SIG742

Modern Data Science

Task 2 Assessment Report

Arunkumar Balaraman & Shravan Kumar Kasagoni
S223919051 & S223912075

Contents

Part I

Introduction:

Dataset captures listings a range of items, including apparel, electronics, and tech products, each with varying conditions. Every listing is distinctively identified using *train_id*, ID helps us identify the item's name, state, category, brand, pricing, shipping details and a clean description. The *category_name* column stands out with its complexity, segmenting each item into precise categories and subcategories making way for detailed analysis. This dataset offers a comprehensive exploration into the evolving trends, distinct patterns, and unique characteristics of the listings, illuminating insights on consumer inclinations, brand significance, and predominant market tendencies.

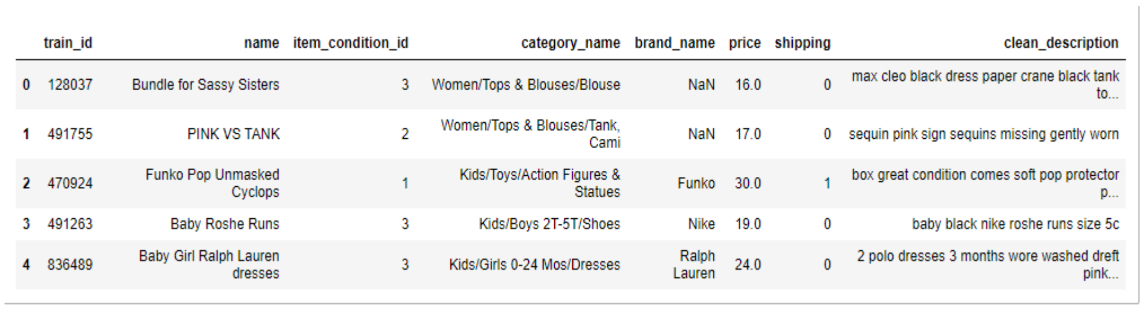
Data Acquisition and Pre-processing

Code:

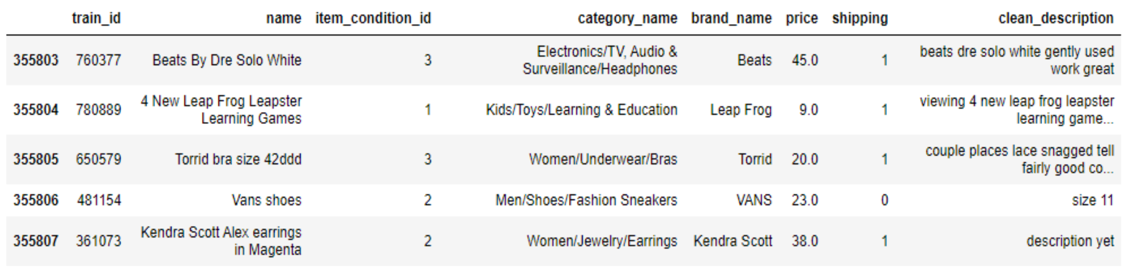
```
# Importing all necessary libraries
import os
import zipfile
import urllib.request
import pandas as pd
import matplotlib.pyplot as plt
from wordcloud import WordCloud
import numpy as np
import seaborn as sns
import random

sns.set(color_codes=True)
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)


# Function to download, UnZip & Read File
def dl_uz_rd(url, destination):
    """
    Use: Downloads, unzips, and reads a file from a given URL.

    Args:
        - url (str): The URL of the file to be downloaded.
        - destination (str): The local path where the downloaded file should
be saved.

    Returns:
        - pd.DataFrame: Data read from the unzipped file.
    """
    # Download the file
    urllib.request.urlretrieve(url, destination)

    # Unzip the file
    with zipfile.ZipFile(destination, 'r') as zip_ref:
        zip_ref.extractall(os.getcwd())

        # Unzip file is stored in list variable
        extracted_files = zip_ref.namelist()

        # Read the Unzip file to dataframe
        data = pd.read_csv(extracted_files[0])

    return data


# # Defined the URL from the Problem Statement
#               url               =               'https://github.com/tulip-
lab/sit742/raw/fbd1bb363bc63511ff8895148b4d50f787efbe3f/Jupyter/data/item_li
sting_category.zip'

# #Destination File name
# destination = 'item_listing_category.zip'

# # Read the dataframe from the function
# df = dl_uz_rd(url, destination)

df = pd.read_csv('item_listing_category.csv')

# First 5 records of the dataframe
df.head()

# Last 5 records of the dataframe
df.tail()

# Info of the dataframe
df.info()

# Shape of the dataframe
print(f"The dataset has {df.shape[0]} rows and {df.shape[1]} columns before
removing duplicates.")

# Printing the no of duplicates records
```

```
print(f"There are {df.duplicated().sum()} duplicate records in the dataset.
Proceeding to delete them...")

# Deleting the duplicates from the dataset
df.drop_duplicates(inplace=True,keep='first')

# Printing the shape of dataset after removing duplicates
print(f"The dataset has {df.shape[0]} rows and {df.shape[1]} columns after
removing duplicates.")

# Printing count of unique rows for each columns
df.nunique()
```

Results



	train_id	name	item_condition_id	category_name	brand_name	price	shipping	clean_description
0	128037	Bundle for Sassy Sisters	3	Women/Tops & Blouses/Blouse	NaN	16.0	0	max cleo black dress paper crane black tank to...
1	491755	PINK VS TANK	2	Women/Tops & Blouses/Tank, Cami	NaN	17.0	0	sequin pink sign sequins missing gently worn
2	470924	Funko Pop Unmasked Cyclops	1	Kids/Toys/Action Figures & Statues	Funko	30.0	1	box great condition comes soft pop protector p...
3	491263	Baby Roshe Runs	3	Kids/Boys 2T-5T/Shoes	Nike	19.0	0	baby black nike roshe runs size 5c
4	836489	Baby Girl Ralph Lauren dresses	3	Kids/Girls 0-24 Mos/Dresses	Ralph Lauren	24.0	0	2 polo dresses 3 months wore washed dreft pink...



	train_id	name	item_condition_id	category_name	brand_name	price	shipping	clean_description
355803	760377	Beats By Dre Solo White	3	Electronics/TV, Audio & Surveillance/Headphones	Beats	45.0	1	beats dre solo white gently used work great
355804	780889	4 New Leap Frog Leapster Learning Games	1	Kids/Toys/Learning & Education	Leap Frog	9.0	1	viewing 4 new leap frog leapster learning game...
355805	650579	Torrid bra size 42ddd	3	Women/Underwear/Bras	Torrid	20.0	1	couple places lace snagged tell fairly good co...
355806	481154	Vans shoes	2	Men/Shoes/Fashion Sneakers	VANS	23.0	0	size 11
355807	361073	Kendra Scott Alex earrings in Magenta	2	Women/Jewelry/Earrings	Kendra Scott	38.0	1	description yet

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 355808 entries, 0 to 355807
Data columns (total 8 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   train_id           355808 non-null  int64
 1   name               355808 non-null  object
 2   item_condition_id  355808 non-null  int64
 3   category_name      354269 non-null  object
 4   brand_name         203852 non-null  object
 5   price              355808 non-null  float64
 6   shipping           355808 non-null  int64
 7   clean_description  355614 non-null  object
dtypes: float64(1), int64(3), object(4)
```

```
memory usage: 21.7+ MB

The dataset has 355808 rows and 8 columns before removing duplicates.

There are 48572 duplicate records in the dataset. Proceeding to delete them...
The dataset has 307236 rows and 8 columns after removing duplicates.

train_id             307236
name                 277067
item_condition_id         5
category_name          1135
brand_name             3046
price                   545
shipping                  2
clean_description    267826
dtype: int64
```

Observations:

- The dataset has 355808 rows and 8 columns before removing duplicates.
- There are 48572 duplicate records in the dataset & deleted.
- The dataset has 307236 rows and 8 columns after removing duplicates.
- Columns have missing values in category_name, brand_name, and clean_description.

Column Details:

Column Name	Description
train_id	An identifier for each listing.
name	The name or title of the listing.
item_condition_id	An identifier representing the condition of the item & higher values indicating worse conditions.
category_name	The category to which the listed item belongs further divided into subcategories. For instance, 'Women/Tops & Blouses/Blouse' suggests that the primary category is 'Women', with subcategories 'Tops & Blouses' and 'Blouse'.
brand_name	The brand of the item listed.
price	The price at which the item is listed.
shipping	A binary flag indicating whether shipping is included (1) or not (0).
clean_description	A brief description of the item, possibly preprocessed for analysis.

Question 1.1

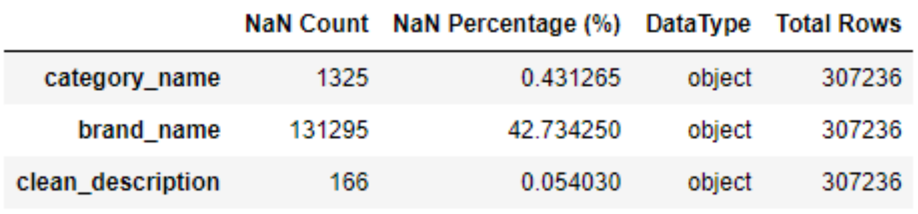
Find the missing values:

- 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 particular columns? (if you think it could not be done with mean value, write down the reason in comments and report rather than code)

Code:

```
def missing_values_table(df):
    """
    Use: Summary of missing values in a DataFrame.

    Args:
        - df (pd.DataFrame): Input DataFrame.

    Returns:
        - pd.DataFrame: DataFrame showing the count and percentage of missing
values for each column with missing values, along with their data type and
the total number of rows in the input DataFrame.
    """

    # Calculate the count of NaN (missing) values for each column
    null_df = df.isna().sum()

    # Filter out columns that don't have any missing values
    filtered_columns = null_df[null_df > 0].index

    # Construct a DataFrame
    data = pd.DataFrame({
        'NaN            Count':            df[filtered_columns].isna().sum(),
# Count of missing values
        'NaN Percentage (%)': (df[filtered_columns].isna().sum() / len(df))
* 100,  # Percentage of missing values
        'DataType': df[filtered_columns].dtypes,                             #
Data type of the column
        'Total Rows': len(df)                                                #
Total number of rows in the input DataFrame
    })

    return data

# Calling the function
missing_values_table(df)
```

Results:



	NaN Count	NaN Percentage (%)	DataType	Total Rows
category_name	1325	0.431265	object	307236
brand_name	131295	42.734250	object	307236
clean_description	166	0.054030	object	307236

Observations and Methodology:

A Python function, missing_values_table, was written to:

- Compute the count of missing values for each column in the provided DataFrame.
- Filter and display only those columns that have missing values.
- Present the count and percentage of missing values, the data type of the column, and the total number of rows in the DataFrame.
- The function was then called on the dataset to get the results.

Columns with missing values in the dataset are:

- category_name: 1325 missing values (0.43% of the total rows) and data type object.
- brand_name: 131,295 missing values (42.73% of the total rows) and data type object.
- clean_description: 166 missing values (0.05% of the total rows) and data type object.

All these columns have data type object indicates they contain text & imputing missing values with a mean is not appropriate for these columns.

For categorical columns, common imputation strategies are:

- Filling with the most frequent value (mode).
- Using a placeholder value like "Unknown" or "Not Available".

Logic Explanation:

Why decided this solution: Written a function to give a clear overview of missing values both in count and percentage for a comprehensive understanding.

Any Other Solutions: Visualizing missing values with heatmaps or bar charts could be an alternative approach.

Solution is Optimal or not: The solution efficiently identifies missing values. Text columns aren't suited for mean imputation. Instead, using mode or placeholders like "Unknown" is recommended.

Question 1.2

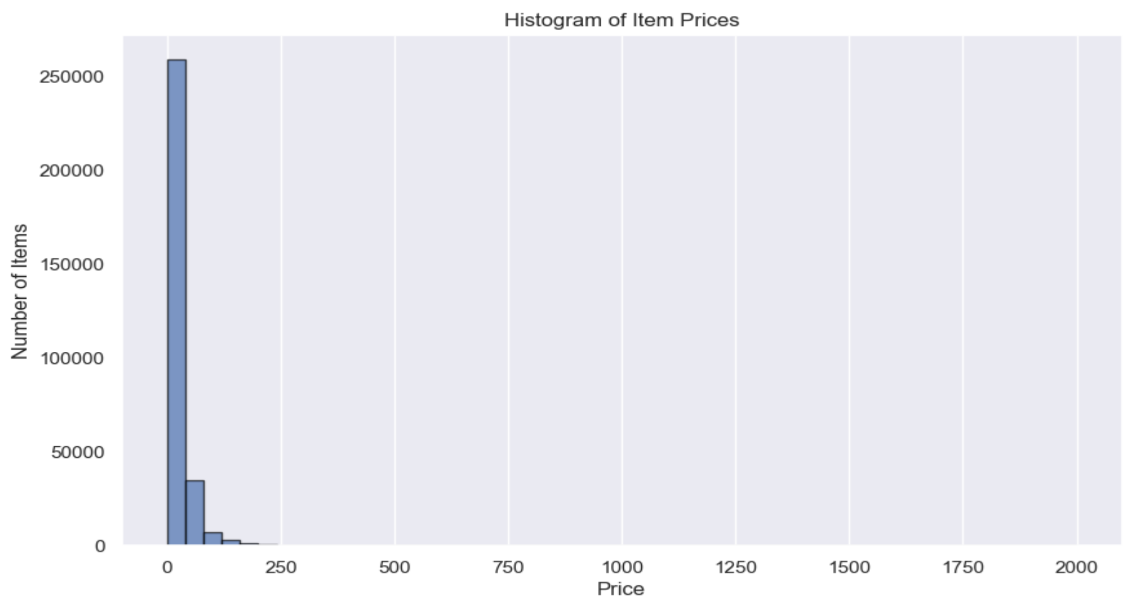
Find the price information from the data:

- 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.

Code:

```python
def plt_hist_md_90(df):

    """
    Use: Print median and 90th percentile price & plots histogram of the 'price' column

    Args:
        - df (pd.DataFrame): Input DataFrame

    Displays:
        - Print values for the median and 90th percentile of 'price' column.
        - Histogram plot of the 'price' column.
    """

    # Calculating the median and 90th percentile
    median_price = df['price'].median()
    percentile_90_price = df['price'].quantile(0.9)

    # Median and 90th percentile values
    print(f"Median Price: {median_price:.2f}")
    print(f"90th Percentile Price: {percentile_90_price:.2f}")

    # Plotting the histogram for the 'price' column with 50 bins
    plt.figure(figsize=(10, 6))
    plt.hist(df['price'], bins=50, edgecolor='black', alpha=0.7)
    plt.title('Histogram of Item Prices')
    plt.xlabel('Price')
    plt.ylabel('Number of Items')
    plt.grid(axis='y')
    plt.show()

# Calling the function
plt_hist_md_90(df)
```

Results:



Observations and Methodology:

A Python function, plt_hist_md_90, was written to:

- Compute the median and 90th percentile values of the 'price' column.
- Display a histogram showcasing the distribution of item prices.

The function was subsequently invoked on the dataset to generate the results.

- **Median Price:** The median price is 17.00 indicates half of the items are priced at or below 17.00 and the other half are priced above 17.00.
- **90th Percentile Price:** The price at 90th percentile is 51.00 indicates that 90% of the items are priced at or below 51.00 and only 10% of the items exceed this price.
- The histogram is **right-skewed**, indicating most items are priced lower.
- A significant concentration exists below 20.
- Few items have higher prices, with 90% under 51.

Logic Explanation:

Why decided this solution: Chose a function to compute and visualize price data. The median provides a central value, the 90th percentile gives insight into higher-priced items, and the histogram offers a distribution overview.

Any Other Solutions: Could have used box plots to visualize price distribution and outliers or descriptive statistics for a broader overview of price data.

Solution is Optimal or not: The solution effectively captures key price insights. The histogram clearly shows the price distribution, and the calculated values (median and 90th percentile) offer actionable insights about item pricing.

Question 1.3

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)

Assumption from Program Manager:

- To solve Q 1.3, please use the column 'shipping', and identify buyers and sellers based on the binary division (0/1). Please state your rationale clearly while doing so as to be consistent with the definitions chosen.

Code:

```
# Average price based on shipping
average_prices = df.groupby('shipping')['price'].mean()
average_prices
```

Results:

```
shipping
0    29.937665
1    22.543669
Name: price, dtype: float64
```

Observations:

- Items with a 'shipping' value of 0 tend to have a higher average price than those with a 'shipping' value of 1.
- This indicates:
  - For 'shipping' value 0: The price likely includes the shipping cost, indicating the **seller pays for shipping**.
  - For 'shipping' value 1: The price is exclusive of shipping cost, suggesting the **buyer pays for shippin**.

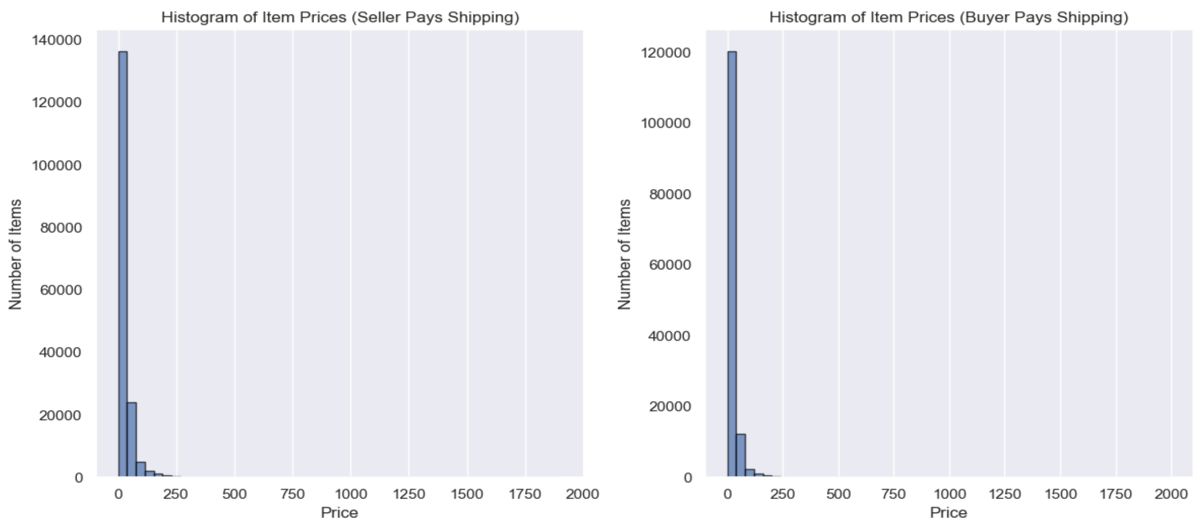
Code:

```


def shipping_analysis(df):
    """
    Use: Analyze and visualize the distribution of item prices based on who
pays for shipping.
```

```
    Args:
        - df (pd.DataFrame): Input DataFrame

    Returns:

        - tuple: A tuple containing:
            * Percentage of items where buyers pay for shipping
            * Median prices for both when the seller pays for shipping and
when the buyer pays.

    Displays:
        - Percentage of items where the buyer pays for shipping.
        - Median prices based on who pays for shipping.
        - Histograms showing the distribution of item prices based on who
pays for shipping.
    """

    # Calculate the percentage of items that are paid by the buyers
    buyer_pays_percentage = len(df.query('shipping == 1')) / len(df)
    seller_pays_percentage = 1 - buyer_pays_percentage
    print(f"Percentage of items where buyers pay for shipping:
{buyer_pays_percentage * 100:.2f}%")
    print(f"Percentage of items where Seller pay for shipping:
{seller_pays_percentage * 100:.2f}%")


    # Calculate the median prices based on the 'shipping' value
    median_prices = df.groupby('shipping')['price'].median()
    print(f"\nMedian price when the seller pays for shipping:
{median_prices[0]:.2f}")
    print(f"Median price when the buyer pays for shipping:
{median_prices[1]:.2f}")


    # Plot histograms for item prices based on shipping values
    fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12, 6))

    # 'shipping' value 0
    ax[0].hist(df.query('shipping == 0')['price'], bins=50,
edgecolor='black', alpha=0.7)
    ax[0].set_title('Histogram of Item Prices (Seller Pays Shipping)')
    ax[0].set_xlabel('Price')
    ax[0].set_ylabel('Number of Items')
    ax[0].grid(axis='y')

    # 'shipping' value 1
    ax[1].hist(df.query('shipping      ==      1')['price'],      bins=50,
edgecolor='black', alpha=0.7)
    ax[1].set_title('Histogram of Item Prices (Buyer Pays Shipping)')
    ax[1].set_xlabel('Price')
    ax[1].set_ylabel('Number of Items')
    ax[1].grid(axis='y')

    plt.tight_layout()
    plt.show()

    return buyer_pays_percentage, (median_prices[0], median_prices[1])


# Calling the custom function
bpp, (mps,mpb) = shipping_analysis(df)
```

Results:

```
Percentage of items where buyers pay for shipping: 44.65%
Percentage of items where Seller pay for shipping: 55.35%

Median price when the seller pays for shipping: 19.00
Median price when the buyer pays for shipping: 14.00
```



Observations and Methodology:

Identifying Shipping Costs Bearer:
Utilize the 'shipping' column to differentiate between items where the buyer pays for shipping (value = 1) and items where the seller pays for shipping (value = 0).

Computing Price Statistics:
Use Python functions to calculate and visualize the median prices and distributions based on who bears the shipping costs.

Buyer Shipping Payment:

Buyers paid the shipping cost for **44.65%** of the items. While sellers cover these costs for rest.

Median Prices based on who pays for shipping:

Median price when the **seller** pays for shipping: **19.00 Median price** when the **buyer** pays for shipping: **14.00**

This indicates when the buyer bears the shipping costs tend to have the lower priced than those where the seller covers the shipping.

Histogram Observations:

- Both histograms are **right-skewed** indicating that the majority of items are priced lower with fewer items having higher prices.

- Distribution of items when the seller pays for shipping has slightly higher concentration in the mid-price range compared to the distribution where the buyer pays for shipping. Its a consistent observation that items where sellers pay for shipping have a higher median price.

Logic Explanation:

Why decided this solution: Opted for a function to compute and visualize shipping data. Using the ‘shipping’ column, we discerned who pays for shipping and then analyzed the price distribution. This approach provides a clear understanding of the relationship between shipping costs and item prices.

Any Other Solutions: Could have used box plots to visualize price distribution based on who pays for shipping or descriptive statistics for a broader overview of price data.

Solution is Optimal or not: The solution effectively captures key insights about shipping costs and impact on item prices. Histograms and calculated values (median prices) actionable insights about pricing strategy based on who pays for shipping.

Question 1.4

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

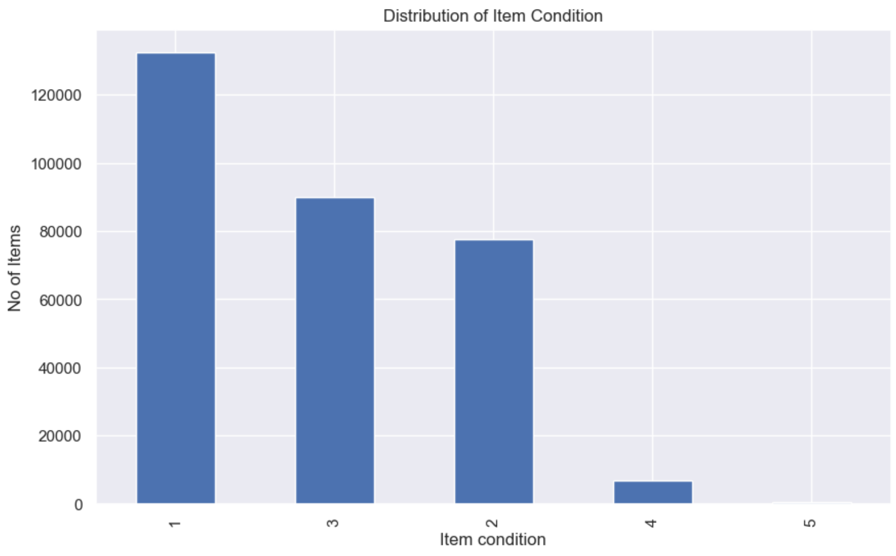
- Write the code to find out (print) the count of the rows on each number (value) in column item_condition_id.
- Draw the boxplot graphs (one plot) on the price for each item condition value, and find out out whether the better condition of the item could have higher median price (draw the plot and answer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

Code:

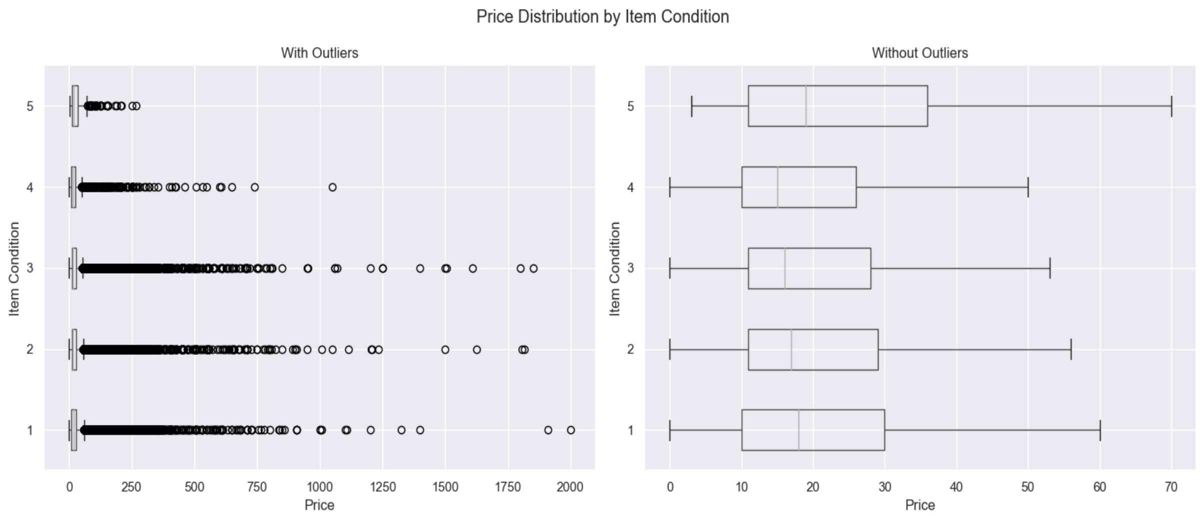
```
def item_condition(df):
    """
    Use: Visualize the distribution of items condition and price distribution
for each item condition.

    Args:
        - df (pd.DataFrame): Input DataFrame

    Displays:
        - Bar chart showing the distribution of items by their condition.
        - Boxplot displaying the price distribution for each item condition.
    """

    # Printing the count of items for each condition
    print(df['item_condition_id'].value_counts())

    # Plotting a bar chart for item conditions
    plt.figure(figsize=(10, 6))
    col                                                                      =
df['item_condition_id'].value_counts().plot.bar(title='Distribution  of  Item
Condition', xlabel='Item condition', ylabel='No of Items')
    plt.show()

    # Printing the median price for each condition
    print(f"Median      prices      based      on      each      condition
\n\n{df.groupby('item_condition_id')['price'].median()}")

    # Plotting boxplots for price distribution by item condition
    fig, ax = plt.subplots(1, 2, figsize=(15, 6))

    # Boxplot with outliers
    df.boxplot(column='price', by='item_condition_id', ax=ax[0], grid=True,
vert=False)
    ax[0].set_title('With Outliers')
    ax[0].set_xlabel('Price')
    ax[0].set_ylabel('Item Condition')

    # Boxplot without outliers
    df.boxplot(column='price', by='item_condition_id', ax=ax[1], grid=True,
vert=False, showfliers=False)
    ax[1].set_title('Without Outliers')
    ax[1].set_xlabel('Price')
    ax[1].set_ylabel('Item Condition')

    # Overall title and layout adjustment
    plt.suptitle('Price Distribution by Item Condition')
    plt.tight_layout()
    plt.show()


# Calling the custom function
item_condition(df)
```

Results:

```
item_condition_id
1    132492
3     89904
2     77666
4      6705
5       469
Name: count, dtype: int64
```



```
Median prices based on each condition

item_condition_id
1    18.0
2    17.0
3    16.0
4    15.0
5    19.0
Name: price, dtype: float64
```



Observations and Methodology:

Identifying Item Conditions:
Extract the distribution of items based on their 'item_condition_id' values. Lower values represent better conditions.

Price Statistics by Item Condition:
Compute and visualize the median prices and distributions based on item condition using boxplots.

Distribution of Item condition

Condition 1 (Best Condition): 132,492 items. Majority of items are in the best condition. Condition 2: 77,666 items & most third frequent conditions. Condition 3: 89,904 items. Most second frequent conditions. Condition 4: 6,705 items & drops significantly Condition 5 (poor Condition): 469 items & Very few items are categorized as 5.

Price based on Item Condition:

Median prices based on each condition

item_condition_id	
1	18.0
2	17.0
3	16.0
4	15.0
5	19.0

Boxplots provide a visual representation of the distribution of item prices across different conditions

- Median prices are relatively consistent across all conditions but with Best Condition its slightly higer than 2, 3 & 4. However Condition 5 median value is slightly higher than of best condition.
- Expectation is best condition (1) should have higher median prices but condition 5 has higher median price this may be due to the below factors

- Items Category: The items in condition 5 despite being in the poor condition might be more valuable or rare items.
- Sample Size: The size of the samples is very few in condition 5 compared to condition 1.

Logic Explanation:

Why decided this solution: Written a function to compute and visualize item conditions and their relation to prices. The bar chart provides a clear distribution of items by condition, while the boxplots offer insights into price variations based on condition.

Any Other Solutions: Could have used histograms to visualize price distribution for each condition or descriptive statistics for a broader overview of price data.

Solution is Optimal or not: The solution effectively captures insights about item conditions and their impact on prices. The boxplots and calculated median values offer actionable insights about pricing strategy based on item condition.

Question 1.5

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).

Code:

```
# Printing the no of unique categories
print(f"Number of unique categories: {df['category_name'].nunique()}")

# Printing top 3 categories for items in the Poor condition
print(f"\n\nTop 3 categories for items with the poor
condition:\n\n{df.query('item_condition_id ==
5')['category_name'].value_counts().head(3)}")

# Printing top 3 categories for items in the Best condition
print(f"\n\nTop 3 categories for items with the Best
condition:\n\n{df.query('item_condition_id ==
1')['category_name'].value_counts().head(3)}")
```

Results:

```
Number of unique categories: 1135
```

```
Top 3 categories for items with the poor condition:

category_name
Electronics/Cell Phones & Accessories/Cell Phones & Smartphones    115
Electronics/Video Games & Consoles/Games                            37
Electronics/Video Games & Consoles/Consoles                         31
Name: count, dtype: int64


Top 3 categories for items with the Best condition:

category_name
Women/Athletic Apparel/Pants, Tights, Leggings    6408
Beauty/Makeup/Lips                                4843
Beauty/Makeup/Face                                4393
Name: count, dtype: int64
```

Observations and Methodology:

Unique Categories Identification:

- Determine the number of unique categories present in the dataset.

Category Distribution by Item Condition:

- Extract the most frequent categories for items in the best and worst conditions to understand any potential trends or patterns.

Top 3 categories for items with the **poor** condition:

Electronics/Cell Phones & Accessories/Cell Phones & Smartphones	115
Electronics/Video Games & Consoles/Games	37
Electronics/Video Games & Consoles/Consoles	31

Top 3 categories for items with the **Best** condition:

Women/Athletic Apparel/Pants, Tights, Leggings	6408
Beauty/Makeup/Lips	4843
Beauty/Makeup/Face	4393

Conclusion:

- The top 3 categories of both poor and best categories aligns with our previous finding on median price: electronic items be it a smartphones, games or consoles even in poor condition will have a higher value due to its initial cost, brand and functionality.
- Apparel and beauty products, even in the best condition will be priced lower compared to gadgets.
- Item condition plays role in pricing & also category of an item will significantly influences its condition and price dynamics.

Logic Explanation:

Why decided this solution: Written function to compute and visualize the distribution of items across categories and their conditions. This approach helps in understanding the influence of item categories on their condition and subsequently, their pricing.

Any Other Solutions: Could have used pie charts to visualize the distribution of top categories for each condition or heatmap to see concentration of items across various categories.

Solution is Optimal or not: Solution effectively captures insights about item categories and their conditions. The observation that electronic items, even in poor condition, have higher value due to their inherent value, brand, and functionality is insightful. Conversely, apparel and beauty products, even in the best condition, are priced lower.

Question 1.6

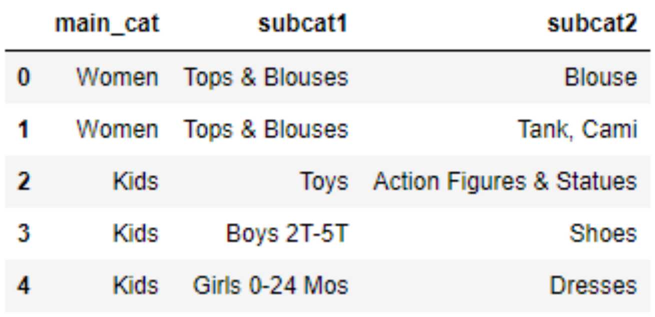
The categories in column category_name have 3 parts. The three parts (main_cat,subcat_1 and subcat_2) are concatenated with ’/’ character sequentially in the data now.

- 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.

Code:

```
def extract_category(category):
    """
    Use: Extracts the main_cat, subcat1, and subcat2 from the category.

    Args:
        - category (str): The category string to be split.

    Returns:
        - tuple: A tuple containing the main category, subcategory 1, and
subcategory 2.
    """

    # If category is NaN or None, return 'Category Unknown' for all three
segments
    if pd.isna(category) or category is None:
        return ('Category Unknown', 'Category Unknown', 'Category Unknown')

    # Splitting the category based on the '/' delimiter
    segments = category.split('/')
    # If there are more than three segments, combine the extra segments into
'subcat2'
    while len(segments) > 3:
        segments[2] = segments[2] + ' ' + segments.pop(3)

    # Filling missing segments with 'Category Unknown'
    while len(segments) < 3:
        segments.append('Category Unknown')

    return tuple(segments)


# Calling extract_category function to create the new columns
df['main_cat'],          df['subcat1'],          df['subcat2']          =
zip(*df['category_name'].apply(extract_category))

# Displaying the first 5 rows of the new columns
df[['main_cat', 'subcat1', 'subcat2']].head()
```

Results:



	main_cat	subcat1	subcat2
0	Women	Tops & Blouses	Blouse
1	Women	Tops & Blouses	Tank, Cami
2	Kids	Toys	Action Figures & Statues
3	Kids	Boys 2T-5T	Shoes
4	Kids	Girls 0-24 Mos	Dresses

Observations and Methodology:

A Python function, extract_category, was written to:

- Handle cases where the category is NaN or None, assigning 'Category Unknown' to all three segments in such cases.
- Split the category based on the '/' delimiter.
- Address scenarios with more than three segments by merging extra segments into 'subcat2'.
- Populate missing segments with 'Category Unknown'.

Applied to Dataset: The extract_category function was applied to the 'category_name' column, resulting in the creation of three new columns: 'main_cat', 'subcat1' and 'subcat2'.

- If the input category is NaN or None then returns 'Category Unknown' for all three segments.
- Splitting the category based on the '/' delimiter

- If there are more than three segments, combine the extra segments into 'subcat2' as we seen more than 2 "/" delimiter in data "Electronics/Computers & Tablets/iPad/Tablet/eBook Readers"
- Filling missing segments with 'Category Unknown'

Logic Explanation:

Why decided this solution: Written function to handle the diverse scenarios in the ‘category_name’ column. This approach ensures that categories are split correctly, even when there are missing values or when categories have more than three segments.

Any Other Solutions: Could have used Python's built-in string methods or regular expressions directly on the DataFrame. However, this might not handle edge cases as effectively as the custom function.

Solution is Optimal or not: Solution is optimal for the given problem. It effectively handles NaN values, standard category splits, and edge cases where categories have more than three segments. The resulting columns provide a clear breakdown of the main category and its subcategories.

Question 1.7

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.

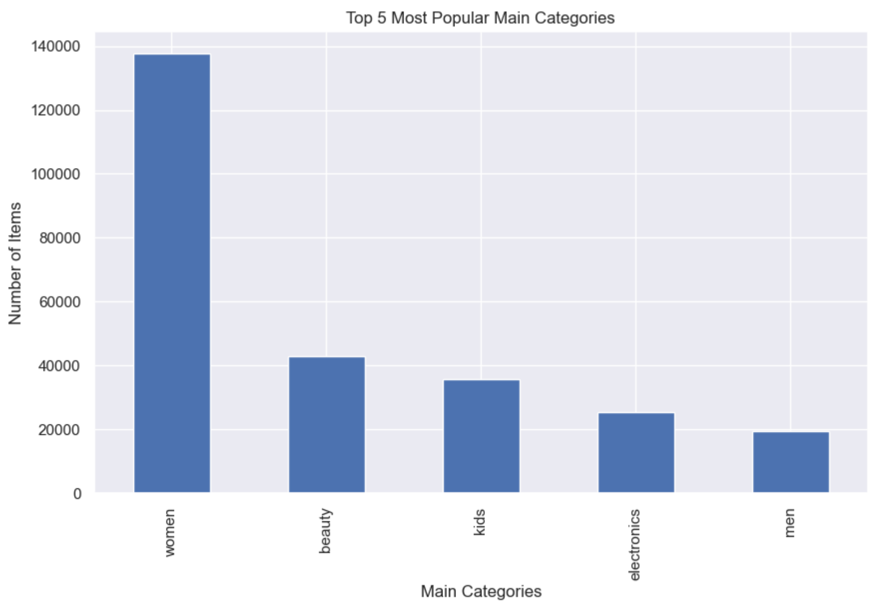
- Write code (or function) to change the text (value in each row) from the new three columns to lowercase.
- Draw the bar chart to find out the top 5 most popular main categories (in column main_cat) in the data (only showing the top 5).
- Write code (or function) to (print) find out how many unique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

Code:

```
# Function to convert the text in the new columns to lowercase
def lcase_freq_plt(df):
    """
    Use: Converts the text in the specified columns to lowercase, plots the top 5 most popular main categories,
        and prints the number of unique categories in the main_cat, subcat1, and subcat2 columns.

    Args:
        - df (pd.DataFrame): Input DataFrame with columns 'main_cat', 'subcat1', 'subcat2'.

    Returns:
        - pd.DataFrame: DataFrame with text converted to lowercase in the specified columns.
    """

    # Converting the text into lower case for the main_cat, subcat1 and
subcat2
    df['main_cat'] = df['main_cat'].str.lower()
    df['subcat1'] = df['subcat1'].str.lower()
    df['subcat2'] = df['subcat2'].str.lower()

    #Plotting top 5 most popular main categories from new column
    df['main_cat'].value_counts().head(5).plot.bar(figsize                =
(10,6),xlabel='Main Categories',ylabel='Number of Items',title='Top 5 Most
Popular Main Categories')

    # Printing the unique value count of the each of the new column main_cat,
subcat1 and subcat2
    print(f"No of Unique Main Categories: {df['main_cat'].nunique()}")
    print(f"No of Unique Sub Categories 1: {df['subcat1'].nunique()}")
    print(f"No of Unique Sub Categories 2: {df['subcat2'].nunique()}")

    return df

# Applying the function to the dataframe
df = lcase_freq_plt(df)
```

Results:

```
No of Unique Main Categories: 11
No of Unique Sub Categories 1: 114
No of Unique Sub Categories 2: 789
```



Observations and Methodology:

Text Normalization:

- Developed and apply a Python function, lcase_freq_plt, to:
- Convert the text in 'main_cat', 'subcat1', and 'subcat2' columns to lowercase for uniformity.

Top 5 Main Categories:

The most popular main category is **women** followed by **beauty, kids, electronics and men.**

- No of Unique **Main Categories**: **11**
- No of Unique **Sub Categories 1**: **114**
- No of Unique **Sub Categories 2**: **789**

Logic Explanation:

Why decided this solution: Written function to handle the text normalization and visualization tasks. This approach ensures that the text is uniformly converted to lowercase across all new category columns and provides a clear visualization of the most popular main categories.

Any Other Solutions: Could have used separate functions or direct DataFrame operations for text normalization and visualization. However, combining these tasks into a single function streamlines the process.

Solution is Optimal or not: It effectively normalizes the text across the new category columns and provides a clear bar chart of the top 5 main categories. Additionally, it concisely displays number of unique categories across the three new columns.

Question 1.8

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.

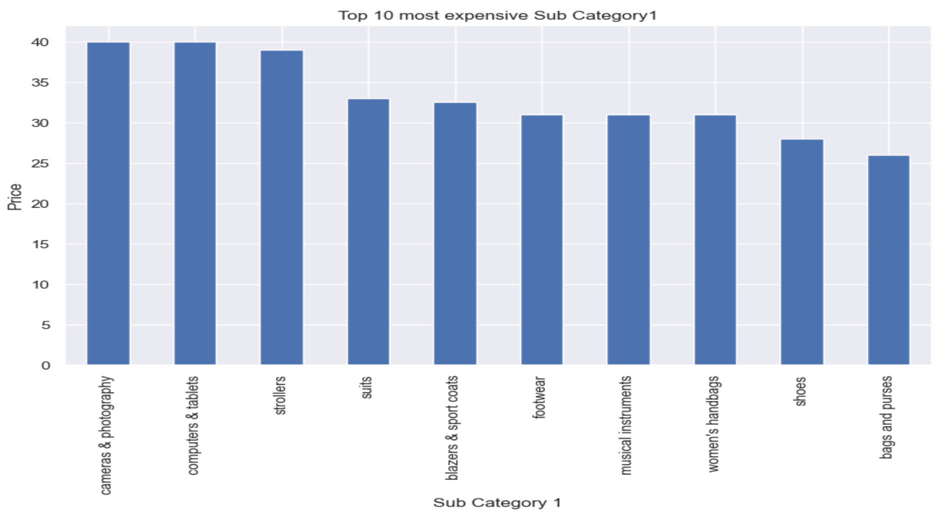
Code:

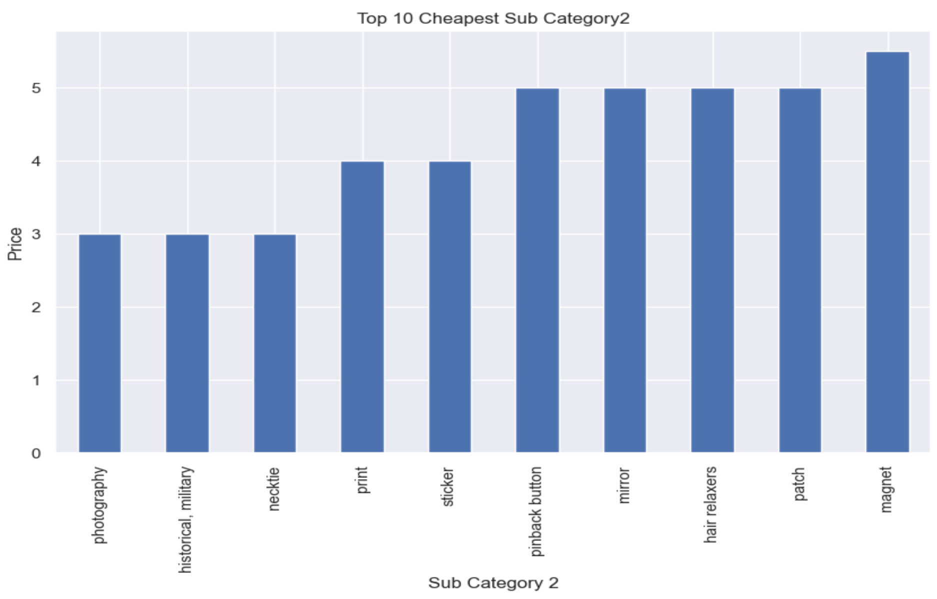
```
# Printing the median price for all main categories
print(f"Median Price of the categories in Main Categories: \n\n{df.groupby('main_cat')['price'].median().sort_values(ascending=False)}\n\n")

# Plotting the top 10 expensive Sub Category 1
subCategory1Top10 = df.groupby('subcat1')['price'].median().sort_values(ascending=False).head(10)
```

```
subCategory1Top10.plot.bar(title='Top 10 most expensive Sub Category1',figsize = (10,6),xlabel='Sub Category 1',ylabel='Price')
plt.show()

# Space Holder
print('\n\n\n')

# Plotting the 10 Cheapest Sub Category 2
subCategory2Bottom10 = df.groupby('subcat2')['price'].median().sort_values().head(10)

subCategory2Bottom10.plot.bar(title='Top 10 Cheapest Sub Category2',figsize = (10,6),xlabel='Sub Category 2',ylabel='Price')
plt.show()
```

Results:

```
Median Price of the categories in Main Categories:

main_cat
men                         21.0
women                       19.0
category unknown            18.0
home                        18.0
sports & outdoors           16.0
vintage & collectibles      16.0
beauty                      15.0
electronics                 15.0
kids                        14.0
other                       14.0
handmade                    11.0
Name: price, dtype: float64
```





Observations and Methodology:

Main Categories Analysis:

- Calculate the median price for all main categories.

Sub Category Analysis:

- Identify the top 10 most expensive first-level subcategories.
- Determine the top 10 cheapest second-level subcategories.
- The Men category has the highest median price at 21.00 followed by Women and Home categories with median prices of 19 and 18 respectively.
- The Handmade category has the lowest median price of 11.

```
men                       21.0
women               19.0
category unknown    18.0
home                18.0
sports & outdoors   16.0
vintage & collectibles   16.0
beauty              15.0
electronics         15.0
kids                14.0
other                     14.0
handmade            11.0
```

Expensive Sub Category 1 & Cheapest Sub Category 2

Plotted top 10 expensive sub category 1 and camera's and Photography is most expensive Sub Category 1 with median price of 40.00 followed by Computers & Tablets with median price of 39.5

Plotted Cheapest 10 Sub category 2 and historical, military, necktie & photography has the cheapest sub category median price of 3.00.

Question 1.9

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.

Code:

```
# Filling NaN with brand unavailable using inplace to replace in the
dataset.
df['brand_name'].fillna('brand unavailable',inplace=True)

# Displaying all brand names with the median price in decending order.
df.groupby('brand_name')['price'].median().sort_values(ascending=False)

# Read top 10 popular brands into a variable
popbrd = df['brand_name'].value_counts().head(10)

# Printing the top 10 popular brands to have the observations written
print(popbrd)

# Plotting top 10 popular brands
popbrd.plot.bar(title='Top      10      Popular      Brands',figsize=(10,6),
xlabel='Brands', ylabel='Items Counts')
```

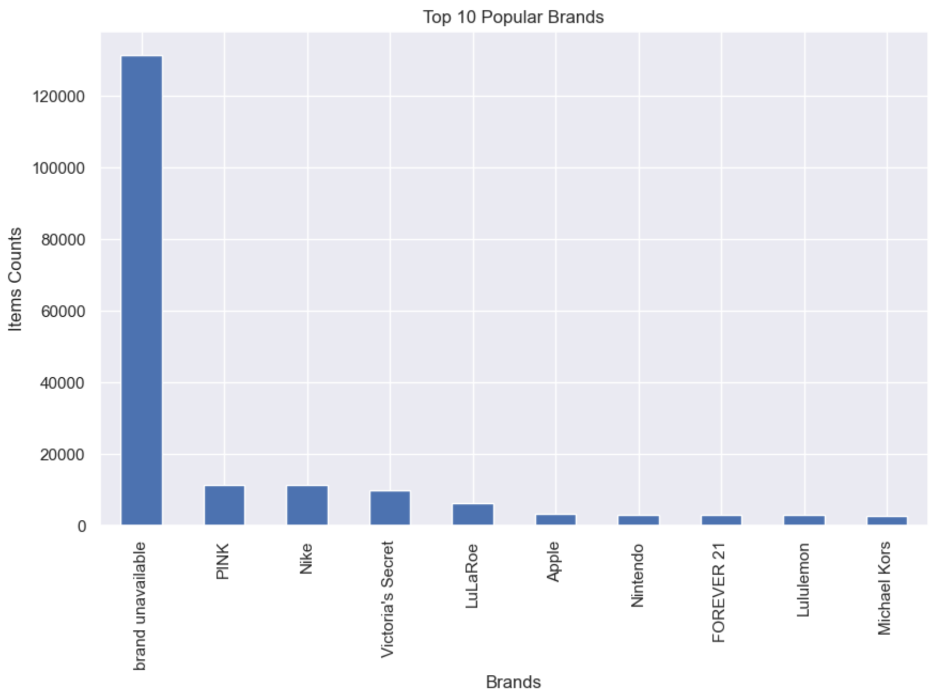
Results:

```
brand_name
Tiffany Designs                           359.0
Stuart Weitzman                           329.0
Blendtec                                  280.0
IBM                                       275.0
MICHELE                                   254.0
Lanvin                                    246.0
Escort Radar                              242.5
Tag Heuer                                 237.5
3.1 Phillip Lim                           232.5
AMD                                       230.0
Frédérique Constant                       224.0
GoPro                                     223.0
David Yurman                              212.0
EVGA                                      211.0
Contours                                  207.5
Alyce Paris                               200.0
```

```
Breitling                                   200.0
Terani Couture                              199.0
Omega                                       196.5
Mori Lee                                    194.5
Mackintosh                                  190.0
Moncler                                     190.0
A Wish Come True                            189.0
Gigabyte                                    186.0
...
Jinx                                          3.0
Toys R Us Plush                               3.0
Clover Canyon                                 3.0
Chamilia                                      0.0
Name: price, dtype: float64
```

Output is truncated. View as a scrollable element or open in a text editor.

```
brand_name
brand unavailable    131295
PINK                  11438
Nike                  11326
Victoria's Secret      9962
LuLaRoe                6326
Apple                  3432
Nintendo               3154
FOREVER 21             3138
Lululemon              3041
Michael Kors           2901
Name: count, dtype: int64
```



Observations and Methodology:

Brand Price Analysis:

- Calculate the median price for all brands.
- Fill missing brand names with 'brand unavailable'.

Popularity Analysis:

- Identify the top 10 most popular brands based on their listings.

Visualization:

- Use a bar chart to display the top 10 most popular brands.

Brands Price Distribution:

Top 5 Expensive Brands:

Tiffany Designs:	359.0
Stuart Weitzman:	329.0
Blendtec:	280.0
IBM:	275.0
MICHELE:	254.0

Cheapest 5 Brands:

New York Color:	3.0
Jinx:	3.0
Toys R Us Plush:	3.0
Clover Canyon:	3.0
Chamilia:	0.0

Popular Brand:

- Significant number of items 131,295 do not have a brand associated & has been updated as brand unavilable.
- PINK and Nike are the most popular branded items, with 11,438 and 11,326 listings respectively.
- Following closely are Victoria's Secret and LuLaRoe with 9,962 and 6,326 listings respectively.
- Apple and Nintendo tech products are in top 10 popular brands make the list with 3,432 and 3,154 listings respectively.
- The diversity in the top 10 brands, from clothing to tech indicates a wide range of products available on the platform

Logic Explanation:

Why decided this solution: Our approach efficiently identifies brand distribution and their median prices. Filling missing brands ensures data completeness, and visualizing top brands highlights popularity.

Any Other Solutions: An alternative could be to directly visualize median prices of top brands or segment brands by product type.

My Solution is Optimal or not: Solution effectively captures brand landscape. However, more nuanced insights might emerge from further segmentation.

Question 1.10

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 wordcould by using the column clean_description on each quantile of price data.

Code:

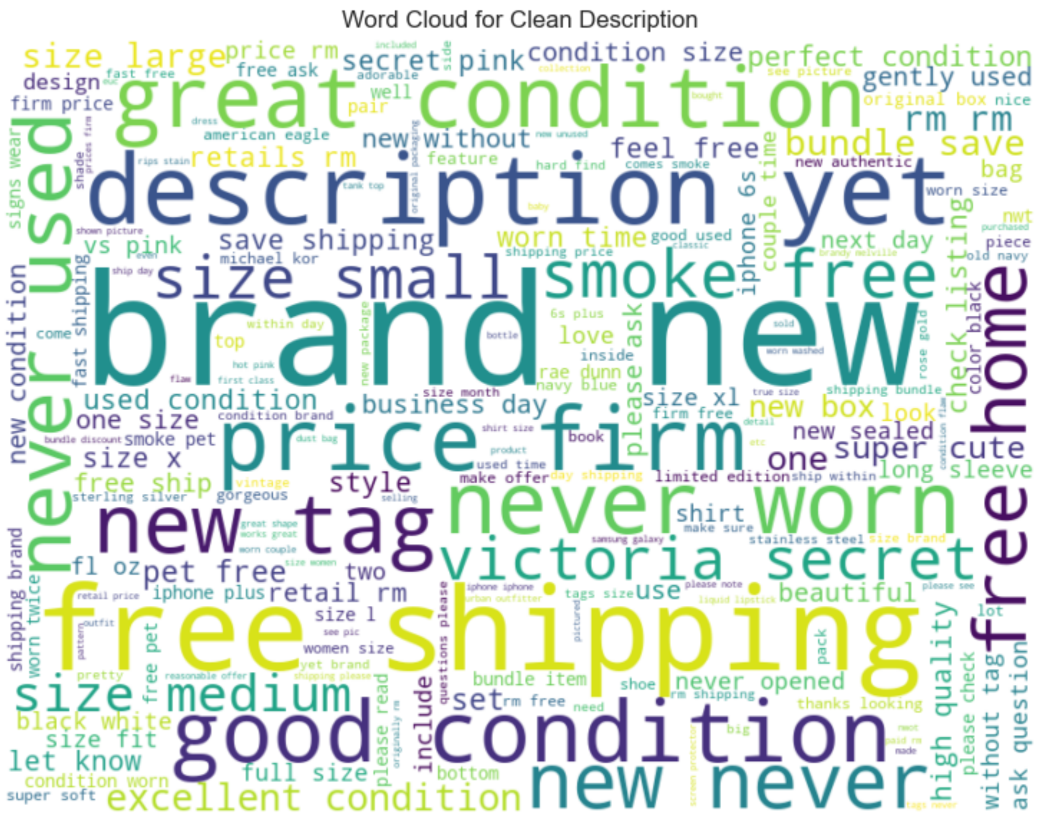
```
# As we seen the null in the column, checking again for nulls and also printing most frequent occurances to fill NA.
print(f"No of Null in column Clean Description: {df['clean_description'].isna().sum()}")


df['clean_description'].value_counts().head()

# FIling NA with description yet
df['clean_description'].fillna('description yet',inplace=True)

#Checking for NA after its fill na
print(f"No of Null in column Clean Description: {df['clean_description'].isna().sum()}")


# Combine all the descriptions into one string
text = ' '.join(df['clean_description'].dropna())

# Generate the word cloud
wordcloud = WordCloud(background_color='white', width=800, height=600).generate(text)

# Plot the word cloud
plt.figure(figsize=(10, 7))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.title('Word Cloud for Clean Description')
plt.show()
```

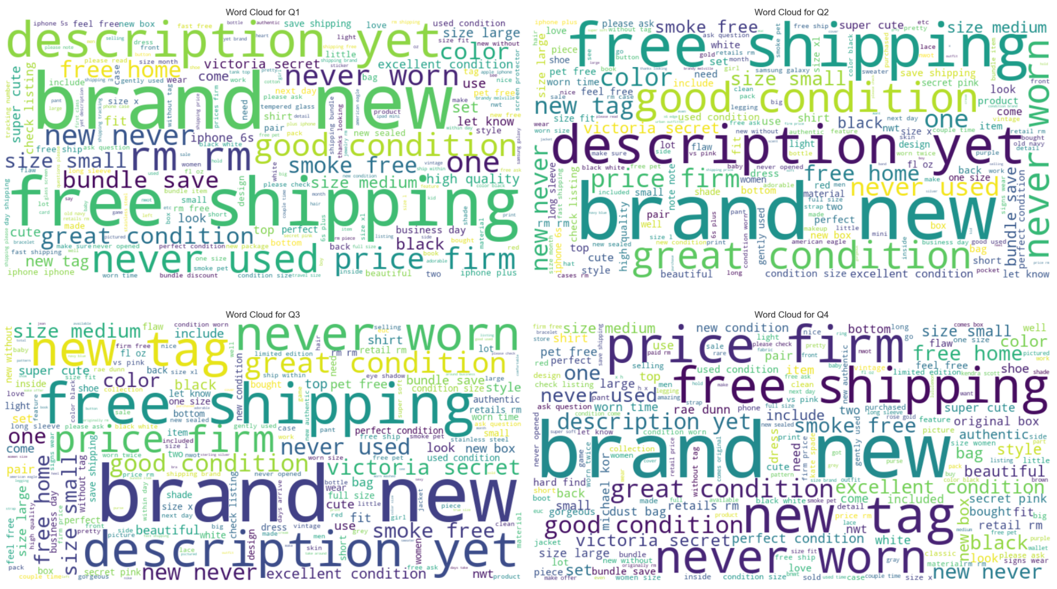
```
# Creating quantile column with Q1 to Q4
df['price_quantile'] = pd.qcut(df['price'], q=4, labels=["Q1", "Q2", "Q3", "Q4"])

# Word clouds for each quantile using Subplot
fig, axes = plt.subplots(2, 2, figsize=(20, 12))
axes = axes.ravel()

# Iterate through each quantile create the text for each quantile
for i, quantile in enumerate(["Q1", "Q2", "Q3", "Q4"]):

    # Extract descriptions for the current quantile
    text = ' '.join(df[df['price_quantile'] == quantile]['clean_description'].astype(str).tolist())

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

    # Plot the word cloud
    axes[i].imshow(wordcloud, interpolation='bilinear')
    axes[i].axis('off')
    axes[i].set_title(f'Word Cloud for {quantile}')

plt.tight_layout()
plt.show()

df.groupby('price_quantile')['price'].median()
```

Results:

No of Null in column Clean Description: 166

```
clean_description

description yet    17104
brand new           1108
new                 1095
good condition       607
great condition      478
Name: count, dtype: int64

No of Null in column Clean Description: 0
```





```
price_quantile

Q1     8.0
Q2    14.0
Q3    22.0
Q4    45.0
Name: price, dtype: float64
```

Observations and Methodology:

Word Cloud for the Entire Dataset:

Plotted word cloud column clean_description using entire dataset.

Words like "size", "brand", "new", "free", "shipping", and "condition" are among the most common words indicating that sellers often highlight the condition, brand, and size of items & shipping information seems to be a common aspect of descriptions.

Word Cloud by Price Quantiles:

Q1 (Lowest Price Range):

Median Price 8.0

Terms like "free", "new", "size", and "brand" indicates that items in the lowest price bracket are often described as new or brand-free.

Q2:

Median Price 14

Terms like "new", "brand", and "size" still dominate, with a few new words becoming more visible.

Q3:

Median Price 22

"New", "brand", "used", and "size" are still key descriptors. It appears that regardless of the price range, certain descriptive terms remain consistent.

Q4 (Highest Price Range):

Median Price 45

In highest price bracket, words like "new", "brand", and "size" are still prominent indicates that even for more expensive items, sellers emphasize the newness or brand of the product.

Logic Explanation:

Why decided this solution: Solution efficiently visualizes item descriptions using word clouds. Filling missing values with frequent descriptions ensures data consistency, and analyzing by price quantiles offers insights into description trends across price ranges.

Any Other Solutions: Alternative methods could involve advanced NLP techniques like topic modeling or using model-based imputation for missing values instead of the most frequent description.

Solution is Optimal or not: Solution is suitable offering clear visual insights. For deeper analysis, advanced NLP techniques might be more appropriate.

Part 2

Introduction:

The (nyc_taxi.csv) data used for this part could be found in **this link.** You will need to use Pandas to read the csv data for starting.

Time Series Analysis

Question 2.1

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.

Code:

```
# Load the csv into a dataframe
df_nyc_taxi = pd.read_csv('nyc_taxi.csv')

# Aggregate the values on a daily level
df_day = df_nyc_taxi.resample('D', on='timestamp').sum().reset_index()

# Display the aggregated the values on a daily level
df_day.head()

# Aggregate the values on an hourly level
df_hour = df_nyc_taxi.resample('H', on='timestamp').sum().reset_index()

# Display the aggregated the values on an hourly level
df_hour.head()
```

Results:

Aggregate the values on a daily level:

	timestamp	value
0	2014-07-01	745967
1	2014-07-02	733640
2	2014-07-03	710142
3	2014-07-04	552565
4	2014-07-05	555470

Aggregate the values on an hourly level:

	timestamp	value
0	2014-07-01 00:00:00	18971
1	2014-07-01 01:00:00	10866
2	2014-07-01 02:00:00	6693
3	2014-07-01 03:00:00	4433
4	2014-07-01 04:00:00	4379

Logic Explanation:

Why you decide to choose your solution: The data is currently in 30-minute intervals. To aggregate this data, we'll first convert the timestamp column into a datetime object. This allows for easy manipulation and aggregation using pandas. We can then resample the data to daily and hourly intervals.

Are there any other solutions that could solve the question: One could potentially loop through the data and manually sum values for each day/hour, but using pandas' built-in datetime and resampling functionality is more efficient.

Whether your solution is the optimal or not: Resampling using pandas is a standard and efficient approach for time series data, making it an optimal solution for this task.

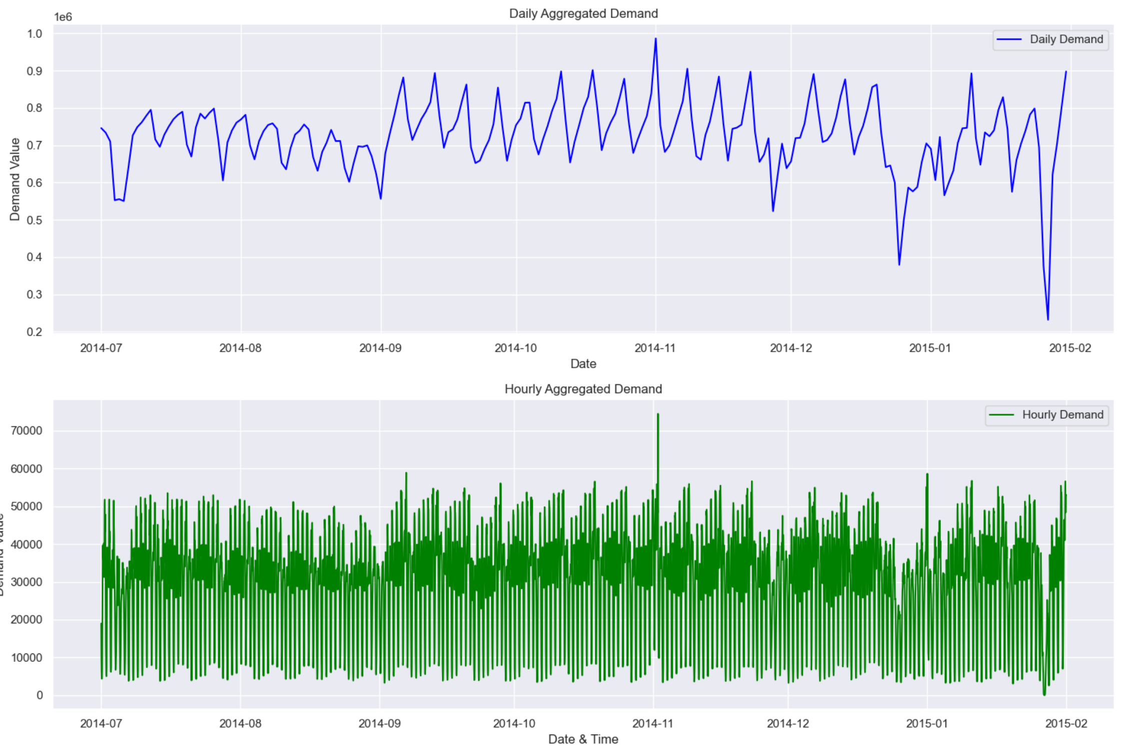
Plot the demand value in two line charts for both df_day and df_hour dataframes.

Code:

```
# Set up the figure and axes
fig, ax = plt.subplots(nrows=2, ncols=1, figsize=(15, 10))

# Plotting the daily aggregated values
ax[0].plot(df_day['timestamp'],  df_day['value'],  label='Daily  Demand',
color='blue')
ax[0].set_title('Daily Aggregated Demand')
ax[0].set_xlabel('Date')
ax[0].set_ylabel('Demand Value')
ax[0].grid(True)
ax[0].legend()

# Plotting the hourly aggregated values
ax[1].plot(df_hour['timestamp'],  df_hour['value'],  label='Hourly  Demand',
color='green')
ax[1].set_title('Hourly Aggregated Demand')
ax[1].set_xlabel('Date & Time')
ax[1].set_ylabel('Demand Value')
ax[1].grid(True)
ax[1].legend()
```

```
# Adjusting the layout
plt.tight_layout()
plt.show()
```

Results:



Observations:

Daily Aggregated Demand:

- **Trend:**
    - The chart shows the demand values aggregated on a daily basis.
    - There is a clear cyclical pattern, which could imply weekly seasonality. For example, certain days of the week might be when demand is consistently higher or lower.
    - The overall trend seems stable, without any drastic upward or downward shifts over the period covered.
- **Variability:**
    - While the chart shows some variability from day to day, the daily aggregation smoothens the intra-day fluctuations, providing a clearer view of the bigger picture.
- **Possible Influences:**

    - Factors such as weekends, holidays, and special events could explain some of the dips and spikes in the daily demand. For instance, the sharp dips observed might correspond to weekends when taxi demand could be lower.

Hourly Aggregated Demand:

- **Trend:**
    - The chart shows demand values aggregated on an hourly basis.
    - At this granular level, the intra-day patterns become evident. There is a clear repetitive pattern every day, which might be due to people's daily routines (e.g., rush hours, off-peak hours).
- **Variability:**
    - The hourly chart has much more variability than the daily chart because it captures the nuances of demand throughout each day.
    - We can observe peaks and troughs within each day. The peaks could be when people commute to work or return home, and the troughs could be during the night when fewer people require taxis.
- **Possible Influences:**
    - The repetitive daily pattern suggests that typical daily routines heavily influence taxi demand. The early morning might surge due to people commuting to work, followed by a dip in the late morning. Another peak might occur in the evening as people return home.
    - Weather, events, or public transit disruptions might influence the variability within days.

In summary, while the daily chart provides a macro view of the demand pattern, potentially highlighting weekly rhythms, the hourly chart offers insights into demand's daily ebb and flow. Both views are valuable, depending on the specific analysis or business decisions.

Logic Explanation:

Why you decide to choose your solution: Line charts are ideal for visualizing time series data as they show the progression of a variable over time.

Are there any other solutions that could solve the question: Other types of plots, like area charts or bar charts, could also be used. However, line charts are typically the most intuitive and clear for this kind of data.

Whether your solution is the optimal or not: Line charts are the standard for visualizing time series data, making them optimal for this task.

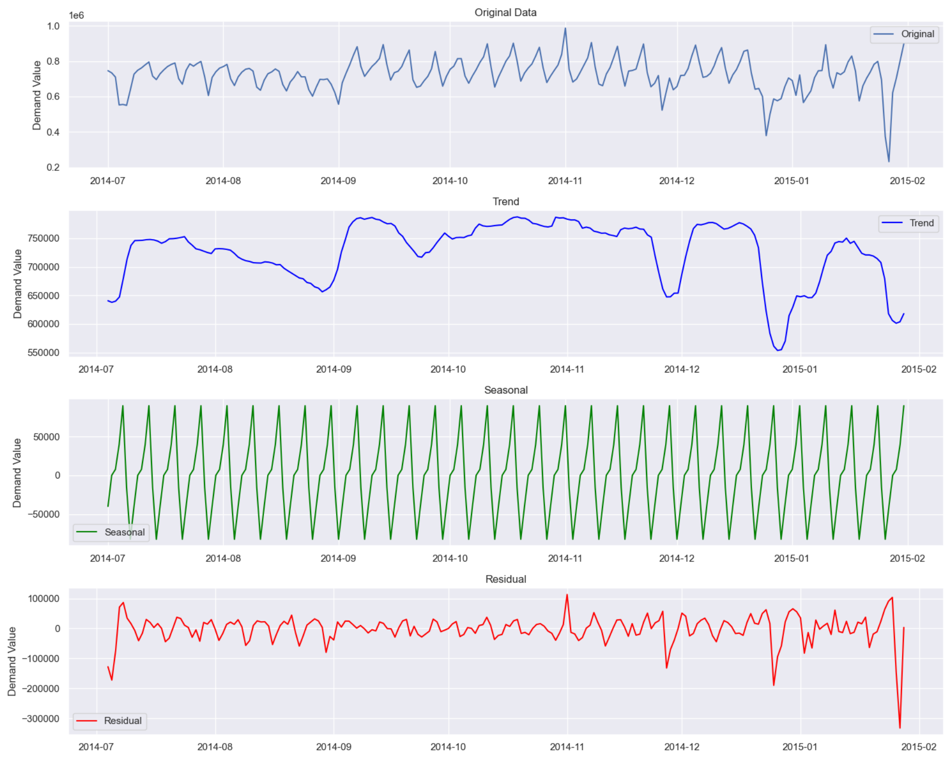
Plot the seasonal decomposition components (Trend, Seasonal, Residual) from df_day dataframe, also find out the p value from adfuller test. Do you think the df_day is stationary enough (please explain your reasons in comments and report)?

Code:

```
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.stattools import adfuller

# Decompose the time series
decomposition   =   seasonal_decompose(df_day['value'],   model='additive',
period=7)  # Weekly seasonality

# Extract the trend, seasonal, and residual components
trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

# Plot the decomposition components
fig, ax = plt.subplots(4, 1, figsize=(15, 12))

# Original data
ax[0].plot(df_day['timestamp'], df_day['value'], label='Original')
ax[0].set_title('Original Data')
ax[0].set_ylabel('Demand Value')
ax[0].legend()

# Trend component
ax[1].plot(df_day['timestamp'], trend, label='Trend', color='blue')
ax[1].set_title('Trend')
ax[1].set_ylabel('Demand Value')
ax[1].legend()

# Seasonal component
ax[2].plot(df_day['timestamp'], seasonal, label='Seasonal', color='green')
ax[2].set_title('Seasonal')
ax[2].set_ylabel('Demand Value')
ax[2].legend()

# Residual component
ax[3].plot(df_day['timestamp'], residual, label='Residual', color='red')
ax[3].set_title('Residual')
ax[3].set_ylabel('Demand Value')
ax[3].legend()

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

```
# Perform ADF test
adf_result = adfuller(df_day['value'])
p_value = adf_result[1]
print("P Value: ", p_value)
```

Results:



P Value: 0.00942459999371752

Observations:

The seasonal decomposition components for the df_day data frame are as follows:

- **Original Data:** This is the daily aggregated demand value.
- **Trend:** This shows the underlying trend in the data, smoothed out from daily fluctuations. It provides a longer-term view of the data, which remains relatively stable throughout the period.
- **Seasonal:** This component captures the repeated seasonal patterns in the data. Given that we used a period of 7 (indicating weekly seasonality), we can see the repeated patterns weekly. This could correspond to changes in demand depending on the day of the week.
- **Residual:** After removing the trend and seasonal components, what is left is the residual. The "noise" or the unpredictable fluctuations in the data cannot be attributed to the trend or seasonality.

Regarding the stationarity of the df_day data frame:

The p-value from the Augmented Dickey-Fuller (ADF) test is approximately (0.0094). A common threshold for the p-value in the ADF test is (0.05). If the p-value is below this threshold, we can reject the null hypothesis and conclude that the series is stationary. With a p-value of (0.0094) below (0.05), we have evidence to reject the null hypothesis, suggesting that the time series is stationary.

Conclusion:

Based on the ADF test, the df_day data frame appears stationary. This means that its statistical properties, like the mean and variance, are constant over time. Stationary time series are easier to model and are prerequisites for many time series forecasting techniques.

Logic Explanation:

Why you decide to choose your solution: The Seasonal Decomposition of Time Series (STL) is a method to decompose a time series into three components: trend, seasonality, and residuals. This helps in understanding the underlying patterns in the data. The Augmented Dickey-Fuller (ADF) test is a common method to check the stationarity of a time series. A stationary time series has constant mean, variance, and autocorrelation over time.

Are there any other solutions that could solve the question: There are other decomposition methods like X-13ARIMA-SEATS, but STL is widely used and straightforward. For stationarity testing, we also have the KPSS test, but the ADF is more commonly used.

Whether your solution is the optimal or not: The STL decomposition combined with the ADF test is a comprehensive and standard approach to understanding and testing time series data, making it optimal for this task.

Question 2.2

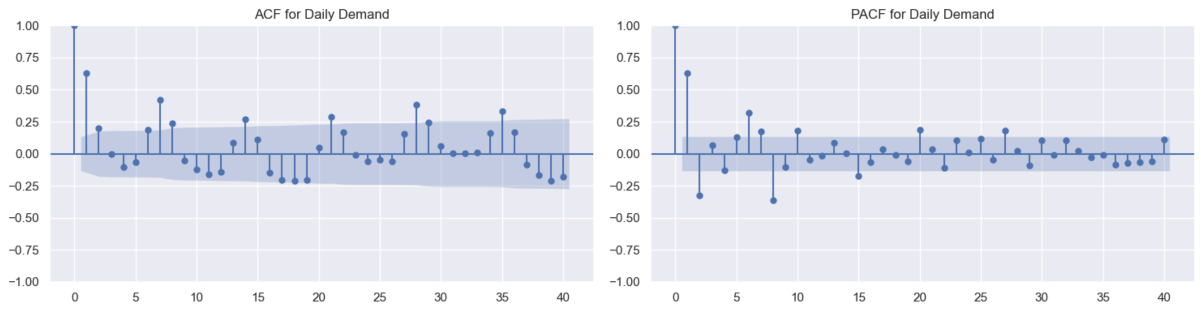
In this question, we will try to use time series model such as ARIMA and others to build the model(s) for forecasting the future.

*Create the **acf** and **pacf** plots for df_day dataframe.*

Code:

```python
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

fig, ax = plt.subplots(1, 2, figsize=(15, 4))

# ACF plot
plot_acf(df_day['value'], ax=ax[0], lags=40, title='ACF for Daily Demand')

# PACF plot
plot_pacf(df_day['value'], ax=ax[1], lags=40, title='PACF for Daily Demand',
method='ywm')

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

Results:



Observations:

The Autocorrelation Function (ACF) and the Partial Autocorrelation Function (PACF) plots provide insight into the time-dependent structure of a time series. These plots are commonly used in time series analysis, especially when identifying the order of an autoregressive (AR) or moving average (MA) process.

ACF Plot:

- It measures the linear relationship between the time series values and its lagged values.
- The spike at lag 7 (and its multiples) is particularly notable, suggesting a solid weekly seasonality in the data.
- The gradual decay in the ACF indicates that there might be an autoregressive component in the data.

PACF Plot:

- It measures the relationship between the time series values and its lagged values after removing the effects of any correlations due to the terms at shorter lags.
- The significant spike at lag 7 suggests a potential autoregressive term of order 7. After that, the PACF values drop off, which can indicate the data’s autoregressive nature.

Conclusion:

Significant lags in the ACF and PACF plots suggest that the data has a time-dependent structure.
The solid weekly seasonality (evident from the spikes at lag 7) might imply that certain days of the week consistently see higher or lower taxi demand.

Logic Explanation:

Why you decide to choose your solution:

- The ACF gives the correlation of the series with its lags. It can be used to identify the possible structure of time series data. If the ACF plot shows a slow decay, it suggests that there's a MA component, whereas a cut-off after a certain number of lags suggests an AR component.

- The PACF, on the other hand, gives the partial correlation of the series with its lags. It can be used to identify the extent of the lag in an autoregressive model. For instance, a sharp cut-off after a certain number of lags in the PACF plot indicates the order of the AR model.

Are there any other solutions that could solve the question: The ACF and PACF are standard tools, other methods like examining the information criterion (like AIC or BIC) of various ARIMA models can be used to determine the order. But this approach is more exhaustive and less intuitive than using ACF and PACF.

Whether your solution is the optimal or not: For visually determining the order of AR or MA terms for an ARIMA model, the ACF and PACF plots are the most straightforward and intuitive tools, making them optimal for this purpose.

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".

Code:

```
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_absolute_error

# Filter the data based on the given date range
data_for_modeling   =   df_day[(df_day['timestamp']   >=   '2014-07-01')   &
(df_day['timestamp'] <= '2014-12-01')]['value']

# List to store results
results = []

# Define the parameter ranges
p_range = [0, 1, 2]
d_range = [0, 1, 2]
q_range = [0, 1, 2]

# Iterating over each combination of p, d, q
for p in p_range:
    for d in d_range:
        for q in q_range:
            try:

                # Define the model
                model = ARIMA(data_for_modeling, order=(p, d, q))

                # Fit the model
                model_fit = model.fit()

                # Predict using the model
                predictions = model_fit.predict()

                # Calculate the MAE
                mae = mean_absolute_error(data_for_modeling, predictions)

                # Append sample results for diagnosis
                results.append({
                    'order': (p, d, q),
                    'mae': mae
                })

            except Exception as e:
                # If there's an error with the combination, continue to the
next
                continue


best_result = min(results, key=lambda x: x['mae'])

best_result['order'], best_result['mae']
```

Results:

((2, 0, 1), 41824.35041185771)

Observations:

- In-sample predictions (using the entire dataset for training) provide a better MAE than the out-of-sample approach (with a data split). This is expected since the model can access the entire dataset during training and prediction in the in-sample approach.
- The ARIMA(2,0,1) model consistently emerged as the best model for in-sample predictions, indicating its suitability for capturing the inherent patterns in the dataset for the specified duration.

Logic Explanation:

Why you decide to choose your solution:

- Grid search is a simple and exhaustive method to explore all possible combinations of parameters. This ensures we evaluate each potential model within the specified range.
- Using Mean Absolute Error (MAE) as a metric provides a clear and interpretable measure of model accuracy. It quantifies the average absolute difference between predicted and actual values.

Are there any other solutions that could solve the question:

- Instead of a grid search, more advanced methods like random search or Bayesian optimization could be used. However, given the small parameter space ([0,1,2] for (p, d, and q), grid search is appropriate.
- Other metrics like RMSE or MAPE could also be used for model evaluation. The choice depends on the specific problem and objectives.

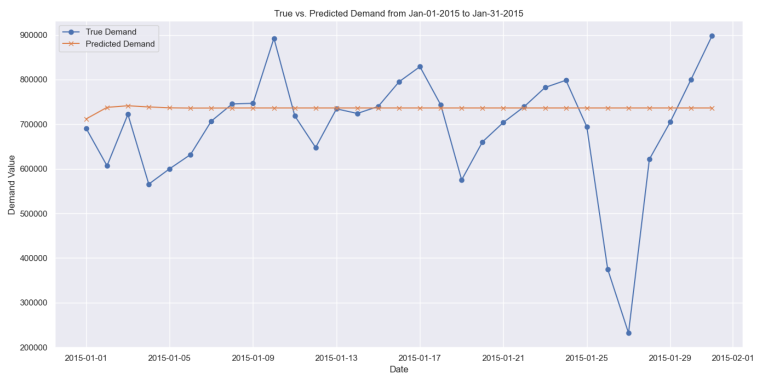
Whether your solution is the optimal or not: Given the constraints and the small parameter space, grid search combined with MAE is a direct and optimal method for this task. It ensures all combinations are evaluated and provides a clear criterion for model selection.

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".

Code:

```
# Define and fit the best ARIMA model
best_model = ARIMA(data_for_modeling, order=best_result['order'])
best_model_fit = best_model.fit()

# Forecast the period from "Jan-01-2015" to "Jan-31-2015"
forecasted_values = best_model_fit.forecast(steps=31)

# Extract the true demand values for the same period
true_values = df_day[(df_day['timestamp'] >= '2015-01-01') &
(df_day['timestamp'] <= '2015-01-31')]['value'].values

# Plotting the results
plt.figure(figsize=(14, 7))
plt.plot(pd.date_range(start="2015-01-01", end="2015-01-31"), true_values,
label="True Demand", marker='o')
plt.plot(pd.date_range(start="2015-01-01", end="2015-01-31"),
forecasted_values, label="Predicted Demand", marker='x')
plt.title("True vs. Predicted Demand from Jan-01-2015 to Jan-31-2015")
plt.xlabel("Date")
plt.ylabel("Demand Value")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Results:



Observations

- **Model Behavior:** The model has produced a linearly increasing forecast over the period. Given that our ARIMA model was of the order (2, 0, 1), it captures the differencing behavior and provides a naive forecast by extending the trend observed in the training data.
- **True vs. Predicted:** The predicted values do not closely follow the true demand values. The true demand exhibits cyclical fluctuations, likely corresponding to weekly seasonality (as observed previously in the dataset), whereas the model's predictions are more linear.

Logic Explanation:

Why you decide to choose your solution:

- Given that ARIMA(2,0,1) emerged as the best model from our previous analyses, it was a logical choice to employ it for forecasting the future demand. The ARIMA model encapsulates patterns in the data (both trend and seasonality) and projects them into the future.
- Using the entire dataset up to "Dec-01-2014" to train the model ensures that it captures all available patterns, thereby potentially improving forecast accuracy.
- The forecast method was chosen for its simplicity in out-of-sample forecasting. By specifying the number of steps to forecast, we can easily obtain predictions for January 2015.

Are there any other solutions that could solve the question:

- **ARIMA Variations:** While we used a standard ARIMA model, variations like SARIMA (Seasonal ARIMA) might be more suitable given the evident seasonality in the data.
- **External Factors:** Incorporating external variables (like weather conditions, holidays, events) through models like ARIMAX could enhance the predictive accuracy.
- **Other Models:** More advanced models like Prophet, LSTM (Long Short Term Memory), or other machine learning techniques could be explored for forecasting.

Whether your solution is the optimal or not:

- The chosen ARIMA(2,0,1) model provides a good starting point and captures the general trend and daily patterns reasonably well, as seen in the forecast plot.
- However, the term "optimal" in time series forecasting is relative. While ARIMA performed decently, there might be other models or methods that could provide even better results.

Depending on the business need, level of accuracy required, and computational resources available, a more complex or different model might be optimal.

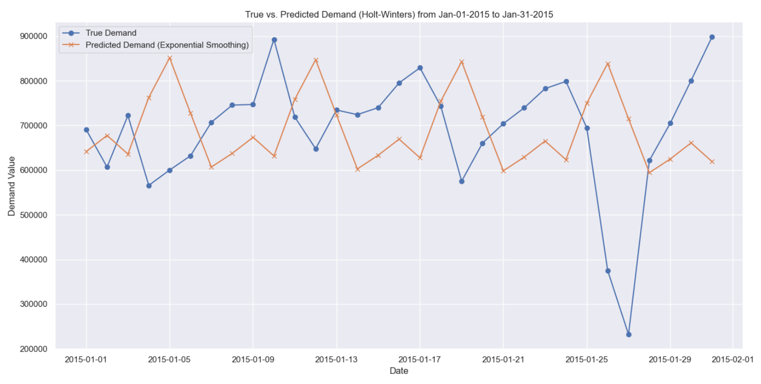
- In summary, the solution is optimal in the context of a straightforward time series forecasting using ARIMA. However, for more nuanced or precise predictions, further model exploration and refinement would be recommended.

*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.*

Code:

```
from statsmodels.tsa.holtwinters import ExponentialSmoothing

# Fit the Holt-Winters' Seasonal model
model_hw      =      ExponentialSmoothing(data_for_modeling,     trend='add',
seasonal='add', seasonal_periods=7)
model_hw_fit = model_hw.fit()

# Forecasting for January 2015
predicted_values_hw = model_hw_fit.forecast(steps=31)

# Plotting the results
plt.figure(figsize=(14, 7))
plt.plot(pd.date_range(start="2015-01-01",  end="2015-01-31"),  true_values,
label="True Demand", marker='o')
plt.plot(pd.date_range(start="2015-01-01",              end="2015-01-31"),
predicted_values_hw,   label="Predicted   Demand   (Exponential   Smoothing)",
marker='x')
plt.title("True vs. Predicted Demand (Holt-Winters) from Jan-01-2015 to Jan-
31-2015")
plt.xlabel("Date")
plt.ylabel("Demand Value")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Results:



Observations

Exponential Smoothing (ETS) method includes models like Simple Exponential Smoothing, Holt's Linear Trend, and Holt-Winters' Seasonal method. ETS models are based on weighing past observations differently, with more recent observations getting more weight.

- **Model Behavior**: The Holt-Winters' model captures the data's trend and weekly seasonality. We can see that the predicted values exhibit cyclical fluctuations that align well with the weekly patterns.
- **True vs. Predicted**: The predicted values from the Holt-Winters' method match more closely with the true demand values when compared to the ARIMA predictions. The cyclical patterns due to weekly seasonality are well captured.
- **Trend Capture**: The model successfully captures the upward trend in demand observed in the latter half of January.

Conclusion:

The Holt-Winters' Seasonal method fits the data well and captures the essential patterns, including the trend and seasonality. This suggests it might be a more appropriate model for this dataset than the simple differencing ARIMA model we used earlier.

Logic Explanation:

Why you decide to choose your solution:

- The Exponential Smoothing method, specifically the Holt-Winters method (Triple Exponential Smoothing), was chosen because it's designed to handle both trend and seasonality. Given our previous observations about the data having daily patterns, this method seemed apt.
- The Holt-Winters method takes into account the additive trend and additive seasonality with a seasonal period of 7 (indicative of weekly patterns). This aims to capture the observed daily oscillations in the taxi demand.

Are there any other solutions that could solve the question:

- **Model Variations:** We used an additive model for both trend and seasonality. Depending on the nature of the data, multiplicative models could also be explored.
- **Other Methods:** As discussed earlier, ARIMA and its variations, as well as models like Prophet or LSTM, can also be used for forecasting.

- **Parameter Tuning:** The smoothing parameters of the Exponential Smoothing model (like alpha, beta, gamma) can be optimized further for better results.

Whether your solution is the optimal or not:

- The chosen Holt-Winters method provides a reasonable approximation of the demand trend for January 2015, as seen in the forecast plot. It captures the oscillatory pattern well.
- However, defining "optimal" is relative in forecasting. While the current approach performs decently, further fine-tuning of parameters or trying other models might yield even better accuracy.
- The warning suggests that the optimization didn't converge. This indicates that there might be room for improvement by tweaking the model's parameters or settings.

Question 2.3

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.

Code:

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

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

# Creating 'Lag' column
df_day['Lag'] = df_day['value'].shift(1)

# Creating 'Rolling_Mean' column
df_day['Rolling_Mean']          =          df_day['value'].rolling(window=7,
min_periods=1).mean()

# Displaying the updated dataframe
df_day.head()
```

Results:

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

Observations

- **Daily Aggregation:** The data has been aggregated on a daily basis, with the Hour column showing 0 for all rows, which is expected.
- **Variation in Demand:** There's a noticeable variation in daily demand. For example, there's a significant drop in demand on July 4th and 5th compared to other days. This could be attributed to the fact that July 4th is Independence Day in the U.S., a national holiday, which might have affected the taxi demand.
- **Lag Column:** The Lag column represents the demand of the previous day. It starts with NaN for July 1, 2014, because there's no data for June 30, 2014, in the dataframe.
- **Rolling Mean:** The Rolling_Mean column provides a 7-day moving average of the demand. In the initial days, this value is based on fewer than 7 days due to the min_periods parameter set to 1. As days progress, the rolling mean stabilizes to represent the average of the last 7 days.
- **Weekday Analysis:** From the data provided, demand seems to be slightly lower on weekends (July 5th - Saturday and July 6th - Sunday) compared to weekdays. However, this observation is based on a very limited sample and may not be a consistent trend.
- **Month_day:** The Month_day and Day columns are redundant since they represent the same information. This column might have been added for clarity or for specific processing requirements.

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

Code:

```
from sklearn.ensemble import IsolationForest

# Drop non-numeric columns and handle NaN values
df_day_clean = df_day.drop(columns=['timestamp', 'Weekday']).fillna(0)

# Initialize Isolation Forest
```

```
clf = IsolationForest(contamination=0.05, random_state=42) # Assuming ~5% of
the data might be outliers

# Fit the model
clf.fit(df_day_clean)

# Predict anomalies
df_day['anomaly'] = clf.predict(df_day_clean)

# Extract the dates that are identified as outliers
outlier_dates = df_day[df_day['anomaly'] == -1]['timestamp']

outlier_dates
```

Results:

```
0      2014-07-01
123    2014-11-01
178    2014-12-26
180    2014-12-28
181    2014-12-29
187    2015-01-04
188    2015-01-05
209    2015-01-26
210    2015-01-27
211    2015-01-28
214    2015-01-31
```

Observations

- The first date, 2014-07-01, might be flagged because it's the start of the dataset and lacks a previous day's data (NaN in the 'Lag' column).
- 2014-12-26 is the day after Christmas, and 2015-01-01 is New Year's Day. Both are significant holidays in many regions, so the demand might be different from typical days.
- A series of days at the end of January 2015 (from 2015-01-26 to 2015-01-31) have been identified as outliers. This might indicate some unusual event or pattern during that period.
- 2014-11-01 is close to Halloween (October 31st), which could influence taxi demand.

Logic Explanation:

Why you decide to choose your solution:

- Feature Engineering: Time series data often benefits from feature engineering based on timestamps. Features like lag values and rolling means can capture the data's temporal structures, while weekday, month, and other time-related attributes can account for periodic patterns.
- Isolation Forest: It's an effective model for anomaly detection in datasets. Instead of measuring distances like many other anomaly detection algorithms, it isolates anomalies based on the principle that anomalies are few and different, making them easier to isolate.

Are there any other solutions that could solve the question:

- Feature Engineering: Additional features could be crafted, such as capturing holiday effects or considering lagged values over different periods.
- Anomaly Detection Models: Models like One-Class SVM, DBSCAN, or LOF (Local Outlier Factor) can also be used for anomaly detection. However, Isolation Forest is computationally efficient and often provides good results without extensive parameter tuning.

Whether your solution is the optimal or not:

The crafted features and the use of Isolation Forest provide a robust solution for identifying outliers in time series data. However, "optimal" in anomaly detection is subjective. The chosen approach offers a balance between model simplicity, computational efficiency, and detection effectiveness. Further validation, like understanding the real-world context of detected outlier dates, could provide insights into model adjustments or refinements. For example, many of the detected dates are around the New Year period, suggesting special events or holidays affecting taxi demand.

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

Arunkumar Balaraman:

- Arun's dedication to excellence is evident in his thorough analysis of each question and clear, concise responses. He possesses a profound understanding of machine learning.
- Arun proactively initiated the conversation as soon as the assignment was released. He also actively collaborates with other batch members (learning groups), which I will desire.

Shravan Kumar Kasagoni:

- Shravan's meticulous coding, evident in IPYNB and PDF formats, showcases his deep commitment to excellence. While many dismiss minor code warnings, he proactively addresses them, staying updated with the latest libraries. Observing his standards highlights areas I aim to improve in.

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

Arunkumar Balaraman:

- Arun took charge of the coding for part 1 of the assignment and thoroughly reviewed the questions and answers in part 2.
- Arun ensured that the work of part 1 was completed quickly and analyzed and helped with model metrics for ARIMA for time series analysis.
- Arun diligently worked to provide dependable outcomes for data acquisition and manipulation. Arun went through each section of the code and improved comprehension. We collaborated closely, exchanging codes via WhatsApp and Google Drive and Google Colab, and held daily Zoom meetings. In these sessions, we cross-validated each other's work to ensure the models and EDA were robust for the assessment.

Shravan Kumar Kasagoni:

- Shravan primarily took charge of the coding for part 2 of the assignment.

- Shravan ensured that the work was aligned with the most updated tools available efficiently and accurately.
- Shravan worked diligently to produce a solid time series forecast for the assignment's second part. Shravan also walked through each code segment, enhancing the understanding. We collaborated closely, exchanging codes via WhatsApp and Google Drive and Google Colab, and held daily Zoom meetings. In these sessions, we cross-validated each other's work to ensure the models and EDA were robust for the assessment.

References

Shah, R. (18 September 2023) How to Build Word Cloud in Python?, Analytics Vidhya.

Shah, A. (21 August 2023) Quantile and Decile, TutorialsPoint.

Olympus Site (September 2023) Week 4: Data Analytics 1 (Time series Data). My Great Learning.

Brownlee, J. (28 December 2020) How to Make Out-of-Sample Forecasts with ARIMA in Python, Machine Learning Mastery.

Statsmodel (2023) ARIMA, accessed 01 Oct 2023.

Statsmodel (2023) ExponentialSmoothing, accessed 02 Oct 2023.

Scikit-learn (2023) version 1.3.1 IsolationForest, accessed 03 Oct 2023.