Table of Content

Table of Tables

1. Data Understanding

The data set used in this coursework is related to the "NYC 311 Customer Service Request Analysis." It includes records of non-emergency service requests made to New York City's 311 system, which is a public hotline for citizens to report issues or concerns in their locality. The dataset reveals common complaints such as noise disturbances and blocked driveways. It also contains detailed information like the dates when the requests were created and closed, the nature of the complaints, location data, and the duration of the problems. This dataset is structured to support analysis of trends in customer requests, the responsiveness of relevant authorities, and the management of complaints in New York City, offering valuable insights into urban service management.

The table below shows the keys column from the dataset:

Table 1 Key Columns Description in dataset

S. No	Column Name	Description	Data Type
1	Unique Id	It is a unique identifier for every request submitted.	Int64
2	Created Date	It displays the date and time when the service request was submitted.	Object
3	Closed Date	It indicates the date and time when the service request was resolved.	Object
4	Complaint Type	It specifies the type of issue or complaint.	Object

5	Location	It indicates the location or area from which the request was made.	Object
6	Agency	It specifies the authority responsible for handling the request.	Object
7	Status	It indicates the present status of the request.	Object

2. Data Preparation

a. Importing the dataset

[31]:
```
# Importing essential libraries for data manipulation, visualization, and statistical analysis

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
```

[35]:
```
# Loading the dataset containing customer service requests

df = pd.read_csv('Customer Service_Requests_from_2010_to_Present.csv')
df.head()
```

```
C:\Users\bhand\AppData\Local\Temp\ipykernel_10640\3771020419.py:3: DtypeWarning: Columns (48,49) have mixed types. Specify dtype option on import or set low_memory=False.
  df = pd.read_csv('Customer Service_Requests_from_2010_to_Present.csv')
```

[35]:

	Unique Key	Created Date	Closed Date	Agency	Agency Name	Complaint Type	Descriptor	Location Type	Incident Zip	Incident Address	...	Bridge Highway Name	Bridge Highway Direction	Road Ramp	Bridge Highway Segment	Garag L Nan
0	32310363	12/31/2015 11:59:45 PM	01-01-16 0:55	NYPD	New York City Police Department	Noise - Street/Sidewalk	Loud Music/Party	Street/Sidewalk	10034.0	71 VERMILYEA AVENUE	...	NaN	NaN	NaN	NaN	Na
1	32309934	12/31/2015 11:59:44 PM	01-01-16 1:26	NYPD	New York City Police Department	Blocked Driveway	No Access	Street/Sidewalk	11105.0	27-07 23 AVENUE	...	NaN	NaN	NaN	NaN	Na
2	32309159	12/31/2015 11:59:29 PM	01-01-16 4:51	NYPD	New York City Police Department	Blocked Driveway	No Access	Street/Sidewalk	10458.0	2897 VALENTINE AVENUE	...	NaN	NaN	NaN	NaN	Na
3	32305098	12/31/2015 11:57:46 PM	01-01-16 7:43	NYPD	New York City Police Department	Illegal Parking	Commercial Overnight Parking	Street/Sidewalk	10461.0	2940 BAISLEY AVENUE	...	NaN	NaN	NaN	NaN	Na
4	32306529	12/31/2015 11:56:58 PM	01-01-16 3:24	NYPD	New York City Police Department	Illegal Parking	Blocked Sidewalk	Street/Sidewalk	11373.0	87-14 57 ROAD	...	NaN	NaN	NaN	NaN	Na

5 rows × 53 columns

In the above photo, I have imported the required python libraries for analysis, and I have loaded the NYC 311 dataset. Loaded the CSV file. The head () function in the code displays the first five rows of the dataset along with its columns. This output confirms that the dataset has been successfully loaded and offers a quick preview of its structure.

b. Provide Insights on the Information

```
[41]: # Displaying information about the data set

      df.info()
      df.shape
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 300698 entries, 0 to 300697
Data columns (total 53 columns):
 #   Column                          Non-Null Count   Dtype
---  ------                          --------------   -----
 0   Unique Key                      300698 non-null  int64
 1   Created Date                    300698 non-null  object
 2   Closed Date                     298534 non-null  object
 3   Agency                          300698 non-null  object
 4   Agency Name                     300698 non-null  object
 5   Complaint Type                  300698 non-null  object
 6   Descriptor                      294784 non-null  object
 7   Location Type                   300567 non-null  object
 8   Incident Zip                    298083 non-null  float64
 9   Incident Address                256288 non-null  object
 10  Street Name                     256288 non-null  object
 11  Cross Street 1                  251419 non-null  object
 12  Cross Street 2                  250919 non-null  object
 13  Intersection Street 1           43858 non-null   object
 14  Intersection Street 2           43362 non-null   object
 15  Address Type                    297883 non-null  object
 16  City                            298084 non-null  object
 17  Landmark                        349 non-null     object
 18  Facility Type                   298527 non-null  object
 19  Status                          300698 non-null  object
 20  Due Date                        300695 non-null  object
 21  Resolution Description          300698 non-null  object
 22  Resolution Action Updated Date  298511 non-null  object
 23  Community Board                 300698 non-null  object
 24  Borough                         300698 non-null  object
 25  X Coordinate (State Plane)      297158 non-null  float64
 26  Y Coordinate (State Plane)      297158 non-null  float64
 27  Park Facility Name              300698 non-null  object
 28  Park Borough                    300698 non-null  object
 29  School Name                     300698 non-null  object
 30  School Number                   300698 non-null  object
 31  School Region                   300697 non-null  object
 32  School Code                     300697 non-null  object
 33  School Phone Number             300698 non-null  object
 34  School Address                  300698 non-null  object
 35  School City                     300698 non-null  object
 36  School State                    300698 non-null  object
 37  School Zip                      300697 non-null  object
 38  School Not Found                300698 non-null  object
 39  School or Citywide Complaint    0 non-null       float64
 40  Vehicle Type                    0 non-null       float64
 41  Taxi Company Borough            0 non-null       float64
 42  Taxi Pick Up Location           0 non-null       float64
 43  Bridge Highway Name             243 non-null     object
 44  Bridge Highway Direction        243 non-null     object
 45  Road Ramp                       213 non-null     object
 46  Bridge Highway Segment          213 non-null     object
 47  Garage Lot Name                 0 non-null       float64
 48  Ferry Direction                 1 non-null       object
 49  Ferry Terminal Name             2 non-null       object
 50  Latitude                        297158 non-null  float64
 51  Longitude                       297158 non-null  float64
 52  Location                        297158 non-null  object
dtypes: float64(10), int64(1), object(42)
memory usage: 121.6+ MB
[41]: (300698, 53)
```

The info () function reveals that the dataset contains 300,698 rows and 53 columns, with data types such as int64, object, and float64 depending on the column. The output from the `shape` attribute further confirms the overall dimensions of the dataset.

c. **Convert the columns "Created Date" and "Closed Date" to datetime datatype and create a new column "Request_Closing_Time" as the time elapsed between request creation and request closing.**

[45]:
```
# Converting 'Created Date' and 'Closed Date' columns to datetime format
df['Created Date'] = pd.to_datetime(df['Created Date'])
df['Closed Date'] = pd.to_datetime(df['Closed Date'])

# Calculating the request resolution time in hours and storing it in a new column
df['Request_Closing_Time'] = (df['Closed Date'] - df['Created Date']).dt.total_seconds() / 3600

# Displaying the first few rows to verify the new 'Request_Closing_Time' column
df[['Created Date', 'Closed Date', 'Request_Closing_Time']].head()
```

[45]:

	Created Date	Closed Date	Request_Closing_Time
0	2015-12-31 23:59:45	2016-01-01 00:55:00	0.920833
1	2015-12-31 23:59:44	2016-01-01 01:26:00	1.437778
2	2015-12-31 23:59:29	2016-01-01 04:51:00	4.858611
3	2015-12-31 23:57:46	2016-01-01 07:43:00	7.753889
4	2015-12-31 23:56:58	2016-01-01 03:24:00	3.450556

The code shown in the attached screenshot converts the "Converted Date" and "Closed Date" columns into datetime objects using pd.to_datetime(). It then calculates the "Request_Closing_Time" by finding the difference between the Created Date and Closed Date and converts this duration into hours for easier interpretation. The output displays the first five rows, confirming that the conversion and calculation were successfully carried out.

d. **Write a python program to drop irrelevant Columns.**

```
[51]: # Defining a list of unnecessary columns to be removed from the dataset
columns_to_drop = ['Agency Name','Incident Address','Street Name','Cross Street 1','Cross Street 2',
                   'Intersection Street 1','Intersection Street 2','Address Type','Park Facility Name',
                   'Park Borough','School Name','School Number','School Region','School Code',
                   'School Phone Number','School Address','School City','School State','School Zip',
                   'School Not Found','School or Citywide Complaint','Vehicle Type','Taxi Company Borough',
                   'Taxi Pick Up Location','Bridge Highway Name','Bridge Highway Direction','Road Ramp',
                   'Bridge Highway Segment','Garage Lot Name','Ferry Direction','Ferry Terminal Name',
                   'Landmark','X Coordinate (State Plane)','Y Coordinate (State Plane)','Due Date',
                   'Resolution Action Updated Date','Community Board','Facility Type','Location']

# Filtering out only those columns from the list that actually exist in the current DataFrame
columns_to_drop = list(set(columns_to_drop).intersection(set(df.columns)))

# Dropping the selected columns from the DataFrame
df_cleaned = df.drop(columns=columns_to_drop)

# Displaying the remaining columns and their count
print("The remaining columns are:")
print(df_cleaned.columns.tolist())
print(f"Number of columns remaining: {df_cleaned.shape[1]}")

# Displaying the first few rows of the cleaned dataset
df_cleaned.head()
```

```
The remaining columns are:
['Unique Key', 'Created Date', 'Closed Date', 'Agency', 'Complaint Type', 'Descriptor', 'Location Type', 'Incident Zip', 'City', 'Status', 'Resolution De
scription', 'Borough', 'Latitude', 'Longitude', 'Request_Closing_Time']
Number of columns remaining: 15
```

The code in the attached screenshot removes the specified irrelevant columns. It also uses the `intersection` method to prevent errors in case some columns are missing from the dataframe. The output shows a list of the remaining relevant columns.

e. Write a python program to remove the NaN missing values from updated dataframe.

```
[55]: # Checking the number of missing (NaN) values in each column before cleaning
df_cleaned.isnull().sum()

# Storing and displaying the original shape of the dataset before dropping missing values
original_shape = df_cleaned.shape
print(f"Original shape: {original_shape}")

# Dropping all rows that contain any missing values
df_cleaned = df_cleaned.dropna()

# Storing and displaying the shape of the dataset after dropping missing values
new_shape = df_cleaned.shape
print(f"New shape: {new_shape}")
print(f"Number of rows removed: {original_shape[0] - new_shape[0]}")

# Verifying that all missing values have been removed
print("Missing values after cleaning:")
df_cleaned.isnull().sum()
```

```
Original shape: (291107, 15)
New shape: (291107, 15)
Number of rows removed: 0
Missing values after cleaning:
[55]: Unique Key              0
Created Date            0
Closed Date             0
Agency                  0
Complaint Type          0
Descriptor              0
Location Type           0
Incident Zip            0
City                    0
Status                  0
Resolution Description  0
Borough                 0
Latitude                0
Longitude               0
Request_Closing_Time    0
dtype: int64
```

The code in the attached screenshot eliminates all NaN (missing) values from the dataframe. To do this, it first checks for any NaN values, then creates a new dataframe to retain the original shape before removal. It then replaces the rows containing NaN values and checks the updated shape. The output confirms that there are no missing values left in the dataframe after the NaN values are removed.

f. Write a python program to see the unique values from all the columns in the dataframe.

```
[59]: # Displaying the number of unique values for each column and listing them if they are few

for column in df_cleaned.columns:
    unique_values = df_cleaned[column].nunique()
    print(f"\nColumn: {column}")
    print(f"Number of unique values: {unique_values}")

    # Displaying actual unique values and their counts if the number is manageable
    if unique_values < 30:
        print("Value counts:")
        value_counts = df_cleaned[column].value_counts().sort_values(ascending=False)
        print(value_counts)
    else:
        # If too many unique values, display a sample instead
        print(f"Too many unique values to display. Sample values: {df_cleaned[column].sample(5).tolist()}")
```

```
Column: Unique Key
Number of unique values: 291107
Too many unique values to display. Sample values: [31163753, 31262119, 32140184, 31414672, 32232101]

Column: Created Date
Number of unique values: 251970
Too many unique values to display. Sample values: [Timestamp('2015-11-09 11:19:00'), Timestamp('2015-07-22 19:35:34'), Timestamp('2015-09-12 16:46:0
0'), Timestamp('2015-11-05 18:42:00'), Timestamp('2015-05-11 21:38:00')]

Column: Closed Date
Number of unique values: 231991
Too many unique values to display. Sample values: [Timestamp('2015-11-28 21:56:42'), Timestamp('2015-09-01 00:06:00'), Timestamp('2015-09-13 21:32:2
8'), Timestamp('2015-05-03 20:22:00'), Timestamp('2015-06-22 01:18:26')]

Column: Agency
Number of unique values: 1
Value counts:
```

The code in the attached screenshot displays the unique values from all columns in the dataset. It iterates through each column in the df_cleaned dataframe to show the number of unique values along with the values themselves. For each column, it first prints the column name and the total number of unique values using nunique(). If a column has fewer than 30 unique values, it lists all unique values along with their frequencies, sorted from most to least frequent using value_counts(). For columns with 30 or more unique values, it instead shows a random sample of 5 values using sample(5) to provide an overview of the data without overwhelming the output.

3. Data Analysis

a. **Write a Python program to show summary statistics of sum, mean, standard deviation, skewnezss, and kurtosis of the data frame.**

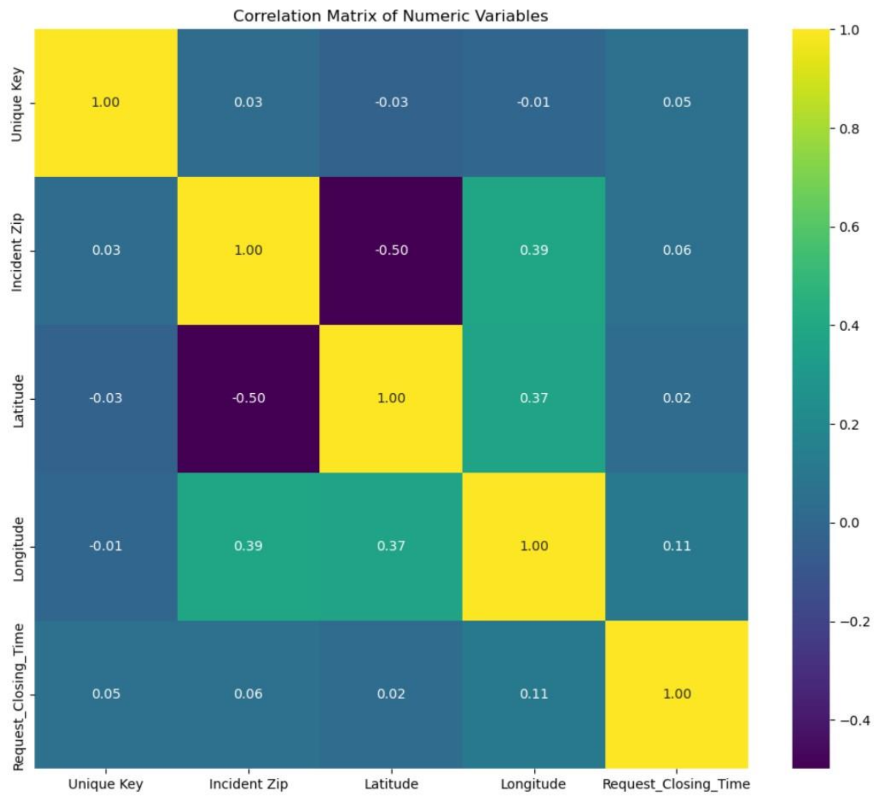
```
[63]: # Selecting only numerical columns from the cleaned dataset
numeric_df = df_cleaned.select_dtypes(include=['number'])

# Calculating various statistical measures
sum = numeric_df.sum()
mean = numeric_df.mean()
std = numeric_df.std()
skew = numeric_df.skew()
kurt = numeric_df.kurtosis()

# Creating a summary DataFrame to display all statistics together
summary_stats = pd.DataFrame({
    'Sum': sum,
    'Mean': mean,
    'Std Dev': std,
    'Skewness': skew,
    'Kurtosis': kurt
})

# Displaying the summary statistics
print("Summary Statistics:")
print(summary_stats)
```

```
Summary Statistics:
                              Sum          Mean      Std Dev   Skewness  \
Unique Key           9.112108e+12  3.130158e+07  575377.738707   0.016898
Incident Zip         3.160833e+09  1.085798e+04     580.280774  -2.553956
Latitude             1.185553e+07  4.072568e+01       0.082411   0.123114
Longitude           -2.152010e+07 -7.392504e+01       0.078654  -0.312739
Request_Closing_Time 1.254358e+06  4.308926e+00       6.062641  14.299525

                       Kurtosis
Unique Key            -1.176593
Incident Zip          37.827777
Latitude              -0.734818
Longitude              1.455600
Request_Closing_Time 849.777081
```

The code in the attached screenshot displays summary statistics including the sum, mean, standard deviation, skewness, and kurtosis for the numeric columns in the dataframe. To achieve this, the `stats` library from SciPy is imported, and its functions are used to calculate and show the desired results from the dataframe.

b. **Write a Python program to calculate and show correlation of all variables.**

```
[29]: # Extracting only numeric columns from the cleaned DataFrame
numeric_columns = df_cleaned.select_dtypes(include=['number'])

# Computing the correlation matrix
corr_matrix = numeric_columns.corr()

# Plotting the heatmap of the correlation matrix
plt.figure(figsize=(12, 10))
sns.heatmap(data=corr_matrix, annot=True, cmap='viridis', fmt='.2f')
plt.title('Correlation Matrix of Numeric Variables')
plt.show()

print("Correlation:")
corr_matrix
```



	Unique Key	Incident Zip	Latitude	Longitude	Request_Closing_Time
Unique Key	1.000000	0.025492	-0.032613	-0.008621	0.053126
Incident Zip	0.025492	1.000000	-0.499081	0.385934	0.057182
Latitude	-0.032613	-0.499081	1.000000	0.368819	0.024497
Longitude	-0.008621	0.385934	0.368819	1.000000	0.109724
Request_Closing_Time	0.053126	0.057182	0.024497	0.109724	1.000000

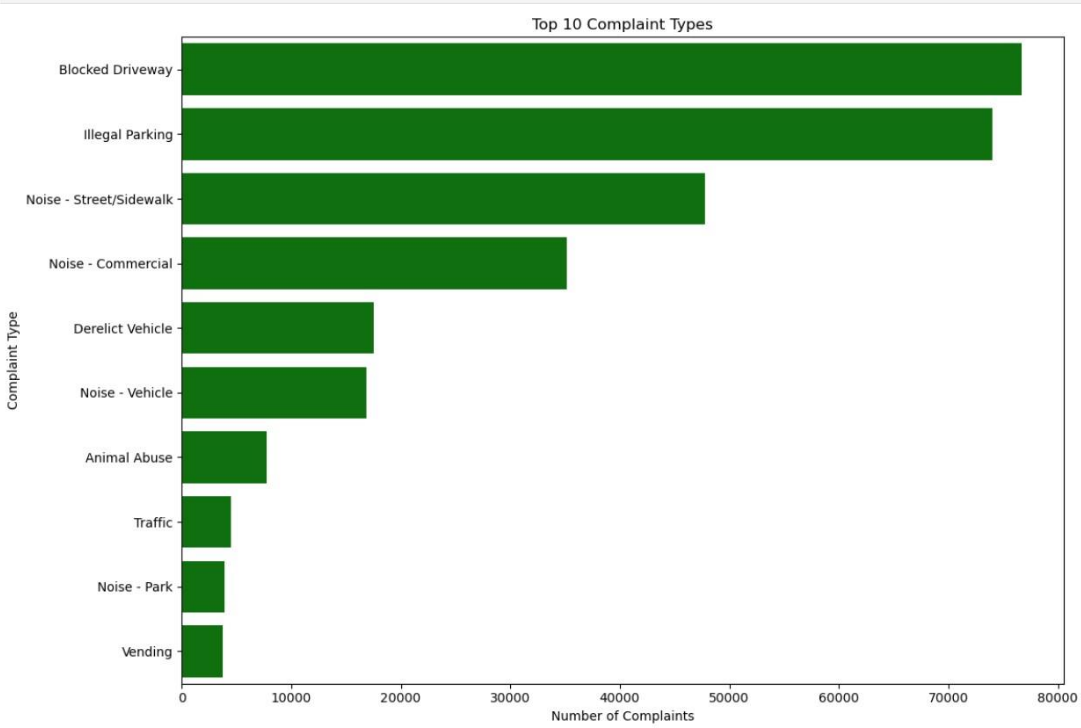
The code in the attached screenshot calculates and displays the correlation of the numeric columns in the dataframe. This is done using the `corr()` function, which computes the correlation between the numeric columns. The results are presented in the tabular format above.

4. Data Exploration

a. Provide four major insights through visualization that you come up after data mining.

Insight 1:

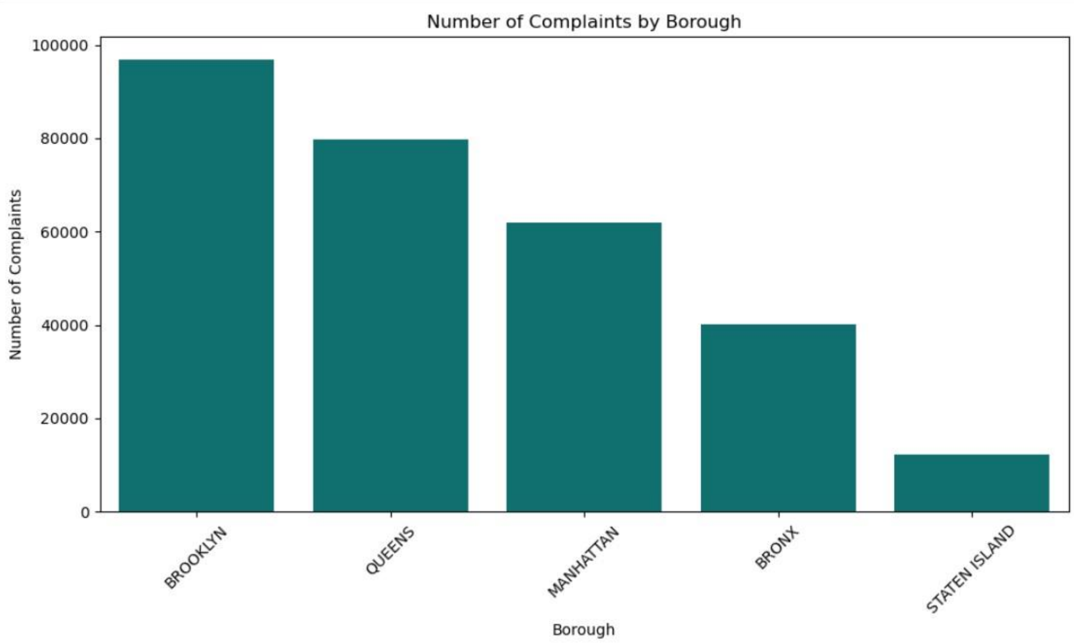
```
[37]: # Insight 1: Visualization of top complaint types

plt.figure(figsize=(12, 8))

# Identifying the 10 most frequent complaint types
most_common_complaints = df_cleaned['Complaint Type'].value_counts().nlargest(10)

# Creating a bar plot with a specific color
sns.barplot(
    x=most_common_complaints.values,
    y=most_common_complaints.index,
    color='green'
)

plt.title('Top 10 Complaint Types')
plt.xlabel('Number of Complaints')
plt.tight_layout()
plt.savefig('top_complaints.png')
plt.show()
```



This code creates a visualization showing the ten most frequent complaint types in a cleaned dataset. First, it sets the figure size to 12 by 8 inches. Then, it calculates how often each complaint type appears and selects the top ten. A horizontal bar chart is drawn using Seaborn, where the xaxis represents the number of complaints, the y-axis lists the complaint types, and the bars are colored green. A title is added to the plot along with a label for the x-axis. The layout is adjusted to prevent overlapping elements, the chart is saved as a PNG image named "top_complaints.png", and finally, the plot is displayed.

Insight 2:

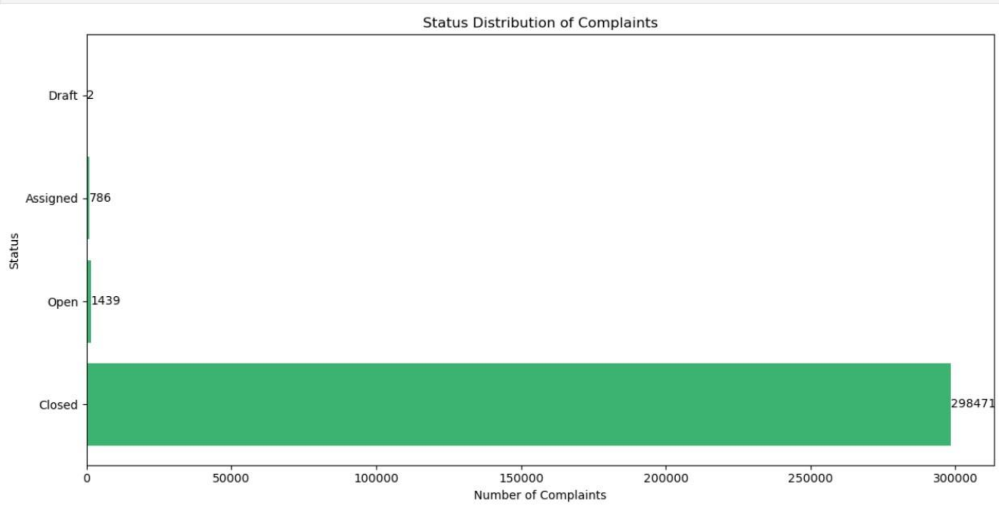
```
[41]: # INSIGHT 2: Complaint Distribution by Borough

plt.figure(figsize=(10, 6))
complaints_by_borough = df_cleaned['Borough'].value_counts()

# Creating bar plot for borough-wise complaint distribution with custom color
sns.barplot(
    x=complaints_by_borough.index,
    y=complaints_by_borough.values,
    color='teal'
)

plt.title('Number of Complaints by Borough')
plt.ylabel('Number of Complaints')
plt.xticks(rotation=45)
plt.tight_layout()
plt.savefig('borough_complaints.png')
plt.show()
```



The code shows borough complaint distribution with bar visual elements. Using the value_counts() method the procedure identifies the borough complaint rates within df_cleaned. Seaborn utilizes barplot() to display the figure which shows borough names on the x-axis while complaint counts appear on the y-axis. The visual display of the bar chart elements utilizes teal color to enhance readability. The plot becomes more readable when the 10x6 inch figure dimension includes labels alongside a rotated x-axis. Last in the analysis sequence 'Number of Complaints by Borough' displays through 'borough_complaints.png' while using plt.show(). The graphical display aids quick determination of boroughs with the highest complaint rates for assessing community service problems and public dissatisfaction.

Insight 3:

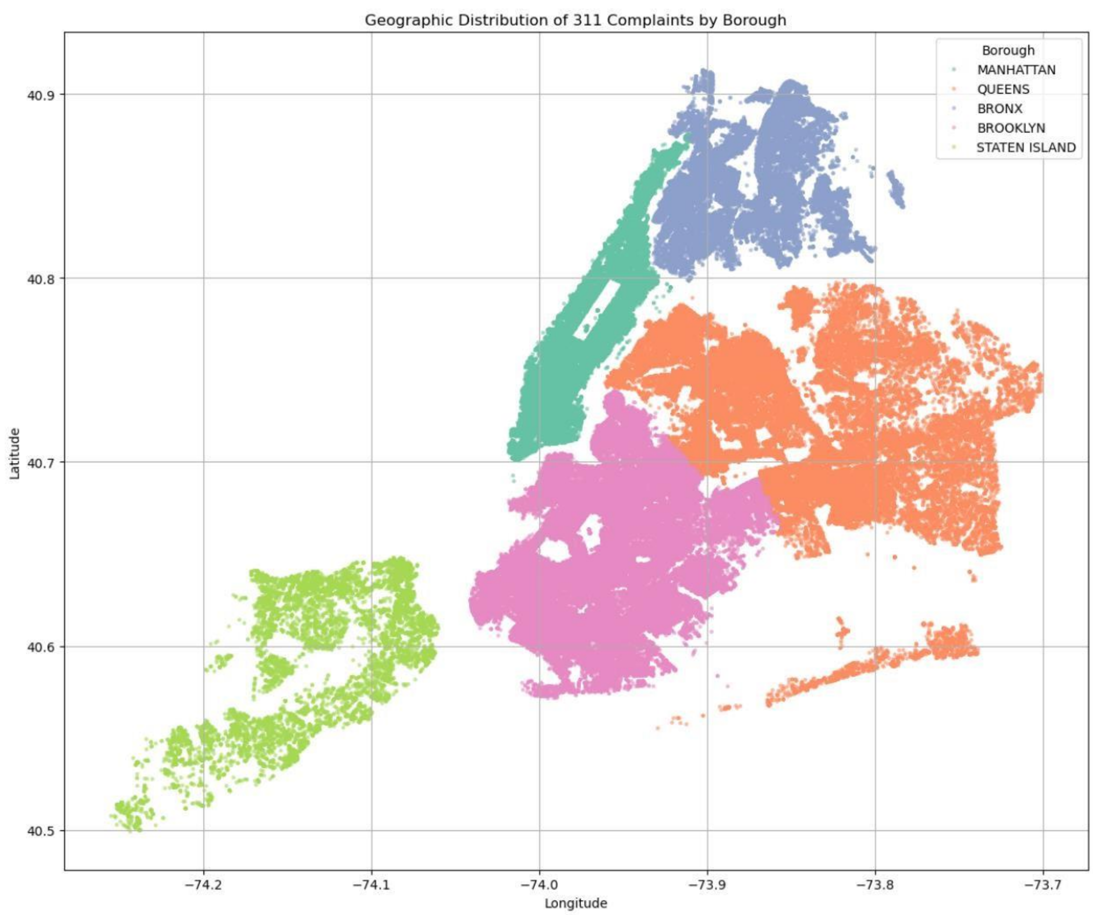
```
[45]: # INSIGHT 3: Status Distribution of Complaints

plt.figure(figsize=(12, 6))

# Counting complaints by their status
complaint_status_counts = df['Status'].value_counts()

# Creating horizontal bar plot with a new color
plt.barh(complaint_status_counts.index, complaint_status_counts.values, color='mediumseagreen')
plt.xlabel('Number of Complaints')
plt.ylabel('Status')
plt.title('Status Distribution of Complaints')

# Adding data labels to bars
for i, count in enumerate(complaint_status_counts.values):
    plt.text(count, i, str(count), va='center')

plt.tight_layout()
plt.savefig('status_distribution.png')
plt.show()
```



The code produces a graphical representation which shows complaint status distributions in the database through horizontal bar elements. Measuring 12 by 6 inches in size the figure appears by running plt.figure(figsize=(12, 6)). The program determines 'Status' column unique values frequency using df\['Status'].value_counts() and stores this result in complaint_status_counts. The bar chart displays complaint statuses along the y-axis and their count frequencies on the xaxis using mediumseagreen colors for all bars. The x-axis and y-axis labels together with the title appear through plt.xlabel(), plt.ylabel() and plt.title() functions. The for-loop assigns bar end labels which display specific count numbers to each bar while plt.tight_layout() enables all figure elements to fit properly without any element overlapping. The saved plot appears as 'status_distribution.png' through plt.savefig() and the chart displays through plt.show() to show immediately.

Insight 4:

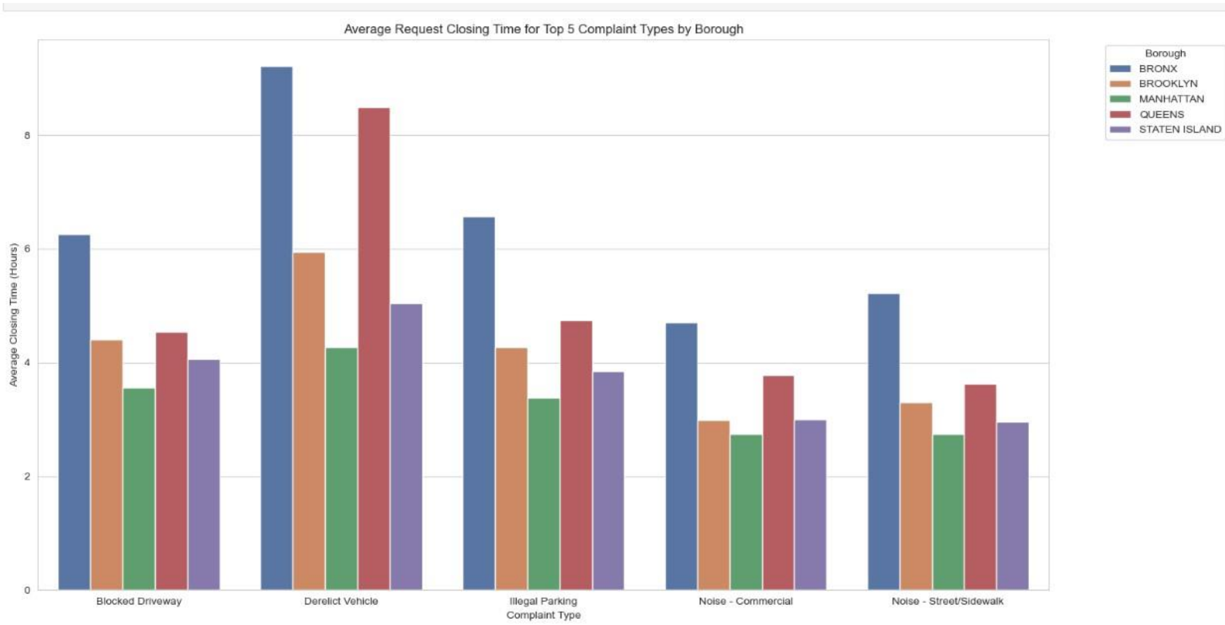
```
•[49]: # INSIGHT 4: Geographic Distribution of Complaints by Borough

plt.figure(figsize=(12, 10))

# Filtering out rows with missing geographical or borough data
filtered_df = df_cleaned.dropna(subset=['Latitude', 'Longitude', 'Borough'])

# Scatter plot showing complaint distribution by geographic coordinates with a new color palette
sns.scatterplot(
    x='Longitude',
    y='Latitude',
    data=filtered_df,
    hue='Borough',
    palette='Set2',
    alpha=0.6,
    s=10,
    edgecolor=None,
    linewidth=0
)

plt.title('Geographic Distribution of 311 Complaints by Borough')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.grid(True)
plt.legend(title='Borough', loc='upper right')
plt.tight_layout()
plt.savefig('geographic_distribution_by_borough.png')
plt.show()
```



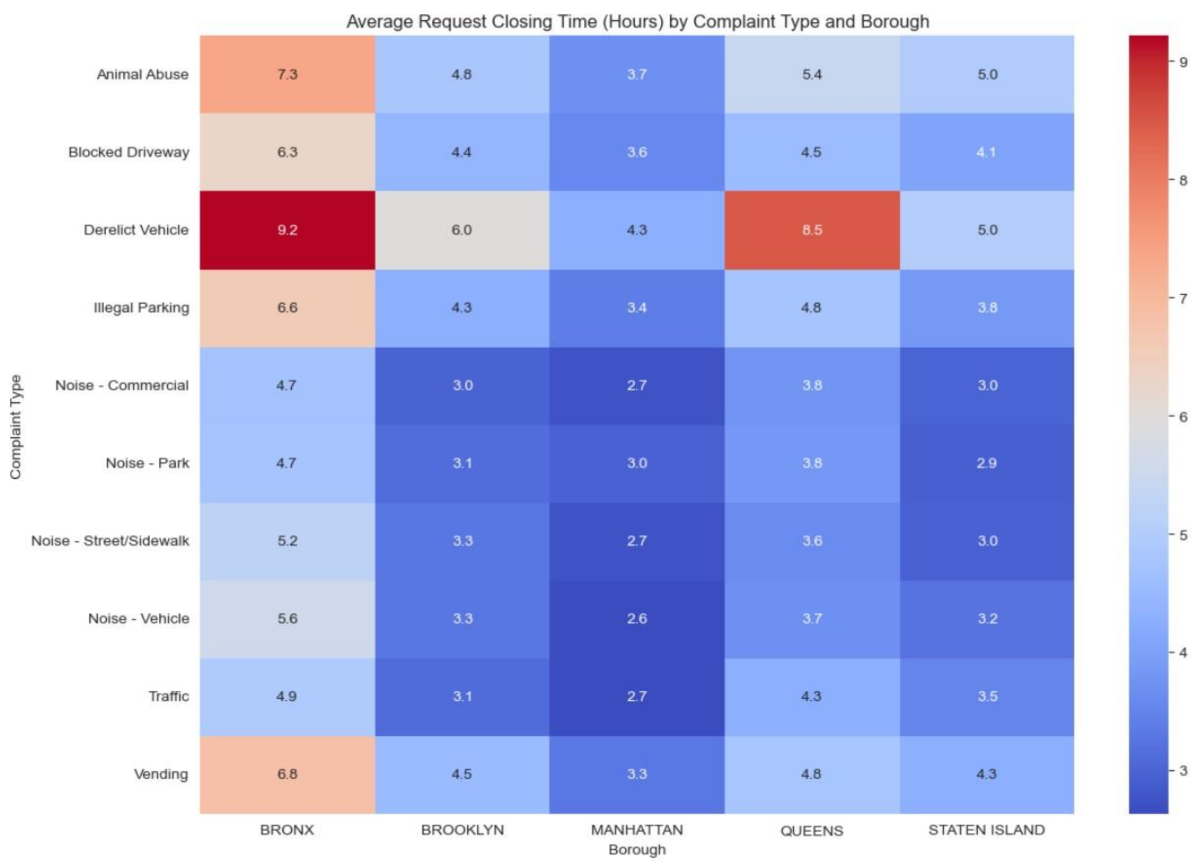
Through its execution the code displays scattered points to show how 311 complaints are geographically distributed between NYC boroughs while using location data. The program first eliminates missing information and after that it utilizes `sns.scatterplot()` to create a map displaying complaint positions which are colored according to the 'Set2' palette based on borough. The plot contains visual elements such as a title and axis labels and legend and it saves the image before presentation. The visual presentation shows where complaints exist throughout different geographical regions.

ii. Arrange the complaint types according to their average 'Request_Closing_Time', categorized by various locations. Illustrate it through graph as well.

```
•[53]: # Step 1: Calculate average closing time grouped by Borough and Complaint Type
closing_time_avg = df_cleaned.groupby(['Borough', 'Complaint Type'])['Request_Closing_Time'].mean().reset_index()

# Step 2: Identify the top 5 most frequent complaint types
most_frequent_complaints = df_cleaned['Complaint Type'].value_counts().nlargest(5).index

# Step 3: Filter data to include only those top 5 complaint types
top_complaints_data = closing_time_avg[closing_time_avg['Complaint Type'].isin(most_frequent_complaints)]

# Step 4: Create grouped bar plot with updated color palette
plt.figure(figsize=(16, 8))
sns.set_style("whitegrid")

sns.barplot(
    data=top_complaints_data,
    x='Complaint Type',
    y='Request_Closing_Time',
    hue='Borough',
    palette='deep'  # Changed from Set2 to deep for a more vibrant look
)

plt.title('Average Request Closing Time for Top 5 Complaint Types by Borough')
plt.xlabel('Complaint Type')
plt.ylabel('Average Closing Time (Hours)')
plt.legend(title='Borough', bbox_to_anchor=(1.05, 1), loc='upper left')
plt.tight_layout()
plt.savefig('avg_response_time_grouped_bar.png')
plt.show()

# Step 5: Generate heatmap of average request closing time

# Create pivot table: rows = Complaint Type, columns = Borough
heatmap_data = closing_time_avg.pivot(index='Complaint Type', columns='Borough', values='Request_Closing_Time')

# Optional: Focus on top 10 complaints for clarity
top10_complaint_types = df_cleaned['Complaint Type'].value_counts().nlargest(10).index
heatmap_data = heatmap_data.loc[heatmap_data.index.isin(top10_complaint_types)]

# Plot heatmap color
plt.figure(figsize=(12, 8))
sns.heatmap(heatmap_data, annot=True, cmap='coolwarm', fmt='.1f')
plt.title('Average Request Closing Time (Hours) by Complaint Type and Borough')
plt.tight_layout()
plt.savefig('response_time_heatmap.png')
plt.show()
```



Average Request Closing Time for Top 5 Complaint Types by Borough
Borough
BRONX
BROOKLYN
MANHATTAN
QUEENS
STATEN ISLAND
Average Closing Time (Hours)
0
2
4
6
8
Blocked Driveway
Derelict Vehicle
Illegal Parking
Noise - Commercial
Noise - Street/Sidewalk
Complaint Type



Average Request Closing Time (Hours) by Complaint Type and Borough
Complaint Type
Animal Abuse 7.3 4.8 3.7 5.4 5.0
Blocked Driveway 6.3 4.4 3.6 4.5 4.1
Derelict Vehicle 9.2 6.0 4.3 8.5 5.0
Illegal Parking 6.6 4.3 3.4 4.8 3.8
Noise - Commercial 4.7 3.0 2.7 3.8 3.0
Noise - Park 4.7 3.1 3.0 3.8 2.9
Noise - Street/Sidewalk 5.2 3.3 2.7 3.6 3.0
Noise - Vehicle 5.6 3.3 2.6 3.7 3.2
Traffic 4.9 3.1 2.7 4.3 3.5
Vending 6.8 4.5 3.3 4.8 4.3
BRONX
BROOKLYN
MANHATTAN
QUEENS
STATEN ISLAND
Borough
3
4
5
6
7
8
9

The code executes an examination of 311 response durations which emphasizes the leading complaint varieties with specific attention to differences between New York City administrative areas. The first step determines the average `Request_Closing_Time` values according to Boroughs with Complaint Types. The system selects the 5 most common complaint types and executes the data filter procedures. The code establishes grouped bar plots that show closing time averages by complaint type across all boroughs through a color scheme designed for readability.

The next segment in the code creates a heatmap which displays broader assessment capabilities. The code develops a pivot table which shows complaint types as rows against boroughs as columns while the table cells show the average closing times. Only ten most prevalent complaint types are represented in the heatmap for visualization clarity purposes. The precise average values from the heatmap use the `coolwarm` color scheme while being saved as an image. The visualizations enable comparison of speed in complaint response through showing distinctive rates across New York City boroughs.

5. Statistical Testing

Test 1: Whether the average response time across complaint types is similar or not.

- State the Null Hypothesis (H0) and Alternate Hypothesis (H1).
- Perform the statistical test and provide the p-value.
- Interpret the results to accept or reject the Null Hypothesis.

```python
# Test 1: Whether the average response time across complaint types is similar or not

print("Test 1: Analysis of Response Time Across Complaint Types")
print("Null Hypothesis (H0): The average response time is the same across all complaint types")
print("Alternative Hypothesis (H1): The average response time differs significantly across complaint types")

# Selecting top 10 complaint types based on frequency
top_10_types = df_cleaned['Complaint Type'].value_counts().nlargest(10).index
subset_df = df_cleaned[df_cleaned['Complaint Type'].isin(top_10_types)]

# Preparing grouped response time data for ANOVA
anova_data = [grp['Request_Closing_Time'].dropna() for _, grp in subset_df.groupby('Complaint Type')]
complaint_labels = list(subset_df.groupby('Complaint Type').groups.keys())

# Displaying descriptive statistics
print("\nBasic statistics for each complaint type:")
for complaint, data in zip(complaint_labels, anova_data):
    print(f"{complaint}: Mean = {data.mean():.2f} hours, Std = {data.std():.2f}, Count = {len(data)}")

# Running one-way ANOVA
from scipy import stats
f_value, p_val = stats.f_oneway(*anova_data)

print(f"\nANOVA Test Results:")
print(f"F-statistic: {f_value:.4f}")
print(f"p-value: {p_val:.4f}")

# Result interpretation
alpha = 0.05
if p_val < alpha:
    print(f"\nConclusion: Since the p-value ({p_val:.4f}) is less than the significance level ({alpha}), "
          f"we reject the null hypothesis. There is significant evidence that the average response time "
          f"differs across complaint types.")
else:
    print(f"\nConclusion: Since the p-value ({p_val:.4f}) is greater than the significance level ({alpha}), "
          f"we fail to reject the null hypothesis. There is insufficient evidence that the average response time "
          f"differs across complaint types.")

# Post-hoc Tukey HSD test if ANOVA result is significant
if p_val < alpha:
    print("\nSince the ANOVA test is significant, performing post-hoc Tukey HSD test:")

    import numpy as np
    from statsmodels.stats.multicomp import pairwise_tukeyhsd

    # Flatten data for Tukey test
    categories = []
    times = []

    for comp_type, grp in subset_df.groupby('Complaint Type'):
        closing_times = grp['Request_Closing_Time'].dropna()
        categories.extend([comp_type] * len(closing_times))
        times.extend(closing_times)

    # Running Tukey HSD test
    tukey_result = pairwise_tukeyhsd(endog=np.array(times), groups=np.array(categories), alpha=0.05)
    print(tukey_result)
```

```
Test 1: Analysis of Response Time Across Complaint Types
Null Hypothesis (H0): The average response time is the same across all complaint types
Alternative Hypothesis (H1): The average response time differs significantly across complaint types

Basic statistics for each complaint type:
Animal Abuse: Mean = 5.22 hours, Std = 8.63, Count = 7744
Blocked Driveway: Mean = 4.74 hours, Std = 5.57, Count = 76676
Derelict Vehicle: Mean = 7.35 hours, Std = 11.07, Count = 17506
Illegal Parking: Mean = 4.48 hours, Std = 5.96, Count = 74021
Noise - Commercial: Mean = 3.14 hours, Std = 4.07, Count = 35144
Noise - Park: Mean = 3.40 hours, Std = 4.02, Count = 3927
Noise - Street/Sidewalk: Mean = 3.44 hours, Std = 5.45, Count = 47747
Noise - Vehicle: Mean = 3.60 hours, Std = 4.62, Count = 16868
Traffic: Mean = 3.45 hours, Std = 4.75, Count = 4466
Vending: Mean = 4.01 hours, Std = 4.76, Count = 3773

ANOVA Test Results:
F-statistic: 877.9279
p-value: 0.0000

Conclusion: Since the p-value (0.0000) is less than the significance level (0.05), we reject the null hypothesis. There is significant evidence that the
average response time differs across complaint types.

Since the ANOVA test is significant, performing post-hoc Tukey HSD test:
                 Multiple Comparison of Means - Tukey HSD, FWER=0.05
```

Multiple Comparison of Means - Tukey HSD, FWER=0.05

group1	group2	meandiff	p-adj	lower	upper	reject
Animal Abuse	Blocked Driveway	-0.4806	0.0	-0.7067	-0.2545	True
Animal Abuse	Derelict Vehicle	2.1288	0.0	1.8701	2.3876	True
Animal Abuse	Illegal Parking	-0.7351	0.0	-0.9615	-0.5086	True
Animal Abuse	Noise - Commercial	-2.0813	0.0	-2.3193	-1.8433	True
Animal Abuse	Noise - Park	-1.8159	0.0	-2.1873	-1.4444	True
Animal Abuse	Noise - Street/Sidewalk	-1.7763	0.0	-2.0085	-1.544	True
Animal Abuse	Noise - Vehicle	-1.6203	0.0	-1.8806	-1.36	True
Animal Abuse	Traffic	-1.7668	0.0	-2.1231	-1.4105	True
Animal Abuse	Vending	-1.2068	0.0	-1.5832	-0.8303	True
Blocked Driveway	Derelict Vehicle	2.6094	0.0	2.4506	2.7683	True
Blocked Driveway	Illegal Parking	-0.2545	0.0	-0.3522	-0.1568	True
Blocked Driveway	Noise - Commercial	-1.6007	0.0	-1.7228	-1.4785	True
Blocked Driveway	Noise - Park	-1.3353	0.0	-1.6455	-1.025	True
Blocked Driveway	Noise - Street/Sidewalk	-1.2956	0.0	-1.4062	-1.1851	True
Blocked Driveway	Noise - Vehicle	-1.1397	0.0	-1.301	-0.9784	True
Blocked Driveway	Traffic	-1.2862	0.0	-1.5781	-0.9943	True
Blocked Driveway	Vending	-0.7262	0.0	-1.0424	-0.41	True
Derelict Vehicle	Illegal Parking	-2.8639	0.0	-3.0233	-2.7046	True
Derelict Vehicle	Noise - Commercial	-4.2101	0.0	-4.3855	-4.0347	True
Derelict Vehicle	Noise - Park	-3.9447	0.0	-4.2795	-3.6099	True
Derelict Vehicle	Noise - Street/Sidewalk	-3.9051	0.0	-4.0726	-3.7375	True
Derelict Vehicle	Noise - Vehicle	-3.7491	0.0	-3.9537	-3.5446	True
Derelict Vehicle	Traffic	-3.8956	0.0	-4.2135	-3.5778	True
Derelict Vehicle	Vending	-3.3356	0.0	-3.6759	-2.9953	True
Illegal Parking	Noise - Commercial	-1.3462	0.0	-1.469	-1.2234	True
Illegal Parking	Noise - Park	-1.0808	0.0	-1.3913	-0.7703	True
Illegal Parking	Noise - Street/Sidewalk	-1.0412	0.0	-1.1525	-0.9299	True
Illegal Parking	Noise - Vehicle	-0.8852	0.0	-1.047	-0.7235	True
Illegal Parking	Traffic	-1.0317	0.0	-1.3239	-0.7396	True
Illegal Parking	Vending	-0.4717	0.0001	-0.7882	-0.1552	True
Noise - Commercial	Noise - Park	0.2654	0.2025	-0.0536	0.5845	False
Noise - Commercial	Noise - Street/Sidewalk	0.305	0.0	0.1718	0.4383	True
Noise - Commercial	Noise - Vehicle	0.461	0.0	0.2834	0.6386	True
Noise - Commercial	Traffic	0.3145	0.0324	0.0133	0.6157	True
Noise - Commercial	Vending	0.8745	0.0	0.5497	1.1993	True
Noise - Park	Noise - Street/Sidewalk	0.0396	1.0	-0.2752	0.3544	False
Noise - Park	Noise - Vehicle	0.1956	0.7086	-0.1404	0.5315	False
Noise - Park	Traffic	0.049	1.0	-0.3658	0.4638	False
Noise - Park	Vending	0.6091	0.0004	0.1768	1.0413	True
Noise - Street/Sidewalk	Noise - Vehicle	0.1559	0.1041	-0.0139	0.3258	False
Noise - Street/Sidewalk	Traffic	0.0094	1.0	-0.2873	0.3061	False
Noise - Street/Sidewalk	Vending	0.5695	0.0	0.2488	0.8901	True
Noise - Vehicle	Traffic	-0.1465	0.9104	-0.4656	0.1726	False
Noise - Vehicle	Vending	0.4135	0.005	0.0721	0.755	True
Traffic	Vending	0.56	0.001	0.1408	0.9793	True

The code tests whether the average response time across different complaint types is similar. It starts by defining the null hypothesis (H0) that the average response time is the same across all complaint types and the alternative hypothesis (H1) that the average response time differs significantly.

The first step involves selecting the top 10 most frequent complaint types. Then, the data is filtered to only include those complaint types. For each complaint type, it gathers the response time Request_Closing_Time data in preparation for the one-way ANOVA test.

Descriptive statistics (mean, standard deviation, and count) for each complaint type are displayed to provide insight into the data before running the ANOVA test. The test calculates the F-statistic and p-value, which are then used to determine if there is a significant difference in average response times across the complaint types. If the p-value is less than 0.05 (the significance level), the null hypothesis is rejected, indicating that the average response time differs across complaint types.

If the ANOVA test is significant, a post-hoc Tukey HSD (Honest Significant Difference) test is performed to identify which specific complaint types have significantly different average response times. The Tukey HSD test compares all possible pairs of complaint types to determine which pairs differ significantly in terms of response time. The results of the Tukey test are printed to show which complaint types have significantly different response times.

Test 2: Whether the type of complaint or service requested, and location are related.

● State the Null Hypothesis (H0) and Alternate Hypothesis (H1).

● Perform the statistical test and provide the p-value.

● Interpret the results to accept or reject the Null Hypothesis.

```
Test 2: Relationship Between Complaint Type and Location (Borough)
Null Hypothesis (H0): Complaint type and Borough are independent
Alternative Hypothesis (H1): There is a significant association between complaint type and Borough

Contingency Table (Counts of Complaint Types by Borough):
Borough                  BRONX  BROOKLYN  MANHATTAN  QUEENS  STATEN ISLAND
Complaint Type
Animal Abuse              1412      2390       1511    1874            557
Blocked Driveway         12740     28119       2055   31621           2141
Derelict Vehicle          1948      5164        530    8102           1762
Illegal Parking           7829     27386      11981   21944           4881
Noise - Commercial        2431     11451      14528    6057            677
Noise - Park               522      1537       1167     634             67
Noise - Street/Sidewalk   8864     13315      20362    4391            815
Noise - Vehicle           3385      5145       5374    2608            356
Traffic                    355      1082       1531    1302            196
Vending                    377       514       2380     477             25

Chi-Square Test Results:
Chi-square statistic: 63794.7474
p-value: 0.0000e+00
Degrees of freedom: 36

Conclusion: Since the p-value (0.0000e+00) is less than the significance level (0.05), we reject the null hypothesis. There is significant evidence of an
association between complaint type and Borough.

Cramer's V: 0.2353
Interpretation of Cramer's V:
0.0 to 0.1: Very weak association
0.1 to 0.3: Weak association
0.3 to 0.5: Moderate association
0.5 to 0.8: Strong association
0.8 to 1.0: Very strong association
```

This code tests whether there is a significant relationship between the type of complaint and the borough location in which it was reported. The null hypothesis (H0) is that complaint type and borough are independent, while the alternative hypothesis (H1) is that there is a significant association between the two variables.

The procedure starts by selecting the top 10 most common complaint types using value_counts().nlargest(10). It then filters the dataset to include only those complaint types and removes rows with missing borough data using dropna(subset=['Borough']). A contingency table is created using pd.crosstab(), which counts the occurrences of each complaint type across different boroughs.

Next, a chi-square test for independence is performed on the contingency table using stats.chi2_contingency(). The chi-square statistic, p-value, and degrees of freedom are printed. The p-value is compared to a significance threshold (0.05). If the p-value is less than 0.05, the null hypothesis is rejected, indicating a significant association between complaint type and borough. If the p-value is greater than 0.05, the null hypothesis is not rejected, suggesting no significant association.

Additionally, the code calculates Cramér's V, a measure of association strength between two categorical variables. The function compute_cramers_v() calculates Cramér's V using the chi-square statistic, and the resulting value is interpreted using a scale (e.g., 0.0 to 0.1 represents a very weak association, 0.1 to 0.3 represents a weak association, etc.). This provides insight into the strength of the relationship between complaint type and borough.

Finally, the results of both the chi-square test and Cramér's V are printed for interpretation.