311_CUSTOMER_SERVICE

July 15, 2023

[ ]:
```
#importing important libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from datetime import datetime as dt
from scipy import stats
```

[ ]:
```
#1.1 Data importation
df = pd.read_csv("311_Service_Requests_from_2010_to_Present.csv",␣
↪low_memory=False, dtype={48:'str', 49:'str'})

### This dataset specified that columns 48, and 49 have mixtypes to resolve␣
↪this warning,
#I specified the data types explicitly for these columns during
#the import using the dtype parameter  passing low_memoery=False in the␣
↪read_csv() function.
```

[ ]:
```
#1.2 Previewing the head of the dataset
#visualizing the head of the dataset gives us an overview of the overall␣
↪dataset.
#However, the visualization processes requires the use of charts.

df.head()
```

[ ]:
```
# Previewing the tail of the dataset gives us an overview of the overall␣
↪dataset.

df.tail()
```

[ ]:
```
#1.3 DataFrame Columns
#df.columns populates the column names in the dataset, it is relevant for␣
↪accessing,
#manipulating, and working with column names in a pandas DataFrame.

df.columns
```

[ ]:
```
#1.4 Shape of dataset
# df.shape is a valuable attribute, I was able to understand the dimensions of␣
↪a DataFrame.
# It aided in data validation,indexing and manipulation tasks.

df.shape
```

[ ]:
```
#1.5 Identifying variables with null values

# Populating columns with NaN values, I used the .isnull() function on the␣
↪dataframe, boolean mask is generated,
#where True indicates the presence of null values in the cBorresponding cells␣
↪and False a datapoint.
# .sum() function helps in presenting the output as a series.

df.isnull().sum()
```

[ ]:
```
# Computing the missing values percentage wise.
df.isnull().mean() * 100
```

[ ]:
```
#basic data exploratory analysis:
df.describe()
```

[ ]:
```
# 2.1 frequency plot to show the number of null values in each column of the␣
↪DataFrame

columns_with_null_values = df.columns[df.isnull().any()].tolist()

# Calculate frequency of null values in each column using .sum()function on .
↪isnull() and applying it to df[columns_with_null_values]
Null_Frequency = df[columns_with_null_values].isnull().sum()

#plt.figure was used to specify the size of the chart.
plt.figure(figsize=(10, 6))

# .Plot was imployed to plot the NaN in each column.
Null_Frequency.plot(kind='bar', color='g', lw=0.2, alpha=0.5 )

# the x and y cordinates.
plt.title=('columns with null values')
plt.xlabel=('Columns')
plt.ylabel=('Frequency')

#plt.show was used to display the plot of columns_with_null_values
plt.show()
```

Plot showing columns with null values

```
[ ]: #2.2 Missing value treatment,

     ## Set the threshold percentage
     Threshold = 70

     missing_percentage = df.isnull().mean() * 100 #Percentatge calculation of␣
      ↪columns with missing values.

     ## Get the column names where the null percentage is greater than the threshold
     columns_to_drop = missing_percentage[missing_percentage > Threshold].index

     dropped_columns = df.drop(columns=columns_to_drop)
```

```
[ ]: dropped_columns
```

```
[ ]: #2.2.1 Records whose closed date values are null

     # .isnull() is applied to closed date, which outputs each element as True if␣
      ↪the corresponding value in df is null and False otherwise.

     #.dropna() dropped the null values in the closed date column,leaving False␣
      ↪values that are true values.

     closed_date_with_null = df['Closed Date'].isnull().dropna().sum()
```

```
[ ]: closed_date_with_null
```

```
[ ]: # The .dropna() is applied on the df and specifies the subset parameter as␣
      ↪'Closed Date'.

     #The inplace=True parameter is used to modify the DataFrame,rather than␣
      ↪creating a new DataFrame.

     date = df.dropna(subset=['Closed Date'], inplace=True)
```

```
[ ]: #2.3
     #df['Created Date'] selects the 'Created Date' column of the DataFrame df.

     #pd.to_datetime() function is used to convert the created date to␣
      ↪datetime64[ns] format,
     #it assigns the converted datetime values back to the 'Created Date' column,␣
      ↪overwriting the previous values.


     df['Created Date'] = pd.to_datetime(df['Created Date'])
     df['Created Date']
```

[ ]:
```
df['Closed Date'] = pd.to_datetime(df['Closed Date'])
df['Closed Date']
```

[ ]:
```
df=df
```

[ ]:
```
#2.3.1 Elapsed date and time

#(df['Closed Date']) is subtracted from (df['Created Date']) using simple␣
↪arithmetic
#to obtain the time difference in closing each complaint type.

elapsed_time = (df['Closed Date'])-(df['Created Date'])
elapsed_time
```

[ ]:
```
#2.3.2 Convert elapsed time to seconds

#dt.seconds is a property of the dt accessor that returns the elapsed time in␣
↪seconds when applied.

Convert_seconds = pd.DataFrame(elapsed_time.dt.seconds)
Convert_seconds
```

[ ]:
```
#2.3.3 descriptive statistics

#.describe() is a descriptive statistics for the variable Convert_seconds,␣
↪which represents the elapsed time in seconds.
# It provides a summary of various statistical measures.

Convert_seconds.describe()
```

[ ]:
```
# 2.3.4 Null values in complaint type and city columns
#.isnull() applied to the complaint type and city column which returns a␣
↪boolean value as True if there is a missing value,
#while the.sum() It calculates the sum of True values (missing values) along␣
↪each column.

null_values = df[['Complaint Type','City']].isnull().sum()
null_values
```

[ ]:
```
# 2.3.5 NaN values in the city column was replaced with Unkown city using .
↪replace() function placed in a dictionary.

df.replace({'City': np.NaN}, 'Unkown City', inplace=True)
```

[ ]:
```
#This  line of code was executed to confirm the absence of null values.
df['City'].isnull().sum()
```

```
[ ]: City_Complaint_Counts= df.groupby('City')['Complaint Type'].value_counts()
     City_Complaint_Counts
```

```
[ ]: counter = 0
     # 2.3.6 Plotting each city's complaint frequencies

     # df.groupby function was used to group the city column by complaints Type␣
      ↪column,
     #this enabled us to understand specific complaint types associated to each␣
      ↪cities.
     #value_counts() was necessarry to understand the total number of specific␣
      ↪complaint types for each cities.
     City_Complaint_Counts = df.groupby('City')['Complaint Type'].value_counts()


     # I initiated a for loop to iterates through the city column in the df.
     #.unique() is a pandas method applied to the 'City' column that returns an␣
      ↪array containing only
     #the unique values from that Series. It eliminates any duplicate values.
     for city in df['City'].unique():

     #['City'].plot In this case, it is used to create a bar plot for the selected␣
      ↪series.
     #kind='bar' specifies the type of plot to create, which is a bar plot in this␣
      ↪case.
     #alpha=0.7 sets the transparency level of the bars to 0.7, making them slightly␣
      ↪transparent.
         City_Complaint_Counts[city].plot(kind='bar', color='green', alpha=0.7,␣
      ↪edgecolor='white', linewidth=1.2)


     #The lines plt.xlabel=('Complaint Type') and plt.ylabel=('Frequency') are used␣
      ↪to set the labels for
     #the x-axis and y-axis, respectively, in a matplotlib plot.
         plt.xlabel=('Complaint Type')
         plt.ylabel=('Frequency')


     # Displays the graphical representation of the dataset
         plt.show()

         print("City:", city)  # Print the city name

         counter += 1 # The counter helps us to populate the desired number of␣
      ↪cities.
         if counter >= 2:
```

```
        break
```

Plot showing specific city's complaint frequency

[ ]:
```
complaint_type = df['Complaint Type']
complaint_type
```

[ ]:
```
#2.3.7 SCATTER PLOT OF THE CONCENTRATION OF COMPLAINT ACROSS BROOKLYN

#creates a new DataFrame called brooklyn by filtering the original DataFrame df␣
↪based on a condition df['City'] == 'BROOKLYN'
# and returning a boolean value True for Brooklyn in the city column.
brooklyn = df[df['City'] == 'BROOKLYN']

#plt.figure was used to determine the size of the graph width wise and height␣
↪alike.
plt.figure(figsize=(10, 15))

# for loop iterates over each unique value in the brooklyn dataframe Complaint␣
↪Type column
#using the .unique()function.
for complaint_type in brooklyn['Complaint Type'].unique():

# applies the boolean mask to the DataFrame brooklyn, returning a new DataFrame␣
↪that contains only
#the rows where the condition is True.
    data = brooklyn[brooklyn['Complaint Type'] == complaint_type]

#This line plots a scatter plot using the 'Longitude' and 'Latitude' columns of␣
↪the DataFrame data.
    plt.scatter(data['Longitude'], data['Latitude'], s=10, alpha=0.5,␣
↪label=str(complaint_type))

#Title of the Scatter plot
plt.title=('Scatter Plot of Complaint Concentration in Brooklyn')
#Label of the x axis
plt.xlabel=('Longitude')
#Label of the y axis
plt.ylabel=('Latitude')
# provides additional information about the mapping of colors to data values in␣
↪the plot,
#for easy understanding of the distribution of complaint type across Brooklyn.
plt.legend()



# Displays the graphical representation of the dataset
plt.show()
```

Each point in the plot represents an individual complaint, with its location determined by the latitude and longitude coordinates.

```
[ ]: #2.3.7 HEXBIN PLOT OF THE CONCENTRATION OF COMPLAINT ACROSS BROOKLYN

     ##plt.figure was used to determine the size of the graph width wise and height␣
      ↪alike.
     plt.figure(figsize=(10, 6))

     #This line plots a Hexbin plot using the 'Longitude' and 'Latitude' columns of␣
      ↪the DataFrame data.
     plt.hexbin(data['Longitude'], data['Latitude'], gridsize=20, cmap='viridis')



     plt.title=('Hexbin plot of Complaint Concentration in Brooklyn')
     #Label of the x axis
     plt.xlabel=('Longitude')
     #Label of the y axis
     plt.ylabel=('Latitude')
     #adds a colorbar to a plot created using Matplotlib. The colorbar is a␣
      ↪rectangular color scale that provides additional
     #information about the mapping of colors to data values in the plot.
     plt.colorbar(label='Count')

     # Displays the graphical representation of the dataset
     plt.show()
```

The hexbin plot visually represents the spatial distribution and concentration of complaints across Brooklyn.

```
[ ]: #3 major types of complaints
     #calculates the frequency of each unique value in the 'Complaint Type' column␣
      ↪of the DataFrame df and stores the result
     #in the variable major_types_of_complaints.
     major_types_of_complaints = df['Complaint Type'].value_counts()
     major_types_of_complaints
```

```
[ ]: #3.1 A BAR CHART PLOT SHOWING COMPLAINTS TYPES IN DECENDING ORDER

     # After getting the unique value in the 'Complaint Type' column
     major_types_of_complaints = df['Complaint Type'].value_counts()

     #plt.figure was used to determine the size of the plot.
     plt.figure(figsize=(11, 7))
```

```
#plt.bar creates a bar plot using the values and index from the␣
↪major_types_of_complaints Series,
plt.bar(major_types_of_complaints.index, major_types_of_complaints.values, lw=1.
↪2, color='pink', label='Count')

plt.title=('major types of complaints')

plt.xlabel=('Complaint Type')
plt.ylabel=('Count')

#to specify the rotation angle for the tick labels. In this case, the angle is␣
↪set to 90 degrees,
#causing the tick labels to rotate vertically.
plt.xticks(rotation=90)


# Displays the graphical representation of the dataset.
plt.show()
```

This graph clearly displays the frequency of the major complaint types in our DataFram

[ ]:
```
#3.2 FREQUENCY OF VARIOUS TYPES OF COMPLAINTS FOR NEW YORK

#This line filters the DataFrame df based on a condition. It creates a new␣
↪DataFrame called
#Frequency_NewYork_Complaints that includes only the rows where the 'City'␣
↪column has the value 'NEW YORK'.
Frequency_NewYork_Complaints = df[df['City'] == 'NEW YORK']


#This line calculates the frequency of each unique combination of 'City' and␣
↪'Complaint Type' in the DataFrame df using .value_counts
#The groupby() method is used to group the data by the 'City' column, and then␣
↪value_counts() is applied to the
#'Complaint Type' column within each group..
city_complaint_counts = df.groupby('City')['Complaint Type'].value_counts()

#This line assigns the string value 'NEW YORK' to the variable specific_city.
#indicating the specific city for which I want to retrieve the complaint counts.
specific_city = 'NEW YORK'


#This line accesses the specific complaint counts for NEW YORK from the␣
↪city_complaint_counts Series.
city_complaint_counts[specific_city]
```

[ ]:
```
#3.3 TOP TEN COMPLAINT TYPES

# This line extracts the 'Complaint Type' column from the DataFrame df and␣
↪assigns it to the variable complaints.

complaints = df['Complaint Type']

#.value_counts calculates the frequency of each unique value in the complaints␣
↪Series.
#.head() is chained to the resulting Series to select the top 10 most frequent␣
↪complaint types
top_ten = complaints.value_counts().head(10)

#This prints out the top ten complaint types from the df.
print(top_ten)
```

[ ]:
```
#3.4 display various types of complaints for each city

# he groupby() method is called on df with 'City' as the grouping column.
#.unique() function is applied to the 'Complaint Type' column within each city␣
↪subgroups  based on the distinct values in the 'City'.
#This allows us to obtain the unique complaint types for each city
grouped = df.groupby('City')['Complaint Type'].unique()

#This line sorts the pandas Series grouped by its index (city names) in␣
↪ascending order.
grouped = grouped.sort_index()

#Initializes the counter
counter = 0

#This line initiates a loop that iterates over the items of the grouped object.
for city, complaint_types in grouped.items():

#This line prints the name of the current city using an f-string.
    print(f"City: {city}")

#This line initiates another loop that iterates over each element in the␣
↪complaint_types array.
    for complaint_type in complaint_types:

#This line prints the current complaint type using an f-string
        print(f"- {complaint_type}")


        counter += 1 # Increment the counter after processing each citys.
```

```
    if counter >= 2:
        break
```

[ ]:
```
#3.5 DataFrame, df_new, which contains cities as columns and complaint types in␣
↪rows

#df['City'].value_counts() calculates the frequency count of each unique value␣
↪in the 'City' column of the df.
#.head(10) selects the top 10 most frequent values from the resulting counts.
#.index retrieves the index (i.e., the unique city names) from the resulting␣
↪Series.
#.to_list() converts the index to a list.
cities=df['City'].value_counts().head(10).index.tolist()

#isin()checks for each value in the 'City' column of df if it is present in the␣
↪cities list.
pop_cities= df[df.City.isin(cities)]

#pd.crosstab() is a function from the pandas library that computes a␣
↪cross-tabulation
#between two or more variables.
df_new=pd.crosstab(pop_cities['City'],pop_cities['Complaint Type'])

df_new
```

[ ]:
```
# Visualize the major types of complaints

#This line groups the DataFrame df by both the 'City' and 'Complaint Type'␣
↪columns.
grouped = df.groupby(['City', 'Complaint Type'])

#.size() is applied to the grouped object, returning a Series that contains the␣
↪count
#of occurrences for each combination of 'City' and 'Complaint Type'.
complaint_counts = grouped.size()

#reset_index() removes the multi-level index and assigns a new integer index to␣
↪the DataFrame.
complaint_counts = complaint_counts.reset_index()

#This line sorts the complaint_counts DataFrame by the 'City' column in␣
↪ascending order
#and the count values (0) in descending order.
complaint_counts = complaint_counts.sort_values(['City', 0], ascending=[True,␣
↪False])
```

```
#his line groups the complaint_counts DataFrame by the 'City' column.
major_complaints = complaint_counts.groupby('City').first()

#.reset_index() removes the existing index and assigns a new integer index to␣
↪the DataFrame.
major_complaints = major_complaints.reset_index()
```

[ ]:
```
#4.1 CHART SHOWING TYPES OF COMPLAINTS IN EACH CITY

import seaborn as sns

# Group the DataFrame by 'City' and 'Complaint Type' and count the occurrences
grouped = df.groupby(['City', 'Complaint Type']).size().unstack(fill_value=0)

# Get the list of cities and complaint types
cities = grouped.index
complaint_types = grouped.columns

# Set the color palette for different complaint types
colors = sns.color_palette('Set3', len(complaint_types))

# Plot the chart
plt.figure(figsize=(15, 10))
bottom = None  # Variable to keep track of the bottom values for stacked bars

# Iterate over each complaint type and plot the stacked bars for each city
for i, complaint_type in enumerate(complaint_types):
    values = grouped[complaint_type]
    plt.bar(cities, values, bottom=bottom, color=colors[i],␣
↪label=complaint_type)


#This conditional statement updates the bottom variable. If bottom is None, it␣
↪assigns it the current values.
#Otherwise, it adds the current values to the existing bottom values.
    if bottom is None:
        bottom = values
    else:
        bottom += values


# Set the chart title, labels, and legend
plt.title=('Number of Complaint Types per City')
plt.xlabel=('City')
plt.ylabel=('Count')
plt.xticks(rotation=90)
plt.legend(loc='upper right', title='Complaint Type')
```

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

From this graph we can infer that the cities with most frequent complaints are, Brooklyn, New York, and Bronx.

[ ]:
```
#4.2 Sorting Complaint type based on average request closing time

# Calculate the average request closing time for each complaint type grouped by␣
↪location
avg_request_closing_time = (df.groupby(['City', 'Complaint Type'])['Closed␣
↪Date']
                            .apply(lambda x: (x - x.min()).mean()))

# Sort the complaint types based on the average request closing time in␣
↪ascending order
sorted_complaints = avg_request_closing_time.sort_values()

# Print the sorted complaints
Average_ClosingTime = pd.DataFrame(sorted_complaints)
Average_ClosingTime
```

[ ]:
```
#5 Average response time across different complaint types

# Calculate the overall average response time
overall_avg_response_time = (df['Closed Date'] - df['Created Date']).mean()

# Calculate the average response time for each complaint type
avg_response_time_per_complaint = (df.groupby('Complaint Type').apply(lambda x:␣
↪(x['Closed Date'] - x['Created Date']).mean())
                                   .dt.total_seconds())

# Print the average response time for each complaint type
print(avg_response_time_per_complaint)
```

The average response time across the complaint types are similar, however, there are exceptions.

[ ]:
```
# 5.1 Visualise the average of Request Closing Time

# Calculate the average request closing time per complaint type
avg_request_closing_time = (df.groupby('Complaint Type').apply(lambda x:␣
↪(x['Closed Date'] - x['Created Date']).mean())
                            .dt.total_seconds())

# Sort the average closing time in descending order
avg_request_closing_time = avg_request_closing_time.sort_values(ascending=False)
```

```
# Plotting the average closing time
plt.figure(figsize=(12, 6))
plt.bar(avg_request_closing_time.index, avg_request_closing_time)
plt.xlabel=('Complaint Type')
plt.ylabel=('Average Request Closing Time')

plt.xticks(rotation=90)
plt.show()
```

Bar plot for average response time per complaint type

[ ]:
```
# Sort the average response time per complaint type in ascending order
avg_response_time_per_complaint = avg_response_time_per_complaint.sort_values()

# Perform statistical analysis and calculate p-values
p_values = {}
for complaint_type, response_time in avg_response_time_per_complaint.items():
    sample = (df[df['Complaint Type'] == complaint_type]['Closed Date'] -␣
 ↪df[df['Complaint Type'] == complaint_type]['Created Date']).dt.
 ↪total_seconds()
    _, p_value = stats.ttest_1samp(sample, overall_avg_response_time.
 ↪total_seconds())
    p_values[complaint_type] = p_value

# Print only significant p-values
for complaint_type, p_value in p_values.items():
    if p_value is not None and p_value < 0.05:
        print(f"Complaint Type: {complaint_type}, p-value: {p_value}")
```

We can infer that variables with p-values <=0.05 indicates statistical significance

[ ]:
```
#6 perform Kruskal-Wallis H test
from scipy import stats

# Calculate the response time for each complaint type
response_times = {}
for complaint_type, group in df.groupby('Complaint Type'):
    response_times[complaint_type] = (group['Closed Date'] - group['Created␣
 ↪Date']).dt.total_seconds()

# Perform Kruskal-Wallis H test
statistic, p_value = stats.kruskal(*response_times.values())

# Print the result
print("Kruskal-Wallis H Test:")
print(f"Statistic: {statistic}")
```

```
print(f"P-value: {p_value}")
```

[ ]:
```
#7.1 fail to reject H0 : all sample distributions are equal

#In this case, the low p-value indicates that the observed differences in the␣
↪response times among the complaint types are unlikely to have occurred by␣
↪chance alone.
#The statistic value (11988.269402358468) represents the test statistic␣
↪computed from the data.
```

[ ]:
```
#7.2 Reject H0:one or more sample distribution are not equal

#In this case, the low p-value (close to zero) indicates that the observed␣
↪differences in
#the response times among the complaint types are highly unlikely to have␣
↪occurred by chance alone.
#The statistic value (11988.269402358468) represents the test statistic␣
↪computed from the data.
```