Minimizing Least-Square Residual Error of Linear Models in Python

Aaron Althauser
IU University of Applied Sciences
Programming with Python, DLMDSPWP01
May 8, 2021

Abstract

This paper outlines the structure and format of a Python program designed to map a dataset of various functions to their best-fitting ideal function based on a least squares method of regression analysis. The source code to the program is discussed in terms of approach taken and theory behind the decision-making process. Functions in each dataset are visualized using Python's Matplotlib, Pandas, and NumPy libraries, and presented in this paper to provide further explanation of the thought process in the design of the program.

Table of Contents

List of Figures

Abbreviations

RSS	Residual Sum of Squares
RMSE	Root Mean Squared Error
MRE	Max Residual Error
WLS	Weighted Least Squares
OOP	Object-Oriented Programming
CSV	Comma-Separated Values

Minimizing Least-Square Residual Error of Linear Models in Python

The Python programming language is known to be a powerful and versatile tool used for data science applications. Due the processing power it holds, to the modular format of the vast amount of written libraries publicly available, and the cumulative open-source development support it has received, Python is an easy first choice as a modeling and analysis tool for data scientists and engineers.

In the following article, Python will be will be utilized to match various functions to a training dataset using linear and polynomial regression techniques. After comparing a testing dataset to each training function, the test data will be then matched according to it's statistical significance in relation to the training function; and if the test data is within range of the calculated significance, it will be individually mapped to one of the ideal functions from the respective dataset. For each function, a model will be built based on the least squares algorithm and evaluated using custom built functions. In the case of a linear function, the Scipy library's Stats.linregress function will be used for parameter estimation, and in the nonlinear case, Numpy's Polynomial class will be used. For the purpose of focusing on minimizing error between the functions, the fine-tuning of certain model hyperparameters will be left out of the scope of this article. For example, the parameter of weight is set to a standard scale of $1/y_i$. The only parameter used to adjust the model will be the polynomial order of the fitted model.

While outlining the decision-making process, the article will explain the rationale behind the construction of the program by presenting data in the form of graphs and tables to fully encapsulate its functionality. Each module then will be explained in depth, highlighting the advantages and limitations both. The program structure will be thoroughly explained so that replication of the results and understanding of the architecture may be reproduced. Following will be a discussion of the results and the possible further avenues of research that, while beyond the scope of this article, could be used to further demonstrate the main argument.

The three main criterion used to assess the test functions' fit are Residual Sum of Squares (RSS), which measures the amount of variance between the the actual data and the fitted model, assuming the variance cannot be explained by independent variables (Draper & Smith, 1998; Rawlings et al., 1998); Root Mean Square Error (RMSE), the square root of the average between squared errors; and Max Residual Error (MRE), defined as the maximum error between the predicted and actual values, or the highest calculated residual (Pedregosa et al., 2011). These metrics were selected due to their relationship with least squares and their ability to explain goodness-of-fit of a calculated regression model—minimized RSS values point to a lesser amount of error observed between actual and predicted data points, where an RSS of zero signifies a perfect model fit (Draper & Smith, 1998). In using the least squares approach, one could use these defined metrics as a benchmark to assess the quality and

generalizability of their created models. In the scope of this article, rationale will be given as to how and why the chosen metrics minimize least squares of the specific functions in the most efficient way.

During the explanation of decision-making behind the program, it will be clear that test functions can be mapped to an ideal function according to how the error between models can be minimized. The most effectual way to minimize residual error between data values will be to choose the model of polynomial order with minimal MRE and RMSE. Not only will the best fitting model be at the point of the minimal MRE and RMSE, but where both metrics have the most radical increase or decrease. For the individual linear function of order one, polynomial regression will be restricted to the first order to prevent an inevitable overfitting of the data.

I. Program Structure

The most efficient way to manipulate and present the linear functions' data using Python will be through the use of classes and modules. Object-oriented programming (OOP) is both powerful and efficient using Python's *class* statement. Classes can be used to replicate real life structures and relationships that allow the program to function using sets of specifically designed functions through *inheritence* and *composition* (Lutz, 2009). Inheritence allows one object to inherit all the properties of another object; this unlimited extention of properties allows for great flexibility when connecting many different parts of a program. In the below source code, inheritence occurs between the Data and Database classes, in order to initialize a dataset as an object while simultaneously establishing a connection with the SQLite databse and maintaining the database's session—this kind of OOP style, which is quite powerful using Python, conveniently allows one to create a table in a database while using that same object to further manipulate its data. In terms of composition, using Python classes allows one to scale a larger program by composing all its separate pieces into one high-functioning unit. In the following section, it will be clear that each module has a specific job in the overarching composition of the program, by using the defined classes and their properties and attributes.

I.I. Modules

I.I.i. Main

The main entry point of the program resides in the Main module. By calling the main() function, the program starts cycling through its iterations. First, polynomial order is defined in a variable called "_n", as a dictionary of function names as its keys and a list of lists, containing different n-orders to compare, as its values. The way the program prioritizes n-order is such that the last order used will

represent the polynomial order of best fit model—the one that will be matched against the test data and stored in the database. After initializing the *_n* variable with the chosen n-orders to compare, the program uses Interface objects to control the manipulation, storage, and presentation options of the data.

I.I.ii. Interface

Using Interface objects to control the program's options allows for greater flexibility of each program cycle while simultaneously increasing its testability. The main idea of the Interface class is that one could call the class using various arguments and keyword arguments to control it as an interface, with the added advantage that each Interface object holds initialized Data, Model, and Graph objects as stored attributes that can be later recalled by the user.

I.I.iii. Database

To implement and establish a connection with an SQLite database, the Database class is called. Class inheritence of attributes from the Database class allows a connection to be established upon creation of a Data object, with its raw data is stored in the object's attributes.

The Database object has functions that start the initial reading and manipulation of the datasets: to read data from a CSV file, to convert the data into a Pandas DataFrame object, to insert the DataFrame object into the database, and to convert the DataFrame into HTML and PDF forms for saving as files. Data objects use these functions and initialized data to fit a regression model to the data. Using the fit_model() function of a Data object, the program has the functionality to create the objects necessary to further mainpulate and present the functions within the datasets.

I.I.iv. Model

A Model object has three main functions that typically work together in tandem to process data. First, the function find_ideal_function() will do as its title describes: it will iterate through each column and row of the ideal functions dataset, and will store the function that has the lowest MRE between it and the current training function, as an attribute of the Model object. Next, after all training functions have been matched with their resepctive ideal functions, match_ideal_functions() will be used to map each instance of the test dataset with one of the chosen ideal functions. Finally, map_test_with_train() is used to map each training function with the test data that was able to be matched according to the least square hyposthesis.

I.I.v. Graph

Graph objects are used to test the hypothesis and present the results of processed data. The methods that can be performed on Graph objects are for creating subplots of multiple functions, for

printing figures of best fit model versus its matched training function, for presenting a graph of the test function data plotted against training data, and for error plots that visualize the differences between error metrics as the order of the fit model increases.

I.II. Exception Handling

Different exception handling methods are employed in the program during runtime to ensure smooth operation of data processing. Error handling is used in instances where the code is moving to a different function and needs to be aware of sucession or failure of previous functions, or when variables need to be initialized when a certain data type is not passed. In the Database module, custom exceptions are written to catch specific events that are critical to the operation of the program: RegressionException is raised when a type of regression for the fit model is not supplied; TableNotCreatedException is caught but ignored when the respective SQLite database table is not created; and CSVError signals the absence of the specified CSV file in the program's directory. Additionally, standard Python exceptions such as AttributeError, KeyError, and ValueError are used to catch and raise errors in the program's logic.

II. Methodology

For the program to accurately match linear functions with closely-fitting ideal functions, a model will be created for each function, individually, using the least squares method. The ability to construct a reliable model that is able to predict data values accurately is dependent on its independent variables, the model parameters that are used as constant control values of the fit function (Sáez & Rittmann, 1992). Using the parameters chosen, each model will generate a batch of data points in the form of a Numpy one-dimensional vector array. Numpy's Polynomial.fit() function uses the least squares algorithm by default to build the parameters for the model (Oliphant, 2008), and will be applied to all functions except the exclusive linear function, which will use Scipy's Stats.linregress() funciton.

II.I. Initial Polynomial Order

Before too much manipulation of the data occurs in the first iteration of the program, the training data values should be displayed graphically to assess how to approach each one. Looking at Figure 1, we see that functions y1 and y2 show data that

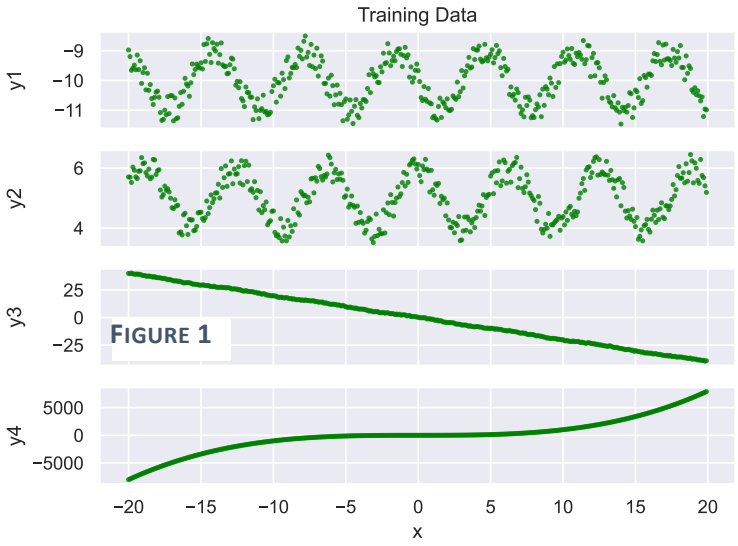


FIGURE 1

conforms to a sinusoidal function, while y3 is linear with a negative slope, and y4 is cubic. From this, one can infer that the polynomial order for both y1 and y2 is going to be higher than that of y3 and y4. Since it can be noticed from the graph that y3 is a linear function, the function will not be expected to have a polynomial order of greater than one, otherwise overfitting of the model data will inevitably occur. From this base knowledge we will not use function y3 to prove the hypothesis, but rather, to compare how error is minimized between function types. Similarly, we can guess that the n-order for y4, a cubic function, will minimize error most efficiently with an order of three. After testing the program during initial construction, the lowest observed RSS value for functions y1 and y2 occurred at n = 36, with an increasing trend in all values as n increased; therefore, 36 was chosen as the maximum value to test for function y1 and y2, as it will be proven that the best fit model will occur before this precise point.

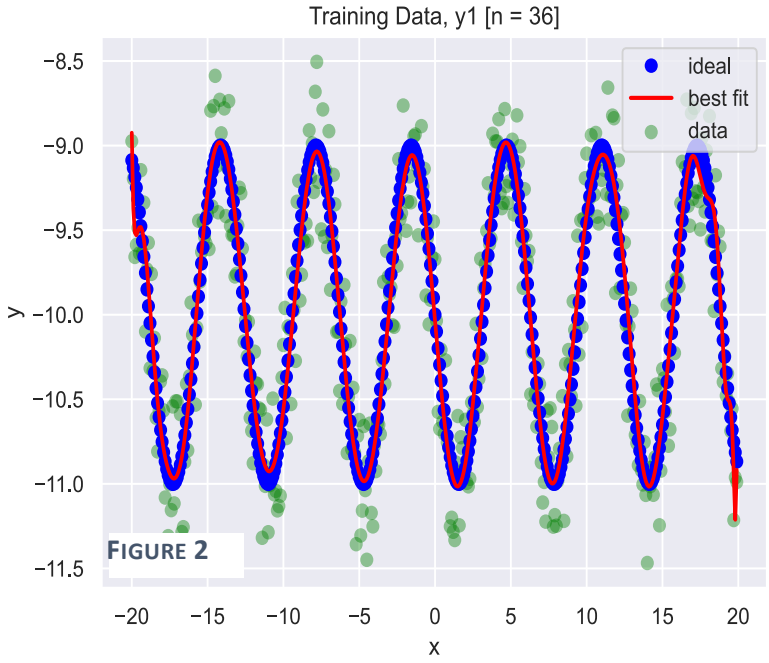


FIGURE 2

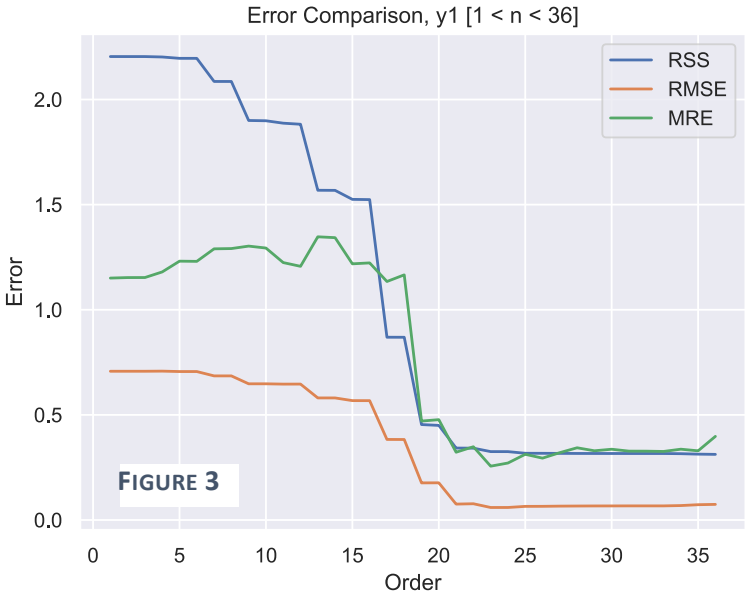


FIGURE 3

Y1. Figure 2 shows the model created for n = 36 plotted against the training data and its matched ideal function. Overfitting is observable on the tails of the graph, so we know this is not the ideal fit. If we look at the error plot for n = 36 (Figure 3), we see the trend for our three chosen metrics appears to be optimal around 19 < n < 24, with an upward spike at n = 36 for MRE.

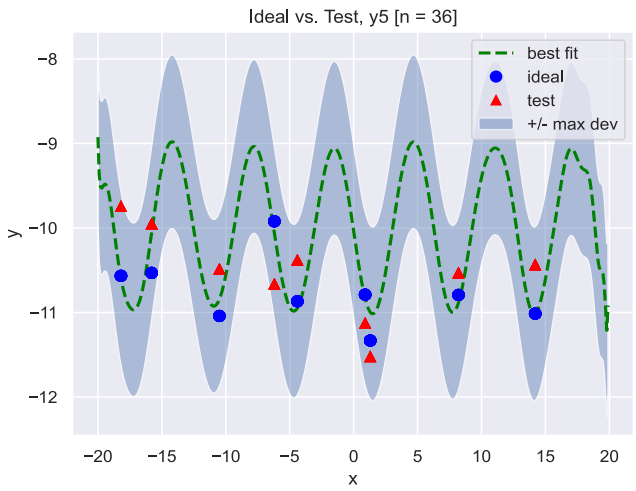


FIGURE 4

Order	RSS	RMSE	MRE
1	2.20429	0.70752	1.15070
2	2.20422	0.70750	1.15305
3	2.20422	0.70751	1.15326
4	2.20227	0.70808	1.17993
5	2.19580	0.70613	1.23111
6	2.19579	0.70611	1.23020
7	2.08586	0.68546	1.28974
8	2.08576	0.68546	1.29106
9	1.90024	0.64781	1.30276
10	1.89888	0.64781	1.29354
11	1.88740	0.64632	1.22441
12	1.88233	0.64652	1.20657
13	1.56834	0.58063	1.34702
14	1.56787	0.58059	1.34293
15	1.52504	0.56785	1.21846
16	1.52390	0.56766	1.22286
17	0.86936	0.38320	1.13439
18	0.86915	0.38290	1.16597
19	0.45396	0.17654	0.47054
20	0.44996	0.17694	0.47728
21	0.34223	0.07570	0.32237
22	0.34171	0.07733	0.34844
23	0.32534	0.05956	0.25606
24	0.32525	0.05961	0.27104
25	0.31742	0.06483	0.31251
26	0.31725	0.06507	0.29426
27	0.31688	0.06603	0.32041
28	0.31655	0.06652	0.34352
29	0.31640	0.06687	0.32926
30	0.31635	0.06696	0.33659
31	0.31625	0.06719	0.32760
32	0.31625	0.06719	0.32744
33	0.31625	0.06719	0.32606
34	0.31544	0.06872	0.33681
35	0.31327	0.07280	0.32904
36	0.31219	0.07432	0.39777

FIGURE 5

Furthermore, if we look at the trends of error metrics as polynomial order increases, in the form of a table of raw values, we will be able to catch significant trends in the data. Viewing Figure 4, it is noticeable that RSS for function y1 has dramatic decreases at n = 17, n = 19, and n =21; while RMSE and MRE reach their minimums at n = 23. This order will be used in a future iteration to further test the hypothesis that this point will signal the order of the most accurate model. Figure 5 shows the result of computing the test cases against the best fit model of n = 36 with the error threshold shown as well. The program was able to save a total of nine test case pairs that were inside the bounds of the maximum

deviation—but with a smaller n-order, the model should have a higher accuracy level by saving fewer test cases.

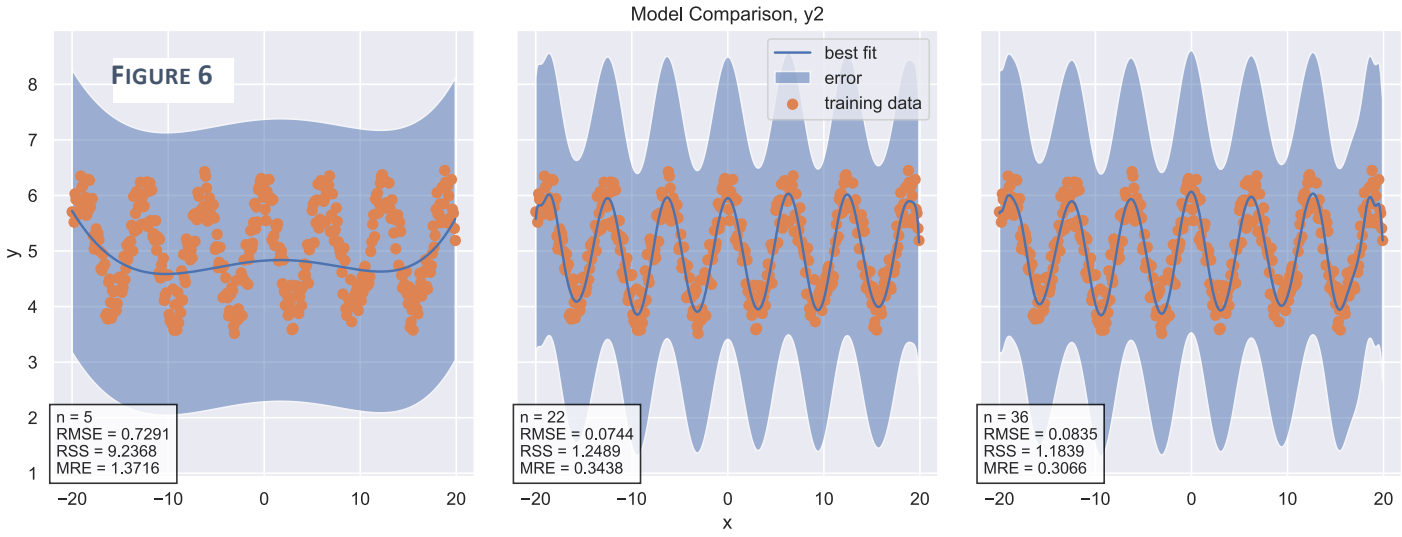


FIGURE 6

Y2. Being a similar function to y1, we can infer that polynomial order will need to be in the same range to create an accurate model. We can confirm this idea by looking at the graphs of the best fit

Order	RSS	RMSE	MRE
1	9.96486	0.75531	1.22445
2	9.69164	0.74577	1.34734
3	9.66575	0.74677	1.34669
4	9.24292	0.72921	1.38535
5	9.23682	0.72914	1.37158
6	8.83974	0.71236	1.34092
7	8.83854	0.71227	1.34113
8	8.79960	0.71122	1.30231
9	8.79767	0.71141	1.30144
10	8.36945	0.68639	1.51290
11	8.36894	0.68642	1.50892
12	7.31206	0.63231	1.42065
13	7.31187	0.63226	1.42076
14	7.11883	0.62058	1.56893
15	7.11346	0.62059	1.56803
16	5.44009	0.51918	1.37711
17	5.43999	0.51905	1.36598
18	2.55462	0.28776	0.94627
19	2.55166	0.28733	0.99996
20	1.43463	0.12477	0.65893
21	1.43253	0.12485	0.61055
22	1.24886	0.07445	0.34384
23	1.24419	0.07647	0.28410
24	1.22454	0.07166	0.23996

FIGURE 7

function for both n = 5 and n = 36 (Figure 6); at n = 5, the best fit line is highly generalizable, but not so accurate, whereas n = 36 leads to overfitting. On Figure 7, it is shown that RSS reaches its minimum at n = 36, but significantly decreases at 17, 18, 20, and 22; at n = 24, MRE and RMSE are at their minimum points, which should signify the order of best fit.

Y3. The linear function y3 will be tested at n = 1 to benchmark the initial statistics, but the Polynomial class from NumPy will not lead to the best fitting model, as there are other libraries in Python created for linear functions specifically, such as the Stats.linregress from SciPy (Virtanen et al, 2020).

Y4. The cubic training function will require a slightly different approach. The n-order of three is going to lead to the best fitting model, however, we should visualize this to verify exactly how the model changes with polynomial order. Figure 8 shows a comparison of the initial n-orders chosen for y4; n = 27 is overfitting the data, while the tails of n = 9 are trending in different directions and n = 3

FIGURE 8

seems to follow the data precisely. While viewing the table of errors for y4 (Figure 9) there is an unusual pattern seen at n = 3: RSS drops dramatically, while RMSE and MRE both severely increase. This could be due to the cubic function changing directions when x is positive.

Order	RSS	RMSE	MRE
1	390.24135	3.85026	6.52474
2	390.23229	4.37729	9.96185
3	13.05971	102.40778	288.68045
4	13.05629	102.69768	304.23988
5	12.51244	88.65774	389.84885
6	12.50938	88.79230	400.77566
7	12.00116	87.17384	479.35170
8	11.99855	87.31788	488.22798
9	11.52752	83.35437	498.43979

FIGURE 9

II.II. Final Iterations

Y1. For an n-order of 20, 18, and 17, we see that the model is still not precise (Figure 10); however, we can observe that as the order increases (at n = 20) the best fit line is getting slightly more accurate with less overfitting (Figure 11). We can confirm that the best fit model will be at the point of drastic decrease, at n = 23, where MRE and RMSE are both at their minimum values; Figure 12

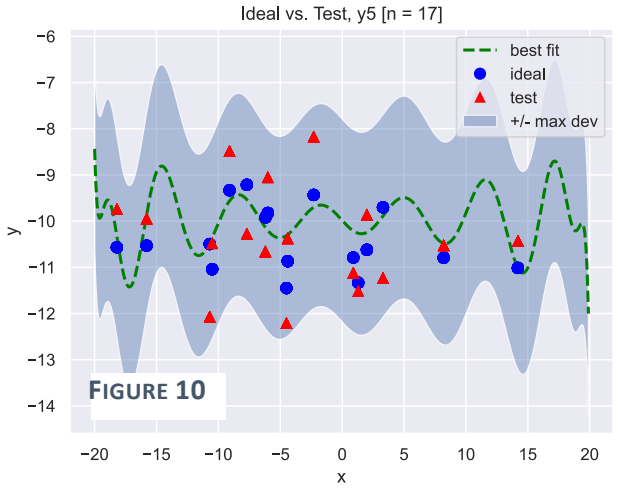


FIGURE 10

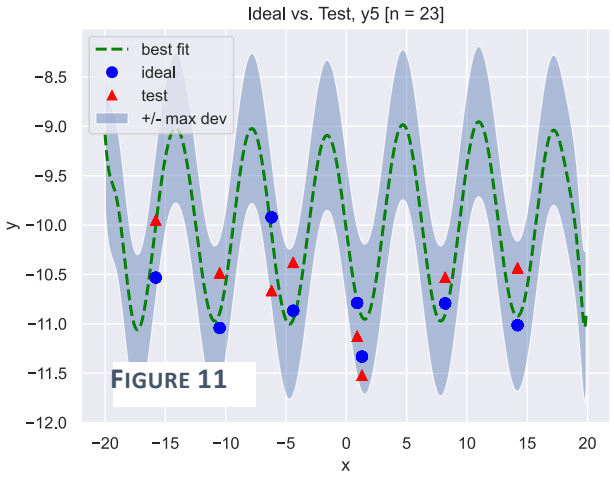


FIGURE 11

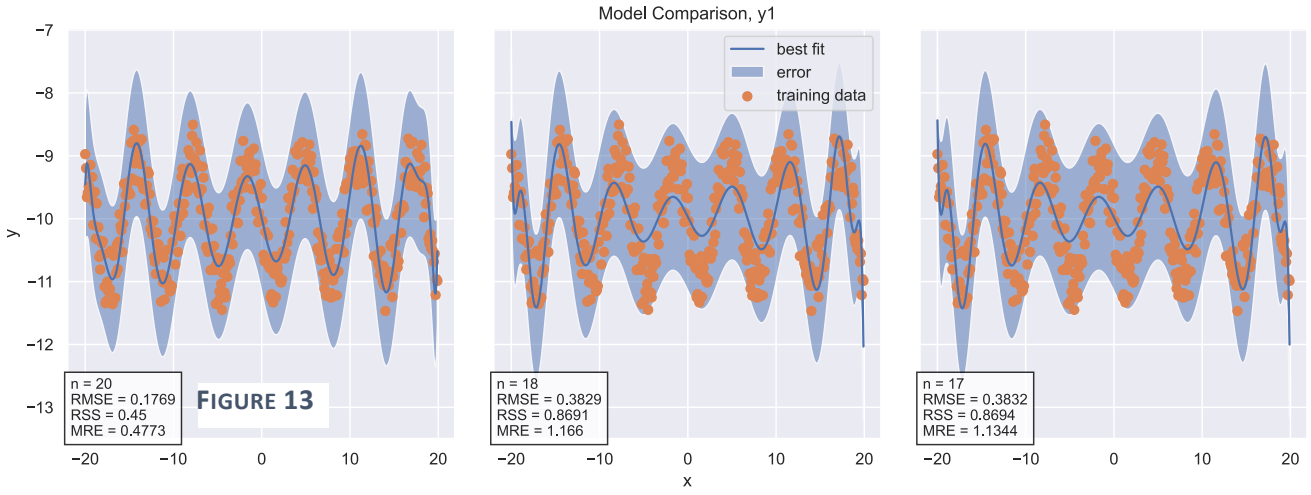


FIGURE 13

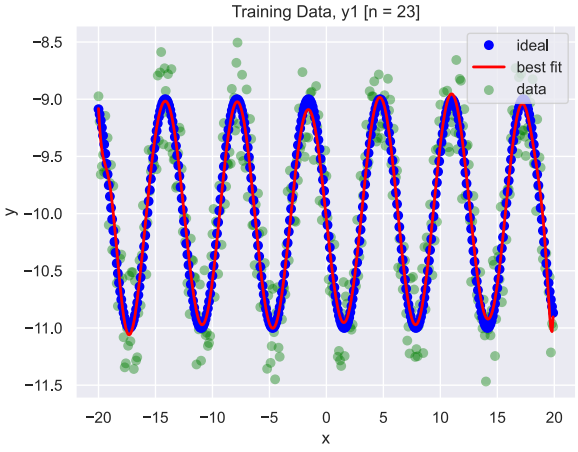
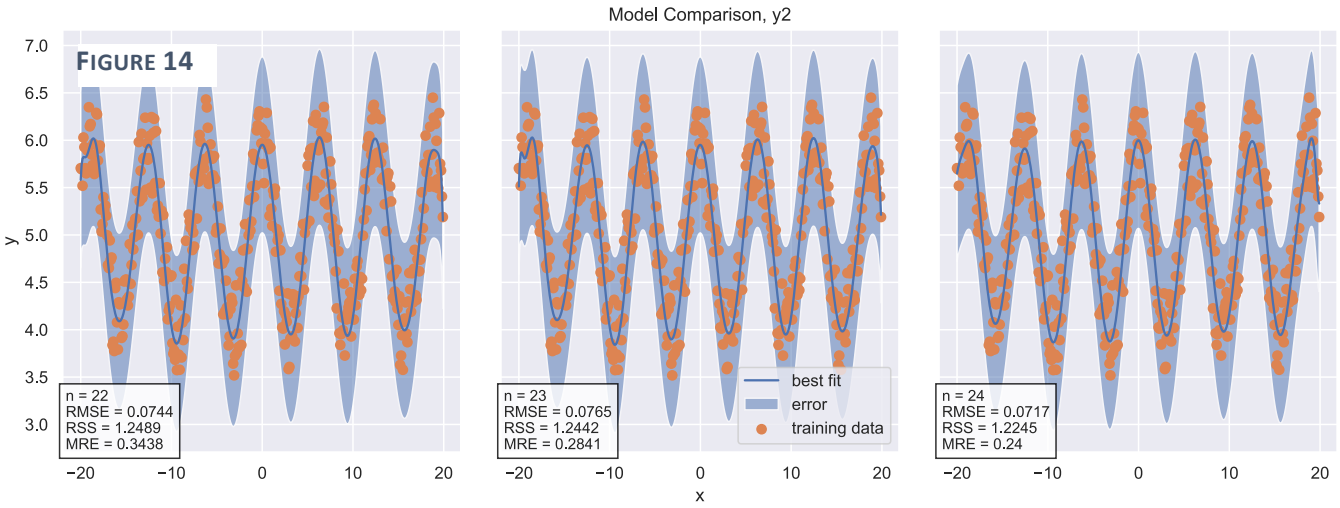


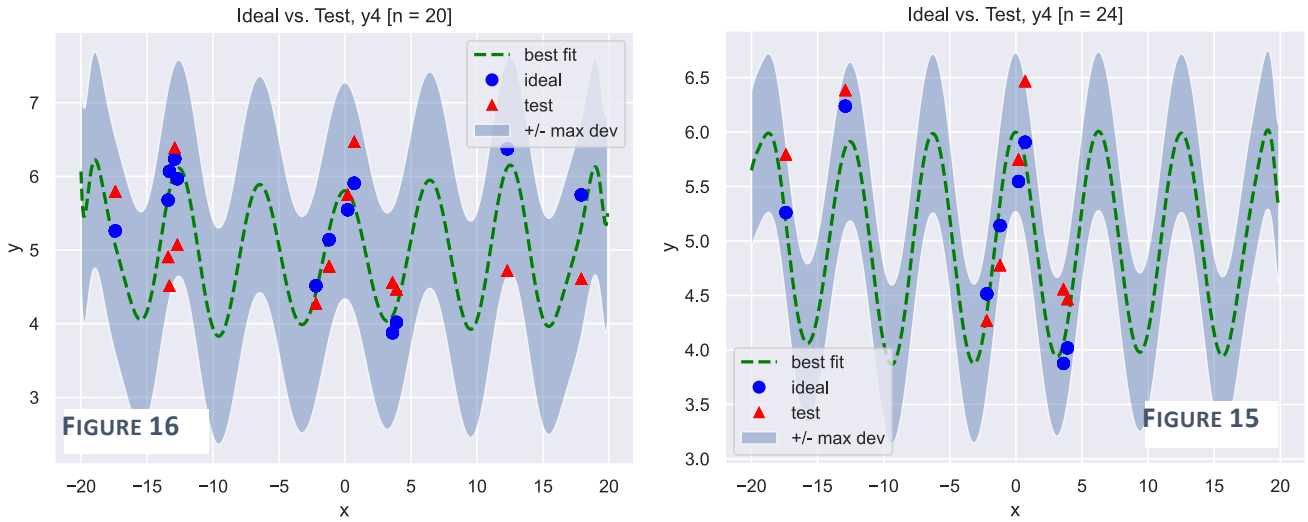
FIGURE 12

validates the reduction of overfitting with lower n-orders, while Figure 13 verifies the precision of the model by showing a reduced amount of test cases residing within the maximum deviation threshold.

Y2. The program effectively eliminated overfitting using an n-order of 24 (Figure 14).



By looking at the plotted test cases against the training and ideal functions for n = 20 and n = 24, it can be confirmed that the model with n-order 24 has a higher level of accuracy and thus saves a lower number of test cases into the database (Figure 15 & Figure 16); therefore, we will use the model with n-order in the final iteration of the program as the chosen model.



Y4. Although the MRE and RMSE are unusually high at n = 3, which does not strengthen the hypothesis held for functions y1 and y2, we can still verify that the model with n = 3 clearly provides the *most drastic change* between previous and concurrent models; which signifies the chosen model should be at n = 3.

III. Discussion

The program was able to prove that fitting a model with a specific polynomial order to a dataset of functions is most accurate when both the MRE and RMSE are minimized, and that the most accurate model will be represented at the specified order when the most drastic increase or decrease of error occurs. This was visible in functions y1 and y2, where the most accurate model was represented by the n-order that had minimal MRE and RMSE. In function y4, it was observed that the most dramatic change in values between models was a significant factor in choosing the best model. Through analysis of the values in a displayed table, and by representing the data visually on graphs, the program effectively showed how overfitting and error was reduced to a minimal point within the scope of this article.

III.i. Further Research

In future studies, one must decide which parameters to adjust in order to further minimize error of the models. For instance, Strutz (2011) identifies the importance of using Weighted Least Squares (WLS) to elimate the effect of outliers on fitting data; instead of all observations having the same importance, they should be relative to the data. In this program, we use a default weight, which does not efficiently reduce the effect of outliers in all functions of the dataset, but instead leaves the parameter fixed to keep consistency for all functions.

To increase functionality of the program, there could be less human involvement in deciding which polynomial order to use, and overall more autonominity in the program cycles. Further research could go into the best way to implement an interface that allows the user to run the program with minimal knowledge of Python. Another useful form of autonomy would be for the program to decide the best fitting model instead of the user pre-determining before the program cycles.

IV. Conclusion

The Python programming language is a powerful tool when used for data science applications, allowing the use of the OOP-style to analyze various statistics of datasets in a highly-customizeable way. Knowledge of libraries such as Matplotlib, Pandas, Numpy, and Sci-Kit Learn are essential skills for an engineer to have for the analysis of data and to perform regression techniques and gain insights about the data at hand. Within the scope of the article, using least squares as a method of minimizing error between models was highly effective on the given datasets, proving that the algorithm is useful in reducing overfitting and residual error of data. The article provided a baseline of where to further experiement with least squares using polynomial order and MRE/RMSE as the evaluated error metrics.

List of Appendices

APPENDIX A: MAIN.PY

```
"""
Aaron Althauser
IU International University of Applied Sciences
Data Science, M.Sc.
"""

from sqlalchemy import create_engine, MetaData
from interface import Interface
import pandas as pd

engine = create_engine("sqlite:///python_models.db", echo=True)
meta = MetaData()
pd.set_option('display.float_format', lambda x: '%.5f' % x)


def main():
    """

    Main entry point of the program.

    Uses keyword args passed to Interface() object to run
    program with specific options.

    """

    # Set polynomial order of the fit function
    # Last number in the list is the chosen model
    n = {
        'y1': [[5, 15, 36], [20, 18, 17], [19, 21, 23]],
        'y2': [[5, 15, 36], [18, 19, 20], [22, 23, 24]],
        'y4': [[3, 9, 27], [7, 8, 9], [1, 2, 3]]
    }

    while len(n['y1']) > 1:

        print(f'len(n) should be > 1: {len(n["y1"])}')

        _n = dict(y1=n['y1'].pop(0),
                  y2=n['y2'].pop(0),
                  y4=n['y4'].pop(0))
        print(f'_n: {_n}')
        df = Interface(_n=_n,
                       print_table=True,
                       plot_order_error=True)
        print(_n)
        df2 = Interface(_n=_n,
                        to_db=False,
                        run_complete=True,
                        compare_models={'m1': df.models_master_1,
                                        'm2': df.models_master_2,
                                        'm3': df.models_master_3})

        print(df2)

    """
    Final iteration
    """

    _n = {
        'y1': n['y1'][0],
        'y2': n['y2'][0],
        'y4': n['y4'][0]
    }

    df = Interface(_n=_n,
                   create_tables=False,
                   print_table=True,
                   plot_training_subplots=True,
                   plot_order_error=True)

    df2 = Interface(_n=_n,
                    to_db=False,
                    run_complete=True,
                    compare_models={'m1': df.models_master_1,
                                    'm2': df.models_master_2,
```

```
                                    'm3': df.models_master_3})

    print(df2)


if __name__ == "__main__":
    main()
```

APPENDIX B: DATABASE.PY

```
"""
Aaron Althauser
IU International University of Applied Sciences
Data Science, M.Sc.
"""

from sqlalchemy import Table, MetaData, create_engine, Column, Float, String, inspect
from sqlalchemy.exc import InvalidRequestError
import pandas as pd
from numpy.polynomial import Polynomial as P
from scipy import stats
from xhtml2pdf import pisa
from model import Model
from visualize import Graph

meta = MetaData()
engine = create_engine("sqlite:///python_models.db", echo=True)


class TableNotCreatedException(Exception):
    """Called if SQLite Table not created"""
    pass


class RegressionException(Exception):
    """Called if type of regression not specified"""
    pass


class CSVError(Exception):
    """Called if csv file not found"""
    pass


class DBTable:
    """Creates a new sqlalchemy.Table object for an SQLite database"""

    def __init__(self, *names, **kwargs):
        self.name, = names
        self.conn = engine
        self.df = pd.DataFrame()
        self.table = Table()

        _create = kwargs.get('_create', True)
        to_db = kwargs.get('to_db', True)

        if _create:
            try:
                # Create graphs for SQLite
                if 'train' in self.name:
                    self.table = Table(
                        "training_data", meta,
                        Column("x", Float, primary_key=True),
                        Column("y1", Float),
                        Column("y2", Float),
                        Column("y3", Float),
                        Column("y4", Float))
                if 'test_' in self.name:
                    self.table = Table(
                        "test_data", meta,
                        Column("x", Float, primary_key=True),
                        Column("y", Float),
                        Column("delta y", Float),
                        Column("ideal_function", Float))
                if 'ideal' in self.name:
```

```
                self.table = Table(
                    "ideal_functions", meta,
                    Column("x", Float, primary_key=True),
                    Column("y1", Float), Column("y2", Float), Column("y3", Float), Column("y4", Float),
                    Column("y5", Float), Column("y6", Float), Column("y7", Float), Column("y8", Float),
                    Column("y9", Float), Column("y10", Float), Column("y11", Float), Column("y12", Float),
                    Column("y13", Float), Column("y14", Float), Column("y15", Float), Column("y16", Float),
                    Column("y17", Float), Column("y18", Float), Column("y19", Float), Column("y20", Float),
                    Column("y21", Float), Column("y22", Float), Column("y23", Float), Column("y24", Float),
                    Column("y25", Float), Column("y26", Float), Column("y27", Float), Column("y28", Float),
                    Column("y29", Float), Column("y30", Float), Column("y31", Float), Column("y32", Float),
                    Column("y33", Float), Column("y34", Float), Column("y35", Float), Column("y36", Float),
                    Column("y37", Float), Column("y38", Float), Column("y39", Float), Column("y40", Float),
                    Column("y41", Float), Column("y42", Float), Column("y43", Float), Column("y44", Float),
                    Column("y45", Float), Column("y46", Float), Column("y47", Float), Column("y48", Float),
                    Column("y49", Float), Column("y50", Float))
            except InvalidRequestError:
                print(f'({self.name}) table already created')
                pass

            if to_db:
                self.csv_to_db()
            else:
                self.csv_to_df()

    def add_test_table(self, name):
        try:
            self.table = Table(
                name, meta,
                Column('a', String), Column('b', String), Column('c', String))
            self.table.create(self.conn, checkfirst=True)
            print('table added')
        except InvalidRequestError as ire:
            print(ire)

    def drop_test_table(self):
        self.table.drop(self.conn, checkfirst=True)
        inspector = inspect(self.conn)
        print('unittest' in inspector.get_table_names())

    def csv_to_db(self):
        """Reads data from CSV file, converts to Pandas Dataframe,
        inserts Dataframe into the database"""
        self.csv_to_df()
        self.df_to_db(pd.DataFrame())
        meta.create_all(engine)
        return True

    def csv_to_df(self):
        """Reads data from CSV file and converts into Pandas Dataframe"""
        if 'train' in self.name:
            self.df = pd.read_csv("./datasets/train.csv")
            return self.df
        elif 'test_' in self.name:
            self.df = pd.read_csv("./datasets/test.csv")
            return self.df
        elif 'ideal' in self.name:
            self.df = pd.read_csv("./datasets/ideal.csv")
            return self.df
        elif 'unittest' in self.name:
            self.df = pd.read_csv(f'./tests/{self.name}.csv')
        else:
            raise CSVError("csv file not found")

    def df_to_db(self, df):
        """Inserts Dataframe object into the database"""
        if df.size == 0:
            print(f'******* size of df is ({df.size})')
            self.df.to_sql(self.name, con=engine, if_exists='replace')
            meta.create_all(engine)
        else:
            df.to_sql(self.name, con=engine, if_exists='replace')
            meta.create_all(engine)

    def df_to_html(self, df, name='table', output_dir='./tables'):
        """Writes HTML file from dataframe"""
        new_file = df.to_html(justify='center', index=False)
        new_file = new_file.replace('<table border="1" class="dataframe">',
```

```
                                    '<table style="border-top:2px solid black;'
                                    'border-bottom:1px solid black;font-family:Arial;" '
                                    'class="dataframe">')
            new_file = new_file.replace('<tr>',
                                    '<tr style="text-align:center;border:0;padding-top:2px;">')
            new_file = new_file.replace('<th>',
                                    '<th style="width:120px;'
                                    'padding-top:3px;margin-bottom:3px;font-size:11pt;">')
            new_file = new_file.replace('<tbody>',
                                    '<tbody style="padding-top:2px;font-size:10pt;">')

            # convert to .pdf
            with open(f'{name}.html', 'w') as f:
                self.html_to_pdf(new_file, f'{output_dir}/{name}.pdf')

    def html_to_pdf(self, source_html, output_filename):
        """Writes .pdf file from HTML"""
        with open(output_filename, "w+b") as f:
            pisa_status = pisa.CreatePDF(source_html, dest=f)
        return pisa_status.err


class Data(DBTable):
    """Inherits attributes from DBTable class.
    Returns best fit model as a Model() object."""

    def __init__(self, *names, **kwargs):
        super().__init__(*names, **kwargs)

    def is_empty(self):
        """Helper function to check if df of Data obj is empty"""
        if self.df.size == 0:
            return True
        else:
            return False

    def fit_model(self, *args, **kwargs):
        """Fit a regression model with training data function"""

        col = args[0]
        _ideal = args[1]
        r_type = args[2]

        order = kwargs.get('order', 1)
        subplot = kwargs.get('subplot', False)
        print_table = kwargs.get('print_table', False)
        table_name = kwargs.get('table_name', '')

        col_name = f'y{col}'
        subplot_array, _n, _rss, _rmse, _max_e, _var = [], [], [], [], [], []
        global model_df

        if r_type == "linear":
            x = self.df['x'].values
            lr = stats.linregress(self.df['x'], self.df[col_name])
            fun = lr.slope * x + lr.intercept
            model = Model(x, fun, col)
            try:
                # match ideal function if ideal Data object is passed
                if not _ideal.is_empty():
                    model.find_ideal_function(_ideal)
                    _rmse.append(model.rmse)
                    _max_e.append(model.max_dev)
                if print_table:
                    model_df = pd.DataFrame()
                    model_df['RMSE'] = [round(float(i), 5) for i in _rmse]
                    model_df['MRE'] = [round(float(i), 5) for i in _max_e]
                    if not table_name:
                        self.df_to_html(model_df, f'{col_name}_linear')
                    else:
                        out = table_name.rsplit('/', 1)
                        self.df_to_html(model_df, f'{out[1]}', output_dir=out[0])
                return model, model_df
            except AttributeError:
                # raised if _ideal df is empty
                raise Exception("ideal df empty or None")

        if r_type == "poly.fit":
```

```
            model = Model([], [], col)
            rss_max = 1000
            # Iterates through orders and returns fit with
            # minimum residual error, with weight=1/y
            for i in range(1, order + 1):
                weight = 1 / self.df[col_name]
                fn = P.fit(self.df['x'], self.df[col_name], i, full=True, w=weight)
                coeff, det = fn
                fn_x, fn_y = coeff.linspace(n=400)
                model = Model(fn_x, fn_y, col, rss=det[0], order=i)
                # match ideal function if ideal Data object is passed
                if not _ideal.is_empty():
                    model.find_ideal_function(_ideal)
                    _n.append(i)
                    _rss.append(model.rss)
                    _rmse.append(model.rmse)
                    _max_e.append(model.max_dev)
                if det[0] < rss_max:
                    subplot_array.append(model)
            if print_table:
                model_df = pd.DataFrame()
                model_df['Order'] = _n
                model_df['RSS'] = [i[0].round(5) for i in _rss]
                model_df['RMSE'] = [i.round(5) for i in _rmse]
                model_df['MRE'] = [i.round(5) for i in _max_e]
                self.df_to_html(model_df, f'{col_name}_n-{order}')
                return model, model_df
            elif not subplot or len(subplot_array) <= 1:
                return model
            else:
                subplot_graph = Graph('Polynomial order and weighted NLR', df=self.df)
                subplot_graph.make_subplots(subplot_array)
                return model
        else:
            raise RegressionException("You must provide a valid type "
                                      "of regression via keyword arg.")
```

APPENDIX C: MODEL.PY

```
"""
Aaron Althauser
IU International University of Applied Sciences
Data Science, M.Sc.
"""

from itertools import starmap
from sklearn.metrics import mean_squared_error, max_error
import pandas as pd
from visualize import Graph
import numpy as np


class Model:
    """
    Fit model to training data and match with an ideal function
    """

    def __init__(self, x, y, col, df=pd.DataFrame(), rss=0, order=1):
        self.x = x
        self.y = y
        self.col = col
        self.rmse = 1000
        self.test_rmse = 0
        self.max_dev = 1000
        self.__max_dev = 1000
        self.rss = rss
        self.order = order
        self.__bf = ""
        self.ideal_col = ""
        self.ideal_col_array = []
        if not df.size == 0:
```

```
        self.df = df

    def get_max_dev(self):
        return self.__max_dev

    def set_max_dev(self, val):
        self.__max_dev = val
        self.max_dev = self.__max_dev

    def get_best_fit(self):
        return self.__bf

    def set_best_fit(self, val):
        self.__bf = val

    def find_ideal_function(self, data):
        """Calculate the deviance between actual and predicted arrays"""

        try:
            for i in range(1, 50):
                col_name = f'y{i}'
                # Compute the RMSE and max error between
                # training function and each ideal function
                # The smallest MRE signals the closest match between functions
                rms_error = mean_squared_error(self.y, data.df[col_name], squared=False)
                if rms_error < self.rmse:
                    self.rmse = rms_error
                max_r = max_error(self.y, data.df[col_name])
                if max_r < self.max_dev:
                    self.max_dev = max_r
                    self.ideal_col = col_name
                    self.ideal_col_array = data.df[col_name]
        except ValueError:
            print("polynomial needs work")

    def match_ideal_functions(self, *args, **kwargs):
        """Match test data with its associated ideal function"""

        dev_list, funs_list = [], []
        idx = 0
        ideal_funs, train_df, models_ = args[0], args[1], args[2]
        map_train = kwargs.get('map_train', True)

        for row in self.df.itertuples(index=False):
            column, diff = "", 100000
            try:
                for c in ideal_funs.keys():
                    rmse = mean_squared_error([row.y], [ideal_funs.at[row.x, c]],
                                              squared=False)
                    if rmse < diff:
                        diff = rmse
                        column = c
            except KeyError as e:
                print(e)
                continue
            except ValueError as v:
                print(v)
                continue
            dev_list.append(diff)
            funs_list.append(column)

        # Append matched ideal functions to each test case
        self.df['delta_y'] = [float(i) for i in dev_list]
        self.df['ideal_fun'] = funs_list
        self.df.sort_values(by=['ideal_fun', 'x'], inplace=True)

        # Check if each ideal function is within the threshold of error
        # Turn value into NaN if not
```

```
        for row in self.df.itertuples(index=False):
            for i, j in models_.items():
                if row.ideal_fun == j.ideal_col:
                    _factor = np.sqrt(j.max_dev) + j.max_dev
                    if row.delta_y > _factor:
                        self.df.iat[idx, 3] = 'n/a'
            idx += 1

        if map_train:
            return self.map_test_with_train(train_df)
        else:
            return self.df

    def map_test_with_train(self, train_df):
        """Maps new test data with training function, to be plotted"""

        test_copy = self.df
        test_copy = test_copy.reset_index(drop=True)
        test_map = {}

        for row in test_copy.itertuples():
            # check training data to see which fun the ideal fun belongs to
            train_row = train_df[train_df['x'] == row.x]
            try:
                for i in range(1, 5):
                    t = train_row[train_row[f'y{i}_if'].isin([row.ideal_fun])]
                    y_col = f'y{i}'
                    if not len(t.index) == 0:
                        nt = train_df
                        nt_ = nt[['x', y_col]]
                        new_train = nt_.set_index('x')
                        # match test(x, y) values with train(x, y) values
                        # to be plotted and tested
                        n = test_copy.loc[test_copy['ideal_fun'] == row.ideal_fun]
                        for r in n.itertuples():
                            ideal, train = row.ideal_fun, y_col
                            if train not in test_map.keys():
                                test_map[train] = {'ideal_fun': ideal, train: [], ideal: []}
                            else:
                                test_map[train][train].append((r.x, new_train.at[r.x, y_col]))
                                test_map[train][ideal].append((r.x, r.y))
            except KeyError:
                break

        return self.compare_errors(test_map, train_df)

    def compare_errors(self, test_map, train_df):
        """Identify errors and plot processed data"""

        models = {}

        for k, v in test_map.items():
            ideal_ = test_map[k]['ideal_fun']
            test_arr = test_map[k][k]
            ideal_arr = test_map[k][ideal_]
            test_axes = list(starmap(lambda x, y: [x, y], test_arr))
            ideal_axes = list(starmap(lambda x, y: [x, y], ideal_arr))
            t_x = [ax[0] for ax in test_axes]
            t_y = [ax[1] for ax in test_axes]
            id_x = [ax[0] for ax in ideal_axes]
            id_y = [ax[1] for ax in ideal_axes]

            # Graph test vs ideal funs
            t_df = pd.DataFrame()
            t_df['x'] = t_x
            t_df['y'] = t_y
            plot_ = Graph(f'Ideal vs. Test', df=t_df)
            model = Model(id_x, id_y, ideal_[1:])
```

```
            # identify max error of training function
            c = f'{k}_max_err'
            tr_err = round(train_df[c][0], 6)

            # match ideal function with training function
            bf = f'{k}_best_fit'
            fit = train_df[bf][0]

            model.set_best_fit(fit)
            model.set_max_dev(tr_err)
            models[k] = model.__bf

            # plot graph
            plot_.plot_model(model, plt_type='test_vs_ideal', fit_model=model.__bf)

        return self.df, models
```

APPENDIX D: VISUALIZE.PY

```
"""
Aaron Althauser
IU International University of Applied Sciences
Data Science, M.Sc.
"""

from matplotlib import pyplot as plt
import pandas as pd
import seaborn as sb
import numpy as np


class PlotTypeException(Exception):
    pass


class Graph:
    """
    Graph object to plot models and data
    """

    def __init__(self, title, **kwargs):
        self.title = title
        self.plt = plt
        self.data = kwargs.get('df', pd.DataFrame())
        self.x = self.data['x']

    def make_subplots(self, title, **kwargs):
        """
        Plot multiple subplots at once

        :param title: Name of graph
        :keyword subdir: Str(), path to subdirectory
        :keyword models: Dict(), fitted models
        """

        sb.set_theme()  # use seaborn default layout
        idx = 0

        subdir = kwargs.get('subdir', './graphs')

        if 'train' in title.lower():
            # Plot training data
            fig, axs = self.plt.subplots(4, 1, sharex=True)
            for i in self.data.keys():
```

```
            if i != "x":
                temp = self.data[i]
                y = [float(i) for i in temp.to_list()]
                y_label = f'y{int(idx) + 1}'
                axs[idx].plot(self.x, y,
                              color='green',
                              alpha=0.8,
                              marker='o',
                              linewidth=0,
                              markersize=2)
                axs[idx].set(ylabel=y_label)
                if idx == 0:
                    axs[idx].set(title=self.title)
                idx += 1

        fig.align_ylabels()
        axs[idx - 1].set(xlabel='x')

        self.plt.tight_layout()
        self.plt.savefig(f'{subdir}/{title}.pdf', bbox_inches='tight')
        self.plt.show(block=False)
        self.plt.pause(1)
        self.plt.close()

        return True

    if 'model' in title.lower():
        # Plot model comparisons
        global model_1, model_2, model_3, factor, _m2, _m3, stats, \
            _m_rmse, _m_rss, _m_max, \
            _m2_rmse, _m2_rss, _m2_max, \
            _m3_rmse, _m3_rss, _m3_max, \
            stats_1, stats_2, stats_3

        if 'models' in kwargs.keys():
            model_1 = kwargs['models']['m1']['m1']
            model_2 = kwargs['models']['m2']['m2']
            model_3 = kwargs['models']['m3']['m3']

        for fn, _m in model_1.items():

            box = {'facecolor': 'white', 'alpha': 0.8, 'edgecolor': 'black'}

            fig, axs = self.plt.subplots(1, 3, sharey=True, figsize=(15, 5))

            try:
                _m2, _m3 = model_2[fn], model_3[fn]
                _m_rmse, _m_rss, _m_max = round(_m.rmse, 4), round(_m.rss[0], 4), round(_m.max_dev, 4)
                _m2_rmse, _m2_rss, _m2_max = round(_m2.rmse, 4), round(_m2.rss[0], 4), round(_m2.max_dev, 4)
                _m3_rmse, _m3_rss, _m3_max = round(_m3.rmse, 4), round(_m3.rss[0], 4), round(_m3.max_dev, 4)
                factor = np.sqrt(_m.max_dev) + _m.max_dev

                stats_1 = f'n = {_m.order}\n' \
                          f'RMSE = {_m_rmse}\n' \
                          f'RSS = {_m_rss}\n' \
                          f'MRE = {_m_max}'

                stats_2 = f'n = {_m2.order}\n' \
                          f'RMSE = {_m2_rmse}\n' \
                          f'RSS = {_m2_rss}\n' \
                          f'MRE = {_m2_max}'

                stats_3 = f'n = {_m3.order}\n' \
                          f'RMSE = {_m3_rmse}\n' \
                          f'RSS = {_m3_rss}\n' \
                          f'MRE = {_m3_max}'

            except TypeError:
```

```
                '''Occurs when type is linear'''

                _m2, _m3 = model_2[fn], model_3[fn]
                _m_rmse, _m_max = round(_m.rmse, 4), round(_m.max_dev, 4)
                _m2_rmse, _m2_max = round(_m2.rmse, 4), round(_m2.max_dev, 4)
                _m3_rmse, _m3_max = round(_m3.rmse, 4), round(_m3.max_dev, 4)
                factor = np.sqrt(_m.max_dev) + _m.max_dev

                stats_1 = f'n = {_m.order}\n' \
                          f'RMSE = {_m_rmse}\n' \
                          f'MRE = {_m_max}'

                stats_2 = f'n = {_m.order}\n' \
                          f'RMSE = {_m_rmse}\n' \
                          f'MRE = {_m_max}'

                stats_3 = f'n = {_m.order}\n' \
                          f'RMSE = {_m_rmse}\n' \
                          f'MRE = {_m_max}'

            finally:

                axs[0].fill_between(_m.x, _m.y - factor, _m.y + factor, alpha=0.5)
                axs[0].plot(_m.x, _m.y)
                axs[0].scatter(self.data['x'], self.data[fn])
                axs[0].annotate(stats_1, xy=(2, 2), xycoords='axes points', bbox=box, fontsize=10)
                axs[0].set_ylabel('y')

                axs[1].fill_between(_m2.x, _m2.y - factor, _m2.y + factor, alpha=0.5, label='error')
                axs[1].plot(_m2.x, _m2.y, label='best fit')
                axs[1].scatter(self.data['x'], self.data[fn], label='training data')
                axs[1].annotate(stats_2, xy=(2, 2), xycoords='axes points', bbox=box, fontsize=10)
                axs[1].legend()
                axs[1].set_xlabel('x')
                axs[1].set(title=f'{title}, {fn}')

                axs[2].fill_between(_m3.x, _m3.y - factor, _m3.y + factor, alpha=0.5)
                axs[2].plot(_m3.x, _m3.y)
                axs[2].scatter(self.data['x'], self.data[fn])
                axs[2].annotate(stats_3, xy=(2, 2), xycoords='axes points', bbox=box, fontsize=10)

                fig.subplots_adjust(wspace=0.1)
                self.plt.savefig(f'{subdir}/{fn}_{title}_{_m.order}{_m2.order}{_m3.order}.pdf', bbox_inches='tight')
                self.plt.show(block=False)
                self.plt.pause(1)
                self.plt.close()

        return True

def plot_model(self, model_, **kwargs):
    """

    Plot models and save matplotlib figures

    :param model_: Model(), model to be plotted
    :keyword plt_type: Str(), type of plot to make
    ['best fit', 'test vs ideal', 'error']
    :keyword with_rmse: Bool() = False
    :keyword fit_model: Model(), used to compare with test data
    :keyword subdir: Str(), path to subdirectory
    :keyword _col: Str(), col name
    """

    sb.set_theme()  # use seaborn default theme
    global col_name

    plt_type = kwargs.get('plt_type', None)
    with_rmse = kwargs.get('with_rmse', False)
```

```
fit_model = kwargs.get('fit_model', None)
subdir = kwargs.get('subdir', './graphs')
_col = kwargs.get('_col')

try:
    col_name = f'y{model_.col}'
    if fit_model:
        self.plt.title(f'{self.title}, {col_name} [n = {fit_model.order}]')
    else:
        self.plt.title(f'{self.title}, {col_name} [n = {model_.order}]')
except AttributeError:
    pass  # if the model_ is for error plot

try:
    if 'best fit' in plt_type:

        y = self.data[col_name]

        self.plt.scatter(self.x, y,
                         label="data", alpha=0.4, color='green')

        if with_rmse:
            self.plt.plot(self.x, model_.ideal_col_array, "o",
                          linewidth=0, label="ideal", color='blue')

        self.plt.plot(model_.x, model_.y, "-",
                      label="best fit", linewidth=2, color='red')
        self.plt.xlabel('x')
        self.plt.ylabel('y')
        self.plt.legend()
        self.plt.savefig(f'{subdir}/{col_name}_n-{model_.order}_bestfit.pdf',
                         bbox_inches='tight')
        self.plt.show(block=False)
        self.plt.pause(1)
        self.plt.close()

        return True

    if 'test' in plt_type or 'vs' in plt_type or 'ideal' in plt_type:
        factor = np.sqrt(fit_model.max_dev) + fit_model.max_dev

        self.plt.fill_between(fit_model.x, fit_model.y - factor, fit_model.y + factor, alpha=0.4,
                              label="+/- max dev")
        self.plt.plot(fit_model.x, fit_model.y, "--", color="green", linewidth=2, markersize=0,
                      label="best fit")
        self.plt.plot(self.x, self.data['y'], "o", color="blue", linewidth=0, markersize=7, label="ideal")
        self.plt.plot(model_.x, model_.y, marker="^", color="red", linewidth=0, markersize=6,
                      label="test")
        self.plt.xlabel('x')
        self.plt.ylabel('y')
        self.plt.legend()
        self.plt.savefig(f'{subdir}/y{fit_model.col}_n-{fit_model.order}_test-vs-ideal.pdf',
                         bbox_inches='tight')
        self.plt.show(block=False)
        self.plt.pause(1)
        self.plt.close()

        return True

    if 'error' in plt_type:
        try:
            if model_['Order'].size > 1:  # no need to plot order of 1

                self.plt.plot(model_['Order'], model_['RSS'], label='RSS')
                self.plt.plot(model_['Order'], model_['RMSE'], label='RMSE')
                self.plt.plot(model_['Order'], model_['MRE'], label='MRE')

                self.plt.title(f'Error Comparison, {_col} '
```

```
                                f'[{min(model_["Order"])} < n < {max(model_["Order"])}]')
                    self.plt.xlabel('Order')
                    self.plt.ylabel('Error')
                    self.plt.legend()
                    self.plt.savefig(f'{subdir}/{_col}_n-{max(model_["Order"])}_errorplot.pdf')
                    self.plt.show(block=False)
                    self.plt.pause(1)
                    self.plt.close()

            except KeyError:  # if plot is linear
                pass

    except TypeError:
        raise PlotTypeException("you must provide a plot type")
```

APPENDIX E: INTERFACE.PY

```
"""
Aaron Althauser
IU International University of Applied Sciences
Data Science, M.Sc.
"""

from database import Data
from model import Model
from visualize import Graph
import pandas as pd


class NSizeError(Exception):
    pass


def check_n_size(n):
    """Helper function to ensure _n is correct size"""
    size = [len(i) for i in n.values()]
    if len(n) == 3 \
            and len(n) == size[0] \
            and len(n) == size[1] \
            and len(n) == size[2]:
        return True
    else:
        print(f'_n size ({size}) does not match')
        return False


class Interface:
    """
    Program runner from main.py module
    """

    def __init__(self, **kwargs):
        """
        :keyword map_train: run Model functions entirely and
        plot matched ideal vs test data
        :keyword to_db: create SQLite db table for created Data obj
        :keyword plt_type: which type of fit algorithm to run, 'linear' or 'best fit'
        :keyword with_rmse: include rmse values in graph
        :keyword print_table: save stats from model comparisons as a .pdf table
        :keyword plot: plot data and save
        :keyword plot_training_subplots: display and save training data as subplots
        :keyword compare_models: shortcut to just plot comparison of fitted models,
        values is dict of fitted model dicts
        """

        to_db = kwargs.get('to_db', True)
        _create = kwargs.get('create_tables', True)

        self.train = Data('training_data', to_db=to_db)
        self.ideal = Data('ideal_functions', to_db=to_db)
        self.test = Data('test_data', _create=_create)
```

```
        self.train_master = self.train.csv_to_df()
        self.train_graph = Graph("Training Data", df=self.train.csv_to_df())
        self.ideal_fn_dict = {'x': self.train.df['x']}

        self._n = kwargs.get('_n', {})
        self.plt_type = kwargs.get('plt_type', 'best fit')
        self.with_rmse = kwargs.get('with_rmse', True)
        self.print_table = kwargs.get('print_table', True)
        self.plot = kwargs.get('plot', True)
        self.plot_order_error = kwargs.get('plot_order_error', False)

        map_train = kwargs.get('map_train', True)
        continue_matching = kwargs.get('continue_matching', True)

        self.models = dict()
        self.models_master_1 = dict()
        self.models_master_2 = dict()
        self.models_master_3 = dict()
        self.result = tuple()

        global model

        if 'compare_models' in kwargs.keys():
            models = kwargs.get('compare_models')
            self.train_graph.make_subplots('Model Comparison',
                                           models={'m1': models['m1'],
                                                   'm2': models['m2'],
                                                   'm3': models['m3']})

        if 'plot_training_subplots' in kwargs.keys():
            self.train_graph.make_subplots(self.train_graph.title)

        if continue_matching:
            check_n_size(self._n)
            idx = 1
            while self._n['y1']:
                n = dict(y1=self._n['y1'].pop(0),
                         y2=self._n['y2'].pop(0),
                         y4=self._n['y4'].pop(0))
                self._fit(n, idx)
                idx += 1

            self.ideal_fn_df = pd.DataFrame(data=self.ideal_fn_dict)
            self.ideal_fn_df = self.ideal_fn_df.set_index('x')

            self.test_df = self.test.csv_to_df()
            test_model = Model(self.test_df['x'], self.test_df['y'], 1, df=self.test_df)

            finals = test_model.match_ideal_functions(self.ideal_fn_df,
                                                      self.train_master,
                                                      self.models,
                                                      map_train=map_train)

            if 'run_complete' in kwargs.keys():
                self.test.df_to_db(finals[0])
            else:
                self.result = finals

    def _fit(self, n, idx):

        _m = f'm{idx}'
        new_models = dict()
        for i in range(1, 5):

            col = f'y{i}'
            _if, _max, _bf = f'y{i}_if', f'y{i}_max_err', f'y{i}_best_fit'

            if i != 3:
                model, error_df = self.train.fit_model(
                    i, self.ideal, 'poly.fit',
                    order=n[col], print_table=self.print_table)
            else:
                model, error_df = self.train.fit_model(
                    i, self.ideal, 'linear',
                    print_table=self.print_table)

            new_models[col] = model
            self.models[col] = model
```

```
            self.ideal_fn_dict[model.ideal_col] = model.ideal_col_array
            self.train_master[_if] = model.ideal_col
            self.train_master[_max] = model.max_dev
            self.train_master[_bf] = model

            if self.plot:
                self.train_graph.plot_model(model,
                                            plt_type=self.plt_type,
                                            with_rmse=self.with_rmse)

            if self.plot_order_error:
                self.train_graph.plot_model(error_df, plt_type='error', _col=col)

        if idx == 1:
            self.models_master_1[_m] = new_models
        if idx == 2:
            self.models_master_2[_m] = new_models
        if idx == 3:
            self.models_master_3[_m] = new_model
```

APPENDIX F: TEST_INTERFACE.PY

```
import unittest
import pandas as pd
from interface import Interface, NSizeError


class InterfaceTest(unittest.TestCase):

    def setUp(self):
        self._n = {
            'y1': [5, 21, 36],
            'y2': [5, 22, 36],
            'y4': [3, 9, 27]
        }

    def test_training_subplots(self):
        self.assertTrue(Interface(plot_training_subplots=True,
                                  continue_matching=False))

    def test_without_plot(self):
        df = Interface(plot=False,
                       map_train=False,
                       _n=self._n,
                       to_db=False,
                       create_tables=False)
        self.assertEqual(type(pd.DataFrame()), type(df.result),
                         'should return pandas df')

    def test_with_plot(self):
        df = Interface(map_train=True,
                       _n=self._n,
                       to_db=False,
                       create_tables=False)
        self.assertEqual(type(tuple()), type(df.result),
                         'should return pandas df')
        self.assertEqual(type(dict()), type(df.result[1]))

    def test_compare_models(self):
        df = Interface(map_train=True,
                       _n=self._n,
                       to_db=False,
                       create_tables=False)

        df2 = Interface(continue_matching=True,
                        _n=self._n,
                        compare_models={'m1': df.models_master_1,
                                        'm2': df.models_master_2,
                                        'm3': df.models_master_3})

    def test_run_complete(self):
        df = Interface(map_train=True,
                       _n=self._n,
                       to_db=False,
```

```
                        create_tables=False,
                        run_complete=True)
        print(df.test.df.size)


if __name__ == '__main__':
    unittest.main()
```

APPENDIX G: TEST_DATABASE.PY

```
import unittest
from sqlalchemy import create_engine, MetaData
from database import Data, RegressionException
import pandas as pd
import numpy as np
from model import Model


class DatabaseTest(unittest.TestCase):
    engine = create_engine("sqlite:///python_models.db", echo=True)
    meta = MetaData()

    def setUp(self):
        self.df = pd.DataFrame(np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
                               columns=['a', 'b', 'c'])
        self._data = Data("unittest")
        self._data.add_test_table('unittest')

    def test_csv_to_df(self):
        self.df.to_csv('./tests/unittest.csv')
        self._data.csv_to_df()
        self.assertEqual(type(pd.DataFrame()), type(self._data.df))

    def test_csv_to_db(self):
        self.assertTrue(self._data.csv_to_db())

    def test_fit_model(self):
        with self.assertRaises(RegressionException):
            self._data.fit_model(1, Data('ideal'), 'real')
        self.df = pd.DataFrame([[1.5555555, 2.55555555], [4.5555555, 5.5555555]],
                               columns=['x', 'y1'])
        self.df.to_csv('./tests/unittest.csv')
        self._data.csv_to_df()
        with self.assertRaises(Exception):
            self._data.fit_model(1, None, 'linear')
        # test print_table arg
        with open('./datasets/ideal.csv', 'r') as idf:
            ideal = Data('ideal', _create=False)
            ideal.csv_to_df()
            self.assertFalse(ideal.is_empty(), 'df obj should be populated')
            model, model_df = self._data.fit_model(1, ideal, 'linear',
                                                   print_table=True, table_name='./tests/unittest')
            self.assertEqual(type(model), type(Model(self.df['x'], self.df['y1'], 1)))
            self.assertEqual(model.x[0], self.df['x'][0])

    def tearDown(self):
        try:
            self._data.drop_test_table()
        except AttributeError:
            print('table not dropped')


if __name__ == '__main__':
    unittest.main()
```

APPENDIX H: TEST_VISUALIZE.PY

```
import unittest
import numpy as np
import pandas as pd

from model import Model
from visualize import Graph, PlotTypeException
```

```
class VisualizeTest(unittest.TestCase):

    def setUp(self):
        rand_x, rand_y = np.random.randint(-20, 20, size=400), \
                         np.random.randint(-1000, 1000, size=400)
        self.df = pd.DataFrame()
        self.df['x'] = rand_x
        self.df['y1'] = rand_y
        self.df['y2'] = rand_y
        self.df['y3'] = rand_y
        self.df['y4'] = rand_y
        self.df.sort_values(by=['x', 'y1', 'y2', 'y3', 'y4'], inplace=True)
        self.testGraph = Graph('train', df=self.df)
        self.subdir = './tests'

        self.m1 = Model(self.df['x'], self.df['y1'], 1, rss=0.02345, order=1)
        self.m2 = Model(self.df['x'], self.df['y2'], 1, rss=0.02345, order=2)
        self.m3 = Model(self.df['x'], self.df['y3'], 1, rss=0.12345, order=3)
        self.m4 = Model(self.df['x'], self.df['y4'], 1, rss=0.12345, order=4)

        self.m1d = {'m1': {'y1': self.m1, 'y2': self.m2, 'y3': self.m3, 'y4': self.m4}}
        self.m2d = {'m2': {'y1': self.m1, 'y2': self.m2, 'y3': self.m3, 'y4': self.m4}}
        self.m3d = {'m3': {'y1': self.m1, 'y2': self.m2, 'y3': self.m3, 'y4': self.m4}}

    def test_make_subplots(self):
        testGraphPlots = self.testGraph.make_subplots(
            self.testGraph.title, subdir=self.subdir
        )
        self.assertTrue(testGraphPlots)
        with open(f'{self.subdir}/{self.testGraph.title}.pdf', 'r') as file:
            self.assertTrue(file, 'file should have been created in (subdir) dir')

        # test model comparison

        testGraphTest = Graph('model', df=self.df)
        tg = testGraphTest.make_subplots('model',
                                         models={'m1': self.m1d, 'm2': self.m2d, 'm3': self.m3d},
                                         subdir=self.subdir)
        self.assertTrue(tg)
        for i in range(1, 4):
            with open(f'{self.subdir}/y{i}_{testGraphTest.title}.pdf', 'r') as file:
                self.assertTrue(file, 'file should have been created in (subdir) dir')

    def test_plot_model(self):
        with self.assertRaises(PlotTypeException):
            self.testGraph.plot_model(self.m1)
        self.assertTrue(self.testGraph.plot_model(
            self.m1, plt_type='best fit', subdir='./tests')
        )

        rand_x, rand_y = np.random.randint(-20, 20, size=400), \
                         np.random.randint(-1000, 1000, size=400)
        df = pd.DataFrame()
        df['x'] = rand_x
        df['y'] = rand_y
        fit_model = Model(df['x'], df['y'], 1)
        testGraph = Graph('idealtest', df=df)
        self.assertTrue(testGraph.plot_model(
            fit_model, plt_type='ideal', subdir='./tests', fit_model=fit_model)
        )


if __name__ == '__main__':
    unittest.main()
```

APPENDIX I: TEST_MODEL.PY

```
import unittest

from sklearn.metrics import mean_squared_error

from database import Data
from model import Model
```

```
import pandas as pd
import numpy as np


class ModelTest(unittest.TestCase):

    def setUp(self):
        rand_x, rand_y = np.random.randint(-20, 20, size=400), \
                         np.random.randint(-1000, 1000, size=400)
        self.df = pd.DataFrame()
        self.df['x'] = rand_x
        self.df['y'] = rand_y
        self.df.sort_values(by=['x', 'y'], inplace=True)
        self.testModel = Model(self.df['x'], self.df['y'], 1, df=self.df)

        rand_x, rand_y = np.random.randint(-20, 20, size=400), \
                         np.random.randint(-1000, 1000, size=400)
        self.dfTest = pd.DataFrame()
        self.dfTest['x'] = rand_x
        self.dfTest['y1'] = rand_y
        self.dfTest['y2'] = rand_y
        self.dfTest['y3'] = rand_y
        self.dfTest['y4'] = rand_y
        self.dfTest.sort_values(by=['x', 'y1', 'y2', 'y3', 'y4'], inplace=True)
        self.subdir = './tests'

        self.m1 = Model(self.dfTest['x'], self.dfTest['y1'], 1, rss=0.02345, order=1)
        self.m2 = Model(self.dfTest['x'], self.dfTest['y2'], 1, rss=0.02345, order=2)
        self.m3 = Model(self.dfTest['x'], self.dfTest['y3'], 1, rss=0.12345, order=3)
        self.m4 = Model(self.dfTest['x'], self.dfTest['y4'], 1, rss=0.12345, order=4)
        self.m1d = {'y1': self.m1, 'y2': self.m2, 'y3': self.m3, 'y4': self.m4}
        self.m2d = {'y1': self.m1, 'y2': self.m2, 'y3': self.m3, 'y4': self.m4}
        self.m3d = {'y1': self.m1, 'y2': self.m2, 'y3': self.m3, 'y4': self.m4}

    def test_find_ideal_function(self):
        with open('./datasets/ideal.csv', 'r') as csv:
            idealData = Data('ideal', _create=False)
            idealData.csv_to_df()
            self.assertEqual(self.testModel.__getattribute__('rmse'), 1000,
                             'rmse should be at the default 1000')
            self.testModel.find_ideal_function(idealData)
            self.assertNotEqual(self.testModel.__getattribute__('rmse'), 1000,
                                'rmse should change after function runs')
            self.assertTrue(self.testModel.__getattribute__('ideal_col'),
                            'ideal_col should change after function runs')
            self.assertNotEqual(self.testModel.__getattribute__('max_dev'), 1000,
                                'max_dev should change after function runs')
            self.assertNotEqual(self.testModel.df.size, 0,
                                'Model() df should be init')


if __name__ == '__main__':
    unittest.main()
```

APPENDIX J: GIT COMMANDS

> git clone iu_model
> git checkout development
> git commit -a -m “New feature x”
> git push origin development
> git request-pull v1.0 https://repo.url master

Bibliography

Draper, N. R., & Smith, H. (1998). *Applied Regression Analysis: Third Edition* (3rd ed., Ser. Wiley Series in Probability and Statistics). Wiley-Interscience.

Oliphant, T. E. (2008). *Polynomial.fit*. NumPy. https://numpy.org/doc/stable/reference/generated/numpy.polynomial.polynomial.Polynomial.fit.html#numpy.polynomial.polynomial.Polynomial.fit.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., & Prettenhofer, P. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Rawlings, J. O., Dickey, D. A., & Pantula, S. G. (1998). Least Squares Estimation. In *Applied regression analysis: a research tool* (Ser. Springer Texts in Statistics, pp. 3–22). essay, Springer.

Strutz, T. (2011). Data fitting and uncertainty a practical introduction to weighted least squares and beyond. Vieweg + Teubner.

Sáez, P. B., & Rittmann, B. E. (1992). Model-parameter estimation using least squares. *Water Research*, *26*(6), 789–796. https://doi.org/10.1016/0043-1354(92)90010-2

Virtanen P., Gommers R., Oliphant T. E., Haberland M., Reddy T., Cournapeau D., Burovski E., and SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17(3), 261-272.