SIG742 - Modern Data Science Assignment 2

Questions & Answers

Part 1 : Data Acquisition and Manipulation

Data Ingestion, Importing necessary Libraries in the solution,

```
[ ] from google.colab import drive
    drive.mount('/content/drive/', force_remount=True)

    Mounted at /content/drive/

[ ] # To install word clouduse below code
    #! pip install wordcloud
    #! pip install pmdarima
    #! pip install setuptools wheel
    #! pip install pystan==2.19.1.1
    #! pip install prophet
    #! pip install fbprophet
    #! pip install statsmodels

[ ] import os as os
    import pandas as pd
    import numpy as np
    import seaborn as sns
    import requests
    import zipfile
    import matplotlib.pyplot as plt
    from statsmodels.tsa.seasonal import seasonal_decompose
    from statsmodels.tsa.stattools import adfuller
    from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
    from statsmodels.tsa.arima_model import ARIMA
    from sklearn.metrics import mean_absolute_error
    from sklearn.ensemble import IsolationForest
    from wordcloud import WordCloud
    from pmdarima import auto_arima
    from datetime import datetime
    from statsmodels.tools.eval_measures import meanabs
    from statsmodels.tsa.holtwinters import ExponentialSmoothing
```

```
[ ] #-----Extract data from Zip File--------------#
    #with zipfile.ZipFile('/content/drive/My Drive/item_listing_category.zip', 'r') as zip_ref:
    #    zip_ref.extractall('/content/drive/My Drive/item_listing_category')

[ ] folder_path = '/content/drive/My Drive/item_listing_category' #----- Setting Folder Path----#
    all_dataframes = []
    for filename in os.listdir(folder_path):
        if filename.endswith('.csv'):
            file_path = os.path.join(folder_path, filename)
            df = pd.read_csv(file_path)
            all_dataframes.append(df)

    combined_df_part1 = pd.concat(all_dataframes, ignore_index=True)

    df=combined_df_part1.copy(deep=True)
    df.head()
```

	train_id	name	item_condition_id	category_name	brand_name	price	shipping	clean_descript
0	128037	Bundle for Sassy Sisters	3	Women/Tops & Blouses/Blouse	NaN	16.0	0	max cleo black d paper crane b tank
1	491755	PINK VS TANK	2	Women/Tops & Blouses/Tank, Cami	NaN	17.0	0	sequin pink sequins miss gently w

Data was loaded from Google drive from the folder where the code has extracted the contents of the zip file, then we combine all available files into single data frame which is further used in the solution.

Q1.1 Find the missing values:

• Write the function missing values table and use the dataframe as the input. The function should return the information of missing values by column (only for columns which have missing values and the returned value should be the count of rows has missing values)

• For columns which have missing values, could you impute the missing values with the mean value of the columns? (If you think it could not be done with mean value, write down the reason in comments and report rather than code)

Answer:

```
def missing_values_table(df):
    # Calculate the total number of missing values in each column
    missing_values = df.isnull().sum()

    # Create a DataFrame to store the missing value information
    missing_table = pd.DataFrame({'Missing Values': missing_values})

    # Filter the columns with missing values (count > 0)
    missing_table = missing_table[missing_table['Missing Values'] > 0]

    return missing_table

# Using df as input in the function
missing_table = missing_values_table(df)
print(missing_table)
```

```
                   Missing Values
category_name                1539
brand_name                 151956
clean_description             194
```

- We create a new function missing_values_table which will provide us with a table of missing value in data provided to it.
- We then pass our data through the function, and we find that there are three categories which have missing values. As we see:
    - category_name has 1539 missing values.
    - brand_name has 151956 & clean_description has 194 missing values.
- We will not be able to impute these missing values with mean value as they are categorical in nature and mean value can only be derived for numeric values.

Q1.2 Find the price information from the data:

• Write code to print the median price of the items in the data.

• What is the 90th percentile value on the price.

• Draw the histogram chart for the price of the items in the data with 50 bins.

Answer:

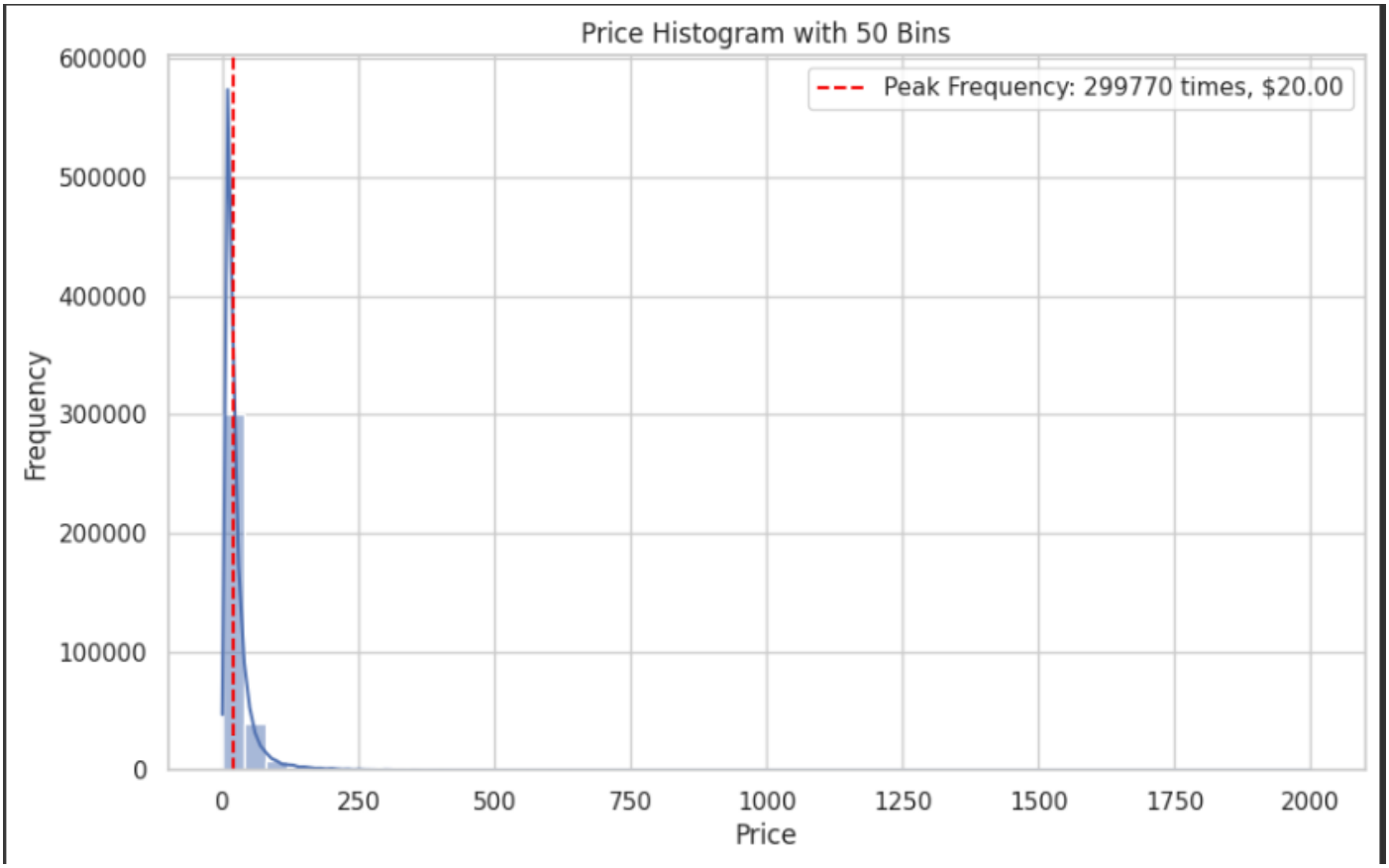
```
print('Median Price:', df['price'].median())
```

```
Median Price: 17.0
```

```
# Calculate the 90th percentile of the "price" column
percentile_90 = np.percentile(df['price'].dropna(), 90)
print(percentile_90)
```

```
51.0
```

```python
# Set the style for Seaborn plots (optional)
sns.set(style="whitegrid")

# Plot a histogram for the "price" column with 50 bins
plt.figure(figsize=(10, 6))  # Adjust the figure size
sns.histplot(data=df, x='price', bins=50, kde=True)
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.title('Price Histogram with 50 Bins')

# Find the bin with the highest frequency
hist, bin_edges = np.histogram(df['price'], bins=50)
peak_bin_index = np.argmax(hist)
peak_bin_value = (bin_edges[peak_bin_index] + bin_edges[peak_bin_index + 1]) / 2
peak_frequency = hist[peak_bin_index]

# Add a vertical line at the peak frequency
plt.axvline(x=peak_bin_value, color='red', linestyle='--', label=f'Peak Frequency: {peak_frequency} times, ${peak_bin_value:.2f}')
plt.legend()

plt.show()
```



- 90th percentile value on the price function is 51.0. We found this using the numpy library with .percentile() function, where we dropped null values to get the 90th percentile.
- We also drew a histogram chart using the seaborn library and matplotlib with 50 bins. We used whitegride style in the seaborn library. Setting the figure size to 10 by 6. On the x axis we have frequency and on y axis there's price.
- We create a histogram of the "price" column from the DataFrame df with 50 bins. We are able to see that frequency peaks at $20.00.

Q1.3 Exploring the shipping information from the data:

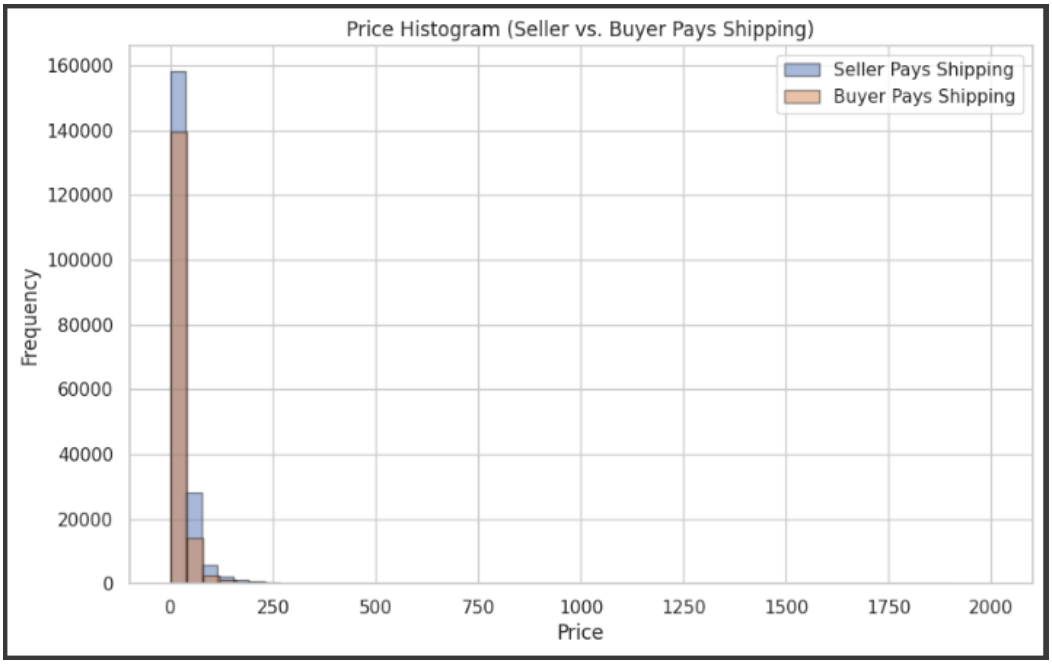
- Write code to find out the percentage of the items that are paid by the buyers.
- Draw (two) histogram graphs in one plot on the price for seller pays shipping and buyer pays shipping (50 bins).
- When buying the items online, do you need to pay higher price if seller pays for the shipping? Write the code to find out (Compare the median price of items paid by buyers and items paid by sellers and explain the result in the comment and report). (Optional: You could use the subplot from EDA)

Answer:

```
# Calculate the total number of items
total_items = len(df)

# Calculate the number of items paid by buyers (where price is greater than 0)
items_paid_by_buyers = len(df[df['price'] > 0])

# Calculate the percentage
percentage_paid_by_buyers = (items_paid_by_buyers / total_items) * 100

# Print the result
print(f"Percentage of items paid by buyers: {percentage_paid_by_buyers:.2f}%")
```

```
Percentage of items paid by buyers: 99.95%
```

```
# Filter the DataFrame for seller pays shipping and buyer pays shipping
seller_pays_shipping = df[df['shipping'] == 0]
buyer_pays_shipping = df[df['shipping'] == 1]

# Create a figure and axis for the subplots
fig, ax = plt.subplots(figsize=(10, 6))

# Plot histograms for seller pays shipping and buyer pays shipping on the same axes
ax.hist(seller_pays_shipping['price'].dropna(), bins=50, edgecolor='k', alpha=0.5, label='Seller Pays Shipping')
ax.hist(buyer_pays_shipping['price'].dropna(), bins=50, edgecolor='k', alpha=0.5, label='Buyer Pays Shipping')

# Add labels and legend
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.title('Price Histogram (Seller vs. Buyer Pays Shipping)')
plt.legend()

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



- To know the percentage of the items that are paid by the buyers we first define the number of entries as total_items and then find the values in the price column that are not zero or that are free as item_paid_by_buyers.
- We then divide item_pasid_by_buyers with total_items and multiply it by 100.
- We find that 99.95% of people paid for the item they ordered.
- We then draw a histogram by filtering the dataframe into two different datasets: seller_pays_shipping and buyer_pays_shipping. Then we create a figure size of 10, 6 with 50 bins, x label is price and y label is frequency. We can compare the price when seller pays for shipping and when the buyer pays for shipping.

```
# Calculate the median price of items paid by buyers (buyer pays shipping)
median_price_buyer_pays_shipping = df[df['shipping'] == 1]['price'].median()

# Calculate the median price of items paid by sellers (seller pays shipping)
median_price_seller_pays_shipping = df[df['shipping'] == 0]['price'].median()

# Print the results
print(f"Median price of items paid by buyers: ${median_price_buyer_pays_shipping:.2f}")
print(f"Median price of items paid by sellers: ${median_price_seller_pays_shipping:.2f}")

# Compare the medians and explain the result
if median_price_buyer_pays_shipping < median_price_seller_pays_shipping:
    print("Items paid by buyers have a lower median price, indicating that buyers may pay less when the seller pays for shipping.")
elif median_price_buyer_pays_shipping > median_price_seller_pays_shipping:
    print("Items paid by buyers have a higher median price, indicating that buyers may pay more when they have to pay for shipping.")
else:
    print("The median prices are the same for items paid by buyers and sellers.")
```

```
Median price of items paid by buyers: $14.00
Median price of items paid by sellers: $19.00
Items paid by buyers have a lower median price, indicating that buyers may pay less when the seller pays for shipping.
```

- We calculate the median price of items for which the buyer pays for shipping. It filters the DataFrame to include only rows where the 'shipping' column is equal to 1 (indicating buyer pays shipping) and then computes the median price from the 'price' column.
- Similarly, next we calculate the median price of items for which the seller pays for shipping. It filters the DataFrame to include only rows where the 'shipping' column is equal to 0 (indicating seller pays shipping) and then computes the median price from the 'price' column. then we print results of the two and compare using if condition.

Q1.4 You are required to find out the item condition information from the data. Lower the number (value), the better condition of the item.

- Write the code to find out (print) the count of the rows on each number (value) in column item_condition_id.
- Draw the boxplot graphs (one plot) on the price for each item condition value, and find out out whether the better condition of the item could have higher median price (draw the plot and answer this question in the comment and report).

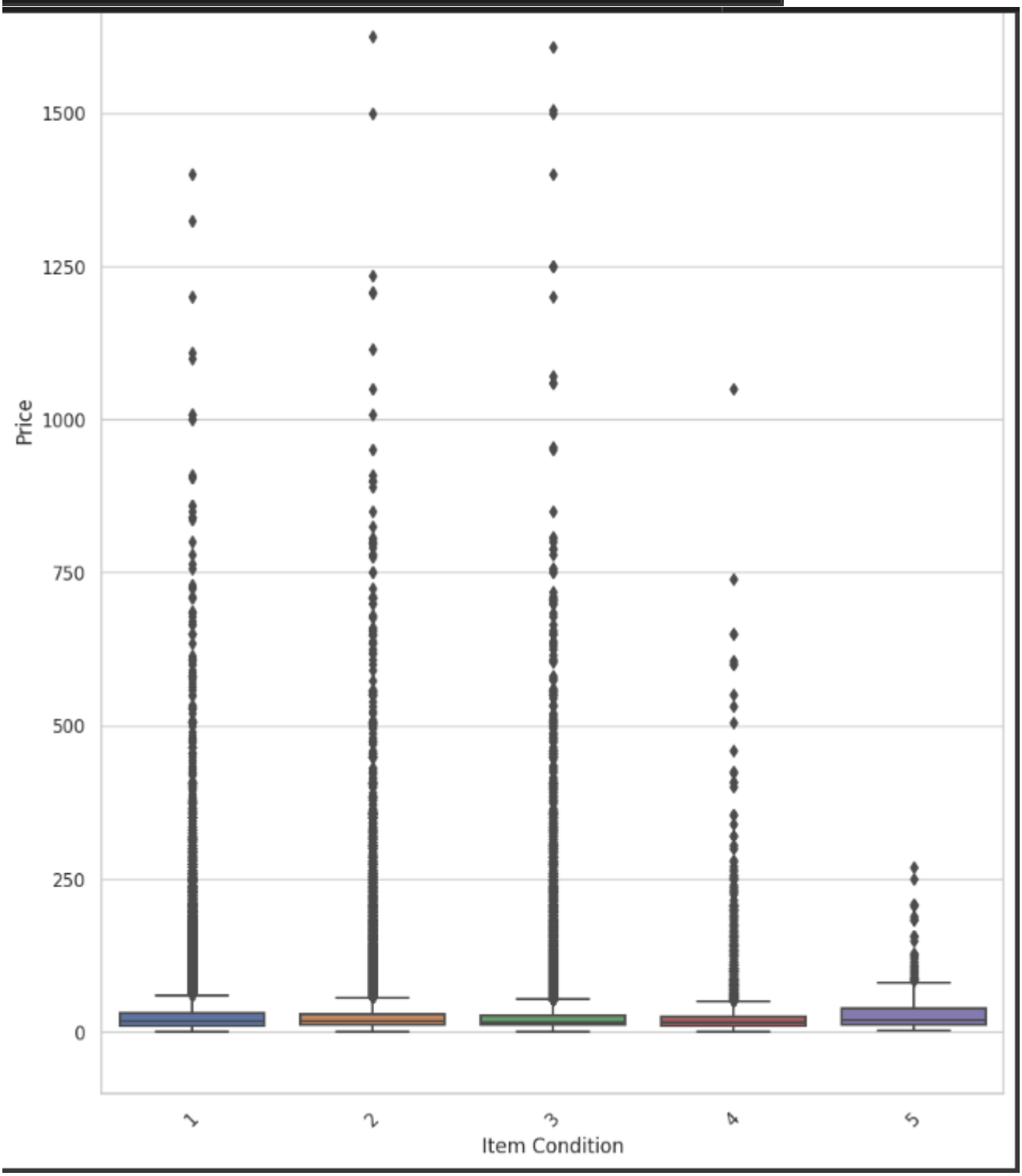
Answer:

```
# Count the rows for each unique value in the "item_condition_id" column
condition_counts = df['item_condition_id'].value_counts()

# Print the count of rows for each condition
print("Item Condition Counts:")
for condition_id, count in condition_counts.items():
    print(f"Condition {condition_id}: {count} rows")
```

```
Item Condition Counts:
Condition 1: 153402 rows
Condition 3: 104248 rows
Condition 2: 89843 rows
Condition 4: 7768 rows
Condition 5: 547 rows
```

- We have 5 unique conditions for the condition column. We use .value_counts() function to find condition counts.
- Maximum items are in condition 1.

```
# Set the style for Seaborn plots (optional)
sns.set(style="whitegrid")

# Create a boxplot for price vs. item_condition_id
plt.figure(figsize=(10, 15))  # Optional: Adjust the figure size
sns.boxplot(data=df, x='item_condition_id', y='price')
plt.xlabel('Item Condition')
plt.ylabel('Price')
plt.title('Boxplot of Price by Item Condition')
plt.xticks(rotation=45)  # Optional: Rotate x-axis labels for better visibility

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



- We draw a box plot to check if the median price for a better condition item is more. We find that its not so.

Q1.5 Conduct the category analysis and find out the relevant information:

- Write the code to find out (print) how many unique categories you could find from column category_name.
- For the items with worst condition only (highest value from item_condition_id), write code to (print) find out the top 3 categories (now you probably understand the findings you had in Question 1.4).

Answer:

```
df['category_name'].nunique()
```

```
1135
```

```
# Create a pivot table to count the number of items in each category for the worst condition
pivot_table = pd.pivot_table(df[df['item_condition_id'] == df['item_condition_id'].max()],
                             values='name',
                             index='category_name',
                             aggfunc='count')

# Sort the pivot table by the count of items in descending order
sorted_pivot = pivot_table.sort_values(by='name', ascending=False)

# Get the top 3 categories
top_categories = sorted_pivot.head(3)

# Print the top 3 categories
print("Top 3 Categories for Items with Worst Condition:")
print(top_categories)
```

```
Top 3 Categories for Items with Worst Condition:
                                                 name
category_name
Electronics/Cell Phones & Accessories/Cell Phon...   137
Electronics/Video Games & Consoles/Games              42
Electronics/Video Games & Consoles/Consoles           35
```

- .nunique() function gives us a total of 1135 unique entries in category_name.
- We used pivot table sort values to 3 worst items in the worst condition and found that electronics accessories and video game consoles are generally there.

Q1.6 The categories in column category_name have 3 parts. The three parts (main_cat,subcat_1 and subcat_2) are concatenated with '/' character sequentially in the data now.

- Write the function (must be function) to split the text content (string value in each row) in column category_name by '/' character. you need to handle the exception in the function for those has missing values (NaN). For missing values (NaN), the results from splitting should be "Category Unknown", "Category Unknown", "Category Unknown".

- Use the above function you wrote to create three new columns main_cat,subcat_1 and subcat_2 with corresponding values from the result of splitting. Print out the dataframe to show the top 5 rows for three new columns main_cat,subcat_1 and subcat_2.

Answer:

```python
def split_category_name(category_name):
    try:
        # Split the category_name by '/' and extract the three parts
        parts = category_name.split('/')
        main_cat = parts[0].strip()
        subcat_1 = parts[1].strip()
        subcat_2 = parts[2].strip()
    except AttributeError:
        # Handle missing values (NaN)
        main_cat = "Category Unknown"
        subcat_1 = "Category Unknown"
        subcat_2 = "Category Unknown"
    except IndexError:
        # Handle cases where there are not enough parts
        main_cat = "Category Unknown"
        subcat_1 = "Category Unknown"
        subcat_2 = "Category Unknown"

    return main_cat, subcat_1, subcat_2

# Apply the function to the 'category_name' column and create new columns
df[['main_cat', 'subcat_1', 'subcat_2']] = df['category_name'].apply(split_category_name).apply(pd.Series)

# Print the first few rows to verify the result
print(df[['category_name', 'main_cat', 'subcat_1', 'subcat_2']].head())
```

```
                     category_name main_cat        subcat_1  \
0        Women/Tops & Blouses/Blouse    Women  Tops & Blouses
1    Women/Tops & Blouses/Tank, Cami    Women  Tops & Blouses
2  Kids/Toys/Action Figures & Statues     Kids           Toys
3             Kids/Boys 2T-5T/Shoes     Kids      Boys 2T-5T
4        Kids/Girls 0-24 Mos/Dresses     Kids  Girls 0-24 Mos

                 subcat_2
0                  Blouse
1              Tank, Cami
2  Action Figures & Statues
3                   Shoes
4                 Dresses
```

```python
# Apply the function to the 'category_name' column and create new columns
df[['main_cat', 'subcat_1', 'subcat_2']] = df['category_name'].apply(split_category_name).apply(pd.Series)

# Print the top 5 rows of the new columns
print("Top 5 Rows of main_cat, subcat_1, and subcat_2:")
df.head()
```

```
Top 5 Rows of main_cat, subcat_1, and subcat_2:
```

	train_id	name	item_condition_id	category_name	brand_name	price	shipping	clean_description	main_cat	subcat_1	subcat_2
0	128037	Bundle for Sassy Sisters	3	Women/Tops & Blouses/Blouse	NaN	16.0	0	max cleo black dress paper crane black tank to...	Women	Tops & Blouses	Blouse
1	491755	PINK VS TANK	2	Women/Tops & Blouses/Tank, Cami	NaN	17.0	0	sequin pink sign sequins missing gently worn	Women	Tops & Blouses	Tank, Cami

- The solution involves string operation by splitting the '**category_name**' into 3 parts. We are using the '**split**' method to split the text into a list of substrings based on the forward slash '**/**' as the delimiter. '**categories**' will be a list containing the main category, subcategory 1, and subcategory 2,
- We can use python regular expressions package for splitting the text, but as our input text pattern and the delimiter are simple, it is recommended to use split function as it is more concise, readable and performs well for such basic splitting tasks.

Q1.7 After splitting the category for column category_name, we now have the three main details regarding to the category information. However, we need to clean the text in each of the new three columns in lowercase.

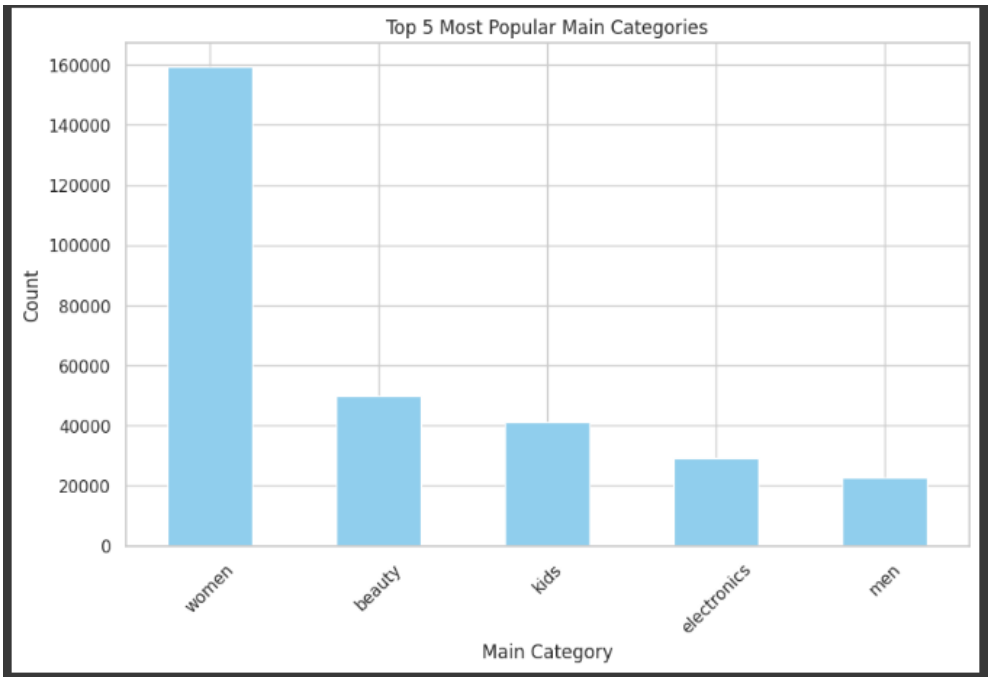
- Write code (or function) to change the text (value in each row) from the new three columns to lowercase.
- Draw the bar chart to find out the top 5 most popular main categories (in column main_cat) in the data (only showing the top 5).
- Write code (or function) to (print) find out how many unique main categories (in column main_cat), unique first sub-categories (in column subcat_1) and unique second sub-categories (in column subcat_2) respectively.

Answer:

```
def lowercase_categories(df):
    # Apply lowercase transformation to the specified columns
    df['main_cat'] = df['main_cat'].str.lower()
    df['subcat_1'] = df['subcat_1'].str.lower()
    df['subcat_2'] = df['subcat_2'].str.lower()

# Call the function to convert the categories to lowercase
lowercase_categories(df)

# Print the first few rows to verify the result
df.head()
```

	train_id	name	item_condition_id	category_name	brand_name	price	shipping	clean_description	main_cat	subcat_1	subcat_2
0	128037	Bundle for Sassy Sisters	3	Women/Tops & Blouses/Blouse	NaN	16.0	0	max cleo black dress paper crane black tank to...	women	tops & blouses	blouse
1	491755	PINK VS TANK	2	Women/Tops & Blouses/Tank, Cami	NaN	17.0	0	sequin pink sign sequins missing gently worn	women	tops & blouses	tank, cami

```
# Calculate the top 5 most popular main categories
top_main_categories = df['main_cat'].value_counts().head(5)

# Create a bar chart
plt.figure(figsize=(10, 6))
top_main_categories.plot(kind='bar', color='skyblue')
plt.title('Top 5 Most Popular Main Categories')
plt.xlabel('Main Category')
plt.ylabel('Count')
plt.xticks(rotation=45)  # Rotate x-axis labels for better visibility

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



```
# Find the number of unique main categories in the 'main_cat' column
unique_main_categories_count = df['main_cat'].nunique()

# Find the number of unique first sub-categories in the 'subcat_1' column
unique_subcat_1_count = df['subcat_1'].nunique()

# Find the number of unique second sub-categories in the 'subcat_2' column
unique_subcat_2_count = df['subcat_2'].nunique()

# Print the counts
print(f"Number of unique main categories: {unique_main_categories_count}")
print(f"Number of unique first sub-categories: {unique_subcat_1_count}")
print(f"Number of unique second sub-categories: {unique_subcat_2_count}")
```

```
Number of unique main categories: 11
Number of unique first sub-categories: 114
Number of unique second sub-categories: 788
```

- To convert the text to lowercase we use the '**lower'** method on each new column separately,
- We can even club all the new columns to make a dataframe and then use '**applymap'** method to apply the 'lower' function element-wise to all elements of the dataframe.
- Same way we can use applymap method to apply '**nunique**' function element-wise to the combined dataframe. but we don't want t complicate the solution and we want to keep the code more readable.
- To plot the graphs we have used matplotlib plot function to plot the graphs, we can also use seaborn.

Q1.8 Exploring the price and categories.

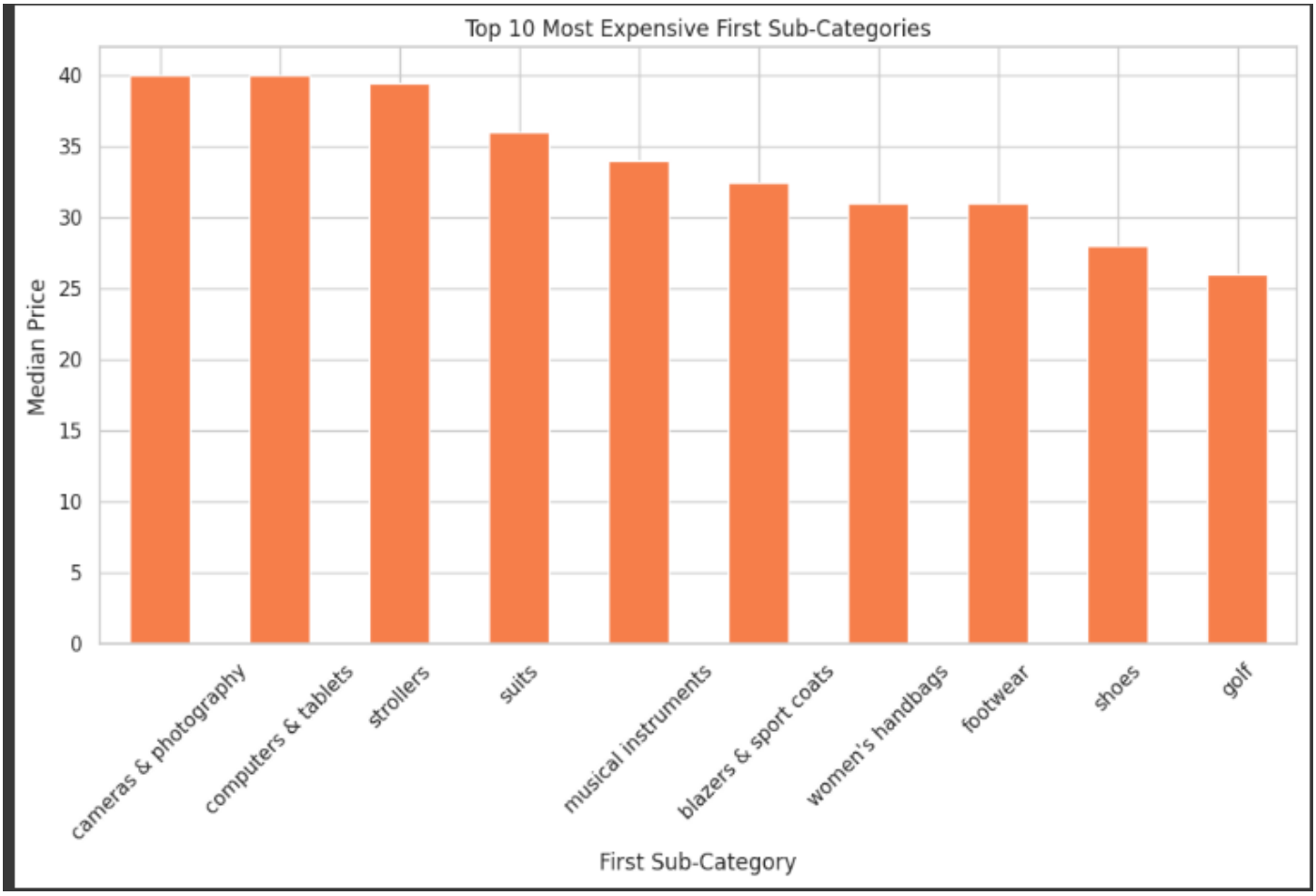
• Write code to (print) find out the median price for all the categories in new column main_cat.
• Draw the bar chart to find out the top 10 most expensive first sub-categories (in column subcat_1) in the data.
• Draw the bar chart to find out the top 10 cheapest second sub-categories (in column subcat_2) in the data.

Answer:

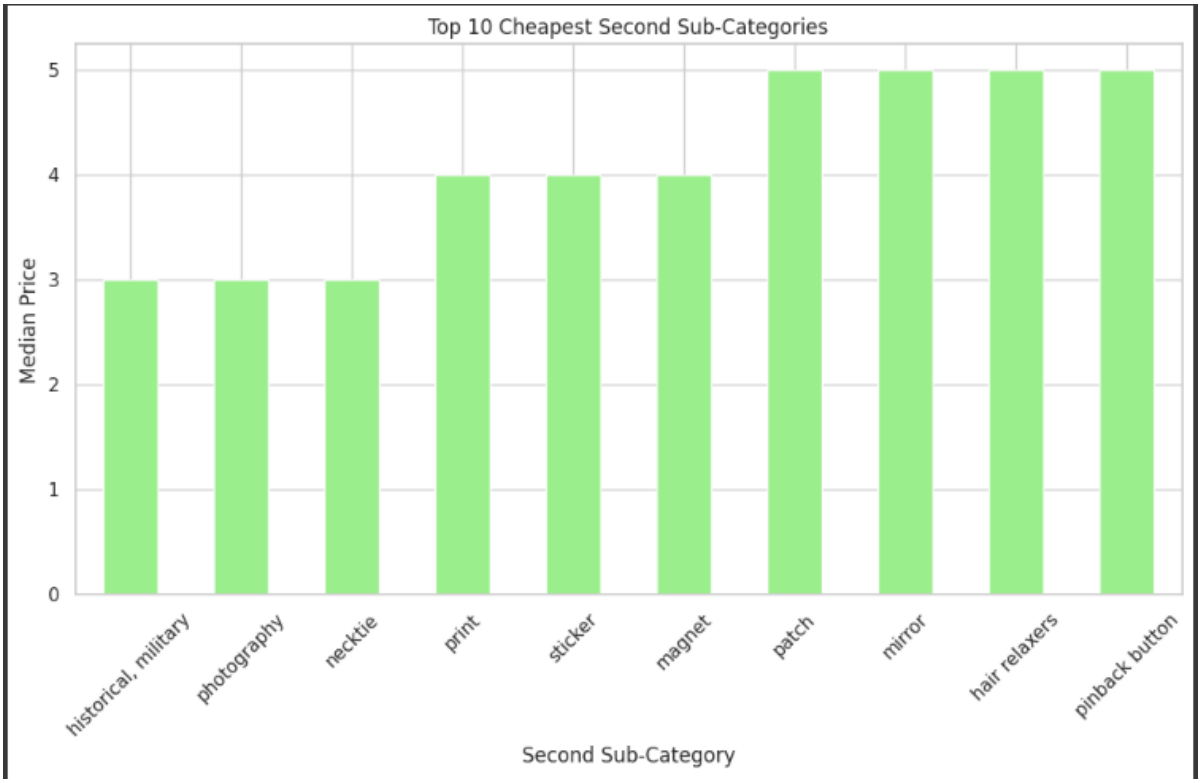
```
# Group the DataFrame by 'main_cat' and calculate the median price for each category
median_prices_by_main_cat = df.groupby('main_cat')['price'].median()

# Print the median prices for each main category
print("Median Price for Each Main Category:")
print(median_prices_by_main_cat)
```

```
Median Price for Each Main Category:
main_cat
beauty                  15.0
category unknown        18.0
electronics             15.0
handmade                12.0
home                    18.0
kids                    14.0
men                     21.0
other                   14.0
sports & outdoors       16.0
vintage & collectibles  16.0
women                   19.0
Name: price, dtype: float64
```

```python
# Calculate the median price for each first sub-category and sort in descending order
top_10_expensive_subcat_1 = df.groupby('subcat_1')['price'].median().sort_values(ascending=False).head(10)

# Create a bar chart
plt.figure(figsize=(12, 6))
top_10_expensive_subcat_1.plot(kind='bar', color='coral')
plt.title('Top 10 Most Expensive First Sub-Categories')
plt.xlabel('First Sub-Category')
plt.ylabel('Median Price')
plt.xticks(rotation=45)  # Rotate x-axis labels for better visibility

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




```python
# Calculate the median price for each second sub-category and sort in ascending order
top_10_cheapest_subcat_2 = df.groupby('subcat_2')['price'].median().sort_values().head(10)

# Create a bar chart
plt.figure(figsize=(12, 6))
top_10_cheapest_subcat_2.plot(kind='bar', color='lightgreen')
plt.title('Top 10 Cheapest Second Sub-Categories')
plt.xlabel('Second Sub-Category')
plt.ylabel('Median Price')
plt.xticks(rotation=45)  # Rotate x-axis labels for better visibility

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



- To find the median price for all categories in a column, first we can use '**groupby**' function which splits the dataframe into different groups for each category in the 'main_cat' column, then we can apply '**median**' on the '**price**' column to find the median for each group and then sort and get the top 10 values,
- We groupby the 'subcat_1', apply median and sort the values in descending order.
- We groupby the subcat_2 apply median and sort the values in ascending order.
- We can also use pivot_table function on the data frame as a replacement of groupby,which can be more concise.

Q1.9 Exploring the price and brand.

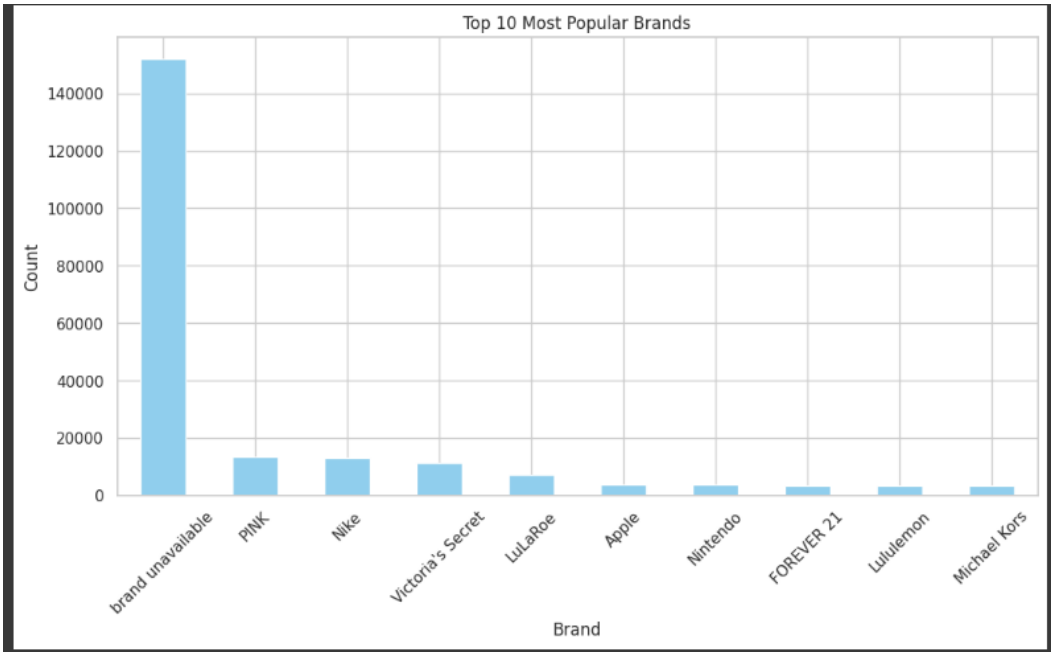
- Write code to (print) find out the median price for all the brands (fill NaN with 'brand unavailable').
- Draw the bar chart to find out the top 10 most popular brands in the data.

Answer:

```
# Fill missing values in the 'brand_name' column with 'brand unavailable'
df['brand_name'].fillna('brand unavailable', inplace=True)

# Group the DataFrame by 'brand_name' and calculate the median price for each brand
median_prices_by_brand = df.groupby('brand_name')['price'].median()

# Print the median prices for each brand
print("Median Price for Each Brand:")
print(median_prices_by_brand)
```

```
Median Price for Each Brand:
brand_name
% Pure              14.0
10.Deep             18.0
21men               10.0
3.1 Phillip Lim    232.5
3M©                 15.0
                   ...
timi & leslie       65.0
tokidoki            18.0
totes ISOTONER      14.0
triangl swimwear    44.0
vineyard vines      21.0
Name: price, Length: 3047, dtype: float64
```

```python
# Calculate the top 10 most popular brands by counting occurrences
top_10_popular_brands = df['brand_name'].value_counts().head(10)

# Create a bar chart
plt.figure(figsize=(12, 6))
top_10_popular_brands.plot(kind='bar', color='skyblue')
plt.title('Top 10 Most Popular Brands')
plt.xlabel('Brand')
plt.ylabel('Count')
plt.xticks(rotation=45)  # Rotate x-axis labels for better visibility

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



- The solution first calculates the median price for each brand, filling NaN values with '**brand unavailable**' using **groupby** and **fillna**. It then creates a bar chart to display the top 10 most popular brands using **value_counts** and **matplotlibs bar plot function**.
- An alternative approach for finding popular brands is to use **nlargest** on the aggregated result of **groupby('brand_name'). size()**, which counts occurrences.
- Both solutions offer insights into brand pricing and popularity, with the choice depending on whether the focus is on median prices or count-based popularity.

Q1.10 Item Description Analysis.

- Could you draw the wordcloud chart by using the column clean_description.
- Divide the data with quantiles of the price (using qcut from pandas to obtain the first/second/third/fourth quantile).
- Draw the word could by using the column clean_description on each quantile of price data

Answer:

```
# Combine all descriptions into a single string
text = ' '.join(df['clean_description'].astype(str))

# Generate the word cloud
wordcloud = WordCloud(width=800, height=400, background_color='white').generate(text)

# Create a plot
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('Word Cloud for Item Descriptions')
plt.show()
```


```
# Define the number of quantiles you want (e.g., 4 for quartiles)
num_quantiles = 4

# Split the data into quantiles based on price
quantiles = pd.qcut(df['price'], num_quantiles, labels=False)

# Create a separate word cloud for each quantile
for i in range(num_quantiles):
    # Filter the data for the current quantile
    quantile_data = df[quantiles == i]

    # Combine descriptions into a single string
    text = ' '.join(quantile_data['clean_description'].astype(str))

    # Generate the word cloud
    wordcloud = WordCloud(width=800, height=400, background_color='white').generate(text)

    # Create a plot for the current quantile
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.axis('off')
    plt.title(f'Word Cloud for Quantile {i+1} of Price')

    # Show the plot for the current quantile
    plt.show()
```

- This solution employs word clouds to visualize the most frequent terms in **item descriptions** within each price quantile, providing a concise representation of language patterns.
- An alternative approach could involve sentiment analysis; However, word clouds are effective for quick, intuitive visualization of high-frequency terms, making them suitable for initial exploratory analysis.
- They are especially helpful when seeking a visual overview of common terms associated with different price ranges.

Part 2 : Time series analysis exercise

Data Ingestion

```
folder_path = '/content/drive/My Drive/NY_Taxi_Data' #----- Setting Folder Path----#

#----- Combining multiple files (if any) in the Folder Path into single data frame ----#

all_dataframes = []
for filename in os.listdir(folder_path):
    if filename.endswith('.csv'):
        file_path = os.path.join(folder_path, filename)
        df = pd.read_csv(file_path)
        all_dataframes.append(df)

combined_df = pd.concat(all_dataframes, ignore_index=True)


combined_df.head(2) #----- Data Check----#
```

Data was loaded from Google drive from the folder where the code has extracted the contents of the zip file, then we combine all available files into single data frame which is further used in the solution.

Q2.1 The dataset used here is the New York City Taxi Demand dataset. The raw data is from the NYC Taxi and Limousine Commission. The data included here consists of aggregating the total number of taxi passengers into 30-minute buckets. In this question, we will simply process the data and explore the time series.

- Create two new dataframes df_day and df_hour by aggregating the demand value on daily and hourly level.
- Plot the demand value in two line charts for both df_day and df_hour dataframes.
- Plot the seasonal decomposition components (Trend, Seasonal, Residual) from df_day dataframe, also find out the p value from adfuller test. Do you think the df_day is stationary enough (please explain your reasons in comments and report)?

Answer:

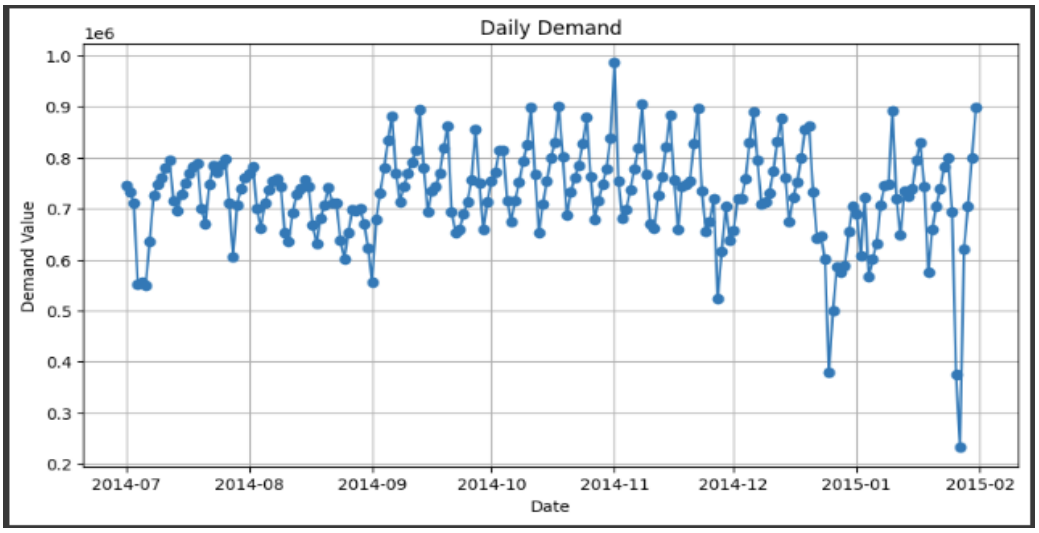
```
#-----Convert the 'timestamp' column to a datetime object ----#
df['timestamp'] = pd.to_datetime(df['timestamp'])

#-----Set 'timestamp' as the index ----#
df.set_index('timestamp', inplace=True)

#-----Resample data on daily and hourly level and calculate the sum ----#
df_day_temp = df.resample('D').sum()
df_hour_temp = df.resample('H').sum()


#-----Adding DateStamp Index as column ----#
df_day = df_day_temp.copy(deep=True)
df_day['timestamp'] = df_day.index
df_hour =df_hour_temp.copy(deep=True)
df_hour['timestamp'] = df_hour.index
```

```
df_day.info() #----- Data Check----#
```

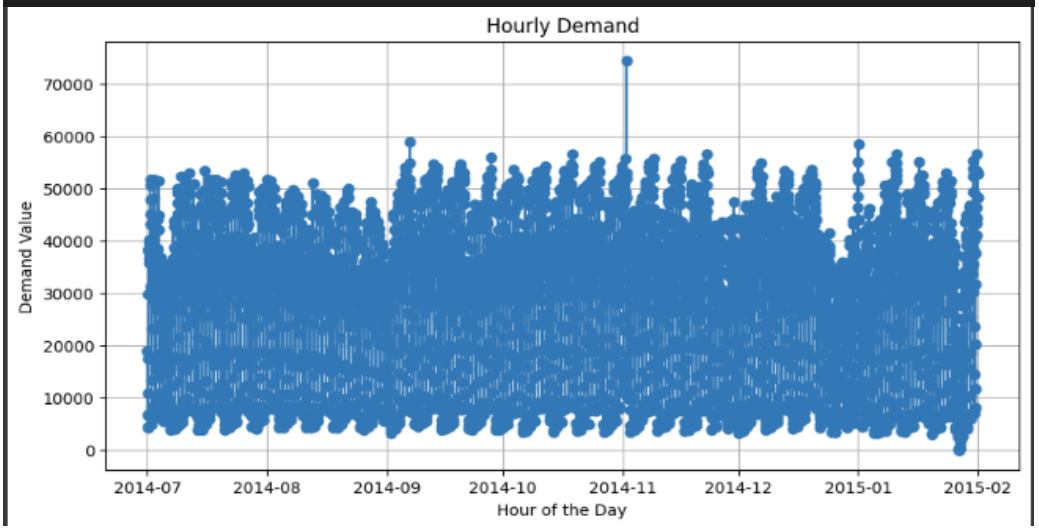
```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 215 entries, 2014-07-01 to 2015-01-31
Freq: D
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   value      215 non-null    int64
 1   timestamp  215 non-null    datetime64[ns]
dtypes: datetime64[ns](1), int64(1)
memory usage: 5.0 KB
```

- In the above code we convert the datatype of the “timestamp” column and push it as index of the dataframe, then we aggregate the demand data in the “value” column at day and hour level using the resample function.
- Then for plotting data we add the index of the dataframe as column and proceed with plotting the graphs

```
#----- Plot df_day----#
plt.figure(figsize=(10, 5))
plt.plot(df_day['timestamp'], df_day['value'], marker='o', linestyle='-')
plt.title('Daily Demand')
plt.xlabel('Date')
plt.ylabel('Demand Value')
plt.grid(True)
plt.show()
```

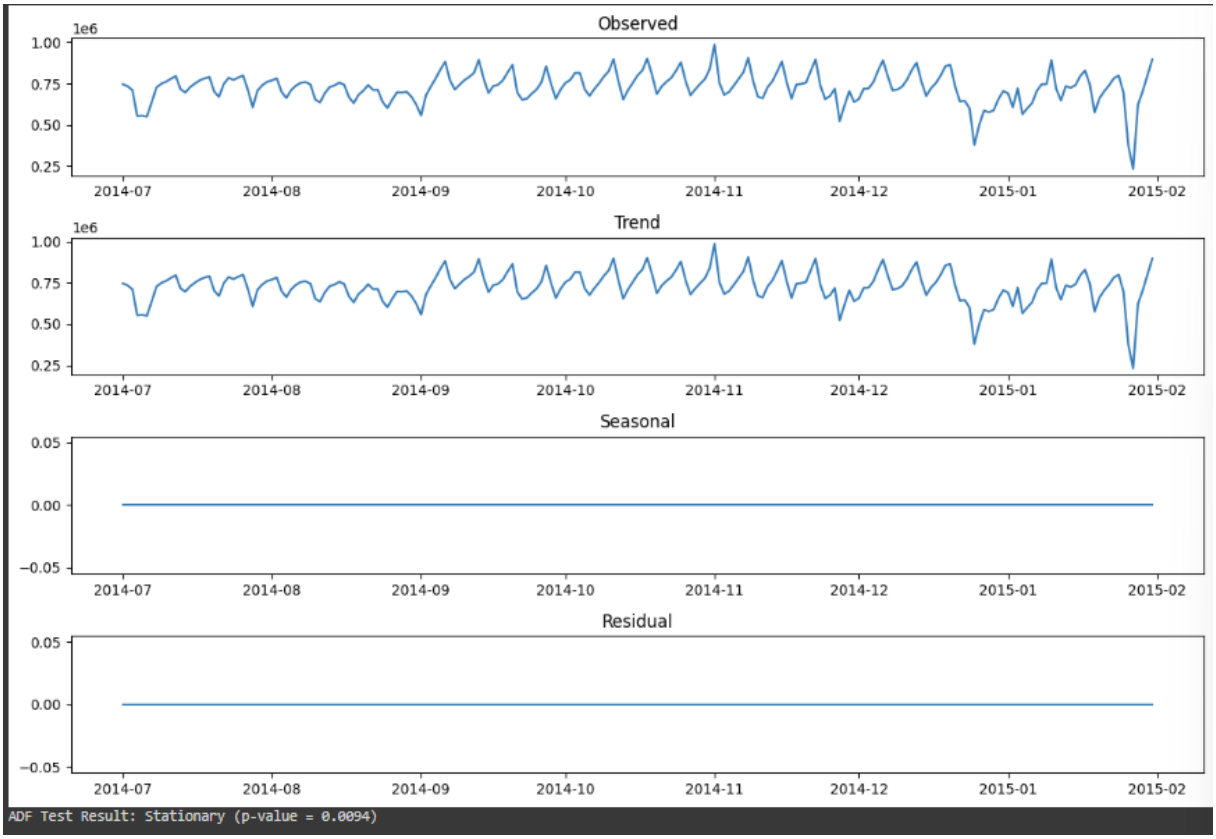



```
#-----Plot df_hour----#
plt.figure(figsize=(10, 5))
plt.plot(df_hour['timestamp'], df_hour['value'], marker='o', linestyle='-')
plt.title('Hourly Demand')
plt.xlabel('Hour of the Day')
plt.ylabel('Demand Value')
plt.grid(True)
plt.show()
```



- In the above code we plot the Daily demand and Hourly demand using the dataframe which contain aggregated demand values at day and hour level.

```python
df_day = df_day_temp.copy(deep=True)
df_day['timestamp'] = df_day.index

#-----Perform seasonal decomposition----#
result = seasonal_decompose(df_day['value'], model='additive', period=1)

#-----Plot the components----#
plt.figure(figsize=(12, 8))

plt.subplot(411)
plt.plot(df_day['timestamp'], result.observed)
plt.title('Observed')

plt.subplot(412)
plt.plot(df_day['timestamp'], result.trend)
plt.title('Trend')

plt.subplot(413)
plt.plot(df_day['timestamp'], result.seasonal)
plt.title('Seasonal')

plt.subplot(414)
plt.plot(df_day['timestamp'], result.resid)
plt.title('Residual')

plt.tight_layout()
plt.show()

#-----Perform ADF test to check for stationarity----#
adf_result = adfuller(df_day['value'])
p_value = adf_result[1]

#-----Check stationarity----#
if p_value <= 0.05:
    stationarity_status = "Stationary (p-value = {:.4f})".format(p_value)
else:
    stationarity_status = "Not Stationary (p-value = {:.4f})".format(p_value)

print("ADF Test Result:", stationarity_status)
```




```
ADF Test Result: Stationary (p-value = 0.0094)
```

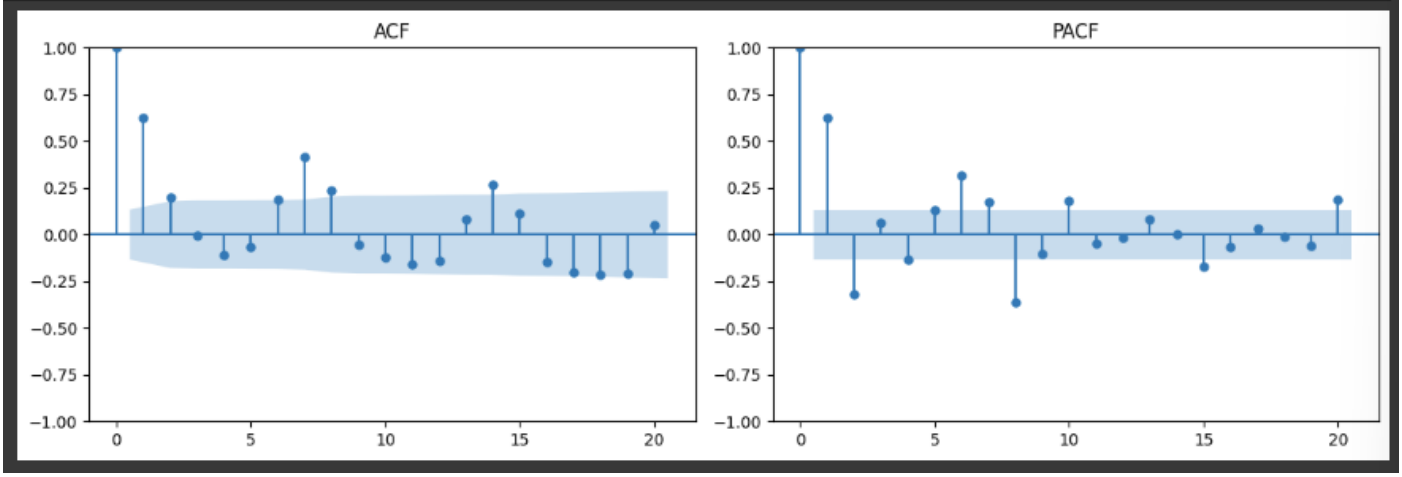
- In the above code, seasonal decomposition is performed using the `seasonal_decompose` function from statsmodels, breaking down daily demand data into observed, trend, seasonal, and residual components, which are then plotted.
- An alternative approach could involve different decomposition methods or changing to a multiplicative model depending on the data's characteristics.
- We also check stationarity, using Augmented Dickey-Fuller (ADF) test to calculate the p-value and determine stationarity based on a threshold of 0.05.

Q2.2 In this question, we will try to use time series model such as ARIMA and others to build the model(s) for forecasting the future.

• Create the acf and pacf plots for df_day dataframe.
• Find the best model with different parameters on ARIMA model. The parameter range for p,d,q are all from [0, 1, 2]. In total, you need to find out the best model with lowest Mean Abosulate Error from 27 choices based on the time from ”Jul-01-2014” to ”Dec-01-2014”.
• Using the best model in above steps to forecast the time from ”Jan-01-2015” to ”Jan-31-2015”. Plot the predicted value and the true demand value from ”Jan-01-2015” to ”Jan-31-2015”.
• Could you think of any other model (not as same as ARIMA) could do the forecasting for demand value from ”Jan-01-2015” to ”Jan-31-2015”? You could choose one model (except ARIMA) and train the model based on the demand value from ”Jul-01-2014” to”Dec-01-2014” (same training data as the ARIMA). Hint: there are some resources
regarding other time series forecasting models such as prophet here and also the exponential smoothing here.

Answer:

```
#-----Plot ACF and PACF plots to determine orders (p, d, q)----#
plt.figure(figsize=(12, 4))

plt.subplot(121)
plot_acf(df_day['value'], lags=20, ax=plt.gca())
plt.title('ACF')

plt.subplot(122)
plot_pacf(df_day['value'], lags=20, ax=plt.gca())
plt.title('PACF')

plt.tight_layout()
plt.show()
```

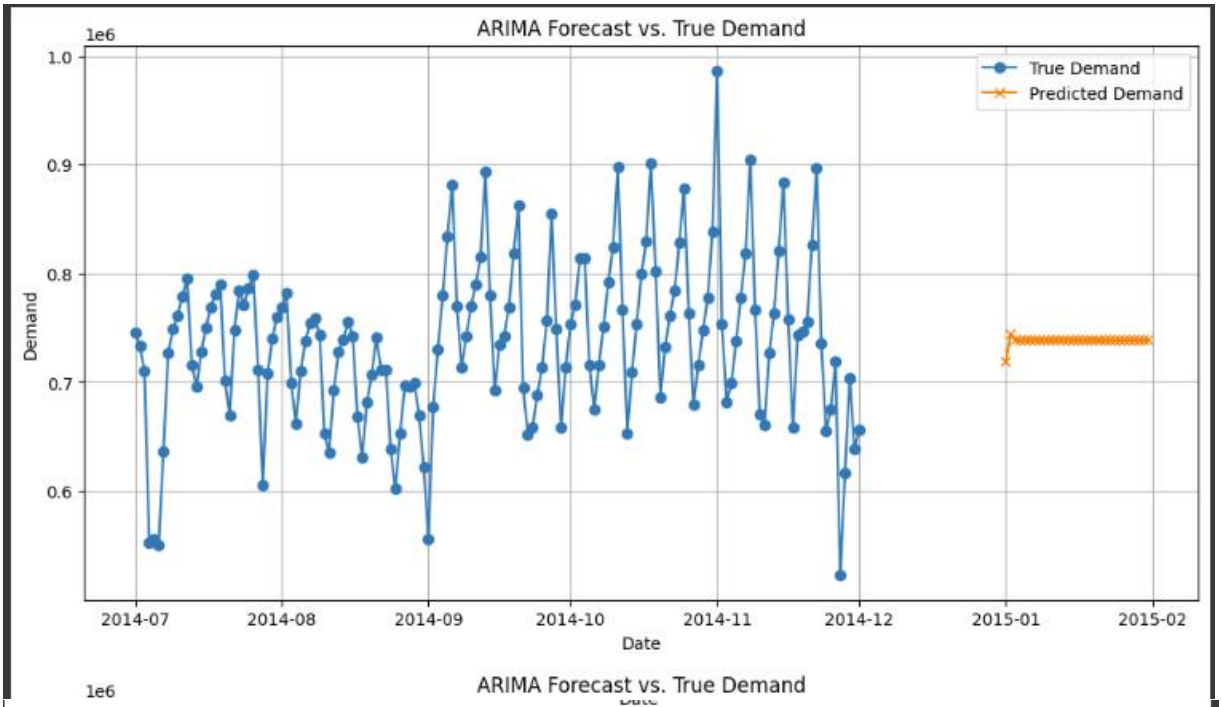


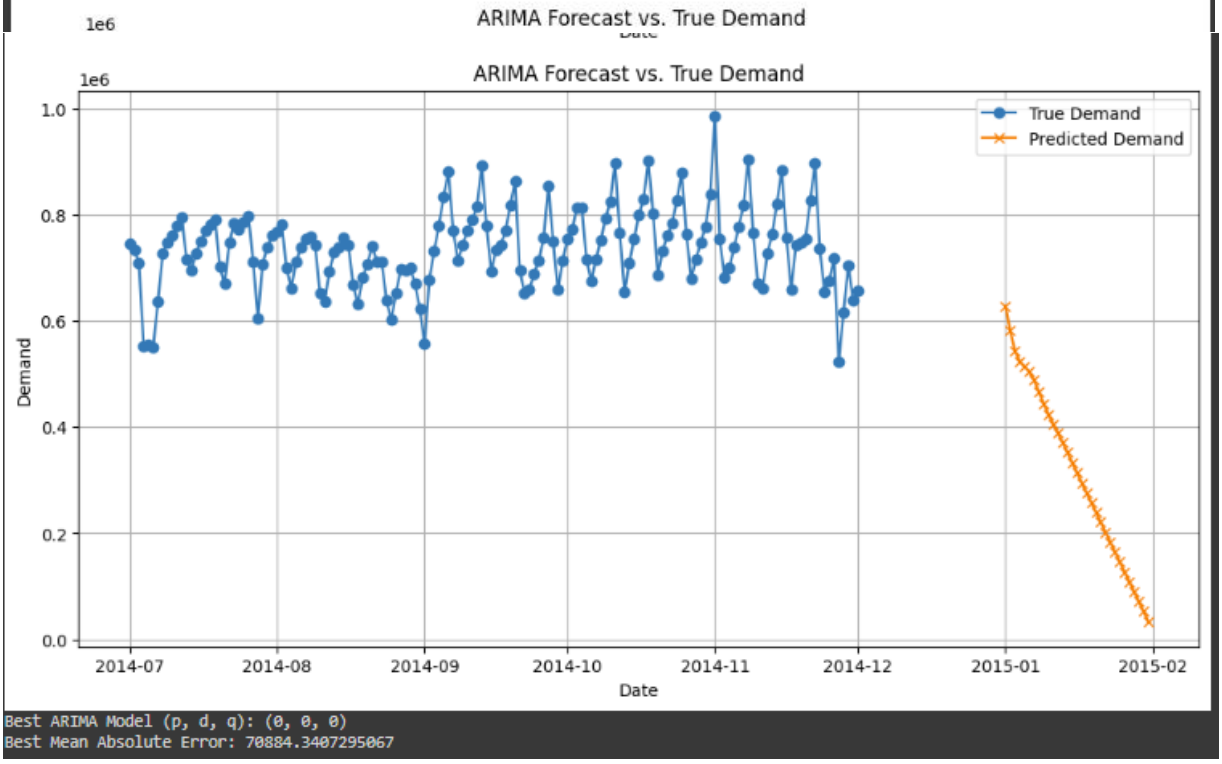
- The code above uses ACF (AutoCorrelation Function) and PACF (Partial AutoCorrelation Function) plots to determine the orders (p, d, q) for time series modeling. They help identify the lag values that show significant autocorrelation and partial autocorrelation, for selecting appropriate parameters in models like ARIMA.
- Alternative Solutions for determining model orders include using automated techniques like grid search or information criteria (AIC, BIC) to find the best-fitting model.
- Also the ACF and PACF plots are a widely accepted visual tool for identifying potential values of p and q for ARIMA modelling and is often optimal choice for this purpose.

```
df_day_ARIMA = df_day_temp.copy(deep=True)
df_day_ARIMA.head()#----- Data Check----#
```

timestamp	value
2014-07-01	745967
2014-07-02	733640
2014-07-03	710142

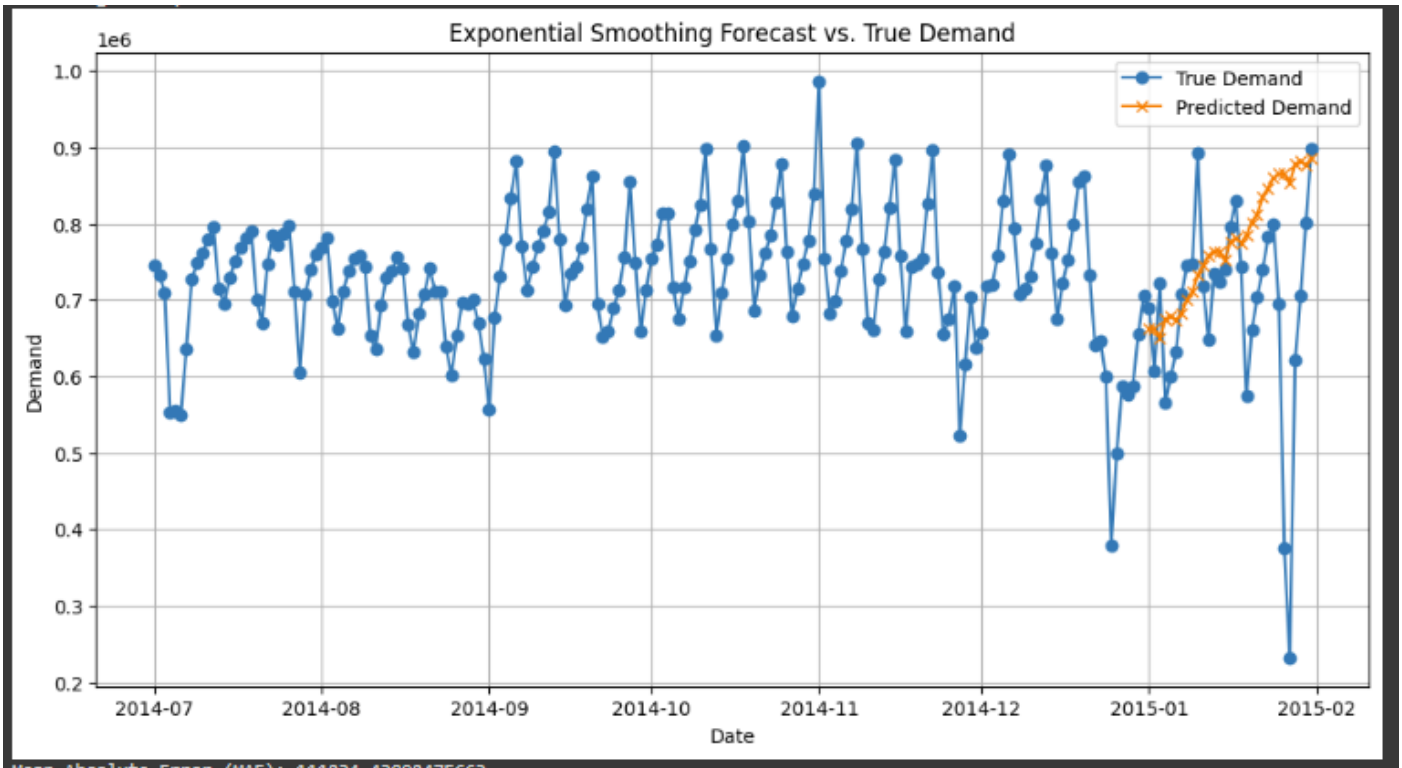
```
#----- Define the range of p, d, q values----#
p_values = [0, 1, 2]
d_values = [0, 1, 2]
q_values = [0, 1, 2]
#----- Possible combinations of p, d, q values----#
param_combinations = [(p, d, q) for p in p_values for d in d_values for q in q_values]
data = df_day_ARIMA #----Data load----#
#-----Define the date range ----#
start_date = datetime(2014, 7, 1)
end_date = datetime(2014, 12, 1)
data = data[start_date:end_date]
best_mae = float('inf')
best_model = None
best_order = None

for p, d, q in param_combinations:
    try:
        model = auto_arima(data, start_p=p, d=d, start_q=q, max_p=2, max_d=2, max_q=2,
                           seasonal=False, trace=False, error_action='ignore', suppress_warnings=True)
        model.fit(data)

        #---Forecast of Jan 2015----#
        start_date = datetime(2015, 1, 1)
        end_date = datetime(2015, 1, 31)
        forecast = model.predict(n_periods=(end_date - start_date).days + 1, return_conf_int=False)

        #----Plot the forecast values and true demand values----#
        plt.figure(figsize=(12, 6))
        plt.plot(data.index, data['value'], label='True Demand', marker='o')
        plt.plot(pd.date_range(start=start_date, end=end_date, freq='D'), forecast, label='Predicted Demand', marker='x')
        plt.xlabel('Date')
        plt.ylabel('Demand')
        plt.title('ARIMA Forecast vs. True Demand')
        plt.legend()
        plt.grid(True)
        plt.show()

        #----Calculate MAE for this model----#
        mae = meanabs(data['value'][-len(forecast):], forecast)

        #---Check if this model has a lower MAE----#
        if mae < best_mae:
            best_mae = mae
            best_model = model
            best_order = (p, d, q)

    except Exception as e:
        continue
print(f"Best ARIMA Model (p, d, q): {best_order}")
print(f"Best Mean Absolute Error: {best_mae}")
```





- The code above iterates through different combinations of p, d, and q values for ARIMA modeling to find the best model with the lowest Mean Absolute Error (MAE) for selecting the most suitable parameters for time series forecasting.
- Alternative Solutions would be Grid search and optimization techniques like AIC or BIC could be used to find optimal parameters.
- The solution provided is practical for finding the best ARIMA model within the specified parameter range. It automatically tests various combinations, for parameter selection.

```python
data = df_day_ARIMA #----Data load----#


#---Define the date range ----#
start_date = datetime(2014, 7, 1)
end_date = datetime(2014, 12, 1)
train_data = data[start_date:end_date]

#----Training Exponential Smoothing model----#
model = ExponentialSmoothing(train_data['value'], trend='add', seasonal='add', seasonal_periods=12)
model_fit = model.fit(optimized=True)

#---Make forecast for Jan 2015----#
start_date = datetime(2015, 1, 1)
end_date = datetime(2015, 1, 31)
forecast = model_fit.forecast(steps=(end_date - start_date).days + 1)

#----Plot the forecast values and true demand values----#
plt.figure(figsize=(12, 6))
plt.plot(data.index, data['value'], label='True Demand', marker='o')
plt.plot(pd.date_range(start=start_date, end=end_date, freq='D'), forecast, label='Predicted Demand', marker='x')
plt.xlabel('Date')
plt.ylabel('Demand')
plt.title('Exponential Smoothing Forecast vs. True Demand')
plt.legend()
plt.grid(True)
plt.show()

#----Calculate MAE for the forecast----#
mae = meanabs(data['value'][-len(forecast):], forecast)
print(f"Mean Absolute Error (MAE): {mae}")
```




```
Mean Absolute Error (MAE): 111234.43892475663
```

Th code above uses the Exponential Smoothing model for forecastinginstead of ARIMA, I chose to use this method because it's a well-established method for time series forecasting and can capture both trend and seasonality, which are common patterns in demand data.

There are several other time series forecasting models apart from Exponential Smoothing.

1. Prophet
2. SARIMA (Seasonal ARIMA).

Q2.3 In this question, we will detect the anomaly within the df_day dataframe.

• Create the Weekday column according to the timestamp column in df_day dataframe. The value in Weekday column should be from ['Monday', 'Tuesday', 'Wednesday', 'Thursday','Friday', 'Saturday', 'Sunday']. Also create the Hour, Day, Month, Year, Month_day (numeric format on day of the month), Lag (yesterday's demand value), and Rolling_Mean (rolling 7 days mean demand value, minimized period is 1) 7 new columns in df_day dataframe according to the timestamp column.

• Using Isolation Forest with above crafted features in df_day to find out the date which is identified as 'outlier'.

```
#----Create a 'Weekday' column ----#
df_day['Weekday'] = df_day['timestamp'].dt.strftime('%A')

#-----Create 'Hour', 'Day', 'Month', and 'Year' columns----#
df_day['Hour'] = df_day['timestamp'].dt.hour
df_day['Day'] = df_day['timestamp'].dt.day
df_day['Month'] = df_day['timestamp'].dt.month
df_day['Year'] = df_day['timestamp'].dt.year

#-----Create a 'Month_day' column----#
df_day['Month_day'] = df_day['timestamp'].dt.day

#-----Create a 'Lag' column ----#
df_day['Lag'] = df_day['value'].shift(1)

#-----Create a 'Rolling_Mean' column ----#
df_day['Rolling_Mean'] = df_day['value'].rolling(window=7, min_periods=1).mean()
```

```
             value  timestamp    Weekday  Hour  Day  Month  Year  Month_day  \
timestamp
2014-07-01  745967 2014-07-01    Tuesday     0    1      7  2014          1
2014-07-02  733640 2014-07-02  Wednesday     0    2      7  2014          2
2014-07-03  710142 2014-07-03   Thursday     0    3      7  2014          3
2014-07-04  552565 2014-07-04     Friday     0    4      7  2014          4
2014-07-05  555470 2014-07-05   Saturday     0    5      7  2014          5
...            ...        ...        ...   ...  ...    ...   ...        ...
2015-01-27  232058 2015-01-27    Tuesday     0   27      1  2015         27
2015-01-28  621483 2015-01-28  Wednesday     0   28      1  2015         28
2015-01-29  704935 2015-01-29   Thursday     0   29      1  2015         29
2015-01-30  800478 2015-01-30     Friday     0   30      1  2015         30
2015-01-31  897719 2015-01-31   Saturday     0   31      1  2015         31

                Lag   Rolling_Mean
timestamp
2014-07-01      NaN  745967.000000
2014-07-02  745967.0  739803.500000
2014-07-03  733640.0  729916.333333
2014-07-04  710142.0  685578.500000
2014-07-05  552565.0  659556.800000
...              ...            ...
2015-01-27  375311.0  617971.285714
2015-01-28  232058.0  606190.857143
2015-01-29  621483.0  601270.714286
2015-01-30  704935.0  603860.714286
2015-01-31  800478.0  618035.142857

[215 rows x 10 columns]
```

```
#-----Data Load----#
features = df_day[['value', 'Hour', 'Day', 'Month', 'Year', 'Month_day', 'Lag', 'Rolling_Mean']]

features = features.fillna(0) #----- Replacing NaN values----#

features.head()#----- Data Check----#
```

timestamp	value	Hour	Day	Month	Year	Month_day	Lag	Rolling_Mean
2014-07-01	745967	0	1	7	2014	1	0.0	745967.000000
2014-07-02	733640	0	2	7	2014	2	745967.0	739803.500000
2014-07-03	710142	0	3	7	2014	3	733640.0	729916.333333
2014-07-04	552565	0	4	7	2014	4	710142.0	685578.500000
2014-07-05	555470	0	5	7	2014	5	552565.0	659556.800000

```
#---- Isolation Forest model----#
iso_forest = IsolationForest(contamination='auto', random_state=42)
iso_forest.fit(features)
outliers = iso_forest.predict(features) #-----Predict outliers (-1) and inliers (1)----#
df_day['Outlier'] = outliers
outlier_dates = df_day[df_day['Outlier'] == -1]['timestamp']#-----Find the dates identified as outliers----#

#-----Print the dates identified as outliers----#
print("Outlier Dates:")
print(outlier_dates)
```

```
Outlier Dates:
timestamp
2014-07-01   2014-07-01
2014-07-04   2014-07-04
2014-07-05   2014-07-05
2014-07-06   2014-07-06
2014-07-07   2014-07-07
2014-07-31   2014-07-31
2014-08-31   2014-08-31
2014-09-01   2014-09-01
2014-09-02   2014-09-02
2014-10-31   2014-10-31
2014-11-01   2014-11-01
2014-11-02   2014-11-02
2014-11-28   2014-11-28
2014-11-29   2014-11-29
2014-11-30   2014-11-30
2014-12-01   2014-12-01
2014-12-02   2014-12-02
2014-12-03   2014-12-03
2014-12-25   2014-12-25
2014-12-26   2014-12-26
2014-12-27   2014-12-27
2014-12-28   2014-12-28
2014-12-29   2014-12-29
2014-12-30   2014-12-30
2014-12-31   2014-12-31
2015-01-01   2015-01-01
2015-01-02   2015-01-02
2015-01-03   2015-01-03
2015-01-04   2015-01-04
2015-01-05   2015-01-05
2015-01-06   2015-01-06
2015-01-07   2015-01-07
2015-01-08   2015-01-08
2015-01-09   2015-01-09
2015-01-10   2015-01-10
2015-01-11   2015-01-11
2015-01-12   2015-01-12
2015-01-13   2015-01-13
2015-01-14   2015-01-14
2015-01-15   2015-01-15
2015-01-16   2015-01-16
2015-01-17   2015-01-17
2015-01-18   2015-01-18
2015-01-19   2015-01-19
2015-01-20   2015-01-20
2015-01-21   2015-01-21
2015-01-22   2015-01-22
2015-01-23   2015-01-23
2015-01-24   2015-01-24
2015-01-25   2015-01-25
2015-01-26   2015-01-26
2015-01-27   2015-01-27
2015-01-28   2015-01-28
2015-01-29   2015-01-29
2015-01-30   2015-01-30
2015-01-31   2015-01-31
```

- The code above is to create the additional columns (Weekday, Hour, Day, Month, Year, Month_day, Lag, Rolling_Mean) in the `df_day` dataframe. These features can help us understand the data better and identify patterns.
- The 'Weekday' column categorizes days of the week, 'Hour' extracts the hour of the day, and 'Day,' 'Month,' and 'Year' break down the date. 'Month_day' represents the day of the month, 'Lag' captures the previous day's demand value, and 'Rolling_Mean' calculates the rolling mean over a 7-day window.
- An alternative solution would be to use other anomaly detection algorithms such as One-Class SVM or Local Outlier Factor (LOF) instead of Isolation Forest.
- The solution provided is a reasonable approach for crafting relevant time-related features and using Isolation Forest for anomaly detection.

References

1. Pandas: https://pandas.pydata.org/pandas-docs/stable/index.html accessed on 23 September 2023
2. NumPy: https://numpy.org/doc/ accessed on 23 September 2023
3. Seaborn: https://seaborn.pydata.org/ accessed on 23 September 2023
4. Requests: https://docs.python-requests.org/en/latest/ accessed on 25 September 2023
5. Matplotlib: https://matplotlib.org/stable/contents.html accessed on 28 September 2023
6. Statsmodels: https://www.statsmodels.org/stable/index.html accessed on 30 September 2023
7. Scikit-Learn (sklearn): https://scikit-learn.org/stable/documentation.html accessed on 28 September 2023
8. Wordcloud: https://github.com/amueller/word_cloud accessed on 29 September 2023
9. pmdarima: https://alkaline-ml.com/pmdarima/ accessed on 02 October 2023
10. DateTime: https://docs.python.org/3/library/datetime.html accessed on 04 October 2023
11. ExponentialSmoothing(statsmodels.tsa.holtwinters): accessed on 06 October 2023 https://www.statsmodels.org/stable/generated/statsmodels.tsa.holtwinters.ExponentialSmoothing.html

What have you learned with your team members from the second assignment?

- From the second assignment, our team learned several valuable lessons in data analysis and time series forecasting.
- We enhanced our data preprocessing skills, including handling missing values, calculating descriptive statistics, and creating new features for analysis.
- We gained practical experience in visualizing data through various plots and charts, which helped us interpret and communicate our findings effectively.
- In the time series analysis part, we worked on ARIMA modeling, autocorrelation, and the importance of selecting the right parameters.
- We also explored alternative models like Exponential Smoothing for forecasting. Additionally, we learned how to assess stationarity and handle anomalies in time series data.
- Overall, the assignment improved our data analysis, visualization, and modeling capabilities, fostering collaboration and knowledge sharing within our team.

What is the contribution of each team member for finishing the second assignment?

- Part 1 and Part 2 questions were worked out by Saurabh and Vamsi accordingly and Baswaraj has coordinated the peer review, documentation and code standardization and formatting.
- All three of us have discussed on all questions to decide the approach for each question and on alternate options to address the questions.
- We have collaborated on the inputs to the code, report and video to ensure that we present the best and mutually agreed solution to the group assignment.