Solent University
Faculty of Business, Law and Digital Technologies

PhoneDB Software Report

Author : **Deborah Adetayo ADEDIGBA**
Solent ID : **102026038**
Course Title : **Programming For Problem Solving (COM728)**
Module Leader : **Jarutas Andritsch**
Date : **January 7, 2024.**

Table of Contents

Table of Figures

List of Tables

1. Overview

In an era dominated by technological advancements, the intricacies of mobile devices play a pivotal role in shaping our digital landscape. The PhoneDB website is a collection of information offering insight into the data on various devices.

This project aims to retrieve data from a CSV file using the CSV module, analyse data using the pandas module, and visualise the data using the matplotlib module. This project will help provide an intricate perception of the mobile technology PhoneDB data.

The dataset is in the form of a CSV file that contains 48 columns and 1,271 rows. Each row in the file represents a single record for a particular device. The data file is stored in the same directory as the other software files.

Requirement	Status
Load the data from a CSV file using the CSV reader	completed
Retrieve the model's name, manufacturer, weight, price, and price unit for the device(s) based on the OEM_id	completed
Retrieve the band, model name, RAM capacity, market regions, and the date when the information was added for the device(s) associated with a specified code name.	completed
Retrieve the oem_id, release date, announcement date, dimensions, and device category of the device(s) based on a specified RAM capacity	completed
Retrieve information from the user's chosen columns and apply a specific condition related to an individual device.	completed
Load data from a CSV file using the pandas module	completed
Identify the top 5 regions where a specific band of devices was sold.	completed
Analyse the average price of devices within a specific band, all in the same currency.	completed
Analyse the average mass for each manufacturer and display the list of average mass for all manufacturers.	completed
Analyse the data to derive meaningful insights based on the user's unique selection, distinct from the previous requirements.	completed
Load data from a CSV file using the pandas module	completed
Create a chart visually representing the proportion of RAM types for devices in the current market.	completed
Create a chart to visually compare the number of devices for each USB connector type	completed
Create charts illustrating the monthly average price trends (in GBP) for devices released yearly from 2020 to 2023. Each chart should focus on a specific year.	completed
Create a visualisation of the user's selection to showcase information related to device features	completed

Table 1: Requirement Completion

2. Project Implementation

This project comprises five modules and a notebook, which will be elaborated in the following sections.

2.1. Project Structure

This project comprises five modules and one juypter notebook, the main.ipynb. The modules are tui.py, Loading_dataset.py, Data_retriever.py, Data_analysis.py and Data_visualization.py. The modules are for functions for text-user interface, loading the data with different formats, retrieving data, analysing the data using pandas, and for visualisation.

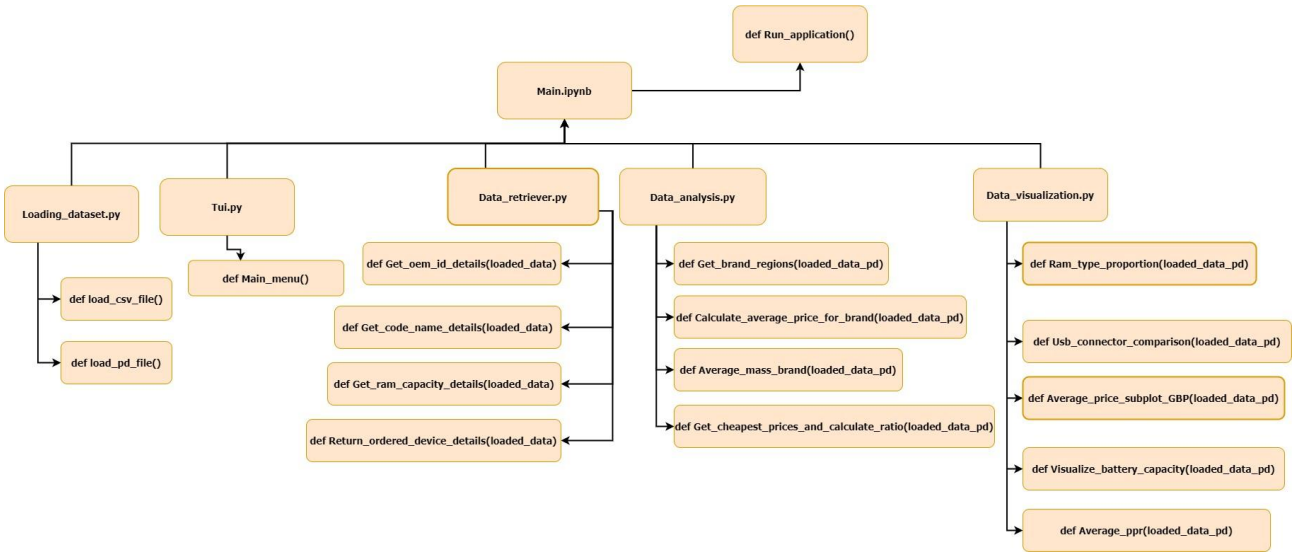

Fig 1: Flowchart of the project structure

2.2. Modules

These sections contain the self-built modules used by implementing and importing all the modules in the notebook(main.ipynb).

2.2.1. Loading_dataset

This module takes a variable file_location to store the user input of the location of the data, and it has two functions for loading the data using a CSV and Pandas method.

2.2.1.1. Load_csv_file

This function uses a global variable called "file_location" and a while loop to request a valid file. It reads a CSV file using the CSV module, appends the data to a variable called "phone_data", and prints a success message. It also handles potential errors such as invalid CSV format, file not found, and IO errors with corresponding error messages, as in Fig 2.

```
# reading in the csv data from csv file using csv module
import csv
file_location = None  # Initialize the location variable

def load_csv_file():
    global file_location  # Use the global variable
    while True:
        file_location = input("Enter the file location: ")
        try:
            phone_data = []
            with open(file_location, 'r', encoding='utf-8') as file:
                reader = csv.reader(file)
                next(reader)
                for line in reader:
                    phone_data.append(line)
            print("\n\033[1mFetching data...\033[0m")
            print(f"Successfully loaded the \033[1m{file_location}\033[0m dataset.")
            return phone_data

        except csv.Error:
            print(f"\033[1m\033[31mInvalid CSV file:\033[0m {file_location}")
        except FileNotFoundError:
            print(f"\033[1m\033[31mFile not found:\033[0m {file_location}")
        except IOError:
            print(f"\033[1m\033[31mCouldn't read\033[0m {file_location}.")

        retry = input("Do you want to enter a new file location? (yes/no): ")
        if retry.lower() != 'yes':
            print("\033[1mExiting file loading...\033[0m\n")
            return None  # Returning None to indicate an unsuccessful attempt
```

Fig 2: Loading the CSV file

2.2.1.2. Load_pd_file

The function first checks if file_location is not None, indicating that the user has entered a valid file location. Inside the try-except block, it attempts to read the CSV file using the Pandas method to a DataFrame and return the Phone_data dataframe in Fig 3.

```
import pandas as pd
# Reading in the csv data as a pandas file using pandas
def load_pd_file():
    global file_location  # Use the global variable
    if file_location is not None:
        try:
            phone_data = pd.read_csv(file_location, encoding='utf-8')
            print("\n\033[1mFetching data...\033[0m")
            print(f"Successfully loaded the \033[1m{file_location}\033[0m pandas dataset.")
            return phone_data
        except pd.errors.EmptyDataError:
            print(f"\033[1m\033[31mEmpty CSV file:\033[0m {file_location}")
        except FileNotFoundError:
            print(f"\033[1m\033[31mFile not found:\033[0m {file_location}")
        except pd.errors.ParserError:
            print(f"\033[1m\033[31mCouldn't read\033[0m {file_location}. It may not be a valid CSV file.")
        except Exception as e:
            print(f"\033[1m\033[31mAn error occurred while loading the file:\033[0m {str(e)}")
```

Fig 3: Loading the CSV file using pandas

2.2.2. Tui

This module contains the functions used to build the user menu and give the user a good interface and user-friendly experience.

2.2.2.1. Main_menu

The PhoneDB application has a main menu that allows users to choose between sections A, B, and C for loading, retrieving, analysing, and visualising artefact data. It includes a user-friendly interface, handles user inputs, and has options to continue or exit the program in each section, as seen in Fig 4.

```
# the tui for the program interacting with the modules
def main_menu():
    # user text displays
    print("\033[1m\033[34mWelcome to PhoneDB!\033[0m")
    print("\033[1m\033[4mBrief Background\033[0m")
    print("\033[32mPhoneDB website is your go-to information hub for smartphones, tablets, PDAs, and mobile devices.\033[0m")
    print("\033[32mOur platform offers a comprehensive collection of data and various services to help you find the most suitable mobile
device.\033[0m")
    print()
    print("\033[1m\033[31mThank you for choosing PhoneDB! Let's explore the world of mobile devices together.\033[0m")

    username = input("What is your name? ")
    username = username.upper()
    print(f"\nYes, {username}, let's get to work.")
    # loading the neccessary modules
    import Loading_dataset
    import Data_retriever
    import Data_analysis
    import Data_visualization
```

Fig 4: main_menu function of Tui.py

This function also includes loops and if-else statements to navigate the menus better. It also provides all the necessary data required for each section, as shown in Fig 5.

```
                    # Ask the user if they want to continue in section B
                    continue_choice_B = input("Do you want to perform another operation in section B? (yes/no): ").lower()
                    # loop to keep asking till a valid entry is entered
                    while continue_choice_B not in ['yes', 'no']:
                        print("Invalid choice. Please enter 'yes' or 'no'.")
                        continue_choice_B = input("Do you want to perform another operation in section B? (yes/no): ").lower()

                    # option to return to main menu
                    if continue_choice_B == 'no':
                        print("Returning to the main menu...")
                        break

            elif choice1 == "C":
                while True:
                    try:
                        if loaded_data_pd is None:
                            print("Please load the dataset first in Section B.")
                            break

                        print("\nPlease select an operation:")
                        print("9. Get the proportion of the RAM types for devices in the current market using a pie chart")
                        print("10. Get the number of devices for each USB connector type using a bar chart")
                        print("11. Get the monthly average price trends (in GBP) for devices released in each year from 2020 to 2023.")
                        print("12. Get a bar chart for the average battery capacity and the relationship between display size and battery
capacity with a scatter plot")
                        print("13. Get the average price performance ratio bar chart")

                        choice_C = int(input("Enter your choice (9-13): "))
```

Fig 5: An if-else statement for navigation of main_menu

2.2.3. Data_retriever

This performs various forms of retriever of details from the given dataset using some functions.

2.2.3.1. Get_oem_id_details(loaded_data)

This function searches for details of a specified OEM ID in the loaded data. It stores the output in a dictionary and prints it if found, as in Fig 6. The function handles errors and provides clarifying messages.

```
# function to return the specific details based on OEM_id of a device
def get_oem_id_details(loaded_data):
    while True:
        try:
            oem_id = input("Enter the OEM ID: ").lower()  # Convert input to lowercase

            # creating a dictionary to store the details
            oem_details = {}

            # Loop through each row in the CSV file
            for row in loaded_data:
                # Check if the row has enough elements and compare in lowercase
                if row and len(row) > 0 and row[0].lower() == oem_id:
                    # If the OEM ID is found, store the details in the dictionary
                    oem_details["OEM ID"] = oem_id
                    oem_details["Model"] = row[2] if len(row) > 2 else ""
                    oem_details["Manufacturer"] = row[6] if len(row) > 6 else ""
                    oem_details["Weight"] = row[14] if len(row) > 14 else ""
                    oem_details["Price"] = row[15] if len(row) > 15 else ""
                    oem_details["Price Unit"] = row[16] if len(row) > 16 else ""

                    # Print the OEM ID before "OEM ID Details"
                    print("\033[44m\033[1m" + f"{oem_id} OEM ID Details:" + "\033[0m")
                    print("\033[1m" + "-" * 60 + "\033[0m")
                    print("\033[1mModel:\033[0m", oem_details["Model"])
                    print("\033[1mManufacturer:\033[0m", oem_details["Manufacturer"])
                    print("\033[1mWeight:\033[0m", oem_details["Weight"])
                    print("\033[1mPrice:\033[0m", oem_details["Price"], oem_details["Price Unit"])
                    print("\033[1m" + "-" * 60 + "\033[0m")
```

Fig 6: get_oem_id_details function getting information using oem_id

When the function does not find the OEM_id entered by the user, it gives a print statement and loops again to ask if they want to use a new entry, as shown below in Fig 7.

```
        # If the loop completes without finding a match
        print("\033[1m" + "No match for the OEM ID details found." + "\033[0m")
        retry = input("Do you want to enter a new OEM ID? (yes/no): ")
        if retry.lower() != 'yes':
            print("\033[1m" + "Exiting OEM ID details retrieval..." + "\033[0m\n")
            break  # Exit the loop to stop asking for a new OEM ID

    except ValueError:
        print("\033[41m\033[1m" + "Invalid input. Please enter a valid OEM ID." + "\033[0m")
    except IndexError:
        print("\033[41m\033[1m" + "Index error. Make sure your data has enough columns." + "\033[0m")
    except Exception as e:
        print("\033[41m\033[1m" + f"An error occurred: {e}" + "\033[0m")
```

Fig 7: Error handling when oem_id is not found

2.2.3.2. Get_code_name_details(loaded_data)

This function prompts the user for a code name, converts it to lowercase, searches for matching rows in the dataset, and displays the relevant details. It also includes error handling and a user-friendly interface for output display, as in Fig 8.

```python
# function to get code_name from user and return specific details about the code names
def get_code_name_details(loaded_data):
    while True:
        try:
            result_rows = []
            code_name = input("Enter the code name: ").lower()  # Convert input to lowercase

            # Create a flag to check if the code name is found
            code_name_found = False

            # Loop through each row in the CSV data
            for row in loaded_data:
                if row and row[7].lower() == code_name:  # Compare in lowercase
                    # If the code name is found, create a dictionary with details
                    value1 = row[34]
                    value2 = row[2]
                    value3 = row[23]
                    value4 = row[44]
                    value5 = row[45]
                    row_dict = {
                        "Band": value1,
                        "Model name": value2,
                        "RAM": value3,
                        "Market regions": value4,
                        "Date of addition": value5
                    }
                    result_rows.append(row_dict)

                    code_name_found = True
```

Fig 8: Function returning phone details based on the code name

2.2.3.3. Get_ram_capacity_details(loaded_data)

The function takes RAM size input, fetches matching rows from the dataset, creates a dictionary with relevant details, and appends it to the result list. It shows the rows in groups of 20 and allows the user to see more by entering 'next' as seen in Fig 9. Error handling ensures seamless user interaction.

```python
        if not ram_size_found:
            print("\033[1mNo match for the RAM size details found.\033[0m")
        else:
            # displaying 20 rows at time if available
            num_rows = 20
            start_pos = 0
            end_pos = min(start_pos + num_rows, len(result_rows))

            while True:
                for i in range(start_pos, end_pos):
                    row = result_rows[i]
                    print()
                    print("\033[1m" + "-" * 100 + "\033[0m")
                    print("\033[1mOEM Id:\033[0m", row["OEM Id"])
                    print("\033[1mRelease date:\033[0m", row["Release date"])
                    print("\033[1mAnnouncement date:\033[0m", row["Announcement date"])
                    print("\033[1mDimensions:\033[0m", row["Dimensions"])
                    print("\033[1mDevice category:\033[0m", row["Device category"])

                if end_pos >= len(result_rows):
                    print("\033[1m" + "-" * 100 + "\033[0m")
                    print("\033[1mNo more rows available.\033[0m")
                    break

                user_input = input("Enter 'next' to retrieve the next 20 rows, or 'quit' to exit: ").lower()

                while user_input not in ['next', 'quit']:
                    print("\033[1mInvalid input.\033[0m Please enter 'next' to retrieve more rows or 'quit' to exit.")
                    user_input = input("Enter 'next' to retrieve the next 20 rows, or 'quit' to exit: ").lower()
```

Fig 9: Code showing a user-friendly display of output

2.2.3.4. Return_ordered_device_details(loaded_data)

The function prompts the user for the brand name, device type, and price range. It then filters the loaded data based on these criteria, extracts relevant device details, and sorts the results by release date, as seen in Fig 10. The function is designed to interactively retrieve user preferences for selecting electronic devices based on specific criteria (brand, device type,

price range) from the dataset, helping them streamline their search and decide based on their preference.

```
# extracts specific device details, and sorts the results by release date
def return_ordered_device_details(loaded_data):
    try:
        result_rows = []

        while True:
            # List of available brands
            brands = ["Samsung", "Xiaomi", "Sharp", "Sony", "Lenovo", "Asus", "BBK", "T-Mobile", "ZTE", "Nokia", "Microsoft", "LG",
"Apple", "Motorola"]
            brand_string = ", ".join(brands)
            print(f"We have the following brand of phones:\n{brand_string}")

            brand_name = input("Enter the brand name (or 'quit' to exit): ").strip().lower()

            if brand_name == 'quit':
                break

            # List of available device types
            device_types = ["Tablet", "Smartwatch", "Smartphone"]

            print("We have the following device types:")
            for device_type in device_types:
                print(device_type)

            device_type = input("Enter the device type: ").strip().lower()

            # Input validation for price range
            while True:
                try:
                    min_price = float(input("Enter the minimum price for your phone: "))
                    max_price = float(input("Enter the maximum price for your phone: "))
                    break
                except ValueError:
                    print("Please enter valid numeric values for price.")
```

Fig 10: Gets ordered list of device details to aid user's decision-making

The function incorporates input validation, a date parsing function to sort the dates in Fig 11.

```
    # Function to manually parse the date
    def parse_date(date_string):
        day, month, year = map(int, date_string.split('-'))
        return (year, month, day)

    # Loop through your data to find matching devices
    for row in loaded_data:
        if len(row) > 43 and row[1].strip().lower() == brand_name and row[9].strip().lower() == device_type:
            price = float(row[15]) if row[15] else 0.0
            if min_price <= price <= max_price:
                value1 = row[43]
                value2 = row[24]
                value3 = row[19]
                value4 = row[29]
                value5 = row[35]
                value6 = parse_date(row[3])
                value7 = row[15]
                row_dict = {
                    "Battery Capacity": value1,
                    "Storage Space": value2,
                    "Additional Software Details": value3,
                    "Pixel Density": value4,
                    "Sim-Card-Type": value5,
                    "Released Date": value6,
                    "Price": value7
                }
                result_rows.append(row_dict)

    # Sort the result_rows by the 'Released Date'
    result_rows = sorted(result_rows, key=lambda x: x["Released Date"])
```

Fig 11: Function parsing the release date and displaying required output to the user

2.2.4. Data_analysis

When the user enters the analysis section without entering the location of the CSV file, it ensures that the location has been given in the earlier section. If not, it redirects the user to the **Load_csv_file** function that takes in the file location from the user. This functionality is included under the elif statement for section B in the **Tui.py** module, as in Fig 12.

```
elif choice1 == "B":
    if loaded_data is None:
        print("Please load the dataset first in Section A.")
    else:
        while True:
            print("\nPlease select an operation:")
            print("5. Load your data from a CSV file into a pandas DataFrame")
            print("6. Get the top 5 regions a brand is sold")
            print("7. Get the average price of devices for a brand in a particular or all currencies")
            print("8. Get the average mass of the device for each manufacturer")
            print("9. Get the recommendation from the top 5 cheap devices with a price performance ratio")
            choice_B = int(input("Enter your choice (5-9): "))
```

Fig 12: If-else statement to ensure the availability of data before analysis with pandas

2.2.4.1. Get_brand_regions(loaded_data_pd)

According to the dataset, the function enables users to input a device brand and supply insights into the top regions where the specified brand is famous. The function cross-checks user input against available brand names, groups the data, and displays the top regions for the selected brand, as in Fig 13.

```
def get_brand_regions(loaded_data_pd):
    while True:
        try:
            brand_names = ", ".join(loaded_data_pd['brand'].unique())
            print(f"Available device brand names in this dataset:\n{brand_names}")

            user_brand = input("Enter the brand you want to see (or 'quit' to exit): ").strip().lower()

            if user_brand == 'quit':
                break  # Exit the loop if the user wants to quit

            # Validate user input against available brand names
            if user_brand not in loaded_data_pd['brand'].str.lower().unique():
                raise ValueError("Invalid brand name. Please enter a valid brand name.")

            # Group data to find the top regions for each brand
            brand_counts = loaded_data_pd.groupby(['market_regions', 'brand']).size().reset_index(name='counts')
            top_regions_by_brand = brand_counts.groupby('brand').apply(lambda x: x.nlargest(5, 'counts'))
            top_regions_by_brand = top_regions_by_brand.drop(columns='brand').reset_index()

            user_brand_data = top_regions_by_brand[top_regions_by_brand['brand'].str.lower() == user_brand]

            if not user_brand_data.empty:
                print(f"Top regions for brand: {user_brand}")
                for _, row in user_brand_data.iterrows():
                    print(f"Region: {row['market_regions']} - Count: {row['counts']}")
            else:
                print(f"No data available for brand: {user_brand}")

        except ValueError as ve:
            print(f"\033[31mValueError: {ve}. Please try again.\033[0m")
        except Exception as e:
            print(f"\033[31mAn error occurred: {e}. Please try again.\033[0m")
```

Fig 13: Top brand regions using count for a particular brand

2.2.4.2.Calculate_average_price_for_brand(loaded_data_pd)

This function takes two inputs - device brand and currency - and returns the average price for that brand in the given currency. It checks the input, filters the data, and returns the average price for the specified brand and currency or all brands and currencies in Fig 14.

```
def calculate_average_price_for_brand(loaded_data_pd):
    while True:
        try:
            # Get user input for the brand
            brand_name = ", ".join(loaded_data_pd['brand'].unique())
            print(f"The device brand names in this dataset are:\n {brand_name}")

            brand_input = input("Enter the brand (or 'all' for all brands): ").strip().lower()
            valid_brand_options = loaded_data_pd['brand'].str.lower().unique()

            while brand_input != 'all' and brand_input not in valid_brand_options:
                print(f"Invalid input. Please choose from the following options: {', '.join(valid_brand_options)}")
                brand_input = input("Enter the brand (or 'all' for all brands): ").strip().lower()

            # Get user input for the currency
            currency = ", ".join(loaded_data_pd['price_currency'].unique())
            print(f"These are the currencies you can choose from: \n {currency}")

            currency_input = input("Enter the currency (or 'all' for all currencies): ").strip().lower()
            valid_currency_options = loaded_data_pd['price_currency'].str.lower().unique()

            while currency_input != 'all' and currency_input not in valid_currency_options:
                print(f"Invalid input. Please choose from the following options: {', '.join(valid_currency_options)}")
                currency_input = input("Enter the currency (or 'all' for all currencies): ").strip().lower()
```

Fig 14: The average price by brand

This function also allows the user to check for other brands or currencies, as seen in Fig 15.

```
    if not brand_avg_prices.empty:
        num_rows = 20
        start_pos = 0
        end_pos = min(start_pos + num_rows, len(brand_avg_prices))

        while True:
            # Display the current set of rows
            print(brand_avg_prices.iloc[start_pos:end_pos])

            if end_pos >= len(brand_avg_prices):
                print("\033[1mNo more rows available.\033[0m")
                break

            user_input = input("Enter 'next' to retrieve the next 20 rows, or 'quit' to exit: ").strip().lower()
            if user_input == 'next':
                start_pos += num_rows
                end_pos = min(start_pos + num_rows, len(brand_avg_prices))
            elif user_input == 'quit':
                break

        retry = input("Would you like to calculate for another brand/currency (yes/no)? ").strip().lower()
        if retry != 'yes':
            break
    else:
        print("\nNo data available for average price by brand and currency.")
```

Fig 15: Code showing additional check on another brand

2.2.4.3. Average_mass_brand(loaded_data_pd)

This function uses group-by to group the data by manufacturer and calculates each manufacturer's mean weight in grams. The function uses a reusable **get_user_input** helper function for user input validation in Fig 16, ensuring a smoother interaction and display of the function as in Fig 17.

```
def get_user_input(prompt, valid_options):
    while True:
        user_input = input(prompt).strip().lower()
        if user_input in valid_options:
            return user_input
        print(f"Invalid input. Please choose from the following options: {', '.join(valid_options)}")
```

Fig 16: User input validation

```
def average_mass_brand(loaded_data_pd):
    try:
        # Group the data by 'manufacturer' and calculate the average 'weight_gram'
        manufact_avg_mass = loaded_data_pd.groupby('manufacturer')['weight_gram'].mean().round(2).reset_index()

        if not manufact_avg_mass.empty:
            num_rows = 20
            start_pos = 0
            end_pos = min(start_pos + num_rows, len(manufact_avg_mass))

            while True:
                # Display the current set of rows without external modules
                print("\033[1mAverage Mass by Brand\033[0m")
                print(f"\033[4m\033[1m{'Manufacturer':<30}{'Average Weight (grams)':<20}\033[0m")

                for _, row in manufact_avg_mass.iloc[start_pos:end_pos].iterrows():
                    print(f"{row['manufacturer']:<40}{row['weight_gram']:<25}")

                if end_pos >= len(manufact_avg_mass):
                    print("\033[1mNo more rows available.\033[0m")
                    break

                user_input = get_user_input("\nEnter 'next' to view the next 20 rows or 'quit' to exit: ", ['next', 'quit'])

                if user_input == 'next':
                    start_pos += num_rows
                    end_pos = min(start_pos + num_rows, len(manufact_avg_mass))
                elif user_input == 'quit':
                    break
        else:
            print("\nNo data available for average mass by brand.")

    except Exception as e:
        print(f"\n\033[31mAn error occurred: {e}. Please try again.\033[0m")
        # Return None in case of an error
        return None
```

Fig 17: Brand based on the average mass of devices

2.2.4.4. Get_cheapest_prices_and_calculate_ratio(loaded_data_pd)

The function allows users to input a brand and returns the top 5 cheapest devices for that brand in Fig 18. Users choose a specific device (via its OEM ID) to calculate and display its price-performance ratio using the second function, **calculate_price_performance_ratio**, which is responsible for calculating the price-performance ratio based on predefined weightage factors for CPU, RAM, storage, and price.

```
def get_cheapest_prices_and_calculate_ratio(loaded_data_pd):
    while True:
        try:
            brand_names = ", ".join(loaded_data_pd['brand'].unique())
            print(f"Available device brand names in this dataset:\n{brand_names}")

            user_brand = input("Enter the brand to find the top 5 cheapest prices (or 'quit' to exit): ").strip().lower()

            if user_brand == 'quit':
                return  # Exit the function if the user wants to quit

            filtered_data = loaded_data_pd[loaded_data_pd['brand'].str.lower() == user_brand]

            if not filtered_data.empty:
                top_5_cheapest = filtered_data.nsmallest(5, 'price')[['oem_id', 'price']]
                print(f"Top 5 cheapest prices for brand: {user_brand}")
                for _, row in top_5_cheapest.iterrows():
                    print(f"OEM ID: {row['oem_id']} - Price: {row['price']}")

                user_id_input = input("Enter the OEM ID of the device to calculate the price-performance ratio (or 'quit' to return to
brand selection): ").strip()

                if user_id_input == 'quit':
                    return  # Return to brand selection

                # Validate the entered OEM ID
                if user_id_input not in loaded_data_pd['oem_id'].values:
                    print(f"Invalid OEM ID: {user_id_input}. Please enter a valid OEM ID.")
                else:
                    calculate_price_performance_ratio(loaded_data_pd, user_id_input)
                    break  # Exit the loop after a successful calculation
            else:
                print(f"No data available for brand: {user_brand}")
        except Exception as e:
            print(f"\033[31mAn error occurred: {e}. Please try again.\033[0m")
```

Fig 18: Top 5 cheapest brands

This function also includes a normalised ratio and recommendations based on the calculated values. This set of functions enhances the ability of the users to make informed decisions when selecting devices based on their price and performance. The purpose of the function is to help users identify the affordable options that benefit them based on the device's performance, as seen in Fig 19.

```
def calculate_price_performance_ratio(loaded_data_pd, user_id_input):
    df = loaded_data_pd

    # Define weightage factors
    weight_cpu = 0.4
    weight_ram = 0.3
    weight_storage = 0.2
    weight_price = 0.1

    # Calculate a performance score based on specifications
    df['Performance_Score'] = (weight_cpu * df['cpu_clock']) + \
                              (weight_ram * df['ram_capacity']) + \
                              (weight_storage * df['non_volatile_memory_capacity'])

    # Calculate price-performance ratio
    df['Price_Performance_Ratio'] = df['Performance_Score'] / df['price']

    # Normalize the price-performance ratios
    df['Normalized_Price_Performance'] = (df['Price_Performance_Ratio'] - df['Price_Performance_Ratio'].min()) / \
                                         (df['Price_Performance_Ratio'].max() - df['Price_Performance_Ratio'].min())

    # Sort the devices by their normalized price-performance ratios
    df = df.sort_values(by='Normalized_Price_Performance', ascending=False)

    # Find and display information about the specified device
    selected_device = df[df['oem_id'] == user_id_input]
    if not selected_device.empty:
        brand = selected_device['brand'].values[0]
        price = selected_device['price'].values[0]
        performance_score = selected_device['Performance_Score'].values[0]
        price_performance_ratio = selected_device['Price_Performance_Ratio'].values[0]
```

Fig 19: Calculate the price-performance ratio for a user to make a decision

2.2.5. Data_visualization

This module has functions that help create charts to display essential data.

2.2.5.1. Ram_type_proportion(loaded_data_pd)

This function generates a pie chart showing the dataset's RAM type distribution for devices. It counts the occurrences of each RAM type and displays the proportions as percentages on the chart. The chart includes RAM type, percentage, and title labels.

2.2.5.2. Usb_connector_comparison(loaded_data_pd)

This function generates a bar chart comparing the counts of different USB connector types for devices in the dataset. It uses value_counts to calculate the number of occurrences of each USB connector type and displays the chart with the count of each bar for easy readability, as in Fig 20.

```python
# Create a chart to visually compare the number of devices for each USB connector type
def usb_connector_comparison(loaded_data_pd):
    usb_connector_counts = loaded_data_pd['usb_connector'].value_counts()
    # Create the bar chart
    plt.figure(figsize=(8, 6))
    bars = plt.bar(usb_connector_counts.index, usb_connector_counts.values)

    # Set labels and title
    plt.xlabel('USB Connector')
    plt.ylabel('Count')
    plt.title('Barchart showing the Comparison of USB Connector Types for Devices')

    # Display the counts above the bars
    for index, value in enumerate(usb_connector_counts):
        plt.text(index, value, str(value), ha='center', va='bottom')


    plt.show()
```

Fig 20: Display bar chart for USB connectors

2.2.5.3. Average_price_subplot_GBP(loaded_data_pd)

This function generates subplots of line graphs showing the average prices of devices for the years 2020 to 2023 in GBP (British Pounds). The data is grouped by year and month, and then the average monthly price is calculated in Figure 21 and plotted for each year within the subplot.

```python
# create a subplot to visualize the line graphs for the price for year 2020 to 2023
def subplot_GBP(loaded_data_pd):
    # Assuming 'released_date' is in the format 'dd-mm-yyyy'
    loaded_data_pd['released_date'] = pd.to_datetime(loaded_data_pd['released_date'], format='%d-%m-%y')

    # Extract the years and create a new column
    loaded_data_pd['released_year'] = loaded_data_pd['released_date'].dt.year
    loaded_data_pd['released_month'] = loaded_data_pd['released_date'].dt.month

    # Split the data by year into separate DataFrames
    yearly_data = {}
    for year, data in loaded_data_pd.groupby('released_year'):
        yearly_data[year] = data

    GBP_2020 = yearly_data[2020]
    GBP_2021 = yearly_data[2021]
    GBP_2022 = yearly_data[2022]
    GBP_2023 = yearly_data[2023]
    # Group by year and month and calculate the average price for each month
    monthly_average_prices_2020 = GBP_2020.groupby(['released_year', 'released_month'])['price'].mean().reset_index()
    monthly_average_prices_2021 = GBP_2021.groupby(['released_year', 'released_month'])['price'].mean().reset_index()
    monthly_average_prices_2022 = GBP_2022.groupby(['released_year', 'released_month'])['price'].mean().reset_index()
    monthly_average_prices_2023 = GBP_2023.groupby(['released_year', 'released_month'])['price'].mean().reset_index()

    monthly_average_prices_2020.rename(columns={'price': 'average_price(GBP)'}, inplace=True)
    monthly_average_prices_2021.rename(columns={'price': 'average_price(GBP)'}, inplace=True)
    monthly_average_prices_2022.rename(columns={'price': 'average_price(GBP)'}, inplace=True)
    monthly_average_prices_2023.rename(columns={'price': 'average_price(GBP)'}, inplace=True)
```

Fig 21: Grouping and splitting of data monthly for the four years

2.2.5.4. Visualize_battery_capacity(loaded_data_pd)

This function in Fig 22 analyses device patterns based on brand, display size, and battery capacity. It groups numeric battery capacities by brand and calculates the average for each brand. This helps users determine the best brand based on battery capacity and clearly compares brands.

```
def vis_brand(loaded_data_pd):
    # Extract the numeric battery capacities from the format "5000 mAh battery"
    loaded_data_pd['battery_capacity'] = loaded_data_pd['battery_capacity'].str.extract('(\d+)').astype(float)


    # Group data by brand and calculate average battery capacity and display size
    brand_stats = loaded_data_pd.groupby('brand')[['battery_capacity', 'display_diagonal']].mean()
    # Create a bar chart to compare average battery capacity by brand
    plt.figure(figsize=(12, 6))
    plt.bar(brand_stats.index, brand_stats['battery_capacity'], color='blue')
    plt.xlabel('Brand')
    plt.ylabel('Average Battery Capacity (mAh)')
    plt.title('Average Battery Capacity by Brand')
    plt.xticks(rotation=90)

    # Display the counts above the bars
    for index, value in enumerate(brand_stats['battery_capacity']):
        plt.text(index, value, f"{value:.2f} mAh", ha='center', va='bottom',rotation=60)

    # Show the chart
    plt.show()
```

Fig 22: Display the average battery capacity for each brand

Secondly, Figure 23 creates a scatter plot to show the relationship between display size and battery capacity of devices using a colour gradient based on battery capacity, which helps highlight the variations for better visualisation. This function also helps users to observe whether there is a correlation between larger display sizes and higher battery capacities.

```
# Create a scatter plot to show the relationship between display size and battery capacity
# Create a scatter plot with a color gradient
plt.figure(figsize=(10, 6))
scatter = plt.scatter(
    loaded_data_pd['display_diagonal'],
    loaded_data_pd['battery_capacity'],
    c=loaded_data_pd['battery_capacity'],  # Use battery capacity for color gradient
    cmap='viridis',
    alpha=0.5
)
plt.xlabel('Display Diagonal (inches)')
plt.ylabel('Battery Capacity (mAh)')
plt.title('Relationship between Display Size and Battery Capacity')
plt.grid()

# Create a colorbar to show the legend for the color gradient
colorbar = plt.colorbar(scatter)
colorbar.set_label('Battery Capacity (mAh)')

# Show the scatter plot
plt.show()
```

Fig 23: Scatter plot to show the relationship between display size and battery capacity

2.2.5.5. Average_ppr(loaded_data_pd)

This function calculates and visually displays the average Price-Performance Ratio (PPR) for devices grouped by brand. It uses CPU, RAM, storage, and price as weightage, assigning weights to each. The resulting bar chart shows the average PPR for different brands, allowing users to compare the balance between device performance and cost using the performance-to-price ratio in Fig 24.

```python
def average_ppr(loaded_data_pd):
    # Create a bar chart to visualize the average Price-Performance Ratio for all devices
    df = loaded_data_pd

    # Define weightage factors
    weight_cpu = 0.4
    weight_ram = 0.3
    weight_storage = 0.2
    weight_price = 0.1

    # Calculate a performance score based on specifications
    df['Performance_Score'] = (weight_cpu * df['cpu_clock']) + \
                              (weight_ram * df['ram_capacity']) + \
                              (weight_storage * df['non_volatile_memory_capacity'])

    # Calculate price-performance ratio
    df['Price_Performance_Ratio'] = df['Performance_Score'] / df['price']
    # Group by brand and calculate the average Price-Performance Ratio
    average_ppr_by_brand = df.groupby('brand')['Price_Performance_Ratio'].mean().reset_index()
    # Create a bar chart to visualize the average Price-Performance Ratio for different brands
    plt.figure(figsize=(10, 6))
    plt.bar(average_ppr_by_brand['brand'], average_ppr_by_brand['Price_Performance_Ratio'])
    plt.xlabel('Brand')
    plt.ylabel('Average Price-Performance Ratio')
    plt.title('Average Price-Performance Ratio for Different Brands')
    plt.xticks(rotation=45)

    # Display the values for the bars
    for index, value in enumerate(average_ppr_by_brand['Price_Performance_Ratio']):
        plt.text(index, value, f"{value:.2f}", ha='center', va='bottom')

    plt.tight_layout()

    # Save the chart to an image file (optional)
    plt.savefig("average_price_performance_chart.png")

    # Show the chart
    plt.show()
```

Fig 24: Calculates and visually displays the average Price-Performance Ratio (PPR)

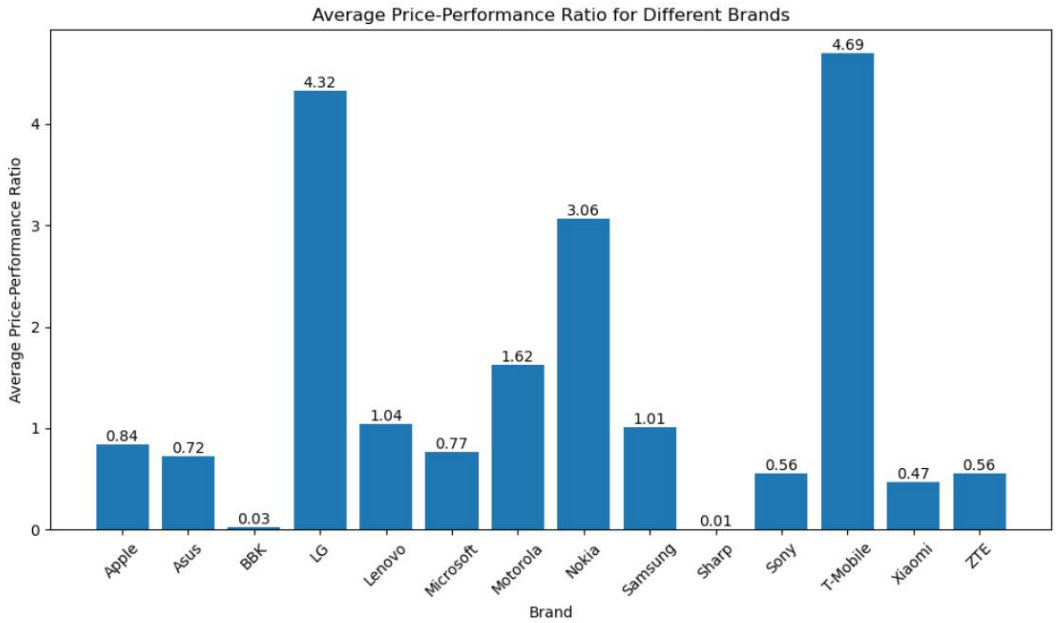

Fig 25:The average Price-performance Ratio for different brand

2.2.6. PhoneDB Main

This notebook takes in the importation of the five modules used for the program, and it contains a function named **Run_application**, which is used to run the program. It initiates the main menu (defined in the Tui module) and kicks off the execution of your PhoneDB program.

3.GitHub Repository Commit Evidence

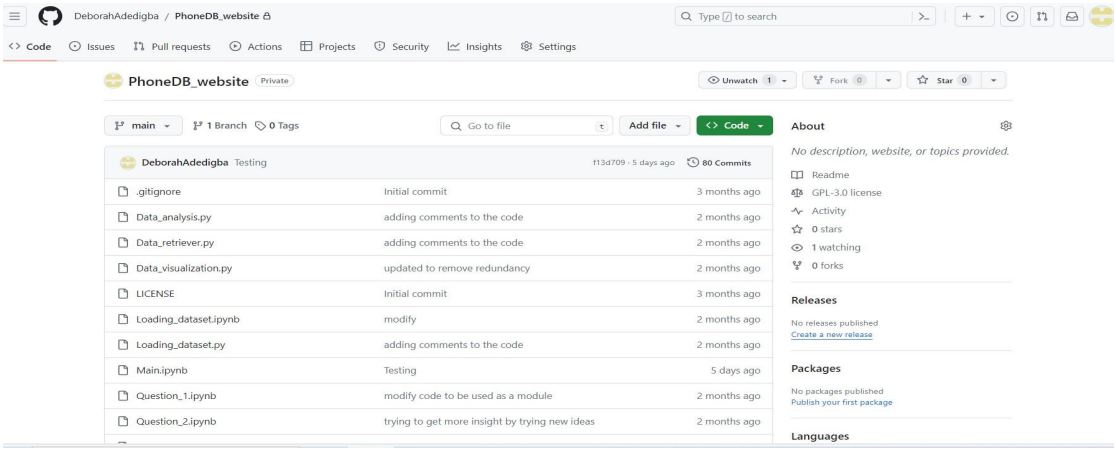

Fig 26: Screen-shot of the GitHub repository

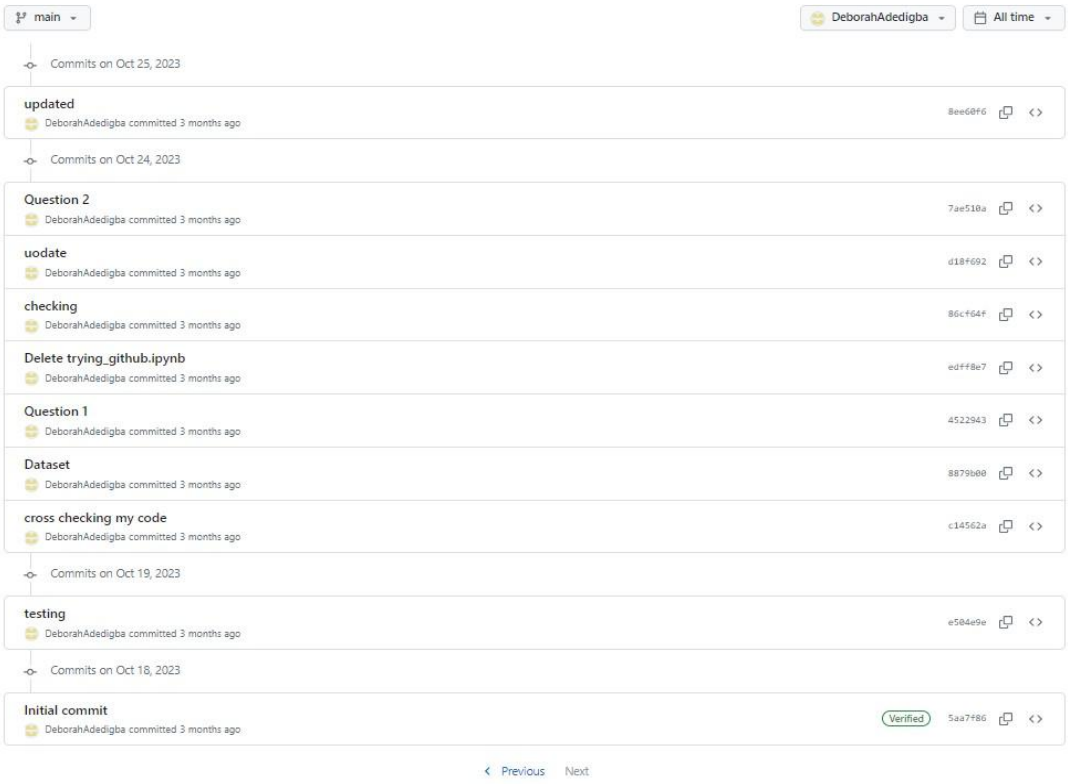

Fig 27: GitHub initial commit for the creation of the application and retrieval and analysis

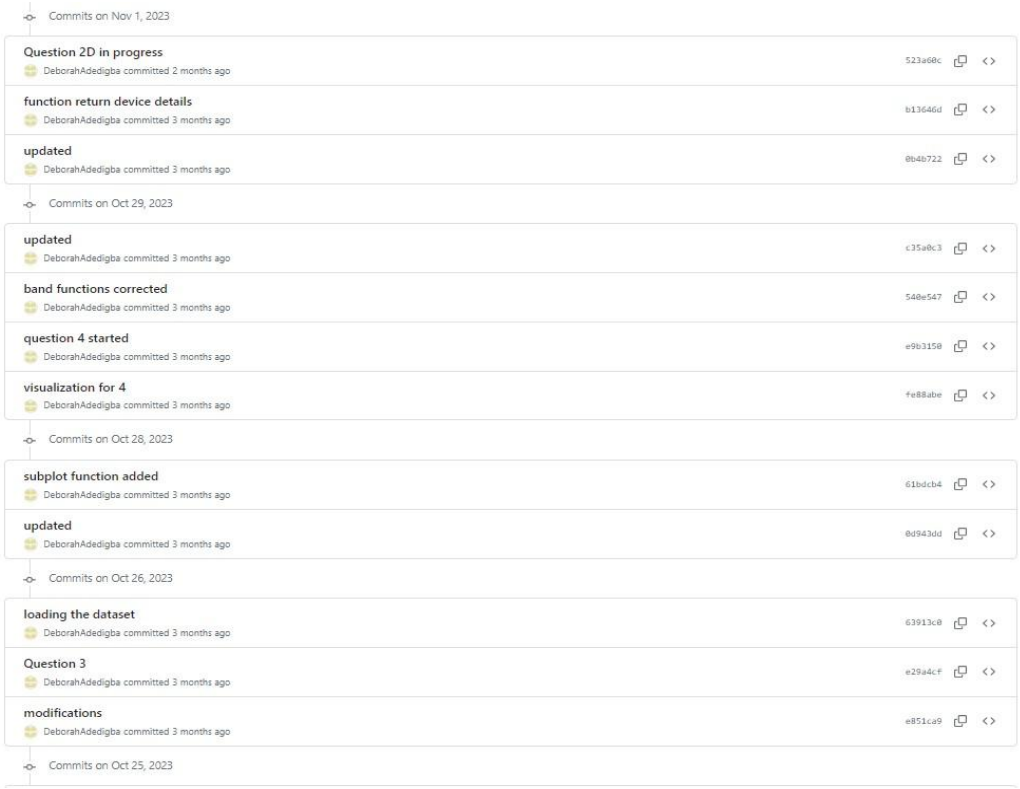

Fig 28: Github commit showing question 3(visualisations) and analysis updates

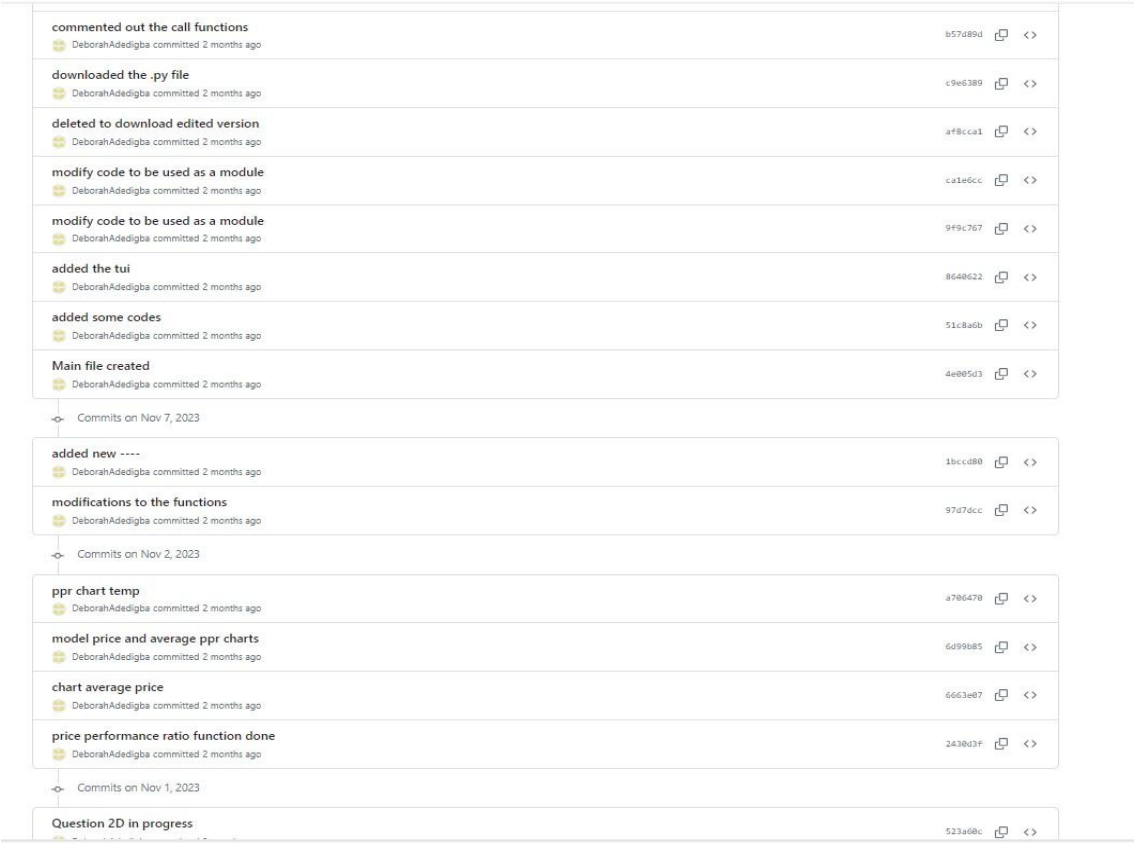

Fig 29: Github commit for finalising charts and creating TUI

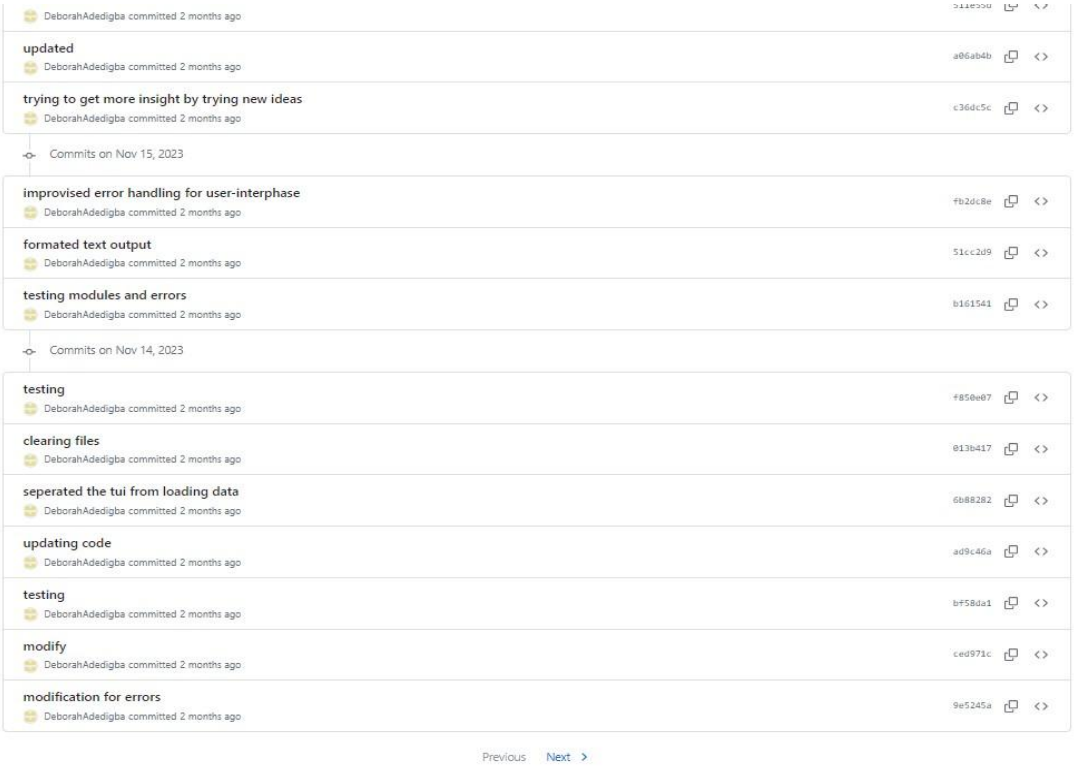

Fig 30: Github commit modification of codes and testing

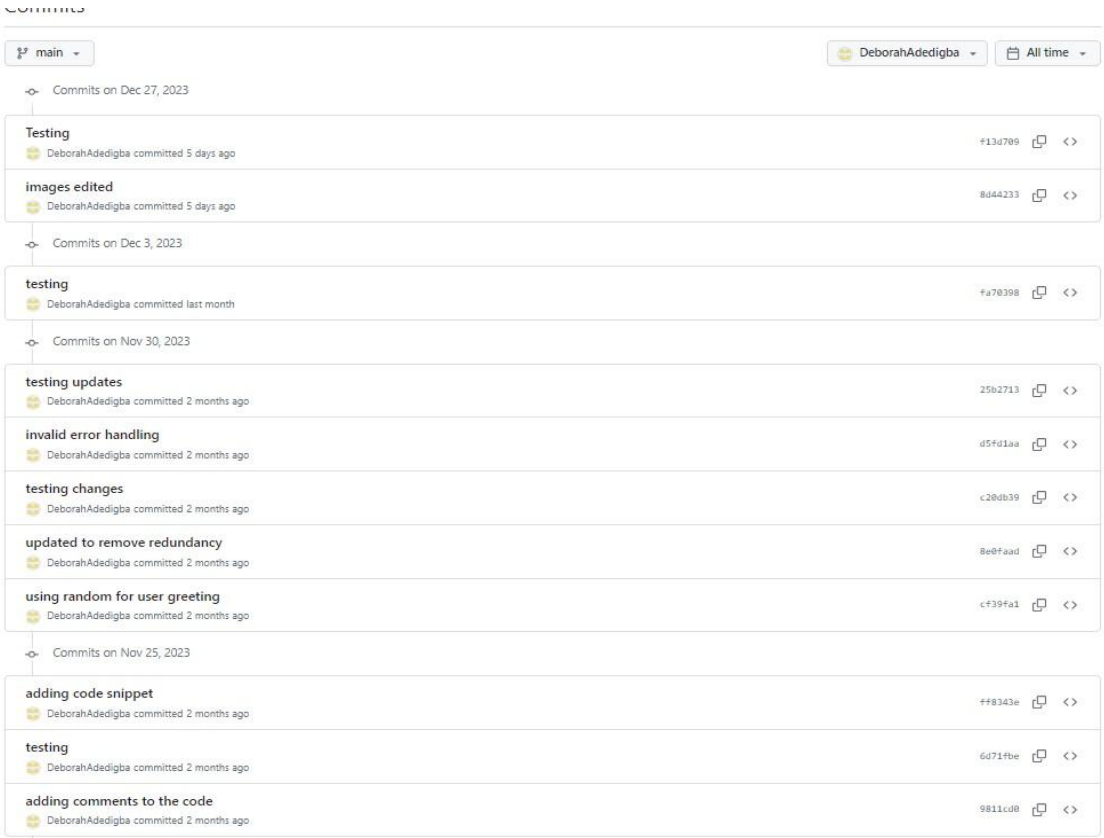

Fig 31: Github commit for testing and adjustment

Appendix

Table 2: Description of the features used from the dataset

S/N	Column Name	Description
1	oem_id	The original equipment manufacturer (OEM) identifier.
2	brand	The brand of the device.
3	model	The model name of the device
4	released_date	The date when the device was released.
5	manufacturer	The company or entity is responsible for manufacturing the device.
6	codename	A code name associated with the device.
7	device_category	The category to which the device belongs
8	width	The width dimension of the device.
9	dimensions	The overall dimensions of the device.
10	weight_gram	The weight of the device is in grams.
11	price	The price of the device.
12	price_currency	The currency in which the device price is specified
13	cpu_clock	The clock speed of the device's central processing unit (CPU)
14	ram_capacity	The capacity of RAM in the device.
15	non_volatile_memory_capacity	The capacity of non-volatile memory in the device.
16	sim_card_slot	Type of SIM card slot supported
17	battery_capacity	The capacity of the device's battery.
18	info_added_date	The date and time when the information was added.
19	market_regions	The regions in which the device is available in the market
20	announced_date	This is when a device is officially introduced to the public by the manufacturer.
21	software_extras	These are additional features in a device's operating system that enhance user experience (e.g., voice command and face recognition).
22	CPU	This is the device's primary component for executing instructions
23	ram_type	This specifies the memory technology used in a RAM module.
24	display_diagonal	Display_diagonal is the size of a device's screen, measured diagonally in inches.
25	pixel_density	Pixel_density indicates the number of pixels per inch on the display, influencing visual clarity.
26	usb_connector	The USB connector is a type of port used for connecting devices.
27	max_charging_power	Max_charging_power is the maximum power a device can receive while charging, indicating its capability.

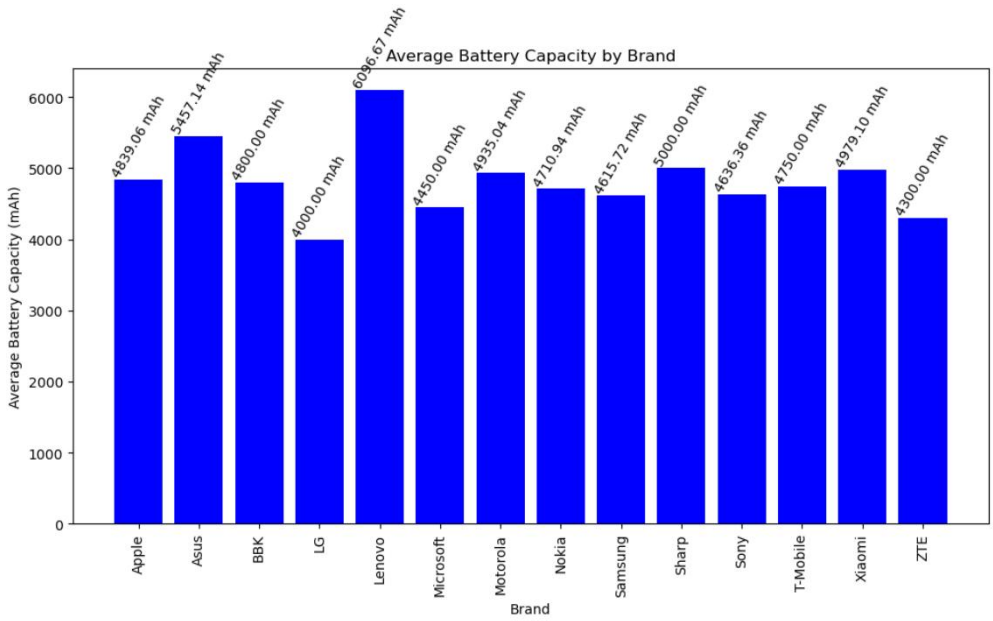

Fig 32: Average battery capacity by brands

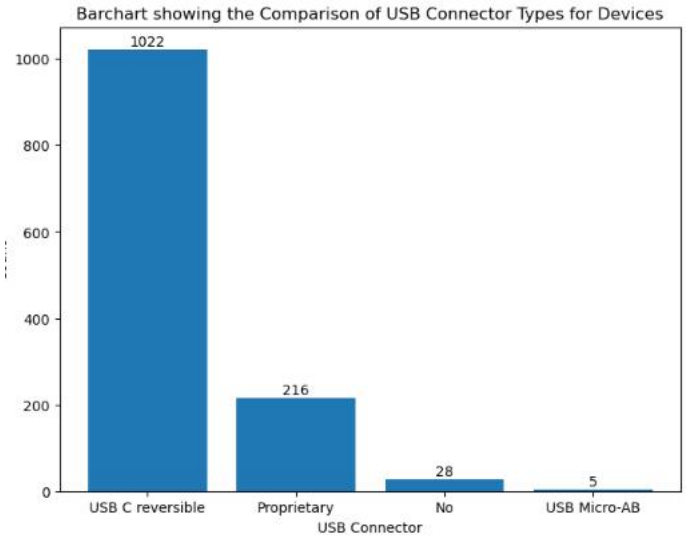

Fig 33: Comparison of USB connector types for devices