ASSESMENT 2

BHANU PRATAP REDDY - 223935599
HARPREET SINGH - 223925166
SAI SIVA SARMA - 223924778

08/10/2023

PART 1

Q 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)

Before we begin unpacking the data, let's import necessary libraries to perform statistical analysis and data manipulation (if required)

```
#Importing all necessary Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
from wordcloud import WordCloud, STOPWORDS
from pylab import rcParams

#Dates
import matplotlib.dates as mdates
from matplotlib.dates import DateFormatter, MonthLocator, HourLocator

#Decomposition
from statsmodels.tsa.seasonal import seasonal_decompose

#Dickey fuller Test
from statsmodels.tsa.stattools import adfuller

#Autocorrelation and Partial Autocorrelation
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

#ARIMA
import statsmodels.api as sm
from statsmodels.tsa.arima.model import ARIMA
from sklearn.metrics import mean_squared_error
from math import sqrt

#Exponential Smoothing
from statsmodels.tsa.api import ExponentialSmoothing

#isolation Forest
from sklearn.ensemble import IsolationForest

# Declaring this to show graphs after inline command instead of plt.show() after every instance of plotting
%matplotlib inline

#Set the max number of rows in output cell to 50
pd.options.display.max_rows = 50


#Ignore unnecessary warnings
from warnings import filterwarnings
filterwarnings("ignore")
```

Initial step is to set up data frame from the csv file and read the data using pandas library and see the data structure along with few rows of raw data.

	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...

The dimensions of the data are as follows

Rows : 355808
Columns : 8

Now let's check for non-null values in the data frame for each column

```
<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
```

We see that few of the variables have missing values. Let's check what are those and see if that can be imputed?

```
category_name          1539
brand_name           151956
clean_description       194
dtype: int64
```

Most of the missing values are from 'brand_name' which sums up to 98% of the missing values on whole. **Here we cannot impute the missing values using median as the data type of the variable is categorical.**

Q 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.

To print the median of each item in the dataframe I used the below lines of code.

```
#Group by the column interested in and the corresponding metrics
median_df = item_category_df.groupby('name').median()
#Drop columns not interested in
cols_to_be_dropped = median_df.columns.drop('price')
median_df.drop(median_df[cols_to_be_dropped],axis=1,inplace=True)
#Rename the columns to match teh operations
median_df.rename(columns = {'price':'Median Price'}, inplace = True)
```

In this code, we're essentially organizing our '**item_category_df'** data by grouping it based on the **'name'** column. This means we're creating smaller subsets of data for each unique '**name**' value and finding the median '**price**' within each of these groups. The resulting dataframe is named **median_df**. We then identify and keep only the '**price**' column in **median_df**, storing the names of other columns in **cols_to_be_dropped**. Lastly, we change the name of the '**price**' column to '**Median Price**' for clarity. This code helps us summarize and simplify our data, making it easier to work with and analyze.

	Median Price
name	
! 3 girls t shirts	10.0
! HOLD ! Distressed Jeans/ Shorts Bundle	75.0
! Labor Day Weekend Sale!Band thumb ring	5.0
! Price drop! Thirty one cinch sack	11.0
! Price lowered! Target brand boots	14.0

As the question is a bit ambiguous, I have used another approach where I have considered the median value of the name column for the entire data set.

```
median_price = df['price'].median()
✓ 0.0s
```

And the output of the above code is **17.0**

To find the 90th percentile of the price column, we have used the below code.

```
#Group by the column interested in and the corresponding metrics
quant_df = item_category_df.groupby('name').quantile(.90)
#Drop columns not interested in
cols_to_be_dropped = quant_df.columns.drop('price')
quant_df.drop(quant_df[cols_to_be_dropped],axis=1,inplace=True)
#Rename the columns to match teh operations
quant_df.rename(columns = {'price':'Price 90th perc'}, inplace = True)
```

This code groups the Dataframe **item_category_df** by the '**name**' column and calculates the 90th percentile value for '**price**' within each group, creating a new Dataframe named **quant_df**. It then identifies and removes columns in **quant_df** other than '**price'** and renames the 'price' column to '**Price 90th perc**'. Essentially, this code summarizes the data by providing the 90th percentile price for each '**name**' category, simplifying analysis and focusing on the most significant pricing information.

The alternative approach where I have considered the 90th Percentile value of the column for the entire data set.

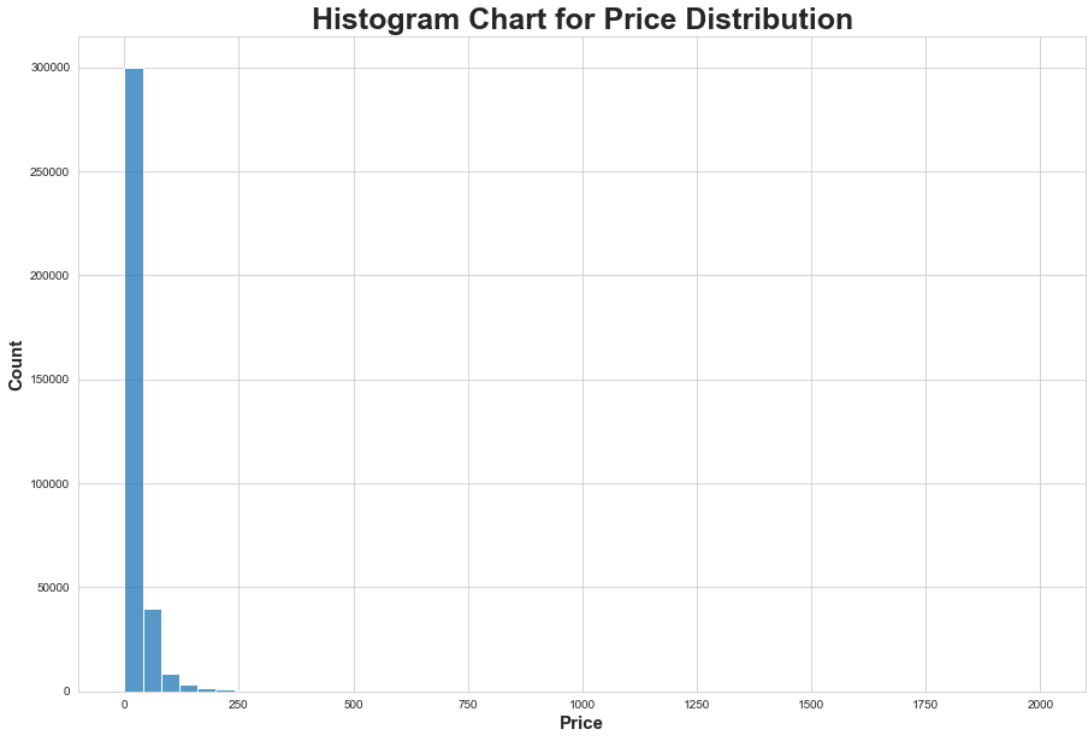
```
percentile_90th = np.percentile(df['price'].values, 90)
```

Output : 51.0

For the above question, the solution I worked on is as follows.

```
#Declare Figure and specify an appropriate size and title
plt.figure(figsize=(15,10))
plt.title("Histogram Chart for Price Distribution",fontsize=25, weight="bold")

#Remove outer border and set grid style
sns.set_style("whitegrid")

#Plot the graph using seabont countplot and specify where data values must appear
ax = sns.histplot(data= item_category_df, x= "price",bins=50);

#Label the axes and rotate the label of the categories so that it is readable
plt.xlabel("Price",fontsize=15, weight="bold");
plt.ylabel("Count",fontsize=15, weight="bold");
```

The above code creates a histogram chart using Python's Matplotlib and Seaborn libraries. It sets the figure size and title for the plot, removes the outer border, and adds a white grid background. It then uses Sanborn's histplot function to plot the distribution of 'price' data from the Dataframe **item_category_df** with ***50 bins***. The code guarantees that the axes have clear labels, which improves understanding, and it improve readability.



You can present the data differently too.

- One option is to use a smooth line plot called a kernel density estimation (KDE) plot. It's like a wavy line that shows how the data is spread out smoothly.
- Another choice is a box plot, which looks like a rectangular box and helps you see the middle value, how spread out the data is, and if there are any unusual points in the price data.

The method you pick depends on what you want to highlight in the data. If you want to see the overall pattern or focus on specific details, that's what guides your choice.

Q 1.3

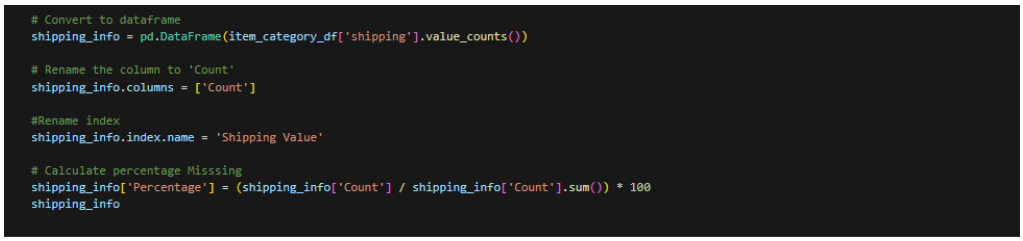
Exploring the shipping information from the data:

- 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)

Exploring the shipping information from the data:

Below code is to find the percentage of items paid by buyers. Here the logic is if the shipping value is 0 then the buyer pays the money, else the seller pays the money. Based on this logic I have worked the rest of the problem statement.



```
# Convert to dataframe
shipping_info = pd.DataFrame(item_category_df['shipping'].value_counts())

# Rename the column to 'Count'
shipping_info.columns = ['Count']

#Rename index
shipping_info.index.name = 'Shipping Value'

# Calculate percentage Misssing
shipping_info['Percentage'] = (shipping_info['Count'] / shipping_info['Count'].sum()) * 100
shipping_info
```

This code takes a column called "shipping" from a data frame called "item_category_df" and creates a new data frame called "shipping_info". It then counts the number of times each unique shipping value appears in the original data frame. Next, it renames the columns in the new data frame for clarity. Finally, it calculates the percentage of times each shipping value appears, relative to the total number of shipping values.

Output :

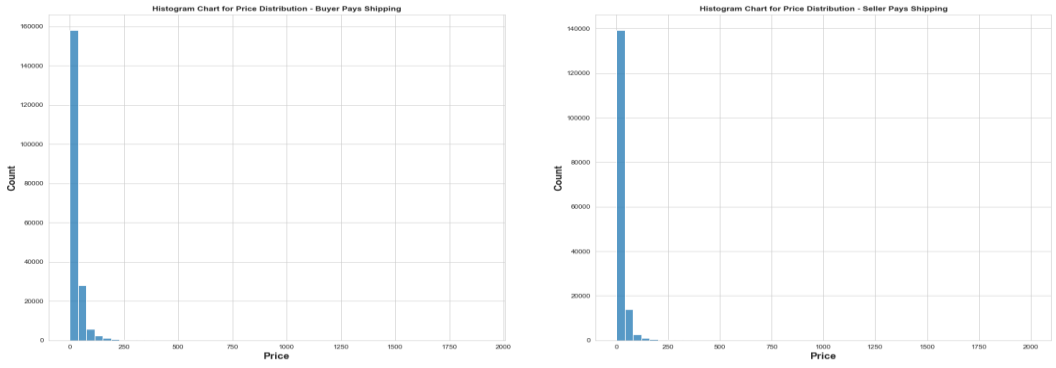
Shipping Value	Count	Percentage
0	197064	55.384927
1	158744	44.615073

So from the above output we can say that the the buyer has paid almost 55% of the times.

```
#Declare Figure and specify an appropriate size and title
plt.figure(figsize=(25,10))
plt.subplot(1,2,1);
plt.title("Histogram Chart for Price Distribution - Buyer Pays Shipping",fontsize=12, weight="bold")

#Remove outer border and set grid style
sns.set_style("whitegrid")

#Plot the graph using seaborn countplot and specify where data values must appear
ax = sns.histplot(data= df_buyer_pays, x= "price",bins=50);

#Label the axes and rotate the label of the categories so that it is readable
plt.xlabel("Price",fontsize=15, weight="bold");
plt.ylabel("Count",fontsize=15, weight="bold");

plt.subplot(1,2,2);
plt.title("Histogram Chart for Price Distribution - Seller Pays Shipping",fontsize=12, weight="bold")

#Remove outer border and set grid style
sns.set_style("whitegrid")

#Plot the graph using seaborn countplot and specify where data values must appear
ax = sns.histplot(data= df_seller_pays, x= "price",bins=50);

#Label the axes and rotate the label of the categories so that it is readable
plt.xlabel("Price",fontsize=15, weight="bold");
plt.ylabel("Count",fontsize=15, weight="bold");
```

This code creates two side-by-side histogram charts to compare the price distribution of items where buyers pay shipping ('**Buyer Pays Shipping**') and items where sellers pay shipping ('**Seller Pays Shipping**').
It sets the

- figure size,
- removes the outer border,
- labels axes with clear titles and rotated category labels for readability.
- The code uses Seaborn's histplot to visualize the price distribution for each category with 50 bins.



```
buy_sell_median_df = item_category_df.groupby('shipping').median()
cols_to_be_dropped = buy_sell_median_df.columns.drop('price')
buy_sell_median_df.drop(buy_sell_median_df[cols_to_be_dropped],axis=1,inplace=True)
buy_sell_median_df.rename(columns = {'price':'Median Price'}, inplace = True)
buy_sell_median_df
```

This code groups the item_category_df DataFrame by the 'shipping' column and calculates the median price for items where buyers pay shipping and where sellers pay shipping. It then drops columns other than 'price', effectively keeping only the median price. Finally, it renames the 'price' column to 'Median Price' for clarity. The resulting DataFrame, buy_sell_median_df, shows the median price for both shipping scenarios.

Output :

shipping	Median Price
0	19.0
1	14.0

Upon examining the output, it's evident that the median price for buyers is higher than for sellers.That means, on average, the prices of items where buyers pay for shipping are higher than the prices of items where sellers cover the shipping costs.

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 whether the better condition of the item could have higher median price (draw the plot and answer this question in the comment and report).

item_condition_id.
Here is the code for the statement mentioned above.

```
# Convert to dataframe
condition_info = pd.DataFrame(item_category_df['item_condition_id'].value_counts())

# Rename the column to 'Count of rows'
condition_info.columns = ['Count of rows']

#Rename index
condition_info.index.name = 'Condition Info'

# Calculate percentage Misssing
condition_info['Percentage'] = (condition_info['Count of rows'] / condition_info['Count of rows'].sum()) * 100
condition_info
```

This code creates a summary table of the item conditions in the **item_category_df** dataset. It counts the number of times each item condition appears and calculates the percentage of each condition compared to the total number of items.

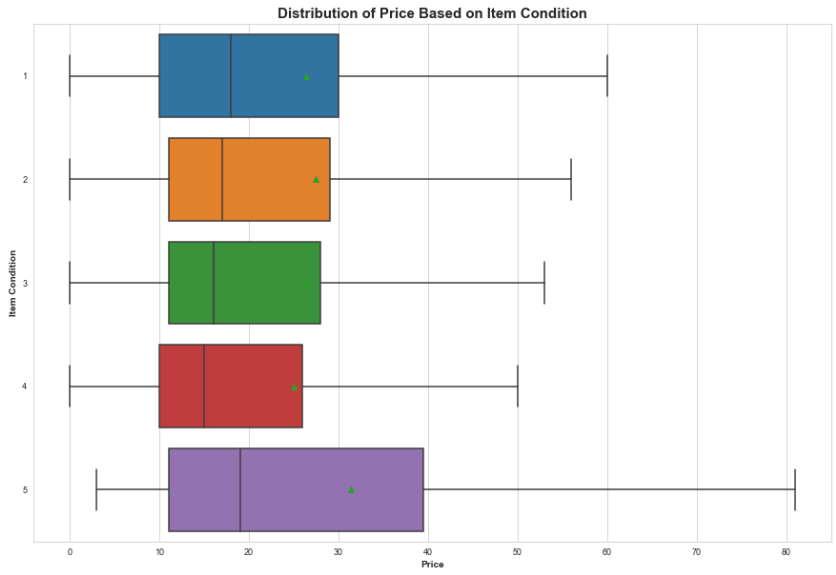
Here is the output of the above code :

Condition Info	Count of rows	Percentage
1	153402	43.113702
3	104248	29.298948
2	89843	25.250416
4	7768	2.183200
5	547	0.153735

Before we move on to the next part of the problem, we need to convert the column **condition_info** (which we created above) to a categorical data type. Below is the code to convert the data as required.

```
condition_info['Percentage'] = (D['Count of rows'] / condition_info['Count of
                               rows'].sum()) * 100
```

Now that we have all the requirements met to construct a box plot, here is the code to separate the box plot of each category and exclude outliers for it to give us a clean view of data points.



Q 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).:

To find the unique categories in the item_category_df['category_name'] column of the dataframe, we are using the value_counts() function on the above column and after that using len() function on the result set returned by value_counts() above.
value_counts() function group the unique values in a category column with their respective number of counts occurring in the dataset.
len() function returns the number of items in an object.

[40]:
```
# Convert to dataframe
unique_categories = pd.DataFrame(item_category_df['category_name'].value_counts())

# Rename the column to 'Count of rows'
unique_categories.columns = ['Count of rows']

#Rename index
unique_categories.index.name = 'Category'

# Calculate percentage Misssing
unique_categories['Percentage'] = (unique_categories['Count of rows'] / unique_categories['Count of rows'].sum()) * 100
unique_categories
```

[40]:

Category	Count of rows	Percentage
Women/Athletic Apparel/Pants, Tights, Leggings	14336	4.046643
Women/Tops & Blouses/T-Shirts	11187	3.157770
Beauty/Makeup/Face	8234	2.324223
Beauty/Makeup/Lips	7188	2.028967
Electronics/Video Games & Consoles/Games	6391	1.803996
...	...	...
Handmade/Patterns/Handmade	1	0.000282
Vintage & Collectibles/Serving/Glassware	1	0.000282
Handmade/Clothing/Dress	1	0.000282
Handmade/Pets/Toy	1	0.000282
Handmade/Books and Zines/Comic	1	0.000282

1135 rows × 2 columns

[17]:
```
print(f'The total No. of Unique Categories are {len(unique_categories)}')
```
```
The total No. of Unique Categories are 1135
```

Unique number of categories is 1135

To find the top 3 categories with worst condition of items, the item_category_df dataset is filtered with a condition, that item_condition_id column should have a value of 5, into a new dataframe worst_item_condition_df. Dataframe value_counts() function is applied on above dataframe worst_item_condition_df and checking the top 3 items in a result set returned by applying .value_counts()

[42]:
```
#initialise dataframe containing entries of only the worst Item Coinditions
worst_item_condition_df = item_category_df[item_category_df['item_condition_id'] == 5]

# Convert to dataframe
worst_unique_categories = pd.DataFrame(worst_item_condition_df['category_name'].value_counts())

# Rename the column to 'Count of rows'
worst_unique_categories.columns = ['Count of rows']

#Rename index
worst_unique_categories.index.name = 'Category'

# Calculate percentage Misssing
worst_unique_categories['Percentage'] = (worst_unique_categories['Count of rows'] / worst_unique_categories['Count of rows'].sum()) * 100
worst_unique_categories
```

[42]:

Category	Count of rows	Percentage
Electronics/Cell Phones & Accessories/Cell Phones & Smartphones	137	25.137615
Electronics/Video Games & Consoles/Games	42	7.706422
Electronics/Video Games & Consoles/Consoles	35	6.422018
Men/Shoes/Athletic	22	4.036697
Electronics/Computers & Tablets/Laptops & Netbooks	21	3.853211
...	...	...
Electronics/Computers & Tablets/Desktops & All-In-Ones	1	0.183486
Other/Books/Education & Teaching	1	0.183486
Vintage & Collectibles/Toy/Car	1	0.183486
Sports & Outdoors/Outdoors/Other	1	0.183486
Other/Office supplies/Shipping Supplies	1	0.183486

120 rows × 2 columns

[55]:
```
print(f"the top 3 categories with worst condition of items:--\n\n{list(worst_unique_categories.index[:3].values)[0]} \
\n{list(worst_unique_categories.index[:3].values)[1]}\n{list(worst_unique_categories.index[:3].values)[2]}")
```

```
the top 3 categories with worst condition of items:--

Electronics/Cell Phones & Accessories/Cell Phones & Smartphones
Electronics/Video Games & Consoles/Games
Electronics/Video Games & Consoles/Consoles
```

The top Item categories with worst condition are **Electronics** and this explains why in 1.4 the items with the worst condition had the highest median price, since electronics are naturally priced higher than most other categories.

Q 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.

Here, split_col(df,colname,marker) is defined that takes 3 arguments, df-> dataframe, colname-> column to be split into and marker-> separator to isolate the values

- The body of function started by creating an empty dataframe with 3 new desired columns Then a for loop is created and iterated with each value of the colname to separate the values using the marker defined. The colname values are checked using try and except block.
- If error comes,that is caught by except block using Attribute error, then Category Unknown is filled in all 3 new columns, else colname is split with a '/' separator to return different values to fill in above created columns
- The loop is run for till the number of rows exhausted
- Finally new dataframe is returned with above created columns along with existing columns

[19]:
```
def split_col(df,colname,marker):
    #initialise temp dataframe with the desired new columns
    temp_df = df
    temp_df.insert(loc = 3,column = 'main_cat',value = '')
    temp_df.insert(loc = 4,column = 'sub_cat_1',value = '')
    temp_df.insert(loc = 5,column = 'sub_cat_2',value = '')

    #Iterate Over the rows and seperate the cateogry and place it
    df_len = len(temp_df)
    for i in tqdm(range(df_len)):
        try:
            temp_var = df[colname][i].split(marker)
            temp_df['main_cat'][i] = temp_var[0]
            temp_df['sub_cat_1'][i] = temp_var[1]
            temp_df['sub_cat_2'][i] = temp_var[2]

        except AttributeError:
            temp_df['main_cat'][i] = "Category Unknown"
            temp_df['sub_cat_1'][i] = "Category Unknown"
            temp_df['sub_cat_2'][i] = "Category Unknown"

    # temp_df.drop(colname,axis=1,inplace=True)
    return temp_df
```

[20]:
```
item_category_df = split_col(df=item_category_df,colname='category_name',marker='/')
item_category_df.head()
```

```
100%|██████████| 355808/355808 [01:30<00:00, 3928.51it/s]
```

[20]:

	train_id	name	item_condition_id	main_cat	sub_cat_1	sub_cat_2	category_name	brand_name	price	shipping	clean_description
0	128037	Bundle for Sassy Sisters	3	Women	Tops & Blouses	Blouse	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	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	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	Kids/Boys 2T-5T/Shoes	Nike	19.0	0	baby black nike roshe runs size 5c

Q 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.

To change the text of newly created 3 columns, lower() built in function is used on the columns

[21]:
```
item_category_df['main_cat'] = item_category_df['main_cat'].str.lower()
item_category_df['sub_cat_1'] = item_category_df['sub_cat_1'].str.lower()
item_category_df['sub_cat_2'] = item_category_df['sub_cat_2'].str.lower()
item_category_df
```

[21]:

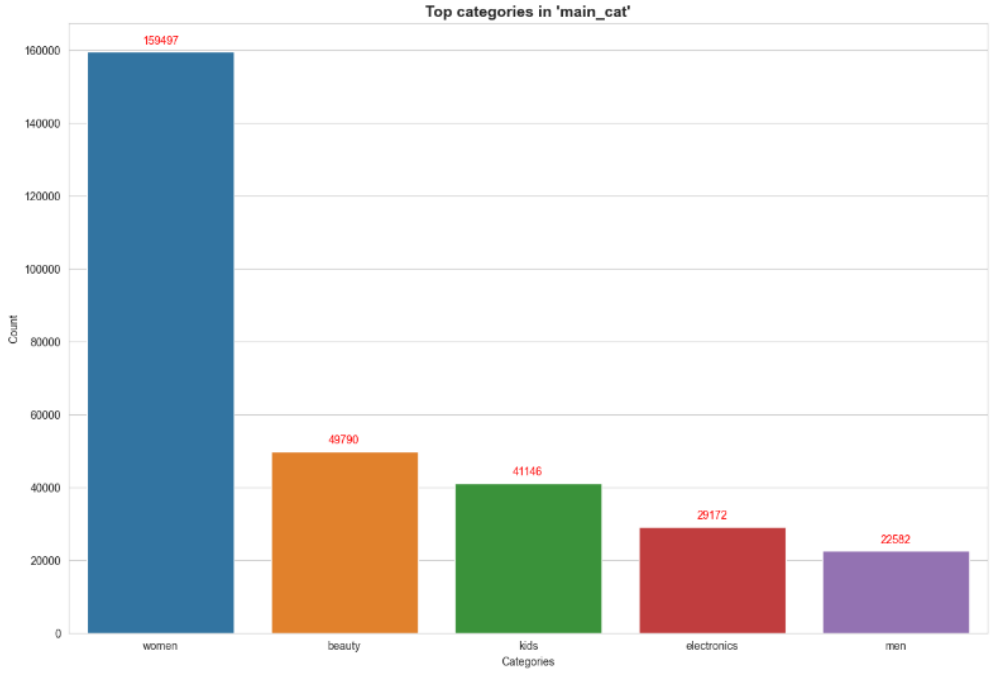
	train_id	name	item_condition_id	main_cat	sub_cat_1	sub_cat_2	category_name	brand_name	price	shipping	clean_description
0	128037	Bundle for Sassy Sisters	3	women	tops & blouses	blouse	Women/Tops & Blouses/Blouse	NaN	16.0	0	max cleo black dress paper crane black tank to...
1	491755	PINK VS TANK	2	women	tops & blouses	tank, cami	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	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	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	Kids/Girls 0-24 Mos/Dresses	Ralph Lauren	24.0	0	2 polo dresses 3 months wore washed dreft pink...
...	...	...	...	...	...	...	...	...	...	...	...

To find the top 5 most popular main categories (in main_cat column), .value_counts() function is used in item_category_df['main_cat'] column and selecting the top 5 rows returned and then creating a barplot

```
# Convert to dataframe
main_cat_count = pd.DataFrame(item_category_df['main_cat'].value_counts())

# Rename the column to 'Count of rows'
main_cat_count.columns = ['Count of rows']

#Rename index
main_cat_count.index.name = 'Category'

# Calculate percentage Misssing
main_cat_count['Percentage'] = (main_cat_count['Count of rows'] / main_cat_count['Count of rows'].sum()) * 100
main_cat_count
```

Category	Count of rows	Percentage
women	159497	44.826704
beauty	49790	13.993502
kids	41146	11.564102
electronics	29172	8.198804
men	22582	6.346681
home	16272	4.573253
vintage & collectibles	11207	3.149732
other	10958	3.079751
handmade	7520	2.113499
sports & outdoors	6125	1.721434
category unknown	1539	0.432537

```
# plot a bar plot with Coefficient on the x-axis and Variable names on y-axis
plt.figure(figsize =(15, 10))
plt.title("Top categories in 'main_cat'",fontsize=14, weight="bold")
sns.set_style("whitegrid")
ax = sns.barplot(x = main_cat_count.index[0:5], y = main_cat_count["Count of rows"][0:5])


plt.ylabel("Count", fontsize =10)
plt.xlabel('Categories', fontsize = 10)
ax.bar_label(ax.containers[0], label_type='edge', color = 'red',rotation =0, fontsize = 10,padding =5);
```



Top 5 main categories are **women, beauty, kids, electronics, men**

To get the unique number of main categories (in column main_cat), unique first sub-categories (in column subcat_1) and unique second sub-categories (in column subcat_2) respectively,
Respective dataframe sub_cat_1_count and sub_cat_2_count are created using value_counts function on the respective columns item_category_df['sub_cat_1'] , item_category_df['sub_cat_2']
And applying len() function on the dataframes

```
[24]: # Convert to dataframe
sub_cat_1_count = pd.DataFrame(item_category_df['sub_cat_1'].value_counts())

# Rename the column to 'Count of rows'
sub_cat_1_count.columns = ['Count of rows']

#Rename index
sub_cat_1_count.index.name = 'Category'

# Calculate percentage Misssing
sub_cat_1_count['Percentage'] = (sub_cat_1_count['Count of rows'] / sub_cat_1_count['Count of rows'].sum()) * 100
sub_cat_1_count
```

[24]:

Category	Count of rows	Percentage
athletic apparel	32185	9.045609
makeup	29917	8.408186
tops & blouses	25666	7.213441
shoes	24138	6.783996
jewelry	14798	4.158985
...	...	...
suits	13	0.003654

```
[25]: # Convert to dataframe
sub_cat_2_count = pd.DataFrame(item_category_df['sub_cat_2'].value_counts())

# Rename the column to 'Count of rows'
sub_cat_2_count.columns = ['Count of rows']

#Rename index
sub_cat_2_count.index.name = 'Category'

# Calculate percentage Misssing
sub_cat_2_count['Percentage'] = (sub_cat_2_count['Count of rows'] / sub_cat_2_count['Count of rows'].sum()) * 100
sub_cat_2_count
```

[25]:

Category	Count of rows	Percentage
t-shirts	14768	4.150553
pants, tights, leggings	14336	4.029139
other	12017	3.377383
face	11962	3.361926
shoes	7647	2.149193
...	...	...
storage cabinets	1	0.000281
chain	1	0.000281

[26]:
```
print(f'The Number of Unique categories in "main_cat": {len(main_cat_count)}')
print(f'The Number of Unique categories in "sub_cat_1": {len(sub_cat_1_count)}')
print(f'The Number of Unique categories in "sub_cat_2": {len(sub_cat_2_count)}')
```

```
The Number of Unique categories in "main_cat": 11
The Number of Unique categories in "sub_cat_1": 114
The Number of Unique categories in "sub_cat_2": 788
```

Unique categories in main_cat : 11
Unique categories in sub_cat_1 : 114
Unique categories in sub_cat_2 : 788

Q 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.

So we start off by grouping all the data by 'main_cat'

```
#Group by the column interested in and the corresponding metrics
median_main_cat_df = item_category_df.groupby('main_cat').median()
#Drop columns not interested in
cols_to_be_dropped = median_main_cat_df.columns.drop('price')
median_main_cat_df.drop(median_main_cat_df[cols_to_be_dropped],axis=1,inplace=True)
#Rename the columns to match the operations
median_main_cat_df.rename(columns = {'price':'Median Price'}, inplace = True)
median_main_cat_df
```

In the first part of Question 1.8, we focus on finding the median price for each category in the 'main_cat' column. The process is straightforward: we use the groupby method, which effortlessly segments the data based on distinct categories. It's a clean and efficient approach, yielding results without unnecessary complexity.

The creation of median_main_cat_df involves grouping by 'main_cat' and calculating the median. We then proceed to drop unrelated columns, retaining only the 'price' data. This streamlined dataframe is both easy to read and ready for further analysis.

We opt for this approach primarily for its elegance. An alternative could involve iterating through the dataframe and applying conditional checks, but that's clearly not as clean or efficient. Our method achieves what is required with as few lines of code as possible.

The Result:

main_cat	Median Price
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

Onto the second part, we start off by:

```
#Group by the column interested in and the corresponding metrics
most_expns_sub_cat_1_df = item_category_df.groupby('sub_cat_1').median()
#Drop columns not interested in
cols_to_be_dropped = most_expns_sub_cat_1_df.columns.drop('price')
most_expns_sub_cat_1_df.drop(most_expns_sub_cat_1_df[cols_to_be_dropped],axis=1,inplace=True)
#Rename the columns to match teh operations
most_expns_sub_cat_1_df.rename(columns = {'price':'Mean Price'}, inplace = True)
# sort the dataframe in descending order
most_expns_sub_cat_1_df = most_expns_sub_cat_1_df.sort_values('Mean Price', ascending = False)
most_expns_sub_cat_1_df
```

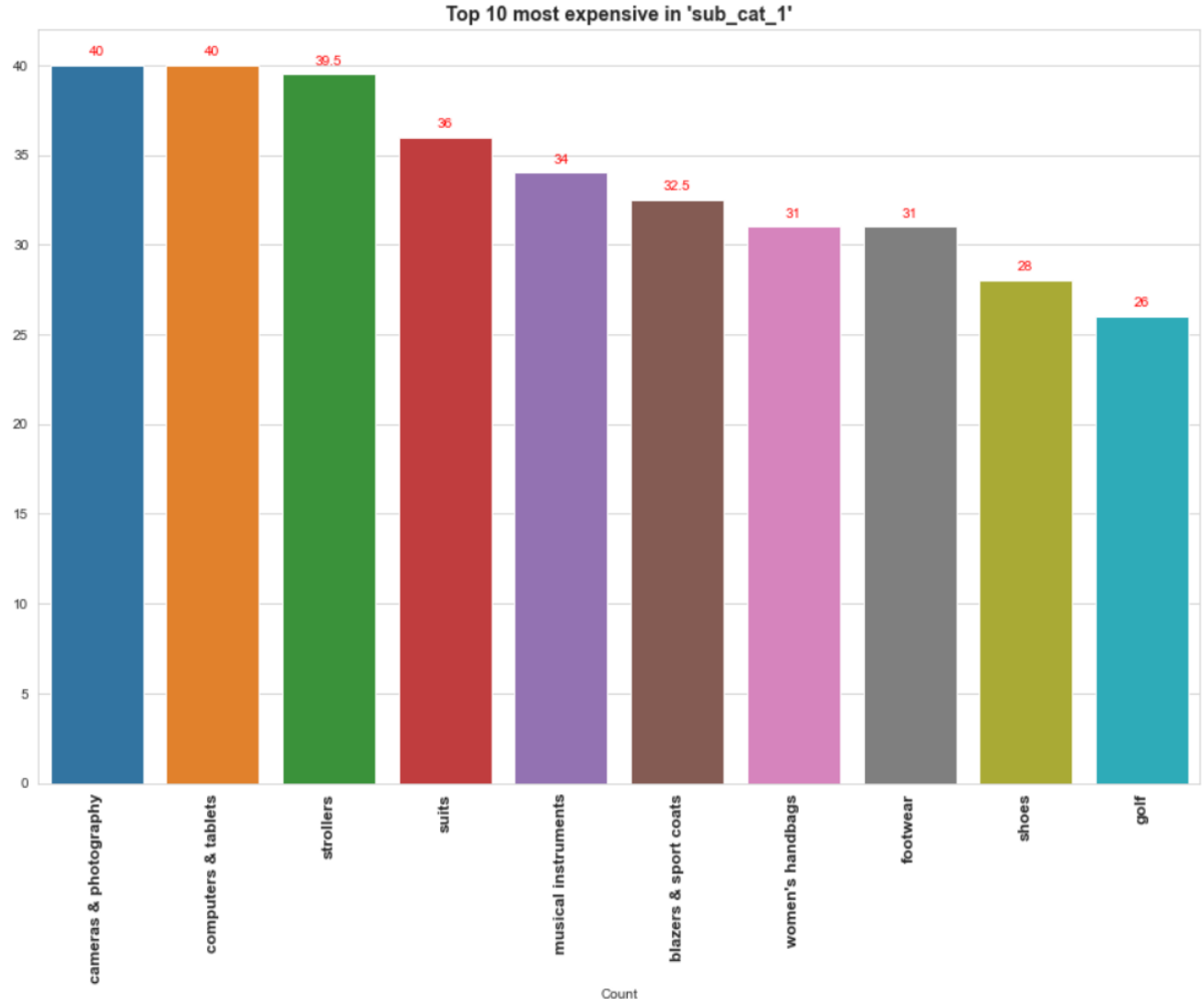
,The objective is to pinpoint the top 10 most expensive sub-categories in the 'sub_cat_1' column.

Similar to the previous step, we employ a groupby operation on 'sub_cat_1' and calculate the median. This approach, though similar, is slightly changed to meet the specific needs of this part of of the question. The median prices are then isolated by eliminating unrelated columns, resulting in a refined dataframe focused solely on the 'sub_cat_1' and their corresponding median prices.

However, there's a small addition this time. We introduce a sorting step, arranging the data in descending order of price. This aligns with the objective of identifying the most expensive

sub-categories but also enhances the readability of the data, allowing for immediate insights at a glance.

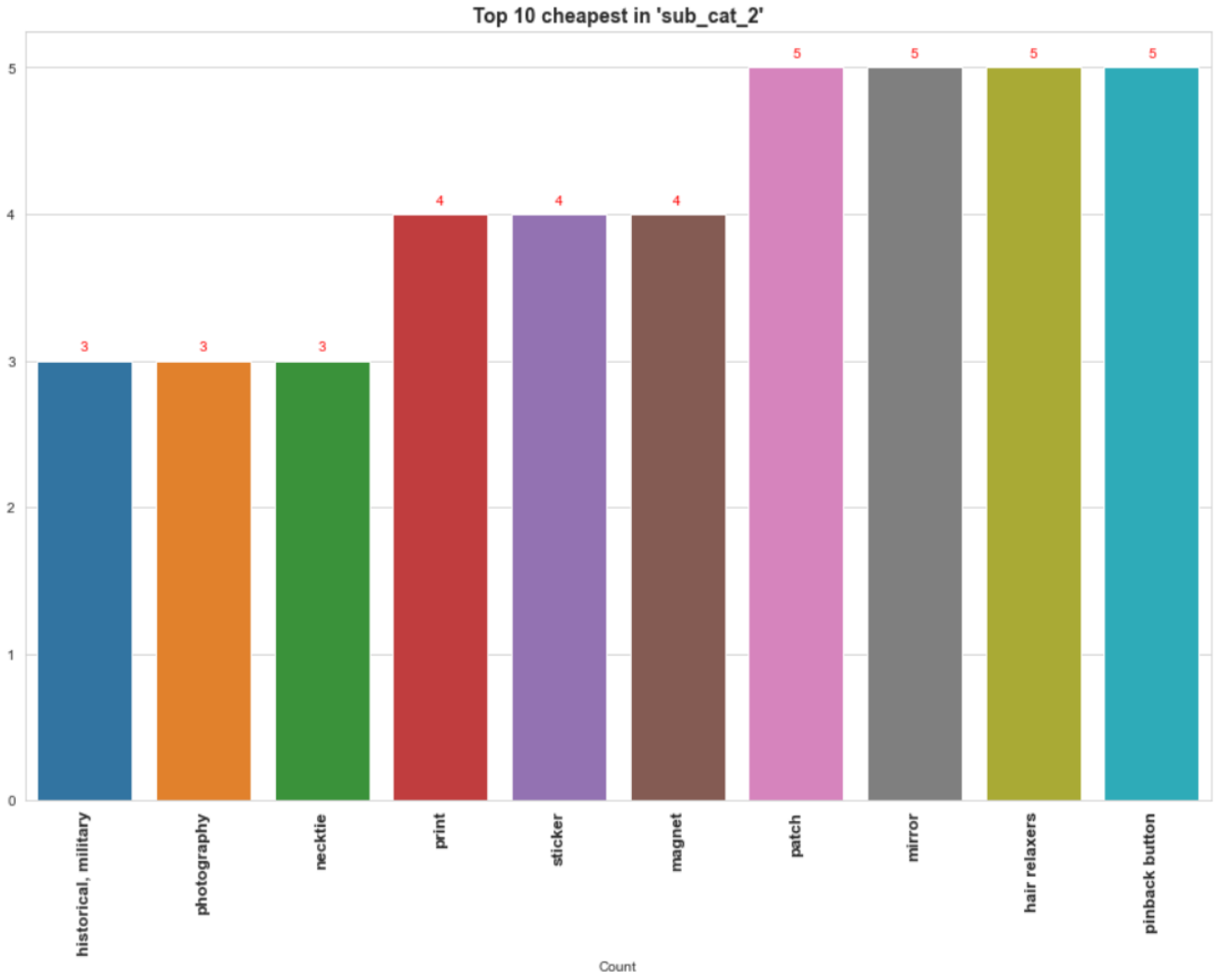
And the Resulting plot:



We do the exact same for the next part but the only change is we change the sorting order:

```
#Group by the column interested in and the corresponding metrics
cheapest_sub_cat_2_df = item_category_df.groupby('sub_cat_2').median()
#Drop columns not interested in
cols_to_be_dropped = cheapest_sub_cat_2_df.columns.drop('price')
cheapest_sub_cat_2_df.drop(cheapest_sub_cat_2_df[cols_to_be_dropped],axis=1,inplace=True)
#Rename the columns to match teh operations
cheapest_sub_cat_2_df.rename(columns = {'price':'Mean Price'}, inplace = True)
# sort the dataframe in Ascending order
cheapest_sub_cat_2_df = cheapest_sub_cat_2_df.sort_values('Mean Price', ascending = True)
cheapest_sub_cat_2_df
```

And the resulting Plot:



Q 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.

We start off by:

```
#replace missing values i.e. NaN with 'brand Unavailable'
item_category_df['brand_name'] = item_category_df['brand_name'].replace(np.NaN,'brand unavailable')
item_category_df
```

And then as we did earlier:

```
#Group by the column interested in and the corresponding metrics
median_brand_df = item_category_df.groupby('brand_name').median()
#Drop columns not interested in
cols_to_be_dropped = median_brand_df.columns.drop('price')
median_brand_df.drop(median_brand_df[cols_to_be_dropped],axis=1,inplace=True)
#Rename the columns to match teh operations
median_brand_df.rename(columns = {'price':'Median Price'}, inplace = True)
median_brand_df
```

As we see we are diving into the relationship between price and brand. The initial task is to handle missing brand names, which we've neatly addressed by replacing NaNs with 'brand unavailable'. This ensures 'Brand' column has no Null values.

To find the median price for each brand, a method similar to the previous one is employed: we execute a groupby operation on the 'brand_name', followed by a median calculation. The irrelevant columns are then discarded, and we're left with an easy-to-read dataframe showcasing each brand and its corresponding median price.

i.e., The Result:

brand_name	Median Price
% 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

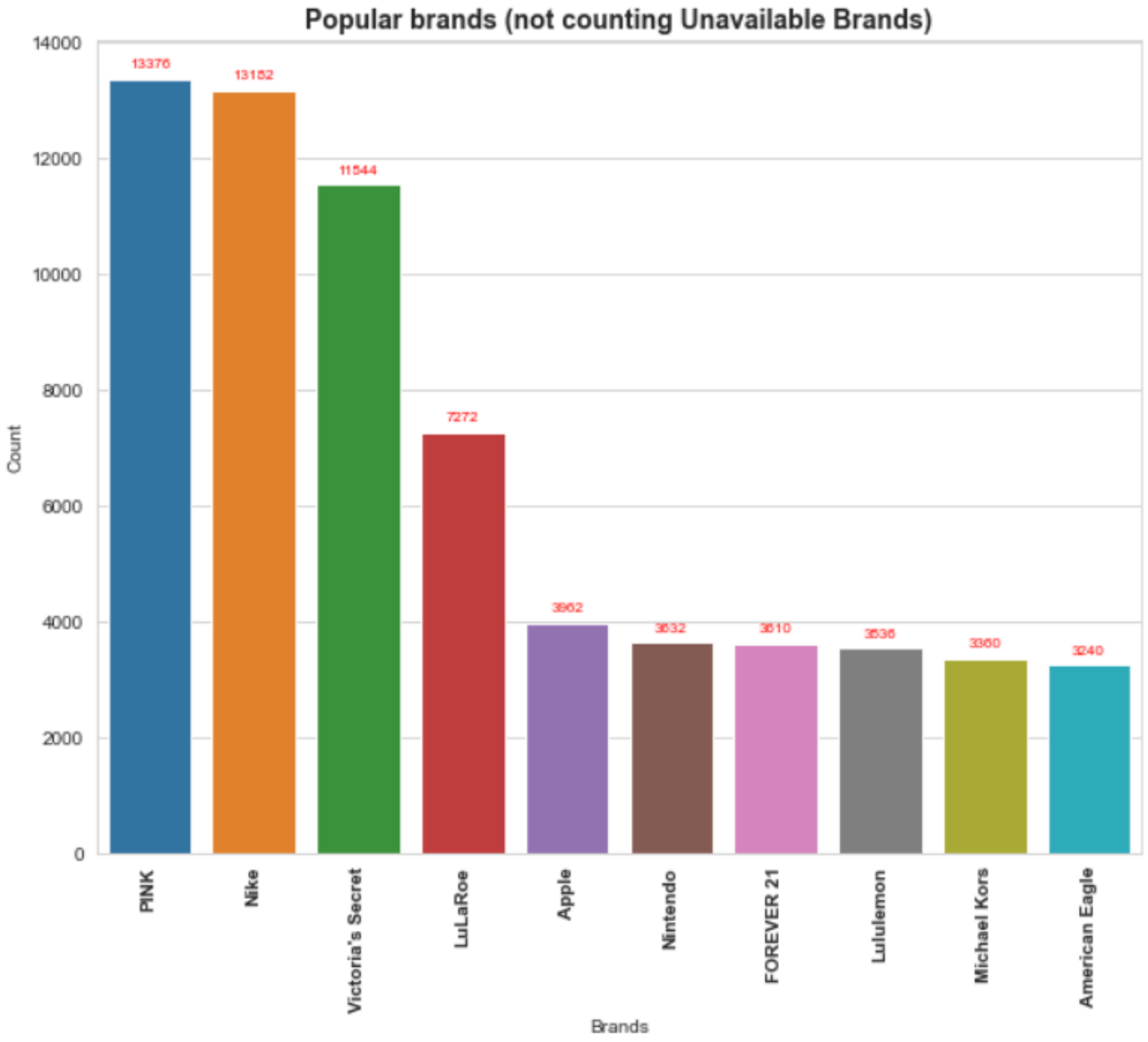
For the second segment of the question, the focus shifts to identifying the top 10 popular brands. A new dataframe, brand_popularity, is created to hold the count of each brand's appearance in the dataset. The additional 'Percentage' column offers insights into each brand's proportion in the dataset—a handy piece for a nuanced analysis.

```
# Convert to dataframe
brand_popularity = pd.DataFrame(item_category_df['brand_name'].value_counts())

# Rename the column to 'Count of rows'
brand_popularity.columns = ['Count of rows']

#Rename index
brand_popularity.index.name = 'Category'

# Calculate percentage Misssing
brand_popularity['Percentage'] = (brand_popularity['Count of rows'] / brand_popularity['Count of rows'].sum()) * 100
brand_popularity
```

The resulting Dataframe:

Category	Count of rows	Percentage
brand unavailable	151956	42.707303
PINK	13376	3.759331
Nike	13182	3.704807
Victoria's Secret	11544	3.244446
LuLaRoe	7272	2.043799
...	...	...
Spiderman	1	0.000281
Moon Collection	1	0.000281
Ralph Lauren Collection	1	0.000281
Kiss Me	1	0.000281
Doncaster	1	0.000281

The Plot:



Now, while this approach is efficient, there might be other nuanced ways to tackle this, especially in the brand popularity calculation. One could consider utilizing more complex metrics that account for other variables like ratings or reviews, offering a multi-dimensional view of popularity. However, for the scope of this question, our approach is sufficient and to the point.

Q 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

Here we're tasked with making sense of the 'clean_description' column via wordcloud, offering a visual representation of the most used words, and how their prominence varies across different price quantiles.

```
new_words = set(STOPWORDS)

all_text = " ".join(str(item) for item in item_category_df['clean_description'])
plt.subplots(figsize=(10,10))
wordcloud = WordCloud(background_color='white',max_words=50,
                            width=1200,stopwords=new_words ,
                            height=1000).generate(all_text)

plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

The first step is crafting a word cloud for the entire dataset. A set of stopwords is defined to filter out common words that don't contribute much to the analysis. We concatenate all descriptions into a single string, and then a word cloud is generated, offering an immediate visual insight into the most common words across all item descriptions.

The Resulting Wordcloud:

Next, we slice the data into four distinct quantiles based on the price. It's done using the .quantile method—a clean and efficient way to segment the data. Each quantile represents a price range, and this segmentation allows for the nuanced analysis of how item descriptions vary across different price points.

Four separate word clouds are generated for each quantile. The process remains consistent: concatenate the descriptions, generate the word cloud, and display it. Each subplot offers a visual representation of word frequency, tailored to the specific price quantile.

The Resulting Word Cloud:

This approach is straightforward. However, one could consider enhancing the stopwords list or utilizing more advanced NLP techniques for a more nuanced insight. Yet, for the scope of this question, this is fine—it's visually intuitive, and the segmentation by price quantiles ensures a level of detail that makes the insights actionable and clear.

PART 2

Q 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.
- 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)?

Here, we have 3 problem statements to solve. So let's start by setting up the data for time series analysis.

- Step 1 is to load the data using pandas function read_csv.
- Step 2 is to convert the timestamp column to date_time format using **to_datetime** function in pandas.
- Now lets check the attributes of the data set

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10320 entries, 0 to 10319
Data columns (total 2 columns):
 #   Column     Non-Null Count  Dtype         
---  ------     --------------  -----         
 0   timestamp  10320 non-null  datetime64[ns]
 1   value      10320 non-null  int64         
dtypes: datetime64[ns](1), int64(1)
memory usage: 161.4 KB
```

Now let's get into the problem statement 1 :

Create two new dataframes df_day and df_hour by aggregating the demand value on daily and hourly level.

For the above statement , I have created 2 datasets as below

df_hour :

```
df_by_hour = nyc_taxi_df.groupby(pd.Grouper(key='timestamp', freq='H')).sum()
df_by_hour
```

With the output of :

timestamp	value
2014-07-01 00:00:00	18971
2014-07-01 01:00:00	10866
2014-07-01 02:00:00	6693
2014-07-01 03:00:00	4433
2014-07-01 04:00:00	4379
...	...
2015-01-31 19:00:00	56577
2015-01-31 20:00:00	48276
2015-01-31 21:00:00	48389
2015-01-31 22:00:00	53030
2015-01-31 23:00:00	52879

5160 rows × 1 columns

df_day :

```
df_by_day = nyc_taxi_df.groupby(pd.Grouper(key='timestamp', freq='D')).sum()
df_by_day
```

With the output of :

timestamp	value
2014-07-01	745967
2014-07-02	733640
2014-07-03	710142
2014-07-04	552565
2014-07-05	555470
...	...
2015-01-27	232058
2015-01-28	621483
2015-01-29	704935
2015-01-30	800478
2015-01-31	897719

215 rows × 1 columns

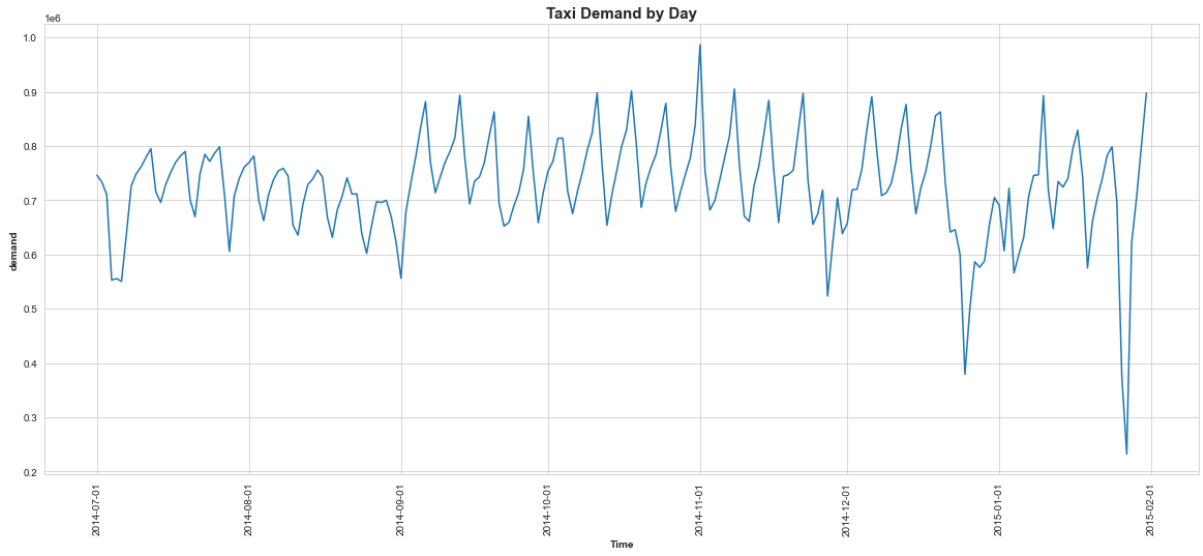
When we compare the 2 dataframes that we created now, dimensions of the hour dataframe is on the higher side compared to day dataframe, because the split happened on hourly basis for this dataframe.

Plot the demand value in two line charts for both df_day and df_hour dataframes.
Coming to the second part of the problem statement, here the requirement is to plot the chart for both the data frames

For df_day dataframe :
Code :

```
# Create figure and plot space
fig, ax = plt.subplots(figsize=(20, 8))

#plot
ax.plot(df_by_day.index.values,df_by_day['value'])

plt.title("Taxi Demand by Day",fontsize=15, weight="bold")
plt.xlabel("Time",fontsize=10,weight="bold")
plt.ylabel("demand",fontsize=10,weight="bold")

# Define the date format
date_form = DateFormatter("%Y-%m-%d")
ax.xaxis.set_major_formatter(date_form)

plt.xticks(rotation = 90);

plt.show()
```

Output :



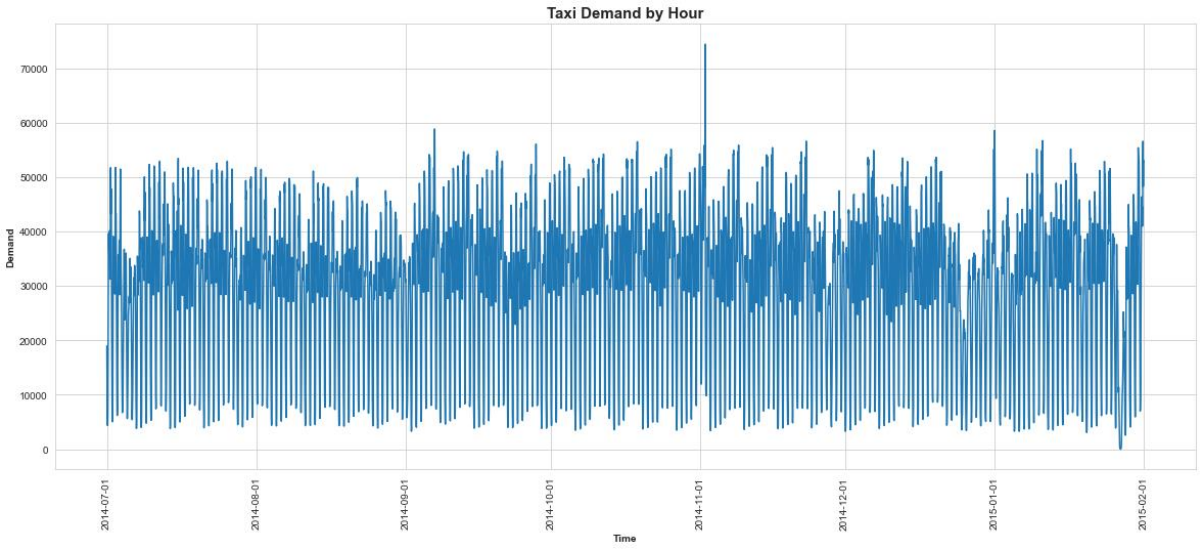
In the above plot, the distribution of demand of taxis is DOD trend for 8 month period.

For df_hour dataframe :

Code :

```
# Create figure and plot space
fig, ax = plt.subplots(figsize=(20, 8))

# Plot
ax.plot(df_by_hour.index.values, df_by_hour['value'])

plt.title("Taxi Demand by Hour", fontsize=15, weight="bold")
plt.xlabel("Time", fontsize=10, weight="bold")
plt.ylabel("Demand", fontsize=10, weight="bold")

# Define the date format
date_form = DateFormatter("%Y-%m-%d")
ax.xaxis.set_major_formatter(date_form)

# Set major locator as monthly and minor locator as 12 hourly
ax.xaxis.set_major_locator(MonthLocator())
ax.xaxis.set_minor_locator(HourLocator(interval=12))

plt.xticks(rotation=90)
plt.grid(True)

plt.show()
```

Output :



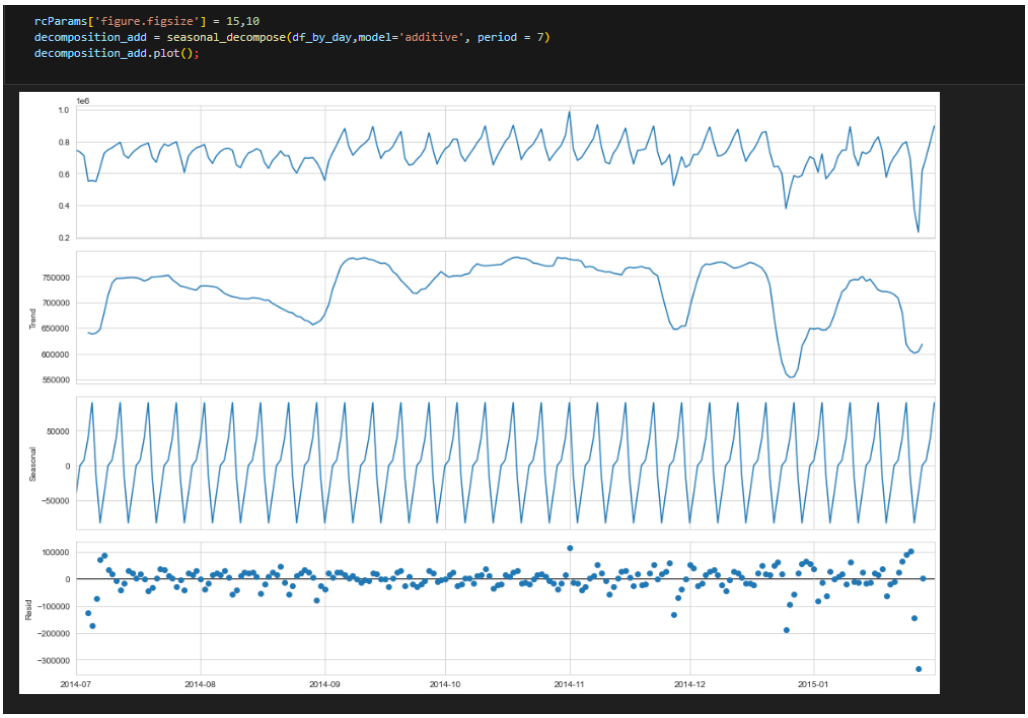
Here is the hourly distribution of the demand of taxis over the period of 8 months.

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)?

For the above statement, we need to decompose the dataframe to analyse trend, seasonality & residual.
For this we need to see if the data is in additive model or multiplicative model. We try both and see what matches the requirement.

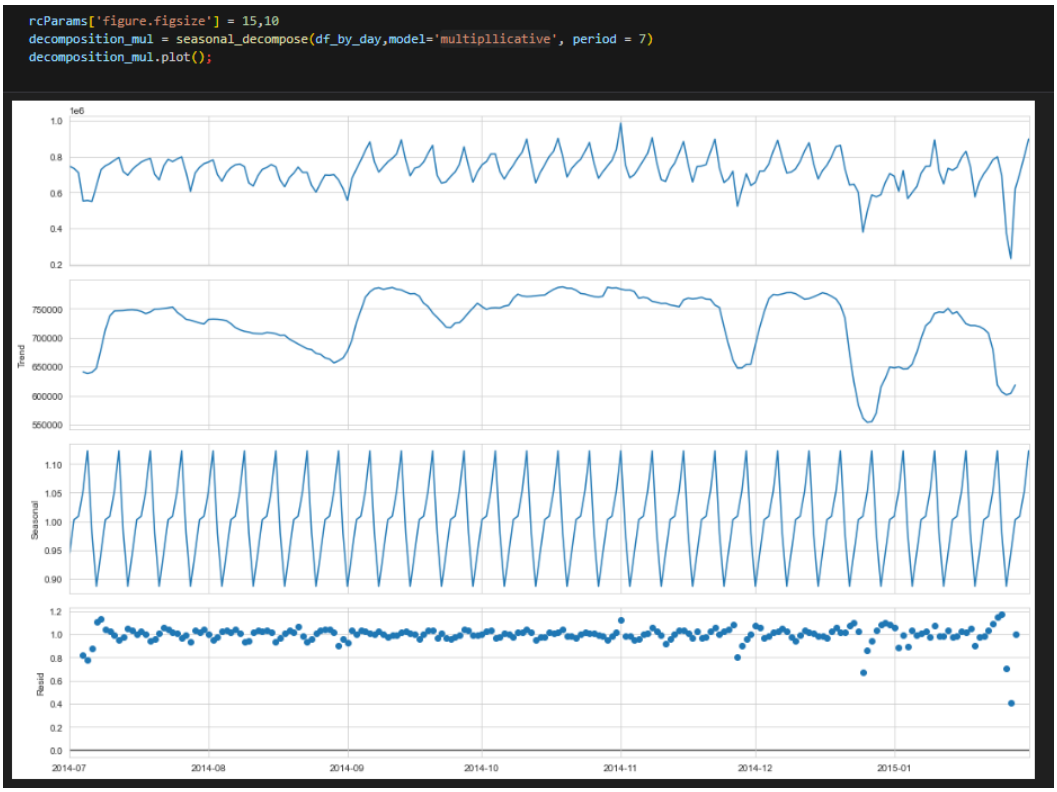
Additive Model :

```
rcParams['figure.figsize'] = 15,10
decomposition_add = seasonal_decompose(df_by_day,model='additive', period = 7)
decomposition_add.plot();
```



Multiplicative Model :

```
rcParams['figure.figsize'] = 15,10
decomposition_mul = seasonal_decompose(df_by_day,model='multipllicative', period = 7)
decomposition_mul.plot();
```



The best suited model for this dataset is additive model as seasonality isrelatively constant.

To check the stationary of the dataframe day_df, we need to perform a statistical test Dicky Fuller test.
Code :

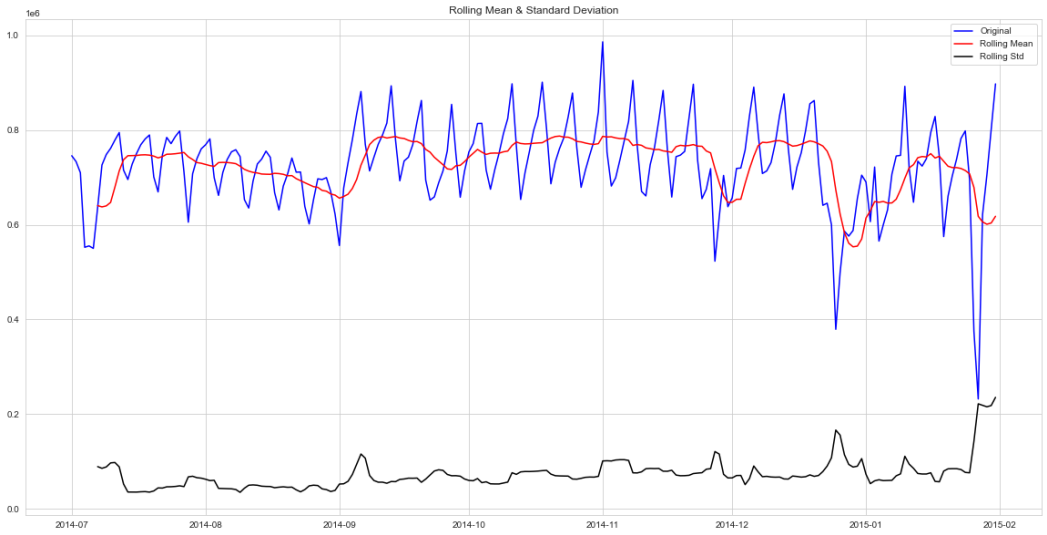
```
## Test for stationarity of the series - Dicky Fuller test
def test_stationarity(timeseries):

    #Determing rolling statistics
    rolmean = timeseries.rolling(window=7).mean() #determining the rolling mean
    rolstd = timeseries.rolling(window=7).std()   #determining the rolling standard deviation

    #Plot rolling statistics:
    orig = plt.plot(timeseries, color='blue',label='Original')
    mean = plt.plot(rolmean, color='red', label='Rolling Mean')
    std = plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean & Standard Deviation')
    plt.show(block=False) ## You plot will stay open - will be seen always

    #Perform Dickey-Fuller test:
    print ('Results of Dickey-Fuller Test:')
    dftest = adfuller(timeseries, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags Used','Number of Observations Used'])
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print (dfoutput,'\n')

    if dfoutput['p-value'] < 0.05:
        print ('Given Time Series is stationary')
    else:
        print ('Given Time Series is NOT stationary')
```

This function, test_stationarity, is used to examine the stationarity of a time series using the Dickey-Fuller test. Time series analysis relies heavily on stationarity. The function does the following tasks:

- Using a window size of 7, calculates the rolling mean and rolling standard deviation of time series data.
- To display trends, plots the original time series data, rolling mean, and rolling standard deviation.
- The Dickey-Fuller test is used to determine if a time series is stationary or not. It computes the test statistic, p-value, lag count, and critical values. The test results are output, including the test statistic and crucial values.

Determines stationarity using the p-value; if the p-value is less than 0.05, the time series is stationary; otherwise, it is not.

Here is the output of the above code:



And the **Results** are as follows,

Results of Dickey-Fuller Test:

Test Statistic	-3.448094
p-value	0.009425
#Lags Used	9.000000
Number of Observations Used	205.000000
Critical Value (1%)	-3.462658
Critical Value (5%)	-2.875744
Critical Value (10%)	-2.574341

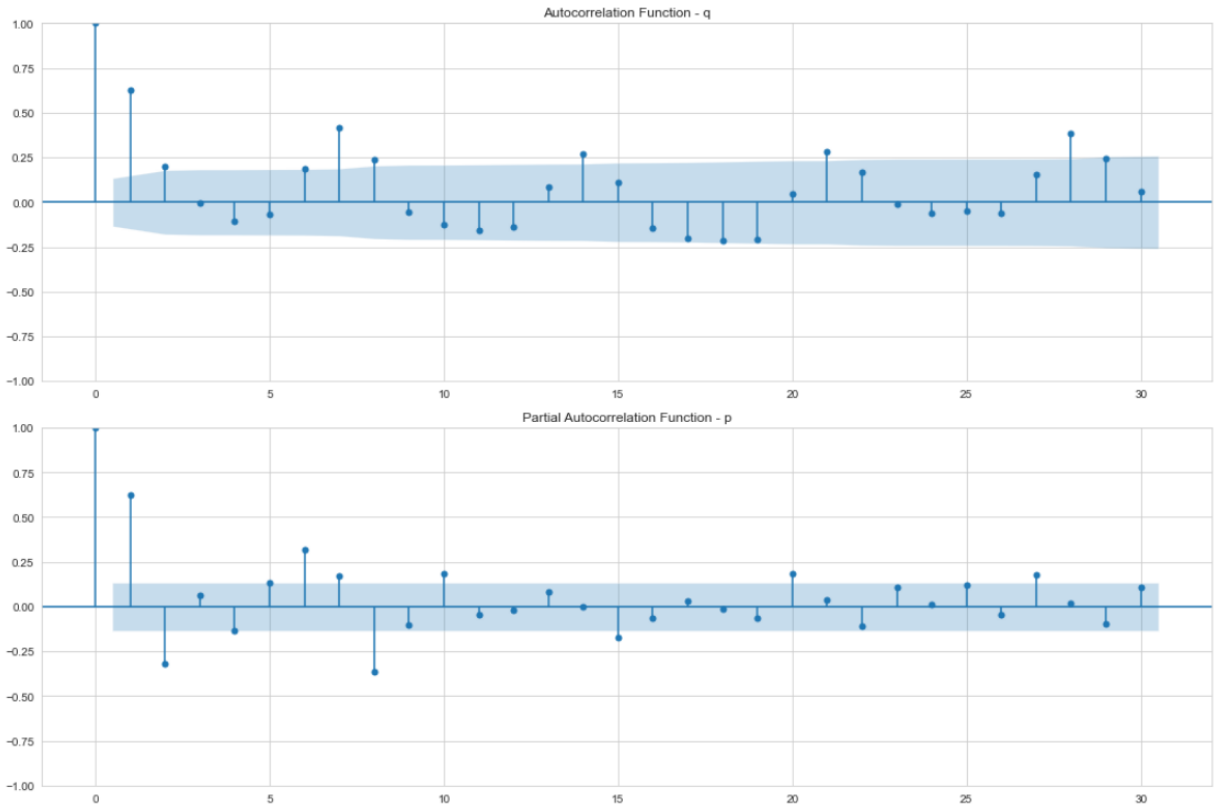
Here he P-value is 0.0094, which is less than 0.05. We can definitely say that this dataset is **stationary.**

Q 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.
- 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.

We start by getting the ACF, PACF plots, and the ARIMA model for time series forecasting. We begin by plotting ACF and PACF to get an initial sense of the data, laying the groundwork for selecting ARIMA parameters.



Now,

```
train_ts = df_by_day[df_by_day.index < '2014-12-01']
test_ts = df_by_day[df_by_day.index >= '2015-01-01']
```

```
## The following loop helps us in getting a combination of different parameters of p and q in the range of 0 and 2

import itertools
p = d = q = range(0, 3)
pdq = list(itertools.product(p, d, q))
print('Some parameter combinations for the Model...')
for i in range(0,len(pdq)):
    print('Model: {}'.format(pdq[i]))
```

Then comes the training phase, where a combination of parameters in the range of 0 to 2 for p, d, and q is employed. We cycle through all 27 combinations, fitting the ARIMA model each time and evaluating its performance based on the Mean Absolute Error. It's a straightforward, brute force approach—exhaustive but effective.

```
# Creating an empty Dataframe with column names only
results_df = pd.DataFrame(columns=['param', 'AIC', 'RMSE', 'MAPE'])

for param in pdq:
    ARIMA_model = ARIMA(train_ts['value'].values, order=param).fit()

    # Calculate RMSE
    predictions = ARIMA_model.predict()
    rmse = sqrt(mean_squared_error(train_ts['value'].values, predictions))

    # Calculate MAP
    map_error = np.mean(np.abs((train_ts['value'].values - predictions) / train_ts['value'].values)) * 100

    results_df = results_df.append({
        'param': param,
        'AIC': ARIMA_model.aic,
        'RMSE': rmse,
        'MAPE': map_error
    }, ignore_index=True)
```

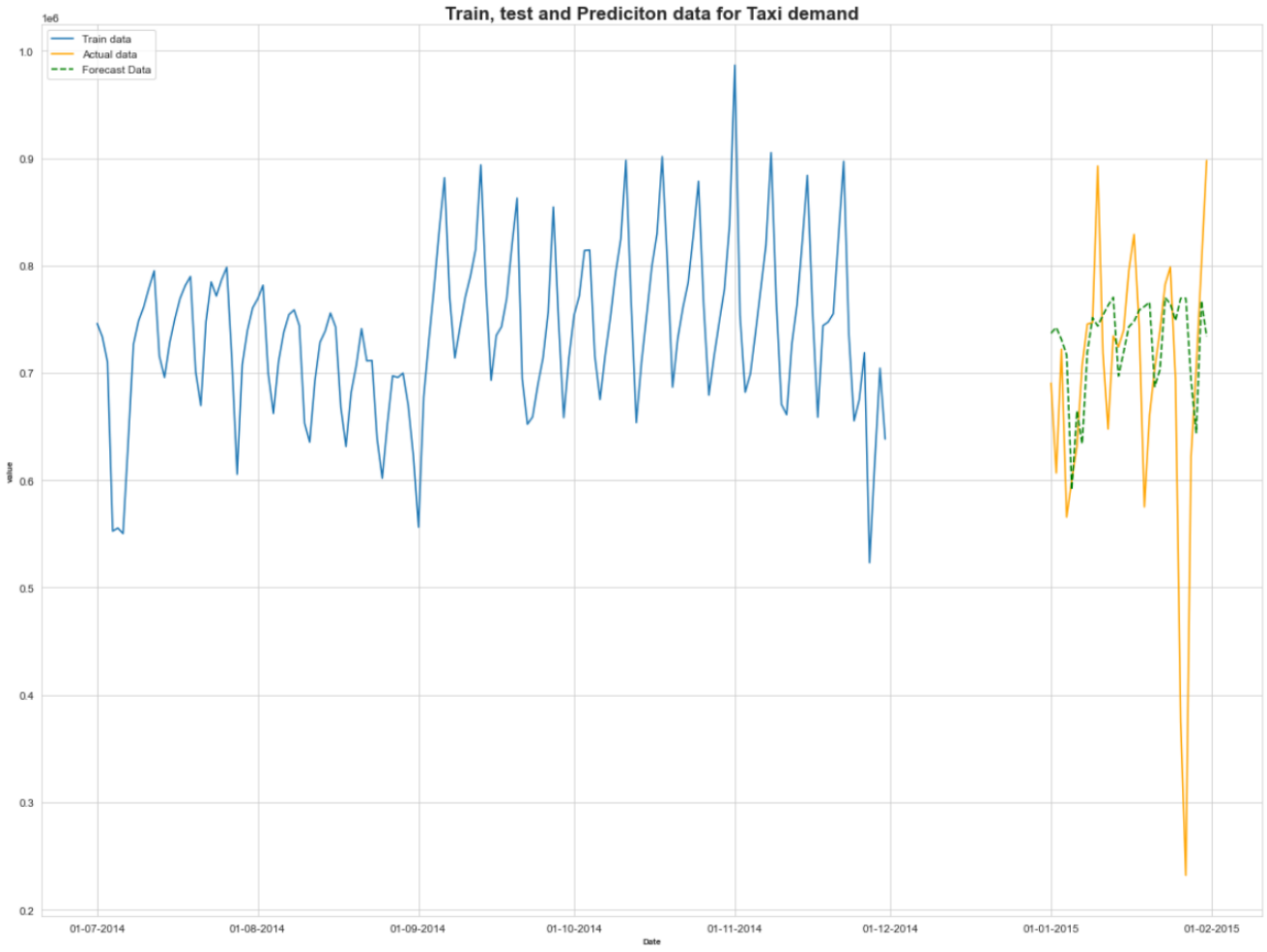
```
# Sort the dataframe by AIC values in ascending order
results_df = results_df.sort_values(by='MAPE', ascending=True)
results_df
```

However, this raises a point of contention. While the loop to try out all parameter combinations is effective, it might not be the most efficient. We could explore a more nuanced parameter tuning approach, perhaps grid search or a similar algorithm, to optimize this process. Additionally, the inclusion of seasonal parameters in the SARIMA model could provide a more accurate forecast.

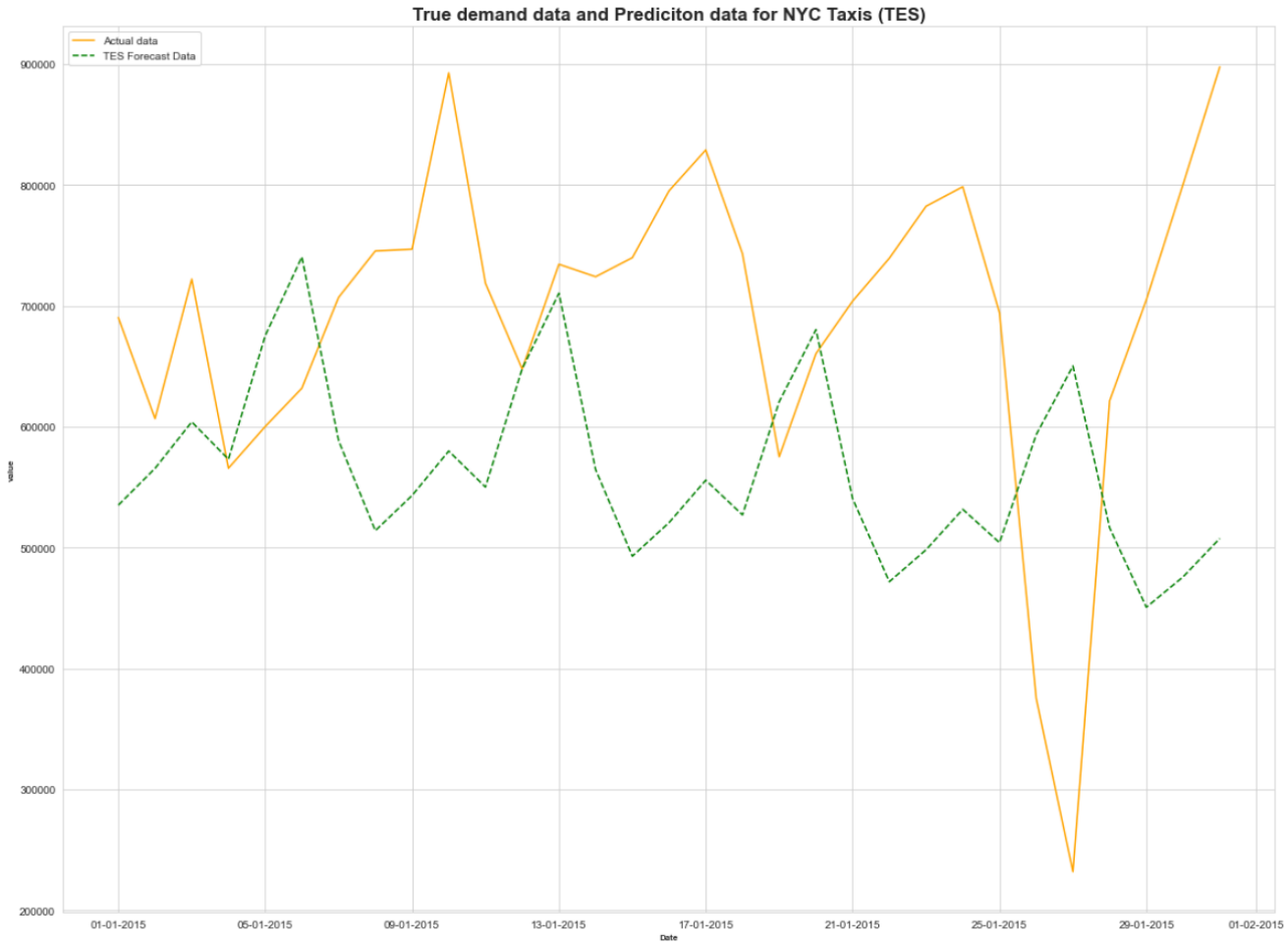
The Result:

	param	AIC	RMSE	MAPE
19	(2, 0, 1)	3791.050482	56102.043643	5.789621
2	(0, 0, 2)	3788.222455	55900.501320	5.820820
18	(2, 0, 0)	3791.098046	56489.975282	5.833000
10	(1, 0, 1)	3790.364223	56363.366074	5.905255
20	(2, 0, 2)	3787.192049	55102.659647	5.914107
11	(1, 0, 2)	3787.807998	55526.046970	5.968419
1	(0, 0, 1)	3796.133843	57874.782256	6.004911
23	(2, 1, 2)	3766.352903	81610.762886	6.140459
9	(1, 0, 0)	3809.717176	60481.823567	6.669490
22	(2, 1, 1)	3787.223496	85417.212169	6.986868
14	(1, 1, 2)	3813.129796	86179.766835	7.309855

The model's then used to predict against test data, forecasting taxi demand for January 2015. The plot vividly contrasts predicted versus actual values, offering a tangible measure of the model's accuracy. Even with the exclusion of December 2014 data the model is fairly accurate.



We also experiment with the Holt-Winters Exponential Smoothing model. It’s another potent tool in the time series forecasting arsenal but doesn’t particularly outperform the ARIMA in this instance.



Q 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.

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

In order to create above new columns, various attributes returned by datetime object are used :-

```
df_by_day['Weekday'] = df_by_day.index.day_name()
df_by_day['Hour'] = df_by_day.index.hour
df_by_day['Day'] = df_by_day.index.day
df_by_day['Month'] = df_by_day.index.month
df_by_day['Year'] = df_by_day.index.year
df_by_day['Month_day'] = df_by_day.index.day
```

For creating "lag" column, shift() function is used :-

```
df_by_day['Lag'] = df_by_day['value'].shift(1)
```

and to create "rolling_mean" column rolling() method is used with window size 7

```
df_by_day['Rolling_Mean'] = df_by_day['value'].rolling(window=7, min_periods=1).mean()
```

[81]:
```
# Create additional columns and utilise datetime funtions to get required outputs
df_by_day['Weekday'] = df_by_day.index.day_name()
df_by_day['Hour'] = df_by_day.index.hour
df_by_day['Day'] = df_by_day.index.day
df_by_day['Month'] = df_by_day.index.month
df_by_day['Year'] = df_by_day.index.year
df_by_day['Month_day'] = df_by_day.index.day
df_by_day['Lag'] = df_by_day['value'].shift(1)
df_by_day['Rolling_Mean'] = df_by_day['value'].rolling(window=7, min_periods=1).mean()

# Handle NaN after creating lag feature
df_by_day = df_by_day.dropna()

# Prepare features for isolation forst model
features = df_by_day[['value', 'Hour', 'Day', 'Month', 'Year', 'Month_day', 'Lag', 'Rolling_Mean']]
df_by_day
```

[81]:

timestamp	value	Weekday	Hour	Day	Month	Year	Month_day	Lag	Rolling_Mean	anomaly
2014-07-04	552565	Friday	0	4	7	2014	4	710142.0	631353.500000	1
2014-07-05	555470	Saturday	0	5	7	2014	5	552565.0	606059.000000	1
2014-07-06	550285	Sunday	0	6	7	2014	6	555470.0	592115.500000	1
2014-07-07	636570	Monday	0	7	7	2014	7	550285.0	601006.400000	1
2014-07-08	726535	Tuesday	0	8	7	2014	8	636570.0	621927.833333	1
...	...	...	...	...	...	...	...	...	...	...
2015-01-27	232058	Tuesday	0	27	1	2015	27	375311.0	617971.285714	-1
2015-01-28	621483	Wednesday	0	28	1	2015	28	232058.0	606190.857143	-1
2015-01-29	704935	Thursday	0	29	1	2015	29	621483.0	601270.714286	-1

For identifying the dates as outliers in df_day dataset , Isolation Forest technique is used here.

An **outlier** is nothing but a data point that differs significantly from other data points in the given dataset.

Isolation forest is a machine learning algorithm for anomaly detection.
It's an unsupervised learning algorithm that identifies anomaly by isolating outliers in the data. Isolation Forest is based on the Decision Tree algorithm. It isolates the outliers by randomly selecting a feature from the given set of features and then randomly selecting a split value between the max and min values of that feature. This random partitioning of features will produce shorter paths in trees for the anomalous data points, thus distinguishing them from the rest of the data.

Isolation Forest class takes below parameters:-

- **Number of estimators:** n_estimators refers to the number of base estimators or trees in the ensemble, i.e. the number of trees that will get built in the forest. This is an integer parameter and is optional. The default value is 100.

- **Max samples:** max_samples is the number of samples to be drawn to train each base estimator. If max_samples is more than the number of samples provided, all samples will be used for all trees. The default value of max_samples is 'auto'. If 'auto', then max_samples=min(256, n_samples)

- **Contamination:** This is a parameter that the algorithm is quite sensitive to; it refers to the expected proportion of outliers in the data set. This is used when fitting to define the threshold on the scores of the samples. The default value is 'auto'. If 'auto', the threshold value will be determined as in the original paper of Isolation Forest.

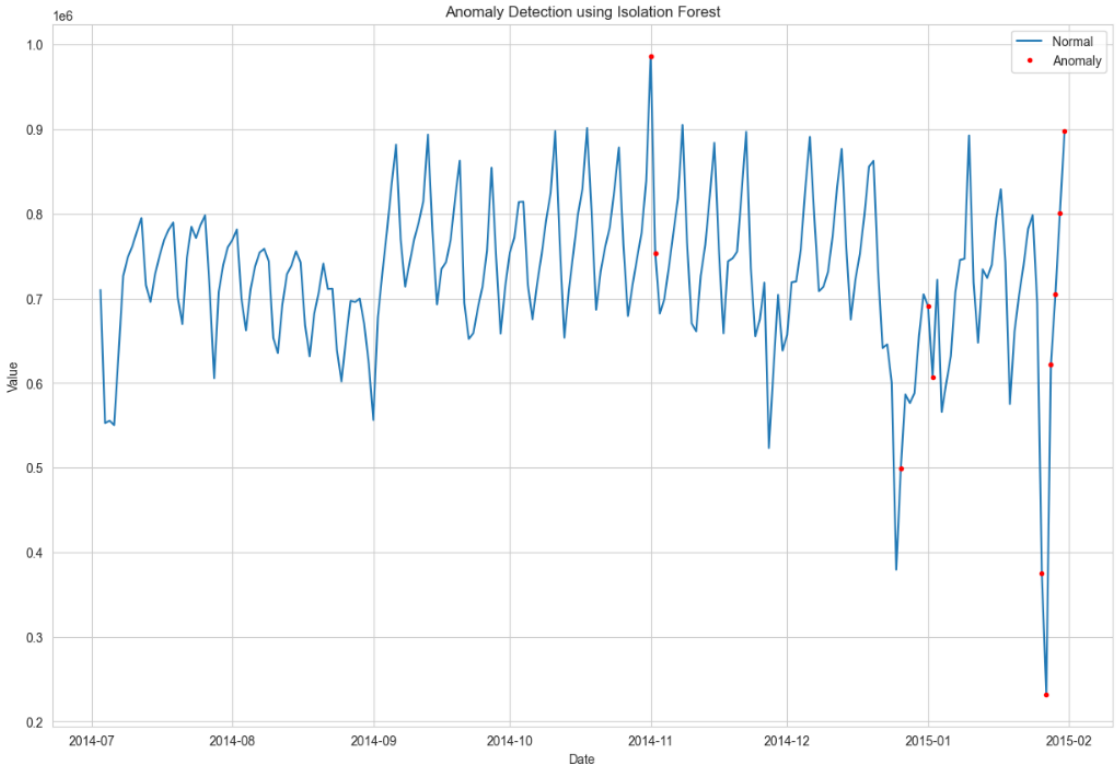
- **Max features:** All the base estimators are not trained with all the features available in the dataset. It is the number of features to draw from the total features to train each base estimator or tree. The default value of max features is one.

It finally returns the anomaly score of each sample (**score of -1 is an outlier**)

[80]:
```
# Train Isolation Forest model
model = IsolationForest(contamination=0.05)
df_by_day['anomaly'] = model.fit_predict(features)

# Print the dates identified as anomolies
print(df_by_day[df_by_day['anomaly'] == -1].index)

# Plot the anamolies
plt.figure(figsize=(15,10))
plt.plot(df_by_day.index, df_by_day['value'], label='Normal')
plt.plot(df_by_day[df_by_day['anomaly'] == -1].index,
         df_by_day[df_by_day['anomaly'] == -1]['value'], 'r.', label='Anomaly')
plt.legend()
plt.title('Anomaly Detection using Isolation Forest')
plt.xlabel('Date')
plt.ylabel('Value')
plt.show()
```

```
DatetimeIndex(['2014-11-01', '2014-11-02', '2014-12-26', '2015-01-01',
               '2015-01-02', '2015-01-26', '2015-01-27', '2015-01-28',
               '2015-01-29', '2015-01-30', '2015-01-31'],
              dtype='datetime64[ns]', name='timestamp', freq=None)
```

```
 '2015-01-02', '2015-01-26', '2015-01-27', '2015-01-28',
 '2015-01-29', '2015-01-30', '2015-01-31'],
dtype='datetime64[ns]', name='timestamp', freq=None)
```



The outlier dates are - **'2014-11-01', '2014-11-02', '2014-12-26', '2015-01-01','2015-01-02', '2015-01-26', '2015-01-27', '2015-01-28', '2015-01-29', '2015-01-30', '2015-01-31'**

CLOSING THOUGHTS

The main takeaway from this project would be that even though each team member had their own methodology and logic to reach the solution through discussion we were able to identify which methodology and logic is most efficient in given scenario and then settle upon that as our collective answer to the question while also double checking each other's work. Some parts of the project had to be compartmentalized to save time which resulted in individual effort for that particular part of the project i.e.:

The contribution of each team member can be quantified as follows:

Team Member	Code Contribution	Report Contribution	Video Contribution	Code and Methodology discussion
Bhanu Pratap Reddy	*All Questions*	*1.8, 1.9, 1. 10 and 2.2*	*1.8, 1.9, 1. 10 and 2.2*	*All Questions*
Harpreet Singh	*All Questions*	*1.5, 1.6, 1.7 and 2.3*	*1.5, 1.6, 1.7 and 2.3*	*All Questions*
Sai Siva Sarma	*All Questions*	*1.1, 1.2, 1.3, 1.4 and 2.1*	*1.1, 1.2, 1.3, 1.4 and 2.1*	*All Questions*