AI-Driven Data Visualization: Revolutionizing Data Analysis Through Automation

Charlie Robinson (200678841)

May 2024

BSc. Computer Science

Supervisor - Lei Shi

Word Count: 16,369

Abstract

In today's data-driven world, the ability to analyse vast datasets swiftly and derive actionable insights is critical, especially in fields such as healthcare. This dissertation introduces an AI-driven data visualization tool aimed at revolutionizing the way data is analysed by enhancing the speed, accessibility, and efficacy of data analysis through automation. A comprehensive literature review pinpoints existing gaps in current AI-driven visualization technologies, highlighting the demand for sophisticated tools. This project integrates advanced machine learning algorithms, including clustering, dimensionality reduction, and anomaly detection, to foster an innovative approach to data visualization. The development of the tool was guided by user feedback and an iterative design process, resulting in a prototype that underscores the tool's potential to streamline data analysis workflows. Usability testing and comparative evaluations demonstrate marked improvements in user experience and analytical performance. The findings underscore the transformative potential of integrating AI in data visualization, paving the way for future research and discussion on democratizing data analysis technology across various sectors. This dissertation documents the progression from concept to execution of a ground-breaking tool that not only pushes the boundaries of data visualization but also contributes to the ongoing integration of AI technologies in analytical practices.

Declaration

"I declare that this document represents my own work except where otherwise stated".

Charlie Robinson

Acknowledgements

I would like to thank first all my lecturers and tutors for their help throughout my journey at Newcastle University. Also, both my Mum and Dad for their continued support in getting to these final stages of my higher education.

Table Of Contents

1 Introduction

1.1 Motivation and Rationale

Motivation:

1. Rapid Increase in Data Volume and Complexity: The exponential growth of data across various sectors, especially in healthcare, has introduced significant challenges in data management and interpretation. Traditional data analysis methods are increasingly unable to cope with the rising complexity and volume of data. This escalating data environment demands more sophisticated and automated solutions to maintain the effectiveness of data analytics.

2. Limitations of Current Data Visualization Tools: Many existing data visualization tools are limited to static representations that do not support automated pattern recognition or insight extraction. This limitation hinders the ability to make rapid, informed decisions, underscoring the need for advanced tools that integrate AI to offer dynamic, insightful visual analytics.

3. The Need for Automation in Data Analysis: There is a broad acknowledgment of the importance of data analytics across various industries, coupled with an urgent need to automate the processing of large datasets to enhance efficiency. Automation in data analysis can significantly boost productivity, improve decision-making, and secure a competitive advantage by enabling faster and more accurate insights.

Rationale:

1. AI as a Solution: AI-driven data visualization is a compelling response to the challenges. AI can streamline the process of pattern recognition and generate insights from complex datasets, thus accelerating data analysis and making it more accessible to a broader audience. By leveraging AI, data visualization tools can transform raw data into actionable intelligence efficiently.

2. Bridging the Gap: My project aims to bridge the significant gap between the current capabilities of data visualization tools and the evolving demands of data-driven decision-making. Through the integration of machine learning algorithms and a user-

centric design, the project develops a tool that is not only powerful but also intuitive and user-friendly, catering to the needs of diverse users.

3. Contribution to the Field: This project is poised to make a substantial contribution to the fields of data visualization and AI. It aligns with ongoing technological advancements and addresses the pressing needs of industries that rely heavily on data analytics, such as healthcare. By enhancing the capabilities of data visualization tools, the project supports more informed decision-making processes and promotes a data-informed organizational culture.

4. Innovation and Impact: The innovative aspects of this project lie in its ability to integrate advanced AI techniques with practical, user-friendly interfaces, potentially setting a new standard in data analysis tools. The anticipated impact of the project includes making data-driven decision-making more accessible, faster, and more accurate, particularly in sectors where timely and precise insights are critical.

Conclusion:

By addressing these key motivations and providing a solid rationale for the development of an AI-driven data visualization tool, this project not only tackles existing gaps in technology but also pioneers' advancements that could reshape how data is utilized in decision-making processes across multiple sectors. This not only reflects a significant step forward in the application of AI in data analytics but also underscores a commitment to improving how organizations leverage data for substantial, impactful decisions.

1.2 Aims and Objectives

Aim:

"To develop an AI-driven data visualization tool that enhances the efficiency and effectiveness of data analysis for users without expertise in data science, thereby democratizing the ability to gain actionable insights from complex datasets."

Objectives:

1. Literature Review: Conduct a comprehensive review of existing AI-driven data visualization tools to identify strengths, limitations, and areas for improvement. This will inform the development of a more effective tool.
2. User Engagement: Identify potential users and engage with them through surveys or interviews to gather insights on their data visualization needs and challenges. This feedback will guide the user-centric design of the tool.
3. Prototype Design: Design AI-driven data visualization prototypes that incorporate advanced machine learning algorithms for intuitive pattern recognition and insight generation.

4. Project Development: Develop the designed prototypes into functional tools, integrating user feedback and ensuring compatibility with various types of datasets.
5. Usability Testing: Conduct usability testing with target users to assess tool effectiveness, usability, and performance. Use this feedback to refine the tool.
6. Evaluation and Iteration: Evaluate the tool's performance based on predefined metrics such as user satisfaction and the relevance of insights generated and iterate on the design based on user feedback and test results.

Further Ethical Considerations:

Since the initial proposal, the project scope has expanded to include a broader range of data types. All data used for testing and development is anonymized and aggregated to ensure privacy and compliance with GDPR."

Structure of the Dissertation:

The dissertation is organized into six main chapters: Introduction, Background, Development, Testing, , Results, and Conclusion & Evaluation. Each chapter systematically builds upon the before exploring the development and impact of the AI-driven data visualization tool.

1.3 Project Schedule

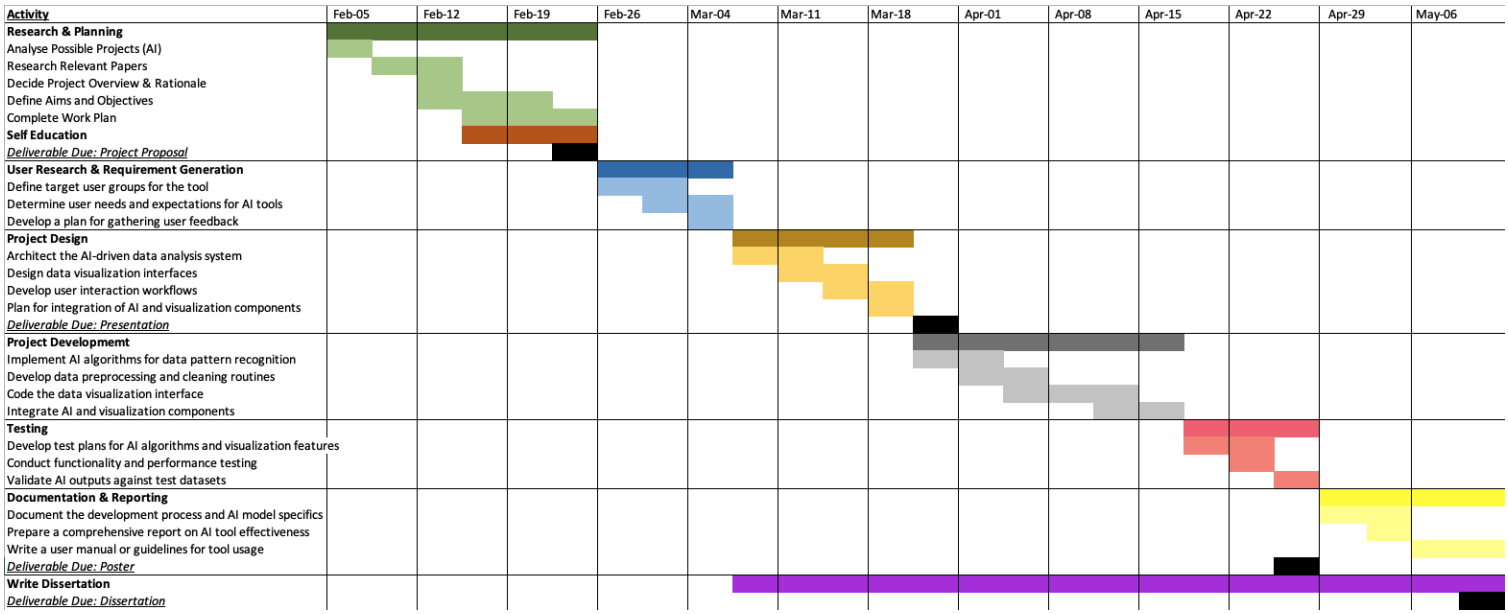


Figure 1.3.1 – Project Schedule

In the introductory chapter of my dissertation, the project schedule is a pivotal component that provides a timeline for the various stages of development. The initial segment, 'Research & Planning,' adhered closely to the established schedule, offering a clear path for immediate tasks, and facilitating a smooth transition into the project.

The subsequent phase, 'User Research & Requirement Generation,' followed a more flexible approach. Given the precise aims and objectives already outlined for the project, this segment received a proportionate level of attention, ensuring a focused direction without the need for extensive user research.

Progressing to 'Project Design,' I found myself moving more swiftly than anticipated. The enthusiasm to transition ideas into tangible outcomes meant that tasks like designing data visualization interfaces naturally migrated into the development phase. This resulted in a condensed design phase that effectively set the stage for the practical aspects of the project.

The bulk of the project's life cycle resided in 'Project Development.' This phase was characterized by iterative trial and error, an integral part of the creative process. By pre-emptively embarking on this segment, I managed to compensate for its demanding nature, ensuring that development was completed within the projected timeframe.

As for 'Testing,' this stage took place after the project development stage was to a level where users could engage with the program with ease as the aim for my usability testing was for people who are inexperienced with data science and insight extraction could use this program to do so. As for functionality, and performance testing I wanted to ensure that any potential issues were addressed prior to the final release. This proactive approach in the project schedule allowed for a thorough vetting of the system, contributing to a robust and user-friendly product.

Lastly, the 'Dissertation Writing' phase began later than initially planned, a testament to the engrossing nature of the development work. Despite this delay, the momentum gained from the project's progression facilitated an efficient transition into this final phase. The insights and results derived from earlier stages provided a rich foundation for crafting the dissertation, enabling a focused and comprehensive articulation of the research findings. This stage was crucial for synthesizing all aspects of the project into a coherent narrative that not only documented the developmental journey but also highlighted the innovations and implications of the research.

2 Background

2.1 AI Background

Artificial Intelligence (AI) is reshaping data visualization by providing tools that can interpret complex data sets with minimal human oversight. According to the 2024 AI Index Report, "AI has surpassed human performance on several benchmarks, yet it trails behind on more complex tasks like competition-level mathematics" [8]. This illustrates that AI able to handle many tasks, and this project is to attempt to use AI to deliver intricate analytical tasks that are crucial for advanced data visualization.

Evolution of AI in Data Visualization:

The progression of AI technologies has led to the development of systems that enhance the interpretability of vast data arrays. The AI Index notes, "Recent advancements have led to the development of strong multimodal models…capable of handling images, text, and, in some instances, audio" [8]. This multimodality is pivotal for data visualization applications that require the integration of various data types.

Core AI Techniques Relevant to Data Visualization:

- Machine Learning Algorithms: These are fundamental for identifying patterns and anomalies in data. As the Stanford HAI outlines, "Large language models keep scaling in size and expense… driving up the capabilities in data processing and analysis" [8].
- Neural Networks and Deep Learning: Essential for complex data operations, these techniques have been vital in advancing AI's applicability in data visualization. "Deep learning allows computational models…to learn representations of data with multiple levels of abstraction," from LeCun, Bengio, and Hinton's seminal work on deep learning [9].
- Natural Language Processing (NLP): This enhances the accessibility of data visualization tools by allowing them to generate summaries and insights in a natural language format. "NLP technologies are rapidly becoming a foundation for interfaces that allow more natural user interaction with machines," [10]

Applications and Case Studies:

In sectors like healthcare, AI-driven visualization tools assist significantly. For instance, "AI enhances the analysis of Electronic Health Records (EHRs) to support diagnostic processes and predictive analytics," highlighting its role in transforming data analysis methods in healthcare [11]

Challenges and Future Directions:

Despite its advancements, AI in data visualization is not without challenges. Issues such as "data privacy concerns, the need for large training datasets, and potential biases in algorithms," are critical areas needing addressing to advance AI use ethically and effectively [12].

Conclusion:

AI-driven data visualization represents a significant technological leap, enhancing the efficiency and effectiveness of data analysis across various domains. As these tools become more sophisticated, they promise to transform traditional data analysis methods, making them more dynamic and accessible.

2.2 Data Processing Techniques

2.2.1Regex

Regular expressions, commonly abbreviated as regex, are specialized tools used extensively in text processing and data manipulation. Originating from theoretical computer science, particularly within automata theory and formal language theory, regex was conceptualized by mathematician Stephen Kleene in the 1950s. His foundational work laid the groundwork for regex's integration into modern programming languages, enhancing text processing capabilities significantly.

Functionality and Applications:

A regular expression is essentially a sequence of characters that forms a search pattern, used primarily in string-searching algorithms for "find" or "find and replace" operations, as well as for validating input. Regex is implemented across various programming languages through comprehensive library support, making it an essential tool for developers.

Typical Uses of Regex:

1. Validation: Regex is crucial for verifying whether a string conforms to a specific pattern, such as email addresses, phone numbers, or URLs, thus ensuring the accuracy and reliability of data within applications.
2. Searching: It enables the identification and location of specific text within larger documents or logs, facilitating efficient data analysis and management.
3. Extraction: Regex can isolate specific segments of text based on defined patterns, which is invaluable for data mining tasks, such as extracting all email addresses from a document or specific transaction IDs from logs.
4. Replacement: It allows for the alteration or substitution of text within strings, enabling bulk modifications in coding and content development processes.

The capabilities of regex extend beyond simple text manipulation, playing a pivotal role in complex data parsing and transformation tasks across diverse programming environments. Its versatility and efficiency make it an indispensable tool in the arsenal of modern software developers.

2.2.2 Dimensionality Reduction

Dimensionality reduction is a fundamental data preprocessing technique used in machine learning to reduce the number of random variables under consideration. This technique is essential for simplifying models, speeding up computation, and helping to alleviate the “curse of dimensionality” which can hamper machine learning algorithms.

Dimensionality reduction can be achieved through various methods, the most common being Principal Component Analysis (PCA) and Singular Value Decomposition (SVD). PCA is particularly popular for its ability to transform a large set of variables into a

smaller one that still contains most of the information in the large set. This is achieved by retaining the principal components, which are the directions in the data that maximize variance [13].

Applications of Dimensionality Reduction:

1. Visualization: Simplifying high-dimensional data to 2D or 3D can help visualize complex datasets.
2. Increased Efficiency: Reducing the number of features can decrease the computational burden on algorithms.
3. Noise Reduction: By eliminating less important variables, dimensionality reduction can help improve the performance of machine learning models.

2.2.3 Clustering

Clustering is a technique used to group a set of objects in such a way that objects in the same group (called a cluster) are more like each other than to those in other groups. It's a method of unsupervised learning, and a common technique for statistical data analysis used in many fields.

The most common clustering techniques include K-means clustering, hierarchical clustering, and DBSCAN (Density-Based Spatial Clustering of Applications with Noise). K-means clustering, for instance, partitions n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster [14].

Applications of Clustering:

1. Market Segmentation: Identifying distinct groups within the customer base to tailor marketing strategies.
2. Social Network Analysis: Detecting communities within large groups of people or organizations.
3. Anomaly Detection: Identifying unusual data points that do not fit into any group.

2.3 Data Visualization Techniques

Data visualization plays a crucial role in understanding, interpreting, and communicating large sets of data. Effective visualization helps to uncover patterns, trends, and correlations that might not be apparent in raw data. In this section, I will explore various visualization techniques that are widely used across different industries and discuss their applicability to data-driven decision-making.

Common Visualization Techniques:

1. Line Graphs and Time Series Analysis: Line graphs are fundamental for displaying data over time, allowing viewers to easily observe trends, seasonal variations, and

cyclical patterns. This method is particularly useful in financial forecasting, environmental monitoring, and any scenario where time-dependent analysis is required.

2. Bar Charts and Histograms: Bar charts are versatile tools for comparing quantities across different categories. Histograms, a type of bar chart, are used to represent the distribution of numerical data, helping identify skewness, outliers, and the central tendency of data sets.

3. Scatter Plots: These are used to visualize the relationship between two quantitative variables, helping to identify correlations, clusters, and outliers. They are especially useful in statistical analysis and predictive modelling to test hypotheses about causal relationships.

4. Heatmaps: A heatmap is a graphical representation of data where individual values contained in a matrix are represented as colours. They are excellent for exploring the correlation between features or visualizing data density on a map.

5. Box Plots: Box plots or box-and-whisker diagrams are used to depict the distribution of a dataset based on a five-number summary: the minimum, first quartile, median, third quartile, and maximum. They are particularly effective for identifying outliers and understanding the variability of the data.

Comparison and Selection Criteria:

Choosing the right visualization technique depends on the nature of the data and the specific insights that need to be communicated. For instance, time series data are best represented by line graphs to highlight trends over time, while categorical data might be better served by bar charts for direct comparisons.

Conclusion:

In this project, while I have primarily utilized tools like Matplotlib and Altair for creating interactive and static visualizations, understanding a broad range of techniques allows for selecting the most effective method based on the data and analysis goals. The choice of visualization technique can significantly influence the interpretation and decision-making processes, making it imperative to choose wisely based on the dataset and the questions at hand. Future considerations will involve evaluating the integration of more complex visualization tools and techniques as data sources grow and analytical needs evolve.

2.4 Literature Review and Related Works:

Background
[1] “Efficient Deep Learning for Visual and Textual Data”
Description:

The paper by Sachin Mehta introduces efficient neural architectures tailored for computer vision and natural language processing tasks. These architectures, including the Efficient Spatial Pyramid (ESP) unit and the pyramidal recurrent unit (PRU), focus on reducing computational overhead while maintaining performance.

Relevance:
This paper presents novel approaches to designing lightweight neural networks for efficient data analysis tasks. By integrating similar efficient architectures into my data visualization tool, we can enhance its performance on edge devices with limited computational resources, aligning with the project's goal of revolutionizing data analysis through automation.

[2] “Natural Language Interface for Data Visualization with Deep Learning Based Language Models”
Description:
This study explores integrating a Deep Learning language model into information visualization software's Natural Language Interface (NLI). Using OpenAI Codex, the authors developed a web application for creating visualizations from text input. A comparative study with a classical NLP approach was conducted to assess usability and suitability for various visualization types.

Relevance:
This paper investigates incorporating a Deep Learning language model into a visual data analysis tool, like my project's goals. It highlights the potential of Deep Learning models, such as Open AI Codex, in enhancing text input for creating visualizations. However, challenges such as unpredictability and comprehension issues are noted. Findings suggest a need for refinement to balance expressiveness and usability. Incorporating insights from this study can inform AI-driven features in my data visualization tool.

[3] “Deep Learning for Anomaly Detection”
Description:
This article provides a review of deep learning techniques for anomaly detection, a crucial area in various research domains. It surveys advancements in deep anomaly detection methods, categorizing them into high-level and fine-grained categories. The review discusses their key features, advantages, disadvantages, and potential future opportunities.

Relevance:
This paper explores the application of deep learning in anomaly detection, which intersects with the data analysis aspect of my project. Understanding the latest advancements in deep anomaly detection methods can inform the development of AI-driven algorithms for pattern recognition and insight generation in my data visualization tool. By incorporating insights from this review, the project can leverage state-of-the-art techniques to enhance anomaly detection capabilities, thereby improving the overall effectiveness of data analysis and decision-making processes.

[4] “An Anatomy of Machine Learning Data Visualization”
Description:
This paper explores the integration of machine learning algorithms with data visualization, emphasizing the importance of data literacy and well-designed visualizations in understanding large datasets. It reviews experiments on data visualization using machine learning algorithms implemented in Python platforms.

Relevance:
This paper delves into the intersection of machine learning and data visualization, which directly aligns with the objectives of my project. Understanding how machine learning techniques can enhance data visualization can inform the development of AI-driven features in my data visualization tool. By integrating insights from this paper, the project can leverage machine learning algorithms to enhance pattern recognition and insight generation, thereby improving the effectiveness of data analysis and decision-making processes.

3. Development

3.1 Data Processing Components

The data processing pipeline in the system is designed to handle complex datasets seamlessly through a series of automated steps. The pipeline is orchestrated by `main.py`, which ensures each component interacts efficiently and effectively.

3.1.1 Data Loading

Firstly, the data must be loaded correctly. This seems obvious but due to the complex nature of this program allowing any dataset to be uploaded, there are a few steps to this ran by `data_loader.py`:

File Type Detection: It determines the file extension of the input and chooses the appropriate method to load the data.

```
elif str(file.name).endswith('.xlsx'):
    return pd.read_excel(file)
elif str(file.name).endswith('.json'):
    return pd.read_json(file)
elif str(file.name).endswith('.txt'):
    return pd.read_csv(file, delimiter='\t')
```

Figure 3.1.1.1 – Identifying file type.

Support for Multiple Formats: It can load data from CSV, XLSX, JSON, and TXT (tab-separated) files.

Encoding Handling for CSVs: For CSV files, it initially tries to read the file using UTF-8 encoding. If that fails, it attempts ISO-8859-1 (Latin1), and then Windows-1252 encodings. This was identified during testing when some CSV files would not load, and this handling had to be added.

```
if str(file.name).endswith('.csv'):
    # Attempt to read a CSV file
    try:
        return pd.read_csv(file, encoding='utf-8')
    except UnicodeDecodeError:
        try:
            return pd.read_csv(file, encoding='ISO-8859-1')  # Try latin1 encoding
        except UnicodeDecodeError:
            return pd.read_csv(file, encoding='cp1252')  # Try Windows-1252 encoding
    except pd.errors.EmptyDataError:
        print("The file is empty. Please upload a valid file.")
        return None
```

Figure 3.1.1.2 – Encoding .CSV files.

Error Handling: It includes error handling for cases where the file cannot be loaded due to issues like empty data or other exceptions, with appropriate error messages printed.

```
except pd.errors.EmptyDataError:
    print("The file is empty. Please upload a valid file.")
    return None
```

Figure 3.1.1.3 – Error Handling.

```
except Exception as e:
    # Catch any other exception and log it
    print(f"Failed to load file due to: {e}")
    return None
```

Figure 3.1.1.4 – Error Handling.

Once the data is loaded, `data_cleaner.py` takes over. This script is crucial as it pre-processes the data to meet the analysis standards required by downstream components. It includes tasks such as removing duplicates, handling missing values, normalizing data, and other necessary transformations to clean and prepare the data. This step is vital for ensuring the quality and reliability of the insights generated by the AI models later in the pipeline.

3.1.2 Column Recognition and Handling

ID Column Removal:

Removing ID columns from datasets is crucial for ensuring that the data used for analysis does not contain irrelevant or misleading features. In my project, I

implemented a `drop_id_columns` function that utilizes several methods to accurately identify and remove such columns based on specific criteria.

Code Implementation and Workflow:

1. Keyword Detection for ID Columns:
  - The function starts by defining a list of keywords commonly associated with ID columns. It then iterates over each column name in the dataset, checking if any of these keywords appear as full words in the column names, which helps avoid mistakenly identifying columns that contain these letters as part of other words.

```
# Define keywords for identifying ID columns
id_keywords = ['_id', 'id_', ' id', 'id ']

# Initialize list to store ID columns
id_columns = []

# Iterate over the columns of the DataFrame
for col in data.columns:
    # Check for full-word match in column name
    if any(re.search(r'\b{}\b'.format(keyword), col.lower()) for keyword in id_keywords):
        # If full-word match is found, add the column to the list of ID columns
        id_columns.append(col)
```

Figure 3.1.2.1 Identifying ID columns.

- This snippet shows the regular expression search that checks for full-word matches, ensuring columns like 'acidity' are not incorrectly identified as ID columns due to the presence of 'id'.

2. Incremental Numeric Detection:
  - After checking for keyword matches, the function examines numeric columns to determine if they increment by one—another common characteristic of ID columns. This check is only performed if the column was not previously identified by keywords.

```
# Check if the data in the column increments by 1 row by row
elif pd.api.types.is_numeric_dtype(data[col]):
    is_id_column = True
    prev_value = None
    for value in data[col]:
        if prev_value is not None and value != prev_value + 1:
            is_id_column = False
            break
        prev_value = value

    if is_id_column:
        id_columns.append(col)
```

Figure 3.1.2.2 - Backup for identifying ID columns.

- This section of the code demonstrates the logic used to detect if a column behaves like an ID column by incrementing by one between rows, serving as a fallback method when keyword detection is insufficient.

3. Column Removal and Dataframe Update:
- Once potential ID columns are identified, they are removed from the dataset. The function then returns the cleaned DataFrame, free of these ID columns.

```
print(f"Dropping ID columns: {id_columns}")

# Drop the ID columns from the DataFrame, ignoring errors
data_no_id = data.drop(columns=id_columns, errors='ignore')

# Return the modified DataFrame
return data_no_id
```

Figure 3.1.2.3 – Dropping ID columns.

- This snippet should be used to illustrate how the identified ID columns are dropped from the DataFrame, effectively cleaning the data for subsequent analysis processes.

Integration of ID Removal in Data Processing:

Incorporating the 'drop_id_columns' function into the data processing pipeline is crucial for enhancing data quality and analysis accuracy. This step strategically removes ID columns, ensuring that only pertinent data influences the analytical results. By stripping the dataset of irrelevant features, this function solidifies the foundation for meaningful analysis, markedly improving the reliability of the insights generated.

The meticulous integration of the drop_id_columns function underscores its importance in the data preprocessing phase of my project. This procedure guarantees that the analysis is not skewed by extraneous data, thereby refining the overall accuracy and trustworthiness of the final outcomes.

Date and Time Processing:

Handling time and date data efficiently is critical due to the variability in how this information is recorded across different datasets. To address this, I developed a robust function, detect_time_date_columns, which identifies and standardises time and date columns, ensuring they are in a consistent format suitable for further analysis.

Detailed Implementation Steps

1. Dictionary Setup for Date and Time Patterns:
- Initially, dictionaries for common date and time formats (`date_formats` and `time_pattern`) are defined. These dictionaries are crucial for recognizing various date and time formats in the dataset.

```
time_date_cols = []
data2norm = data.copy()
date_format_YYYY = {
    "%Y-%m-%d": r'\d{4}\D\d{1,2}\D\d{1,2}'  # YYYY-M-D or YYYY-MM-DD
}
date_formats = {
    "%d/%m/%Y": r'\d{1,2}[\/\- ]\d{1,2}[\/\- ]\d{4}',  # D/M/YYYY or DD/MM/YYYY
    "%m/%d/%Y": r'\d{1,2}[\/\- ]\d{1,2}[\/\- ]\d{4}'   # M/D/YYYY or MM/DD/YYYY
}
time_pattern = r'\d{2}:\d{2}(:\d{2})?(\sAM|\sPM)?'
```

Figure 3.1.2.4 -Dictionary Setup for Date and Time

2. Date Format Recognition:
- The function includes the capability to recognize and differentiate between various date formats, such as 'day/month/year' and 'month/day/year'. From a regex perspective, these formats might appear similar, but the function systematically analyses sample values to determine the correct format. Here is a code snippet that illustrates how a sample of data is taken for each column to identify date formats:

```
for col in data.columns:
    sample_values = data[col].dropna().astype(str).sample(min(100, len(data[col])))
    # Flag to track if column has been processed as date or time
    processed_as_date_or_time = False
```

Figure 3.1.2.5 – Taking Sample Values

- This approach was required because the parsing process by default assumes the loaded date format will be in the country format of the processing machine. The sampling allows this assumption to be overridden, and the actual date format is identified by examining the date data to identify the actual date format of each column based on the assumption that values over 12 must be the month column. This is crucial for accurate data analysis and reporting.

```
def parse_date(
    indexes = sorted([(year_idx, 'Y'), (month_idx, 'M'), (day_idx, 'D')])
    indexes = {item[1]: idx for idx, item in enumerate(indexes)}

    # FIXME: this currently only supports numbers, but should also support month
    #        names, both in the requested locale, and english

    year = numbers[indexes['Y']]
    year = 2000 + int(year) if len(year) == 2 else int(year)
    month = int(numbers[indexes['M']])
    day = int(numbers[indexes['D']])
    if month > 12:
        month, day = day, month
    return datetime.date(year, month, day)
```

post Aa .ab .* No results

Figure 3.1.2.11 – Parse date function

3. Detection and Normalization Process:

- The function iterates through each column, checking against the defined patterns to determine if the column contains date or time data.
- When a match is found, the column is processed accordingly and converted into a standardized format.

```
# Check if any sample value matches the YYYY date format
if any(re.search(pattern, value) for pattern in date_format_YYYY.values() for value in sample_values):
    # Process the column as a date
    data[f'{col}_date'] = pd.to_datetime(data[col], format='%Y-%m-%d', errors='coerce')
    data2norm[f'{col}_date'] = pd.to_datetime(data[col], format='%Y-%m-%d', errors='coerce').dt.strftime('%Y-%m-%d')
    # Mark the column as processed
    processed_as_date_or_time = True
    time_date_cols.append(f'{col}_date')

# Check if any sample value matches any of the other date formats
elif any(re.search(pattern, value) for pattern in date_formats.values() for value in sample_values):
    # Further processing for other date formats
    sample_dates = [value for value in sample_values if any(re.search(date_formats[fmt], value) for fmt in date_formats)]
    if sample_dates:
        # Infer the date format based on the first sample date
        inferred_format = infer_date_format(sample_dates[0])
        if inferred_format:
            # Process the column according to the inferred format
            data[f'{col}_date'] = pd.to_datetime(data[col], format=inferred_format, errors='coerce')
            data2norm[f'{col}_date'] = pd.to_datetime(data[col], format=inferred_format, errors='coerce').dt.strftime('%Y-%m-%d')
            # Mark the column as processed
            processed_as_date_or_time = True
            time_date_cols.append(f'{col}_date')
```

Figure 3.1.2.6 – Normalizing date and times

4. Splitting Combined Date and Time Columns:
- For columns where date and time are combined, this function splits these into separate date and time columns according to identified patterns. This separation is essential for facilitating distinct types of analysis. Dates are treated as time series data, which is effective for generating time series reports. In contrast, times are treated as categorical data, as categorical reporting often yields better insights for time-based analysis. Considering the effectiveness of these approaches, future work could explore enhanced methods for analysing time as a categorical variable alongside dates in time series analyses.

```
if any(re.search(time_pattern, value) for value in sample_values):
    # Process as time
    data[f'{col}_time'] = data[col].apply(parse_time)
    data2norm[f'{col}_time'] = data[col].apply(parse_time2norm)
    # Mark column as processed
    processed_as_date_or_time = True
    #time_date_cols.append(f'{col}_time')
```

Figure 3.1.2.7 – Splitting Date and Time

5. Handling Different Data Types for Analysis and Visualization:
- The function returns two versions of the dataset: one normalized for use in dimensionality reduction and clustering, and another retaining the datetime64 data types suitable for generating time series graphs.

```
    # If the column was processed as date or time, drop the original column
    if processed_as_date_or_time:
        data.drop(col, axis=1, inplace=True)
        data2norm.drop(col, axis=1, inplace=True)


# Assuming the function should return the modified DataFrame and the list of new time/date related columns
return time_date_cols, data, data2norm
```

Figure 3.1.2.8 – Drop original date / time columns.

The `detect_time_date_columns` function is a sophisticated component of the data preprocessing phase, addressing the complexities associated with time and date data in diverse datasets. By standardizing these data types, the function not only prepares the dataset for advanced analytical techniques but also ensures that visualization tools can effectively represent temporal data. This process is vital for the accurate interpretation of time-based patterns and trends within the dataset, enhancing the overall analytical capabilities of the project.

Identifying Financial and Categorical Data Columns:

Accurate identification of financial and categorical columns is crucial for generating insightful graphs and conducting specific analyses. To streamline this process, I have implemented methods that leverage keyword matching and datatype assessments.

Identifying Financial Columns:

1. Keyword-Based Detection:
   - A list of financial keywords is defined, including terms such as "profit," "cost," "income," and various currency names and symbols. These keywords help identify columns likely to contain financial data.

```
financial_keywords = [
    'revenue', 'cost', 'profit', 'expense', 'income', 'gross', 'salary',
    'dollar', 'dollars', 'euro', 'pound', 'pounds', 'sterling', 'yen',
    'rupee', 'ruble', 'real', 'peso', 'franc', 'lira', 'rand', 'krona',
    'won', 'yuan', 'renminbi', 'euros', 'pounds', 'dollars', 'rupees',
]
```

Figure 3.1.2.9 – Financial Keywords

   - This list is used to scan column names for potential financial data, with Regex ensuring only full-word matches are considered to prevent false positives like "Europe" being misidentified.

2. Currency Symbol Detection:
   - Besides keyword detection in column names, the function also checks for currency symbols within the data of each column. This method helps capture financial columns where the header might not clearly indicate the financial nature, but the content does.

```
for col in data.columns:
    # Check for full-word match in column name
    if any(re.search(r'\b{}\b'.format(keyword), col.lower()) for keyword in financial_keywords):
        financial_cols.append(col)
        continue  # If the keyword is found in the column name, no need to check for symbols in its data

    # Check for presence of currency symbols in the data of the column
    if data[col].astype(str).str.contains(r'[\$\£\€]', regex=True).any():
        financial_cols.append(col)
```

Figure 3.1.2.10 – Currency Symbol Detection

Identifying Categorical Columns:
The identification of categorical columns is more straightforward, leveraging pandas' ability to detect non-numeric data types:
- Automatic Detection Using Pandas:
  - Pandas is utilized to automatically classify columns with 'object' datatype as categorical. This process is efficient and ensures that all potential categorical data, typically stored as strings, are accounted for without manual intervention.

3.1.3 Missing Values and Imputation

Handling of values:

The initial step in the data preprocessing phase involves determining how to manage missing values, a critical aspect for ensuring the integrity of the analysis. Users are presented with two options upon uploading their dataset:

1. Imputation Using TPOT: If users opt for imputation, TPOT (Tree-based Pipeline Optimization Tool), an automated machine learning tool, is utilized. TPOT helps in selecting the best strategies to fill in missing values, which may vary between categorical and numerical data.

2. Dropping Rows with Missing Values: Alternatively, users can choose to drop any rows that contain missing values. This option is straightforward and ensures that only complete data entries are used for analysis, which can simplify the visualization and insight detection processes.

```
if handle_missing_values == False:
    print("Dropping rows with missing values.")
    num_rows_dropped = len(data) - len(data.dropna())
    data.dropna(inplace=True)
    normalized_data.dropna(inplace=True)
    print(f"Dropped {num_rows_dropped} rows with missing values.")
```

Figure 3.1.3.1 – Dropping missing values.

Automated Machine Learning with TPOT for Imputation:

The integration of TPOT for automated machine learning specifically targets the challenge of imputing missing values effectively:

1. Detecting Columns with Missing Values:
  - The process starts by identifying which columns within the dataset have missing values. This is crucial for determining where imputation strategies need to be applied.
  - Code snippet for screenshot:

```
# Identify columns with missing values
columns_with_missing_values = data.columns[data.isnull().any()].tolist()
#print("\nColumns with missing values before imputation:", columns_with_missing_values)

# Loop over columns with missing values and apply predictive imputation
for column in columns_with_missing_values:
    data = predictive_imputation(data, column)
return data
```

Figure 3.1.3.2 – Detecting missing values.

```
if handle_missing_values != False:
    normalized_data = handle_missing_values_with_tpot(normalized_data)
```

Figure 3.1.3.5 – Calling function to handle missing values

2. Predictive Imputation Process:
  - TPOT is used to automate the creation of a machine learning model that predicts missing values based on the rest of the data in the dataset. This approach is particularly useful for handling large datasets where manual imputation would be impractical.
  - Code snippet for screenshot:

```
# Fitting the model and predicting the missing values
tpot_model.fit(X_train, y_train)
predicted_values = tpot_model.predict(X_test)
data.loc[data[column_to_impute].isnull(), column_to_impute] = predicted_values
missing_after = data[column_to_impute].isnull().sum()
```

Figure 3.1.3.3 – Predictive Imputation

3. Handling Different Data Types:
  - Depending on whether the column to impute is numeric or categorical, TPOT will configure itself to use either a regression or classification model, respectively. This flexibility allows TPOT to adapt its strategy to the data type, enhancing the accuracy of the imputation.

```
missing_before = data[column_to_impute].isnull().sum()
if missing_before == 0:
    print(f"No missing values in '{column_to_impute}'. No imputation needed.")
    return data

if pd.api.types.is_numeric_dtype(data[column_to_impute]):
    tpot_model = TPOTRegressor(max_time_mins=30, generations=3, population_size=50, verbosity=2, random_state=42)
    print(f"Using TPOTRegressor for numeric column: {column_to_impute}")
else:
    tpot_model = TPOTClassifier(max_time_mins=30, generations=5, population_size=50, verbosity=2, random_state=42)
    print(f"Using TPOTClassifier for categorical column: {column_to_impute}")
```

Figure 3.1.3.4 – Handling Different Data Types

By providing users the choice between dropping rows or imputing values and by employing TPOT for automated machine learning, the project not only enhances the quality of the data but also ensures that the subsequent analyses are based on robust and comprehensive datasets. This approach maximizes the potential for accurate and insightful outcomes from the data analysis process, reflecting a sophisticated and user-oriented handling of preliminary data challenges.

3.1.4 Data Encoding and Normalisation

Categorical Data Encoding:

For effective trend detection within the dataset, it's crucial to ensure that all data is in a numerical format. This requirement poses a challenge when dealing with categorical data, which must be transformed from text to numerical representation through encoding. This step is vital for enabling further processing such as normalization, dimensionality reduction, and clustering.

Encoding Methods:
1. One-Hot Encoding:
  - Best for categorical variables with a small number of unique values (e.g., titles such as Mr., Mrs., Miss, which would be encoded as 1, 2, 3 respectively).
  - Implemented when the unique categories are few to avoid creating excessively sparse matrices.

```
# Determine encoding strategy based on the number of unique values
unique_values = normalized_data[col].nunique()
if unique_values <= 5:
    # OneHot encode if unique values are 5 or less
    encoded = one_hot_encoder.fit_transform(normalized_data[[col]])
    encoded_data = pd.DataFrame(encoded, columns=one_hot_encoder.get_feature_names_out([col]), index=normalized_data.index)
    normalized_data = pd.concat([normalized_data.drop(columns=[col]), encoded_data], axis=1)
```

Figure 3.1.4.1 – One-Hot Encoding

2. Label Encoding:
  - Suitable for columns with a larger number of categories.
  - Each category is assigned a unique integer based on the order of appearance, which is particularly useful for modelling without expanding the dataset dimensionality excessively.

```
else:
    # Label encode if unique values are more than 5
    normalized_data[col] = label_encoder.fit_transform(normalized_data[col])
```

Figure 3.1.4.2 – Label Encoding

Normalization:

Normalization involves adjusting the scale of numerical columns to standardize them to have zero mean and unit variance. This process is essential for techniques like dimensionality reduction and clustering, which depend heavily on the distance between data points.

Standard Scaler Application:
- Numerical columns are scaled using `StandardScaler`, which is crucial for maintaining consistency in data measurement scales across various features.

```
def normalize_data(normalized_data):
    scaler = StandardScaler()
    numerical_cols = normalized_data.select_dtypes(include=['float64', 'int64']).columns
    normalized_data[numerical_cols] = scaler.fit_transform(normalized_data[numerical_cols])
    #print("Normalized data:\n", normalized_data.head())

    return normalized_data
```

Figure 3.1.4.3 - Normalization

Workflow Integration:

1. Initialization of Encoders:
  - Encoders are set up to handle different types of categorical data efficiently.

```
# Initialize encoders
one_hot_encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
label_encoder = LabelEncoder()
```

Figure 3.1.4.4 – Initializing Encoders

2. Data Preparation:
  - The choice between one-hot and label encoding is made based on the number of unique values in each categorical column.

```python
if normalized_data[col].dtype == 'object':
    # Fill NaN values in categorical columns with a placeholder
    #normalized_data[col].fillna('missing', inplace=True)

    # Determine encoding strategy based on the number of unique values
    unique_values = normalized_data[col].nunique()
    if unique_values <= 5:
        # OneHot encode if unique values are 5 or less
        encoded = one_hot_encoder.fit_transform(normalized_data[[col]])
        encoded_df = pd.DataFrame(encoded, columns=one_hot_encoder.get_feature_names_out([col]), index=normalized_data.index)
        normalized_data = pd.concat([normalized_data.drop(columns=[col]), encoded_df], axis=1)
    else:
        # Label encode if unique values are more than 5
        normalized_data[col] = label_encoder.fit_transform(normalized_data[col])
```

Figure 3.1.4.5 – Selecting either One-Hot or Label Encoding

3. Final Data Output:
- After encoding and normalization, the prepared data is ready for downstream processes. The progress and transformations are typically displayed to ensure correctness and transparency.

```python
normalized_data = converted_to_normalize.copy()
print("Normalized data:\n", normalized_data.head(10))
```

Figure 3.1.4.6 – Checking Normalization Success

This structured approach to handling categorical data and normalization ensures that the dataset is optimally prepared for advanced analytical processes, allowing for accurate and insightful trend detection and analysis.

3.1.5 Issues Found in Development of Data Processing

Initially when planning my project and in the early stages of development I was going to use a library called AUTOML, this library would not only handle the missing values and impute these, but it would also automate a lot of the process when it came to dimensionality reduction and clustering. However, as I was installing this library it became apparent it would not be compatible with my system I was developing on (M1 Mac). This raised a few issues as I had to go back to the drawing board and create a new plan on how to automate this process of finding these trends in the database and imputing missing values.
Firstly, I found another library (TPOT) to handle the missing values and use predictive imputation. However, it seemed there wasn't many other libraries that could do what AUTOML could do in a way I wanted it to identify the trends, so I took it upon myself to automate this process myself.

3.2 Model Management

3.2.1 Dimensionality Reduction

In the development of the data analysis pipeline for my project, a key step involved employing dimensionality reduction techniques to help identify underlying trends in the dataset. Dimensionality reduction is crucial for simplifying complex data sets, making them easier to explore and analyse, especially when dealing with high-dimensional data.

Approach to Dimensionality Reduction:

Initially, I considered using an automated library like AutoML to streamline the process of selecting the most efficient machine learning models. However, as mentioned previously, I decided to manually implement this process, focusing on three main dimensionality reduction techniques:
1. PCA (Principal Component Analysis)
2. t-SNE (t-Distributed Stochastic Neighbour Embedding)
3. UMAP (Uniform Manifold Approximation and Projection)

Each of these methods was chosen for their ability to reduce the number of variables in the dataset while preserving its essential characteristics as much as possible.

Methodology and Implementation:

The dataset undergoes each of the three dimensionality reduction methods, and their effectiveness is evaluated based on the silhouette score—a measure from the SKLearn's metrics library. The silhouette score helps in assessing the quality of the clustering achieved by the dimensionality reduction, automating the selection of the most suitable method for the specific dataset.

```python
def complete_analysis_pipeline(data, normalized_data):
    # Ensure 'data' is a DataFrame
    if not isinstance(data, pd.DataFrame):
        raise ValueError("Data needs to be a pandas DataFrame.")

    # Dictionary to store reduced data for each method
    reduced_data_methods = {}
    silhouette_scores = {}

    # Apply each dimensionality reduction method
    reduced_data_methods['PCA'], _ = apply_optimal_pca(normalized_data, 2)
    reduced_data_methods['t-SNE'] = apply_tsne(normalized_data)
    reduced_data_methods['UMAP'] = apply_umap(normalized_data)

    # Calculate silhouette score for each reduced data and find the best one
    for method, reduced_data in reduced_data_methods.items():
        # Temporarily apply KMeans for silhouette score calculation; adjust based on your
        #criteria or data
        #trend_labels = KMeans(n_trends=5, random_state=42).fit_predict(reduced_data)
        trend_labels = optimal_kmeans(reduced_data)
        score = silhouette_score(reduced_data, trend_labels)
        silhouette_scores[method] = score
        #print(f"{method} silhouette score: {score}")

    best_method = max(silhouette_scores, key=silhouette_scores.get)
    #print(f"Best dimensionality reduction method: {best_method} with a
    #silhouette score of {silhouette_scores[best_method]}")

    # Perform trending on the best reduced data
    best_reduced_data = reduced_data_methods[best_method]
```

Figure 3.2.1.1 – Analysis Pipeline

Detailed Function Descriptions:

1. PCA: The `apply_optimal_pca` function is used to perform PCA based on a variance threshold, which dictates how many components are retained to explain the specified amount of variance in the data.

2. t-SNE: The `apply_tsne` function reduces data dimensions while trying to keep similar instances close and dissimilar instances apart. It is particularly useful for visualizing high-dimensional data in low-dimensional spaces.

3. UMAP: The `apply_umap` function is another powerful technique used for dimensionality reduction, known for preserving more of the global data structure compared to t-SNE, making it useful for a broader range of data types.

```python
def apply_optimal_pca(processed_data, variance_threshold=0.95):
    pca = PCA(n_components=variance_threshold)
    reduced_data = pca.fit_transform(processed_data)
    #n_components_optimal = pca.n_components_
    #print(f"Optimal number of components found: {n_components_optimal},
    #explaining {np.sum(pca.explained_variance_ratio_)*100:.2f}% of variance.")

    return reduced_data, variance_threshold

def apply_tsne(processed_data, n_components=2):
    n_samples = processed_data.shape[0]
    perplexity = min(30, max(5, n_samples / 3))  # Adjust perplexity based on the number of samples
    tsne = TSNE(n_components=n_components, perplexity=perplexity, random_state=42)
    reduced_data = tsne.fit_transform(processed_data)

    return reduced_data

def apply_umap(processed_data, n_components=2, n_neighbors=15, min_dist=0.1, random_state=42):
    reducer = umap.UMAP(n_components=n_components, n_neighbors=n_neighbors, min_dist=min_dist, random_state=42)
    reduced_data = reducer.fit_transform(processed_data)

    return reduced_data
```

Figure 3.2.1.2 – PCA, t-SNE, UMAP

The choice of dimensionality reduction method can significantly impact the ability to discern and interpret the underlying patterns in the data. By evaluating each method using the silhouette score, the most effective approach can be selected, enhancing the overall quality of the trend analysis in the dataset. This strategic selection of dimensionality reduction techniques ensures that the data is optimally prepared for clustering and subsequent trend identification, crucial for insightful data-driven decision-making.

3.2.2 Clustering

In the development of this project, I utilized two primary clustering algorithms: K-Means and DBSCAN. Each method was chosen for its unique ability to uncover patterns in data, and I applied custom optimization functions to enhance their effectiveness based on the specific characteristics of our dataset.

K-Means Clustering Implementation:

K-Means is a widely used clustering method that partitions data into K distinct clusters by minimizing the within-cluster sum of squares. In my project, I implemented an optimization function, `optimal_kmeans`, to determine the most effective number of clusters using the Kneed method. This method identifies the 'elbow point'—the point at which the rate of decrease in the within-cluster sum of squares (WCSS) shifts, indicating an optimal number of clusters for minimizing variance within clusters while maximizing variance between clusters.

- **WCSS Calculation:** I calculate the WCSS for a range of potential cluster numbers to find where WCSS begins to level off.

```
wcss = []
for i in range(1, max_clusters + 1):  # Limit the range to a maximum of 4
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10, random_state=42)
    kmeans.fit(data)
    wcss.append(kmeans.inertia_)
```

Figure 3.2.2.1 – WCSS Calculation

- **Elbow Point Detection:** The Kneed algorithm is used to detect the elbow point automatically.

```
# Automatically find the elbow point, limited to the range [1, max_trends]
kn = KneeLocator(range(1, max_clusters + 1), wcss, curve='convex', direction='decreasing')
n_clusters_optimal = kn.knee
```

Figure 3.2.2.2 – Elbow Point Detection

- **Optimal Clustering:** The data is then clustered using the optimal number of clusters determined.

```
# Perform K-means trending with specified number of trends
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
labels = kmeans.fit_predict(data)
```

Figure 3.2.2.3 – Apply Clustering

DBSCAN Clustering Implementation:

DBSCAN is advantageous for its ability to form clusters of arbitrary shapes and sizes, as it groups together closely packed points and marks points in low-density areas as outliers. The `optimal_dbscan` function enhances DBSCAN by automatically determining the optimal `eps` value using the nearest neighbour's method.

- **Nearest Neighbours Calculation:** I calculate the distance to the nearest neighbours to determine the `eps` parameter, which defines the maximum distance between two points for one to be considered in the neighbourhood of the other.

```
# Find the nearest neighbors
neighbors = NearestNeighbors(n_neighbors=2)
neighbors_fit = neighbors.fit(data)
distances, indices = neighbors_fit.kneighbors(data)
```

Figure 3.2.2.4 – Nearest Neighbours Calculation

- **Knee Detection for Optimal Eps:** The optimal `eps` is determined by finding the knee point in the sorted distances, suggesting a density threshold where points become part of the same neighbourhood.

```
# Automatically find the knee point as the optimal eps value
kn = KneeLocator(np.arange(len(distances)), distances, curve='convex', direction='increasing')
eps_optimal = distances[kn.knee]
```

Figure 3.2.2.5 – Knee Detection for Optimal Eps

- **Applying DBSCAN**: With the optimal eps value identified, DBSCAN is applied to the data.

```
# Perform DBSCAN trending with specified eps and min_samples values
dbscan = DBSCAN(eps=eps, min_samples=min_samples)
labels = dbscan.fit_predict(data)
```

Figure 3.2.2.6 – Applying DBSCAN

Integration and Application:

These clustering techniques, integrated through custom functions and optimized parameters, enable the efficient categorization of complex datasets. By implementing these algorithms, I have ensured that the clustering process is not only automated but also tailored to the specific characteristics of the dataset, leading to more meaningful and actionable insights.

This structured approach to clustering not only demonstrates the application of advanced data science techniques but also enhances the overall analytical capabilities of the project, enabling it to handle a wide variety of datasets effectively.

3.2.3 Generating AI Descriptions

As the project progresses towards the conclusion of the analysis pipeline, all necessary steps have been taken to identify trends within the dataset. What remains is to make sense of these trends—a crucial and exciting aspect of this project. The capability to generate detailed, AI-driven descriptions of data trends highlights a significant advancement in data analytics, made possible by recent developments in AI technology.

Process of Generating Descriptions:

The `generate_trend_descriptions` function is key in producing insightful summaries for each identified trend. This process is illustrated step-by-step, and specific screenshots from the code will help visualize the execution of these steps:

1. Initialization and Data Segmentation:
   - The dataset is first augmented with trend labels, segmenting it for focused analysis
   - Columns are classified as numeric or categorical, determining the specific analytical treatment for each type.

```python
def generate_trend_descriptions(data, trend_labels, numeric_metric='var', diff_metric='mean'):
    data['trend'] = trend_labels
    numeric_cols = data.select_dtypes(include=np.number).columns.tolist()
    categorical_cols = data.select_dtypes(exclude=np.number).columns.tolist()
    numeric_cols.remove('trend')
```

Figure 3.2.3.1 – Process Generating Descriptions

2. Statistical Analysis of Numeric Data:
  - Calculation of statistical metrics like variance or standard deviation identifies features with significant variations within each trend
  - The most variable features are highlighted in the summaries, with their average values included for contextual relevance.

```python
# Numeric: Based on specified metric
if numeric_metric == 'var':
    metric_values = trend_data[numeric_cols].var()
elif numeric_metric == 'std':
    metric_values = trend_data[numeric_cols].std()
else:
    raise ValueError("Unsupported numeric_metric provided.")

top_numeric = metric_values.nlargest(3)
for field in top_numeric.index:
    avg_val = trend_data[field].mean()
    standout_fields.append(f"{field} (average: {avg_val:.2f})")
```

Figure 3.2.3.2 – Statistical Analysis of Numeric Data

3. Analysis of Categorical Data:
  - The mode and its frequency within each trend are determined to identify dominant categories
  - The most significant categorical feature is highlighted based on its prevalence, adding depth to the narrative.

```python
# Categorical: Most significant based on frequency
if categorical_cols:
    cat_diffs = {}
    for col in categorical_cols:
        mode = trend_data[col].mode()[0] if not trend_data[col].mode().empty else 'N/A'
        mode_freq = trend_data[col].value_counts(normalize=True).get(mode, 0)
        cat_diffs[col] = mode_freq
    if cat_diffs:
        top_cat = max(cat_diffs, key=cat_diffs.get)
        top_mode = trend_data[top_cat].mode()[0] if not trend_data[top_cat].mode().empty else 'N/A'
        standout_fields.append(f"{top_cat} (most common: {top_mode})")
```

Figure 3.2.3.3 – Analysis of Categorical Data

4. Comparative Analysis Across Trends:
  - Numeric features are compared across trends to highlight unique characteristics
  - Categorical data is similarly analysed to showcase how dominant categories differ from those in other trends.

```python
# Section 2: Differentiation from Other trends
diff_desc = f"\nHow trend {trend_id+1} differs: "
diff_fields = []
for field in top_numeric.index:
    trend_avg = trend_data[field].mean() if diff_metric == 'mean' else trend_data[field].median()
    other_avg = other_trends_data[field].mean() if diff_metric == 'mean' else other_trends_data[field].median()
    difference = "higher" if trend_avg > other_avg else "lower"
    diff_val = abs(trend_avg - other_avg)
    diff_fields.append(f"{field} is {difference} than the average of other trends by {diff_val:.2f}")

if categorical_cols and cat_diffs:
    mode_freq_other_trends = other_trends_data[top_cat].value_counts(normalize=True).get(top_mode, 0)
    freq_diff = cat_diffs[top_cat] - mode_freq_other_trends
    freq_desc = "more common" if freq_diff > 0 else "less common"
    diff_fields.append(f"{top_mode} in {top_cat} is {freq_desc} compared to other trends")

diff_desc += "; ".join(diff_fields) + "."
```

Figure 3.2.3.4 – Comparative Analysis Across Trends

5. Narrative Construction:
  - Insights from both numeric and categorical analyses are combined into a narrative, providing a comprehensive summary of each trend.

```python
    # Combine both sections for the trend's description
    trend_description = standout_desc + diff_desc
    all_descriptions.append(trend_description)

return all_descriptions
```

Figure 3.2.3.5 – Narrative Construction

Examples of Description Generation:

To illustrate how the AI generates descriptions from incoming data and helps users interpret trends, I have put together the following examples:

1. Incoming Data:
  - A dataset includes multiple attributes like gender, race/ethnicity, parental level of education, lunch test preparation course, and scores in math, reading, and writing.

	gender	race/ethnicity	parental level of education	lunch	test preparation course	math score	reading score	writing score
0	female	group B	bachelor's degree	standard	none	72	72	74
1	female	group C	some college	standard	completed	69	90	88
2	female	group B	master's degree	standard	none	90	95	93
3	male	group A	associate's degree	free/reduced	none	47	57	44
4	male	group C	some college	standard	none	76	78	75
5	female	group B	associate's degree	standard	none	71	83	78
6	female	group B	some college	standard	completed	88	95	92
7	male	group B	some college	free/reduced	none	40	43	39
8	male	group D	high school	free/reduced	completed	64	64	67
9	female	group B	high school	free/reduced	none	38	60	50

Figure 3.2.3.6 – Example Dataset

2. Generate Trend Descriptions:
The function ‘generate_trend_descriptions’ takes a dataset, trend labels, and optional parameters defining the metrics for analysis (variance, standard deviation, mean, etc.). It first identifies standout fields within each trend based on numeric variability or standard deviation and the most common categorical values.

It then describes how each trend differs from others, quantitatively and categorically.

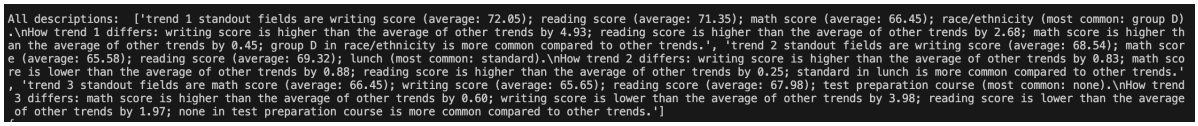


Figure 3.2.3.7 – Example Description Generation

3. AI-Generated Descriptions:
Once the data is analysed, the resulting descriptions are formatted into a structured JSON, which distinctly outlines each trend. This process allows the AI to systematically interpret and highlight key findings from the data. The use of JSON format ensures that the descriptions are not only precise but also easily readable and accessible. This structured approach helps in isolating and emphasizing significant data trends, making them clearer and more actionable for users. Each trend is encapsulated with detailed characteristics and a descriptive analysis, helping to bridge the gap between complex data patterns and practical insights. This format enhances interpretability, particularly for users who may not have extensive experience with data analysis, by presenting the information in a concise and organized manner.

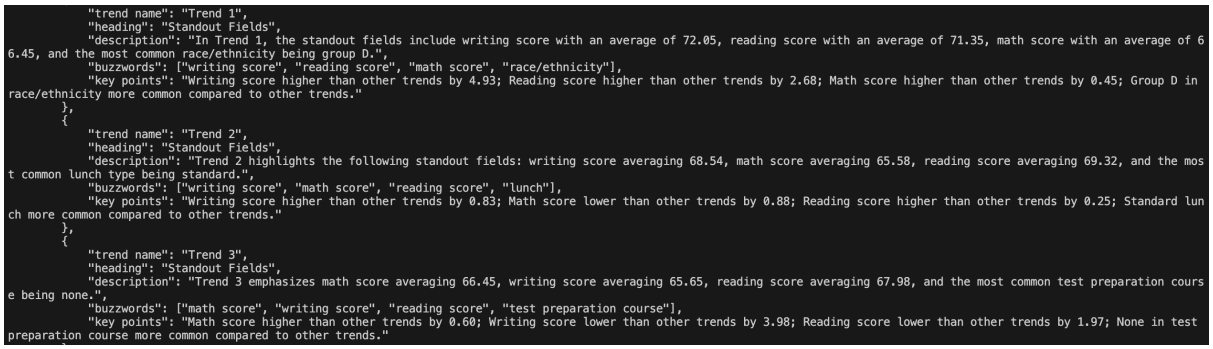


Figure 3.2.3.8 – Example AI Generated Descriptions

These descriptions demonstrate the AI's ability to extract and summarize key patterns from the data, making it easier for users, particularly those with less experience, to understand complex datasets and identify significant trends.

Rationale for AI-Driven Descriptions:

The use of AI-driven descriptions removes the skill required to interpret model trend data, by giving accessible and actionable insights. As datasets grow larger and more complex, traditional data presentation methods are insufficient. Automating the generation of descriptive summaries ensures that users can easily grasp the key insights, thereby enhancing decision-making processes.

The `generate_trend_descriptions` function demonstrates how advanced AI techniques can be integrated into data analysis to not only identify but also richly describe data trends. This methodology facilitates a deeper understanding of the data, transforming raw numbers into meaningful narratives. As AI technology evolves, such

tools will become increasingly essential in extracting valuable insights from complex datasets.

3.3 Visualization Technique

In this project, the choice of visualization libraries—Matplotlib and Altair (Vega-Lite)—was guided by their distinct strengths and suitability for different aspects of data presentation.

- Matplotlib: Renowned for its fine control over almost every element of a plot, Matplotlib excels in the creation of complex, publication-quality graphs, and visualizations. Its capability to generate detailed static images such as heatmaps is particularly valuable for representing correlations within the dataset.

- Altair (Vega-Lite): Known for its declarative syntax, Altair simplifies the creation of complex and interactive visualizations. Its integration with web technologies makes it ideal for creating interactive dashboards that allow users to explore data through visual narratives.

Recognizing that written summaries of data trends provide significant insights, I decided to supplement this with visual representations. Heatmaps, generated using Matplotlib, were particularly effective for this purpose. Heatmaps use colour gradients to represent data values, making them excellent for visualizing the strength and distribution of correlations across multiple variables simultaneously.

To further enrich user interaction the system automatically detects the data types upon dataset upload—Financial, Categorical, and Date/Time. This detection is crucial because it influences the type of visualization applied, ensuring that each data type is represented in the most informative way:

- Financial Data: For financial datasets, bar charts demonstrate the relationships between different financial metrics and categories, helping to uncover financial trends and distributions.

```python
# Aggregate the data
aggregated_data = (data.groupby(cat_col)[financial_col]
                   .sum()
                   .reset_index()
                   .sort_values(financial_col, ascending=False))

# Add 'Other' category if needed
if data[cat_col].nunique() > N:
    aggregated_data = aggregated_data.head(N)
    aggregated_data.loc[aggregated_data.index[-1], cat_col] = 'Other'
    data.loc[~data[cat_col].isin(aggregated_data[cat_col]), cat_col] = 'Other'
    aggregated_data.at[aggregated_data.index[-1], financial_col] = data.loc[data[cat_col] == 'Other', financial_col].sum()

# Create the chart title
chart_title = f'Sum of {financial_col} by {cat_col}'

# Decide between bar or pie chart based on the number of categories
chart = alt.Chart(aggregated_data)
if len(aggregated_data) <= N:
    chart = chart.mark_bar().encode(
        x=alt.X(f'{cat_col}:N', title=cat_col, sort='-y'),
        y=alt.Y(f'{financial_col}:Q', title=f'Sum of {financial_col}'),
        color=alt.Color(f'{cat_col}:N', legend=None),
        tooltip=[alt.Tooltip(f'{cat_col}:N'), alt.Tooltip(f'{financial_col}:Q')]
    )
else:
    chart = chart.mark_arc().encode(
        theta=alt.Theta(field=financial_col, type='quantitative'),
        color=alt.Color(field=cat_col, type='nominal'),
        tooltip=[alt.Tooltip(field=cat_col, type='nominal'), alt.Tooltip(field=financial_col, type='quantitative')]
    )

# Store the chart with the total sum
total_sum = aggregated_data[financial_col].sum()
charts_with_sums[chart_title] = (total_sum, chart.properties(title=chart_title))
```

Figure 3.3.1 – Financial Charts Code

- Categorical Data: Bar charts are also used to explore categorical data, mapping categories against numerical values to reveal underlying patterns.

```python
for col in categorical_cols:
    counts = data[col].value_counts().reset_index()
    counts.columns = [col, 'count']

    # Filter out categories with count less than min_count
    counts = counts[counts['count'] >= min_count]

    # Check the actual number of categories
    actual_categories_count = min(len(counts), N)
    title_text = f"Top {actual_categories_count} {col}" if actual_categories_count < N else f"All {col} Categories"

    # If there are more than N categories, include "Other"
    if len(counts) > N:
        top_counts = counts.head(N-1)
        other_count = counts['count'][N-1:].sum()
        top_counts = top_counts.append({col: 'Other', 'count': other_count}, ignore_index=True)
        top_counts = top_counts.sort_values(by='count', ascending=False)
    else:
        top_counts = counts

    if not top_counts.empty:
        chart = alt.Chart(top_counts).mark_bar(cornerRadius=3).encode(
            x=alt.X(f"{col}:N", sort=None),
            y=alt.Y('count:Q', title='Count'),
            color=alt.Color(f"{col}:N", scale=alt.Scale(scheme='category20'), legend=None),
            tooltip=[alt.Tooltip(f'{col}:N', title='Category'), alt.Tooltip('count:Q', title='Count')]
        ).properties(
            width=200,  # Control the width of individual charts
            height=200,  # Control the height of individual charts
            title=title_text
        )
        individual_charts.append(chart)
```

Figure 3.3.2 – Categorical Charts Code

-Date/Time Data: Line graphs depict how metrics evolve over time, providing insights into trends, seasonal variations, and more.

```
for time_col in time_date_cols:
    print ("data[time_col]", data[time_col])
    # Convert the time column to datetime if not already and drop NaT values
    data[time_col] = pd.to_datetime(data[time_col], errors='coerce')
    print ("dpost pd data[time_col]", data[time_col])
    valid_data = data.dropna(subset=[time_col]).copy()  # Copy to avoid SettingWithCopyWarning

    # Set the index to the time_col for resampling
    valid_data.set_index(time_col, inplace=True)
    valid_data.sort_index(inplace=True)

    aggregate = detect_aggregation_level(valid_data, time_col)
    print("Aggregate: ", aggregate)
    if aggregate == 'daily':
        valid_data = valid_data.resample('D').mean().reset_index()
    elif aggregate == 'weekly':
        valid_data = valid_data.resample('W').mean().reset_index()
    elif aggregate == 'monthly':
        valid_data = valid_data.resample('M').mean().reset_index()
    elif aggregate == 'yearly':
        valid_data = valid_data.resample('A').mean().reset_index()
    # Create line charts for each numerical column
    for num_col in numerical_cols:
        if pd.api.types.is_numeric_dtype(valid_data[num_col]):
            chart = alt.Chart(valid_data).mark_line(point=True).encode(
                x=alt.X(time_col, title='Date', axis=alt.Axis(labelAngle=-45)),
                y=alt.Y(num_col, title=num_col),
                tooltip=[alt.Tooltip(time_col, title='Date'), alt.Tooltip(num_col, title=num_col)]
            ).properties(
                title=f'{num_col} over time' if not title else title,
                width=300,  # Adjust the width to fit two charts per row
                height=150  # Adjust the height as needed
            )

            charts.append(chart)
```

Figure 3.3.3 – Time Series Charts Code

The `detect_aggregation_level` function determines the aggregation level of data based on the time intervals between consecutive dates in a specified column. Initially, the function verifies the presence of the specified time column; if it's not found as a DataFrame column, it resets the DataFrame's index to incorporate it. It then calculates the absolute differences in days between consecutive dates and uses these values to establish the aggregation level. The function categorizes the aggregation level as 'daily' if the maximum difference between dates is up to 1 day, 'weekly' if up to 7 days, 'monthly' if up to 31 days, and 'yearly' if the difference exceeds this. This method allows for dynamic assessment of time series data, facilitating appropriate aggregation for analysis based on the observed time intervals.

```python
def detect_aggregation_level(valid_data, time_col):
    # Make sure to work with a DataFrame where time_col is a column
    if time_col not in valid_data.columns:
        valid_data = valid_data.reset_index()  # Reset index if time_col is not found
    # Calculate the difference between consecutive dates
    date_diffs = valid_data[time_col].diff().dt.days.abs()

    # Define thresholds for daily, weekly, monthly, yearly continuity
    if date_diffs.max() <= 1:
        return 'daily'
    elif date_diffs.max() <= 7:
        return 'weekly'
    elif date_diffs.max() <= 31:
        return 'monthly'
    else:
        return 'yearly'
```

Figure 3.3.4 – Detect Aggregation Level Code

This automated, type-specific visualization approach ensures that users receive the most relevant and insightful visual representations of their data, tailored to the unique characteristics of their datasets.

The strategic use of Matplotlib and Altair caters to both the need for detailed, static visualizations and dynamic, interactive data exploration within the project. By leveraging these tools effectively, the project not only facilitates a deeper understanding of data through visual means but also enhances user engagement by allowing interactive data exploration. This dual approach ensures that the visualizations are not only informative but also engaging and accessible to a broad audience, thereby maximizing the impact of visual data insights.

3.3.1 Matplotlib

Strategic Use of Matplotlib for Heatmaps:

While Altair serves as the primary visualization library due to its interactive capabilities and ease of use in web environments, Matplotlib is selectively employed in your project for its superior functionality in generating heatmaps, particularly correlation heatmaps. This choice underscores a pragmatic approach to tool selection, optimizing the visualization outputs based on the strengths of each library.

Implementation of Heatmaps in Matplotlib:

Correlation heatmaps are a pivotal feature in your application, providing a visual representation of how variables in your dataset relate to each other. This is particularly valuable when presented alongside written summaries of trends, as it allows users to visually digest the relationships and correlations within the data quickly. Matplotlib's comprehensive toolkit for creating heatmaps includes customizable colour gradients and annotations, which enhance the utility and readability of these plots.

Here's a basic example of how I have implemented a correlation heatmap using Matplotlib in your project:

```
import matplotlib.pyplot as plt
import seaborn as sns

# Correlation heatmap
heatmap_fig, ax = plt.subplots(figsize=(10, 8))
sns.heatmap(trend_data.drop('trend', axis=1).corr(), annot=True, fmt=".2f", ax=ax, cmap="coolwarm")
plt.title(f"Feature Correlations in Trend {trend+1}")
```

Figure 3.3.1.1 – Correlation Heatmap

In this code snippet, `seaborn`, which is built on top of Matplotlib, is used for its enhanced visualization features, particularly its heatmap function that integrates well with Matplotlib's plotting capabilities. The `coolwarm` colour map provides a visually appealing and informative scale of correlation values, where warm colours indicate positive correlation, and cool colours denote negative correlation.

Rationale for Using Matplotlib Over Altair for Heatmaps:

The decision to use Matplotlib for heatmaps stems from Altair's current limitations in handling this specific type of plot efficiently, especially when dealing with larger datasets or when requiring detailed customization, such as in the case of correlation matrices. Matplotlib, with its low-level control over plot elements, provides the flexibility needed to produce detailed statistical visualizations.

Integrating Matplotlib with Project Components:

Although primarily used for heatmaps, the integration of Matplotlib into your project is handled to ensure consistency and seamless functionality within the Streamlit-based application. This involves configuring Matplotlib plots to match the aesthetic and interactive elements of Altair visualizations as closely as possible, ensuring a cohesive user experience.

Conclusion:

By leveraging Matplotlib's strengths in creating complex statistical visualizations like heatmaps, your project effectively complements the interactive visualizations created with Altair. This hybrid approach to using visualization libraries ensures that your application not only meets the functional requirements but also adheres to high standards of user experience and data presentation.

3.3.2 Vega-Lite (Altair)

My choice to use Altair for the majority of visualizations in this project is deeply rooted in the practical experience I gained during my data visualization modules. Altair's declarative nature allows for the creation of complex and interactive visualizations efficiently, making it an ideal tool for transforming data into engaging visual narratives. This capability is crucial for my project, which aims to present data in an informative and visually appealing way.

Core Usage in the Project:

Altair's ease of integration with Streamlit, which I use for the project's interactive dashboard, enhances its functionality within a web-based application. This integration is pivotal as it allows me to harness Altair's robust features—such as superior customization and sophisticated interactivity—within an accessible user interface.

Detailed Examples of Altair Implementations:

1. Interactive Scatter Plots: These plots allow for in-depth exploration of the relationships between variables, providing interactive means for users to examine and understand these relationships more profoundly.

```python
def create_scatter_plot(data, x_col, y_col):
    return alt.Chart(data).mark_circle(size=60).encode(
        x=alt.X(x_col, title=x_col),
        y=alt.Y(y_col, title=y_col),
        tooltip=[x_col, y_col]
    )

def create_scatter_plot_with_line(data, x_col, y_col):
    # Base chart for scatter points
    scatter_plot = alt.Chart(data).mark_circle(size=60).encode(
        alt.X(x_col, type='quantitative', title=x_col),
        alt.Y(y_col, type='quantitative', title=y_col),
        tooltip=[x_col, y_col]  # Tooltips on hover
    )

    # Regression line
    regression_line = scatter_plot.transform_regression(
        x_col, y_col, method="linear"
    ).mark_line(color='red')

    # Combine the scatter plot and the regression line
    final_chart = scatter_plot + regression_line

    return final_chart
```

Figure 3.3.2.1- Scatter Plots

I also created two scatter plot functions, and based on whether the user wants a regression line on the chart or not, the correct function will be called.

2. Dynamic Line Charts: Ideal for demonstrating changes over time, these charts make extensive use of Altair's interactivity, enhancing the user's ability to engage with temporal data trends effectively.

```
# Create line charts for each numerical column
for num_col in numerical_cols:
    if pd.api.types.is_numeric_dtype(valid_data[num_col]):
        chart = alt.Chart(valid_data).mark_line(point=True).encode(
            x=alt.X(time_col, title='Date', axis=alt.Axis(labelAngle=-45)),
            y=alt.Y(num_col, title=num_col),
            tooltip=[alt.Tooltip(time_col, title='Date'), alt.Tooltip(num_col, title=num_col)]
        ).properties(
            title=f'{num_col} over time' if not title else title,
            width=300,  # Adjust the width to fit two charts per row
            height=150  # Adjust the height as needed
        )

        charts.append(chart)
```

Figure 3.3.2.2 – Dynamic Line Charts

In the screenshot above, this is an example of me creating line charts for time series graphs.

Selective Interactive Features in Vega-Lite:

While Altair offers extensive interactive capabilities, such as zooming and panning, I have chosen to implement these features selectively. Based on the specific needs of the web page and the data presented, extensive interactivity such as zoom, and pan is not always necessary. Instead, focus is placed on tooltips, selections, and other forms of interaction that do not overwhelm the user experience but still provide significant insight and engagement.

Conclusion:

The familiarity with Altair I developed through my coursework has been instrumental in effectively employing its capabilities in this project. The visualizations created using Altair not only serve the functional requirements of the application but significantly enhance the user experience by making the exploration of data intuitive and engaging. This approach ensures that the application does more than just display data; it provides a comprehensive tool for interactive data exploration and analysis, enabling users to gain deeper insights and make more informed decisions.

3.3.3 Integration with Streamlit

Streamlit provides a robust platform for integrating and displaying visualizations dynamically within a web application. In my project, Streamlit's capabilities are leveraged to effectively showcase interactive data visualizations that are both engaging and informative.

Dynamic Visualization Integration:

1. Conditional Visualization Rendering:
  - The application uses Streamlit's sidebar options to let users select different visualization types dynamically. Based on the user's selection, corresponding visualizations are rendered in the main view. This flexibility allows users to interactively explore various aspects of the data.

```
if page == "Analysis Results":
    # Initialize the subpage list with default page(s)
    analysis_subpages = ["General Analysis", "Anomalies"]
```

Figure 3.3.3.1 Conditional Rendering

2. Visualization Functions:
  - Various visualization functions such as `plot_financial_barcharts`, `plot_categorical_barcharts`, and `plot_time_series_charts` are integrated into the Streamlit interface. These functions are called based on the analysis type selected by the user. This modular approach facilitates the easy addition or modification of visualizations without disrupting other parts of the application.

```
elif analysis_page == "Financial":
    # Display financial analysis results
    st.write('Financial analysis results:')
    st.caption('This section is dedicated to showcasing the financial aspects of the dataset.
    st.write("<br>", unsafe_allow_html=True)
    # Generate the financial chart using the stored DataFrame and column information
    financial_chart = plot_financial_barcharts(st.session_state['processed_data'],
                                               st.session_state.get('categorical_cols', []),
                                               st.session_state.get('financial_cols', []))
    st.altair_chart(financial_chart, use_container_width=True)
```

Figure 3.3.3.2 – Visualization Functions

3. Interactive Elements:
  - Streamlit seamlessly integrates with libraries like Altair to provide interactive visualizations that include features like tooltips, selections, and dynamic querying. These elements make the visualizations not only more engaging but also more insightful, as users can interact with the data directly to uncover deeper trends.

```
# Generate the categorical chart using the stored DataFrame and column information
categorical_chart = plot_categorical_barcharts(st.session_state['processed_data'],
                                               st.session_state.get('categorical_cols', []))
st.altair_chart(categorical_chart, use_container_width=True)
```

Figure 3.3.3.3 – Interactive Elements

Enhanced User Engagement:

Streamlit's interactivity enhances user engagement significantly. Users are not passive recipients of pre-rendered charts; instead, they interact with the data, choosing what to

view and how to view it. This interaction deepens the user's understanding and retention of the information presented.

Streamlined Interface for Insights:

By utilizing Streamlit's framework, I can ensure that the application is not only functional but also intuitive. The ability to dynamically control what is displayed, coupled with the ease of switching between different data views, allows the application to cater to a diverse audience, from data scientists to business stakeholders.

The integration of Streamlit in displaying visualizations has been pivotal in my project. It simplifies the presentation of complex datasets and enhances the accessibility of data-driven insights through interactive and visually appealing charts. This capability is crucial for a data visualization tool aimed at facilitating informed decision-making in a user-friendly environment. By harnessing the power of Streamlit, the project successfully bridges the gap between complex data analysis and practical, actionable insights.

3.4 Integration and System Architecture

Originally when designing my projects frontend and interactivity, I planned to use Flask to connect my backend and frontend, using React to run the frontend service and to write this in JavaScript, something I have used many times before but something I am not exceptional in. Frontend for me was a lower priority task and something I did not want to move onto and confuse myself with until the backend was at a proficient level. I therefore developed the backend to a very proficient level and when I got to the point where I had to start producing charts on a webpage to check compatibility etc, I then started working on my frontend. However, I very quickly realized I didn't want to put too much time into the frontend of my project as I wanted actual machine learning and the AI of the project to be the wow factor, aside from the visualizations. This is when I began researching again and I came across the dashboard service 'Streamlit' which could simplify the frontend experience and seamlessly integrate into my program.

3.4.1 Streamlit

During the initial phase of researching frontend services for my dissertation project, I discovered Streamlit. It quickly became evident that Streamlit would integrate seamlessly with the Python-based backend of my project. Installation was straightforward—requiring only the addition of the Streamlit library to my environment—which allowed me to dive directly into learning and implementing this tool in my application.

Streamlit's appeal lies in its ability to turn data scripts into shareable web apps quickly. This feature was crucial for my project as it allowed for the dynamic presentation of data visualizations and interactive elements without the need for complex web development skills. Here's how I incorporated Streamlit into the project:

```
# Process Data button
if st.button('Process Data'):
    if uploaded_file is not None:
        with st.spinner('Processing... Please wait'):
```

Figure 3.4.1.1 – Processing Data button and spinner

Dynamic Visualization Rendering:

Streamlit's powerful session state management allows for maintaining state across user interactions. By utilizing `st.session_state`, I could keep track of processed data and control the visibility of visualizations based on user inputs:

```
# Initialize session state for processed and normalized data
if 'processed_data' not in st.session_state:
    st.session_state['processed_data'] = None
if 'show_visualizations' not in st.session_state:
    st.session_state['show_visualizations'] = False
```

Figure 3.4.1.2 – Session State handling

This approach enables the application to respond interactively to user inputs without re-running the entire script from the beginning, thereby enhancing performance and user experience.

Interactive Analysis and User Interface:

Streamlit's sidebar and main interface components were used to offer users the ability to dynamically select different types of data analyses and visualizations:

```
page = st.sidebar.selectbox('Select a page', page_options, key='page_selection')

if page == "Analysis Results":
    # Initialize the subpage list with default page(s)
    analysis_subpages = ["General Analysis", "Anomalies"]

    # Check if 'financial_cols' is defined and has entries
    if len(st.session_state.get('financial_cols', [])) > 0:
        analysis_subpages.append("Financial")

    # Check if 'categorical_cols' is defined and has entries
    if len(st.session_state.get('categorical_cols', [])) > 0:
        analysis_subpages.append("Categorical")

    # Check if 'time_date_cols' is defined and has entries
    if len(st.session_state.get('time_date_cols', [])) > 0:
        analysis_subpages.append("Time Series")
```

Figure 3.4.1.3 – Interactive Analysis and User Interface

Each selection triggers specific functions that generate visualizations appropriate for the data type, such as financial bar charts, categorical comparisons, or time series graphs. This modularity makes it easy to maintain and update different parts of the application without affecting others.

Dynamic Data Exploration:

A significant part of this project's frontend functionality is enabled through Streamlit, particularly the 'Explore Data' section. This interactive environment allows users to engage directly with their data, crafting custom visualizations on-the-spot. Users can dynamically select data dimensions for the x and y axes, enabling them to generate various charts and graphs, such as bar plots, scatter plots, histograms, and box plots. This feature significantly enhances user experience by providing flexibility and immediate feedback, ideal for exploratory data analysis.
Here's how this is implemented:

```
elif page == 'Explore Data':
    st.write('Explore Data')
    # Sidebar options for chart selection
    st.sidebar.header("Charts and Plots")
    plot_options = ["Bar plot", "Scatter plot", "Histogram", "Box plot"]
    selected_plot = st.sidebar.selectbox("Choose a plot type", plot_options, key='selected_plot')

    # 'processed_data' has the data to plot
    data = st.session_state['processed_data']


    # Only create charts if data is available
    if data is not None and selected_plot:
        column_options = data.columns.tolist()
```

Figure 3.4.1.4 – Explore Data Page Selection

```python
if selected_plot == "Bar plot":
    x_axis = st.sidebar.selectbox("Select category axis", column_options, key='x_axis')
    y_axis = st.sidebar.selectbox("Select value axis", column_options, key='y_axis')
    chart = alt.Chart(data).mark_bar().encode(
        x=x_axis,
        y=y_axis
    )
    chart = configure_chart(chart)
    st.altair_chart(chart, use_container_width=True)
    download_chart(chart, "bar_plot")

elif selected_plot == "Scatter plot":
    x_axis = st.sidebar.selectbox("Select x-axis", column_options, key='x_axis_scatter')
    y_axis = st.sidebar.selectbox("Select y-axis", column_options, key='y_axis_scatter')
    show_regression = st.sidebar.checkbox("Show Regression Line", value=False)
    if show_regression:
        chart = create_scatter_plot_with_line(data, x_axis, y_axis)
    else:
        chart = create_scatter_plot(data, x_axis, y_axis)
    chart = configure_chart(chart)
    st.altair_chart(chart, use_container_width=True)
    download_chart(chart, "scatter_plot")

elif selected_plot == "Histogram":
    column = st.sidebar.selectbox("Select a column for histogram", column_options, key='hist_column')
    bins = st.sidebar.slider("Number of bins", min_value=1, max_value=100, value=30, key='hist_bins')
    chart = alt.Chart(data).mark_bar().encode(
        alt.X(column, bin=alt.Bin(maxbins=bins)),
        y='count()'
    )
    chart = configure_chart(chart)
    st.altair_chart(chart, use_container_width=True)
    download_chart(chart, "histogram")

elif selected_plot == "Box plot":
    x_axis = st.sidebar.selectbox("Select category axis", column_options, key='x_axis_box')
    y_axis = st.sidebar.selectbox("Select value axis", column_options, key='y_axis_box')

    # Ensure the y-axis is treated as a quantitative variable
    if data[y_axis].dtype not in ['float64', 'int64']:
        st.warning(f"The selected value axis '{y_axis}' is not continuous.")
    else:
        chart = alt.Chart(data).mark_boxplot().encode(
            x=x_axis,
            y=y_axis
    )
```

Figure 3.4.1.5 – Chart Selection Explore Data

In the 'Explore Data' section, Streamlit not only facilitates real-time interaction with data but also extends the capability to download these custom charts as HTML, preserving the interactivity of the visualizations. This adaptability ensures that users can explore and manipulate their data in myriad ways directly within the web application, fostering a deeper understanding and more creative analysis of the data.

Conclusion:

The integration of Streamlit into my project not only simplified the development of a user-friendly web application but also enriched the project's capacity to deliver interactive data visualizations. Streamlit bridged the gap between complex data

processing backends and an accessible frontend, allowing users to explore and interact with their data in a meaningful way. This setup ensures that the insights generated are not only based on sophisticated data analysis but are also presented in a way that is intuitive and actionable for users from various backgrounds.

3.4.2 OpenAI

The integration of OpenAI's API into this project serves as the capstone to the analytical process, transforming raw data insights into comprehensible summaries that enhance user understanding. The key steps involved in this integration highlight the seamless fusion of complex data analytics with advanced AI-driven text generation, ensuring that users receive not only data but also meaningful interpretations.

Setup and Implementation:

The initial setup involved configuring the OpenAI API within the project environment, facilitated by securely storing the API key in a `.env` file to ensure security and ease of access. This setup allows the project to communicate with OpenAI's powerful models, which are instrumental in generating user-friendly summaries of data trends.

```
load_dotenv()

def generate_summary(descriptions, retry_lmit=3):
  client = OpenAI(
    api_key=os.environ.get("OPENAI_API_KEY"),
  )
  new_descriptions = ', '.join(descriptions)
```

Figure 3.4.2.1 - OpenAI Setup

This code snippet ensures that the API key is loaded securely, maintaining the integrity of the application's data processing and communication with the OpenAI servers.

Data Handling and API Interaction:

To adhere to OpenAI's API usage constraints, such as the rate limit of 3 requests per minute, the project was designed to bundle trend descriptions into a single API call. This approach not only respects the API's usage policy but also optimizes the interaction process, reducing latency and potential timeouts.

```
attempts = 0
while attempts < retry_lmit:
  response = client.chat.completions.create(
      model="gpt-3.5-turbo-0125",
      max_tokens=4096,
      temperature=0.7,
      messages=[
          {"role": "system", "content": "Your task is to generate a JSON structure containing trends of data.
          {"role": "user", "content": new_descriptions},
      ],
  )
```

Figure 3.4.2.2 – API Call

This implementation demonstrates the efficient use of API calls by sending all trend descriptions in a single request, which is then processed by the AI to return a structured JSON response containing all summaries.

Response Handling and Error Management:

The project includes robust error handling mechanisms to manage and rectify any issues that arise during the interaction with the AI, such as response parsing errors or the absence of expected keywords. This ensures reliability and consistency in the AI-generated summaries, providing a fallback mechanism to request a new summary if the initial attempt fails.

```
    try:
        data = json.loads(response.choices[0].message.content)
    except json.JSONDecodeError as e:
        print(f"JSON decoding failed: {e}")
        attempts += 1
        time.sleep(20)  # Wait a bit before retrying
        continue
```

Figure 3.4.2.4 – OpenAI Error Handling

This snippet demonstrates the project's capability to handle JSON parsing errors gracefully, attempting retries as needed to ensure that valid and useful summaries are generated.

Extracting and Structuring Data:

Upon reception, the JSON data is thoroughly parsed to isolate key elements such as headings, descriptions, buzzwords, and detailed points for each trend. This information is then formatted for web presentation:
Python

```
if isinstance(data.get('trends'), list):
    for trend in data['trends']:
        heading = trend.get('heading', 'No Heading')
        description = trend.get('description', 'No Description')
        buzzwords = trend.get('buzzwords', 'No Buzzwords')
        key_points_str = trend.get('key points', '')
        key_points_list = key_points_str.split('; ')

        key_points_html = '<ul>' + ''.join(f'<li>{point}</li>' for point in key_points_list) + '</ul>'
```

Figure 3.4.2.5 – Extracting and Structuring Data

This function ensures each trend summary is clearly presented, with distinct sections for easy reading and comprehension.

Handling and Displaying Content:

The extracted data is then incorporated into the project's interface using HTML to enhance readability and visual appeal. The structured format ensures that each aspect of the trend summary is accessible, allowing users to quickly grasp the essential insights conveyed by the AI.

Conclusion:

The integration of OpenAI's API into the project is pivotal for translating complex data analytics into accessible summaries. This not only makes the insights more actionable for users but also showcases the potential of combining advanced AI technologies with data analytics platforms. The careful handling of API requests and responses ensures that the project remains efficient and user-centric, providing a seamless experience from data upload to insight generation.

3.5 Technical Setup and Deployment

Initially, the tool is deployed locally to facilitate rapid development and immediate feedback, essential in the early stages of testing. This approach allows for quick iterations based on direct observations and user feedback.

Local Server Configuration:

For local testing, the tool is hosted via Streamlit, which allows Python scripts to be easily converted into interactive web applications. Here’s a concise guide on how the local deployment is handled:

1. Environment Setup: Installation of Python and necessary libraries such as Streamlit, Altair, and Matplotlib using pip, with a `requirements.txt` file to manage dependencies.
2. Running the Application: The tool is launched locally by executing `streamlit run app.py` in the terminal, where `app.py` is the main script containing the Streamlit application.

3. Accessing the Tool: Once the server is active, the tool can be accessed through a web browser at the default address `localhost:8501`.

Cloud Deployment Considerations:

While the initial focus is on local deployment for testing, future considerations include moving the application to a cloud platform to improve accessibility and scalability. This transition would allow for broader user access and potentially better handling of larger datasets.

3.5.1 Hardware and Software Requirements

Hardware Requirements:
- A standard computer with at least 8GB of RAM and a modern multi-core processor to efficiently handle the computational needs of the tool.

Software Requirements:
- Operating System: Compatible across Windows, macOS, and Linux.
- Python Version: Python 3.6 or higher.
- Libraries: Key dependencies include Streamlit, Pandas, Altair, Matplotlib, and Seaborn, specified in a `requirements.txt` file for easy setup.
- Browser: Modern web browsers such as Google Chrome, Firefox, Safari, or Edge are required to access the web application.

User Testing Approach:

The initial phase of user testing involves conducting one-to-one sessions where participants interact with the tool hosted on a local webpage. This setting allows me to directly observe user interactions and gather valuable qualitative feedback. Tasks will be set for users, and their experience and any issues encountered will be noted in real-time, providing insights critical for refining the user interface and overall functionality of the tool.

Conclusion:

This section outlines the technical and deployment strategies utilized for the project, emphasizing local testing and future considerations for cloud deployment. User testing protocols are designed to ensure the tool is user-friendly and meets the practical needs of its intended audience. These steps are crucial for preparing the tool for wider deployment and ensuring it provides valuable, actionable insights to users. Further exploration of cloud deployment will be discussed in the "Future Work" section of the conclusion, considering how to expand the tool’s reach and capabilities.

3.5.2 GitHub

Version control is indispensable in software development, serving as a backbone for managing project iterations and historical versions. GitHub, renowned for its robust version control capabilities, plays a crucial role in this project, particularly through its integration with Visual Studio, which enhances my development workflow.[15]

GitHub Integration with Visual Studio:

Using GitHub directly within Visual Studio streamlines my development process significantly. This integration allows me to leverage GitHub's version control capabilities without leaving the development environment. Here's how I utilize this setup:

- Committing Changes: Changes are made within Visual Studio and can be committed directly from the IDE. This process bypasses the need for command-line interactions, making version control more accessible and less error-prone.
- Pushing Updates: After committing changes, I can push them to the GitHub repository directly from Visual Studio. This integration ensures that updates are synchronized with the remote repository without switching tools.

Advantages of Branching with Visual Studio:

Branching is a vital feature of GitHub that I extensively use to manage different development streams like new features or bug fixes. Visual Studio's GitHub integration simplifies managing branches:

- Branch Creation and Switching: I can create and switch branches directly within Visual Studio. This allows for seamless context switching between different development focuses without manual commands.
- Merging and Pull Requests: When a feature is ready, I use Visual Studio to initiate pull requests and merge branches. This process includes automatic conflict resolution tools, simplifying the integration of changes.

Collaborative Benefits and Code Reviews:

GitHub's pull request feature is particularly beneficial for collaborative reviews, which I handle through Visual Studio. This method allows for inline commenting and discussions, enhancing the quality of code reviews and fostering a collaborative development environment.

Integrating GitHub with Visual Studio provides a powerful, streamlined toolset that supports efficient version control and collaborative development directly within my primary development environment. This setup not only keeps the development process organized but also enhances productivity by minimizing the need to switch contexts between coding and version control tasks. Through this integrated approach, the project benefits from robust version management while maintaining a focus on developing high-quality software.

3.5.3 Visual Studio and GitHub Copilot Integration

In the development of this project, the integration of GitHub Copilot with Visual Studio has been instrumental. GitHub Copilot is an AI-powered code completion tool that significantly enhances coding efficiency by suggesting whole lines or blocks of code as you type, based on the context provided by existing code.

Key Features of GitHub Copilot:

- Context-Aware Code Suggestions: GitHub Copilot analyses the code you're working on and provides suggestions for the next lines of code, including complex algorithms and API calls, which can dramatically speed up the development process.
- Learning from Comments: The tool can generate code based on comments describing the intended functionality, making it easier to translate ideas directly into executable code.
- Support for Multiple Languages and Frameworks: Copilot is designed to work with a multitude of programming languages and frameworks, which aligns well with the diverse technologies used in modern software projects, including those handled in Visual Studio.

Practical Use of GitHub Copilot in the Project:

During the development of this project, GitHub Copilot has served as a virtual pair programmer, offering suggestions, and automatically filling in repetitive or boilerplate code, which has allowed me to focus more on solving unique problems and less on mundane coding tasks. Here's how Copilot has been integrated into the workflow:

1. Code Completion: As I type, Copilot suggests complete lines or blocks of code, which I can accept, modify, or reject. This functionality is especially helpful for quickly building out new features without constantly referencing documentation.

2. Learning from Existing Code: Copilot adapts to the codebase and follows established patterns and practices in the project, ensuring that suggestions are consistent with the project's coding standards.

3. Efficiency in Coding: By reducing the amount of code I need to write manually; Copilot helps decrease development time and increases the focus on strategic problem-solving.

Advantages for Project Development:

- Increased Development Speed: Suggestions by Copilot are based on a vast repository of open-source code, which means that it often suggests optimal or innovative solutions that I might not have considered.
- Enhanced Learning: Working with Copilot has also been educational, exposing me to new libraries and coding patterns that enrich my development skills.

- Reduced Coding Errors: The tool helps catch syntactical errors and offers corrections, which improves the overall code quality and reduces the time spent on debugging.

Drawbacks of Using GitHub Copilot:

While GitHub Copilot speeds up development and aids learning, its limitations must be acknowledged. Occasionally, it suggests incorrect or suboptimal code, requiring users to carefully review each suggestion rather than accepting them outright. Furthermore, in complex coding scenarios, Copilot may repetitively suggest ineffective solutions, exposing its limitations in handling advanced projects. These experiences highlight the importance of using Copilot as a supportive tool rather than a substitute for human expertise, encouraging developers to leverage its capabilities while remaining aware of its potential pitfalls.

The use of GitHub Copilot integrated within Visual Studio has significantly enhanced the development process for this project. This setup not only boosts productivity but also ensures that the code is robust and aligns with current programming best practices. The seamless integration of these advanced tools underscores my commitment to leveraging cutting-edge technologies to build a sophisticated, efficient, and user-friendly data visualization tool.

4. Testing

4.1 Testing Plan

Testing plays a crucial role in ensuring the reliability and usability of software projects. For this project, a comprehensive testing strategy was employed, encompassing usability testing, functional testing, and performance testing. Each of these testing methodologies targets specific aspects of the software to ensure it meets both the technical specifications and the user's needs.

4.1.1 Test Methodology

1. Usability Testing:
   - Objective: To observe real users interacting with the system to evaluate the user interface and overall user experience.
   - Method: Tasks were given to users of two different abilities to perform specific actions on the platform. Observations and feedback were collected to identify any usability issues or improvements.

2. Functional Testing:
   - Objective: To verify that all features function according to the specified requirements.
   - Method: Systematic tests were conducted to ensure all functionalities, such as data upload, visualization rendering, and interaction with AI-generated summaries, work as intended across different scenarios.

3. Performance Testing:
  - Objective: To evaluate the system's performance under various conditions to ensure reliability and efficiency.
  - Method: Performance metrics such as response time, speed, and resource usage were measured under load. This helped in identifying performance bottlenecks and areas for optimization.

4.1.2 Implementation of Testing

- Usability Testing: This was conducted with two users of differing expertise, monitored in a controlled setting to evaluate the application's ease of use and accessibility. This approach helped identify user interface challenges and validate the system's design across varying levels of user experience.
- Functional Testing: Each function of the system was tested individually and then in combination with others. This included testing for data handling, accuracy of visualizations, and the responsiveness of the Streamlit interface.
- Performance Testing: Tests were conducted to assess how well the application performs when subjected to large datasets and simultaneous user requests. Tools like JMeter or LoadRunner were potentially utilized to simulate various operational loads.

The structured approach to testing ensured that the application was rigorously vetted before deployment. Usability improvements were made based on direct user feedback, functionality was validated against requirements, and performance enhancements were applied to ensure the application could handle real-world use cases effectively. This thorough testing process underscores the project's commitment to quality and user satisfaction.

4.2 Testing Results

4.2.1 Usability Testing Results

User 1 is relatively new to using data visualization tools and lacks extensive experience with advanced data analytics platforms. They are interested in learning how to interpret data visualizations effectively and are keen on improving their data literacy skills. User 1's interactions with the software primarily focus on basic functionalities, and they may require more intuitive design features and additional guidance to navigate through complex data manipulation tasks efficiently.

Appendix 1 – User 1 Usability Test Results
User 2, on the other hand, comes with a more robust background in data science and is familiar with various data visualization techniques and software. Their experience allows them to navigate the tool with greater ease and confidence. User 2 seeks advanced features that can provide deeper insights and allow for more complex data manipulations. They are better equipped to utilize sophisticated tools without

significant guidance, appreciating features that offer extensive customization and detailed analytical capabilities.

Appendix 2 – User 2 Usability Test Results

Observations and Recommendations:

User Differences: User 01 encountered various challenges, particularly with navigation and understanding of the application's features, likely due to their lower experience level. User 02, with more experience, completed tasks more swiftly and with higher satisfaction.

Common Issues:

Navigation and Clarity: User 01 had trouble with navigating to specific analytics features and understanding AI-generated content.
Tool Accessibility: Difficulty in locating and using visualization tools was a major hurdle for User 01.

4.2.2 Functional Testing Results

Test Case 1: Data Upload
Objective: Ensure the system accepts and correctly handles file uploads.

Test ID	Test	Expected Result	Result
TC1.1	Upload valid file formats (CSV, XLSX, JSON, TXT) and verify successful upload.	File type will be accepted.	Expected Result.
TC1.2	Attempt to upload unsupported file formats and expect rejection.	File will be rejected.	The file was rejected, the user was prompted to upload a supported file type.
TC1.3	Upload files with maximum and exceeding maximum allowed sizes to test size restrictions.	Maximum will be allowed, not exceeding maximum.	When maximum size (200mb) was uploaded It was accepted however when a file exceeding the maximum was uploaded it was rejected.
TC1.4	Upload files with no data in.	It will upload however when they click to process the data it will be rejected.	The upload was accepted but when the user attempted to process the data is raised an error saying

			file must contain content.

Test Case 2: Data Processing
Objective: Validate data processing logic including transformations, imputations, and handling of diverse datasets.

Test ID	Test	Expected Result	Result
TC2.1	Process datasets with missing values and without selecting 'Impute missing values'.	Missing rows will be dropped.	Expected Result.
TC2.2	Process datasets with missing values and select 'impute missing values'	All missing rows will have a value imputed based on the mean, median and mode of other values in the column.	Expected Result.
TC2.3	Process datasets that are perfectly clean to verify processing without imputation.	All data was accepted, and no imputations made.	Expected Result.
TC2.4	Verify handling of corrupt data entries during processing.		
TC2.5	Test data transformation logic for accuracy across different datasets.		

Test Case 3: Visualization Generation
Objective: Test the generation of various charts and visualizations based on user selections and data inputs.

Test ID	Test	Expected Result	Result
TC3.1	Generate each type of chart (Bar, Scatter, Box, Histogram) and verify correct data representation.	All graphs will be produced correctly with the correct data and chart type.	Expected Result
TC3.2	Input datasets with outlier values to see how they are visualized.	The anomalies section will identify these visually using a histogram displaying the upper and lower	Expected Result

		quartile and any values outside this.	
TC3.3	Test responsiveness of visualizations to different data scales.	The graphs intervals will dynamically adjust to the size of the data to ensure visually pleasing data extraction.	Expected Result
TC3.4	Validate chart rendering with real-time data updates.	Charts will update in real time if the X or Y axis are changed or if the user adds a regression line.	Expected Result
TC3.5	Ensure visualizations maintain integrity with extremely large datasets.	Charts will only show the top 20 of results otherwise the rest of the data will be put in an 'other' section to keep the integrity of the visualizations.	Expected Result.
TC3.6	Test 'Download as HTML' on all Altair charts to ensure all download correctly.	The user will click the 'Download as HTML' button and they will receive a download of the chosen chart keeping the properties and interactivity of the chart.	Expected Result User also has the option to download the chart as a PNG once it's downloaded as a HTML.
TC3.7	Test 'Generate PDF Report' and 'Download Now' buttons to download the general analysis as a PDF report.	The user will click 'Generate PDF report' and a 'Download Now' button will appear so they can download the PDF report.	Expected Result The Bold and Italic styles on text is lost.

Test Case 4: Streamlit Interface Interaction
Objective: Ensure the user interface components function as expected when interacted with.

Test ID	**Test**	**Expected Result**	**Result**
TC4.1	Use all UI controls (sliders, buttons, checkboxes, etc.) to modify inputs and verify changes affect outputs appropriately.	All UI controls will act accordingly with no errors or freezes.	Expected Result

TC4.2	Navigate between different pages or sections to ensure UI consistency and state management.	All pages will load on first command ensuring none of the data from other pages are displayed.	Sometimes if there is a Java error with a chart the error message will appear on all screens and the application must be restarted.
TC4.3	Verify session state is preserved during user interaction.	Ensure that data is not reloaded when changing page for example when moving off the general analysis page, and back the AI descriptions are not re-generated.	Expected Result.
TC4.4	Confirm error messages are displayed properly when an exception occurs in the UI.	When an error message is passed into the terminal, this error or warning message is displayed on the UI to the user.	Expected Result
TC4.5	Test all help tooltips if they display the help message when hovered over.	When a user hovers their mouse over a help box '?' the correct help message will appear with guidance.	Expected Result

Test Case 5: OpenAI API Integration
Objective: Confirm that the application correctly communicates with the OpenAI API, handling responses and errors.

Test ID	Test	Expected Result	Result
TC5.1	Test successful API calls and correct response handling.	The AI is passed the generated descriptions, and it will pass back the responses as a JSON so they can be split into separate responses.	Expected Result
TC5.2	Simulate API failures (e.g., network issues, invalid responses) to test error handling and user notifications.	If an API fails the message will try to resend 3 times, if it doesn't work after the third try an error message will appear prompting the user to	Expected Result

		restart the application and try again.	
TC5.3	Verify API security measures (e.g., authentication, data encryption).	Ensure API key is kept in an .env file and this is not included in GitHub commits.	Expected Result
TC5.4	Check API response times and performance under load.	Ensure no more than 3 requests are submitted a minuet by adding timeouts between requests.	Expected Result Timeout of 20 seconds per request to ensure no more than 3 a minuet

4.2.3 Performance Testing

In this section, the performance testing methodology focuses on rigorously evaluating the system's performance under various conditions, ensuring it meets the high standards required for reliability and efficiency in real-world applications. The performance testing approach involved systematically measuring critical performance metrics such as response time, speed, and which functions are used across different scenarios and loads.

For this test I have prepared two databases of different sizes with 20% of the data missing at random to test how the size of the file can affect the performance.

System Specifications for Performance Testing:

The performance tests were conducted on a system equipped with the following specifications to ensure a reliable and consistent testing environment:

- Device: Apple MacBook Pro
- Chip: Apple M1 Pro
- Operating System: macOS Sonoma 14.2.1
- Memory: 16GB

Performance Test 1

In this initial test, we will evaluate the system's efficiency and responsiveness using a relatively small file, approximately 60KB in size. This dataset has been purposefully selected to include missing values which will be identified and dropped as part of the test process. This will help us understand the system's capability in handling data cleaning tasks with smaller datasets, ensuring that basic operations such as loading, processing, and data manipulation are executed smoothly and efficiently. This test is checking the performance when producing the general analysis.

Dropping Missing Values:

function calls (3464801 primitive calls) in 18.968 seconds

Ncalls	Percall(s)	Function
1	18.054	main.py:10(process_file)
1	17.992	model_manager.py:22(complete_analysis_pipeline)
1	14.642	OpenAI.py:12(generate_summary)
1	14.620	utils/_utils.py:243(wrapper)
1	14.620	openai/resources/chat/completions.py:615(create)
1	14.616	openai/_base_client.py:1199(post)
1	14.616	openai/_base_client.py:893(request)
1	14.616	openai/_base_client.py:910(_request)
1	14.614	httpx/_client.py:881(send)
1	14.610	httpx/_client.py:930(_send_handling_auth)

Now I will perform the same test with the same small file. However, this time I will be imputing the missing values to evaluate the effect this has on the performance.

Imputing Missing Values:
function calls (510169896 primitive calls) in 287.305 seconds **(4 Minute and 47 Seconds)**

Ncalls	Percall(s)	Function
1	286.443	main.py:10(process_file)
1	272.843	data_cleaner.py:16(preprocess_data)
1	272.791	data_cleaner.py:88(handle_missing_values_with_tpot)
1	272.790	data_cleaner.py:232(predictive_imputation)
1	272.754	tpot/base.py:688(fit)
1	269.912	utils.py:138(wrapper)
1	264.542	tpot/gp_deap.py:177(eaMuPlusLambda)
1	258.952	sklearn/base.py:1457(wrapper)
1	255.628	sklearn/pipeline.py:423(fit)
1	238.587	tpot/base.py:1475(_evaluate_individuals)

Performance Test 1 Results Overview:

Dropping Missing Values:
In the performance test for dropping missing values, the system handled 3,464,801 primitive calls in a total of 18.968 seconds. The analysis of the function calls reveals:
- The `process_file` function consumed the most time at 18.054 seconds, followed closely by the `complete_analysis_pipeline` at 17.992 seconds, indicating substantial time dedicated to initial data processing and management.
- A significant amount of time was also spent on functions associated with the OpenAI API, including various network-related tasks such as requests and responses, all of

which consistently required about 14.6 seconds. This highlights the dependency on external API calls which formed a major part of the operation time.

Imputing Missing Values:
For the imputation of missing values, the function call count escalated dramatically to 510,169,896, with the entire process taking 287.305 seconds (approximately 4 minutes and 47 seconds). The key observations include:
- The `main.py:10(process_file)` function again dominated the runtime at 286.443 seconds, suggesting intensive computational efforts are involved in managing and processing the file with imputation techniques.
- The `data_cleaner.py` script's functions (`preprocess_data`, `handle_missing_values_with_tpot`, and `predictive_imputation`) all recorded similar and substantial times, indicating a heavy load due to the complexity of imputing data using TPOT, an automated machine learning tool.
- The use of TPOT for data imputation, especially through genetic programming approaches (`eaMuPlusLambda`, `fit`, `_evaluate_individuals`), significantly increased the computational burden, as seen in the extended processing times for these functions.

Performance Test 2

Performance Test 2 will assess the application's capability to efficiently handle a larger dataset, approximately 517KB, focusing on two primary data cleaning processes: dropping missing values and imputing them. This test aims to record and analyse the performance impact of these methods on a significantly bigger file size compared to previous tests. The outcomes will provide valuable insights into the system's scalability and responsiveness, highlighting how well it manages more extensive data operations which are crucial for real-world applications.

Dropping Missing Values:
function calls (16788886 primitive calls) in 19.164 seconds

Ncalls	Percall(s)	Function
1	18.203	main.py:10(process_file)
1	18.074	model_manager.py:22(complete_analysis_pipeline)
1	7.111	OpenAI.py:12(generate_summary)
1	7.089	utils/_utils.py:243(wrapper)
1	7.089	openai/resources/chat/completions.py:615(create)
1	7.085	openai/_base_client.py:1199(post)
1	7.085	openai/_base_client.py:893(request)
1	7.085	openai/_base_client.py:910(_request)
1	7.049	httpx/_client.py:881(send)
1	7.034	httpx/_client.py:930(_send_handling_auth)

Imputing Missing Values:

function calls (1644030374 primitive calls) in 4108.539 seconds **(68 Minute and 28 Seconds)**

Ncalls	Percall(s)	Function
1	4107.225	main.py:10(process_file)
1	4061.230	data_cleaner.py:16(preprocess_data)
1	4061.142	data_cleaner.py:88(handle_missing_values_with_tpot)
1	580.163	data_cleaner.py:232(predictive_imputation)
1	580.120	tpot/base.py:688(fit)
1	1.350	utils.py:138(wrapper)
1	576.721	tpot/gp_deap.py:177(eaMuPlusLambda)
1	0.474	sklearn/base.py:1457(wrapper)
1	141.969	base.py:1475(_evaluate_individuals)
1	2.969	gp_deap.py:393(_wrapped_cross_val_score)

Performance Test 2 Results Overview

Dropping Missing Values:
The performance test for dropping missing values involved a total of 16,788,886 primitive calls, completing the process in 19.164 seconds. The key functions and their execution times indicate a relatively efficient process:
- The `process_file` function and `complete_analysis_pipeline` were the most time-consuming, each taking over 18 seconds. This suggests that the bulk of the time was spent processing the file and managing the data cleaning pipeline at a high level.
- API-related functions, notably those from OpenAI and HTTPX, also recorded significant times, all hovering around 7 seconds. These functions dealt primarily with network communications and external API interactions, which are essential for the integration but less related to the core data processing tasks.

Imputing Missing Values:
The imputation of missing values was significantly more resource-intensive, with 1,644,030,374 primitive calls and a total execution time of 4108.539 seconds (approximately 68 minutes and 28 seconds). The detailed performance metrics for this task highlight several critical aspects:
- The `process_file` and `data_cleaner` functions, especially those involved in preprocessing and handling missing values with TPOT, took the longest times, with the `process_file` function alone taking over 4107 seconds, suggesting a heavy computational load.
- The predictive imputation and fitting processes via TPOT also demanded considerable time, indicating the complexity and computational intensity of using machine learning algorithms to impute missing data.
- The TPOT operations, including evolutionary algorithms (`eaMuPlusLambda`) and individual evaluations (`_evaluate_individuals`), were notably time-consuming, which reflects the iterative and resource-heavy nature of genetic programming used for optimizing the machine learning pipeline.

Overall Comparison

In comparing the performance tests, the impact of TPOT imputation on system performance is particularly notable with larger datasets. The increase in execution time is significant, especially because the larger dataset contained more numerical columns. Since TPOT operates on a per-column basis and each column requires separate processing—going through multiple generations of machine learning models—the computational load increases substantially.

Key Insights from the Performance Comparison:
1. Computational Demand: The imputation process using TPOT for a larger dataset with more numerical columns dramatically escalates both the number of function calls and execution time—from 19 seconds to over an hour for approximately 68 minutes. This reflects the added complexity and resource demands when dealing with more extensive numerical data.

2. Scalability and Efficiency: The high resource consumption by TPOT during imputation highlights scalability issues, particularly for large datasets where quick data processing is crucial. This situation necessitates considering more efficient methods or optimizing the existing process to better balance between accuracy and performance.

5. Results

5.1 Starter Page

The initial interface of the AI-driven data visualization tool sets the stage for user interaction by providing a streamlined and intuitive entry point. This starter screen, as shown in the image below, is designed to guide users smoothly into the data analysis process. It prominently features a file upload section where users can drag and drop their dataset or use the browse option to select files. This screen is the gateway through which users begin their journey of exploring and understanding their data, with options to specify how missing values should be handled right before initiating the data processing. This approach ensures that users are well-informed about the functionalities available and can make decisions that best suit their analytical needs.

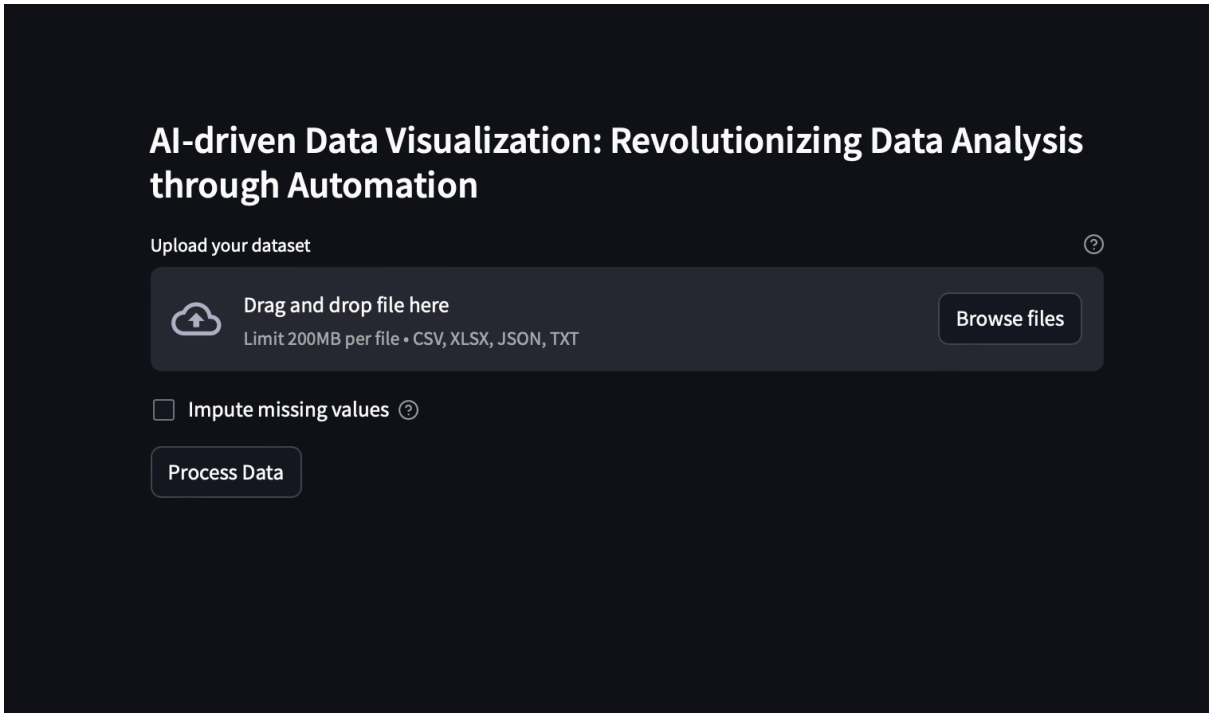


Figure 5.1.1 – Starter Page

The addition of tooltip help buttons greatly enhances the user experience by providing on-demand explanations for features like 'Upload your dataset' and 'Impute missing values'. Integrated based on user feedback, these tooltips help demystify the platform's functionalities, making the tool more accessible and user-friendly for newcomers.

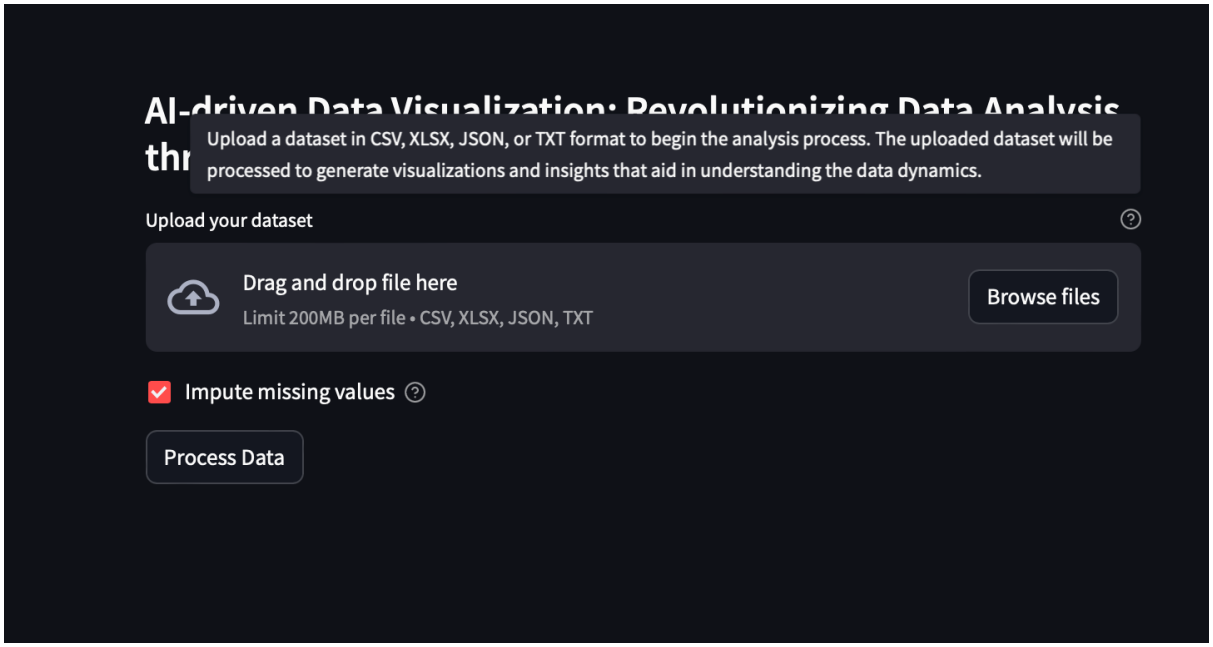


Figure 5.1.2 – Upload Dataset help tooltip.

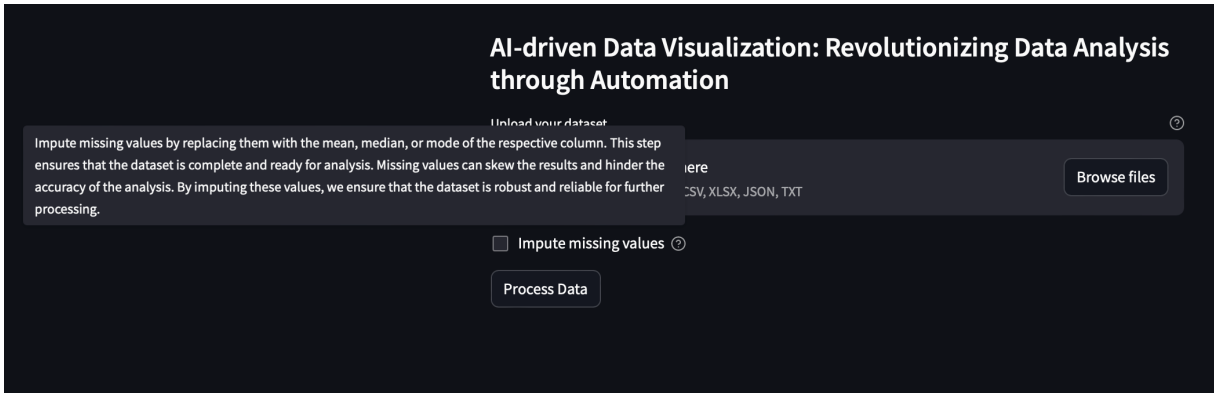


Figure 5.1.3 – Impute Missing Values help tooltip.

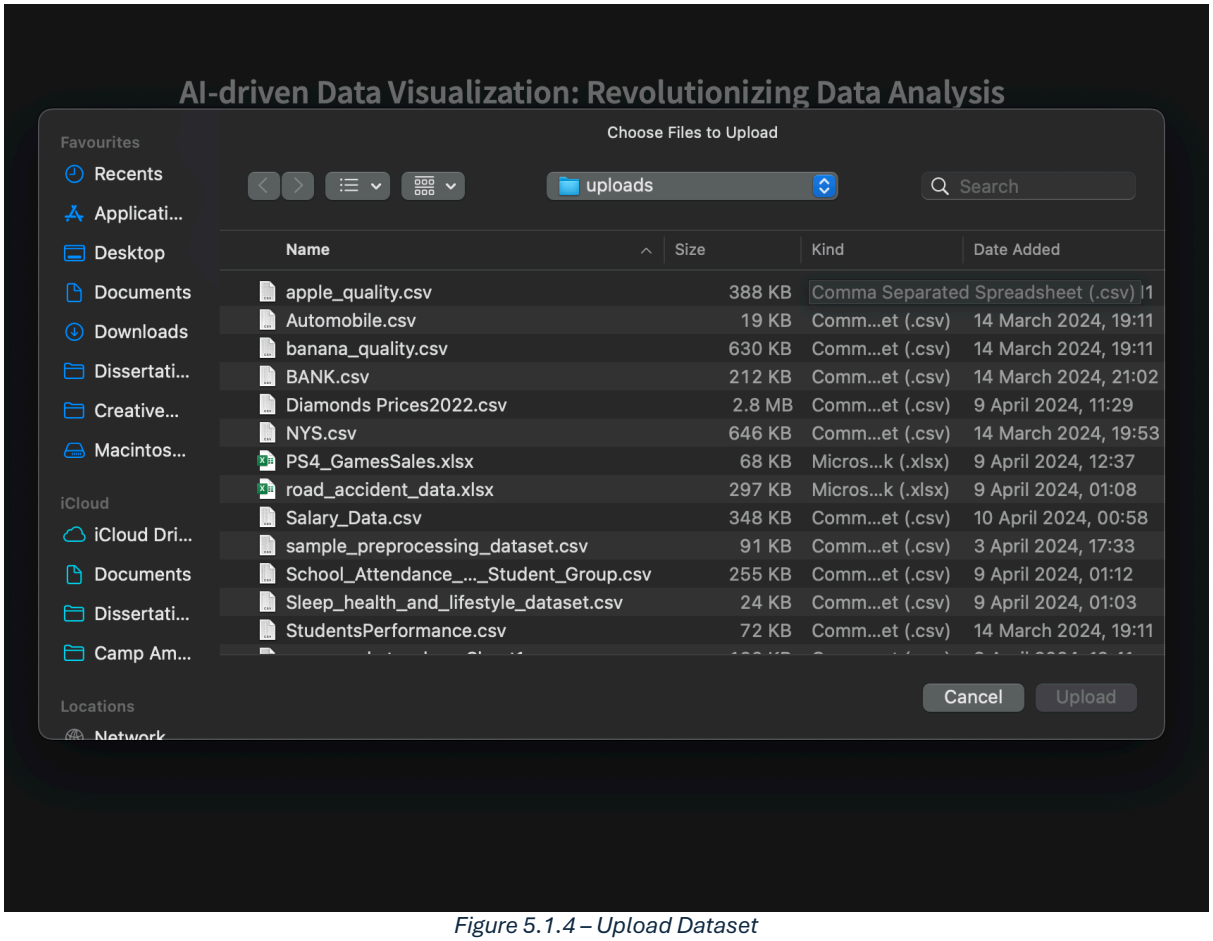


Figure 5.1.4 – Upload Dataset

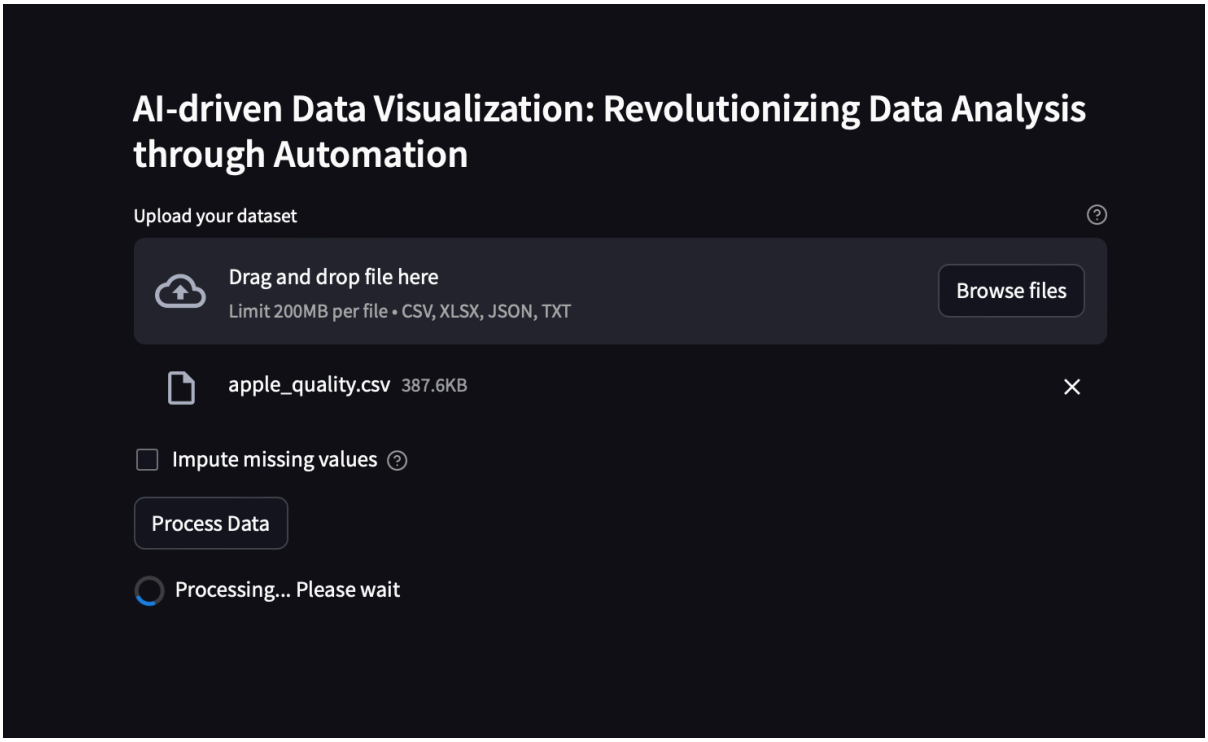


Figure 5.1.5 Processing Spinner

5.2 General Analysis Page

The general analysis section of the dissertation provides a comprehensive overview of the dataset, highlighting essential statistics and distribution characteristics. This section is pivotal as it leverages advanced data visualization techniques, including correlation heatmaps, to illustrate the interrelationships within the data. Such visualizations are instrumental in uncovering underlying trends and offering insights that facilitate a deeper understanding of the dataset's nuances.

This section goes beyond mere statistical analysis; it integrates AI-driven tools to automatically generate a title, description, buzzwords, and key points for each identified trend. This approach not only enhances the accessibility of the data but also enriches the user's analytical experience by distilling complex data into understandable and actionable insights. The use of descriptive analytics helps encapsulate the essence of the data trends, making the analysis both insightful and practical for users from various backgrounds.

By focusing on key data attributes and their interconnections, the general analysis section serves as a foundational component of the dissertation, setting the stage for more detailed explorations in subsequent sections. This methodical unpacking of the data underscores the tool's capability to transform raw data into strategic knowledge, thereby demonstrating the effectiveness of the AI-driven visualization approach adopted in this project.

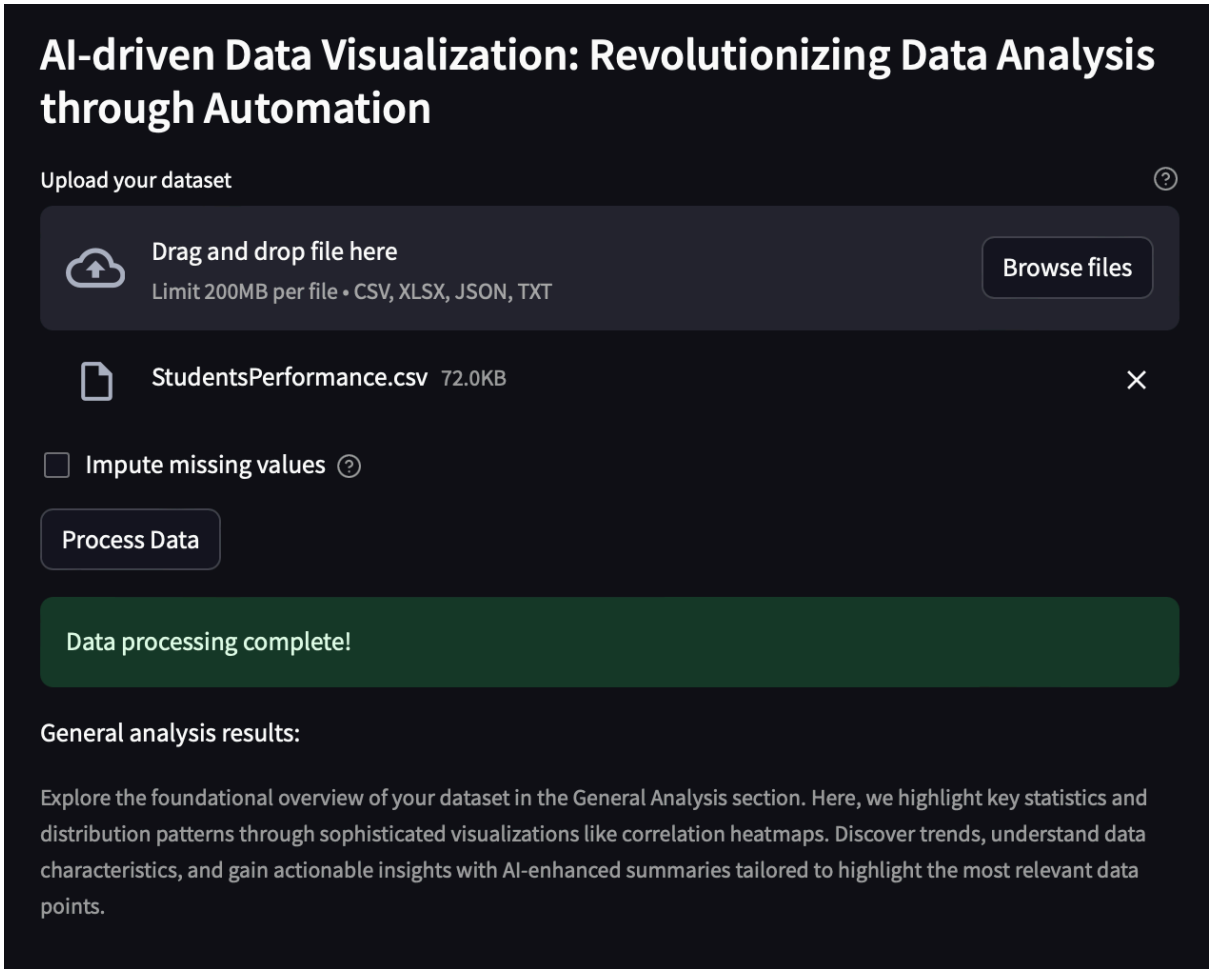
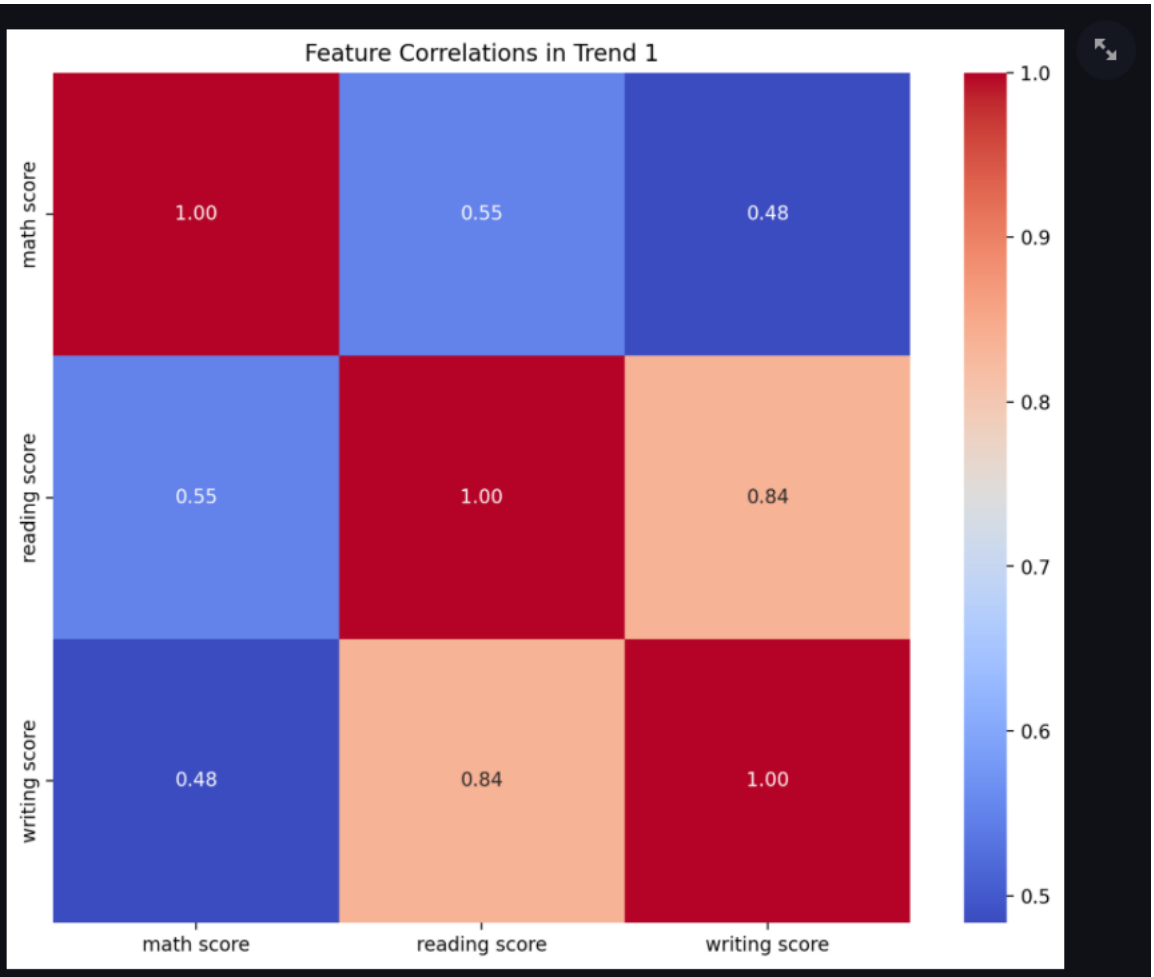


Figure 5.2.1 – Processing Data Complete



High Achievements

In this trend, students have performed exceptionally well across math, writing, and reading scores. The average math score stands at 78.78, writing at 82.45, and reading at 83.05. Moreover, the most common lunch type among students is standard. What sets this trend apart is the significant lead in scores compared to other trends: math score surpasses the average by 20.87, writing score by 23.68, and reading score by 22.83. Additionally, the prevalence of standard lunch is notably higher in this trend.

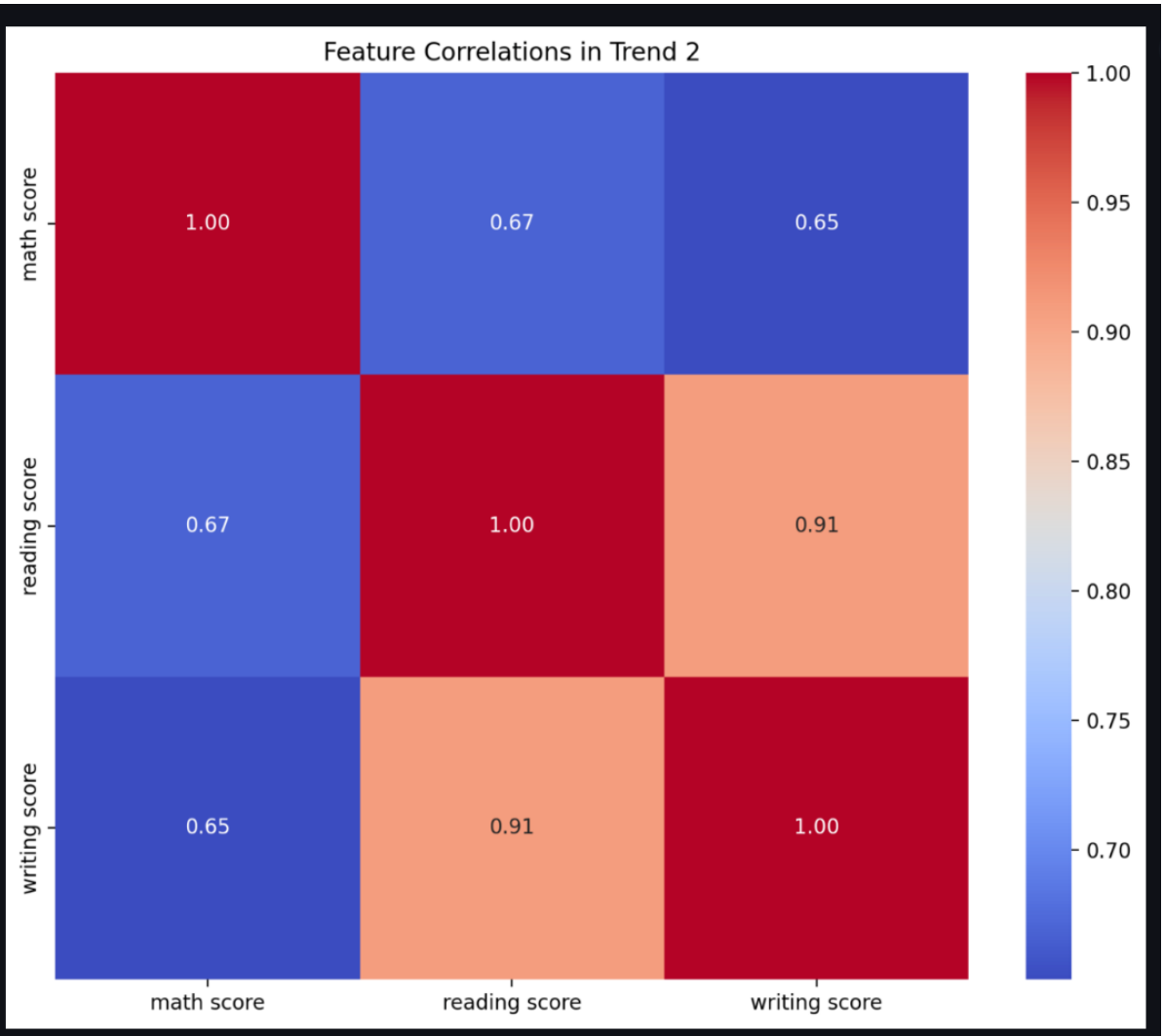
Buzzwords:

['high achievements', 'exceptional scores', 'significant lead']

Key points:

- Math score: 78.78 (avg)
- Writing score: 82.45 (avg)
- Reading score: 83.05 (avg)
- Standard lunch most common
- Scores significantly higher than average;

Figure 5.2.2 – Example Trend 1 Heatmap and Description



Low Performances

This trend showcases students with lower scores in math, writing, and reading, with averages at 57.91, 58.77, and 60.22 respectively. Interestingly, the most common test preparation course status is 'none'. The distinguishing factor of this trend is the notable lag in scores compared to other trends: math score falls below the average by 20.87, writing score by 23.68, and reading score by 22.83. Additionally, the absence of a test preparation course is more prevalent in this trend.

Buzzwords:

['low performances', 'lagging scores', 'no test prep']

Key points:

- Math score: 57.91 (avg)
- Writing score: 58.77 (avg)
- Reading score: 60.22 (avg)
- No test prep most common
- Scores notably lower than average;

Figure 5.2.3 – Example Trend 2 Heatmap and Description

In the General Analysis section, a distinct approach to data presentation is adopted. Unlike other sections where interactivity is emphasized, here users can download the entire overview as a PDF report. This facilitates a comprehensive and professional review of the dataset's key statistics and distributions, suitable for presentations or record-keeping. To generate this report, users need to click the 'Generate PDF Report' button. Once the report is prepared, a 'Download Now' button will appear, allowing users to easily access and save the detailed analysis. This process ensures the information is presented in a static, user-friendly format that is both shareable and retainable.

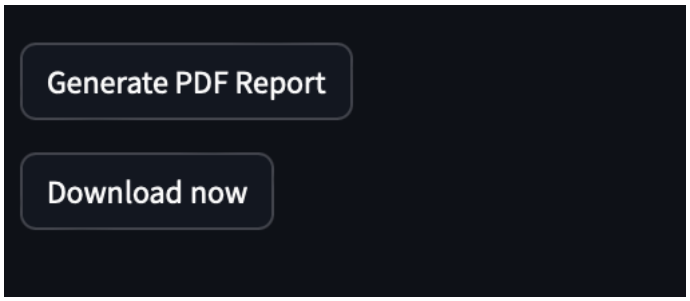


Figure 5.2.4 – Generate and Download PDF report.

Appendix 5.2.1 – PDF Report.

5.3 Sidebar Menu

The sidebar in the visualization dashboard serves as a dynamic and intuitive navigation tool, allowing users to select various types of data analyses with ease. Depending on the type of data uploaded, the sidebar dynamically adjusts to present only the relevant analysis options. For datasets containing categorical, financial, or time/date data, specific charting options become available, tailored to these data types. This ensures that users have a streamlined and focused experience, accessing only those visualizations that are applicable to their data, thereby enhancing usability and efficiency of analysis.

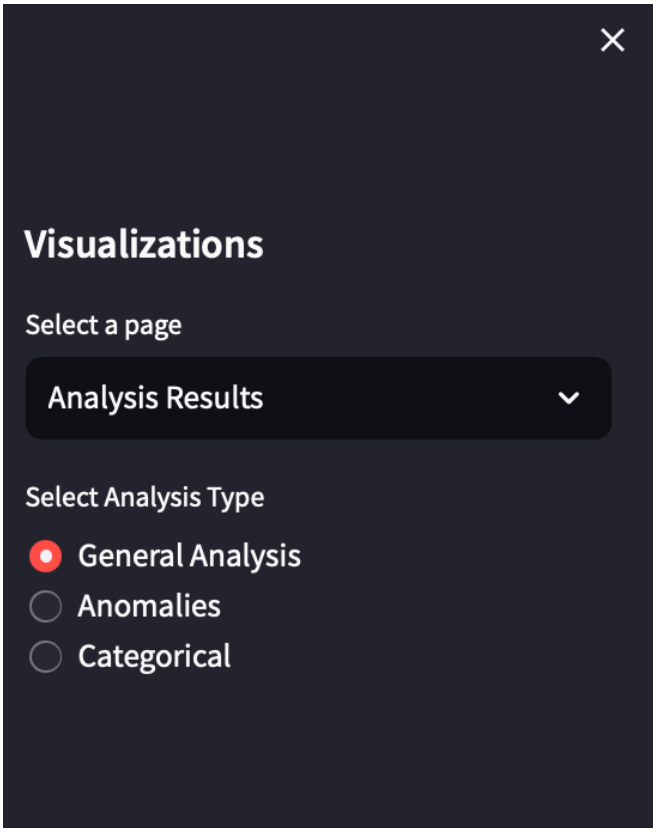


Figure 5.3.1 – Analysis Results Menu Options

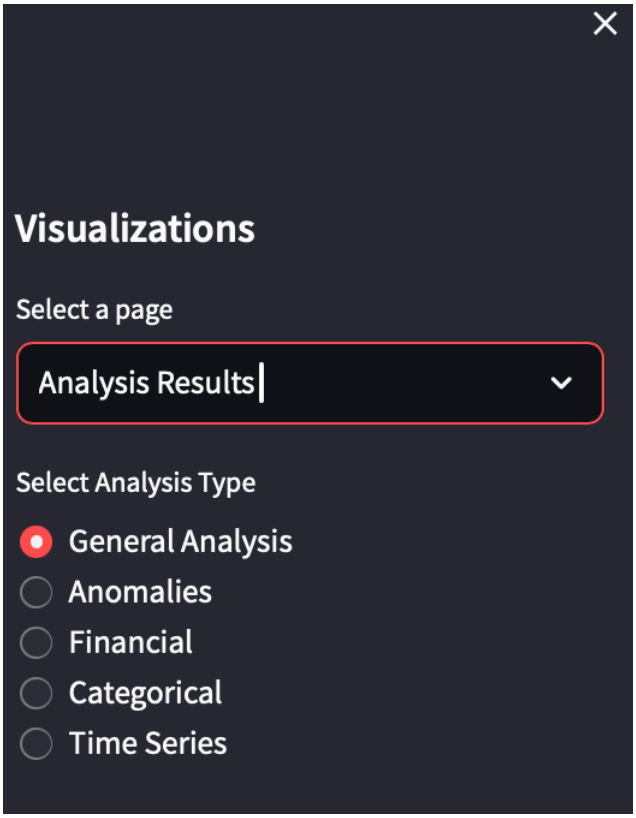


Figure 5.3.2 – Analysis Results Dynamic Options Based on Dataset

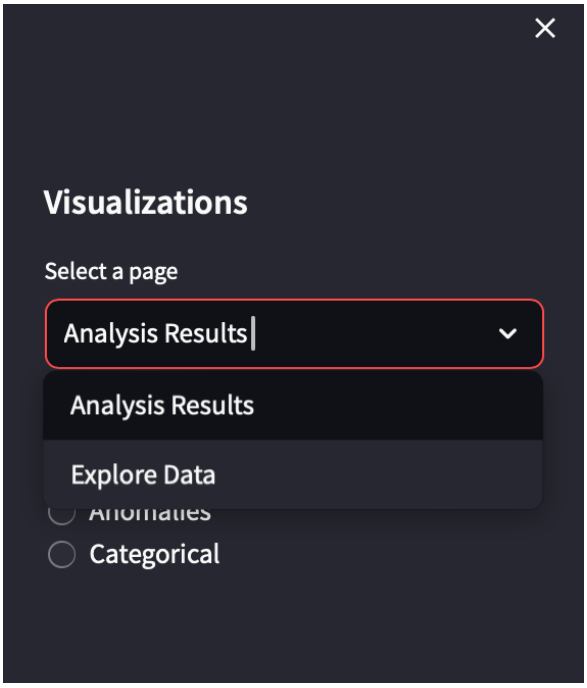


Figure 5.3.3 – Page Options

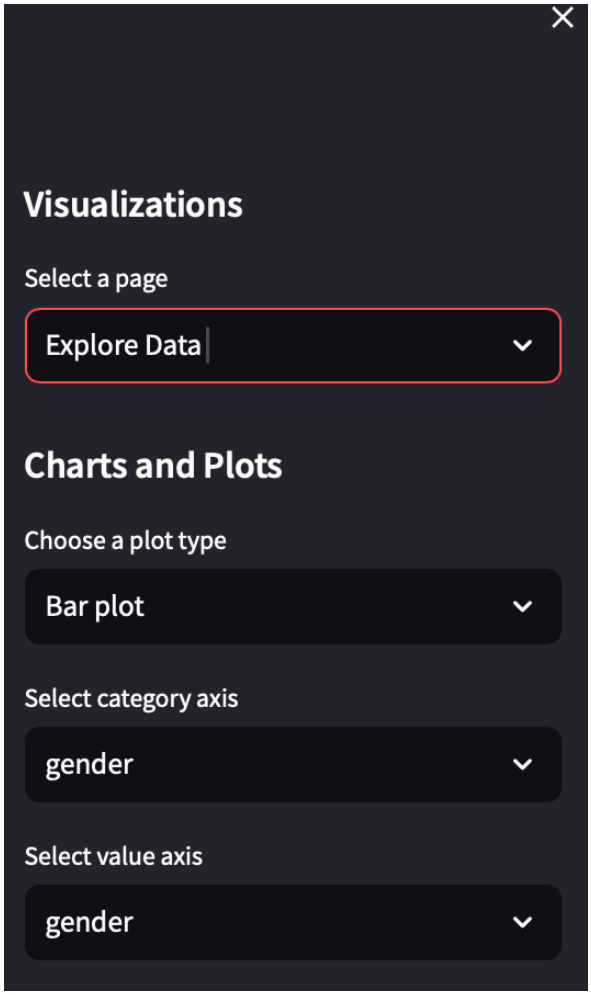


Figure 5.3.4 – Explore Data Options

5.4 Anomaly Charts

The Anomalies section in the visualization dashboard is designed to facilitate the detection and analysis of outliers within your dataset through box plots. Box plots are an effective tool for statistically summarizing data distributions, focusing on the quartiles and ranges which highlight data points that deviate significantly from the norm. This visualization helps in quickly identifying and assessing outliers, which are crucial for understanding potential anomalies that could affect data analysis outcomes. By leveraging these insights, users can take pre-emptive measures to address data quality issues or investigate underlying causes for such deviations.

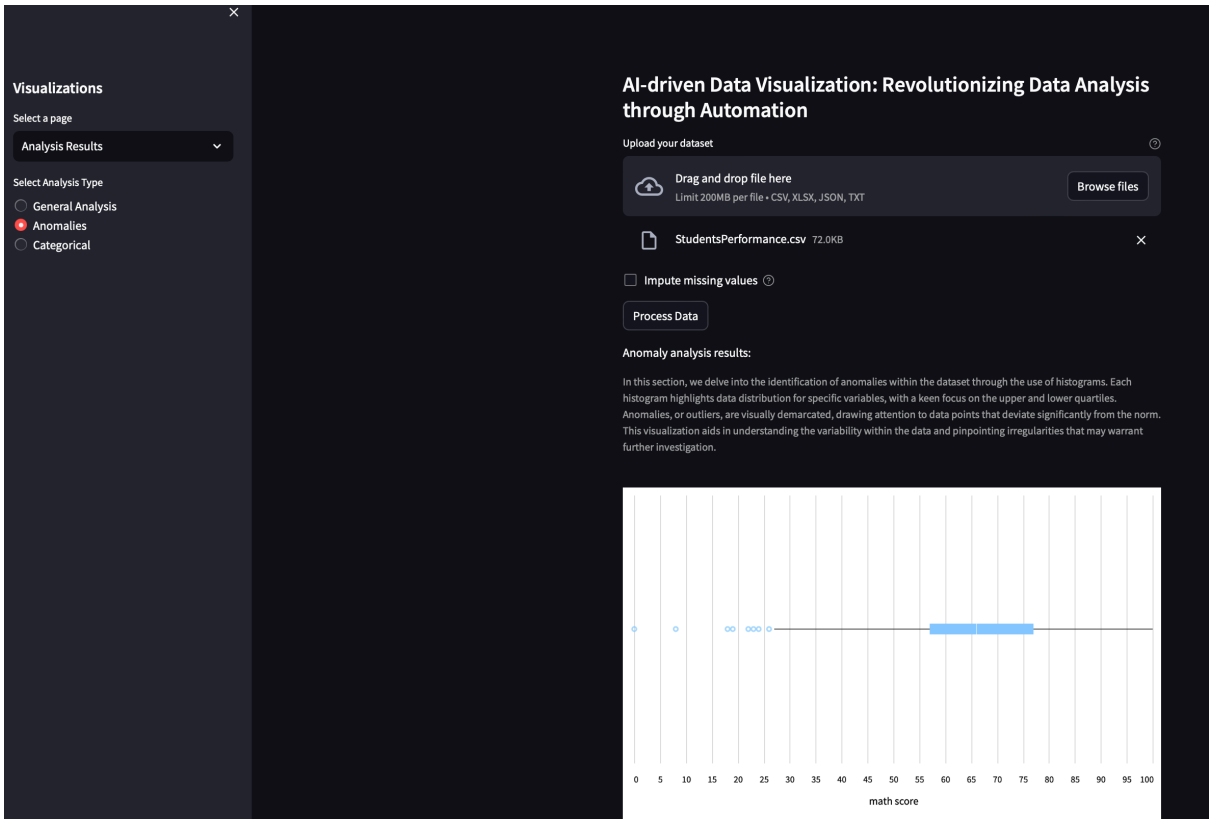


Figure 5.4.1 – Example of Anomalies Page

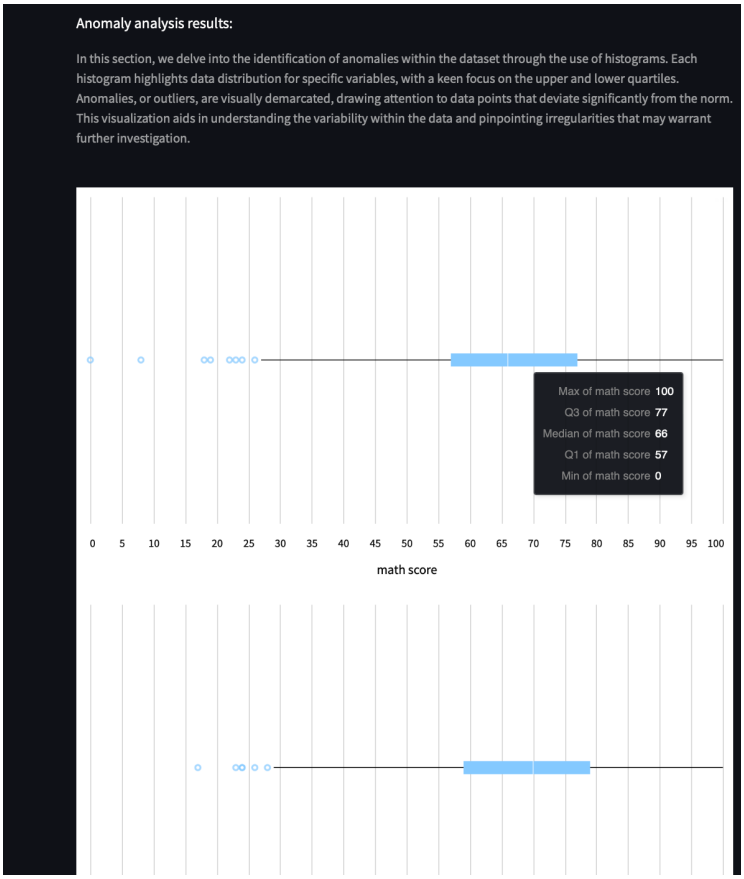


Figure 5.4.2 – Example of Anomalies chart interactivity

In this tool, charts from the "Anomalies" section can be downloaded as HTML files to preserve their interactivity. This functionality is crucial for users who need to perform further analysis or presentations outside the application. Downloading the chart as an HTML file enables users to retain dynamic elements such as tooltips and clickable components, which are not possible in static image formats like PNG. Although once they have downloaded the HTML file, they are presented with the option to save as a PNG.

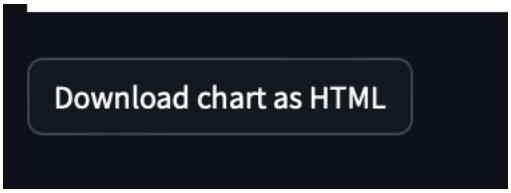


Figure 5.4.3 – Download Chart as HTML button.

5.5 Categorical Charts

In this section, the program autonomously generates visual representations for categorical data within the dataset. It creates intuitive bar charts that delineate the distribution and relationship among categorical variables, facilitating a deeper understanding of underlying trends and inconsistencies. This automated visualization process efficiently highlights significant insights that are crucial for comprehensive data analysis.

Figure 5.5.1 – Example of Categorical Charts

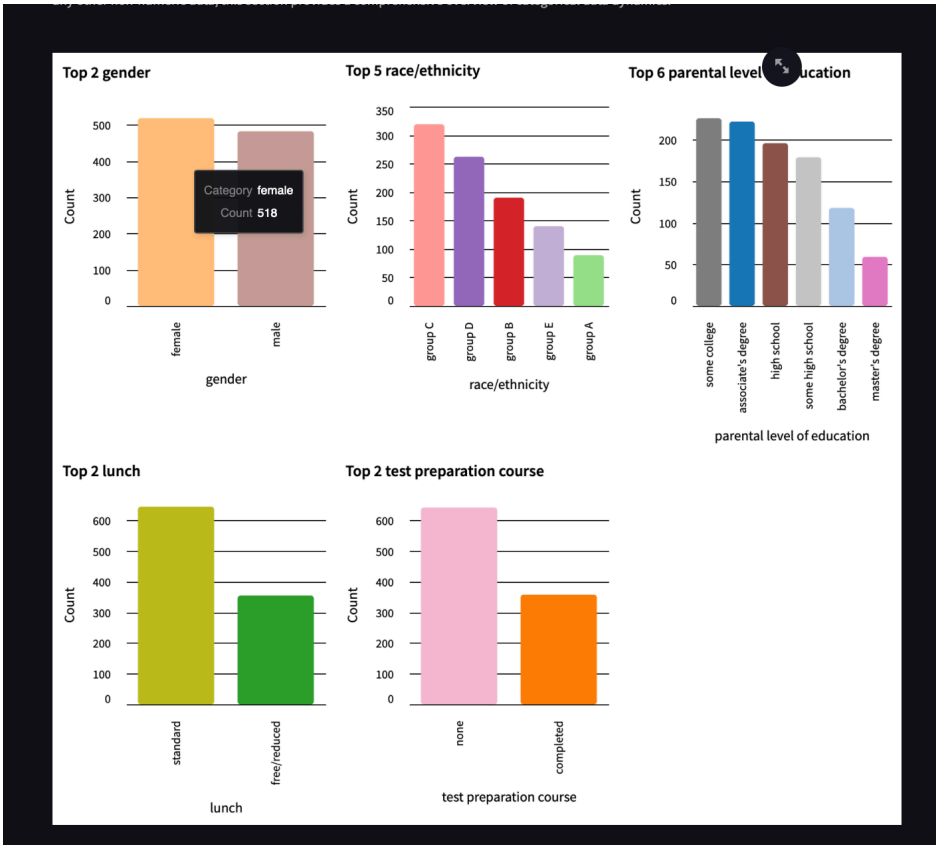


Figure 5.5.2 – Example of Categorical Charts Interactivity

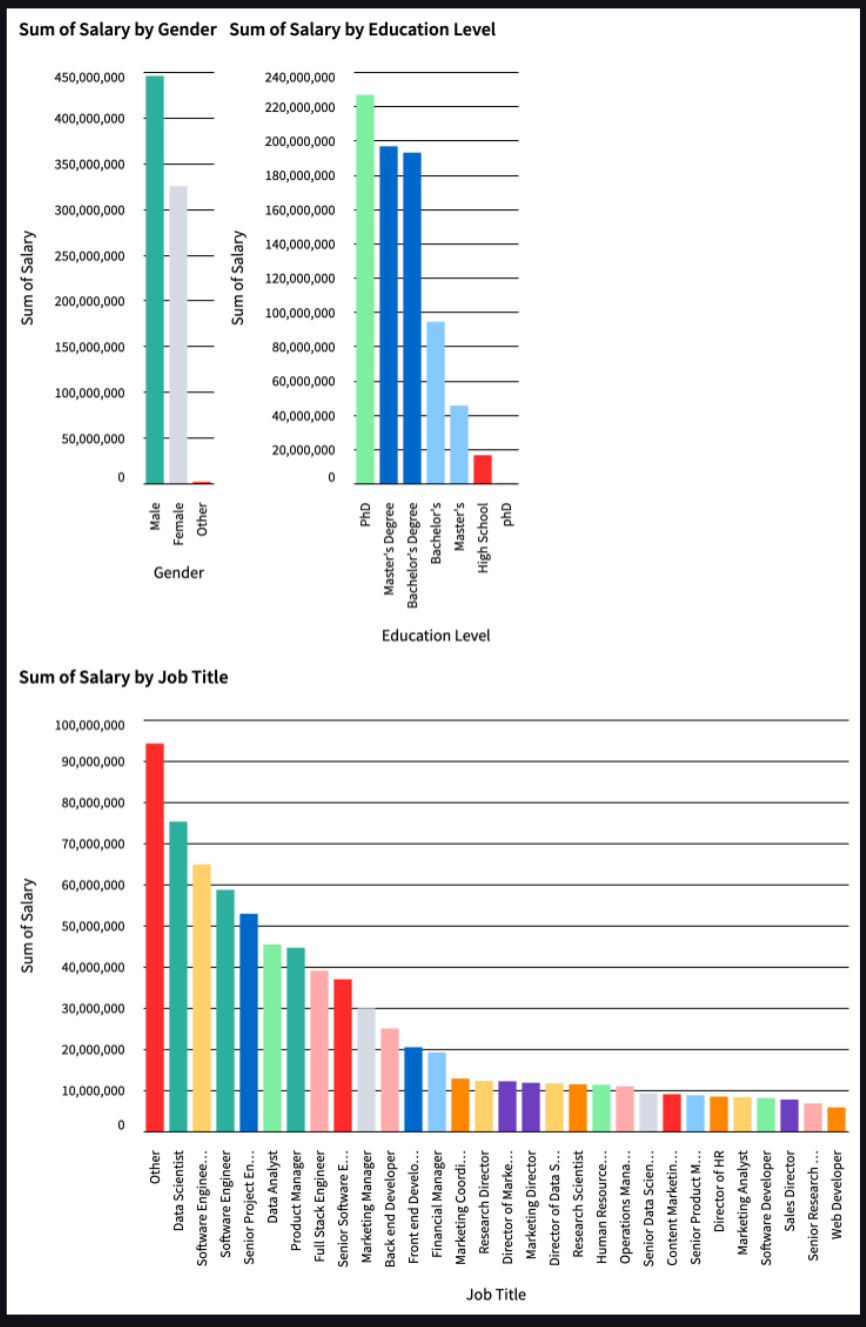
Again, like the anomalies chart, users are given the option to download the chart as a HTML file to keep the interactivity with the option of a PNG when downloaded.

5.6 Financial Charts

In the "Financial Charts" section, the tool automatically generates visualizations for financial data found in the dataset. This section produces bar charts that compare financial metrics across different categorical dimensions, providing a clear visualization of financial performance and trends. These charts aid in the quick identification of financial patterns and anomalies, making it easier for users to understand financial dynamics and make informed decisions based on the visualized financial data.

Financial analysis results:

This section is dedicated to showcasing the financial aspects of the dataset. By recognizing all financial-related columns, it presents an in-depth analysis through bar charts that pair these financial metrics with categorical columns. This method of visualization not only simplifies the comparison across different categories but also provides a clear perspective on financial trends and distributions within the dataset. Whether its revenue, expenses, or any other financial parameter, this section brings critical financial insights to the forefront.



Download chart as HTML

Figure 5.6.1 – Financial Charts

In the Financial Charts section of the data visualization tool, charts such as bar charts are used to present financial data effectively. This particular bar chart aggregates and compares salary totals across different job titles, displaying the sum of salaries associated with each role. To maintain clarity and avoid overcrowding in the visualization, the chart limits the display to the top 30 categories by default. Categories beyond this threshold are grouped into an 'Other' category, ensuring the chart remains legible and focused while still providing a comprehensive overview of the financial data distribution across various job titles. This approach helps in highlighting the most significant data points without overwhelming the user with too much detailed information at once.

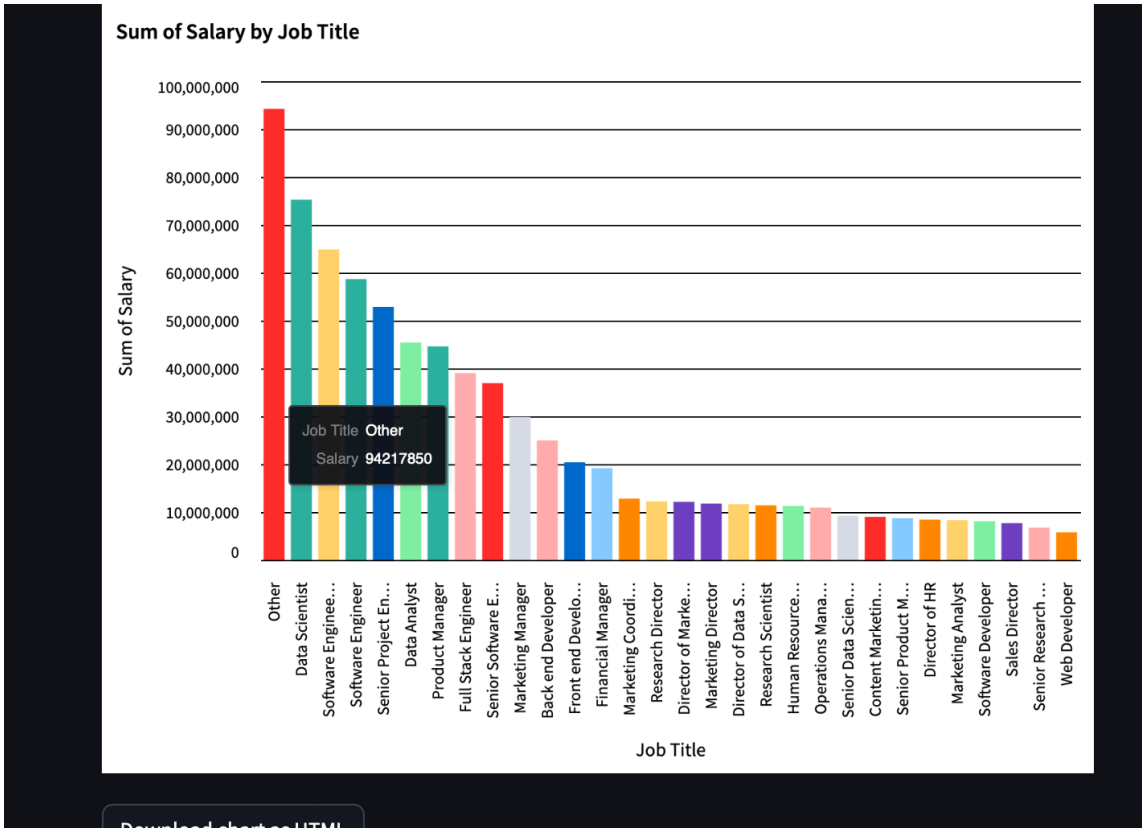


Figure 5.6.2 – Financial Chart Interactivity

5.7 Time Series Charts

In the Time Series Charts section, users can explore how data variables evolve over time through dynamic line graphs. This section is particularly valuable for datasets that include temporal data points such as dates or times, enabling users to trace trends, identify seasonal patterns, or detect cyclical changes. These visualizations not only

help in understanding the longitudinal aspects of the data but also assist in forecasting future trends based on historical patterns.

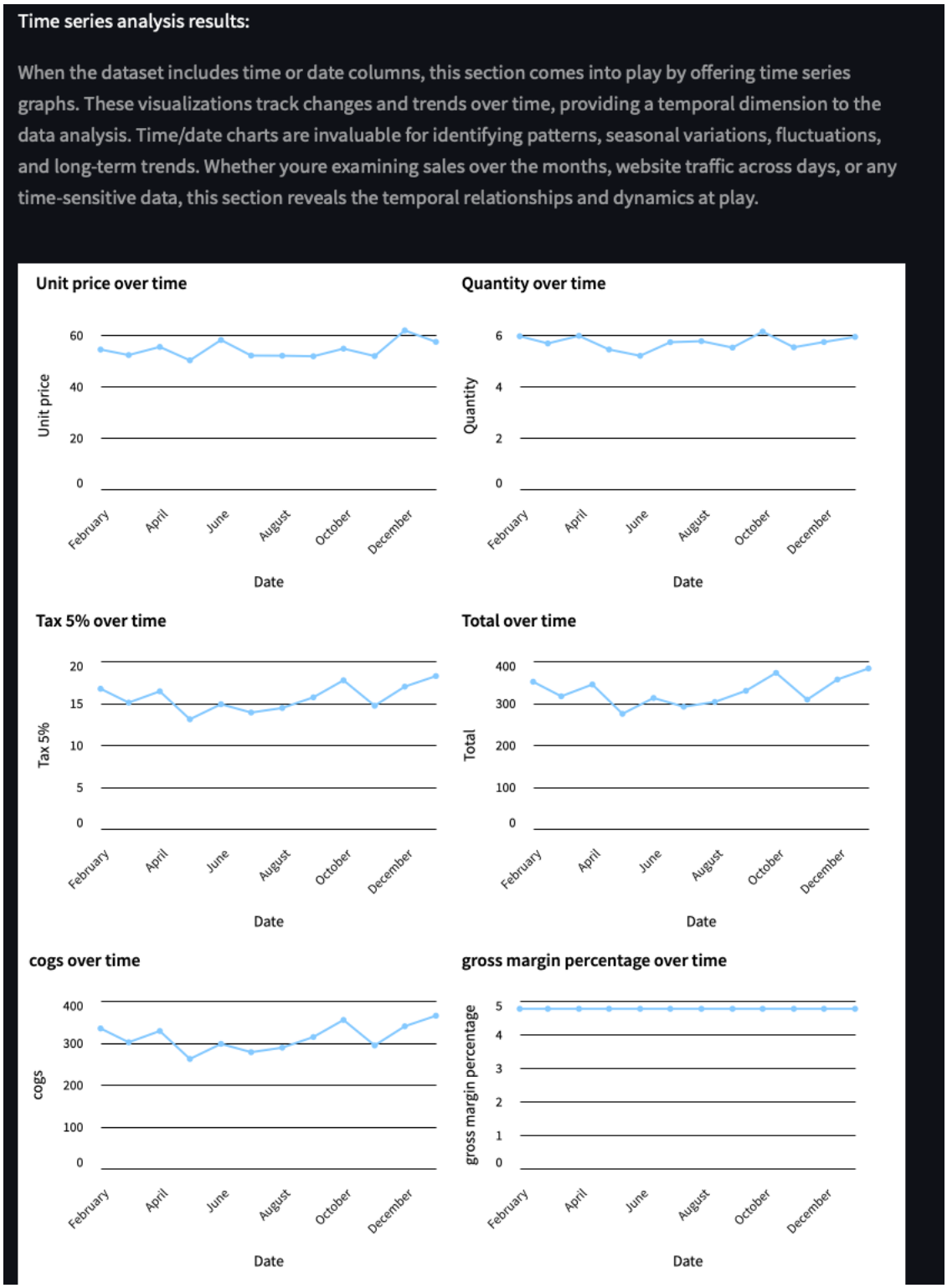


Figure 5.7.1 – Time Series Charts

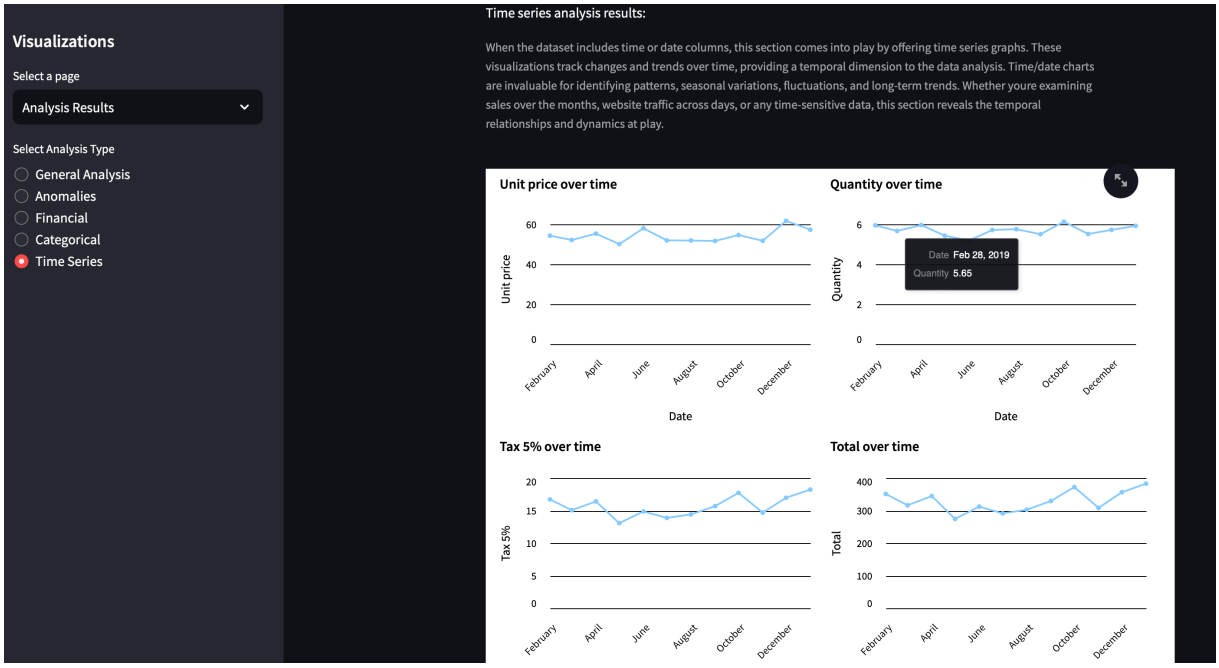


Figure 5.7.2 – Time Series Chart Iteractivity

5.8 Explore Data Page

The "Explore Data" section of the dashboard empowers users to actively engage with their data by crafting custom visualizations. This interactive module offers the tools to create bar plots, scatter graphs, box plots, or histograms. Users can define their own parameters for these charts by selecting the desired variables for the x and y axes from their dataset. This functionality enhances user involvement and personal exploration of the dataset, allowing for a deeper, hands-on analysis tailored to specific informational needs or queries.

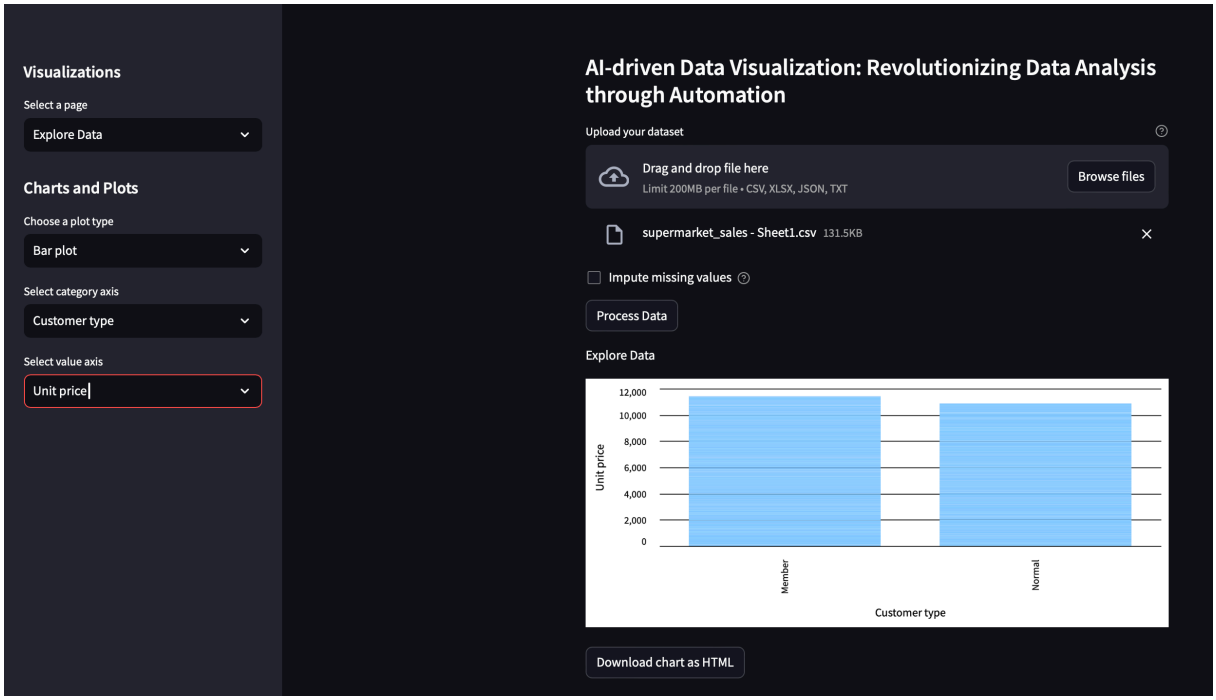


Figure 5.8.1 – Create a Bar Plot

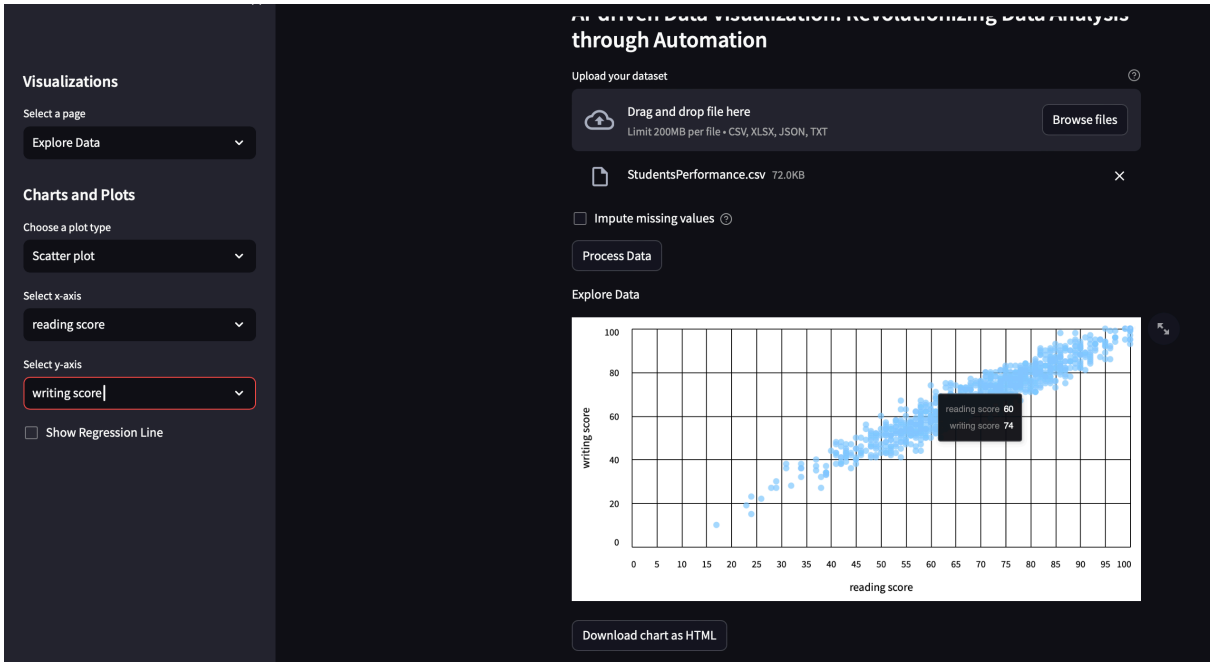


Figure 5.8.2 – Create a Scatter Plot

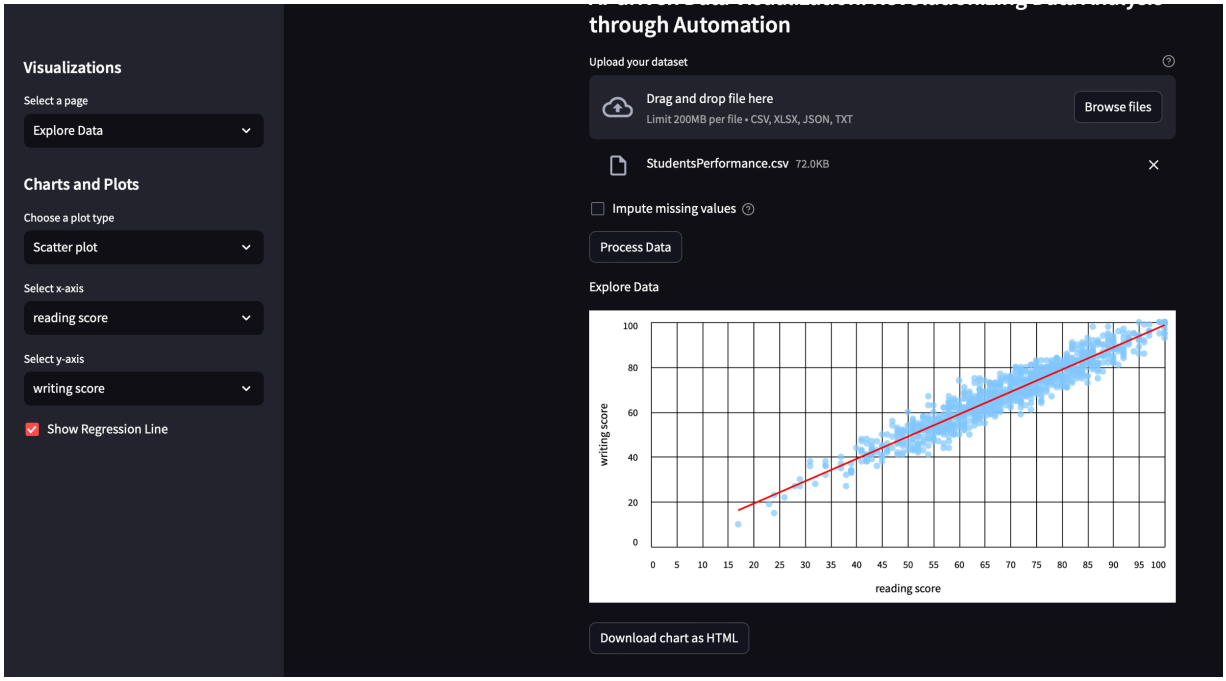


Figure 5.8.3 – Create a Scatter Plot with Regression Line

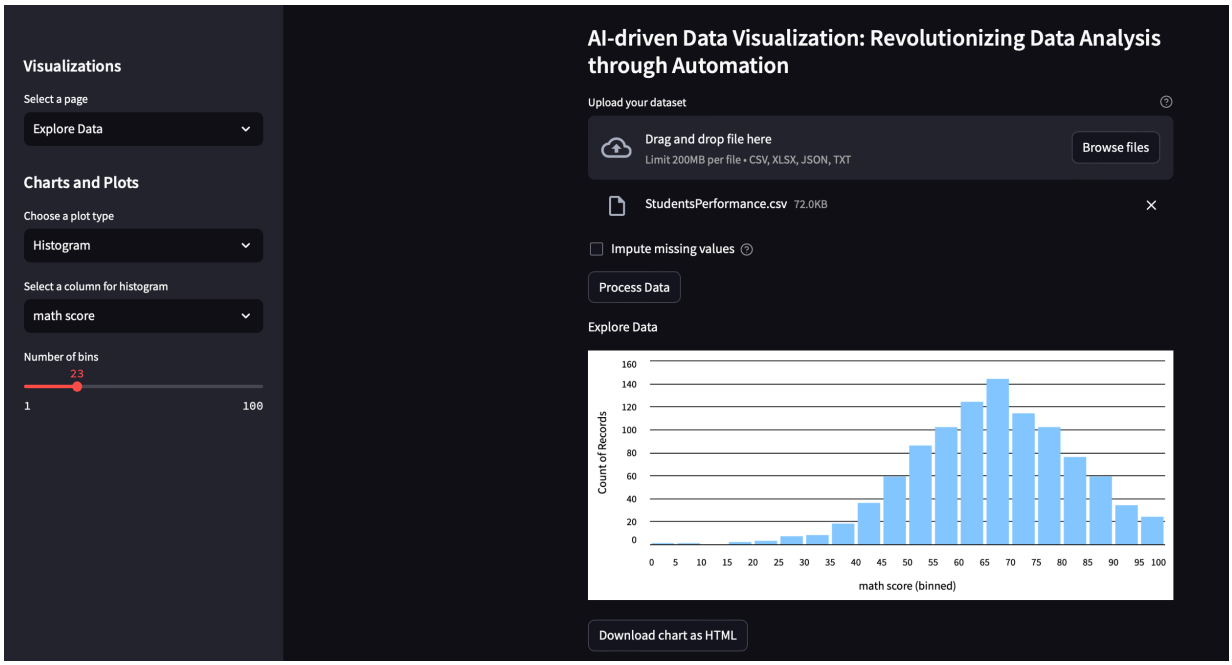


Figure 5.8.4 – Create a Histogram with Dynamic Bins

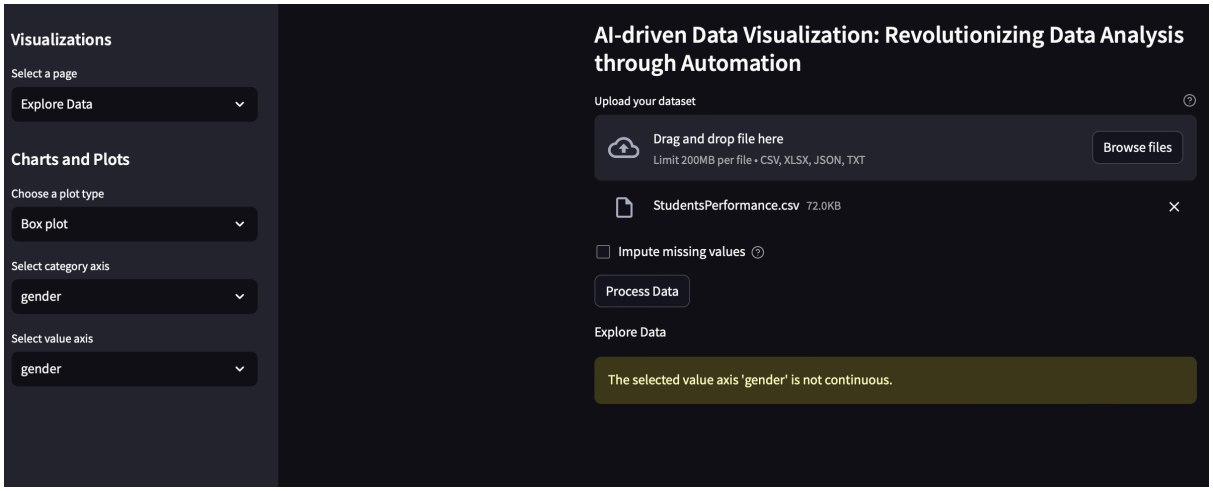


Figure 5.8.5 – Create a Box Plot warning.

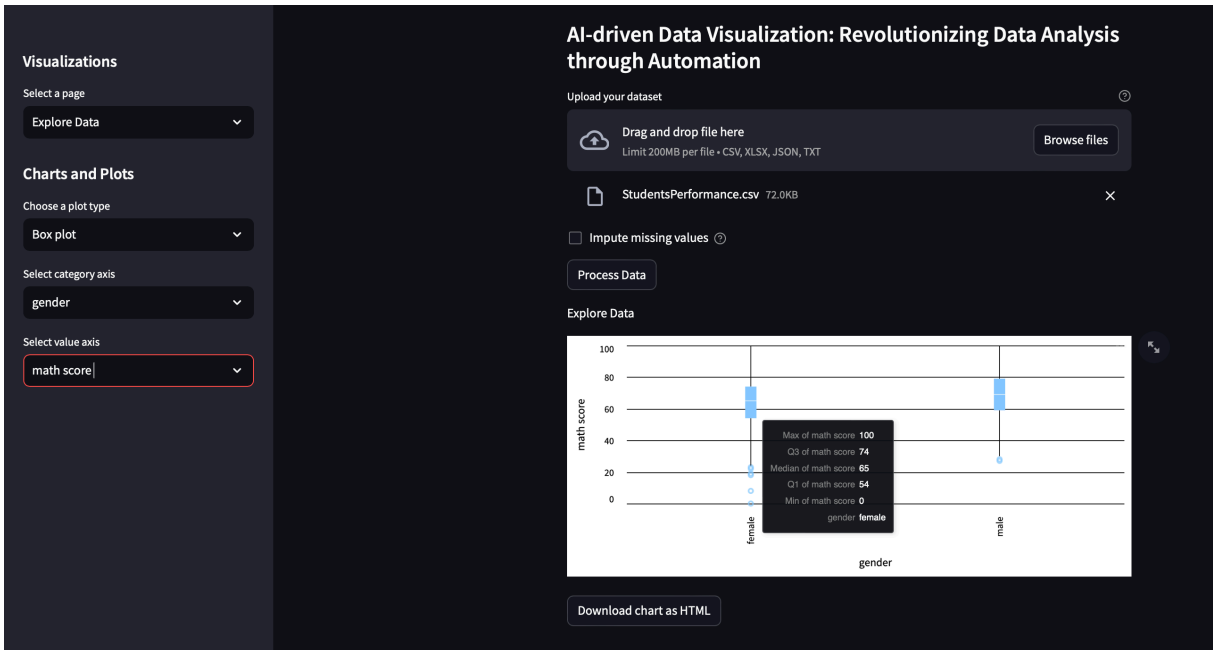


Figure 5.8.6 – Create a Box Plot Working

6: Conclusion and Evaluation

6.1 Insights from Testing

Improvement Recommendations from Users:

Enhanced Onboarding and Tutorials: Initial feedback highlighted a need for clearer guidance on navigating and utilizing the platform's features. To address this, interactive tutorials, and comprehensive guides, including tooltip help buttons, have been introduced. These additions aim to assist users in maximizing the tool's capabilities, ensuring they can leverage its functionalities fully from the start.

Customizable User Help Options: Testing revealed the importance of tailoring help options to match user expertise levels. This is something I have identified to be implemented for future work and was not implemented due to time constraints.

Export Charts Feature: Experienced users expressed a need for functionality to export visualizations created within the "Explore Data" section. Responding to this, 'Save' and 'Export' options have now been integrated, allowing users to download and utilize their charts outside the platform, enhancing the tool's utility for presentations and reports.

These improvements were informed by a comprehensive usability testing phase, which provided critical insights into user interactions and satisfaction. The adaptations made not only enhance the tool's functionality but also significantly improve its intuitiveness and accessibility, thereby increasing overall user engagement and satisfaction.

Adaptations and Improvements

Based on the feedback and results from the usability tests, several key adaptations were made to refine and enhance the tool:

1. Interactive Guidance Integration: To make the tool more accessible, particularly for users with limited data science expertise, interactive step-by-step guides and tooltips were integrated throughout the application.

2. Functionality Enhancements: New features such as the ability to export data visualizations directly from the tool were added. This not only responded to specific user requests but also augmented the tool’s practicality for professional and academic use.

3. Error Handling and Feedback Mechanisms: Improvements were made in error handling and feedback mechanisms to ensure that users are adequately informed of any issues and understand how to resolve them or proceed when anomalies occur.

These adaptations are critical in evolving the tool to meet user expectations and needs more effectively, ensuring the longevity and sustainability of the platform. By continually integrating user feedback into the development process, the tool remains aligned with the needs of its target audience, thereby enhancing its efficacy and user engagement.

6.2 Project Conclusion

The project successfully developed an AI-driven data visualization tool aimed at enhancing the accessibility and efficacy of data analysis for non-expert users. By integrating advanced AI methodologies, the tool facilitates the extraction of actionable insights from complex datasets, effectively democratizing data science capabilities. Key achievements include the creation of a user-friendly interface with Streamlit,

incorporation of OpenAI for generating insightful summaries, and the implementation of sophisticated data processing techniques that prepare data for insightful visualization.

Comparison to Initial Aim:

Aim: To develop an AI-driven data visualization tool to improve data analysis efficiency and effectiveness, making it accessible for users without data science expertise.

The project achieved its primary aim of enhancing the efficiency and accessibility of data analysis, thus broadening the ability for a wider audience to engage with and benefit from sophisticated data insights. The tool's development from concept to implementation adhered closely to the planned objectives, with modifications driven by user-centric feedback, model tuning and performance metrics leading the way to a completed program.

Appendix 3 - Comparison to initial Objectives

6.3 Future work and Longevity

As this project matures and evolves, there are several avenues for enhancement and expansion that can further increase its impact and utility. Below are the key areas identified for future development:

Cloud Deployment:

Transitioning the application to a cloud-based platform is a pivotal step towards scalability and accessibility. By leveraging cloud computing, the tool can accommodate larger datasets and support more concurrent users without compromising performance. This shift will also facilitate easier updates and maintenance, ensuring that the tool remains robust against the rapidly changing technological landscape.

As the project transitions to a cloud-based platform, prioritizing data protection is crucial. Implementing strong encryption, secure access controls, and continuous monitoring are essential to safeguard sensitive information from unauthorized access and breaches. These robust data security measures will build user trust by ensuring their data is handled responsibly and securely in the cloud.

Support for Larger Databases:
To enhance the tool's capability to handle increasingly complex datasets, future iterations will focus on optimizing data processing algorithms and expanding the backend infrastructure. This includes refining data ingestion and storage processes to manage larger volumes of data efficiently. Improving these aspects will make the tool more attractive to enterprises and research institutions dealing with big data.

Diverse Database Compatibility:

Expanding the tool's compatibility to include a wider array of databases and data formats is essential. This will involve developing additional parsers and converters to handle diverse data structures from various industries, such as healthcare, finance, and e-commerce. Enabling this capability will broaden the tool's applicability, helping users from different sectors to gain actionable insights from their specific data more effectively.

Enhanced AI-Driven Insights:

To push the boundaries of what the tool can offer in terms of insights, integrating more advanced AI algorithms is necessary. This entails employing cutting-edge machine learning models that can delve deeper into data analysis, providing more nuanced and predictive insights. Enhancements might include more sophisticated natural language processing techniques to improve the clarity and relevance of AI-generated summaries and expanding the AI's understanding of complex datasets.

Customizable User Help Options:

Feedback from usability testing highlighted the need for help options that can adapt to user expertise levels. While initial adaptations have included tooltip help buttons for immediate assistance, future developments will focus on a more sophisticated system that can identify new users, potentially through a login system, and tailor the help options accordingly. This feature will enhance user experience by providing personalized support, thus making the tool more accessible to a broader audience.

Implications for Future Research:

The advancements in AI-driven data visualization tools not only offer immediate practical benefits but also pave the way for substantial academic and industrial research opportunities. The integration of more sophisticated AI capabilities into data visualization can spur further studies in fields such as predictive analytics, human-computer interaction, and automated decision-making systems. Additionally, by setting new standards in data handling and visualization, this project can contribute to shaping regulatory and ethical guidelines in the use of AI technologies in data analysis.

These enhancements aim not only to increase the functional capacity of the tool but also to ensure its sustainability and relevance in the long term. By focusing on these areas, the project can continue to provide significant value to its users, helping them navigate and interpret complex data landscapes with greater ease and accuracy.

References

[1]
"[1] S. Mehta, 'Efficient Deep Learning for Visual and Textual Data.' Order No. 28321173, University of Washington, United States -- Washington, 2021."

[2]
A. Stöckl, “Natural Language Interface for Data Visualization with Deep Learning Based Language Models,” IEEE Xplore, Jul. 01, 2022. https://ieeexplore.ieee.org/document/10017653

[3]
G. Pang, C. Shen, L. Cao, and A. V. D. Hengel, “Deep Learning for Anomaly Detection,” ACM Computing Surveys, vol. 54, no. 2, pp. 1–38, Mar. 2021, doi: https://doi.org/10.1145/3439950.

[4]
A. B. Gumelar, “An Anatomy of Machine Learning Data Visualization,” IEEE Xplore, Sep. 01, 2019. https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8884340 (accessed Feb. 17, 2020).

[5]
McKinsey & Company, “Making data analytics work: Three key challenges | McKinsey,” www.mckinsey.com, Mar. 01, 2013. https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/making-data-analytics-work&#8203 (accessed Feb. 22, 2024).

[6]
Wahyu Sardjono, Astari Retnowardhani, Robert Emil Kaburuan, and Aninda Rahmasari, “Artificial intelligence and big data analysis implementation in electronic medical records,” Dec. 2021, doi: https://doi.org/10.1145/3512576.3512618.

[7]
E. Rose, Usability Testing Plan Template: A flexible tool for planning and teaching usability evaluation. 2023. doi: https://doi.org/10.1145/3548658.

[8]
Stanford University, “The AI Index Report – Artificial Intelligence Index,” aiindex.stanford.edu, 2023. https://aiindex.stanford.edu/report/

[9]
Y. LeCun, Y. Bengio, and G. Hinton, “Deep Learning,” Nature, vol. 521, no. 7553, pp. 436–444, May 2015, doi: https://doi.org/10.1038/nature14539.

[10]
J. Hirschberg and C. D. Manning, “Advances in natural language processing,” Science, vol. 349, no. 6245, pp. 261–266, Jul. 2015, doi: https://doi.org/10.1126/science.aaa8685.

[11]

H. M. Krumholz, “Big Data And New Knowledge In Medicine: The Thinking, Training, And Tools Needed For A Learning Health System,” Health Affairs, vol. 33, no. 7, pp. 1163–1170, Jul. 2014, doi: https://doi.org/10.1377/hlthaff.2014.0053.

[12]
T. H. Davenport, “Artificial Intelligence for the Real World,” Harvard Business Review, Mar. 09, 2018. https://hbr.org/webinar/2018/02/artificial-intelligence-for-the-real-world

[13]
I. T. Jolliffe, Principal Component Analysis. New York: Springer-Verlag, 2002. doi: https://doi.org/10.1007/b98835.

[14]
J. MacQueen, “Some methods for classification and analysis of multivariate observations,” projecteuclid.org, 1967. https://projecteuclid.org/euclid.bsmsp/1200512992

[15]
Charlie Robinson, “Charlie592/DissertationProject,” GitHub, May 07, 2024. https://github.com/Charlie592/DissertationProject (accessed May 07, 2024).

Appendix

Task	User 1 Completion	User 1 Time (min)	User 1 issues	User 1 Satisfaction
Data Upload	Yes	1	None	Satisfied
Process the Data	Yes	1	Confused by processing options	Moderately Satisfied
Imputing or Dropping Missing Rows	Yes	1	Unclear options for handling missing data	Satisfied
Read and Extract Insights from AI Description	Yes	4	Difficulty understanding some AI terminologies	Satisfied
Read and Extract Insights from Heatmap	Yes	3	Misunderstanding heatmap data points	Satisfied
Navigating to Anomalies Charts Screen	Yes	1	Brief confusion on navigation	Very Satisfied

Navigating to Categories Charts Screen	Yes	1	None	Satisfied
Navigating to Financial Charts Screen	Yes	1	None	Very Satisfied
Navigating to Time Series Charts Screen	Yes	1	None	Very Satisfied
Changing Page to Explore Data Page	Yes	1	None	Very Satisfied
Create a Box Plot with Web Tools	Yes	3	Difficulty understanding tool usage	Satisfied
Create a Histogram with Web Tools	Yes	4	Found tools but slow to apply settings	Moderately Satisfied
Create a Bar Plot with Web Tools	Yes	2	Minor confusion over axis selection	Satisfied
Create a Scatter Graph with Web Tools	Yes	3	None	Very Satisfied
Create a Scatter Graph with a Regression Line	Yes	3	Needed guidance on regression feature	Moderately Satisfied
Upload a New Dataset	Yes	3	Initial trouble with if they needed to remove the other dataset	Satisfied

Appendix 1 - User 1 Usability Test Results

Task	User 2 Completion	User 2 Time (min)	User 2 issues	User 2 Satisfaction
Data Upload	Yes	1	None	Very Satisfied
Process the Data	Yes	2	None	Very Satisfied
Imputing or Dropping Missing Rows	Yes	1	None	Very Satisfied
Read and Extract Insights from AI Description	Yes	2	None	Very Satisfied
Read and Extract Insights from Heatmap	Yes	1	None	Very Satisfied
Navigating to Anomalies Charts Screen	Yes	1	None	Very Satisfied
Navigating to Categories Charts Screen	Yes	1	None	Very Satisfied
Navigating to Financial Charts Screen	Yes	1	None	Very Satisfied
Navigating to Time Series Charts Screen	Yes	1	None	Very Satisfied
Changing Page to Explore Data Page	Yes	1	None	Very Satisfied
Create a Box Plot with Web Tools	Yes	1	None	Very Satisfied
Create a Histogram with Web Tools	Yes	2	None	Very Satisfied
Create a Bar Plot with Web Tools	Yes	1	None	Very Satisfied
Create a Scatter Graph with Web Tools	Yes	1	None	Very Satisfied

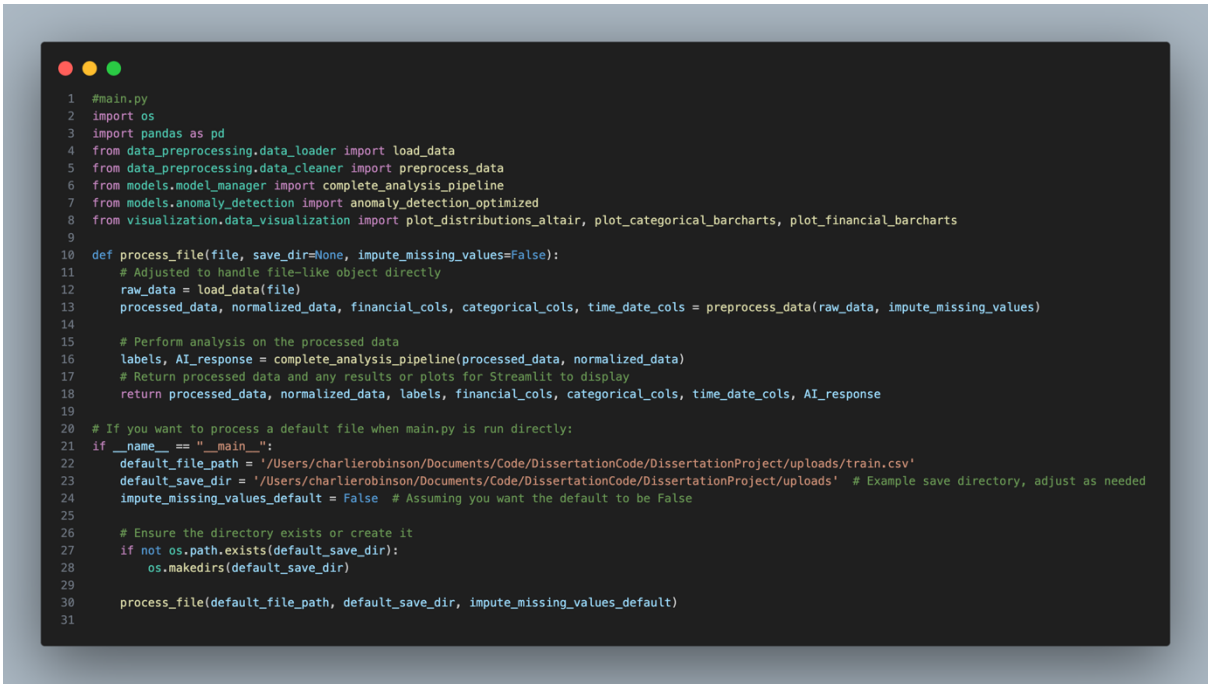
Create a Scatter Graph with a Regression Line	Yes	2	None	Very Satisfied
Upload a New Dataset	Yes	1	None	Very Satisfied

Appendix 2 - User 2 Usability Test Results

Objectives	**Explanation**	**What was done**
1 – Literature Review	Conduct a review of literature on data visualization and AI algorithms by 19th Feb identifying key strengths, limitations, and improvement areas in existing AI-driven data visualization tools. Measure success by the number of tools analysed and the comprehensiveness of the review.	The comprehensive review was completed, providing a solid foundation for understanding the current landscape of AI-driven visualization tools. This review identified key areas for improvement, particularly in user engagement and intuitive interfaces, which were directly addressed in the tool's design.
2 – User Engagement	Identify potential users through interviews, surveys, or focus groups by 4th March, gathering insights on their needs and challenges. Measure success by the number of participants engaged and the depth of insights gathered.	Initial user surveys and continuous feedback loops were integral to the design process, ensuring that the tool's development was guided by actual user needs and preferences. This objective was met through iterative testing and refinement phases that incorporated user feedback directly into design improvements such as download options and help tooltips.
3 – Project Design	Design AI-driven data visualization prototypes that incorporate advanced machine learning algorithms for pattern recognition by 18th March, ensuring compatibility with various datasets. Success is measured by the prototype's adherence to design specifications and compatibility standards.	Prototypes were developed, focusing on usability and the incorporation of machine learning algorithms for automated insight generation. These prototypes were instrumental in refining the tool’s functionality as it took lots of tuning of the models to achieve the best results and
4 – Project Development	Develop the designed prototypes into functional tools by 15th April, integrating AI capabilities to enhance data visualization.	The tool was fully developed, integrating the planned AI functionalities, and ensuring it could handle various data types

	Measure success by development milestones, functionality, and AI integration effectiveness. Development phases are Data loading, Data Analysis and Data Visualisation.	effectively. This development was consistent with the prototypes and user feedback, ensuring high functionality and compatibility.
5 – Usability Evaluation	Conduct usability testing with target users by 25th April, to assess tool effectiveness, usability, and performance, comparing it with traditional methods. Measure success through usability scores, user feedback, and performance benchmarks across the three steps listed in project development.	Usability testing confirmed the tool's effectiveness, usability, and performance. Feedback from these sessions was crucial in finalizing the tool, especially from both an experienced and inexperienced users' viewpoint to get varied opinions across the board highlighting its practical utility and ease of use.
Objective 6 – Task Prioritization and Management	To implement a rigorous task prioritization framework by 11TH March, ensuring that high-priority tasks essential to the project's core objectives are identified and addressed first. This will involve assessing each task's impact on project outcomes, resource requirements, and dependencies. Measure success by the on-time completion of critical project milestones as a Minimal Viable Product (MVP). Followed by effective allocation of resources to complex tasks, thereby mitigating the risk of project delays due to time constraints.	Evaluation and Iteration: Successfully implemented a rigorous task prioritization framework, ensuring high-priority tasks crucial to the project's core objectives were identified and addressed first. This objective was achieved through a systematic assessment of each task's impact on project outcomes, resource allocation, and dependencies, leading to the on-time completion of critical milestones in the form of a Minimal Viable Product (MVP). This strategic focus facilitated the effective allocation of resources to more complex tasks, significantly mitigating the risk of project delays due to time constraints and ensuring continual progress toward project goals.

Appendix 3 – Comparison to Initial Objectives

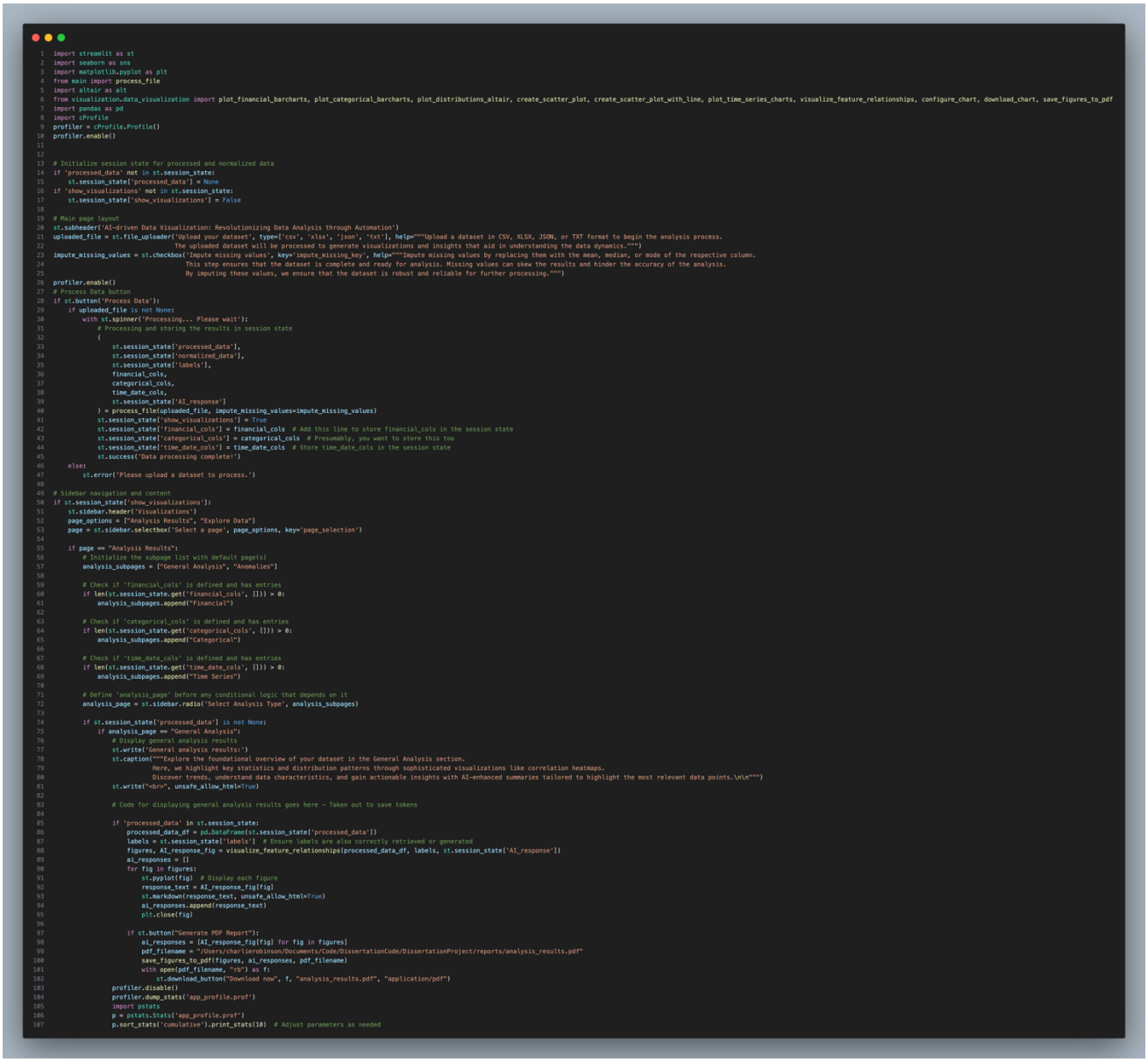


```python
#main.py
import os
import pandas as pd
from data_preprocessing.data_loader import load_data
from data_preprocessing.data_cleaner import preprocess_data
from models.model_manager import complete_analysis_pipeline
from models.anomaly_detection import anomaly_detection_optimized
from visualization.data_visualization import plot_distributions_altair, plot_categorical_barcharts, plot_financial_barcharts

def process_file(file, save_dir=None, impute_missing_values=False):
    # Adjusted to handle file-like object directly
    raw_data = load_data(file)
    processed_data, normalized_data, financial_cols, categorical_cols, time_date_cols = preprocess_data(raw_data, impute_missing_values)

    # Perform analysis on the processed data
    labels, AI_response = complete_analysis_pipeline(processed_data, normalized_data)
    # Return processed data and any results or plots for Streamlit to display
    return processed_data, normalized_data, labels, financial_cols, categorical_cols, time_date_cols, AI_response

# If you want to process a default file when main.py is run directly:
if __name__ == "__main__":
    default_file_path = '/Users/charlierobinson/Documents/Code/DissertationCode/DissertationProject/uploads/train.csv'
    default_save_dir = '/Users/charlierobinson/Documents/Code/DissertationCode/DissertationProject/uploads'  # Example save directory, adjust as needed
    impute_missing_values_default = False  # Assuming you want the default to be False

    # Ensure the directory exists or create it
    if not os.path.exists(default_save_dir):
        os.makedirs(default_save_dir)

    process_file(default_file_path, default_save_dir, impute_missing_values_default)

```

Appendix 4 - Main.py

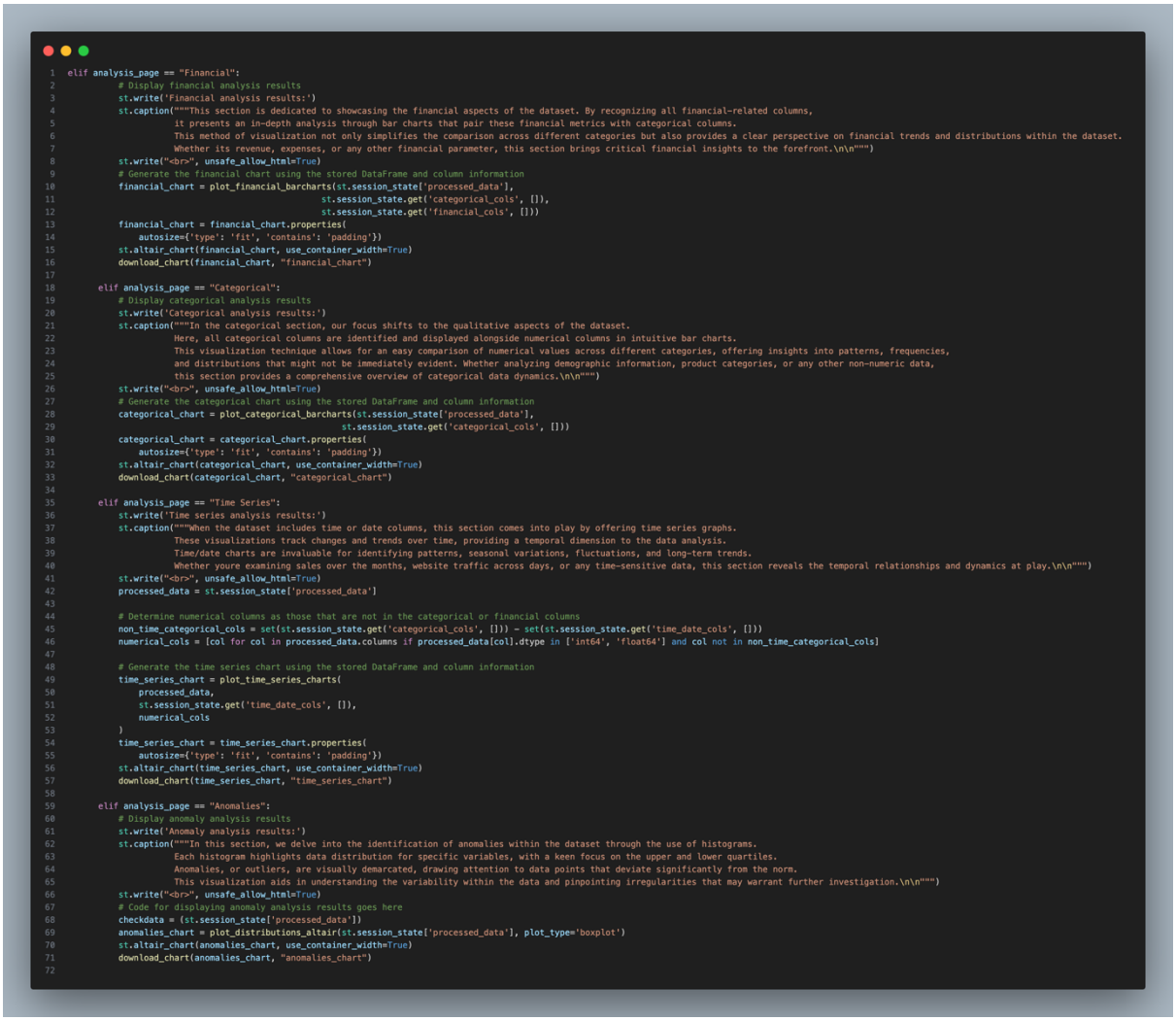
```python
#OpenAI
from dotenv import load_dotenv
from openai import OpenAI
import os
import time
import json
import streamlit as st
import re

load_dotenv()

def generate_summary(descriptions, retry_lmit=3):
  client = OpenAI(
    api_key=os.environ.get("OPENAI_API_KEY"),
  )
  new_descriptions = ', '.join(descriptions)

  attempts = 0
  while attempts < retry_lmit:
    response = client.chat.completions.create(
        model="gpt-3.5-turbo-0125",
        max_tokens=4096,
        temperature=0.7,
        messages=[
            {"""role": "system", "content": "Your task is to generate a JSON structure containing trends of data.
            Each trend should include details such as 'trend name', 'heading', 'description', 'buzzwords', and 'key points'.
             The JSON structure must start with a 'trends' key, which holds an array of trend objects.
             Each object in the array represents a trend and should provide a detailed, easily understandable description of at least 100 words,
              ensuring clarity for audiences unfamiliar with data science. Use plain language and avoid technical jargon.
             Descriptions should contextualize the data effectively, incorporating the trend name seamlessly.
             Key points should be semicolon delimited, and there should be more than one key point for each trend.
             The heading should be concise, with a maximum of 5 words. Include text formatting but do not assume
             it's about any particular subject or to any particular audience. Ensure the response adheres strictly to this JSON format."""},
            {"role": "user", "content": new_descriptions},
        ],
    )

    #print(response.choices[0].message.content)

    try:
        data = json.loads(response.choices[0].message.content)
    except json.JSONDecodeError as e:
        print(f"JSON decoding failed: {e}")
        attempts += 1
        time.sleep(20)  # Wait a bit before retrying
        continue

    combined_array = []
    if isinstance(data.get('trends'), list):
        for trend in data['trends']:
            heading = trend.get('heading', 'No Heading')
            description = trend.get('description', 'No Description')
            buzzwords = trend.get('buzzwords', 'No Buzzwords')
            key_points_str = trend.get('key points', '')
            key_points_list = key_points_str.split('; ')

            key_points_html = '<ul>' + ''.join(f'<li>{point}</li>' for point in key_points_list) + '</ul>'

            combined_text = f"""
            <div style='font-weight: bold;'>{heading}</div>
            <div>{description}</div>
            <div style='font-weight: bold;'>Buzzwords:</div>
            <div>{buzzwords}</div>
            <div style='font-weight: bold;'>Key points:</div>
            {key_points_html}
            """

            combined_array.append(combined_text)

        return combined_array  # Successful processing, return the result
    else:
        print("'trends' key is missing or not in expected format. Retrying...")
        attempts += 1
        time.sleep(20)  # Wait a bit before retrying
```

Appendix 5 - OpenAI.py



```python
import streamlit as st
import seaborn as sns
import matplotlib.pyplot as plt
from main import process_file
import altair as alt
from visualization.data_visualization import plot_financial_barcharts, plot_categorical_barcharts, plot_distributions_altair, create_scatter_plot, create_scatter_plot_with_line, plot_time_series_charts, visualize_feature_relationships, configure_chart, download_chart, save_figures_to_pdf
import pandas as pd
import cProfile
profiler = cProfile.Profile()
profiler.enable()


# Initialize session state for processed and normalized data
if 'processed_data' not in st.session_state:
    st.session_state['processed_data'] = None
if 'show_visualizations' not in st.session_state:
    st.session_state['show_visualizations'] = False

# Main page layout
st.subheader('AI-driven Data Visualization: Revolutionizing Data Analysis through Automation')
uploaded_file = st.file_uploader('Upload your dataset', type=['csv', 'xlsx', 'json', 'txt'], help="""Upload a dataset in CSV, XLSX, JSON, or TXT format to begin the analysis process.
                                 The uploaded dataset will be processed to generate visualizations and insights that aid in understanding the data dynamics.""")
impute_missing_values = st.checkbox('Impute missing values', key='impute_missing_key', help="""Impute missing values by replacing them with the mean, median, or mode of the respective column.
                                    This step ensures that the dataset is complete and ready for analysis. Missing values can skew the results and hinder the accuracy of the analysis.
                                    By imputing these values, we ensure that the dataset is robust and reliable for further processing.""")
profiler.enable()
# Process Data button
if st.button('Process Data'):
    if uploaded_file is not None:
        with st.spinner('Processing... Please wait'):
            # Processing and storing the results in session state
            (
                st.session_state['processed_data'],
                st.session_state['normalized_data'],
                st.session_state['labels'],
                financial_cols,
                categorical_cols,
                time_date_cols,
                st.session_state['AI_response']
            ) = process_file(uploaded_file, impute_missing_values=impute_missing_values)
            st.session_state['show_visualizations'] = True
            st.session_state['financial_cols'] = financial_cols  # Add this line to store financial_cols in the session state
            st.session_state['categorical_cols'] = categorical_cols  # Presumably, you want to store this too
            st.session_state['time_date_cols'] = time_date_cols  # Store time_date_cols in the session state
            st.success('Data processing complete!')
    else:
        st.error('Please upload a dataset to process.')

# Sidebar navigation and content
if st.session_state['show_visualizations']:
    st.sidebar.header('Visualizations')
    page_options = ["Analysis Results", "Explore Data"]
    page = st.sidebar.selectbox('Select a page', page_options, key='page_selection')

    if page == "Analysis Results":
        # Initialize the subpage list with default page(s)
        analysis_subpages = ["General Analysis", "Anomalies"]

        # Check if 'financial_cols' is defined and has entries
        if len(st.session_state.get('financial_cols', [])) > 0:
            analysis_subpages.append("Financial")

        # Check if 'categorical_cols' is defined and has entries
        if len(st.session_state.get('categorical_cols', [])) > 0:
            analysis_subpages.append("Categorical")

        # Check if 'time_date_cols' is defined and has entries
        if len(st.session_state.get('time_date_cols', [])) > 0:
            analysis_subpages.append("Time Series")

        # Define 'analysis_page' before any conditional logic that depends on it
        analysis_page = st.sidebar.radio('Select Analysis Type', analysis_subpages)

        if st.session_state['processed_data'] is not None:
            if analysis_page == "General Analysis":
                # Display general analysis results
                st.write('General analysis results:')
                st.caption("""Explore the foundational overview of your dataset in the General Analysis section.
                           Here, we highlight key statistics and distribution patterns through sophisticated visualizations like correlation heatmaps.
                           Discover trends, understand data characteristics, and gain actionable insights with AI-enhanced summaries tailored to highlight the most relevant data points.\n\n""")
                st.write("<br>", unsafe_allow_html=True)

                # Code for displaying general analysis results goes here - Taken out to save tokens

                if 'processed_data' in st.session_state:
                    processed_data_df = pd.DataFrame(st.session_state['processed_data'])
                    labels = st.session_state['labels']  # Ensure labels are also correctly retrieved or generated
                    figures, AI_response_fig = visualize_feature_relationships(processed_data_df, labels, st.session_state['AI_response'])
                    ai_responses = []
                    for fig in figures:
                        st.pyplot(fig)  # Display each figure
                        response_text = AI_response_fig[fig]
                        st.markdown(response_text, unsafe_allow_html=True)
                        ai_responses.append(response_text)
                        plt.close(fig)

                    if st.button("Generate PDF Report"):
                        ai_responses = [AI_response_fig[fig] for fig in figures]
                        pdf_filename = "/Users/charlierobinson/Documents/Code/DissertationCode/DissertationProject/reports/analysis_results.pdf"
                        save_figures_to_pdf(figures, ai_responses, pdf_filename)
                        with open(pdf_filename, "rb") as f:
                            st.download_button("Download now", f, "analysis_results.pdf", "application/pdf")
                profiler.disable()
                profiler.dump_stats('app_profile.prof')
                import pstats
                p = pstats.Stats('app_profile.prof')
                p.sort_stats('cumulative').print_stats(10)  # Adjust parameters as needed
```

Appendix 6 - Streamlit Part1



```
elif analysis_page == "Financial":
        # Display financial analysis results
        st.write('Financial analysis results:')
        st.caption("""This section is dedicated to showcasing the financial aspects of the dataset. By recognizing all financial-related columns,
                   it presents an in-depth analysis through bar charts that pair these financial metrics with categorical columns.
                   This method of visualization not only simplifies the comparison across different categories but also provides a clear perspective on financial trends and distributions within the dataset.
                   Whether its revenue, expenses, or any other financial parameter, this section brings critical financial insights to the forefront.\n\n""")
        st.write("<br>", unsafe_allow_html=True)
        # Generate the financial chart using the stored DataFrame and column information
        financial_chart = plot_financial_barcharts(st.session_state['processed_data'],
                                                   st.session_state.get('categorical_cols', []),
                                                   st.session_state.get('financial_cols', []))
        financial_chart = financial_chart.properties(
            autosize={'type': 'fit', 'contains': 'padding'})
        st.altair_chart(financial_chart, use_container_width=True)
        download_chart(financial_chart, "financial_chart")

    elif analysis_page == "Categorical":
        # Display categorical analysis results
        st.write('Categorical analysis results:')
        st.caption("""In the categorical section, our focus shifts to the qualitative aspects of the dataset.
                   Here, all categorical columns are identified and displayed alongside numerical columns in intuitive bar charts.
                   This visualization technique allows for an easy comparison of numerical values across different categories, offering insights into patterns, frequencies,
                   and distributions that might not be immediately evident. Whether analyzing demographic information, product categories, or any other non-numeric data,
                   this section provides a comprehensive overview of categorical data dynamics.\n\n""")
        st.write("<br>", unsafe_allow_html=True)
        # Generate the categorical chart using the stored DataFrame and column information
        categorical_chart = plot_categorical_barcharts(st.session_state['processed_data'],
                                                       st.session_state.get('categorical_cols', []))
        categorical_chart = categorical_chart.properties(
            autosize={'type': 'fit', 'contains': 'padding'})
        st.altair_chart(categorical_chart, use_container_width=True)
        download_chart(categorical_chart, "categorical_chart")

    elif analysis_page == "Time Series":
        st.write('Time series analysis results:')
        st.caption("""When the dataset includes time or date columns, this section comes into play by offering time series graphs.
                   These visualizations track changes and trends over time, providing a temporal dimension to the data analysis.
                   Time/date charts are invaluable for identifying patterns, seasonal variations, fluctuations, and long-term trends.
                   Whether youre examining sales over the months, website traffic across days, or any time-sensitive data, this section reveals the temporal relationships and dynamics at play.\n\n""")
        st.write("<br>", unsafe_allow_html=True)
        processed_data = st.session_state['processed_data']

        # Determine numerical columns as those that are not in the categorical or financial columns
        non_time_categorical_cols = set(st.session_state.get('categorical_cols', [])) - set(st.session_state.get('time_date_cols', []))
        numerical_cols = [col for col in processed_data.columns if processed_data[col].dtype in ['int64', 'float64'] and col not in non_time_categorical_cols]

        # Generate the time series chart using the stored DataFrame and column information
        time_series_chart = plot_time_series_charts(
            processed_data,
            st.session_state.get('time_date_cols', []),
            numerical_cols
        )
        time_series_chart = time_series_chart.properties(
            autosize={'type': 'fit', 'contains': 'padding'})
        st.altair_chart(time_series_chart, use_container_width=True)
        download_chart(time_series_chart, "time_series_chart")

    elif analysis_page == "Anomalies":
        # Display anomaly analysis results
        st.write('Anomaly analysis results:')
        st.caption("""In this section, we delve into the identification of anomalies within the dataset through the use of histograms.
                   Each histogram highlights data distribution for specific variables, with a keen focus on the upper and lower quartiles.
                   Anomalies, or outliers, are visually demarcated, drawing attention to data points that deviate significantly from the norm.
                   This visualization aids in understanding the variability within the data and pinpointing irregularities that may warrant further investigation.\n\n""")
        st.write("<br>", unsafe_allow_html=True)
        # Code for displaying anomaly analysis results goes here
        checkdata = (st.session_state['processed_data'])
        anomalies_chart = plot_distributions_altair(st.session_state['processed_data'], plot_type='boxplot')
        st.altair_chart(anomalies_chart, use_container_width=True)
        download_chart(anomalies_chart, "anomalies_chart")
```

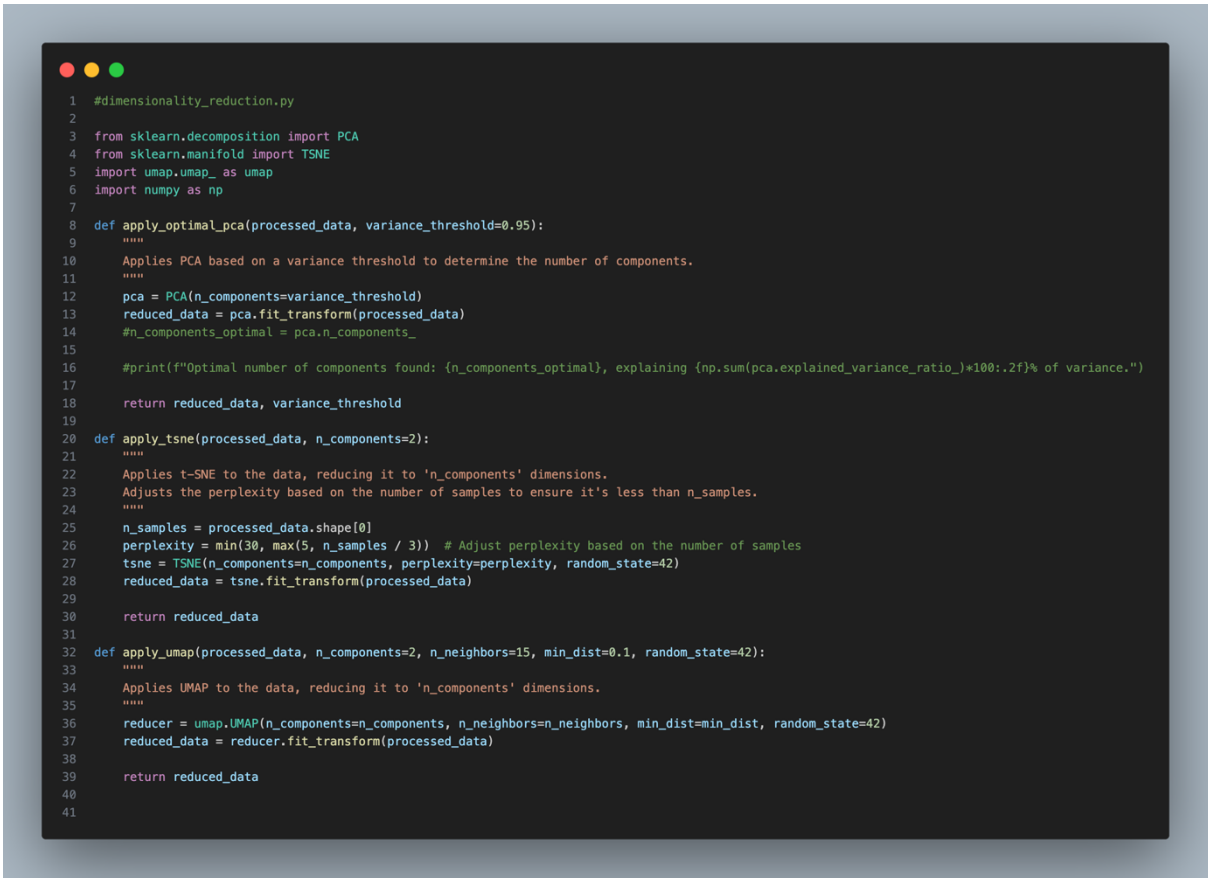
Appendix 7 - Streamlit Part2

```
elif page == 'Explore Data':
    st.write('Explore Data')
    # Sidebar options for chart selection
    st.sidebar.header("Charts and Plots")
    plot_options = ["Bar plot", "Scatter plot", "Histogram", "Box plot"]
    selected_plot = st.sidebar.selectbox("Choose a plot type", plot_options, key='selected_plot')

    # 'processed_data' has the data to plot
    data = st.session_state['processed_data']


    # Only create charts if data is available
    if data is not None and selected_plot:
        column_options = data.columns.tolist()

        if selected_plot == "Bar plot":
            x_axis = st.sidebar.selectbox("Select category axis", column_options, key='x_axis')
            y_axis = st.sidebar.selectbox("Select value axis", column_options, key='y_axis')
            chart = alt.Chart(data).mark_bar().encode(
                x=x_axis,
                y=y_axis
            ).properties(
                width=600,
                height=300,
                background='white',
                padding={"left": 10, "right": 10, "top": 10, "bottom": 10}
            ).configure_view(
                stroke='transparent'
            ).configure_axis(
                labelColor='black',
                titleColor='black',
                gridColor='black',
                domainColor='black',
                tickColor='black'
            ).configure_title(
                color='black'
            )

            chart = configure_chart(chart)
            st.altair_chart(chart, use_container_width=True)
            download_chart(chart, "bar_plot")

        elif selected_plot == "Scatter plot":
            x_axis = st.sidebar.selectbox("Select x-axis", column_options, key='x_axis_scatter')
            y_axis = st.sidebar.selectbox("Select y-axis", column_options, key='y_axis_scatter')
            show_regression = st.sidebar.checkbox("Show Regression Line", value=False)
            if show_regression:
                chart = create_scatter_plot_with_line(data, x_axis, y_axis)
            else:
                chart = create_scatter_plot(data, x_axis, y_axis)
            chart = configure_chart(chart)
            st.altair_chart(chart, use_container_width=True)
            download_chart(chart, "scatter_plot")

        elif selected_plot == "Histogram":
            column = st.sidebar.selectbox("Select a column for histogram", column_options, key='hist_column')
            bins = st.sidebar.slider("Number of bins", min_value=1, max_value=100, value=30, key='hist_bins')
            chart = alt.Chart(data).mark_bar().encode(
                alt.X(column, bin=alt.Bin(maxbins=bins)),
                y='count()'
            ).properties(
                width=600,
                height=300,
                background='white',
                padding={"left": 10, "right": 10, "top": 10, "bottom": 10}
            ).configure_view(
                stroke='transparent'
            ).configure_axis(
                labelColor='black',
                titleColor='black',
                gridColor='black',
                domainColor='black',
                tickColor='black'
            ).configure_title(
                color='black'
            )
            chart = configure_chart(chart)
            st.altair_chart(chart, use_container_width=True)
            download_chart(chart, "histogram")

        elif selected_plot == "Box plot":
            x_axis = st.sidebar.selectbox("Select category axis", column_options, key='x_axis_box')
            y_axis = st.sidebar.selectbox("Select value axis", column_options, key='y_axis_box')

            # Ensure the y-axis is treated as a quantitative variable
            if data[y_axis].dtype not in ['float64', 'int64']:
                st.warning(f"The selected value axis '{y_axis}' is not continuous.")
            else:
                chart = alt.Chart(data).mark_boxplot().encode(
                    x=x_axis,
                    y=y_axis
            ).properties(
                width=600,
                height=300,
                background='white',
                padding={"left": 10, "right": 10, "top": 10, "bottom": 10}
            ).configure_view(
                stroke='transparent'
            ).configure_axis(
                labelColor='black',
                titleColor='black',
                gridColor='black',
                domainColor='black',
                tickColor='black'
            ).configure_title(
                color='black'
            )
                st.altair_chart(chart, use_container_width=True)
                download_chart(chart, "box_plot")



        else:
            st.warning("Please select at least one feature to include in the model.")
```

Appendix 8 - Streamlit Part 3

```python
# model_manager.py
from .clustering import apply_kmeans, apply_dbscan
from .clustering import optimal_kmeans, optimal_dbscan
from sklearn.model_selection import train_test_split
from models.dimensionality_reduction import apply_optimal_pca, apply_tsne, apply_umap
from models.anomaly_detection import anomaly_detection_optimized  # Assume this function exists in your anomaly_detection.py
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.metrics import silhouette_score
from sklearn.cluster import KMeans
from visualization.data_visualization import plot_distributions_altair
import matplotlib
matplotlib.use('Agg')  # Use the Anti-Grain Geometry non-GUI backend suited for scripts and web deployment
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from OpenAI import generate_summary
import streamlit as st


def complete_analysis_pipeline(data, normalized_data):
    # Ensure 'data' is a DataFrame
    if not isinstance(data, pd.DataFrame):
        raise ValueError("Data needs to be a pandas DataFrame.")

    # Dictionary to store reduced data for each method
    reduced_data_methods = {}
    silhouette_scores = {}

    # Apply each dimensionality reduction method
    reduced_data_methods['PCA'], _ = apply_optimal_pca(normalized_data, 2)
    reduced_data_methods['t-SNE'] = apply_tsne(normalized_data)
    reduced_data_methods['UMAP'] = apply_umap(normalized_data)

    # Calculate silhouette score for each reduced data and find the best one
    for method, reduced_data in reduced_data_methods.items():
        # Temporarily apply KMeans for silhouette score calculation; adjust based on your
        #criteria or data
        #trend_labels = KMeans(n_trends=5, random_state=42).fit_predict(reduced_data)
        trend_labels = optimal_kmeans(reduced_data)
        score = silhouette_score(reduced_data, trend_labels)
        silhouette_scores[method] = score
        #print(f"{method} silhouette score: {score}")

    best_method = max(silhouette_scores, key=silhouette_scores.get)
    #print(f"Best dimensionality reduction method: {best_method} with a
    #silhouette score of {silhouette_scores[best_method]}")

    # Perform trending on the best reduced data
    best_reduced_data = reduced_data_methods[best_method]
    labels = choose_and_apply_trending(best_reduced_data)
    descriptions = generate_trend_descriptions(data, labels)
    AI_response={}
    AI_response = generate_summary(descriptions)
    #print(normalized_data)

    return labels, AI_response



def generate_trend_descriptions(data, trend_labels, numeric_metric='var', diff_metric='mean'):
    data['trend'] = trend_labels
    numeric_cols = data.select_dtypes(include=np.number).columns.tolist()
    categorical_cols = data.select_dtypes(exclude=np.number).columns.tolist()
    numeric_cols.remove('trend')

    all_descriptions = []

    for trend_id in np.unique(trend_labels):
        trend_data = data[data['trend'] == trend_id]
        other_trends_data = data[data['trend'] != trend_id]

        # Section 1: Standout Fields
        standout_desc = f"trend {trend_id+1} standout fields are "
        standout_fields = []

        # Numeric: Based on specified metric
        if numeric_metric == 'var':
            metric_values = trend_data[numeric_cols].var()
        elif numeric_metric == 'std':
            metric_values = trend_data[numeric_cols].std()
        else:
            raise ValueError("Unsupported numeric_metric provided.")

        top_numeric = metric_values.nlargest(3)
        for field in top_numeric.index:
            avg_val = trend_data[field].mean()
            standout_fields.append(f"{field} (average: {avg_val:.2f})")

        # Categorical: Most significant based on frequency
        if categorical_cols:
            cat_diffs = {}
            for col in categorical_cols:
                mode = trend_data[col].mode()[0] if not trend_data[col].mode().empty else 'N/A'
                mode_freq = trend_data[col].value_counts(normalize=True).get(mode, 0)
                cat_diffs[col] = mode_freq
            if cat_diffs:
                top_cat = max(cat_diffs, key=cat_diffs.get)
                top_mode = trend_data[top_cat].mode()[0] if not trend_data[top_cat].mode().empty else 'N/A'
                standout_fields.append(f"{top_cat} (most common: {top_mode})")

        standout_desc += "; ".join(standout_fields) + "."

        # Section 2: Differentiation from Other trends
        diff_desc = f"\nHow trend {trend_id+1} differs: "
        diff_fields = []
        for field in top_numeric.index:
            trend_avg = trend_data[field].mean() if diff_metric == 'mean' else trend_data[field].median()
            other_avg = other_trends_data[field].mean() if diff_metric == 'mean' else other_trends_data[field].median()
            difference = "higher" if trend_avg > other_avg else "lower"
            diff_val = abs(trend_avg - other_avg)
            diff_fields.append(f"{field} is {difference} than the average of other trends by {diff_val:.2f}")

        if categorical_cols and cat_diffs:
            mode_freq_other_trends = other_trends_data[top_cat].value_counts(normalize=True).get(top_mode, 0)
            freq_diff = cat_diffs[top_cat] - mode_freq_other_trends
            freq_desc = "more common" if freq_diff > 0 else "less common"
            diff_fields.append(f"{top_mode} in {top_cat} is {freq_desc} compared to other trends")

        diff_desc += "; ".join(diff_fields) + "."

        # Combine both sections for the trend's description
        trend_description = standout_desc + diff_desc
        all_descriptions.append(trend_description)
        print("\n\nAll descriptions: ", all_descriptions)

    return all_descriptions


def choose_and_apply_trending(data):
    # Example simplistic criteria: Dataset size
    if len(data) < 1:  # Assuming larger datasets might have more complex trend shapes
        #print("Using DBSCAN...")
        labels = optimal_dbscan(data)
    else:
        #print("Using KMeans...")
        labels = optimal_kmeans(data)

    return labels
```

Appendix 9 - Model_Manager.py



```python
#dimensionality_reduction.py

from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
import umap.umap_ as umap
import numpy as np

def apply_optimal_pca(processed_data, variance_threshold=0.95):
    """
    Applies PCA based on a variance threshold to determine the number of components.
    """
    pca = PCA(n_components=variance_threshold)
    reduced_data = pca.fit_transform(processed_data)
    #n_components_optimal = pca.n_components_

    #print(f"Optimal number of components found: {n_components_optimal}, explaining {np.sum(pca.explained_variance_ratio_)*100:.2f}% of variance.")

    return reduced_data, variance_threshold

def apply_tsne(processed_data, n_components=2):
    """
    Applies t-SNE to the data, reducing it to 'n_components' dimensions.
    Adjusts the perplexity based on the number of samples to ensure it's less than n_samples.
    """
    n_samples = processed_data.shape[0]
    perplexity = min(30, max(5, n_samples / 3))  # Adjust perplexity based on the number of samples
    tsne = TSNE(n_components=n_components, perplexity=perplexity, random_state=42)
    reduced_data = tsne.fit_transform(processed_data)

    return reduced_data

def apply_umap(processed_data, n_components=2, n_neighbors=15, min_dist=0.1, random_state=42):
    """
    Applies UMAP to the data, reducing it to 'n_components' dimensions.
    """
    reducer = umap.UMAP(n_components=n_components, n_neighbors=n_neighbors, min_dist=min_dist, random_state=42)
    reduced_data = reducer.fit_transform(processed_data)

    return reduced_data
```

Appendix 10 - Dimensionality_Reduction.py

```python
# trending.py

from sklearn.cluster import KMeans, DBSCAN
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import NearestNeighbors
from kneed import KneeLocator
import numpy as np
import matplotlib.pyplot as plt


def optimal_kmeans(data, max_clusters=5):
    wcss = []
    for i in range(1, max_clusters + 1):  # Limit the range to a maximum of 4
        kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10, random_state=42)
        kmeans.fit(data)
        wcss.append(kmeans.inertia_)

    # Automatically find the elbow point, limited to the range [1, max_trends]
    kn = KneeLocator(range(1, max_clusters + 1), wcss, curve='convex', direction='decreasing')
    n_clusters_optimal = kn.knee
    #print(f"Optimal number of trends: {n_clusters_optimal}")

    return apply_kmeans(data, n_clusters=n_clusters_optimal)

def apply_kmeans(data, n_clusters=3):
    if n_clusters is None or n_clusters < 1:
        n_clusters = 1
    # Perform K-means trending with specified number of trends
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    labels = kmeans.fit_predict(data)
    return labels

def optimal_dbscan(data):
    # Find the nearest neighbors
    neighbors = NearestNeighbors(n_neighbors=2)
    neighbors_fit = neighbors.fit(data)
    distances, indices = neighbors_fit.kneighbors(data)

    distances = np.sort(distances, axis=0)[:,1]

    # Automatically find the knee point as the optimal eps value
    kn = KneeLocator(np.arange(len(distances)), distances, curve='convex', direction='increasing')
    eps_optimal = distances[kn.knee]
    #print(f"Optimal eps value: {eps_optimal}")

    return apply_dbscan(data, eps=eps_optimal, min_samples=9)

def apply_dbscan(data, eps=0.5, min_samples=5):
    # Perform DBSCAN trending with specified eps and min_samples values
    dbscan = DBSCAN(eps=eps, min_samples=min_samples)
    labels = dbscan.fit_predict(data)
    return labels
```

Appendix 11 - Clustering.py

```python
import pandas as pd

def load_data(file):
    # Debugging: Print the file type to help diagnose issues
    print(f"Loading file: {file.name}, type: {type(file)}")

    try:
        # Determine the file extension and choose the loading method accordingly
        if str(file.name).endswith('.csv'):
            # Attempt to read a CSV file
            try:
                return pd.read_csv(file, encoding='utf-8')
            except UnicodeDecodeError:
                try:
                    return pd.read_csv(file, encoding='ISO-8859-1')  # Try latin1 encoding
                except UnicodeDecodeError:
                    return pd.read_csv(file, encoding='cp1252')  # Try Windows-1252 encoding
            except pd.errors.EmptyDataError:
                print("The file is empty. Please upload a valid file.")
                return None
        elif str(file.name).endswith('.xlsx'):
            return pd.read_excel(file)
        elif str(file.name).endswith('.json'):
            return pd.read_json(file)
        elif str(file.name).endswith('.txt'):
            return pd.read_csv(file, delimiter='\t')
    except Exception as e:
        # Catch any other exception and log it
        print(f"Failed to load file due to: {e}")
        return None

```

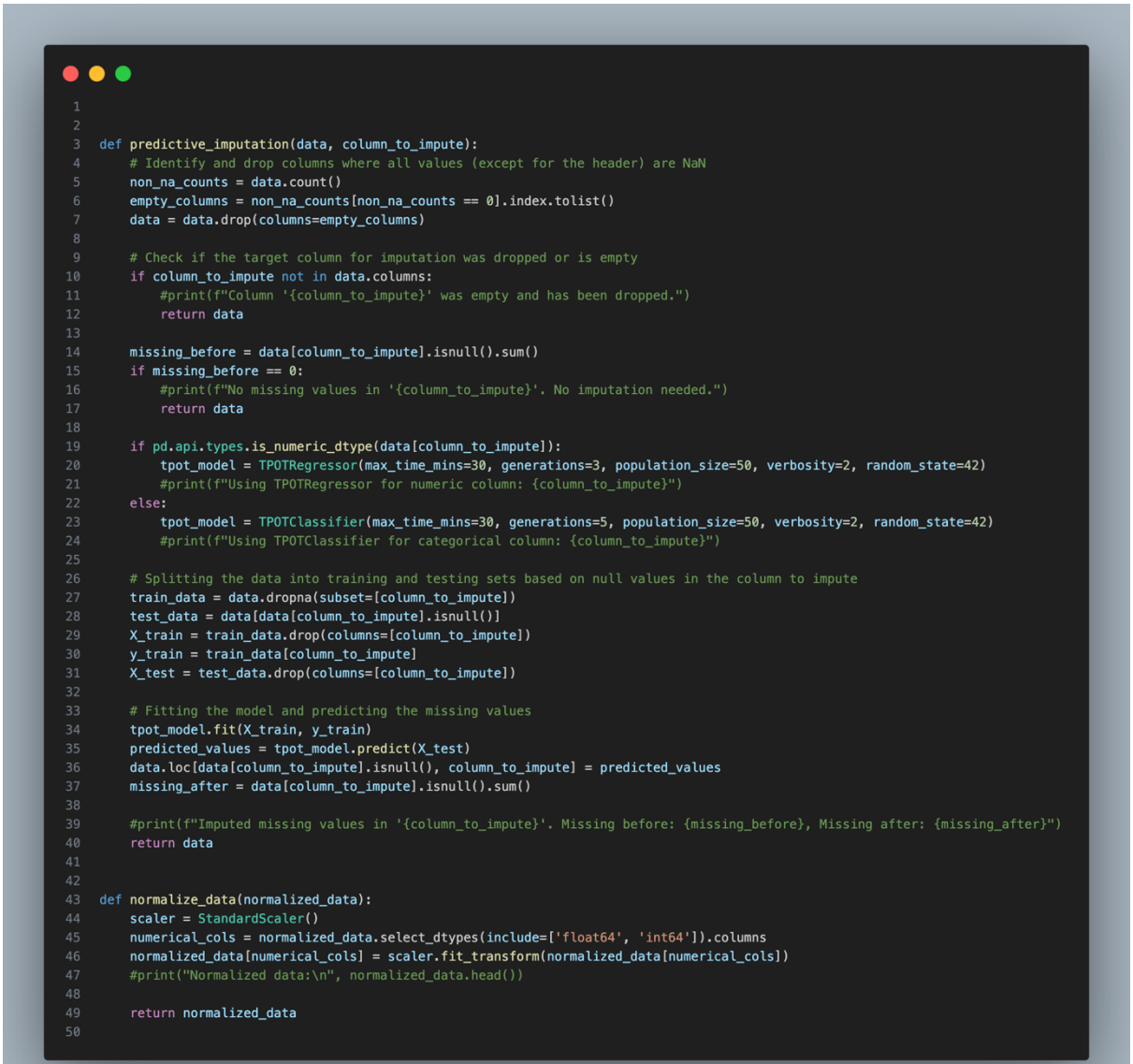
Appendix 12 – Data_loader.py

```python
# datacleaner.py
from tpot import TPOTRegressor, TPOTClassifier
import pandas as pd
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder
import numpy as np
from tpot import TPOTRegressor, TPOTClassifier
import pandas as pd
from sklearn.preprocessing import OneHotEncoder, LabelEncoder, StandardScaler
import pandas as pd
import numpy as np
from dateutil import parser
import re


def preprocess_data(data, handle_missing_values):

    data = drop_id_columns(data)
    financial_cols = detect_financial_columns(data)
    time_date_cols, converted_data, converted_to_normalize = detect_time_date_columns(data)
    data = converted_data
    #print("Time/Date columns:", time_date_cols)
    #print(data.dtypes)
    #print("Preprocessed Data Test: \n", data.head(10))
    #print(converted_to_normalize.dtypes)
    categorical_cols = data.select_dtypes(include=['object']).columns

    normalized_data = converted_to_normalize.copy()
    print("Normalized data test:\n\n", normalized_data.head(10),"\n\n")

    if handle_missing_values == False:
        #print("Dropping rows with missing values.")
        num_rows_dropped = len(data) - len(data.dropna())
        data.dropna(inplace=True)
        normalized_data.dropna(inplace=True)
        #print(f"Dropped {num_rows_dropped} rows with missing values.")


    # Initialize encoders
    one_hot_encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
    label_encoder = LabelEncoder()

    for col in normalized_data.columns:
        if normalized_data[col].dtype == 'object':
            # Fill NaN values in categorical columns with a placeholder
            #normalized_data[col].fillna('missing', inplace=True)

            # Determine encoding strategy based on the number of unique values
            unique_values = normalized_data[col].nunique()
            if unique_values <= 5:
                # OneHot encode if unique values are 5 or less
                encoded = one_hot_encoder.fit_transform(normalized_data[[col]])
                encoded_df = pd.DataFrame(encoded, columns=one_hot_encoder.get_feature_names_out([col]), index=normalized_data.index)
                normalized_data = pd.concat([normalized_data.drop(columns=[col]), encoded_df], axis=1)
            else:
                # Label encode if unique values are more than 5
                normalized_data[col] = label_encoder.fit_transform(normalized_data[col])

    if handle_missing_values != False:
        normalized_data = handle_missing_values_with_tpot(normalized_data)

    normalized_data = normalize_data(normalized_data)

    return data, normalized_data, financial_cols, categorical_cols, time_date_cols

def detect_financial_columns(data):
    financial_keywords = [
        'revenue', 'cost', 'profit', 'expense', 'income', 'gross', 'salary',
        'dollar', 'dollars', 'euro', 'pound', 'pounds', 'sterling', 'yen',
        'rupee', 'ruble', 'real', 'peso', 'franc', 'lira', 'rand', 'krona',
        'won', 'yuan', 'renminbi', 'euros', 'pounds', 'dollars', 'rupees',
    ]

    financial_cols = []
    for col in data.columns:
        # Check for full-word match in column name
        if any(re.search(r'\b{}\b'.format(keyword), col.lower()) for keyword in financial_keywords):
            financial_cols.append(col)
            continue  # If the keyword is found in the column name, no need to check for symbols in its data

        # Check for presence of currency symbols in the data of the column
        if data[col].astype(str).str.contains(r'[\$\£\€]', regex=True).any():
            financial_cols.append(col)

    return financial_cols


def handle_missing_values_with_tpot(data):
    # Identify columns with missing values
    columns_with_missing_values = data.columns[data.isnull().any()].tolist()
    #print("\nColumns with missing values before imputation:", columns_with_missing_values)

    # Loop over columns with missing values and apply predictive imputation
    for column in columns_with_missing_values:
        data = predictive_imputation(data, column)
    return data

```

Appendix 13 - Data_cleaner1

```python

def detect_time_date_columns(data):
    time_date_cols = []
    data2norm = data.copy()
    date_format_YYYY = {
        "%Y-%m-%d": r'\d{4}\D\d{1,2}\D\d{1,2}'  # YYYY-M-D or YYYY-MM-DD
    }
    date_formats = {
        "%d/%m/%Y": r'\d{1,2}[\/\- ]\d{1,2}[\/\- ]\d{4}',  # D/M/YYYY or DD/MM/YYYY
        "%m/%d/%Y": r'\d{1,2}[\/\- ]\d{1,2}[\/\- ]\d{4}'   # M/D/YYYY or MM/DD/YYYY
    }
    time_pattern = r'\d{2}:\d{2}(:\d{2})?(\sAM|\sPM)?'


    def infer_date_format(date_str):
        for format, regex in date_formats.items():
            if re.match(regex, date_str):
                return format
        return None  # Unknown format

    def parse_time(time_str):
        for fmt in ('%H:%M', '%I:%M:%S %p', '%H:%M:%S'):
            try:
                # Parse the time string using the given format
                parsed_time = pd.to_datetime(time_str, format=fmt)
                # Return only the hour part as a string
                return str(parsed_time.hour).zfill(2)  # zfill(2) ensures a two-digit format, like '02', '11'
            except ValueError:
                # If parsing fails, continue to the next format
                continue
        return None

    def parse_time2norm(time_str):
        for fmt in ('%H:%M', '%I:%M:%S %p', '%H:%M:%S'):
            try:
                # Return only the time part
                return pd.to_datetime(time_str, format=fmt).time()
            except ValueError:
                continue
        return None # or raise an exception


    for col in data.columns:
        sample_values = data[col].dropna().astype(str).sample(min(100, len(data[col])))
        # Flag to track if column has been processed as date or time
        processed_as_date_or_time = False

        # Check if any sample value matches the YYYY date format
        if any(re.search(pattern, value) for pattern in date_format_YYYY.values() for value in sample_values):
            # Process the column as a date
            data[f'{col}_date'] = pd.to_datetime(data[col], format='%Y-%m-%d', errors='coerce')
            data2norm[f'{col}_date'] = pd.to_datetime(data[col], format='%Y-%m-%d', errors='coerce').dt.strftime('%Y-%m-%d')
            # Mark the column as processed
            processed_as_date_or_time = True
            time_date_cols.append(f'{col}_date')

        # Check if any sample value matches any of the other date formats
        elif any(re.search(pattern, value) for pattern in date_formats.values() for value in sample_values):
            # Further processing for other date formats
            sample_dates = [value for value in sample_values if any(re.search(date_formats[fmt], value) for fmt in date_formats)]
            if sample_dates:
                # Infer the date format based on the first sample date
                inferred_format = infer_date_format(sample_dates[0])
                if inferred_format:
                    # Process the column according to the inferred format
                    data[f'{col}_date'] = pd.to_datetime(data[col], format=inferred_format, errors='coerce')
                    data2norm[f'{col}_date'] = pd.to_datetime(data[col], format=inferred_format, errors='coerce').dt.strftime('%Y-%m-%d')
                    # Mark the column as processed
                    processed_as_date_or_time = True
                    time_date_cols.append(f'{col}_date')


        if any(re.search(time_pattern, value) for value in sample_values):
            # Process as time
            data[f'{col}_time'] = data[col].apply(parse_time)
            data2norm[f'{col}_time'] = data[col].apply(parse_time2norm)
            # Mark column as processed
            processed_as_date_or_time = True
            #time_date_cols.append(f'{col}_time')

        # If the column was processed as date or time, drop the original column
        if processed_as_date_or_time:
            data.drop(col, axis=1, inplace=True)
            data2norm.drop(col, axis=1, inplace=True)


    # Assuming the function should return the modified DataFrame and the list of new time/date related columns
    return time_date_cols, data, data2norm



import pandas as pd

def drop_id_columns(data):
    # Define keywords for identifying ID columns
    id_keywords = ['_id', 'id_', ' id', 'id ']

    # Initialize list to store ID columns
    id_columns = []

    # Iterate over the columns of the DataFrame
    for col in data.columns:
        # Check for full-word match in column name
        if any(re.search(r'\b{}\b'.format(keyword), col.lower()) for keyword in id_keywords):
            # If full-word match is found, add the column to the list of ID columns
            id_columns.append(col)

        # Check if the data in the column increments by 1 row by row
        elif pd.api.types.is_numeric_dtype(data[col]):
            is_id_column = True
            prev_value = None
            for value in data[col]:
                if prev_value is not None and value != prev_value + 1:
                    is_id_column = False
                    break
                prev_value = value

            if is_id_column:
                id_columns.append(col)

        #if len(data[col]) == data[col].nunique():
            #id_columns.append(col)

    # Print information about the columns and ID columns
    #print(f"Columns: {data.columns}")
    #print(f"Dropping ID columns: {id_columns}")

    # Drop the ID columns from the DataFrame, ignoring errors
    data_no_id = data.drop(columns=id_columns, errors='ignore')

    # Return the modified DataFrame
    return data_no_id

```

Appendix 14 - Data_cleaner2



```python

def predictive_imputation(data, column_to_impute):
    # Identify and drop columns where all values (except for the header) are NaN
    non_na_counts = data.count()
    empty_columns = non_na_counts[non_na_counts == 0].index.tolist()
    data = data.drop(columns=empty_columns)

    # Check if the target column for imputation was dropped or is empty
    if column_to_impute not in data.columns:
        #print(f"Column '{column_to_impute}' was empty and has been dropped.")
        return data

    missing_before = data[column_to_impute].isnull().sum()
    if missing_before == 0:
        #print(f"No missing values in '{column_to_impute}'. No imputation needed.")
        return data

    if pd.api.types.is_numeric_dtype(data[column_to_impute]):
        tpot_model = TPOTRegressor(max_time_mins=30, generations=3, population_size=50, verbosity=2, random_state=42)
        #print(f"Using TPOTRegressor for numeric column: {column_to_impute}")
    else:
        tpot_model = TPOTClassifier(max_time_mins=30, generations=5, population_size=50, verbosity=2, random_state=42)
        #print(f"Using TPOTClassifier for categorical column: {column_to_impute}")

    # Splitting the data into training and testing sets based on null values in the column to impute
    train_data = data.dropna(subset=[column_to_impute])
    test_data = data[data[column_to_impute].isnull()]
    X_train = train_data.drop(columns=[column_to_impute])
    y_train = train_data[column_to_impute]
    X_test = test_data.drop(columns=[column_to_impute])

    # Fitting the model and predicting the missing values
    tpot_model.fit(X_train, y_train)
    predicted_values = tpot_model.predict(X_test)
    data.loc[data[column_to_impute].isnull(), column_to_impute] = predicted_values
    missing_after = data[column_to_impute].isnull().sum()

    #print(f"Imputed missing values in '{column_to_impute}'. Missing before: {missing_before}, Missing after: {missing_after}")
    return data


def normalize_data(normalized_data):
    scaler = StandardScaler()
    numerical_cols = normalized_data.select_dtypes(include=['float64', 'int64']).columns
    normalized_data[numerical_cols] = scaler.fit_transform(normalized_data[numerical_cols])
    #print("Normalized data:\n", normalized_data.head())

    return normalized_data

```

Appendix 15 - Data_cleaner3