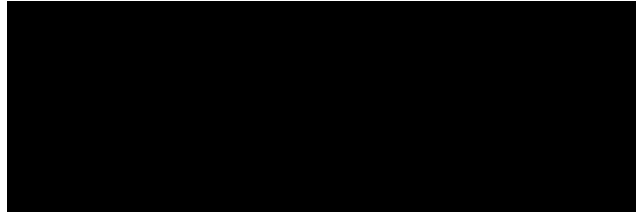


Databases and Analytics

Assessment

Course: BSc (Hons) Computer Science

Module: Databases and Analytics

Module Leader:

Student: Nunzio Emanuele Sgroi

Student ID:

Academic Year: 2023/2024

Document Structure

This assignment comprises three sections, each containing multiple deliverables. The sections were completed directly in Google Colab and subsequently exported as PDF files. These PDFs were then merged into a single comprehensive document. Each section was written in a separate Colab file, which can be accessed through the following links:

- **Section 1:** https://colab.research.google.com/drive/1fzzf0FBnk2ukGCLKvw2346pe1CL3gTP3?usp=sharing
- **Section 2:** https://colab.research.google.com/drive/1_9TzogdAztpgoW8XdZnkNt_tUU2D-Qcx?usp=sharing
- **Section 3**: https://colab.research.google.com/drive/1eyrpGsk9sNse-Gz_zWgZiYmGQSNSq9Zg?usp=sharing

Section 1 - Analyzing the Energy Efficiency of Buildings

Learning Outcome: Executing SQL Queries in R - Data Analytics in R

Student Name: Nunzio Emanuele Sgroi

Student ID:

Energy Efficiency of Buildings

Description:

Analyzing the Energy Efficiency of Buildings

The dataset includes features such as surface/wall/roof area, glass area, glass area distribution and orientation of 768 simulated building shapes. Based on this data, it is aimed to estimate the heating and cooling load of the building. In this age where environmental awareness is very important, it can be a super consultancy initiative to reduce our carbon footprint, as we still haven't switched to renewable energy sources.

Data Set Information:

We perform energy analysis using 12 different building shapes. The buildings differ with respect to the glazing area, the glazing area distribution, and the orientation, amongst other parameters. We simulate various settings as functions of the afore-mentioned characteristics to obtain 768 building shapes. The dataset comprises 768 samples and 8 features, aiming to analyse the energy consumption patterns in buildings and to predict two real valued responses.

Attribute Information:

The dataset contains eight attributes (or features, denoted by X1...X8) and two responses (or outcomes, denoted by Y1 and Y2). The aim is to use the eight features to analyze the dataset and predict each of the two responses.

- **X1:** Relative Compactness
- **X2:** Surface Area
- **X3:** Wall Area
- **X4:** Roof Area
- **X5:** Overall Height
- **X6:** Orientation
- **X7:** Glazing Area
- **X8:** Glazing Area Distribution
- **Y1:** Heating Load
- **Y2:** Cooling Load

A dataset “Energy_dataset.csv” is provided for data analytics. Further relevant assumptions can be made about the aspects not specified above if required.

Section 1 - Deliverable 1

Importing the Dataset: (2 Marks)

- How to import the dataset to GitHub?
- Can you explain how to import a dataset to GitHub?

To import a dataset to GitHub, follow these steps:

1. **Create a GitHub Repository:**
   - Navigate to GitHub and log in to your account.
   - Click on the "+" icon in the upper-right corner and select "New repository".
   - Provide a name for the repository, add a description, and choose the visibility (public or private).
   - Click on "Create repository".
2. **Upload the Dataset:**
   - Open the newly created repository.
   - Click on the "Add file" button and select "Upload files".
   - Drag and drop the dataset file (e.g., `Energy_dataset.csv`) into the upload area or click to choose the file from the file explorer.
   - Add a commit message (e.g., "Add Energy_dataset.csv").
   - Click on "Commit changes" to upload the dataset to the repository.

3. **Using Git Commands:** Alternatively, the dataset can be uploaded to GitHub using Git commands:
    - Initialize a local Git repository and add the remote repository URL.
    - Add the dataset file, commit the changes, and push to GitHub.

Example Git Commands:

```
# Open your terminal or command prompt

# Navigate to the project directory
cd path/to/the/project

# Initialize a new Git repository
git init

# Add the remote repository URL
git remote add origin https://github.com/Emanuele-Sgroi/example.git

# Add the dataset file to the repository
git add Energy_dataset.csv

# Commit the changes
git commit -m "Add Energy_dataset.csv"

# Push the changes to the GitHub repository
git push -u origin main
```

Section 1 - Deliverable 2

How to import the dataset in Google Colab? (3 Marks)

- Can you demonstrate how to select, insert, update, and delete records from tables using SQL statements?
- How to use mathematical expressions/aggregate functions/arithmetic functions to query and manipulate data?

How to import the dataset on Google Colab

The Dataset can be imported into Google Colab either manually or directly from GitHub:

1. **Manually Uploading the Dataset:**
    - Open Google Colab notebook.
    - Click on the "Files" icon on the left sidebar to open the file explorer.
    - Provide a name for the repository, add a description, and choose the visibility (public or private).
    - Click on the "Upload" button and select the dataset file (Energy_dataset.csv) from the local machine".
    - Load the dataset (code below)

```
import pandas as pd

# Load the dataset
df = pd.read_csv('Energy_dataset.csv')
```

2. Importing the Dataset from GitHub:**
    - Ensure that the dataset is uploaded to GitHub as described above.
    - Copy the URL of the raw dataset file from your GitHub repository.
    - Use the following code to load the dataset directly from GitHub:

```
import pandas as pd

# example of how the link would look like
url = "https://raw.githubusercontent.com/Emanuele-Sgroi/repository/main/Energy_dataset.csv"
df = pd.read_csv(url)
```

Demonstrating SQL Operations:

To demonstrate SQL operations such as select, insert, update, and delete it was used `sqldf` function from the `pandasql` library:

1. Installing pandasql:

```
!pip install pandasql
```

2. Importing pandasql: (Dataset already loaded from the code above)

```
import pandasql as psql

# Display the first few rows of the dataframe
df.head()
```

	Relative Compactness	Surface Area	Wall Area	Roof Area	Overall Height	Orientation	Glazing Area	Glazing Area Distribution	H
0	0.98	514.5	294.0	110.25	7.0	2	0.0	0	
1	0.98	514.5	294.0	110.25	7.0	3	0.0	0	
2	0.98	514.5	294.0	110.25	7.0	4	0.0	0	
3	0.98	514.5	294.0	110.25	7.0	5	0.0	0	

Next steps: View recommended plots

3. Selecting Records:

```
# Select records from the dataframe
query = "SELECT * FROM df LIMIT 5"
result = psql.sqldf(query, locals())

# Display the result
result
```

	Relative Compactness	Surface Area	Wall Area	Roof Area	Overall Height	Orientation	Glazing Area	Glazing Area Distribution	H
0	0.98	514.5	294.0	110.25	7.0	2	0.0	0	
1	0.98	514.5	294.0	110.25	7.0	3	0.0	0	
2	0.98	514.5	294.0	110.25	7.0	4	0.0	0	
3	0.98	514.5	294.0	110.25	7.0	5	0.0	0	

Next steps: View recommended plots

4. Inserting Records:

Inserting records directly into a DataFrame using SQL is not straightforward. However, records can be appended to a DataFrame using `pandas`.

Equivalent SQL Query to insert a record:

```
INSERT INTO df (Relative_Compactness, Surface_Area, Wall_Area, Roof_Area, Overall_Height, Orientation, Glazing_Area, Glazing_Area_Distribution, He
VALUES (0.85, 563.5, 318.5, 122.5, 7.0, 2, 0.1, 3, 29.25, 28.30);
```

```
# Insert a new record into the dataframe
new_record = pd.DataFrame({
    'Relative Compactness': [0.85],
    'Surface Area': [563.5],
    'Wall Area': [318.5],
    'Roof Area': [122.5],
    'Overall Height': [7.0],
    'Orientation': [2],
    'Glazing Area': [0.1],
    'Glazing Area Distribution': [3],
    'Heating Load': [29.25],
    'Cooling Load': [28.30]
})

df = pd.concat([df, new_record], ignore_index=True)

# Display the last few rows of the dataframe to confirm insertion
df.tail()
```

	Relative Compactness	Surface Area	Wall Area	Roof Area	Overall Height	Orientation	Glazing Area	Glazing Area Distribution
764	0.62	808.5	367.5	220.5	3.5	2	0.4	5
765	0.62	808.5	367.5	220.5	3.5	3	0.4	5
766	0.62	808.5	367.5	220.5	3.5	4	0.4	5
767	0.62	808.5	367.5	220.5	3.5	5	0.4	5

5. Updating Records:

While direct SQL updates on DataFrames are not possible with `sqldf`, the equivalent DataFrame operation can be performed as follows:

SQL Query to update:

```
UPDATE df
SET Heating_Load = 30.00
WHERE Relative_Compactness = 0.85;
```

Equivalent DataFrame operation:

```
# Update a record in the dataframe
df.loc[df['Relative Compactness'] == 0.85, 'Heating Load'] = 30.00

# Display the updated records
df[df['Relative Compactness'] == 0.85]
```

Relative Compactness	Surface Area	Wall Area	Roof Area	Overall Height	Orientation	Glazing Area	Glazing Area Distribution

6. Deleting Records:

Deleting records directly from a DataFrame using SQL is not straightforward. However, records can be removed from a DataFrame using pandas.

```
DELETE FROM df
WHERE Relative_Compactness = 0.85;
```

```
# Delete a record from the dataframe
df = df[df['Relative Compactness'] != 0.85]

# Display the dataframe to confirm deletion
df.tail()
```

	Relative Compactness	Surface Area	Wall Area	Roof Area	Overall Height	Orientation	Glazing Area	Glazing Area Distribution
763	0.64	784.0	343.0	220.5	3.5	5	0.4	5
764	0.62	808.5	367.5	220.5	3.5	2	0.4	5
765	0.62	808.5	367.5	220.5	3.5	3	0.4	5
766	0.62	808.5	367.5	220.5	3.5	4	0.4	5

Using Mathematical Expressions/Aggregate Functions/Arithmetic Functions to Query and Manipulate Data:

Example of using mathematical expressions, aggregate functions, and arithmetic functions in SQL queries:

Equivalent SQL Query to get the average heating load:

```
SELECT AVG(`Heating Load`) as Average_Heating_Load FROM df;
```

```
# Example of using aggregate functions to get the average heating load
query = "SELECT AVG(`Heating Load`) as Average_Heating_Load FROM df"
average_heating_load = psql.sqldf(query, locals())
average_heating_load
```

	Average_Heating_Load
0	22.307201

Example of using mathematical expressions:

Equivalent SQL Query to calculate total load:

```
SELECT `Heating Load` + `Cooling Load` AS Total_Load FROM df LIMIT 5;
```

```
# Example of using mathematical expressions to calculate total load
query = "SELECT `Heating Load` + `Cooling Load` AS Total_Load FROM df LIMIT 5"
total_loads = psql.sqldf(query, locals())
total_loads
```

	Total_Load
0	36.88
1	36.88
2	36.88
3	36.88
4	49.12

Next steps: View recommended plots

Example of using arithmetic functions:

Equivalent SQL Query to find the maximum heating load:

```
SELECT MAX(`Heating Load`) as Max_Heating_Load FROM df;
```

```
# Example of using arithmetic functions to find the maximum heating load
query = "SELECT MAX(`Heating Load`) as Max_Heating_Load FROM df"
max_heating_load = psql.sqldf(query, locals())
max_heating_load
```

	Max_Heating_Load
0	43.1

Section 1 - Deliverable 3

Executing SQL Queries in R: (4 Marks)

- How do you execute SQL queries in R?
- Can you provide three examples of SQL queries executed within R?

Executing SQL queries in R

To execute SQL queries in R, the `sqldf` package is used, which is different from the `pandasql` library used before. The `sqldf` package in R allows running SQL statements on R data frames, facilitating the manipulation and querying of data in a familiar SQL syntax.

Steps to Execute SQL Queries in R:

1. **Install the `sqldf` package**.
2. **Load the `sqldf` library**.
3. **Use the `sqldf()` function** to run SQL queries on data frames.

Example Process:

1. Install and load the `sqldf` package.
2. Load the dataset.

Below are the steps and code to execute these in an R runtime within Google Colab.

```
# Install the sqldf package
# install.packages("sqldf") ### Commented because already installed

# Load the sqldf library
library(sqldf)

# Load sample data
data <- read.csv("Energy_dataset.csv")

# Clean column names to avoid issues with spaces and special characters
colnames(data) <- make.names(colnames(data))

# Display the cleaned column names
print(colnames(data))

# Display the first few rows of the dataset
head(data)
```

```
Loading required package: gsubfn

Loading required package: proto

Warning message:
"no DISPLAY variable so Tk is not available"
Loading required package: RSQLite

 [1] "Relative.Compactness"      "Surface.Area"
 [3] "Wall.Area"                 "Roof.Area"
 [5] "Overall.Height"            "Orientation"
 [7] "Glazing.Area"              "Glazing.Area.Distribution"
 [9] "Heating.Load"              "Cooling.Load"
```

A data.frame: 6 × 10

	Relative.Compactness	Surface.Area	Wall.Area	Roof.Area	Overall.Height	Orientation	Glazing.Area	Glazing.Area.Distribution	H
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<int>	<dbl>	<int>	
1	0.98	514.5	294.0	110.25	7	2	0	0	
2	0.98	514.5	294.0	110.25	7	3	0	0	
3	0.98	514.5	294.0	110.25	7	4	0	0	
4	0.98	514.5	294.0	110.25	7	5	0	0	
5	0.90	563.5	318.5	122.50	7	2	0	0	
6	0.90	563.5	318.5	122.50	7	3	0	0	

The subsequent examples will demonstrate how to run various types of SQL queries using the `sqldf` package with R environment.

```
# Example of a SELECT query
query <- "SELECT * FROM data LIMIT 5"
result <- sqldf(query)
print(result)
```

```
  Relative.Compactness Surface.Area Wall.Area Roof.Area Overall.Height
1                 0.98        514.5     294.0    110.25              7
2                 0.98        514.5     294.0    110.25              7
3                 0.98        514.5     294.0    110.25              7
4                 0.98        514.5     294.0    110.25              7
5                 0.90        563.5     318.5    122.50              7
  Orientation Glazing.Area Glazing.Area.Distribution Heating.Load Cooling.Load
1           2            0                         0        15.55        21.33
2           3            0                         0        15.55        21.33
3           4            0                         0        15.55        21.33
4           5            0                         0        15.55        21.33
5           2            0                         0        20.84        28.28
```

```
# Example of a COUNT query
query <- "SELECT COUNT(*) AS Row_Count FROM data"
result <- sqldf(query)
print(result)
```

```
  Row_Count
1       768
```

```
# Example to calculate the average heating load and cooling load for each combination of relative compactness and orientation
query <- 'SELECT "Relative.Compactness", Orientation, AVG("Heating.Load") AS Avg_Heating_Load, AVG("Cooling.Load")
AS Avg_Cooling_Load FROM data GROUP BY "Relative.Compactness", Orientation'
result <- sqldf(query)

# Print the top 10 results
print(head(result, 10))
```

```
   Relative.Compactness Orientation Avg_Heating_Load Avg_Cooling_Load
1                  0.62           2         14.07563         15.34375
2                  0.62           3         14.35500         15.38625
3                  0.62           4         14.35187         15.14562
4                  0.62           5         14.35062         15.10063
5                  0.64           2         16.65812         20.12312
6                  0.64           3         16.61125         20.12500
7                  0.64           4         16.60187         20.33500
8                  0.64           5         16.59000         20.32562
9                  0.66           2         12.86375         15.75875
10                 0.66           3         12.83875         15.78625
```

Section 1 - Deliverable 4

Data Manipulation and Transformation in R: (3 Marks)

- Explain how to perform data manipulation and transformation tasks using built-in functions and packages in R.
- Can you demonstrate three examples of data manipulation tasks using R functions and packages?

Data manipulation and transformation using built-in functions and packages in R:

Data manipulation and transformation in R can be efficiently performed using packages such as `dplyr` and `tidyr`. These packages provide a suite of functions for data cleaning, filtering, summarizing, and reshaping.

Common functions in `dplyr`:

- `filter()`: Filter rows based on conditions.
- `select()`: Select specific columns.
- `mutate()`: Add or modify columns.
- `summarize()`: Summarize data by groups.
- `arrange()`: Sort data.

Common functions in `tidyr`:

- `gather()`: Convert wide data to long format.
- `spread()`: Convert long data to wide format.
- `unite()`: Combine multiple columns into one.
- `separate()`: Split a column into multiple columns.

Demonstration of data manipulation tasks using R functions and packages:

```
# Install necessary packages
install.packages("dplyr")
install.packages("tidyr")

# Load necessary packages
library(dplyr)
library(tidyr)
```

```
Installing package into '/usr/local/lib/R/site-library'
(as 'lib' is unspecified)

Installing package into '/usr/local/lib/R/site-library'
(as 'lib' is unspecified)
```

```
# Load the dataset
data <- read.csv("Energy_dataset.csv")
```

```
# Example 1: Filter rows where Heating Load is greater than 20
filtered_data <- data %>% filter(Heating.Load > 20)
head(filtered_data)
```

A data.frame: 6 ×

	Relative.Compactness	Surface.Area	Wall.Area	Roof.Area	Overall.Height	Orientat
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<i
1	0.90	563.5	318.5	122.5	7	
2	0.90	563.5	318.5	122.5	7	
3	0.90	563.5	318.5	122.5	7	
4	0.79	637.0	343.0	147.0	7	
5	0.79	637.0	343.0	147.0	7	
6	0.79	637.0	343.0	147.0	7	

```
# Example 2: Select specific columns (Relative Compactness, Surface Area, Heating Load)
selected_data <- data %>% select(Relative.Compactness, Surface.Area, Heating.Load)
head(selected_data)
```

A data.frame: 6 × 3

	Relative.Compactness	Surface.Area	Heating.Load
	<dbl>	<dbl>	<dbl>
1	0.98	514.5	15.55
2	0.98	514.5	15.55
3	0.98	514.5	15.55
4	0.98	514.5	15.55
5	0.90	563.5	20.84
6	0.90	563.5	21.46

```
# Example 3: Create a new column for Total Load (Heating Load + Cooling Load)
data_with_total_load <- data %>% mutate(Total.Load = Heating.Load + Cooling.Load)
head(data_with_total_load)
```

A data.frar

	Relative.Compactness	Surface.Area	Wall.Area	Roof.Area	Overall.Height	Orientat
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<i
1	0.98	514.5	294.0	110.25	7	
2	0.98	514.5	294.0	110.25	7	
3	0.98	514.5	294.0	110.25	7	
4	0.98	514.5	294.0	110.25	7	
5	0.90	563.5	318.5	122.50	7	
6	0.90	563.5	318.5	122.50	7	

Section 1 - Deliverable 5

Data Visualization in R: (3 Marks)**

- How do you create informative and visually appealing plots using the R package for data visualization?
- Provide three examples of different types of plots that can be created using R and explain their significance in data analysis.

Creating informative and visually appealing plots using the R package for data visualization:

To create informative and visually appealing plots in R, the `ggplot2` package is recommended. `ggplot2` is a powerful and flexible package for creating complex and customized visualizations. It is based on the Grammar of Graphics, which provides a coherent system for describing and building graphs.

Steps to create a plot using `ggplot2`:

1. Install and load the `ggplot2` package.
2. Use the `ggplot()` function to define the aesthetics of the plot.
3. Add layers to the plot using `geom_*` functions (e.g., `geom_point`, `geom_line`, `geom_bar`).
4. Customize the plot with themes, labels, and other enhancements.

A demonstration of installation and usage is provided below.

```
# Install necessary packages
install.packages("ggplot2")

# Load necessary packages
library(ggplot2)

# Dataset already loaded
```

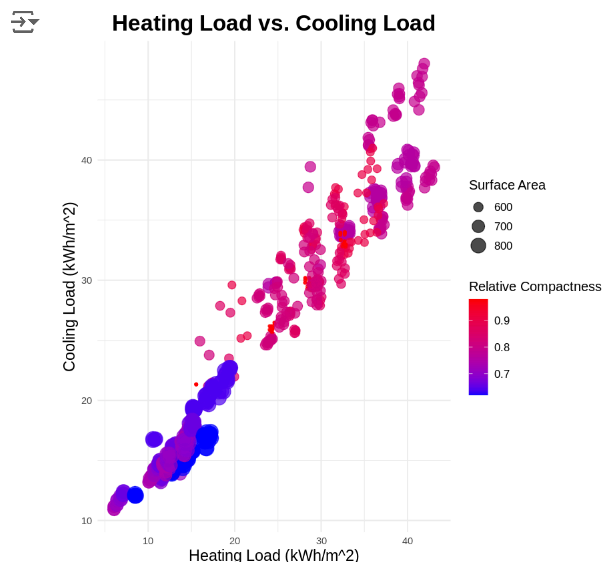
```
Installing package into '/usr/local/lib/R/site-library'
(as 'lib' is unspecified)
```

Example 1: Scatter Plot

Significance: Scatter plots are used to visualize the relationship between two continuous variables.

The example shows the relationship between Heating Load and Cooling Load, with color representing Relative Compactness and size representing Surface Area. This plot helps identify correlations and patterns between the two variables.

```
ggplot(data, aes(x = Heating.Load, y = Cooling.Load)) +
  geom_point(aes(color = Relative.Compactness, size = Surface.Area), alpha = 0.7) +
  scale_color_gradient(low = "blue", high = "red") +
  labs(title = "Heating Load vs. Cooling Load",
       x = "Heating Load (kWh/m^2)",
       y = "Cooling Load (kWh/m^2)",
       color = "Relative Compactness",
       size = "Surface Area") +
  theme_minimal() +
  theme(
    plot.title = element_text(hjust = 0.5, size = 20, face = "bold"),
    axis.title = element_text(size = 14),
    legend.title = element_text(size = 12),
    legend.text = element_text(size = 10)
  )
```

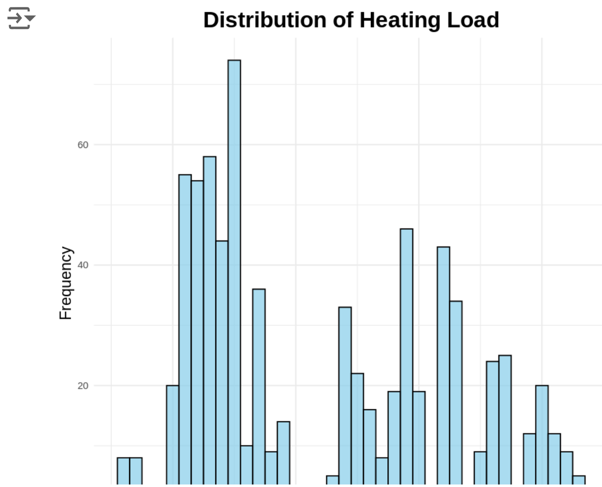


Example 2: Histogram

Significance: Histograms are used to visualize the distribution of a single continuous variable.

The example displays the distribution of Heating Load. Histograms are useful for understanding the frequency distribution of a single variable and identifying any skewness or bimodality in the data.

```
ggplot(data, aes(x = Heating.Load)) +
  geom_histogram(binwidth = 1, fill = "skyblue", color = "black", alpha = 0.7) +
  labs(title = "Distribution of Heating Load",
       x = "Heating Load (kWh/m^2)",
       y = "Frequency") +
  theme_minimal() +
  theme(
    plot.title = element_text(hjust = 0.5, size = 20, face = "bold"),
    axis.title = element_text(size = 14)
  )
```

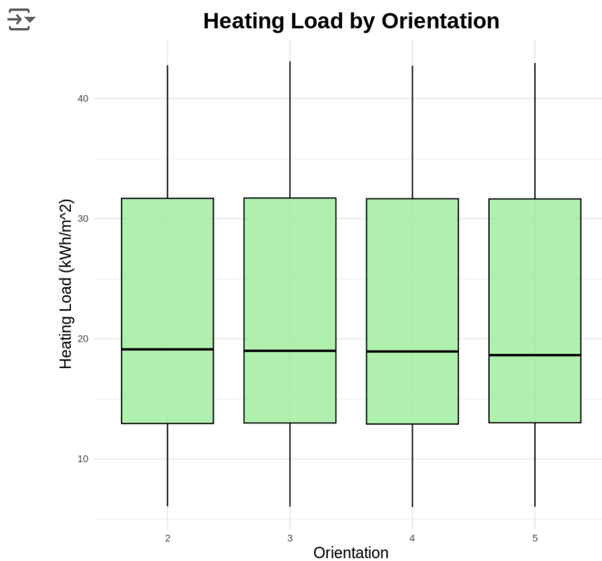


Example 3: Box Plot

Significance: Box plots are used to visualize the distribution of a continuous variable and identify outliers.

The example visualizes the distribution of Heating Load across different orientations. Box plots are useful for comparing distributions and identifying outliers across categories.

```
ggplot(data, aes(x = as.factor(Orientation), y = Heating.Load)) +
  geom_boxplot(fill = "lightgreen", color = "black", alpha = 0.7) +
  labs(title = "Heating Load by Orientation",
       x = "Orientation",
       y = "Heating Load (kWh/m^2)") +
  theme_minimal() +
  theme(
    plot.title = element_text(hjust = 0.5, size = 20, face = "bold"),
    axis.title = element_text(size = 14)
  )
```



End of Section 1

Nunzio Emanuele Sgroi -

Section 2 - Telecom Case Study

Learning Outcome: Pandas and NumPy Exercise for Data Analysis

Student Name: Nunzio Emanuele Sgroi

Student ID: [redacted]

Telecom Case Study

Description

This case study is about a telecom company. Data analytics can help this company to make smarter decisions that lead to higher productivity and more efficient operations. One of the key parameters that should be checked for this company is churn rate. Churn, often known as "churn rate," is the measurement of the number of subscription customers who have cancelled their subscription over a set period of time.

Generally, churn rate, sometimes known as attrition rate in business, is the rate at which customers stop doing business with a company over a given period of time. Churn rate is the percentage of subscribers to a service that discontinue their subscription to that service in each period. In order for a company to expand its clientele, its growth rate (i.e. its number of new customers) must exceed its churn rate.

The first way to reduce the churn rate and consequently improve business retention is to make the most effective use of a telco's internal data. All this data allows telecom companies to know customers better–to the point where they can collect them into precise and consistent segments. In this case study, with some variables we need to show whether a particular customer will switch to another telecom provider or not. In telecom terminology, this is referred to as churning and not churning, respectively. Some improvements will make possible in this company through your data analytics. A dataset "telecom_dataset.csv" is provided for data analytics. You need to complete each step in the following order.

Section 2 - Deliverable 1

Importing and Combining Data: (2 Marks)

- How to import the `telecom_dataset.csv` data and combining all data files into one consolidated dataframe?
- Can you outline the steps to import a CSV file into a Pandas DataFrame?
- How would you ensure data integrity when combining multiple data files into a single DataFrame?
- Could you demonstrate how to verify that the combined DataFrame contains all the data from the individual files without any loss or duplication?

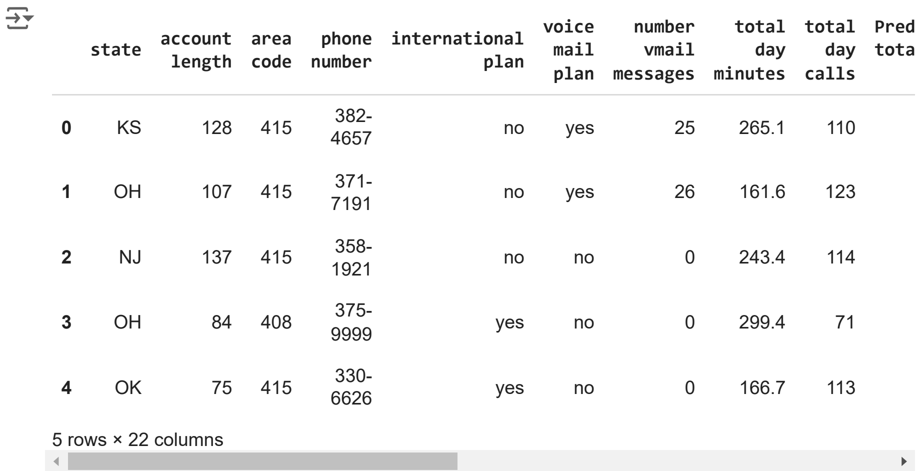
Importing Dataset

To import the `telecom_dataset.csv` data, the Pandas library in Python can be used. The following steps outline the process:

Steps to import a CSV file into a Pandas DataFrame:

1. Install and import the Pandas library.
   - If Pandas is not already installed, it can be installed using `!pip install pandas`.
2. **Load the CSV File into a DataFrame:**
   - Use the `pd.read_csv()` function to load the CSV file into a DataFrame.
   - Display the first few rows of the DataFrame to verify the data.

```
# Import Pandas library
import pandas as pd

# Load the dataset
df = pd.read_csv("telecom_dataset.csv")

# Display the first few rows
df.head()
```



	state	account length	area code	phone number	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	Pred tota
0	KS	128	415	382-4657	no	yes	25	265.1	110	
1	OH	107	415	371-7191	no	yes	26	161.6	123	
2	NJ	137	415	358-1921	no	no	0	243.4	114	
3	OH	84	408	375-9999	yes	no	0	299.4	71	
4	OK	75	415	330-6626	yes	no	0	166.7	113	

5 rows × 22 columns

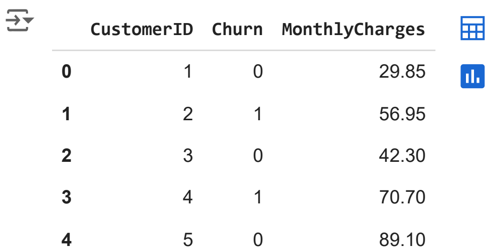
Combining multiple data files into one consolidated DataFrame:

If there are multiple data files to be combined, they can be concatenated into a single DataFrame using the `pd.concat()` function. Here's how to combine multiple DataFrames:

1. Load each CSV file into a separate DataFrame.
2. Combine all DataFrames into one consolidated DataFrame using `pd.concat()`.

The following example uses dummy data for demonstration:

```
# Dummy data 1
data1 = {
    'CustomerID': [1, 2, 3],
    'Churn': [0, 1, 0],
    'MonthlyCharges': [29.85, 56.95, 42.30]
}

# Dummy data 2
data2 = {
    'CustomerID': [4, 5, 6],
    'Churn': [1, 0, 0],
    'MonthlyCharges': [70.70, 89.10, 29.75]
}

# Convert dictionaries to DataFrames
df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)

# Combine all DataFrames into one consolidated DataFrame
combined_dummy_df = pd.concat([df1, df2], ignore_index=True)

# Display the combined dummy DataFrame
combined_dummy_df.head()
```



	CustomerID	Churn	MonthlyCharges
0	1	0	29.85
1	2	1	56.95
2	3	0	42.30
3	4	1	70.70
4	5	0	89.10

Next steps: View recommended plots

Ensuring data integrity when combining multiple data files

Ensuring data integrity is crucial when combining multiple data files. The following steps should be taken:

1. **Check for Consistency in Column Names and Data Types:**
    - Ensure that all DataFrames have the same column names and data types.
2. **Handle Missing Values Appropriately:**
    - Identify and handle any missing values in the DataFrames.
3. **Remove Duplicate Rows:**

- Use the `duplicated()` function to identify duplicate rows.
- Remove duplicate rows using the `drop_duplicates()` function to prevent data loss or duplication.

4. **Verify the Combined DataFrame:**
   - Compare the number of rows in the original and combined DataFrames to ensure no data loss.
   - Display information about the combined DataFrame using the `info()` function.

The following code show an example of how to ensure data integrity, using the combine dummy dataset created above:

```
# Check for duplicate rows
duplicates = combined_dummy_df.duplicated()
if duplicates.any():
    print("Duplicate rows found and will be removed.")
    combined_dummy_df = combined_dummy_df.drop_duplicates()

# Verify the combined DataFrame
print("Number of rows in dummy DataFrame 1:", len(df1))
print("Number of rows in dummy DataFrame 2:", len(df2))
print("Number of rows in combined dummy DataFrame:", len(combined_dummy_df))

# Verify that the combined DataFrame contains all data
print("Combined DataFrame contains all data from individual files:")
combined_dummy_df.info()

# Display the first few rows of the combined dummy DataFrame to confirm
combined_dummy_df.head()
```

```
Number of rows in dummy DataFrame 1: 3
Number of rows in dummy DataFrame 2: 3
Number of rows in combined dummy DataFrame: 6
Combined DataFrame contains all data from individual files:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 3 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   CustomerID      6 non-null      int64  
 1   Churn           6 non-null      int64  
 2   MonthlyCharges  6 non-null      float64
dtypes: float64(1), int64(2)
memory usage: 272.0 bytes
```

	CustomerID	Churn	MonthlyCharges
0	1	0	29.85
1	2	1	56.95
2	3	0	42.30
3	4	1	70.70
4	5	0	89.10

Next steps: View recommended plots

The following example will take into account the dataset `telecom_dataset.csv`, and will show steps to ensure integrity. Despite the fact that this dataset was not combined with other datasets, there are still steps that should be taken to double-check the integrity, which are similar to the steps shown above.

1. **Checking for Missing Values:**
   - Use `isnull()` and `sum()` functions to identify missing values in each column.
2. **Removing Duplicate Rows:**
   - Use the `duplicated()` function to identify duplicate rows.
   - Remove duplicate rows using the `drop_duplicates()` function.
3. **Verifying Data Types:**
   - Ensure that all columns have the expected data types using the `dtypes` attribute.

```
# Check for missing values
missing_values = df.isnull().sum()
print("Missing values in each column:")
print(missing_values)
```

```
Missing values in each column:
state                    0
account length           0
```

```
area code                 0
phone number              0
international plan        0
voice mail plan           0
number vmail messages     0
total day minutes         0
total day calls           0
Predicted total day calls    0
total day charge          0
total eve minutes         0
total eve calls           0
total eve charge          0
total night minutes       0
total night calls         0
total night charge        0
total intl minutes        0
total intl calls          0
total intl charge         0
customer service calls    0
churn                     0
dtype: int64
```

```
# Check for duplicate rows
original_duplicates = df.duplicated()
if original_duplicates.any():
    print("Duplicate rows found and will be removed.")
    df = df.drop_duplicates()
```

```
# Verify data types
print("Data types of each column:")
print(df.dtypes)
```

```
Data types of each column:
state                         object
account length                 int64
area code                      int64
phone number                  object
international plan            object
voice mail plan               object
number vmail messages          int64
total day minutes            float64
total day calls                int64
Predicted total day calls      int64
total day charge             float64
total eve minutes            float64
total eve calls                int64
total eve charge             float64
total night minutes          float64
total night calls              int64
total night charge           float64
total intl minutes           float64
total intl calls               int64
total intl charge            float64
customer service calls         int64
churn                           bool
dtype: object
```

```
# Verify that the original DataFrame is clean
print("Original DataFrame info after cleaning:")
df.info()
```

```
Original DataFrame info after cleaning:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3333 entries, 0 to 3332
Data columns (total 22 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   state                      3333 non-null   object
 1   account length             3333 non-null   int64
 2   area code                  3333 non-null   int64
 3   phone number               3333 non-null   object
 4   international plan         3333 non-null   object
 5   voice mail plan            3333 non-null   object
 6   number vmail messages      3333 non-null   int64
 7   total day minutes          3333 non-null   float64
 8   total day calls            3333 non-null   int64
 9   Predicted total day calls  3333 non-null   int64
 10  total day charge           3333 non-null   float64
 11  total eve minutes          3333 non-null   float64
 12  total eve calls            3333 non-null   int64
 13  total eve charge           3333 non-null   float64
 14  total night minutes        3333 non-null   float64
 15  total night calls          3333 non-null   int64
 16  total night charge         3333 non-null   float64
 17  total intl minutes         3333 non-null   float64
 18  total intl calls           3333 non-null   int64
```

```
 19  total intl charge          3333 non-null   float64
 20  customer service calls     3333 non-null   int64
 21  churn                      3333 non-null   bool
dtypes: bool(1), float64(8), int64(9), object(4)
memory usage: 550.2+ KB
```

```
# Display the first few rows of the cleaned original DataFrame to confirm
df.head()
```

	state	account length	area code	phone number	international plan	voice mail plan	number vmail messages	total day minutes	total day calls	Pred tota
0	KS	128	415	382-4657	no	yes	25	265.1	110	
1	OH	107	415	371-7191	no	yes	26	161.6	123	
2	NJ	137	415	358-1921	no	no	0	243.4	114	
3	OH	84	408	375-9999	yes	no	0	299.4	71	
4	OK	75	415	330-6626	yes	no	0	166.7	113	

5 rows × 22 columns

Section 2 - Deliverable 2

Analyzing Data with NumPy and Pandas: (10 Marks)

- How do you analyze `telecom_dataset.csv` data in NumPy and Pandas libraries?
- Explain how you would handle missing data during data analysis using Pandas.
- Can you calculate and provide examples of commonly used statistics in data analysis, such as mean, median, standard deviation, minimum, maximum, and quantiles, using Pandas and NumPy?
- How would you interpret these statistics in the context of your analysis?

Analyzing `telecom_dataset.csv` data in NumPy and Pandas libraries and handling missing data

The analysis of the `telecom_dataset.csv` data involves several steps using the Pandas and NumPy libraries. Below are the detailed steps, along with the corresponding code snippets.

Step 1: Install and Import the Necessary Libraries

First, it is essential to confirm the installation and importation of the requisite libraries. Pandas serves as the primary tool for data manipulation and analysis, whereas NumPy facilitates various numerical operations.

```
# Import necessary libraries
# import pandas as pd
import numpy as np
```

Step 2: Load the Dataset

Use Pandas to load the `telecom_dataset.csv` data into a DataFrame. This step is already done with the previous deliverable.

Step 3: Understand the Structure of the Data

Inspect the structure of the DataFrame to understand the columns, data types, and any initial observations about the data.

```
# Display basic information about the DataFrame
df.info()

# Display summary statistics of the DataFrame
df.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3333 entries, 0 to 3332
Data columns (total 22 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   state                    3333 non-null   object
 1   account length           3333 non-null   int64
 2   area code                3333 non-null   int64
 3   phone number             3333 non-null   object
 4   international plan       3333 non-null   object
 5   voice mail plan          3333 non-null   object
 6   number vmail messages    3333 non-null   int64
 7   total day minutes        3333 non-null   float64
 8   total day calls          3333 non-null   int64
 9   Predicted total day calls  3333 non-null   int64
 10  total day charge         3333 non-null   float64
 11  total eve minutes        3333 non-null   float64
 12  total eve calls          3333 non-null   int64
 13  total eve charge         3333 non-null   float64
 14  total night minutes      3333 non-null   float64
 15  total night calls        3333 non-null   int64
 16  total night charge       3333 non-null   float64
 17  total intl minutes       3333 non-null   float64
 18  total intl calls         3333 non-null   int64
 19  total intl charge        3333 non-null   float64
 20  customer service calls   3333 non-null   int64
 21  churn                    3333 non-null   bool
dtypes: bool(1), float64(8), int64(9), object(4)
memory usage: 550.2+ KB
```

	account length	area code	number vmail messages	total day minutes	total day calls	Predicted total day calls	
count	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	3333.000000	33
mean	101.064806	437.182418	8.099010	179.775098	100.435644	100.374737	
std	39.822106	42.371290	13.688365	54.467389	20.069084	20.141806	
min	1.000000	408.000000	0.000000	0.000000	0.000000	0.000000	
25%	74.000000	408.000000	0.000000	143.700000	87.000000	87.000000	
50%	101.000000	415.000000	0.000000	179.400000	101.000000	101.000000	
75%	127.000000	510.000000	20.000000	216.400000	114.000000	114.000000	
max	243.000000	510.000000	51.000000	350.800000	165.000000	165.000000	

Step 4: Data Cleaning and Preparation

Before performing any analysis, it is important to clean and prepare the data. This includes handling missing values, removing duplicates, and ensuring the data types are appropriate for analysis. This step is identical to what was already demonstrated in Deliverable 1.

Handling Missing Values:

Identify and handle any missing values in the DataFrame.

```
# Check for missing values
missing_values = df.isnull().sum()
print("Missing values in each column:")
print(missing_values)

# Example: Fill missing values with the mean of the column
#df_filled = df.fillna(df.mean())
```

```
Missing values in each column:
state                        0
account length               0
area code                    0
phone number                 0
international plan           0
voice mail plan              0
number vmail messages        0
total day minutes            0
total day calls              0
Predicted total day calls    0
total day charge             0
total eve minutes            0
total eve calls              0
total eve charge             0
total night minutes          0
total night calls            0
total night charge           0
total intl minutes           0
total intl calls             0
```

```
total intl charge           0
customer service calls      0
churn                       0
dtype: int64
```

Removing Duplicate Rows:

Check for and remove any duplicate rows to ensure data integrity.

```
# Check for duplicate rows
duplicates = df.duplicated()
if duplicates.any():
    print("Duplicate rows found and will be removed.")
    df = df.drop_duplicates()
```

Ensuring Consistent Data Types:

Verify and correct the data types of columns as needed.

```
# Display data types of each column
print("Data types of each column:")
print(df.dtypes)

# Example: Convert a column to a different data type if needed
# df['SomeColumn'] = df['SomeColumn'].astype('desired_dtype')
```

```
Data types of each column:
state                      object
account length              int64
area code                   int64
phone number               object
international plan         object
voice mail plan            object
number vmail messages       int64
total day minutes         float64
total day calls             int64
Predicted total day calls   int64
total day charge          float64
total eve minutes         float64
total eve calls             int64
total eve charge          float64
total night minutes       float64
total night calls           int64
total night charge        float64
total intl minutes        float64
total intl calls            int64
total intl charge         float64
customer service calls      int64
churn                        bool
dtype: object
```

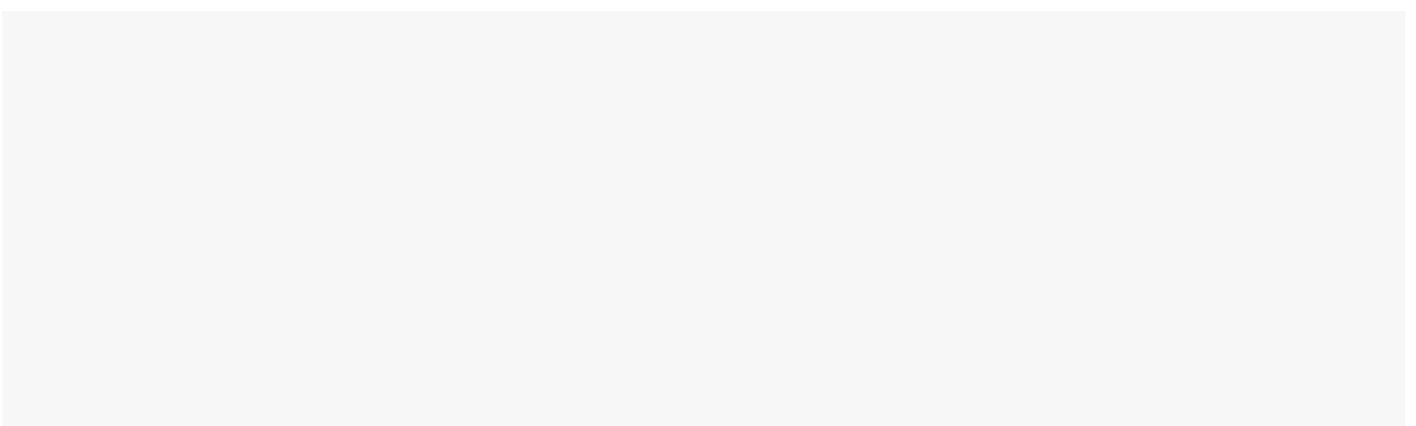
Calculating commonly used statistics in data analysis using Pandas and NumPy

Calculating commonly used statistics helps in understanding the distribution and central tendency of the data. These statistics include:

- **Measures of central tendency**: mean, median
- **Measures of spread**: standard deviation, minimum, maximum
- **Distribution percentiles**: quantiles

Below are detailed examples of how to calculate these statistics using Pandas and NumPy.

Calculating and displaying commonly used statistics using Pandas



```
# Mean: The average of the data
mean_total_day_charge = df['total day charge'].mean()
print(f"Mean Total Day Charge: {mean_total_day_charge}")

# Median: The middle value when the data is ordered
median_total_day_charge = df['total day charge'].median()
print(f"Median Total Day Charge: {median_total_day_charge}")

# Standard Deviation: The spread of the data around the mean
std_total_day_charge = df['total day charge'].std()
print(f"Standard Deviation of Total Day Charge: {std_total_day_charge}")

# Minimum: The lowest value in the column
min_total_day_charge = df['total day charge'].min()
print(f"Minimum Total Day Charge: {min_total_day_charge}")

# Maximum: The highest value in the column
max_total_day_charge = df['total day charge'].max()
print(f"Maximum Total Day Charge: {max_total_day_charge}")

# Quantiles: Values that divide the data into intervals of equal probability
quantiles_total_day_charge = df['total day charge'].quantile([0.25, 0.5, 0.75])
print(f"Quantiles of Total Day Charge:\n{quantiles_total_day_charge}")
```

```
Mean Total Day Charge: 30.562307230723075
Median Total Day Charge: 30.5
Standard Deviation of Total Day Charge: 9.2594345539305
Minimum Total Day Charge: 0.0
Maximum Total Day Charge: 59.64
Quantiles of Total Day Charge:
0.25    24.43
0.50    30.50
0.75    36.79
Name: total day charge, dtype: float64
```

Calculating and displaying commonly used statistics using NumPy

```
# Mean: The average value of the column
mean_total_day_charge = np.mean(df['total day charge'])
print(f"Mean Total Day Charge: {mean_total_day_charge}")

# Median: The middle value when the data is ordered
median_total_day_charge = np.median(df['total day charge'])
print(f"Median Total Day Charge: {median_total_day_charge}")

# Standard Deviation: The spread of the data around the mean
std_total_day_charge = np.std(df['total day charge'])
print(f"Standard Deviation of Total Day Charge: {std_total_day_charge}")

# Minimum: The lowest value in the column
min_total_day_charge = np.min(df['total day charge'])
print(f"Minimum Total Day Charge: {min_total_day_charge}")

# Maximum: The highest value in the column
max_total_day_charge = np.max(df['total day charge'])
print(f"Maximum Total Day Charge: {max_total_day_charge}")

# Quantiles: Values that divide the data into intervals of equal probability
quantiles_total_day_charge = np.quantile(df['total day charge'], [0.25, 0.5, 0.75])
print(f"Quantiles of Total Day Charge:\n{quantiles_total_day_charge}")
```

```
Mean Total Day Charge: 30.562307230723075
Median Total Day Charge: 30.5
Standard Deviation of Total Day Charge: 9.258045395636893
Minimum Total Day Charge: 0.0
Maximum Total Day Charge: 59.64
Quantiles of Total Day Charge:
[24.43 30.5  36.79]
```

Interpretation of Statistics in the Context of Analysis:

- **Mean:** The average value of the dataset. For example, the mean monthly charges provide an idea of the central value around which the monthly charges are distributed. This helps in understanding the typical amount charged to customers.
- **Median:** The middle value when the data is ordered. The median monthly charges help understand the central tendency, especially when there are outliers. If the median is significantly different from the mean, it indicates the presence of outliers or skewed data.
- **Standard Deviation:** Measures the spread of the data around the mean. A higher standard deviation indicates more variability in monthly charges, which can suggest diverse billing amounts among customers.

- **Minimum and Maximum:** The lowest and highest values in the dataset. These values show the range of monthly charges, helping to understand the spectrum of charges applied to customers.
- **Quantiles:** These divide the data into intervals of equal probability. The 25th, 50th (median), and 75th quantiles provide insights into the distribution of monthly charges and help identify the proportion of data below certain thresholds. For instance, knowing the 25th percentile (Q1) can help in understanding the charges below which 25% of the data lies.

By analyzing these statistics, we can gain valuable insights into customer charges. This helps in identifying patterns, anomalies, and potential areas for improving customer retention and reducing churn. For example, if a large proportion of customers are charged significantly less or more than the mean, targeted strategies can be developed to address these segments and improve overall customer satisfaction.

Section 2 - Deliverable 3

Creating Plots for Numerical Features: (3 Marks)

- How do you create some plots for each pair of numerical features in the input dataframe?
- Provide examples of data visualization techniques commonly used in exploratory data analysis and demonstrate how to create them using libraries like Matplotlib or Seaborn.
- How would you decide which type of plot is most appropriate for visualizing the relationship between two numerical features?
- Could you demonstrate how to create pairwise relationship plots for numerical features in a dataset, such as scatter plots, pair plots, and correlation matrices, using Python libraries?

Creating Plots for Numerical Features in a DataFrame

Data visualization is a crucial step in exploratory data analysis (EDA). It aids in understanding patterns, trends, and correlations within the dataset. Using libraries such as Matplotlib and Seaborn, various plots can be created to analyze numerical features effectively.

Step 1: Importing Necessary Libraries

To begin with data visualization, the necessary libraries must be imported. Matplotlib is used for basic plotting, while Seaborn is employed for more advanced visualizations.

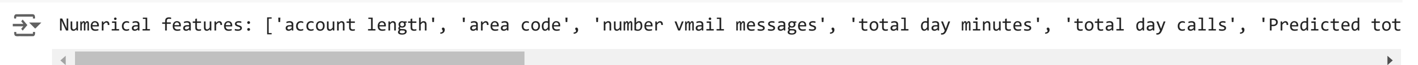
```
# Import necessary libraries
import matplotlib.pyplot as plt
import seaborn as sns
# import Pandas as pd

# Load dataset
# df = pd.read_csv('telecom_dataset.csv')
```

Step 2: Identifying Numerical Features

Before creating plots, it is essential to identify the numerical features in the dataset. This ensures the correct columns are selected for visualization. Typically, numerical features include continuous data points such as charges, minutes, and calls, which are crucial for understanding customer behavior.

```
# Displaying the numerical columns in the dataframe
numerical_features = df.select_dtypes(include=['float64', 'int64']).columns
print(f"Numerical features: {numerical_features.tolist()}")
```

```
Numerical features: ['account length', 'area code', 'number vmail messages', 'total day minutes', 'total day calls', 'Predicted tot
```

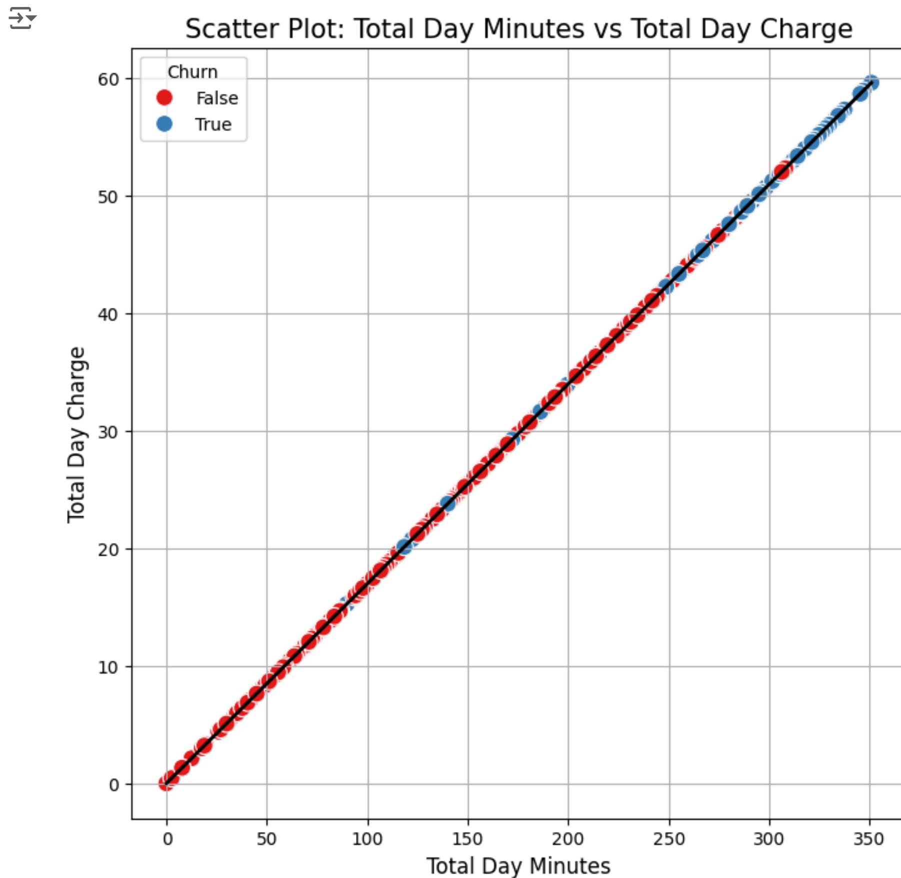


Step 3: Creating Scatter Plots

Scatter plots are effective for visualizing the relationship between two numerical features. They help in identifying trends, patterns, and outliers.

- **Scatter Plot Example:** The relationship between "total day minutes" and "total day charge" can be visualized using a scatter plot. This helps in understanding how the amount of minutes used during the day correlates with the charges. In the example below, the scatter plot provides a visual representation of how "total day minutes" impacts "total day charge". Linear or non-linear trends, as well as anomalies or clusters, can be observed.

```
# Creating a scatter plot (enhanced for better visualization)
plt.figure(figsize=(8, 8))
sns.scatterplot(x='total day minutes', y='total day charge', data=df, hue='churn', palette='Set1', alpha=1, edgecolor='w', s=100)
sns.regplot(x='total day minutes', y='total day charge', data=df, scatter=False, color='black', line_kws={"linewidth":2})
plt.title('Scatter Plot: Total Day Minutes vs Total Day Charge', fontsize=15)
plt.xlabel('Total Day Minutes', fontsize=12)
plt.ylabel('Total Day Charge', fontsize=12)
plt.legend(title='Churn', loc='upper left')
plt.grid(True)
plt.show()
```

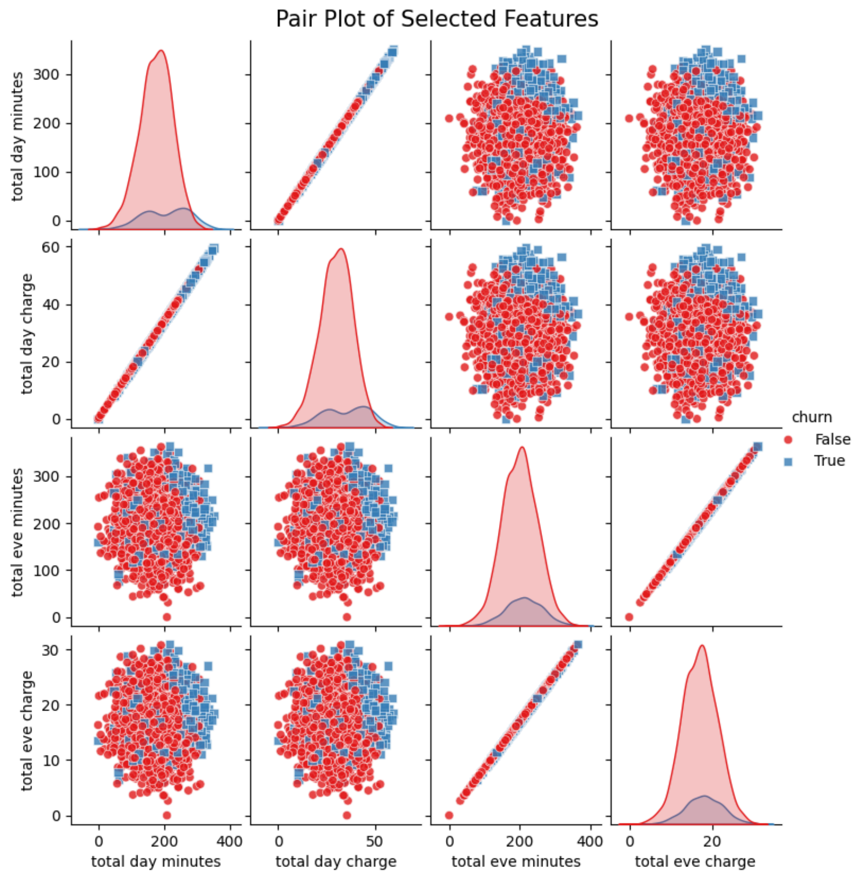


Step 4: Creating Pair Plots

Pair plots (or scatterplot matrices) are useful for examining pairwise relationships between multiple numerical features simultaneously. Seaborn's pairplot function allows for the plotting of multiple scatter plots in a matrix format.

- **Pair Plot Example**: A pair plot can be created for a subset of numerical features to keep the visualization clear and manageable. In the following example, the pair plot allows for the visualization of relationships between "total day minutes", "total day charge", "total eve minutes", and "total eve charge" simultaneously. It helps in understanding the interactions between these features.

```
# Creating a pair plot (enhanced for better visualization)
subset_features = ['total day minutes', 'total day charge', 'total eve minutes', 'total eve charge', 'churn']
pair_plot = sns.pairplot(df[subset_features], hue='churn', palette='Set1', markers=["o", "s"], plot_kws={'alpha': 0.8})
pair_plot.fig.suptitle('Pair Plot of Selected Features', y=1.02, fontsize=15)
pair_plot.fig.set_size_inches(8, 8)
plt.show()
```

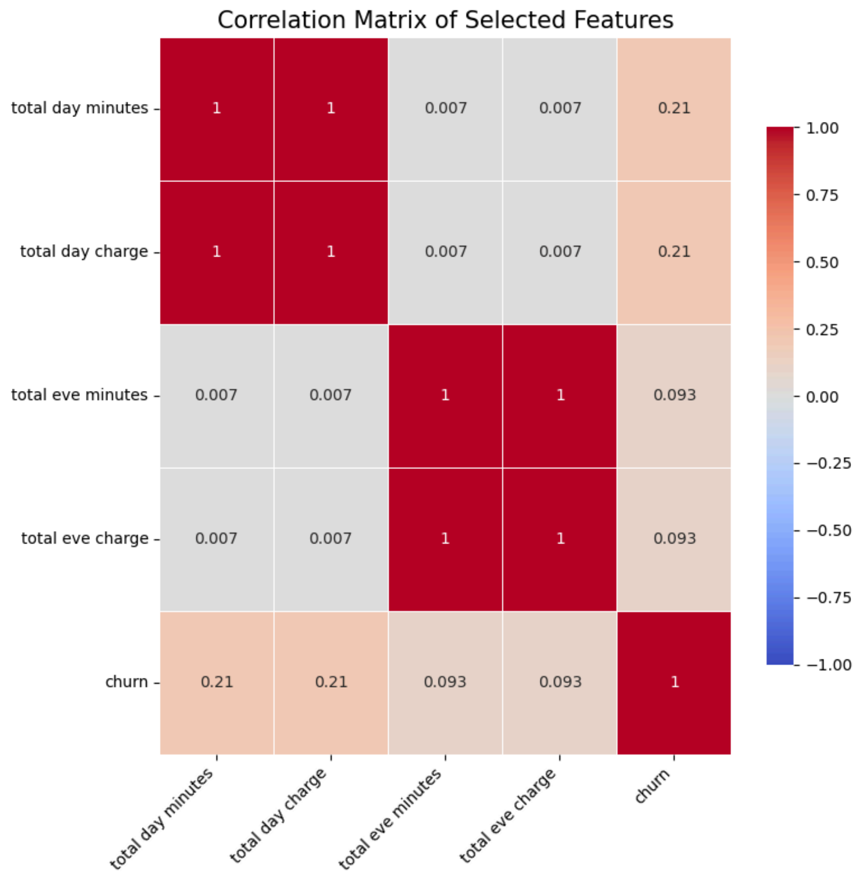


Step 5: Creating Correlation Matrices

Correlation matrices are useful for quantifying the linear relationships between numerical features. The correlation coefficient ranges from -1 to 1, indicating the strength and direction of the relationship.

- **Correlation Matrix Example**: A heatmap can be created to visualize the correlation matrix of numerical features. This helps in identifying which features are strongly correlated. In the example below, the heatmap provides a visual representation of the correlation coefficients between selected numerical features. It helps in identifying pairs of features with strong positive or negative correlations.

```
# Creating a correlation matrix
correlation_matrix = df[subset_features].corr()

# Creating a heatmap for the correlation matrix (enhanced for better visualization)
plt.figure(figsize=(8, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', vmin=-1, vmax=1, linewidths=0.5, linecolor='white', cbar_kws={"shrink": 0.75
plt.title('Correlation Matrix of Selected Features', fontsize=15)
plt.xticks(rotation=45, ha='right')
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()
```



Deciding the Most Appropriate Plot for Visualizing Relationships

To determine the most appropriate type of plot for visualizing the relationship between two numerical features, several factors need to be considered.

1. **Objective of Analysis:**
    - **Understanding Correlation and Trend:** If the goal is to understand the correlation and trend between two numerical features, a scatter plot is often the best choice. Scatter plots allow for easy identification of linear or non-linear relationships and patterns between variables.
    - **Comprehensive Pairwise Relationships:** For a broader view of relationships between multiple numerical features, a pair plot is useful. Pair plots provide a matrix of scatter plots, making it easier to spot relationships across multiple feature pairs.
    - **Summarizing Relationships**: When the aim is to summarize the strength and direction of relationships across multiple pairs of numerical features, a correlation matrix visualized with a heatmap is appropriate. This provides a clear, at-a-glance view of how features correlate with each other.
2. **Data Characteristics:**
    - **Distribution of Data**: If the numerical features have a large range or contain many outliers, scatter plots can highlight these aspects effectively.
    - **Number of Features:** For datasets with many numerical features, pair plots and correlation matrices can provide a more comprehensive overview compared to individual scatter plots.
    - **Data Density**: For dense datasets, scatter plots with transparency (alpha) or hexbin plots can help in visualizing data points without overlapping excessively.
3. **Ease of Interpretation:**
    - **Scatter Plots:** These are straightforward and easy to interpret, especially when enhanced with trend lines and color coding.
    - **Pair Plots:** These offer a detailed view but can become cluttered with too many features. Suitable for exploratory data analysis where multiple relationships are of interest.
    - **Correlation Matrices**: These are ideal for summarizing the relationships in a compact and easy-to-read format. Annotations and color gradients enhance interpretability.

Considerations about the examples provided above

- **Scatter Plot Example:**
  - **Use Case:** Visualizing the relationship between "total day minutes" and "total day charge".
  - **Reason**: A scatter plot can reveal whether there is a linear relationship between the amount of time spent on calls and the charges incurred.
- **Pair Plot Example:**
  - **Use Case:** Exploring relationships between "total day minutes", "total day charge", "total eve minutes", and "total eve charge".
  - **Reason**: A pair plot provides a matrix of scatter plots that shows how these features interact with each other, which is useful for initial exploratory data analysis.
- **Correlation Matrix Example:**
  - **Use Case:** Summarizing the relationships between multiple numerical features in the dataset.
  - **Reason:** A correlation matrix heatmap provides a visual summary of correlations, helping to quickly identify which features are strongly correlated.

By considering the objective of the analysis, data characteristics, ease of interpretation, and potential visualization enhancements, the most appropriate type of plot can be selected to effectively visualize the relationship between two numerical features.

End of Section 2

Nunzio Emanuele Sgroi -

Section 3 - Analysis with MongoDB

Learning Outcome: Analyze operational data on MongoDB

Student Name: Nunzio Emanuele Sgroi

Student ID: [redacted]

Description:

Data Modelling

The process of designing and implementing the database are guided based on the following 4 steps:

1. **Gather requirements.**
2. **Understand relationships between entities.**
3. **Identify the data structure.**
4. **Apply design patterns.**

After reading the instructions, the following tasks in the database need to be designed. The list of tasks and the assumptions are explained in Table 1. The data structure and entity relationships (steps 2 and 3) should be addressed and built based on the most important functionalities identified for the operation of the business.

Tasks:

Task	Subtask	Sub Subtask
1. Designing NoSQL Schema for Company	1.1 Designing collections in Schema	customers
		pastOrders
		products
		ratings
		suppliers
		dailyInventoryRecord
		partners
		partnerHistory
2. Implementation on MongoDB	2.1 Data Insertion	customers
		pastOrders
		products
		ratings
		suppliers
		dailyInventoryRecord
		partners
		partnerHistory
	2.1 Querying	Query (1-20)

Based on the instructions, the focus ofthe business isto deliver all products assoon as possible.

Section 3 - Deliverable 1

Implementation on MongoDB Atlas (Data Insertion): (2 Marks)

- How to do implementation on MongoDB atlas (Data Insertion)?
- Provide two examples of inserting data into a MongoDB Atlas database.

Setting Up the Database on MongoDB Atlas

Before performing data insertion into MongoDB Atlas, it is essential to set up the database. The process involves creating an account, setting up a cluster, configuring access, and obtaining a connection string. Below are the detailed steps:

1. First, an account on MongoDB Atlas must be created. This can be done by visiting the MongoDB Atlas website (https://www.mongodb.com/cloud/atlas) , selecting "Try Free", and proceeding with creating an account with email and password or google.
2. Once logged in, the next step involves building a new cluster. This can be initiated by selecting "Build a Cluster". It is important to choose a preferred cloud provider and region; it is noteworthy that the free tier provides a limited selection of regions. By clicking on "Create Cluster", the process to create the cluster will commence, which might require a few minutes to complete. In my case I selected Google Cloud as cloud provider and selected Belgium as region.
3. The next step is about configuring database access. This is done by navigating to the "Database Access" section located on the left-hand side of the MongoDB Atlas dashboard, then "Add New Database User" and input a username and password for the database user. It is advisable to assign the role as "Atlas Admin" or customize the privileges according to specific needs, followed by clicking "Add User".

4. Following the configuration of database access, the next step is to set up network access. This involves navigating to the "Network Access" section and clicking on "Add IP Address". It is possible to either select "Allow Access from Anywhere" (0.0.0.0/0) or manually add a specific IP address.
5. The final step in the setup process is to obtain the connection string. To achieve this, navigate back to the "Clusters" view and click on the "Connect" button for the cluster. Follow the prompts by selecting "Drivers" to connect to the application. Choose "Python" as the driver and the latest version. MongoDB Atlas will then display a connection string. In my case the connection string looks as follow:

```
mongodb+srv://emanuelsgroi95:hwDvNwEe6e2WiQ28@assignment.9utro4x.mongodb.net/?retryWrites=true&w=majority&appName=Assignment
```

Please note that the password is intentionally included for demonstration purposes specific to this assignment.

With the completion of these steps, the database setup on MongoDB Atlas is finalized, and now it will be possible to perform data insertion and other operations.

Implementation on MongoDB Atlas

After successfully setted up the database, the next step is to connect MongoDB Atlas with the application, in this case Google Colab (using python). The steps include installing necessary libraries, connecting to MongoDB Atlas, and verifying the connection.

Step 1: Install Necessary Libraries

To interact with MongoDB Atlas from Python, the `pymongo` library must be installed.

```
# Installing pymongo library
!pip install pymongo
```

```
Collecting pymongo
  Downloading pymongo-4.7.2-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (670 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 670.0/670.0 kB 11.8 MB/s eta 0:00:00
Collecting dnspython<3.0.0,>=1.16.0 (from pymongo)
  Downloading dnspython-2.6.1-py3-none-any.whl (307 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 307.7/307.7 kB 32.7 MB/s eta 0:00:00
Installing collected packages: dnspython, pymongo
Successfully installed dnspython-2.6.1 pymongo-4.7.2
```

Step 2: Connect to MongoDB Atlas

Establish a connection to MongoDB Atlas using the connection string obtained during the setup process.

```
from pymongo import MongoClient

# MongoDB Atlas details
connection_string = "mongodb+srv://emanuelsgroi95:hwDvNwEe6e2WiQ28@assignment.9utro4x.mongodb.net/?retryWrites=true&w=majority&appName=
client = MongoClient(connection_string)
print("Connected to MongoDB Atlas")
```

```
Connected to MongoDB Atlas
```

Step 3: Verify the Connection

Verify the connection by listing the databases in the MongoDB Atlas cluster.

```
# Listing the databases in the cluster
databases = client.list_database_names()
print("Databases:", databases)
```

```
Databases: ['Test', 'sample_mflix', 'admin', 'local']
```

Methods of Data Insertion into MongoDB Atlas

Data can be inserted into a MongoDB Atlas database using various methods. The two most common approaches are programmatic insertion through code and manual insertion via the MongoDB Atlas interface. Programmatic insertion involves using `pymongo` library to automate the process of inserting documents into a MongoDB collection. This method is suitable for large-scale data operations and automated workflows. Manual insertion, on the other hand, is performed directly through the MongoDB Atlas web interface. This method is useful for quick, small-scale data entries and for users who may not be familiar with programming.

Example 1: Inserting Data Using `pymongo`

This example demonstrates how to insert data into a MongoDB Atlas database using Python and the `pymongo` library. For this particular example, all the datasets in JSON format provided on Blackboard will be inserted into MongoDB Atlas. Therefore, additional libraries such as `json` and `glob` will be used. The dataset files include the following:

- `customers_amazone.json`

- `dailyInventoryRecord.json`
- `partnerHistory.json`
- `partners.json`
- `partners_old.json`
- `pastOrders.json`
- `products.json`
- `ratings.json`
- `suppliers.json`

All these files are already uploaded to Google Colab.

Fixing the JSON Files

The JSON files provided might contain MongoDB Extended JSON format, which includes special fields like $oidand$date. These fields need to be converted to standard JSON format that MongoDB can accept.

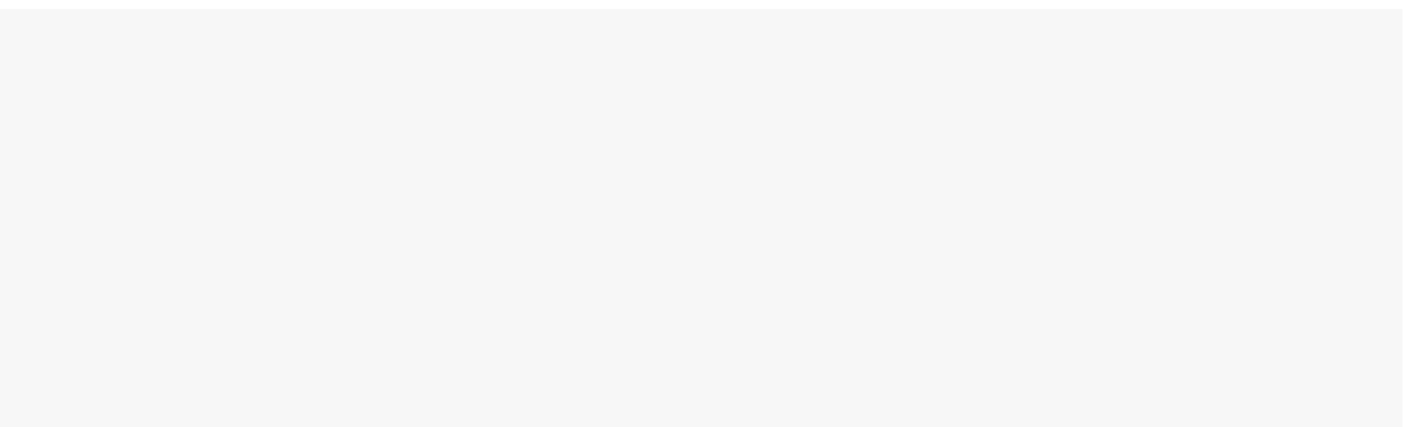
Example of Original File:

```
{
  "_id": { "$oid": "63b852b6a7fc6363d52e5cbe" },
  "partner_id": "PA1",
  "start_date": { "$date": { "$numberLong": "1672617600000" } },
  "end_date": { "$date": { "$numberLong": "1672703999000" } },
  "orders": [
    {
      "order_id": "20230102100909C1",
      "timestamp": { "$date": { "$numberLong": "1672654149000" } },
      "order_details": [
        { "product_id": "FP1", "quantity": 1 },
        { "product_id": "FP2", "quantity": 2 }
      ]
    }
  ]
}
```

Example of Corrected File:

```
{
  "_id": "63b852b6a7fc6363d52e5cbe",
  "partner_id": "PA1",
  "start_date": "2023-01-01T00:00:00Z",
  "end_date": "2023-01-02T23:59:59Z",
  "orders": [
    {
      "order_id": "20230102100909C1",
      "timestamp": "2023-01-02T10:02:29Z",
      "order_details": [
        { "product_id": "FP1", "quantity": 1 },
        { "product_id": "FP2", "quantity": 2 }
      ]
    }
  ]
}
```

The following code converts the MongoDB Extended JSON format to standard JSON format:



```
# Import the necessary libraries
from pymongo import MongoClient
from bson import ObjectId
from datetime import datetime
import json
import glob

# Function to convert MongoDB Extended JSON to standard JSON
def convert_extended_json(doc):
    if isinstance(doc, dict):
        if '$oid' in doc:
            return str(ObjectId(doc['$oid']))
        if '$date' in doc and '$numberLong' in doc['$date']:
            return datetime.utcfromtimestamp(int(doc['$date']['$numberLong']) / 1000.0).isoformat() + 'Z'
        return {k: convert_extended_json(v) for k, v in doc.items()}
    elif isinstance(doc, list):
        return [convert_extended_json(item) for item in doc]
    else:
        return doc

# List of JSON files to be fixed
json_files = [
    'customers_amazone.json',
    'dailyInventoryRecord.json',
    'partnerHistory.json',
    'partners.json',
    'partners_old.json',
    'pastOrders.json',
    'products.json',
    'ratings.json',
    'suppliers.json'
]

# Fixing the JSON files and saving them
for file_name in json_files:
    with open(file_name) as f:
        data = json.load(f)
    fixed_data = convert_extended_json(data)
    with open(f'fixed_{file_name}', 'w') as f:
        json.dump(fixed_data, f)
    print(f"Fixed {file_name} and saved as fixed_{file_name}")
```

```
Fixed customers_amazone.json and saved as fixed_customers_amazone.json
Fixed dailyInventoryRecord.json and saved as fixed_dailyInventoryRecord.json
Fixed partnerHistory.json and saved as fixed_partnerHistory.json
Fixed partners.json and saved as fixed_partners.json
Fixed partners_old.json and saved as fixed_partners_old.json
Fixed pastOrders.json and saved as fixed_pastOrders.json
Fixed products.json and saved as fixed_products.json
Fixed ratings.json and saved as fixed_ratings.json
Fixed suppliers.json and saved as fixed_suppliers.json
```

Inserting the Fixed JSON Files into MongoDB Atlas

With the JSON files fixed, they can now be inserted into MongoDB Atlas.

```
# Select a database
db = client['sample_mflix']

# Function to load JSON data from a file and insert into the specified collection
def insert_data(file_name, collection_name):
    with open(file_name) as f:
        data = json.load(f)
    collection = db[collection_name]
    if isinstance(data, list):
        result = collection.insert_many(data)
        print(f"Inserted {len(result.inserted_ids)} documents into {collection_name}")
    else:
        result = collection.insert_one(data)
        print(f"Inserted document ID: {result.inserted_id} into {collection_name}")

# Mapping of fixed JSON files to their respective collections
file_to_collection = {
    'fixed_customers_amazone.json': 'customers_amazone',
    'fixed_dailyInventoryRecord.json': 'dailyInventoryRecord',
    'fixed_partnerHistory.json': 'partnerHistory',
    'fixed_partners.json': 'partners',
    'fixed_partners_old.json': 'partners_old',
    'fixed_pastOrders.json': 'pastOrders',
    'fixed_products.json': 'products',
    'fixed_ratings.json': 'ratings',
    'fixed_suppliers.json': 'suppliers'
}

# Insert data from each fixed JSON file into the corresponding collection
for file, collection in file_to_collection.items():
    insert_data(file, collection)
```

```
Inserted 20 documents into customers_amazone
Inserted 261 documents into dailyInventoryRecord
Inserted 5 documents into partnerHistory
Inserted 5 documents into partners
Inserted 5 documents into partners_old
Inserted 275 documents into pastOrders
Inserted 55 documents into products
Inserted 275 documents into ratings
Inserted 9 documents into suppliers
```

The collections have been created, and they contain the data from the JSON files. These collections can be viewed inside MongoDB Atlas by navigating to "Database," clicking on the name of the previously created database, and then going to "Collections." The image below demonstrates that all the new collections established in the code have been successfully created, each representing the corresponding dataset and containing the data from the JSON files.

EMANUELE'S ORG - 2024-05-28 > PROJECT 0 > DATABASES

Assignment

Overview | Real Time | Metrics | Collections | Atlas Search | Performance Advisor | Online Archive

DATABASES: 1 COLLECTIONS: 9

+ Create Database

Search Namespaces

sample_mflix

- customers_amazone
- dailyInventoryRecord
- partnerHistory
- partners
- partners_old
- pastOrders
- products
- ratings
- suppliers

sample_mflix.customers_amazone

STORAGE SIZE: 24KB LOGICAL DATA SIZE: 19.85KB TOTAL DOCUMENTS: 20 INDEXES TOTAL SIZE: 20KB

Find | Indexes | Schema Anti-Patterns 0 | Aggregation | Search Indexes

Generate queries from natural language in Compass

Filter Type a query: { field: 'value' }

QUERY RESULTS: 1-20 OF 20

```
_id: "C1"
Customer : "Gunner Ferrell"
Gender : "M"
Age : 51
phone_number : 443454155475
addresses : Array (1)
current_orders : Array (4)
recommended_products : Array (2)
```

```
_id: "C2"
Customer : "Lillie Costa"
Gender : "F"
Age : 30
phone_number : 447137031760
addresses : Array (1)
current_orders : Array (4)
recommended_products : Array (2)
```

Example 2: Inserting Data Manually via MongoDB Atlas Interface

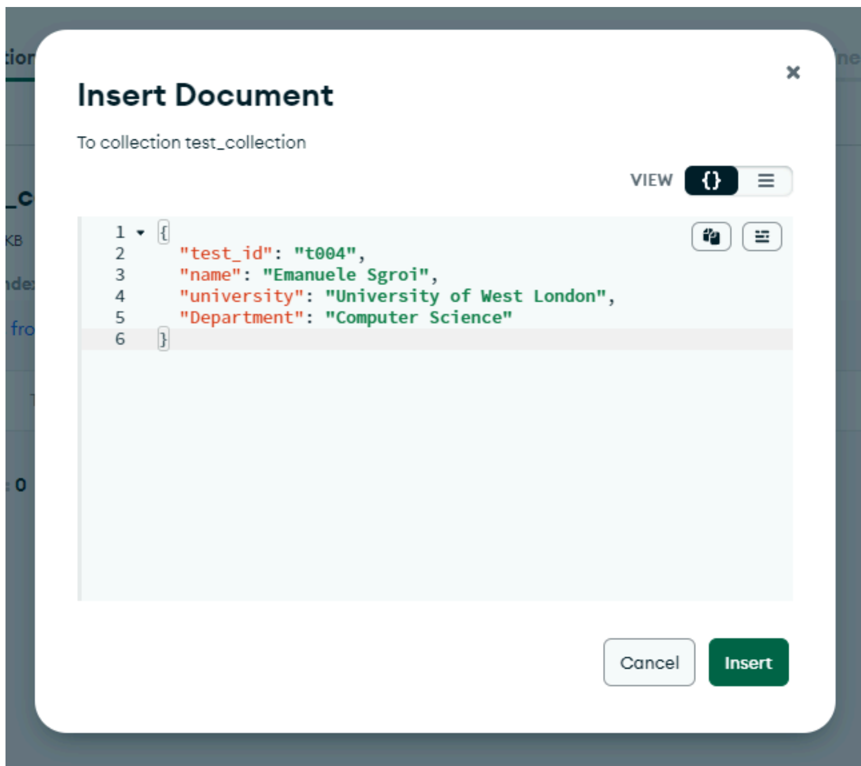
This example demonstrates how to manually insert data into a MongoDB Atlas database through the MongoDB Atlas web interface.

Step-by-Step Process:

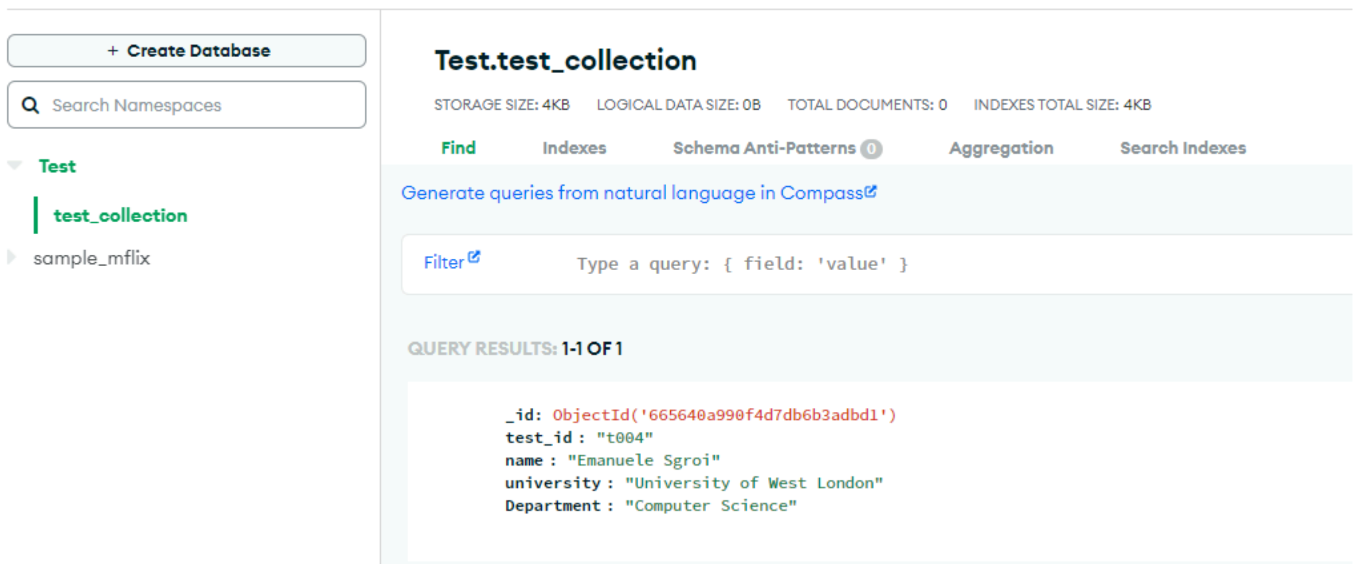
1. **Navigate to the MongoDB Atlas Cluster:** Begin by logging in to MongoDB Atlas and navigating to the cluster where the data will be inserted.
2. **Access the Collections:** Click on the "Collections" button under the desired cluster to open the collections view.
3. **Create a New Database or New Collection:** If the target database or collection does not exist, it is possible to create them manually:
    - To create a database click on "Create Database" then give a database name and collection name
    - To create a collection on a existing database, select the database and click on "Create Collection".

    For this example, I created a new database called "Test" with a collection named "test_collection".
4. **Insert a Document:** Select the target collection and click on the "Insert Document" button. A JSON editor will appear. Enter the document data in JSON format. For instance:

```
{
"test_id": "t004",
"name": "Emanuele Sgroi",
"university": "University of West London",
"Department": "Computer Science"
}
```



5. **Verify the Insertion:** The newly inserted document should appear in the collection view, confirming the successful insertion.



Section 3 - Deliverable 2

Designing NoSQL Schema for Company: (5 Marks)

- How to design a NoSQL schema for a company?
- Provide examples of a NoSQL schema designed for a company.

How to design a NoSQL schema

Designing a NoSQL schema involves several crucial steps to ensure that the database structure is efficient, scalable, and tailored to the specific needs of the application. The following steps outline a methodical approach to designing a NoSQL schema:

1. Gather Requirements: To begin with, it is essential to identify the key entities and attributes required for the application. This involves a thorough understanding of the business requirements and the various types of queries that will be executed on the database. By gathering these requirements, one can ensure that the schema design aligns with the application's objectives and optimizes query performance.

2. Understand Relationships Between Entities: Next, it is important to determine how the entities within the database are related to each other. This step involves deciding whether to embed related data within a document or to reference it in a separate collection. Understanding these relationships helps in structuring the database in a way that supports efficient data retrieval and maintains data integrity.

3. Identify the Data Structure: Defining the structure of each entity is a critical step in schema design. This includes specifying the fields and data types for each entity. Additionally, it is important to consider denormalization for performance optimization, where necessary. Denormalization involves merging related data into a single document to reduce the number of queries needed to retrieve related information.

4. Apply Design Patterns: Finally, applying appropriate NoSQL design patterns is essential for creating an efficient schema. Common design patterns include embedding, referencing, and denormalization. These patterns should be selected based on the access patterns and requirements of the application. By using these design patterns, one can create a schema that is optimized for the specific needs of the application and ensures efficient data retrieval and storage.

Examples of a NoSQL schema designed for a company

The following example is based on the given data of Section 3 of this assessment. In NoSQL schema design, the schema can be represented either in a textual format (JSON) or in a visual format, similar to an Entity-Relationship diagram in SQL. The following textual format presents part of the schema design in JSON format, as specified for this assessment:

Customers Collection:

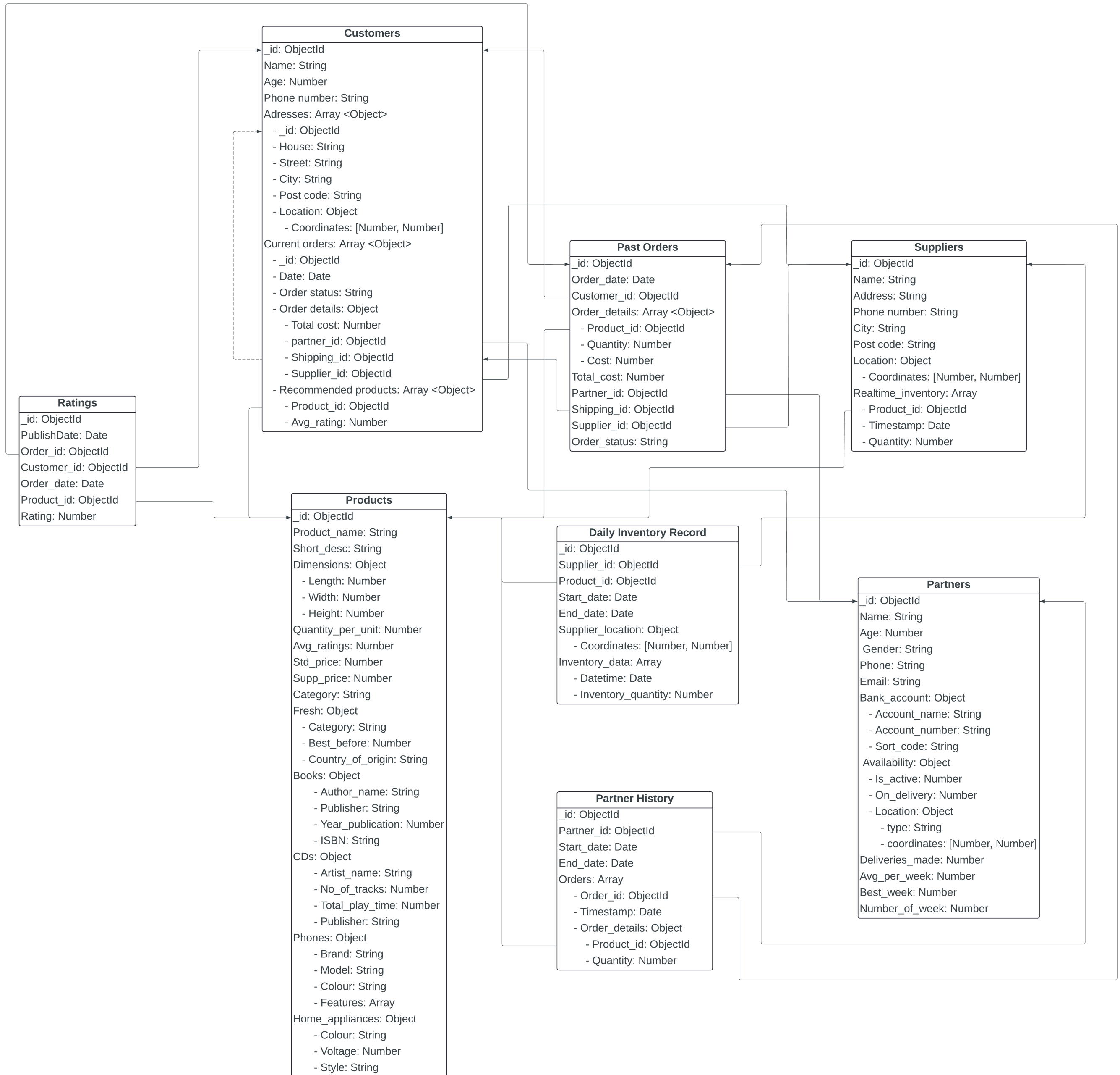
```
{
  "_id": "unique identifier",
  "name": "customer name",
  "gender": "gender",
  "age": "age",
  "phone number": "contact number",
  "addresses": [
    {
      "_id": "unique identifier",
      "house": "house name/number",
      "street": "street name/number",
      "city": "city",
      "post code": "postcode",
      "location": {
        "coordinates": ["longitude", "latitude"]
      }
    }
  ],
  "current orders": [
    {
      "_id": "unique identifier",
      "date": "order date",
      "order status": "status of the order",
      "order details": {
        "total cost": "total cost",
        "partner_id": "unique identifier of the partner",
        "shipping_id": "unique identifier of the shipping address",
        "supplier_id": "unique identifier of the supplier"
      },
      "recommended products": [
        {
          "product_id": "unique identifier of the product",
          "avg_rating": "average rating of the product"
        }
      ]
    }
  ]
}
```

Past Orders Collection:

```
{
  "_id": "unique identifier",
  "order_date": "order date",
  "customer_id": "unique identifier of the customer",
  "order_details": [
    {
      "product_id": "unique product identifier",
      "quantity": "quantity ordered",
      "cost": "unit cost"
    }
  ],
```

```
  "total_cost": "total cost of the order",
  "partner_id": "unique identifier of the partner",
  "shipping_id": "unique identifier of the shipping address",
  "supplier_id": "unique identifier of the supplier",
  "order_status": "status of the order"
}
```

Here is the visual representation of the schema designed for all the data provided:



While in SQL it is common to explicitly define primary keys (PK) and foreign keys (FK) to establish relationships between tables, in NoSQL databases, relationships are typically represented through embedding related documents within a single collection or referencing using unique identifiers. This approach allows for greater flexibility and scalability, which are key advantages of NoSQL databases.

Section 3 - Deliverable 3

Data Manipulation in MongoDB: (10 Marks)

- How to find, insert, delete, retrieve, and update data from a MongoDB database?
- Provide five examples of finding, inserting, deleting, retrieving, and updating data in MongoDB

CRUD operations

MongoDB, like other databases, allows CRUD operations: Create, Read, Update, and Delete. Understanding these operations is fundamental for managing data in MongoDB. Below, are provided five examples demonstrating how to perform each of these operations using Python and the pymongo library.

```
from pymongo import MongoClient
db = client["sample_mflix"]
```

Inserting Data

Inserting data into a MongoDB collection is done using the `insert_one` or `insert_many` methods. This allows for adding single or multiple documents to the collection.

```
# Insert a single document into the customers collection
customer = {
    "name": "Emanuele Sgroi",
    "gender": "Male",
    "age": 29,
    "phone_number": "123-456-7890",
    "addresses": [
        {
            "house": "123",
            "street": "Main St",
            "city": "Anytown",
            "post_code": "12345",
            "location": {
                "coordinates": [-73.935242, 40.730610]
            }
        }
    ],
    "current_orders": []
}

# Insert the document
db.customers_amazone.insert_one(customer)
```

```
InsertOneResult(ObjectId('6656983d19c292fea58b04d7'), acknowledged=True)
```

Finding Data

Finding data in a MongoDB collection is done using the find method, which supports `querying` documents with specific criteria.

```
# Find all customers named "Gunner Ferrell"
query = {"Customer": "Gunner Ferrell"}
results = db.customers_amazone.find(query)

# Print results
for customer in results:
    print(customer)
```

```
{'_id': 'C1', 'Customer': 'Gunner Ferrell', 'Gender': 'M', 'Age': 51, 'phone_number': {'$numberLong': '443454155475'}, 'addresses':
```

Updating Data

Updating data is done using the `update_one` or `update_many` methods, allowing for modifications to existing documents based on specified criteria.

```
# Update the phone number of the customer named "John Doe"
query = {"Customer": "Gunner Ferrell"}
new_values = {"$set": {"phone_number": "987-654-3210"}}

db.customers_amazone.update_one(query, new_values)
```

```
UpdateResult({'n': 1, 'electionId': ObjectId('7fffffff0000000000000012'), 'opTime': {'ts': Timestamp(1716951609, 4), 't': 18},
'nModified': 1, 'ok': 1.0, '$clusterTime': {'clusterTime': Timestamp(1716951609, 4), 'signature': {'hash':
b'>\xde\xeb\x97\xd8\xd5l\xf0\x1e\x1f\xec\xca\xfd\x05\x99\xc5\x1a\x04\x1c\xa0', 'keyId': 7320298377621536775}}, 'operationTime':
Timestamp(1716951609, 4), 'updatedExisting': True}, acknowledged=True)
```

Deleting Data

Deleting data from a MongoDB collection can be achieved using the `delete_one` or `delete_many` methods, which remove documents that match the given criteria.

```
# Delete the customer named "Gunner Ferrell"
query = {"Customer": "Gunner Ferrell"}

db.customers.delete_one(query)
```

```
DeleteResult({'n': 0, 'electionId': ObjectId('7fffffff0000000000000012'), 'opTime': {'ts': Timestamp(1716951644, 3), 't': 18}, 'ok': 1.0, '$clusterTime': {'clusterTime': Timestamp(1716951644, 3), 'signature': {'hash': b'\xa1\x8fL-\x98\xaf\x8b\xf1\xdc\xb3\xc2\x80\x10\xc8a\xd3\xe0\xd0c\x8b', 'keyId': 7320298377621536775}}, 'operationTime': Timestamp(1716951644, 3)}, acknowledged=True)
```

Retrieving Specific Fields

Retrieving specific fields from documents can be done using the `find` method with a projection to limit the returned fields.

```
# Retrieve only the product name and average rating for products with an average rating of at least 3.5
query = {"avg_ratings": {"$gte": 3.5}}
projection = {"name": 1, "avg_ratings": 1, "_id": 0}
results = db.products.find(query, projection)

# Print the results
for product in results:
    print(product)
```

```
{'name': 'Led Zepellin IV', 'avg_ratings': 4}
{'name': 'Hotel California', 'avg_ratings': 4}
{'name': 'Huawei P9', 'avg_ratings': 3.6}
{'name': 'Cold Grave', 'avg_ratings': 4.2}
{'name': 'Thinking, Fast and Slow', 'avg_ratings': 3.8}
{'name': 'Eufy by Anker, BoostIQ RoboVac 11S (Slim), Robot Vacuum Cleaner, Super-Thin', 'avg_ratings': 3.8}
{'name': 'EasyAcc Coffee Mug Warmer', 'avg_ratings': 4}
{'name': 'LED Starry Sky Projector', 'avg_ratings': 3.8}
{'name': 'Tart', 'avg_ratings': 3.6}
{'name': 'Beer', 'avg_ratings': 3.8}
{'name': 'Whiskey', 'avg_ratings': 3.8}
```

Section 3 - Deliverable 4

Creating Plotsfor Numerical Features: (3 Marks)

- How do you create plots for each pair of numerical features in a dataframe?
- Provide examples of creating three different plots for pairwise numerical features.

Creating plots for each pair of numerical features in a dataframe

Creating plots for numerical features is an essential step in data analysis as it helps to visualize the relationships, distributions, and patterns within the data. These visualizations enable a deeper understanding of the data, revealing insights that may not be immediately apparent from the raw data. Python libraries such as `Matplotlib` and `Seaborn` are widely used for creating these plots due to their flexibility and powerful visualization capabilities, as shown previously. Below are examples of creating three different types of plots for pairwise numerical features:

1. **Histogram plots**: Show the distribution of data.
2. **Box plots**: Provide a summary of the data's distribution and highlight outliers.
3. **violin plots**: Combine aspects of box plots and kernel density plots to give a comprehensive view of the data distribution.

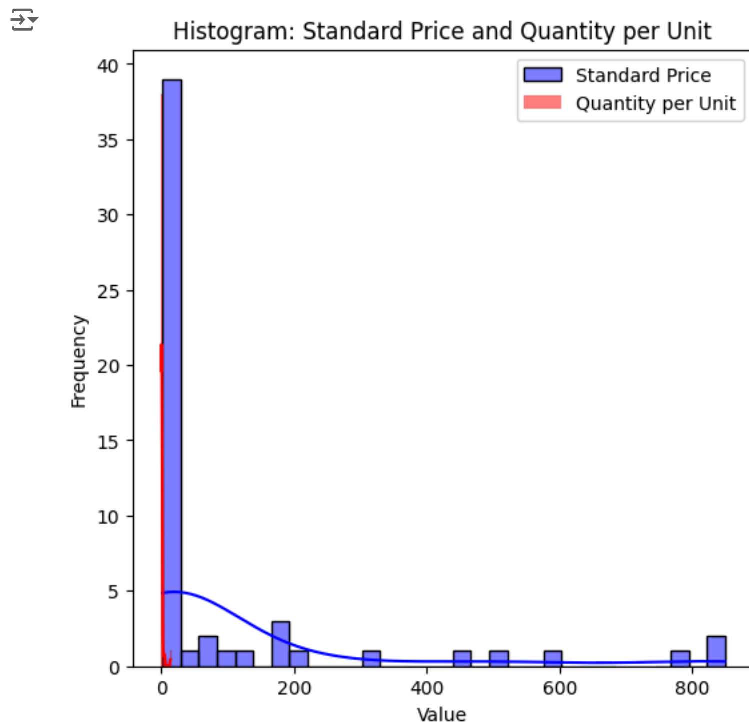
The examples below will use data that was previously uploaded into MongoDB Atlas.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
# from pymongo import MongoClient
```

Example 1: Histogram Plot

A histogram plot shows the distribution of a single numerical variable. Overlaying histograms can compare the distributions of two numerical features.

```
# Load the data from the products collection
products_data = list(db.products.find({}, {"_id": 0, "std_price": 1, "quantity_per_unit": 1}))

# Convert to DataFrame
df_products = pd.DataFrame(products_data)

# Histogram plot for std_price and quantity_per_unit
plt.figure(figsize=(6, 6))
sns.histplot(df_products['std_price'], kde=True, color='blue', label='Standard Price')
sns.histplot(df_products['quantity_per_unit'], kde=True, color='red', label='Quantity per Unit')
plt.title('Histogram: Standard Price and Quantity per Unit')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

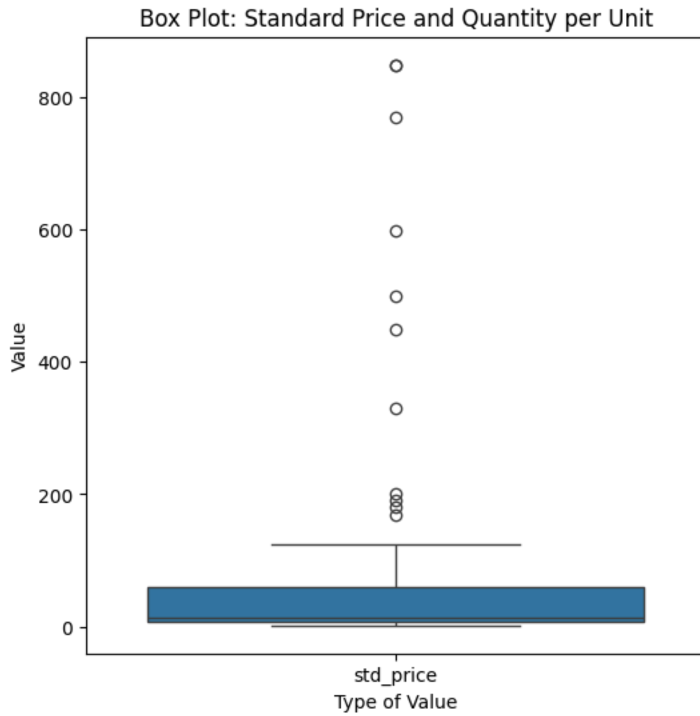


Example 2: Box Plot

A box plot displays the distribution of a numerical variable and can be used to compare distributions across different categories.

```
print("test")
# Box plot for std_price and quantity_per_unit
plt.figure(figsize=(6, 6))
sns.boxplot(data=df_products[['std_price', 'quantity_per_unit']])
plt.title('Box Plot: Standard Price and Quantity per Unit')
plt.xlabel('Type of Value')
plt.ylabel('Value')
plt.show()
```

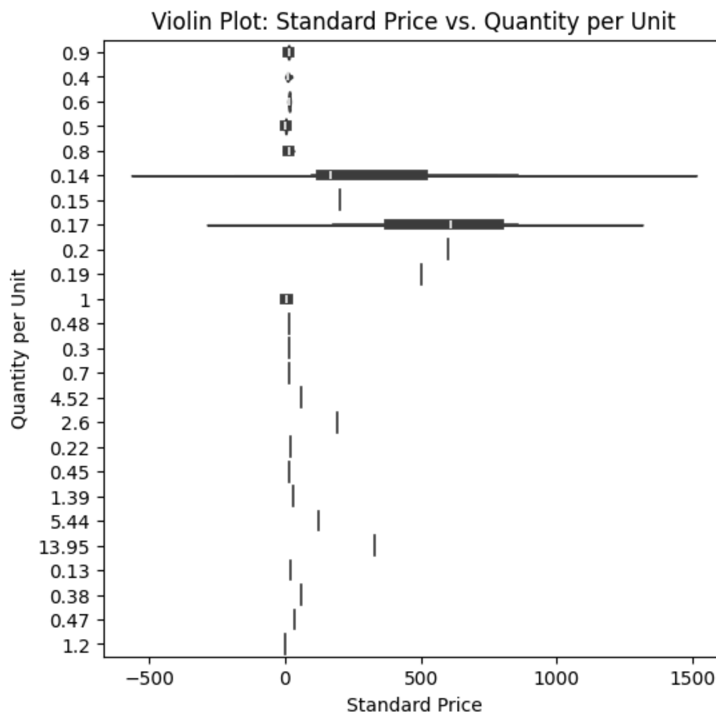
test



Example 3: Violin Plot

A violin plot combines aspects of box plots and kernel density plots to provide a comprehensive view of the data distribution.

```
# Violin plot for std_price vs. quantity_per_unit
plt.figure(figsize=(6, 6))
sns.violinplot(x='std_price', y='quantity_per_unit', data=df_products)
plt.title('Violin Plot: Standard Price vs. Quantity per Unit')
plt.xlabel('Standard Price')
plt.ylabel('Quantity per Unit')
plt.show()
```



These examples demonstrate how to create histogram plots, box plots, and violin plots to visualize pairwise numerical features in a dataframe. Each plot type provides a unique perspective and helps uncover different aspects of the data's distribution and relationships.

End of Section 3