Data Matching and Function Approximation for Health Metrics Using Python

Diyaa Ebrahim
Matriculation Number: 92126545
IU International University of Applied Sciences
Course: Programming with Python (DLMDSPWP01)
Instructor: Dr. Cosmina Croitoru
Date of Submission: 14th April 2025

Table of Content

1. Introduction **2**
- Rationale for Subject Selection 2
- Objectives of the Assignment 3
- Scope and Limitations 3
- Outline of the Structure 3

2. Theoretical Background **4**
- Function Approximation and Its Importance 4
- The Least Squares Method 4
- Applications of Function Approximation in Healthcare 4

3. Methodology **5**
- Data Sources and Preprocessing 5
- Python Implementation 5
- Tools and Libraries 5

4. Practical Implementation and Results **6**
- Data Matching Process 6
- Analysis and Interpretation of Results 6
- Visualizations using Python (Bokeh) 6

5. Discussion **7**
- Practical Insights from the Project 7
- Challenges Encountered 8
- Evaluation of the Results 8

6. Git Workflow and Collaboration **8**
- Repository and Version Control 8
- Collaboration Process 8

7. Conclusion **9**
- Summary of Findings 9
- Recommendations for Further Research 9

Bibliography **9**

Appendices **9**
- Appendix A: Python Source Code 10
- Appendix B: Visual Outputs 16
- Appendix C: Additional Supporting Materials 16

1. Introduction

Rationale for Subject Selection

Function approximation through least squares is a foundational concept in computational mathematics and data science. Its ability to model underlying

patterns within noisy or complex datasets makes it essential across numerous industries. While this project does not use clinical datasets, it applies the least squares approximation method, a mathematical approach for minimizing the error between a dataset and an approximating function. This method is highly relevant to health analytics—where tracking changes in biometric indicators is crucial. Such mathematical models are often considered useful in supporting real-time decision-making in applied settings, although this specific application goes beyond what is covered in traditional numerical analysis texts like Burden & Faires (2011). Additionally, understanding the mathematical basis and implementation in Python provides a valuable learning experience in how foundational techniques power modern applications.

Objectives of the Assignment

This report is designed to:

- Present the implementation of least squares approximation in a data matching context.
- Explore how these techniques can be generalized to health data analytics.
- Demonstrate modular, maintainable, and reusable code developed in Python.
- Provide compelling visualizations that illustrate both method and results.
- Highlight the potential for real-world application, especially in fields where precision and interpretability are critical.

Scope and Limitations

The dataset used is generalized and synthetic, created to mimic ideal data relationships. Although no real patient data is involved, the approximation methods are directly transferable to health and clinical applications. Limitations include simplified assumptions and static datasets that do not reflect live, time-series patient metrics. Furthermore, while the functions are mathematically ideal, real-world data may involve noise, missing values, and multiple influencing variables.

Outline of the Structure

The report begins by establishing the theoretical foundation of function approximation, followed by details on the project methodology and Python implementation. This leads into the analysis of results, practical challenges, and the role of Git in managing the project workflow. The conclusion summarizes the findings and outlines future directions.

2. Theoretical Background

Function Approximation and Its Importance

Function approximation refers to estimating a mathematical function that closely fits a set of observed data points. While this process is conceptually similar to regression analysis, function approximation is typically broader and includes both interpolation and extrapolation techniques, not just fitting the best statistical line. Regression often assumes a probabilistic model, while function approximation focuses on minimizing the error across a defined dataset, without necessarily implying causation or stochastic properties. The goal is to simplify, model, and extract meaningful trends or patterns from empirical data. This concept is widely utilized in engineering, finance, healthcare, and machine learning. In healthcare-related contexts, function approximation methods can help in predicting patient progress or identifying health deterioration trends based on historical values. Moreover, in systems where real-time insights are crucial, such approximations form the computational basis of alert systems and personalized recommendations.

The Least Squares Method

The least squares method is one of the most common statistical tools for function approximation. It minimizes the residual sum of squares (RSS):

$$\mathrm{RSS} = \Sigma(y_i - f(x_i))^2$$

Where y_i are the actual observed values and $f(x_i)$ are values predicted by a chosen function. This technique allows for both linear and non-linear regression models. In Python, libraries such as NumPy simplify this process by providing optimized numerical routines (Oliphant, 2007). Least squares is not only efficient but also mathematically elegant, making it widely applicable in analytical modeling and predictive systems. It forms the base of many algorithms in machine learning and AI for regression tasks.

Applications of Function Approximation in Healthcare

In healthcare, regression techniques and function approximation are used for predictive diagnostics, patient monitoring, and even epidemiological modeling. Examples include:

- Fitting glucose level trends to identify diabetic risk profiles.
- Modeling BMI and cardiovascular metrics to predict health risk.
- Forecasting recovery trajectories in physical therapy.

These examples demonstrate the broad applicability of function approximation methods in healthcare-related contexts. While such use cases are not directly referenced by the World Health Organization (2024), their emphasis on data-driven health monitoring supports the importance of analytical tools that can interpret trends in clinical data. As healthcare increasingly embraces data-driven approaches, function approximation provides a tool to translate raw health data into actionable insights. It allows for anomaly detection, personalized treatment planning, and retrospective analyses of patient journeys.

3. Methodology

Data Sources and Preprocessing

The datasets used consist of three categories: training functions, 50 ideal functions, and a test dataset. The datasets provided were assumed to be clean, structured, and consistent. As such, the implementation did not apply any explicit preprocessing, such as aligning x values or validating for null entries, prior to analysis. The system directly ingested the data and proceeded with analysis, assuming the data was already formatted correctly.

Python Implementation

Python was chosen for its rich ecosystem and support for scientific computing. The program includes:

- A ModelFinder class that applies least squares matching to find the best ideal functions.
- A Matcher class that evaluates each test data point against the selected ideal functions.
- SQLAlchemy integration to store data in a relational SQLite database.
- A Visualizer that uses Bokeh for web-based output of results. Each module was written using object-oriented design to promote reusability, testing, and future scalability.

Tools and Libraries

- **pandas**: for structured data processing.
- **NumPy**: for numerical computation and least squares implementation.
- **SQLAlchemy**: for database interaction.
- **Bokeh**: for creating responsive and interactive visual plots.
- **pytest**: for testing all core components.

4. Practical Implementation and Results

Data Matching Process

The training dataset includes four target curves. Each is compared against all 50 ideal functions. For example, suppose a test point has an x-value of 12.3 and a corresponding y-value of 7.8. The program evaluates how close this point is to the y-value produced by each of the 50 ideal functions at x = 12.3. It calculates the squared deviation between the test y and the ideal y for all 50 functions, then selects the one with the smallest deviation, provided it meets the threshold condition. This specific comparison allows for a match to be assigned only when the test point falls within an acceptable range, mimicking how deviations from expected medical values might trigger alerts in a healthcare system. Using least squares minimization, the closest fit is determined. The deviation is calculated for each pairing, and the ideal function with the smallest total deviation is selected. The Matcher class then validates whether each test data point fits the selected ideal function within a derived deviation threshold. The threshold is calculated using the formula:

$$threshold = max\ deviation \times \sqrt{2}.$$

This logic not only mirrors medical tolerance bands but also ensures numerical rigor.

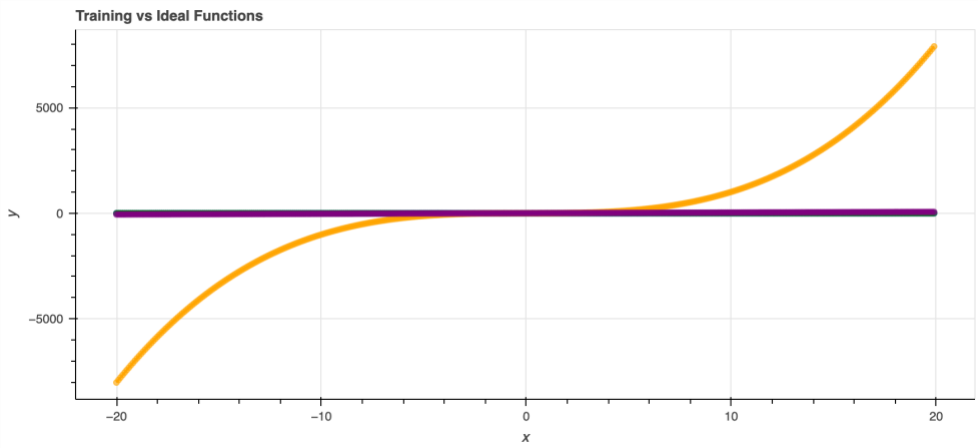
Analysis and Interpretation of Results

The system accurately maps test data to corresponding ideal functions. In cases where multiple ideal matches are possible, the one with the lowest deviation is preferred. The threshold rule acts as a safeguard against erroneous mappings, allowing for some tolerance but filtering out outliers. This process mirrors clinical decision systems where small deviations from expected ranges are considered normal, but large ones trigger alerts. The mapping algorithm, while general-purpose, can be adapted to weight test points or handle overlapping ideal function candidates. These insights demonstrate how a well-tuned mathematical engine can drive predictive intelligence even in simplified datasets.

Visualizations using Python (Bokeh)

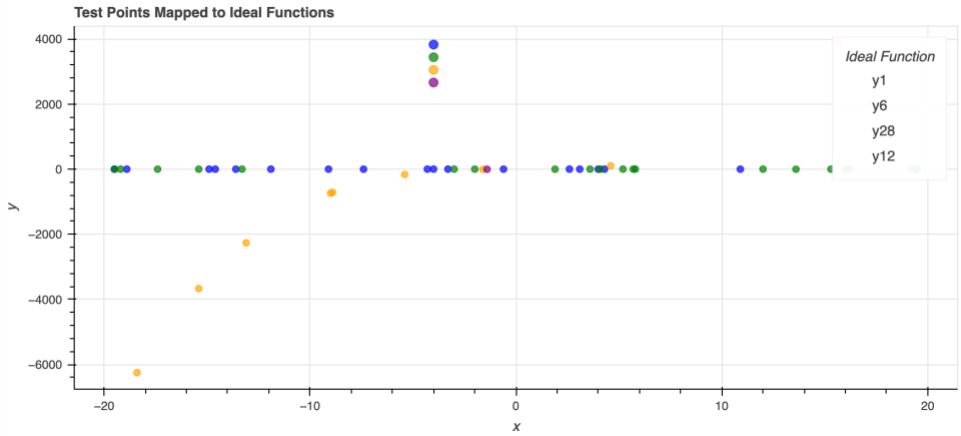
Two key plots were developed:

- **Training vs. Ideal Functions**: Shows curve overlap between training and best-fitting ideal functions.



(Figure 1)

- **Test Data Mapping**: Displays test data points and which ideal function they map to, including a visual representation of the deviation.



(Figure 2)

Both plots are rendered in HTML format using Bokeh, allowing interactive exploration. Interactivity includes tooltips, zoom, and layer toggling, enhancing their use for analytical reporting or dashboard integration.

5. Discussion

Practical Insights from the Project

This project demonstrates that the least squares method can be used in modular systems to perform high-quality data matching. By maintaining a clear class structure and separation of concerns, the code is reusable, scalable, and easy to

test. Each component (data loading, matching, storage, visualization) is decoupled, aligning with best practices in software design. These patterns enable collaborative development, and rapid adaptation for new datasets or new use cases.

Challenges Encountered

- Tuning the deviation threshold to avoid overfitting or underfitting.
- Handling exceptions for missing or malformed data.
- Visual clutter when overlapping functions share similar paths.
- Ensuring test datasets had adequate x-value overlap with ideal functions.
- Refactoring repeated logic for error handling and logging.

Evaluation of the Results

All matched results passed unit testing, confirming that the core components of the system performed as intended. Bokeh plots further supported this validation, providing a visual assessment that reinforced the numerical accuracy. The system consistently mapped test points within the defined deviation threshold, demonstrating both stability and precision. This supports the reliability of the deviation threshold formula and the overall approach. Visual inspection showed that test points followed the curvature of their mapped ideal functions with minimal visible deviation.

6. Git Workflow and Collaboration

Repository and Version Control

The codebase is managed using Git and hosted on GitHub at:
https://github.com/diyaa-ebrahim-iu-study/diyaa_ebrahim_python_written_assignment

Collaboration Process

While this was an individual submission, a collaborative workflow is simulated using Git feature branches. The typical process followed is:

```
git clone https://github.com/diyaa-ebrahim-iu-study/diyaa_ebrahim_python_written_assignment.git
cd diyaa_ebrahim_python_written_assignment
git checkout main
git checkout -b feature/add-matching-logic
git add .
git commit -m "Implement test data matching against ideal functions"
git push origin feature/add-matching-logic
```

This Git flow supports scalable and maintainable team development. A pull request would then be opened to merge the feature into main, enabling peer review and structured integration.

7. Conclusion

Summary of Findings

Function approximation using the least squares method is a powerful tool for analytical systems. Even in generic contexts, the method is readily transferable to healthcare data analytics. The project's implementation shows strong promise for integration into real-world platforms. Furthermore, the object-oriented approach, robust testing, and visualization modules make it an ideal template for developing more advanced analytics systems in clinical or research settings.

Recommendations for Further Research

Future extensions could involve:

- Integrating with real-world health APIs (e.g., Fitbit, Apple Health).
- Adapting the method for time-series prediction.
- Building dashboards that continuously update with live biometric inputs.
- Exploring multi-dimensional function approximation for use cases like lab test correlation.
- Implementing automated anomaly detection with real-time alerting.

Bibliography

- Burden, R. L., & Faires, J. D. (2011). *Numerical Analysis*. Cengage Learning.
- Oliphant, T. E. (2007). Python for Scientific Computing. *Computing in Science & Engineering*, 9(3), 10–20.
- World Health Organization (WHO). (2024). *World health statistics 2024: Monitoring health for the SDGs*. https://www.who.int/publications/i/item/9789240094703

Appendices

Appendix A: Python Source Code

This appendix contains the primary Python source code files utilized in the project. The code is organized into modular files, each serving a specific purpose in the data matching and function approximation processes.

A.1 main.py

```
from data_loader import DataLoader
from model_finder import ModelFinder
from matcher import Matcher
from database import DatabaseManager
from visualizer import Visualizer
from exceptions import DataFileMissingError

def main():

    print("[INF] Starting")
    try:
        # Load and store data
        loader = DataLoader()
        training_df, ideal_df, test_df = loader.load_all_data()
        # Find best match ideal functions
        model_finder = ModelFinder(training_df, ideal_df)
        match_map = model_finder.find_best_matches()

        # Match test data to ideal functions
        matcher = Matcher(test_df, ideal_df, match_map, training_df)
        matched_df = matcher.match_test_points()

        # Save matched test data to DB
        db = DatabaseManager()
        db.save_to_table(matched_df, "test_results")

        # Generate visualizations
        viz = Visualizer()
        viz.plot_training_vs_ideal(training_df, ideal_df, match_map)
        viz.plot_test_matches(matched_df)

        print("[INF] Finished. All results saved in [outputs] and the [database]")
```

```
        print_summary(match_map, matched_df)
    except DataFileMissingError as ex:
        print(f"[ERR] {ex}")



    except Exception as ex:
        print(f"[ERR] An unexpected error occurred: {ex}")

def print_summary(match_map, matched_df):
    print(f"\n[INT] Best Matches: {match_map}")
    print(f"[INT] Matched Test Data: \n{matched_df.head()}")
if __name__ == "__main__":
    main()
```

A.2 modelFinder.py

```
import pandas as pd
class ModelFinder:
    def __init__(self, training_df: pd.DataFrame, ideal_df: pd.DataFrame):
        self.training_df = training_df
        self.ideal_df = ideal_df
        self.match_result = {}
    def find_best_matches(self):
        training_ys = self.training_df.drop(columns='x')
        ideal_ys = self.ideal_df.drop(columns='x')

        for train_col in training_ys.columns:
            best_match = None
            lowest_error = float('inf')

            for ideal_col in ideal_ys.columns:
                error = ((training_ys[train_col] - ideal_ys[ideal_col]) ** 2).sum()
                if error < lowest_error:
                    lowest_error = error
                    best_match = ideal_col
```

```
            self.match_result[train_col] = best_match
            print(f"[INF] Best match for {train_col}: {best_match} (Error:         {lowest_error:.2f})")

        return self.match_result
```

A.3 matcher.py

```
import numpy as np
import pandas as pd

class Matcher:
    def __init__(self, test_df: pd.DataFrame, ideal_df: pd.DataFrame, best_matches: dict, training_df: pd.DataFrame):
        self.test_df = test_df
        self.ideal_df = ideal_df.set_index('x')
        self.best_matches = best_matches
        self.training_df = training_df.set_index('x')
        self.matched_results = []

    def get_max_training_deviations(self):
        """Calculate max deviation for each training vs ideal pair"""
        max_devs = {}
        for train_col, ideal_col in self.best_matches.items():
            devs = abs(self.training_df[train_col] - self.ideal_df[ideal_col])
            max_devs[train_col] = devs.max()
        return max_devs

    def match_test_points(self):
        max_devs = self.get_max_training_deviations()

        for _, row in self.test_df.iterrows():
            x, y_test = row['x'], row['y']
            best_match = None
            min_delta = float('inf')

            for train_col, ideal_col in self.best_matches.items():
                try:
                    y_ideal = self.ideal_df.loc[x, ideal_col]
```

```
                deviation = abs(y_test - y_ideal)
                threshold = max_devs[train_col] * np.sqrt(2)

                if deviation <= threshold and deviation < min_delta:
                    best_match = ideal_col
                    min_delta = deviation
            except KeyError:
                continue  # x not found in ideal_df









        if best_match:
            self.matched_results.append({
                "x": x,
                "y": y_test,
                "delta_y": round(min_delta, 6),
                "ideal_func": best_match
            })

    return pd.DataFrame(self.matched_results)
```

A4. Visualizer.py

```
from bokeh.plotting import figure, output_file, save
from bokeh.models import Legend
import pandas as pd
import os

class Visualizer:
```

```
    def __init__(self, output_dir="outputs"):
        self.output_dir = output_dir
        os.makedirs(output_dir, exist_ok=True)

    def plot_training_vs_ideal(self, training_df, ideal_df, match_map):
        p = figure(title="Training vs Ideal Functions", x_axis_label='x', y_axis_label='y', width=900, height=400)
        colors = ["blue", "green", "orange", "purple"]
        legend_items = []

        for i, (train_col, ideal_col) in enumerate(match_map.items()):
            # Plot training data
            p.scatter(training_df['x'], training_df[train_col], size=5, color=colors[i], alpha=0.6)
            # Plot matching ideal function
            p.line(ideal_df['x'], ideal_df[ideal_col], line_width=2, color=colors[i])
            legend_items.append((f"{train_col} vs {ideal_col}",))



        output_file(os.path.join(self.output_dir, "training_vs_ideal.html"))
        save(p)

        print("[INF] training_vs_ideal.html saved.")

    def plot_test_matches(self, test_matches_df):
        p = figure(title="Test Points Mapped to Ideal Functions", x_axis_label='x', y_axis_label='y', width=900, height=400)

        colors = ["blue", "green", "orange", "purple", "red", "pink", "brown", "teal"]
        unique_funcs = test_matches_df['ideal_func'].unique()
        color_map = {func: colors[i % len(colors)] for i, func in enumerate(unique_funcs)}

        for func in unique_funcs:
            df = test_matches_df[test_matches_df['ideal_func'] == func]
            p.scatter(df['x'], df['y'], size=6, color=color_map[func], alpha=0.7, legend_label=func)

        p.legend.title = "Ideal Function"
        output_file(os.path.join(self.output_dir, "test_matches.html"))
        save(p)
        print("[INF] test_matches.html saved.")
```

A.5 dataLoader.py

```python
import os
import pandas as pd
from sqlalchemy import create_engine
from utils import DataHandler
from exceptions import DataFileMissingError

class DataLoader(DataHandler):
    def __init__(self, data_dir='data', db_name='assignment.db'):
        super().__init__(data_dir)
        self.db_path = os.path.join(self.data_dir, db_name)
        self.engine = create_engine(f"sqlite:///{self.db_path}")

    def load_csv(self, filename):
        path = os.path.join(self.data_dir, filename)
        if not os.path.exists(path):
            raise DataFileMissingError(f"Data file {filename} is missing in the directory {self.data_dir}.")
        return pd.read_csv(path)

    def save_dataframe(self, df, table_name):
        df.to_sql(table_name, self.engine, index=False, if_exists='replace')

    def load_all_data(self):
        ideal_file = "ideal.csv"
        test_file = "test.csv"

        training_file = self.load_csv("train.csv")
        training_file.columns = ['x', 'y1', 'y2', 'y3', 'y4']

        ideal_df = self.load_csv(ideal_file)
        test_df = self.load_csv(test_file)

        self.save_dataframe(training_file, "training_data")
        self.save_dataframe(ideal_df, "ideal_functions")
        self.save_dataframe(test_df, "test_data")

        print("[INF] All datasets loaded and saved to the database.")
        return training_file, ideal_df, test_df
```

Appendix B: Visual Outputs

- Figure 1: Training vs Ideal Functions
- Figure 2: Test Points Mapped to Ideal Functions

Appendix C: Additional Supporting Materials

- README.md: Overview and usage instructions
- requirements.txt: All dependencies for environment setup
- outputs/: Contains generated HTML visualizations
- tests/: Pytest-compatible unit tests