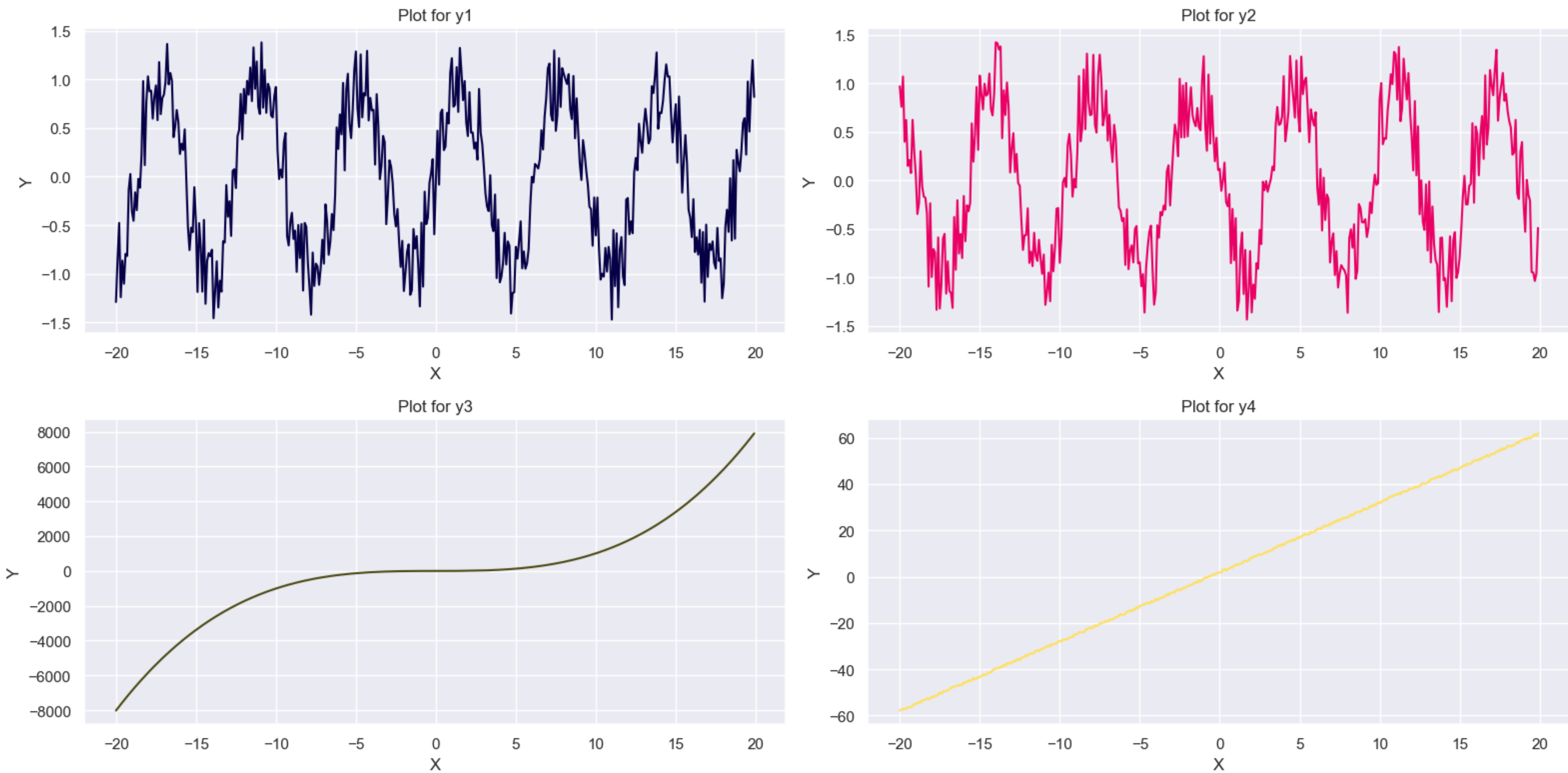
```
Dataset (train) info:

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 5 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   x       400 non-null    float64
 1   y1      400 non-null    float64
 2   y2      400 non-null    float64
 3   y3      400 non-null    float64
 4   y4      400 non-null    float64
dtypes: float64(5)
memory usage: 15.8 KB
None

Dataset (train) shape:

(400, 5)
Dataset (ideal) info:

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 51 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   x       400 non-null    float64
 1   y1      400 non-null    float64
 2   y2      400 non-null    float64
 3   y3      400 non-null    float64
 4   y4      400 non-null    float64
 5   y5      400 non-null    float64
 6   y6      400 non-null    float64
 7   y7      400 non-null    float64
 8   y8      400 non-null    float64
 9   y9      400 non-null    float64
 10  y10     400 non-null    float64
 11  y11     400 non-null    float64
 12  y12     400 non-null    float64
 13  y13     400 non-null    float64
```

```
 14  y14     400 non-null    float64
 15  y15     400 non-null    float64
 16  y16     400 non-null    float64
 17  y17     400 non-null    float64
 18  y18     400 non-null    float64
 19  y19     400 non-null    float64
 20  y20     400 non-null    float64
 21  y21     400 non-null    float64
 22  y22     400 non-null    float64
 23  y23     400 non-null    float64
 24  y24     400 non-null    float64
 25  y25     400 non-null    float64
 26  y26     400 non-null    float64
 27  y27     400 non-null    float64
 28  y28     400 non-null    float64
 29  y29     400 non-null    float64
 30  y30     400 non-null    float64
 31  y31     400 non-null    float64
 32  y32     400 non-null    float64
 33  y33     400 non-null    float64
 34  y34     400 non-null    float64
 35  y35     400 non-null    float64
 36  y36     400 non-null    float64
 37  y37     400 non-null    float64
 38  y38     400 non-null    float64
 39  y39     400 non-null    float64
 40  y40     400 non-null    float64
 41  y41     400 non-null    float64
 42  y42     400 non-null    float64
 43  y43     400 non-null    float64
 44  y44     400 non-null    float64
 45  y45     400 non-null    float64
 46  y46     400 non-null    float64
 47  y47     400 non-null    float64
 48  y48     400 non-null    float64
 49  y49     400 non-null    float64
 50  y50     400 non-null    float64
dtypes: float64(51)
memory usage: 159.5 KB
None
```

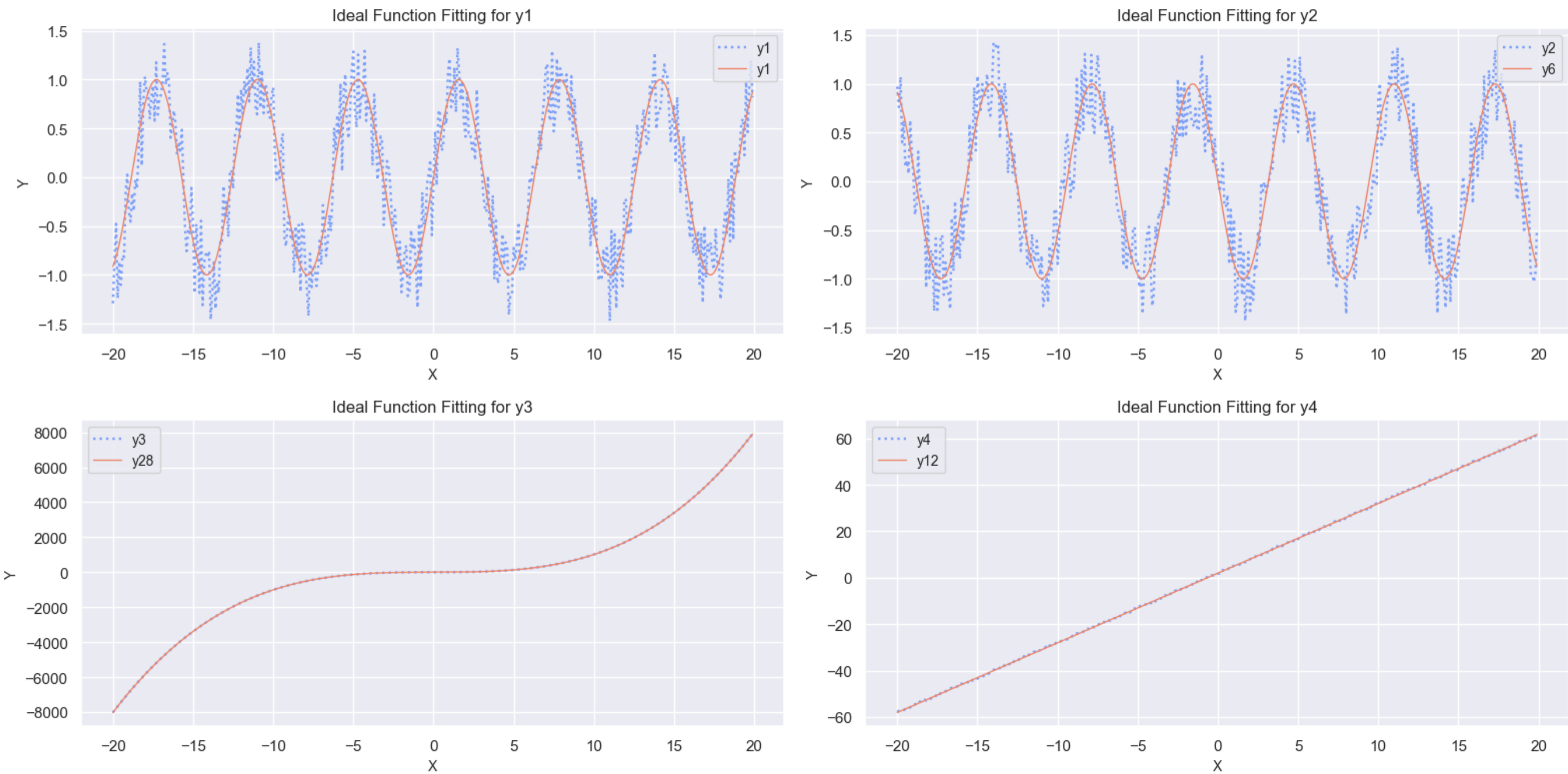
```
Dataset (ideal) shape:

(400, 51)
```

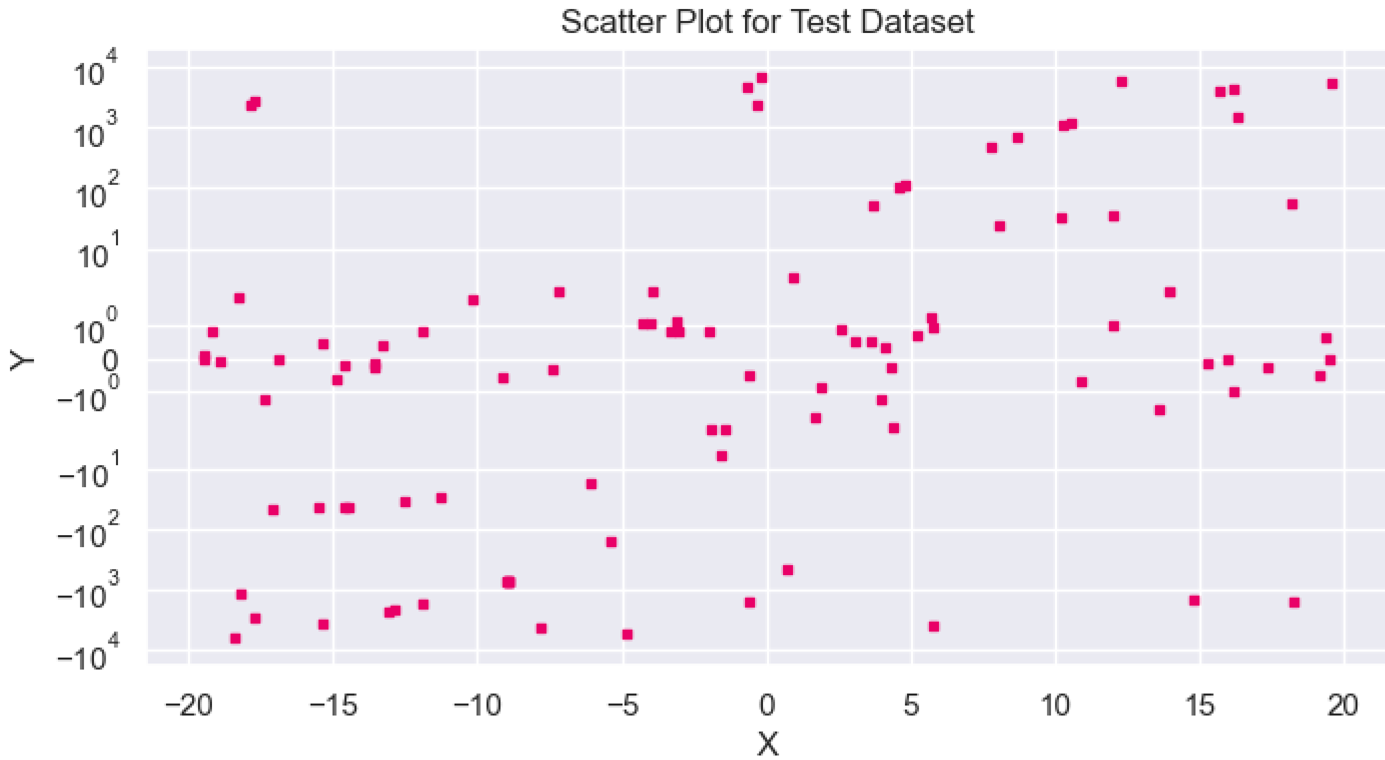
Training Data Ideal Function Mapping
With Corresponding Min Deviation

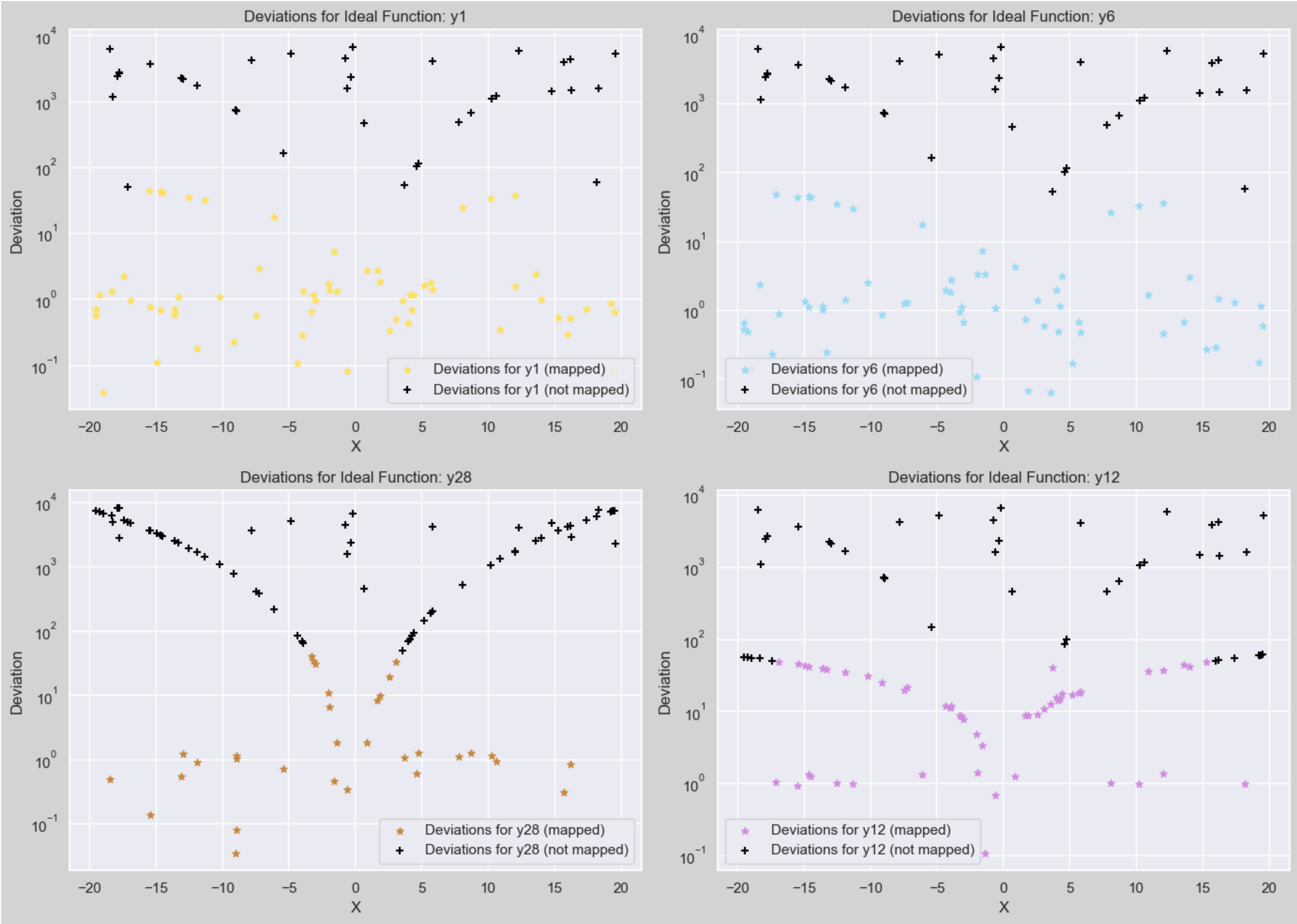
Training Dataset	Ideal Function	Deviation
y1	y1	30.3759234038353
y2	y6	30.86451202640528
y3	y28	34.58738779499796
y4	y12	33.76182777609962



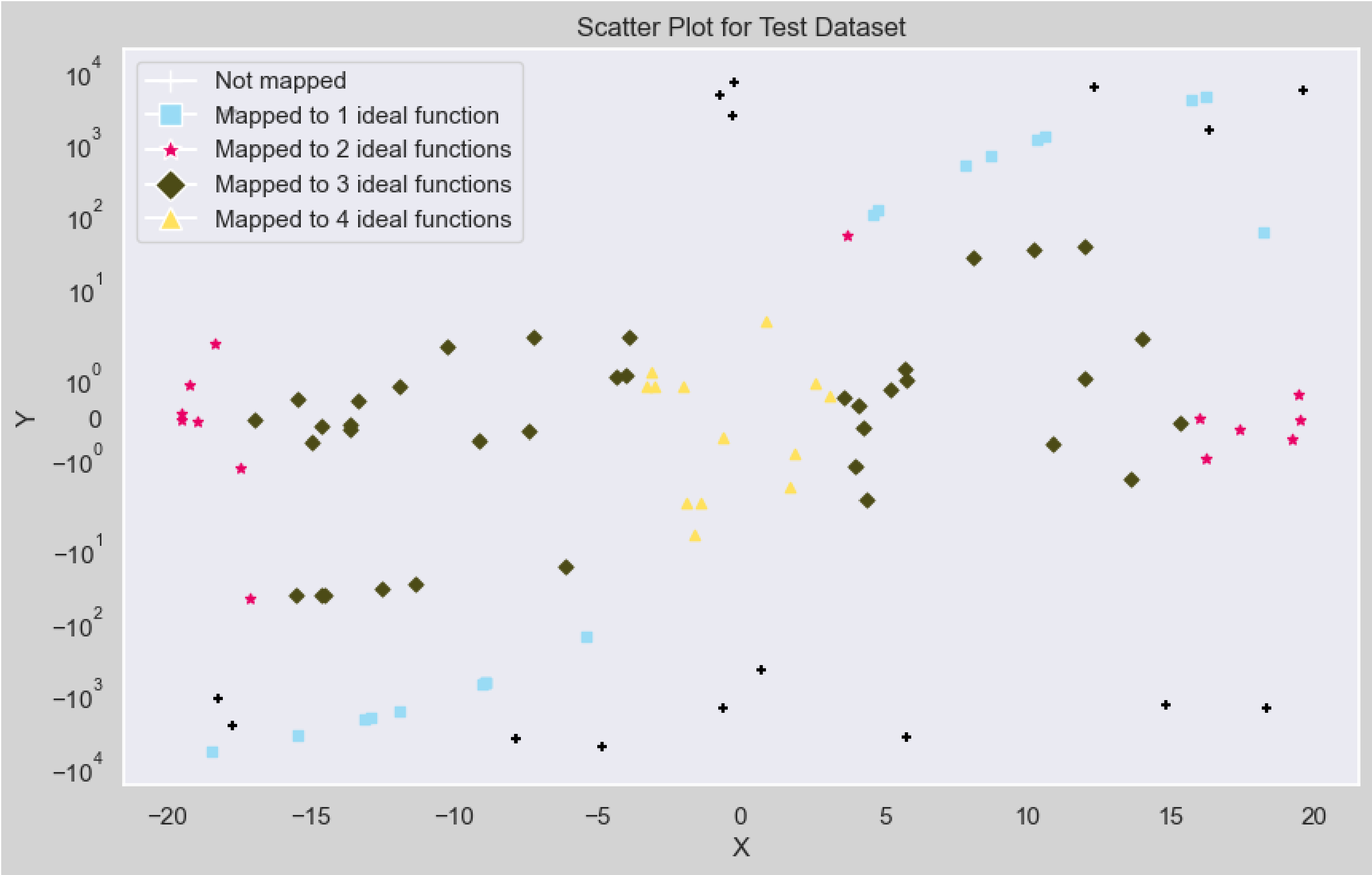
Plot for y1
Plot for y2
Plot for y3
Plot for y4
Y
X


Ideal Function Fitting for y1
y1
y1
Ideal Function Fitting for y2
y2
y6
Ideal Function Fitting for y3
y3
y28
Ideal Function Fitting for y4
y4
y12
X
Y

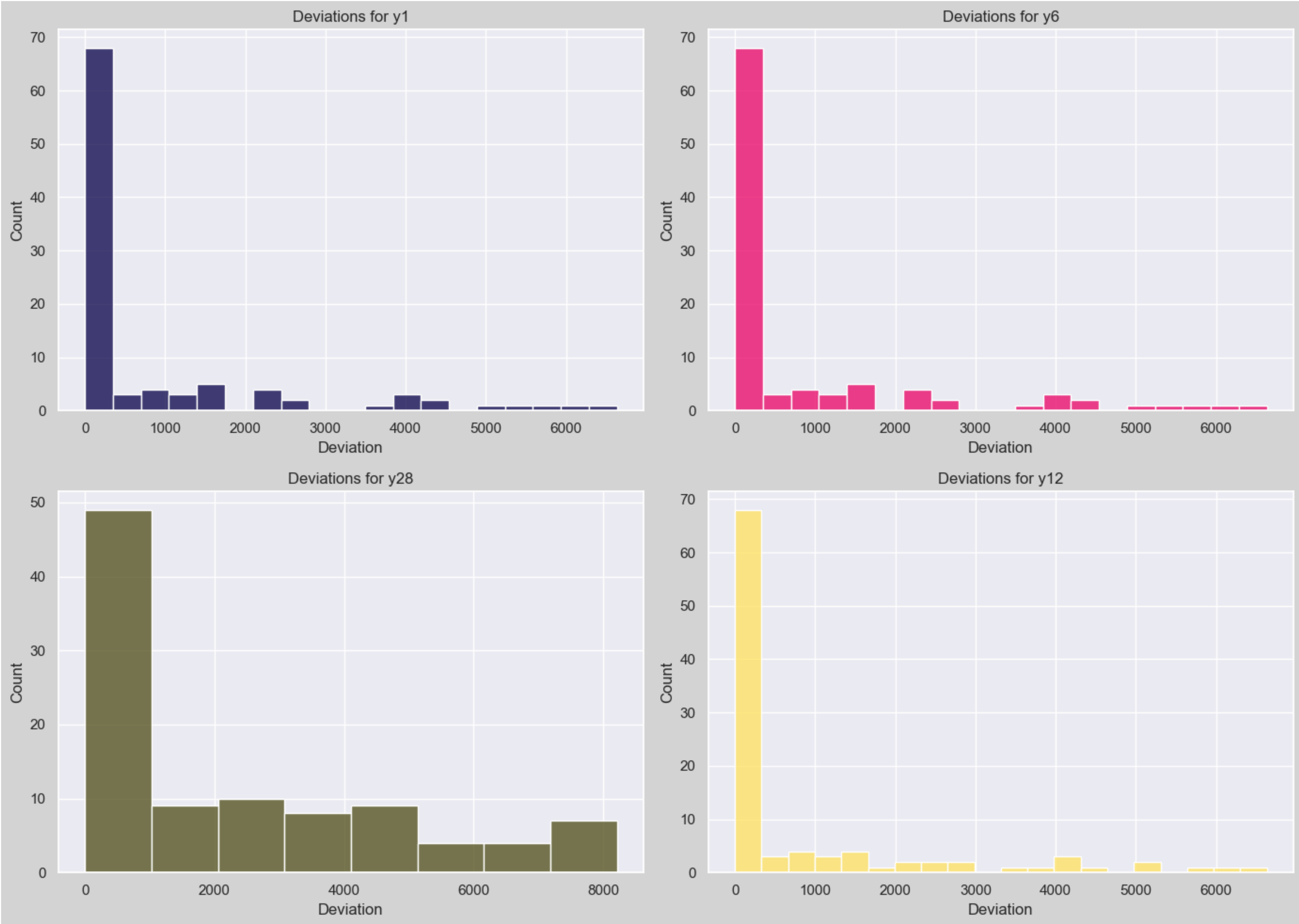

Scatter Plot for Test Dataset
Y
X
10^4
10^3
10^2
10^1
10^0
0
-10^0
-10^1
-10^2
-10^3
-10^4
−20
−15
−10
−5
0
5
10
15
20



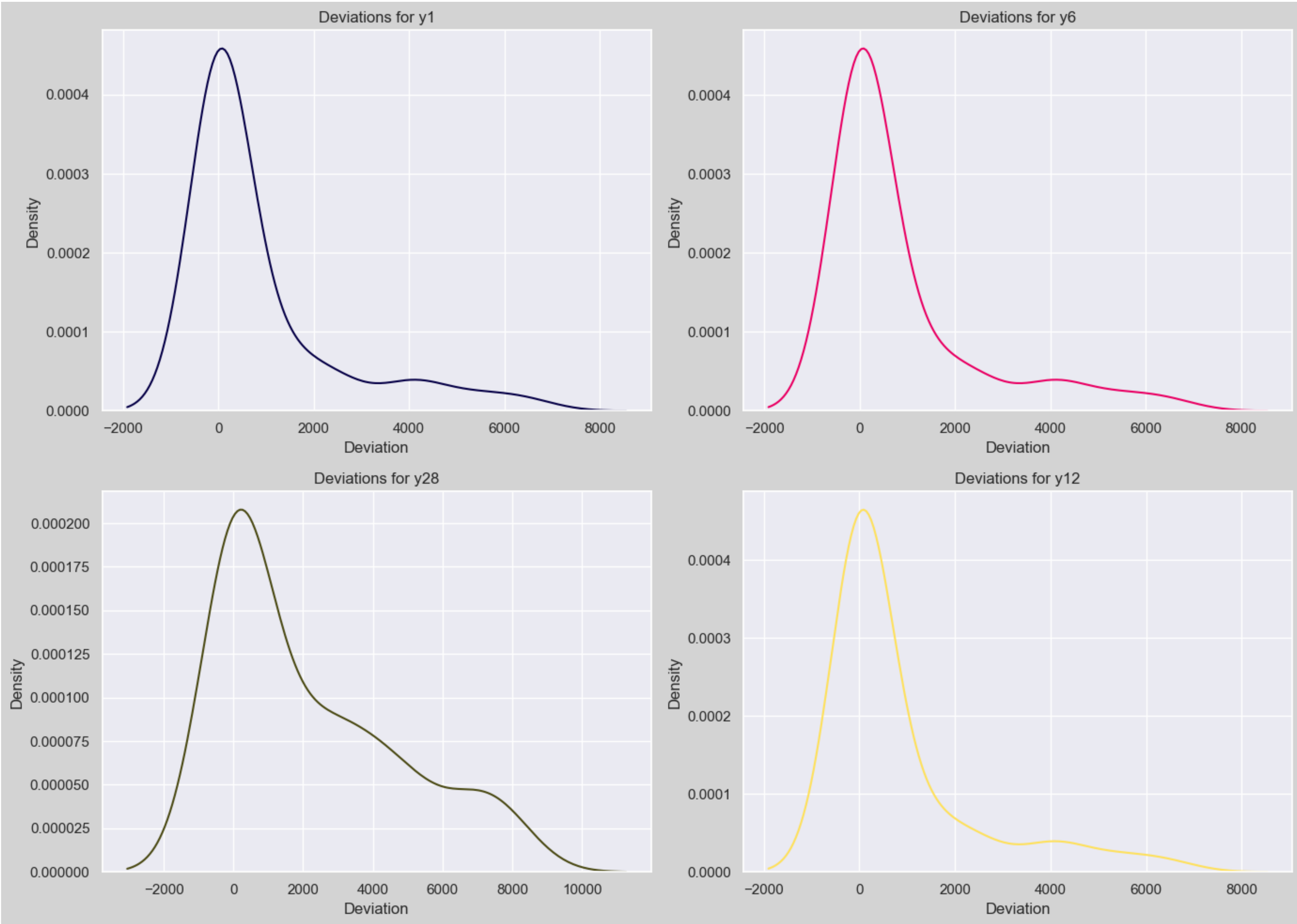
Deviations for Ideal Function: y1
Deviations for y1 (mapped)
Deviations for y1 (not mapped)
Deviations for Ideal Function: y6
Deviations for y6 (mapped)
Deviations for y6 (not mapped)
Deviations for Ideal Function: y28
Deviations for y28 (mapped)
Deviations for y28 (not mapped)
Deviations for Ideal Function: y12
Deviations for y12 (mapped)
Deviations for y12 (not mapped)
Deviation
X



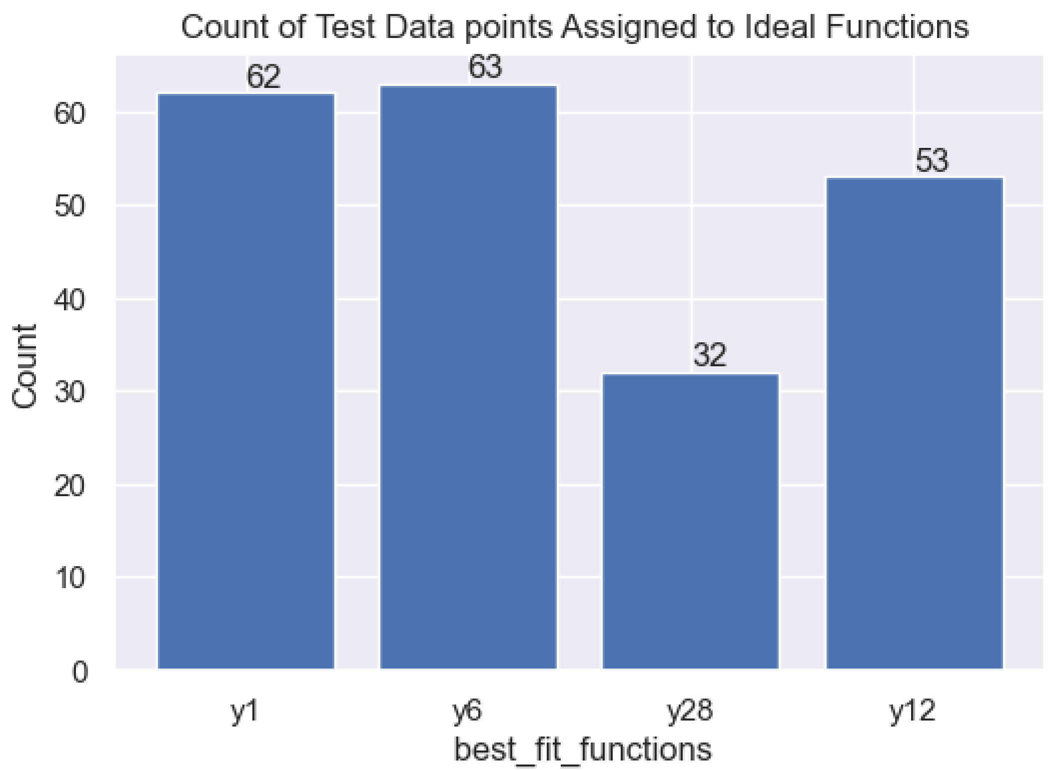
Scatter Plot for Test Dataset
Not mapped
Mapped to 1 ideal function
Mapped to 2 ideal functions
Mapped to 3 ideal functions
Mapped to 4 ideal functions
Y
X



Deviations for y1
Deviations for y6
Deviations for y28
Deviations for y12
Count
Deviation



Deviations for y1
Deviations for y6
Deviations for y28
Deviations for y12
Density
Deviation
0.0000
0.0001
0.0002
0.0003
0.0004
0.000000
0.000025
0.000050
0.000075
0.000100
0.000125
0.000150
0.000175
0.000200
−2000
0
2000
4000
6000
8000
10000



Checking Test Cases

```
In [ ]: import unittest
import pandas as pd
from unittest.mock import patch


class TestIdealFunctionFinder(unittest.TestCase):
    def setUp(self):
        # Set up test data
        self.data1 = {"x": [1.0, 2.0, 3.0], "y": [5.0, 6.0, 7.0]}
        self.data2 = {
            "x": [1.0, 2.0, 3.0],
            "y1": [7.0, 8.0, 9.0],
            "y2": [71.0, 81.0, 91.0],
        }
```

```python
    def test_calculate_best_fit_functions(self):
        df_train = pd.DataFrame(self.data1)
        df_ideal = pd.DataFrame(self.data2)

        ideal_function_finder = IdealFunctionFinder(df_train, df_ideal)
        result = ideal_function_finder.calculate_best_fit_functions()

        expected_best_fit_functions = ["y1"]
        expected_deviation_list = [12.0]

        best_fit_functions = result["Ideal Function"].tolist()
        deviation_list = result["Deviation"].tolist()

        self.assertEqual(best_fit_functions, expected_best_fit_functions)
        self.assertEqual(deviation_list, expected_deviation_list)

    def test_placeholder(self):
        # Placeholder test to ensure both tests are executed
        self.assertTrue(True)


if __name__ == "__main__":
    with patch("sys.argv", [""]):
        unittest.main(argv=[""], exit=False)
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.005s

OK
```