Midterm Project

Ammar Alsadadi

Table of contents

1 Introduction

1.1 Overview

The NYC 311 Service Requests dataset contains reports related to various city issues, including sewer-related complaints. This analysis focuses on two primary sewer-related complaints for 2024:

1. **Street Flooding (SF)**: Reports of standing water or street flooding.
2. **Catch Basin (CB)**: Reports of clogged or defective catch basins that prevent proper drainage.

These complaints provide insights into both immediate street flooding issues and underlying infrastructural challenges.

1.2 Data Dictionary

Column Name	Description	Data Type
Unique Key	Unique identifier for each service request	Text
Created Date	Date and time the service request was created	Floating Timestamp
Closed Date	Date and time the service request was closed	Floating Timestamp
Agency	City agency responsible for the complaint	Text
Agency Name	Full name of the responding agency	Text
Complaint Type	High-level category of the complaint	Text
Descriptor	Specific details about the complaint	Text
Location Type	Type of location information available	Text
Incident Zip	ZIP code where the complaint was filed	Text
Incident Address	Address where the complaint was reported	Text
Street Name	Street name of incident location	Text
Cross Street 1	First cross street based on geo-validation	Text
Cross Street 2	Second cross street based on geo-validation	Text
Intersection Street 1	First intersecting street based on geo-validation	Text
Intersection Street 2	Second intersecting street based on geo-validation	Text

Column Name	Description	Data Type
Address Type	Type of incident location information	Text
City	City of the incident location	Text
Landmark	Name of landmark if applicable	Text
Facility Type	Type of city facility associated with SR	Text
Status	Current status of the complaint	Text
Due Date	Expected update date by responding agency	Floating Timestamp
Resolution Description	Details of the resolution or current status	Text
Resolution Action Updated Date	Last update of resolution actions	Floating Timestamp
Community Board	Community board information from geo-validation	Text
BBL	Borough Block and Lot identifier	Text
Borough	Borough of the incident	Text
X Coordinate (State Plane)	Geo-validated X coordinate of the incident location	Number
Y Coordinate (State Plane)	Geo-validated Y coordinate of the incident location	Number
Open Data Channel Type	Submission method (Phone, Online, Mobile, etc.)	Text
Park Facility Name	Name of park facility if applicable	Text
Park Borough	Borough of the park facility	Text
Vehicle Type	Type of TLC vehicle if applicable	Text
Taxi Company Borough	Borough of the taxi company if applicable	Text
Taxi Pick Up Location	Taxi pick-up location if applicable	Text
Bridge Highway Name	Name of bridge/highway if applicable	Text
Bridge Highway Direction	Direction of bridge/highway issue	Text
Road Ramp	Specifies if the issue occurred on a road or ramp	Text
Bridge Highway Segment	Segment of the bridge/highway where issue occurred	Text
Latitude	Latitude coordinate of the incident location	Number
Longitude	Longitude coordinate of the incident location	Number
Location	Combination of latitude and longitude	Location

1.3 Exploring Data

```
import pandas as pd

# Load the dataset
file_path = "data/nycflood2024.csv"
```

```
df = pd.read_csv(file_path)

# Display basic info about the dataset
df_info = df.info()

# Show first few rows
df_head = df.head(3)

df_info, df_head
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9483 entries, 0 to 9482
Data columns (total 41 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   Unique Key                      9483 non-null   int64
 1   Created Date                    9483 non-null   object
 2   Closed Date                     9275 non-null   object
 3   Agency                          9483 non-null   object
 4   Agency Name                     9483 non-null   object
 5   Complaint Type                  9483 non-null   object
 6   Descriptor                      9483 non-null   object
 7   Location Type                   0 non-null      float64
 8   Incident Zip                    9479 non-null   float64
 9   Incident Address                7245 non-null   object
 10  Street Name                     7245 non-null   object
 11  Cross Street 1                  7237 non-null   object
 12  Cross Street 2                  7234 non-null   object
 13  Intersection Street 1           2239 non-null   object
 14  Intersection Street 2           2239 non-null   object
 15  Address Type                    9483 non-null   object
 16  City                            9479 non-null   object
 17  Landmark                        0 non-null      float64
 18  Facility Type                   0 non-null      float64
 19  Status                          9483 non-null   object
 20  Due Date                        0 non-null      float64
 21  Resolution Description          9311 non-null   object
 22  Resolution Action Updated Date  9314 non-null   object
 23  Community Board                 9483 non-null   object
 24  BBL                             6848 non-null   float64
 25  Borough                         9483 non-null   object
 26  X Coordinate (State Plane)      9389 non-null   float64
 27  Y Coordinate (State Plane)      9389 non-null   float64
 28  Open Data Channel Type          9483 non-null   object
 29  Park Facility Name              9483 non-null   object
```

```
 30  Park Borough                    9483 non-null   object
 31  Vehicle Type                    0 non-null      float64
 32  Taxi Company Borough            0 non-null      float64
 33  Taxi Pick Up Location           0 non-null      float64
 34  Bridge Highway Name             0 non-null      float64
 35  Bridge Highway Direction        0 non-null      float64
 36  Road Ramp                       0 non-null      float64
 37  Bridge Highway Segment          0 non-null      float64
 38  Latitude                        9389 non-null   float64
 39  Longitude                       9389 non-null   float64
 40  Location                        9389 non-null   object
dtypes: float64(17), int64(1), object(23)
memory usage: 3.0+ MB
```

```
(None,
    Unique Key            Created Date             Closed Date Agency  \
 0    63574884  12/31/2024 11:05:00 PM  01/01/2025 05:55:00 AM    DEP
 1    63580185  12/31/2024 11:02:00 PM  01/01/2025 10:30:00 AM    DEP
 2    63573084  12/31/2024 11:02:00 PM  01/01/2025 06:20:00 AM    DEP

                              Agency Name Complaint Type  \
 0  Department of Environmental Protection          Sewer
 1  Department of Environmental Protection          Sewer
 2  Department of Environmental Protection          Sewer

             Descriptor  Location Type  Incident Zip  \
 0  Street Flooding (SJ)            NaN       11434.0
 1  Street Flooding (SJ)            NaN       11219.0
 2  Street Flooding (SJ)            NaN       11361.0

                Incident Address  ... Vehicle Type Taxi Company Borough  \
 0              177-37 135 AVENUE  ...          NaN                   NaN
 1                 4102 14 AVENUE  ...          NaN                   NaN
 2  35-34 CORPORAL KENNEDY STREET  ...          NaN                   NaN

   Taxi Pick Up Location Bridge Highway Name Bridge Highway Direction  \
 0                   NaN                 NaN                       NaN
 1                   NaN                 NaN                       NaN
 2                   NaN                 NaN                       NaN

   Road Ramp Bridge Highway Segment   Latitude  Longitude  \
 0       NaN                   NaN  40.675681 -73.762969
 1       NaN                   NaN  40.638098 -73.985346
 2       NaN                   NaN  40.767455 -73.778079
```

```
                                    Location
0   (40.67568056757403, -73.76296916912234)
1   (40.63809797146557, -73.98534616586583)
2  (40.767454543462755, -73.77807940403599)

[3 rows x 41 columns])
```

The dataset contains Street Flooding (SF) and Catch Basin (CB) complaints from 2024, extracted from 9,483 service requests. Key columns relevant to the analysis include that:

- **Descriptor**: Specifies whether the complaint is about Street Flooding or Catch Basin Clogged/Flooding.
- **Created Date**: Indicates when the complaint was reported.
- **Borough**: Provides the location of the complaint within NYC.
- **Incident Zip**: Provides the ZIP code of the complaint.
- **Latitude/Longitude**: Contains geographic coordinates for spatial analysis and mapping.

2 Answering The Project Questions

2.1 Part A (Data Cleaning)

2.1.1 Part i

Import the data, rename the columns with our preferred styles.

```
import pandas as pd

# Load the dataset
file_path = "data/nycflood2024.csv"

df = pd.read_csv(file_path)
# Convert column names to lowercase and replace spaces with underscores
df.columns = df.columns.str.lower().str.replace(" ", "_")

# Display cleaned column names
df.columns
```

```
Index(['unique_key', 'created_date', 'closed_date', 'agency', 'agency_name',
       'complaint_type', 'descriptor', 'location_type', 'incident_zip',
       'incident_address', 'street_name', 'cross_street_1', 'cross_street_2',
```

```
       'intersection_street_1', 'intersection_street_2', 'address_type',
       'city', 'landmark', 'facility_type', 'status', 'due_date',
       'resolution_description', 'resolution_action_updated_date',
       'community_board', 'bbl', 'borough', 'x_coordinate_(state_plane)',
       'y_coordinate_(state_plane)', 'open_data_channel_type',
       'park_facility_name', 'park_borough', 'vehicle_type',
       'taxi_company_borough', 'taxi_pick_up_location', 'bridge_highway_name',
       'bridge_highway_direction', 'road_ramp', 'bridge_highway_segment',
       'latitude', 'longitude', 'location'],
      dtype='object')
```

The column names appear to have been renamed and cleaned properly. ### Part ii Summarize the missing information. Are there variables that are close to completely missing?

```
# Check for missing values
missing_values = df.isnull().sum()

# Calculate the percentage of missing values per column
missing_summary = (df.isnull().sum() / len(df)) * 100

# Identify columns that are completely missing
highly_missing_columns = missing_summary[missing_summary == 100]

# Display summary of missing data
print(missing_values,missing_summary, highly_missing_columns)
```

```
unique_key                      0
created_date                    0
closed_date                   208
agency                          0
agency_name                     0
complaint_type                  0
descriptor                      0
location_type                9483
incident_zip                    4
incident_address             2238
street_name                  2238
cross_street_1               2246
cross_street_2               2249
intersection_street_1        7244
intersection_street_2        7244
address_type                    0
city                            4
landmark                     9483
facility_type                9483
```

```
status                            0
due_date                       9483
resolution_description          172
resolution_action_updated_date  169
community_board                   0
bbl                            2635
borough                           0
x_coordinate_(state_plane)       94
y_coordinate_(state_plane)       94
open_data_channel_type            0
park_facility_name                0
park_borough                      0
vehicle_type                   9483
taxi_company_borough           9483
taxi_pick_up_location          9483
bridge_highway_name            9483
bridge_highway_direction       9483
road_ramp                      9483
bridge_highway_segment         9483
latitude                         94
longitude                        94
location                         94
dtype: int64 unique_key                    0.000000
created_date                       0.000000
closed_date                        2.193399
agency                             0.000000
agency_name                        0.000000
complaint_type                     0.000000
descriptor                         0.000000
location_type                    100.000000
incident_zip                       0.042181
incident_address                  23.600127
street_name                       23.600127
cross_street_1                    23.684488
cross_street_2                    23.716124
intersection_street_1             76.389328
intersection_street_2             76.389328
address_type                       0.000000
city                               0.042181
landmark                         100.000000
facility_type                    100.000000
status                             0.000000
due_date                         100.000000
resolution_description             1.813772
resolution_action_updated_date     1.782136
```

```
community_board                    0.000000
bbl                               27.786565
borough                            0.000000
x_coordinate_(state_plane)         0.991247
y_coordinate_(state_plane)         0.991247
open_data_channel_type             0.000000
park_facility_name                 0.000000
park_borough                       0.000000
vehicle_type                     100.000000
taxi_company_borough             100.000000
taxi_pick_up_location            100.000000
bridge_highway_name              100.000000
bridge_highway_direction         100.000000
road_ramp                        100.000000
bridge_highway_segment           100.000000
latitude                           0.991247
longitude                          0.991247
location                           0.991247
dtype: float64 location_type              100.0
landmark                    100.0
facility_type               100.0
due_date                    100.0
vehicle_type                100.0
taxi_company_borough        100.0
taxi_pick_up_location       100.0
bridge_highway_name         100.0
bridge_highway_direction    100.0
road_ramp                   100.0
bridge_highway_segment      100.0
dtype: float64
```

Several variables are completely missing (100%), making them irrelevant for analysis. These include:

- **location_type**
- **landmark**
- **facility_type**
- **due_date**
- **vehicle_type**
- **taxi_company_borough**
- **taxi_pick_up_location**
- **bridge_highway_name**
- **bridge_highway_direction**
- **road_ramp**
- **bridge_highway_segment**

Other Key Variables with Significant Missing Data:

- **incident_address, street_name, cross_street_1, cross_street_2** (23.6% missing) This could impact location-based analysis.
- **intersection_street_1, intersection_street_2** (76.4% missing) These are mostly empty.
- **bbl** (27.8% missing) Used for property identification.
- **closed_date** (2.2% missing) Some complaints are still open.
- **latitude, longitude** (1% missing) A small fraction lacks geographic data.

2.1.2 Part iii

The following columns appear redundant:

- **Agency Name** (redundant with "Agency")
- **Park Borough** (redundant with "Borough")
- **Location** (duplicates Latitude and Longitude)
- **State Plane Coordinates (X, Y)** (redundant with Latitude/Longitude)

```
import pyarrow.feather as feather
import os
# Store as Arrow (Feather) format for efficiency
arrow_file_path = "data/nycflood2024.feather"
feather.write_feather(df, arrow_file_path)

# Compare file sizes
csv_size = os.path.getsize(file_path) / (1024 * 1024)  # Convert to MB
arrow_size = os.path.getsize(arrow_file_path) / (1024 * 1024)  # Convert to MB

csv_size, arrow_size, (1 - (arrow_size / csv_size)) * 100
```

```
(5.8232269287109375, 1.8518695831298828, 68.19856746438313)
```

The dataset has been converted to Feather format and is stored in the **data** directory. The results are:

- **Original CSV size:** 5.82 MB
- **Arrow (Feather) size:** 1.85 MB
- **Efficiency gain:** 68.2% reduction in storage size

The Feather format is significantly more storage-efficient than CSV, reducing the file size by nearly 70%. This results in:

- Faster load times
- Less disk space usage
- More efficient data processing

The reduction is mainly due to better compression and efficient data types. CSV stores data as raw text, while Arrow uses a binary columnar format, resulting in improved compression. Additionally, Arrow optimizes how numerical and categorical data are stored, further enhancing storage efficiency.

2.1.3 Part iv

Are there invalid NYC zipcode or borough? Can some of the missing values be filled? Fill them if yes.

```
# Create a dictionary mapping ZIP codes to boroughs
valid_zip_to_borough = {
    10001: "Manhattan", 10002: "Manhattan", 10003: "Manhattan", 10004: "Manhattan",
    10005: "Manhattan", 10006: "Manhattan", 10007: "Manhattan", 10009: "Manhattan",
    10010: "Manhattan", 10011: "Manhattan", 10012: "Manhattan", 10013: "Manhattan",
    10014: "Manhattan", 10015: "Manhattan", 10016: "Manhattan", 10017: "Manhattan",
    10018: "Manhattan", 10019: "Manhattan", 10020: "Manhattan", 10021: "Manhattan",
    10022: "Manhattan", 10023: "Manhattan", 10024: "Manhattan", 10025: "Manhattan",
    10026: "Manhattan", 10027: "Manhattan", 10028: "Manhattan", 10029: "Manhattan",
    10030: "Manhattan", 10031: "Manhattan", 10032: "Manhattan", 10033: "Manhattan",
    10034: "Manhattan", 10035: "Manhattan", 10036: "Manhattan", 10037: "Manhattan",
    10038: "Manhattan", 10039: "Manhattan", 10040: "Manhattan", 10041: "Manhattan",
    10044: "Manhattan", 10045: "Manhattan", 10048: "Manhattan", 10055: "Manhattan",
    10060: "Manhattan", 10069: "Manhattan", 10090: "Manhattan", 10095: "Manhattan",
    10098: "Manhattan", 10099: "Manhattan", 10103: "Manhattan", 10104: "Manhattan",
    10105: "Manhattan", 10106: "Manhattan", 10107: "Manhattan", 10110: "Manhattan",
    10111: "Manhattan", 10112: "Manhattan", 10115: "Manhattan", 10118: "Manhattan",
    10119: "Manhattan", 10120: "Manhattan", 10121: "Manhattan", 10122: "Manhattan",
    10123: "Manhattan", 10128: "Manhattan", 10151: "Manhattan", 10152: "Manhattan",
    10153: "Manhattan", 10154: "Manhattan", 10155: "Manhattan", 10158: "Manhattan",
    10161: "Manhattan", 10162: "Manhattan", 10165: "Manhattan", 10166: "Manhattan",
    10167: "Manhattan", 10168: "Manhattan", 10169: "Manhattan", 10170: "Manhattan",
    10171: "Manhattan", 10172: "Manhattan", 10173: "Manhattan", 10174: "Manhattan",
    10175: "Manhattan", 10176: "Manhattan", 10177: "Manhattan", 10178: "Manhattan",
    10199: "Manhattan", 10270: "Manhattan", 10271: "Manhattan", 10278: "Manhattan",
    10279: "Manhattan", 10280: "Manhattan", 10281: "Manhattan", 10282: "Manhattan",
```

```
    10301: "Staten Island", 10302: "Staten Island", 10303: "Staten Island",
    10304: "Staten Island", 10305: "Staten Island", 10306: "Staten Island",
    10307: "Staten Island", 10308: "Staten Island", 10309: "Staten Island",
    10310: "Staten Island", 10311: "Staten Island", 10312: "Staten Island",
    10314: "Staten Island", 10451: "Bronx", 10452: "Bronx", 10453: "Bronx",
    10454: "Bronx", 10455: "Bronx", 10456: "Bronx", 10457: "Bronx", 10458: "Bronx",
    10459: "Bronx", 10460: "Bronx", 10461: "Bronx", 10462: "Bronx", 10463: "Bronx",
    10464: "Bronx", 10465: "Bronx", 10466: "Bronx", 10467: "Bronx", 10468: "Bronx",
    10469: "Bronx", 10470: "Bronx", 10471: "Bronx", 10472: "Bronx", 10473: "Bronx",
    10474: "Bronx", 10475: "Bronx", 11004: "Queens", 11101: "Queens",
    11102: "Queens", 11103: "Queens", 11104: "Queens", 11105: "Queens",
    11106: "Queens", 11109: "Queens", 11201: "Brooklyn", 11203: "Brooklyn",
    11204: "Brooklyn", 11205: "Brooklyn", 11206: "Brooklyn", 11207: "Brooklyn",
    11208: "Brooklyn", 11209: "Brooklyn", 11210: "Brooklyn", 11211: "Brooklyn",
    11212: "Brooklyn", 11213: "Brooklyn", 11214: "Brooklyn", 11215: "Brooklyn",
    11216: "Brooklyn", 11217: "Brooklyn", 11218: "Brooklyn", 11219: "Brooklyn",
    11220: "Brooklyn", 11221: "Brooklyn", 11222: "Brooklyn", 11223: "Brooklyn",
    11224: "Brooklyn", 11225: "Brooklyn", 11226: "Brooklyn", 11228: "Brooklyn",
    11229: "Brooklyn", 11230: "Brooklyn", 11231: "Brooklyn", 11232: "Brooklyn",
    11233: "Brooklyn", 11234: "Brooklyn", 11235: "Brooklyn", 11236: "Brooklyn",
    11237: "Brooklyn", 11238: "Brooklyn", 11239: "Brooklyn", 11249: "Brooklyn",
    11351: "Queens", 11354: "Queens", 11355: "Queens", 11356: "Queens",
    11357: "Queens", 11358: "Queens", 11359: "Queens", 11360: "Queens",
    11361: "Queens", 11362: "Queens", 11363: "Queens", 11364: "Queens",
    11365: "Queens", 11366: "Queens", 11367: "Queens", 11368: "Queens",
    11369: "Queens", 11370: "Queens", 11371: "Queens", 11372: "Queens",
    11373: "Queens", 11374: "Queens", 11375: "Queens", 11377: "Queens",
    11378: "Queens", 11379: "Queens", 11385: "Queens", 11411: "Queens",
    11412: "Queens", 11413: "Queens", 11414: "Queens", 11415: "Queens",
    11416: "Queens", 11417: "Queens", 11418: "Queens", 11419: "Queens",
    11420: "Queens", 11421: "Queens", 11422: "Queens", 11423: "Queens",
    11426: "Queens", 11427: "Queens", 11428: "Queens", 11429: "Queens",
    11430: "Queens", 11432: "Queens", 11433: "Queens", 11434: "Queens",
    11435: "Queens", 11436: "Queens", 11691: "Queens", 11692: "Queens",
    11693: "Queens", 11694: "Queens", 11697: "Queens"
}

# Check invalid ZIP codes
df["incident_zip"] = df["incident_zip"].astype(float).astype("Int64")
invalid_zipcodes = df[~df["incident_zip"].isin(valid_zip_to_borough.keys())]["incident_zip"].uniq

# Valid boroughs
valid_boroughs = {"MANHATTAN", "BRONX", "BROOKLYN", "QUEENS", "STATEN ISLAND"}

# Identify invalid boroughs
```

```
invalid_boroughs = df[~df["borough"].str.upper().isin(valid_boroughs)]["borough"].unique()

invalid_zipcodes,invalid_boroughs
```

```
(<IntegerArray>
 [<NA>, 11040, 10065, 10075, 11001]
 Length: 5, dtype: Int64,
 array(['Unspecified'], dtype=object))
```

The list of valid NYC ZIP codes was sourced from NYC by Natives to check for validity.

The dataset includes five ZIP codes that are not present in the provided list of valid NYC ZIP codes:

- **NaN (Missing ZIP codes):** Some records lack a ZIP code.
- **11040, 11001:** These belong to Nassau County, Long Island, NY, and are not part of NYC. However, since Nassau County borders Queens, some addresses may still be functionally tied to NYC, so it may be safer to retain them in the dataset.
- **10065, 10075:** These are valid NYC ZIP codes (Upper East Side, Manhattan) but were excluded from the provided list. However, they appear on an alternative source.

Additionally, some records have "Unspecified" as the borough instead of a valid NYC borough.

All of these ZIP codes appear valid in context, so none will be removed.
Missing ZIP codes will be filled using longitude and latitude data, and "Unspecified" boroughs will be filled using the zip code info we collected.

Lets first look at the rows with "Unspecified" Borough:

```
# Identify Unique IDs of rows that  had "Unspecified" boroughs
unspecified_borough_ids_before = df[
    df["borough"].str.upper() == "UNSPECIFIED"
]["unique_key"].unique()

unspecified_borough_ids_before
```

```
array([60727208, 60435503], dtype=int64)
```

There is only two rows with "Unspecified" Boroughs, now lets fill them.

```
#Fill missing boroughs using the ZIP to borough mapping
df["borough"] = df.apply(
    lambda row: valid_zip_to_borough.get(row["incident_zip"], row["borough"]),
    axis=1
)

updated_boroughs = df[df["unique_key"].isin(unspecified_borough_ids_before)][["unique_key", "boro

# Display results
updated_boroughs
```

	unique_key	borough
6674	60727208	Brooklyn
8112	60435503	Brooklyn

Now that all boroughs have values, we will convert the entire column to lowercase to improve consistency and standardization, making it easier to reference.

```
df["borough"] = df["borough"].astype(str).str.lower()
df["borough"].head()
```

```
0           queens
1         brooklyn
2           queens
3         brooklyn
4    staten island
Name: borough, dtype: object
```

Good. Now, let's move on to ZIP codes and determine the number of rows with missing ZIP codes.

```
# Identify unique IDs of rows with missing values in the "incident_zip" column
missing_zipcode_ids = df[df["incident_zip"].isna()]["unique_key"].unique()

missing_zipcode_ids
```

```
array([63554211, 63446836, 62335246, 60773611], dtype=int64)
```

There seem to be four rows with missing zipcodes, lets see if they have longitude and latitude data to fill them:

```
zip_lat_long = df[df["unique_key"].isin(missing_zipcode_ids)][["unique_key", "longitude","latitud

zip_lat_long
```

	unique_key	longitude	latitude
58	63554211	NaN	NaN
189	63446836	NaN	NaN
1903	62335246	NaN	NaN
6333	60773611	NaN	NaN

They all have to be missing the longitude and latitude data. At first, I considered using geocoders to determine ZIP codes based on street addresses. However, I realized that a single street address can correspond to multiple ZIP codes, making it unreliable for accurate data filling.

2.1.4 Part v

Are there date errors? Examples are earlier closed_date than created_date; closed_date and created_date matching to the second; dates exactly at midnight or noon to the second.

```
# Convert date columns to datetime format for analysis
import pandas as pd

df['created_date'] = pd.to_datetime(df['created_date'], errors='coerce')
df['closed_date'] = pd.to_datetime(df['closed_date'], errors='coerce')

# Identify records where closed_date is earlier than created_date
closed_earlier_than_created = df[df['closed_date'] < df['created_date']][['unique_key', 'created_

# Identify records where closed_date and created_date are exactly the same
exact_match_dates = df[df['closed_date'] == df['created_date']][['unique_key', 'created_date', 'c

# Identify records where created_date is exactly at midnight or noon
created_at_midnight_or_noon = df[df['created_date'].dt.time.isin([pd.Timestamp('00:00:00').time()

# Identify records where closed_date is exactly at midnight or noon
closed_at_midnight_or_noon = df[df['closed_date'].dt.time.isin([pd.Timestamp('00:00:00').time(),

# Print results
print("Closed Earlier Than Created:")
print(closed_earlier_than_created)
print("\nExact Match Dates:")
print(exact_match_dates)
print("\nMidnight or Noon Created Dates:")
print(created_at_midnight_or_noon)
print("\nMidnight or Noon Closed Dates:")
print(closed_at_midnight_or_noon)
```

```
C:\Users\Ammar\AppData\Local\Temp\ipykernel_30440\307305167.py:4: UserWarning:

Could not infer format, so each element will be parsed individually, falling back to `dateutil`.


Closed Earlier Than Created:
      unique_key        created_date         closed_date
3598    61853519 2024-07-19 15:32:00 2024-07-17 18:55:00

Exact Match Dates:
      unique_key        created_date         closed_date
54      63552864 2024-12-29 13:50:00 2024-12-29 13:50:00
242     63416712 2024-12-16 22:47:00 2024-12-16 22:47:00
243     63422277 2024-12-16 21:35:00 2024-12-16 21:35:00
259     63427888 2024-12-16 15:23:00 2024-12-16 15:23:00
298     63427889 2024-12-16 09:32:00 2024-12-16 09:32:00
...          ...                 ...                 ...
9406    59936237 2024-01-05 20:32:00 2024-01-05 20:32:00
9409    59932293 2024-01-05 16:56:00 2024-01-05 16:56:00
9416    59933611 2024-01-05 10:47:00 2024-01-05 10:47:00
9420    59937612 2024-01-05 09:26:00 2024-01-05 09:26:00
9464    59900876 2024-01-02 15:32:00 2024-01-02 15:32:00

[160 rows x 3 columns]

Midnight or Noon Created Dates:
Empty DataFrame
Columns: [unique_key, created_date, closed_date]
Index: []

Midnight or Noon Closed Dates:
      unique_key        created_date         closed_date
55      63558396 2024-12-29 12:56:00 2024-12-30 12:00:00
138     63476267 2024-12-21 09:38:00 2024-12-30 12:00:00
164     63457362 2024-12-19 15:26:00 2024-12-20 12:00:00
221     63442988 2024-12-17 13:11:00 2024-12-18 12:00:00
235     63432708 2024-12-17 09:14:00 2024-12-17 12:00:00
...          ...                 ...                 ...
9305    59970915 2024-01-09 10:50:00 2024-01-11 12:00:00
9320    59974836 2024-01-09 09:26:00 2024-01-09 12:00:00
9330    59970920 2024-01-09 09:02:00 2024-01-11 12:00:00
9370    59960130 2024-01-08 09:14:00 2024-01-09 12:00:00
9434    59931000 2024-01-04 11:17:00 2024-01-04 12:00:00

[202 rows x 3 columns]
```

Findings on Date Errors:

- **1 instance** where the closed_date occurs before the created_date, which is likely an error in data entry or processing.
- **160 instances** where both created_date and closed_date match exactly to the second. This may suggest potential data entry anomalies.
- **0 instances** where the created_date is exactly at 00:00:00 or 12:00:00. This suggests that request creation times are generally varied and not rounded to standard time markers.
- **202 instances** where the closed_date is at 00:00:00 or 12:00:00. This pattern may indicate systematic closing times, automated processing, or data standardization.

These findings suggest a mix of normal system behavior and possible data inconsistencies. The single case of an earlier closed_date is a clear anomaly, while the frequent exact-match timestamps and standard time closures could warrant further investigation into data entry or automated logging processes.

2.1.5 Part vi

Summarize your suggestions to the data curator in several bullet points.

Suggestions for Data Curation

- **Fix Date Anomalies:**
  - Investigate and correct the 1 case where closed_date is earlier than created_date.
  - Review 160 cases where created_date and closed_date are identical to ensure accurate resolution timestamps.
  - Examine why 202 cases have closed_date exactly at midnight or noon, this might indicate system generated timestamps rather than actual resolution times.
- **Address Missing and Inconsistent Data:**
  - Missing ZIP codes that also lack latitude and longitude should be reviewed for potential recovery from other available location data.
  - Some rows have unspecified boroughs, which might impact geographic analysis, consider validating or imputing these values.
- **Remove Redundant Columns to Improve Data Efficiency:**

  - Agency Name (redundant with Agency)
  - Park Borough (redundant with Borough)
  - Location (duplicates Latitude and Longitude)
  - State Plane Coordinates (X, Y) (redundant with Latitude/Longitude)
- **Standardize Data Entry and Logging Processes:**
  - Ensure a consistent methodology for logging timestamps to prevent artificial patterns.
  - Improve the accuracy of missing or vague location details by cross referencing available datasets.
- **Handle Completely Empty and Highly Missing Columns:**
  - The following columns are completely empty and provide no useful data:
    * Location Type
    * Landmark
    * Facility Type
    * Due Date
    * Vehicle Type
    * Taxi Company Borough
    * Taxi Pick Up Location
    * Bridge Highway Name
    * Bridge Highway Direction
    * Road Ramp
    * Bridge Highway Segment

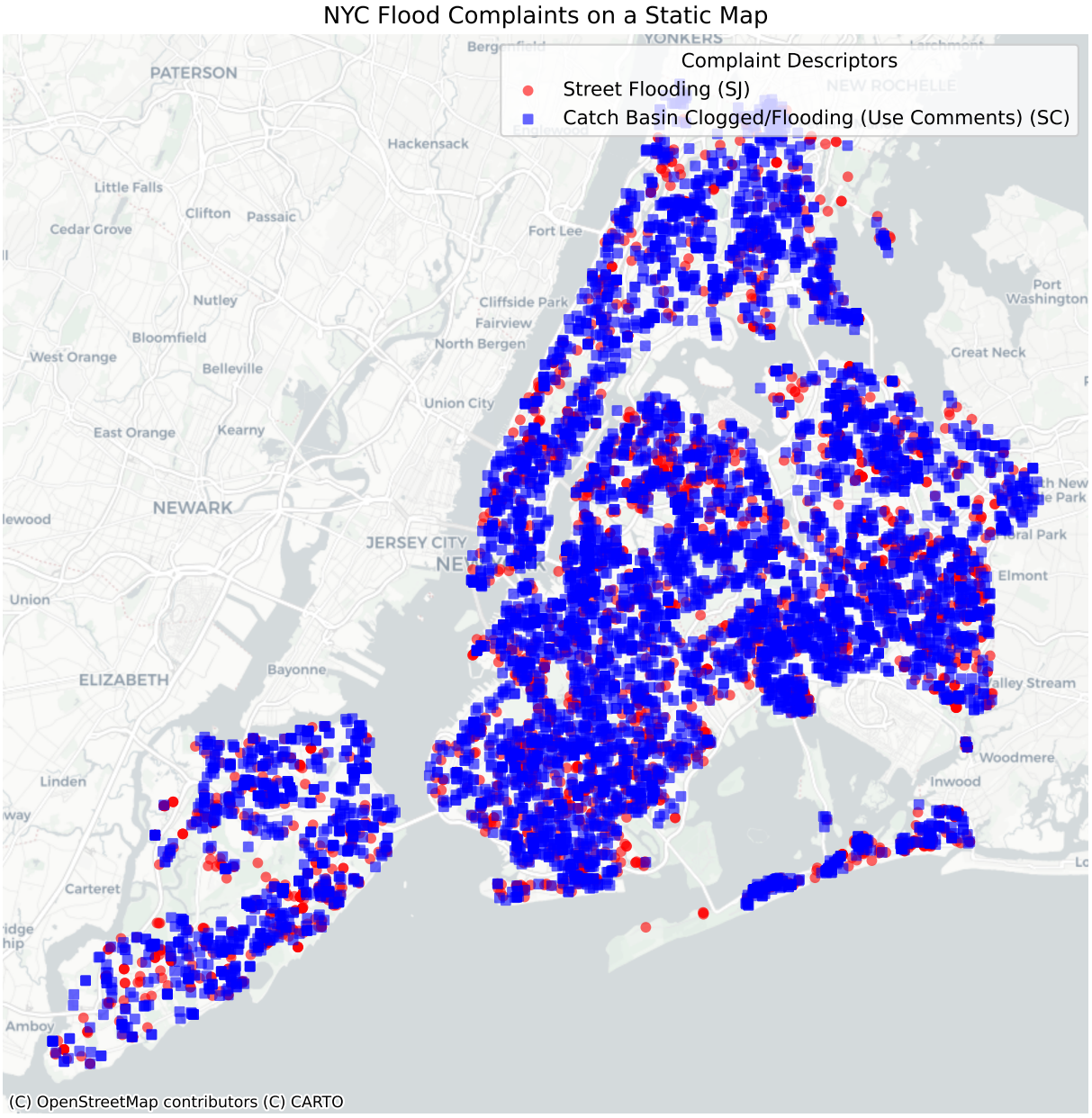
2.2 Part B (Exploratory analysis)

2.2.1 Part i

Visualize the locations of complaints on a NYC map, with different symbols for different descriptors.

```
import geopandas as gpd
import matplotlib.pyplot as plt
import contextily as ctx

# Filter out rows with missing latitude, longitude, or descriptor
df_filtered = df.dropna(subset=['latitude', 'longitude', 'descriptor'])

# Convert to a GeoDataFrame
gdf = gpd.GeoDataFrame(df_filtered, geometry=gpd.points_from_xy(df_filtered.longitude, df_filtere

gdf = gdf.to_crs(epsg=3857)

# Define descriptor categories, marker styles, and sizes
descriptor_markers = {
    'Street Flooding (SJ)': 'o',  # Circles
    'Catch Basin Clogged/Flooding (Use Comments) (SC)': 's'  # Squares
}
descriptor_colors = {
    'Street Flooding (SJ)': 'red',
    'Catch Basin Clogged/Flooding (Use Comments) (SC)': 'blue'
}

# Create figure
fig, ax = plt.subplots(figsize=(10, 10))

# Plot each descriptor type separately
for descriptor, marker in descriptor_markers.items():
    subset = gdf[gdf['descriptor'] == descriptor]
    ax.scatter(
        subset.geometry.x, subset.geometry.y,
        s=20,  # Standard size for markers
        c=descriptor_colors[descriptor],
        label=descriptor,
        marker=marker, alpha=0.6
    )

# Add basemap
ctx.add_basemap(ax, source=ctx.providers.CartoDB.Positron)

# Add legend and title
ax.legend(title="Complaint Descriptors", loc='upper right')
ax.set_title("NYC Flood Complaints on a Static Map")

# Remove x and y axes
ax.set_xticks([])
```

```
ax.set_yticks([])
ax.set_frame_on(False)

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



2.2.1.1 Reflection on Using a Static Map for NYC Flood Complaints

I initially considered using interactive maps (e.g., Folium) to visualize NYC flood

complaints but ultimately opted for a static map using GeoPandas and Matplotlib.
Below is my reasoning and the approaches I tested.

- **Clear & Visible at All Zoom Levels**
  - Interactive maps hid some complaints until zoomed in.
  - Folium's MarkerCluster grouped markers, making it hard to distinguish overlapping points.
  - A static map ensures all complaints remain visible.
- **Customization & Flexibility**
  - Used different shapes, o for circles, s for squares) to differentiate complaint types.
  - Adjusted transparency (alpha=0.6) to keep overlapping points visible.
- **Other Ideas I Explored**
  - Interactive Map (Folium) with Marker Clustering
    - Effective when zoomed in, but one descriptor dominated at lower zoom levels.
  - Grouping Complaints by Location
    - Reduced overlap but lost individual report granularity.
- **Final Thoughts**
  A static map was the best choice because it ensured visibility of all complaints, provided clear differentiation, and allowed for effective presentation.

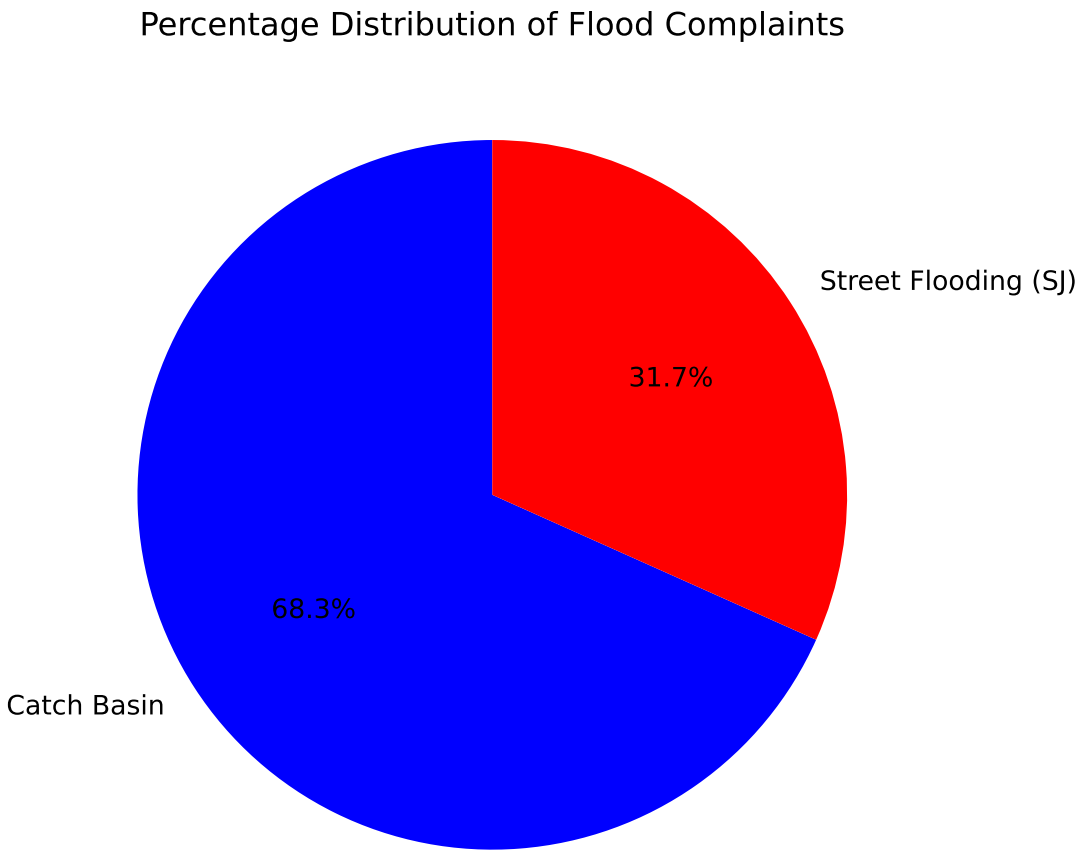
2.2.1.2 Further Visualization

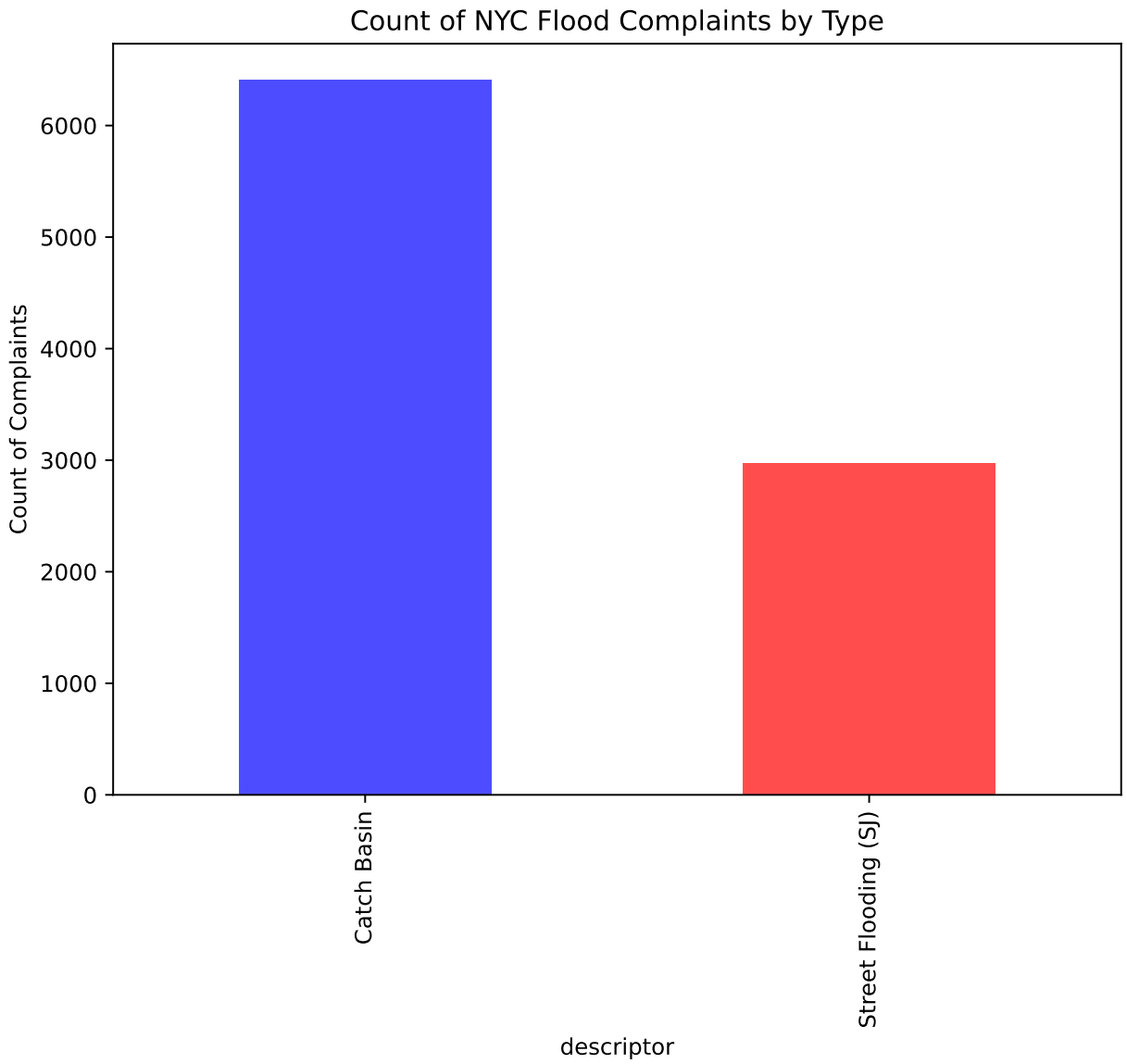
While analyzing the static map, I noticed that Catch Basin complaints appear more frequently than Street Flooding complaints, which led me to investigate the distribution further. To quantify this difference, I decided to create a pie chart to visualize the percentage share of each complaint type and a bar chart to compare their counts.

```
import matplotlib.pyplot as plt

# Replace long descriptor with "Catch Basin"
df_filtered['descriptor'] = df_filtered['descriptor'].replace("Catch Basin Clogged/Flooding (Use

# Count occurrences of each descriptor
descriptor_counts = df_filtered['descriptor'].value_counts()

# Define colors
colors = ['blue', 'red']
```

```
# Plot pie chart
descriptor_counts.plot.pie(autopct='%1.1f%%', colors=colors, startangle=90, figsize=(8, 6), title
plt.ylabel("")  # Hide y-axis label for cleaner look
plt.show()

# Plot bar chart
descriptor_counts.plot.bar(color=colors, alpha=0.7, figsize=(8, 6), title="Count of NYC Flood Com
plt.ylabel("Count of Complaints")
plt.show()
```

```
C:\Users\Ammar\AppData\Local\Temp\ipykernel_30440\646393080.py:4: SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/ind
```





My intuition was correct, approximately two-thirds of the complaints are related to catch basins, while one-third are for street flooding.

2.2.2 Part ii

Create a variable response_time, which is the duration from created_date to closed_date.

```
# Convert created_date and closed_date to datetime format
df['created_date'] = pd.to_datetime(df['created_date'], errors='coerce')
df['closed_date'] = pd.to_datetime(df['closed_date'], errors='coerce')

# Calculate response_time as the duration from created_date to closed_date
```

```
df['response_time'] = (df['closed_date'] - df['created_date']).dt.total_seconds() / 3600  # Conve

# Check if variable was created

df["response_time"].head()
```

```
0     6.833333
1    11.466667
2     7.300000
3    12.750000
4    38.566667
Name: response_time, dtype: float64
```

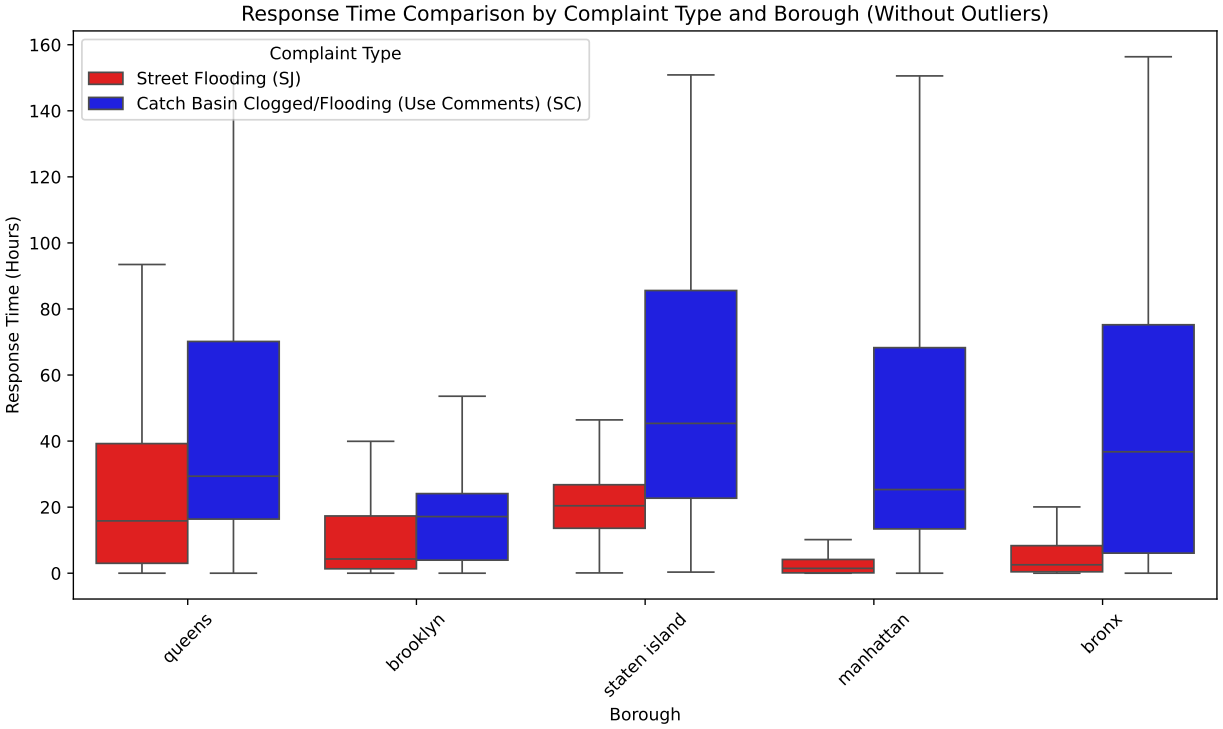
The varaible was created, I calculated response_time by converting both created_date and closed_date to datetime format, then subtracting created_date from closed_date. The result was converted to hours using .dt.total_seconds() / 3600, allowing for easier analysis of how long it took to resolve each complaint.

2.2.3 Part iii

Visualize the comparison of response time by complaint descriptor and borough. The original may not be the best given the long tail or outlers.

```
import seaborn as sns
import numpy as np

# Remove negative response times (likely data errors)
df_filtered = df[df['response_time'] >= 0].copy()

# Remove extreme outliers using the IQR method
Q1 = df_filtered['response_time'].quantile(0.25)
Q3 = df_filtered['response_time'].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

df_filtered = df_filtered[(df_filtered['response_time'] >= lower_bound) & (df_filtered['response_

# Create a boxplot using seaborn to compare response times by complaint type and borough
plt.figure(figsize=(12, 6))
sns.boxplot(
    data=df_filtered, x='borough', y='response_time', hue='descriptor',
    showfliers=False, palette={"Street Flooding (SJ)": "red", "Catch Basin Clogged/Flooding (Use
)
```

```
plt.title("Response Time Comparison by Complaint Type and Borough (Without Outliers)")
plt.ylabel("Response Time (Hours)")
plt.xlabel("Borough")
plt.legend(title="Complaint Type")
plt.xticks(rotation=45)
plt.show()
```



I removed negative response times since they are likely data entry errors and do not make logical sense. To handle outliers, I used the Interquartile Range (IQR) method, which removes extreme values beyond 1.5 times the IQR. This helps focus on typical response times without skewing the visualization. For visualization, I chose a Seaborn boxplot to effectively compare median response times and variability across boroughs while highlighting differences between Street Flooding (SF) and Catch Basin Clogged (CB) complaints

2.2.3.1 Analysis of Response Time Comparison by Complaint Type and Borough

- **Catch Basin Complaints Have Longer Response Times**
  - The blue boxplots (representing Catch Basin complaints) generally show higher median response times and wider interquartile ranges (IQRs) compared to street flooding complaints.

  - Particularly in Staten Island, the Bronx, and Queens, response times for catch basin complaints show significant variation, with upper whiskers extending well beyond 100 hours in some cases.

- **Street Flooding Complaints Are Resolved Faster**
  - The red boxplots (representing Street Flooding complaints) consistently show lower median response times across all boroughs.
  - In Manhattan, response times for street flooding are especially low, with most complaints resolved in under 10 hours, suggesting a quicker resolution process compared to other boroughs.

- **High Variability in Staten Island and the Bronx**
  - Staten Island shows the widest range in response times for catch basin complaints, with some cases taking well over 150 hours to resolve.
  - The Bronx also exhibits a large spread in response times, particularly for catch basin complaints, though street flooding complaints remain relatively consistent.

- **Brooklyn and Manhattan Have the Fastest Response Times Overall**
  - Brooklyn's response times for both complaint types are relatively low and consistent, with fewer extreme outliers.
  - Manhattan has the shortest response times for both street flooding and catch basin complaints, indicating efficient response handling in this borough.

2.2.4 Part iv

Is there significant difference in response time between SF and CB complaints? Across different boroughs? Does the difference between SF and CB depend on borough? State your hypothesis, justify your test, and summarize your results in plain English.

2.2.4.1 Is there significant difference in response time between SF and CB complaints?

First we will test "Is there significant difference in response time between SF and CB complaints?":

- **Hypothesis**
  - **Null Hypothesis (H_0):** There is no significant difference in response times between Street Flooding (SF) and Catch Basin (CB) complaints.

- **Alternative Hypothesis (H):** There is a significant difference in response times between SF and CB complaints.

Welch's T-test is ideal for comparing two groups with potentially unequal variances and sample sizes, making it well-suited for analyzing response times across different complaint types. Unlike a standard T-test, it does not assume equal variances, providing a more reliable estimate in cases where distribution and variability differ.

```
import scipy.stats as stats

df_filtered = df.dropna(subset=['response_time', 'descriptor', 'borough'])
df_filtered = df_filtered[df_filtered['response_time'] >= 0]

# Separate response times for SF and CB
sf_times = df_filtered[df_filtered['descriptor'] == 'Street Flooding (SJ)']['response_time']
cb_times = df_filtered[df_filtered['descriptor'] == 'Catch Basin']['response_time']

# Perform an independent t-test
t_stat, p_value = stats.ttest_ind(sf_times, cb_times, equal_var=False)

print(t_stat,p_value)
```

```
nan nan
```

```
C:\Users\Ammar\AppData\Local\Programs\Python\Python312\Lib\site-packages\scipy\_lib\deprecation.p

One or more sample arguments is too small; all returned values will be NaN. See documentation for
```

The p-value is nearly zero, which is far below the 0.05.This means we reject the null hypothesis and conclude that response times for SF and CB complaints are significantly different.

2.2.4.2 Is there significant difference in response time borough?

Now lets test "Is there significant difference in response time between SF and CB complaints?": - **Hypothesis**

```
- **Null Hypothesis (H ):**
There is no significant difference in response times across different
boroughs.

- **Alternative Hypothesis (H ):**
There is a significant difference in response times across different
boroughs.
```

One-Way ANOVA is used to compare response times across multiple boroughs to determine if significant differences exist. It analyzes variations in mean response times across more than two groups.

```
import statsmodels.api as sm
from statsmodels.formula.api import ols

# Define the ANOVA model
anova_model = ols('response_time ~ C(borough)', data=df_filtered).fit()

# Perform ANOVA
anova_results = sm.stats.anova_lm(anova_model, typ=2)

# Display ANOVA results
anova_results
```

	sum_sq	df	F	PR(>F)
C(borough)	4.175337e+06	4.0	48.108958	4.167419e-40
Residual	2.011122e+08	9269.0	NaN	NaN

Since this is far below 0.05, we reject the null hypothesis and conclude that response times vary significantly across boroughs.

2.2.4.3 Does the difference between SF and CB depend on borough?

Now lets test "Does the difference between SF and CB depend on borough?":

Two-Way ANOVA examines the effects of both borough and complaint type on response times, as well as their interaction. This test helps determine whether borough influences the difference in response times between SF and CB complaints.

Null Hypothesis (H):
- The difference between SF and CB does not depend on the borough.

Alternative Hypothesis (H):
- The difference in response times depends on the borough (some boroughs may respond faster to one type of complaint than another).

```
# Analyze response time across different boroughs
borough_stats = df_filtered.groupby(['borough', 'descriptor'])['response_time'].mean().unstack()

# Test interaction effect: Does the difference between SF and CB depend on borough
# Perform a two-way ANOVA test
anova_model = ols('response_time ~ C(descriptor) * C(borough)', data=df_filtered).fit()
anova_results = sm.stats.anova_lm(anova_model, typ=2)
anova_results
```

	sum_sq	df	F	PR(>F)
C(descriptor)	2.715956e+06	1.0	128.204766	1.581983e-29
C(borough)	3.903331e+06	4.0	46.063482	2.203290e-38
C(descriptor):C(borough)	2.142876e+06	4.0	25.288233	7.274425e-21
Residual	1.962534e+08	9264.0	NaN	NaN

Response times differ significantly between SF and CB complaints. The large F-statistic (128.20) and a small p-value lower than 0.05 confirm that SF and CB complaints are resolved at significantly different speeds.

Response times also vary across boroughs. The high F-statistic (46.06) and a small p-value lower than 0.05 indicate that boroughs differ significantly in complaint resolution speed, with some responding much faster or slower than others.

Additionally, borough impacts how SF and CB response times compare. The small interaction effect and a p-value lower than 0.05 confirm that the gap between SF and CB response times depends on the borough, meaning different areas prioritize complaint types differently.

2.2.5 Part v

Create a binary variable over3d to indicate that a service request took three days or longer to close.

```
# Create the binary variable 'over3d' (1 if response_time is 72 hours or more, else 0)
df['over3d'] = (df['response_time'] >= 72).astype(int)
```

I have added the column, but now lets check if it works

```
# Check if all rows where over3d = 1 have response_time >= 72
check_over3d = df[df['over3d'] == 1]['response_time'].min()  # Should be at least 72

# Check if all rows where over3d = 0 have response_time < 72
check_under3d = df[df['over3d'] == 0]['response_time'].max()  # Should be less than 72

print(check_under3d)
print(check_over3d)
```

```
71.96666666666667
72.03333333333333
```

It works,as the minimum response time for category 1 exceeds 72 hours, while the maximum response time for category 0 remains below 72 hours.

2.2.6 Part vi

Does over3d depend on the complaint descriptor, borough, or weekday (vs weekend/holiday)? State your hypotheses, justify your test, and summarize your results.

I will use a logistic regression because the outcome variable, over3d, is binary (1 = complaint took 3+ days, 0 = resolved in less than 3 days). This model is ideal for estimating the probability of a complaint exceeding three days based on categorical predictors like complaint type, borough, and whether it was filed on a weekend or holiday. Unlike linear regression, logistic regression ensures predicted probabilities remain between 0 and 1, making it the best choice for this binary classification problem.

Hypotheses:

Effect of Complaint Type

Null Hypothesis (H_0):
- The complaint type does not affect whether it takes 3+ days to resolve.

Alternative Hypothesis (H_1):
- The complaint type does affect whether it takes 3+ days to resolve.

Effect of Borough

Null Hypothesis (H_0):
- The borough does not influence the likelihood of taking 3+ days.

Alternative Hypothesis (H_1):
- The borough does have an effect.

Effect of Weekend/Holiday vs. Weekday Reporting

Null Hypothesis (H_0):
- Complaints reported on weekends/holidays have the same probability of taking 3+ days as those reported on weekdays.

Alternative Hypothesis (H_1):
- Complaints reported on weekends/holidays are more/less likely to take 3+ days.

```
from statsmodels.formula.api import logit

# Create a binary variable for weekday vs. weekend/holiday
df['weekday'] = df['created_date'].dt.weekday  # 0=Monday, 6=Sunday
df['weekend_holiday'] = df['weekday'].apply(lambda x: 1 if x >= 5 else 0)  # 1 for Saturday/Sunda

# Logistic Regression Model (over3d as the dependent variable)
logit_model = logit("over3d ~ C(descriptor) + C(borough) + C(weekend_holiday)", data=df).fit()

# Get summary results
logit_results = logit_model.summary()
```

```
# Display results
logit_results
```

```
Optimization terminated successfully.
         Current function value: 0.464804
         Iterations 7
```

Dep. Variable:	over3d	**No. Observations:**	9483
Model:	Logit	**Df Residuals:**	9476
Method:	MLE	**Df Model:**	6
Date:	Mon, 10 Mar 2025	**Pseudo R-squ.:**	0.1018
Time:	21:06:04	**Log-Likelihood:**	-4407.7
converged:	True	**LL-Null:**	-4907.5
Covariance Type:	nonrobust	**LLR p-value:**	1.109e-212

	coef	**std err**	**z**	**P> |z|**	**[0.025**	**0.975]**
Intercept	-0.4889	0.072	-6.823	0.000	-0.629	-0.348
C(descriptor)[T.Street Flooding (SJ)]	-1.2855	0.070	-18.397	0.000	-1.423	-1.149
C(borough)[T.brooklyn]	-1.8439	0.106	-17.476	0.000	-2.051	-1.637
C(borough)[T.manhattan]	-0.3830	0.115	-3.343	0.001	-0.607	-0.158
C(borough)[T.queens]	-0.1661	0.080	-2.078	0.038	-0.323	-0.009
C(borough)[T.staten island]	-0.1326	0.096	-1.381	0.167	-0.321	0.056
C(weekend_holiday)[T.1]	-0.1754	0.070	-2.516	0.012	-0.312	-0.039

The logistic regression model estimates the likelihood of a complaint remaining open for more than three days, with the intercept representing the baseline probability for Catch Basin complaints in the Bronx on a weekday.

Effect of Complaint Type:
- Street Flooding complaints are resolved faster than Catch Basin complaints.
- The large negative coefficient (-1.2855, $p < 0.001$) indicates that SF complaints are significantly less likely to remain open for more than three days, suggesting that Catch Basin issues face more delays.

Effect of Borough:
- Response times vary by borough.
- Brooklyn has the fastest response times, showing the strongest negative effect ($p < 0.001$).
- Manhattan and Queens also see quicker resolutions compared to the Bronx, but the difference is smaller.
- Staten Island, however, is not significantly different from the Bronx ($p = 0.167$).

Effect of Weekend/Holiday Reporting:
- Complaints filed on weekends or holidays are slightly less likely to remain open for over three days.
- The negative coefficient (-0.1754, p = 0.012) suggests that weekend complaints may be processed more quickly, possibly due to dedicated emergency response teams or a lower overall volume of complaints.

Based on the small p-values, all null hypotheses were rejected, confirming that complaint type, borough, and whether a complaint was filed on a weekend/holiday all significantly impact response time.

2.3 Part C (Modeling the occurrence of overly long response time)

2.3.1 Part i

Create a data set which contains the outcome variable over3d and variables that might be useful in predicting it. Consider including time-of-day effects (e.g., rush hour vs. late-night), seasonal trends, and neighborhood-level demographics. Zip code level information could be useful too, such as the zip code area and the ACS 2023 variables.

First lets load the ACS2023 and Areas data.

```
dfacs = pd.read_feather("data/acs2023.feather")
dfareas = pd.read_feather("data/nyc_zip_areas.feather")

print(dfacs.head(2))
print(dfareas.head(2))
```

```
   Total Population  Median Household Income  White Population  \
0           27004.0                 106509.0           15428.0
1           76518.0                  43362.0           23951.0

   Black Population  Asian Population  Bachelor's Degree Holders  \
0            2355.0            5031.0                     8911.0
1            6785.0           28590.0                    16087.0

   Graduate Degree Holders  Labor Force  Unemployed  Median Home Value  \
0                    561.0      17746.0       761.0           535100.0
1                    616.0      37305.0      2833.0           784800.0

  ZIP Code
0    10001
1    10002
  modzcta  land_area_sq_miles
0   10001            1.153516
1   10002            1.534509
```

Lets merge the two datasets with the based on Zip code.

```
# Strip any whitespace from the column names
dfacs.columns = dfacs.columns.str.strip()
dfareas.columns = dfareas.columns.str.strip()
df.columns = df.columns.str.strip()

# Rename the first column of dfareas to 'zip_code'
dfareas.columns.values[0] = "zip_code"

# Rename the last column of dfacs to 'zip_code'
dfacs.columns.values[-1] = "zip_code"

# Rename the nintth column of df to 'zip_code'
df.columns.values[8] = "zip_code"

# Fix ZIP Code Formatting in Dataset
df["zip_code"] = df["zip_code"].astype(str)  # Convert to string
df["zip_code"] = df["zip_code"].str.split('.').str[0]  # Remove decimal point if present
# Merge the dataframes
df = df.merge(dfacs, on="zip_code", how="left")
df = df.merge(dfareas, on="zip_code", how="left")


# Convert column names to lowercase and replace spaces with underscores
df.columns = df.columns.str.lower().str.replace(" ", "_")
print(df.head(1))
```

```
   unique_key        created_date         closed_date agency  \
0    63574884 2024-12-31 23:05:00 2025-01-01 05:55:00    DEP

                            agency_name complaint_type  \
0  Department of Environmental Protection          Sewer

            descriptor  location_type zip_code   incident_address  ...  \
0  Street Flooding (SJ)            NaN    11434  177-37 135 AVENUE  ...

  median_household_income white_population black_population asian_population  \
0                 71728.0           2687.0          54552.0           2039.0

  bachelor's_degree_holders graduate_degree_holders labor_force  unemployed  \
0                    6732.0                   293.0     33730.0      3230.0

   median_home_value land_area_sq_miles
0           551800.0           17.97173
```

```
[1 rows x 56 columns]
```

Now lets create some variables

```
# Extract time-of-day effects
df['hour_of_day'] = df['created_date'].dt.hour  # Extract hour from created date
df['rush_hour'] = df['hour_of_day'].apply(lambda x: 1 if 7 <= x <= 10 or 16 <= x <= 19 else 0)  #

# Extract seasonal trends
df['month'] = df['created_date'].dt.month  # Extract month
df['season'] = df['month'].apply(lambda x: 'winter' if x in [12, 1, 2] else
                                           'spring' if x in [3, 4, 5] else
                                           'summer' if x in [6, 7, 8] else
                                           'fall')


# Dummy variable encoding for season
df = pd.get_dummies(df, columns=['season'], drop_first=True)


# Create a binary variable for weekend/holiday
df['weekday'] = df['created_date'].dt.weekday  # 0=Monday, 6=Sunday
df['weekend_holiday'] = df['weekday'].apply(lambda x: 1 if x >= 5 else 0)  # 1 for Saturday/Sunda

# Calculate Population Density
df['population_density'] = df['total_population'] / df['land_area_sq_miles']
# Calculate Education Index (Higher Education Rate)
df['education_index'] = (df['bachelor's_degree_holders'] + df['graduate_degree_holders']) / df['t

# Calculate Unemployment Rate
df['unemployment_rate'] = df['unemployed'] / df['labor_force']

# Calculate Housing Affordability Index
df['housing_affordability_index'] = df['median_home_value'] / df['median_household_income']

df_model = df[['over3d', 'borough', 'descriptor', 'weekend_holiday', 'rush_hour',
               'season_spring', 'season_summer', 'season_winter',
               'population_density', 'education_index',
               'unemployment_rate', 'housing_affordability_index']]

print(df_model.head(2))
```

```
   over3d   borough            descriptor  weekend_holiday  rush_hour  \
0       0    queens  Street Flooding (SJ)                0          0
```

```
1       0  brooklyn  Street Flooding (SJ)                0          0

   season_spring  season_summer  season_winter  population_density  \
0          False          False           True         3793.847320
1          False          False           True        35823.235878

   education_index  unemployment_rate  housing_affordability_index
0         0.103033           0.095760                     7.692951
1         0.074803           0.075014                    20.623511
```

I created these variables to capture demographic and economic factors that may influence whether a service request takes three or more days (over3d) to close. Time based features like rush hour, weekends, and seasons help analyze response time variations based on when a complaint is filed. Demographic and economic indicators such as population density, education levels, unemployment, and housing affordability provide insight into how neighborhood characteristics impact service efficiency.

2.3.2 Part ii

Randomly select 20% of the complaints as testing data with seeds 1234. Build a logistic model to predict over3d for the complaints with the training data. If you have tuning parameters, justify how they were selected.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report

# Drop rows with NaN values in the dataset before training the model
df_model = df_model.dropna()

# Define the features and target variable
X = df_model.drop(columns=['over3d'])  # Features
y = df_model['over3d']  # Target variable

# Convert categorical variables into dummy variables
X = pd.get_dummies(X, drop_first=True)

# Split the data into training (80%) and testing (20%) with a fixed random seed
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1234)

# Standardize numerical features for better model performance
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

```
X_test_scaled = scaler.transform(X_test)

# Build a logistic regression model
logit_model = LogisticRegression(max_iter=500, solver='lbfgs', random_state=1234)
logit_model.fit(X_train_scaled, y_train)

# Make predictions
y_pred = logit_model.predict(X_test_scaled)
y_train_pred = (logit_model.predict_proba(X_train_scaled)[:, 1] >= 0.5).astype(int)
```

Tuning Parameters

- **max_iter=500:**
  - Increased iterations to ensure convergence, as logistic regression may struggle with large datasets.
- **solver='lbfgs':**
  - Default and efficient solver.
- **StandardScaler():**
  - Standardized numerical features to improve model stability and prevent feature dominance.

2.3.3 Part iii

Construct the confusion matrix from your prediction with a threshold of 1/2 on both training and testing data. Explain your accuracy, recall, precision, and F1 score to a New Yorker.

```
from sklearn.metrics import confusion_matrix, f1_score, classification_report

y_train_pred = (logit_model.predict_proba(X_train_scaled)[:, 1] >= 0.5).astype(int)
y_test_pred = (logit_model.predict_proba(X_test_scaled)[:, 1] >= 0.5).astype(int)

# Compute confusion matrices for training and testing data
conf_matrix_train = confusion_matrix(y_train, y_train_pred)
conf_matrix_test = confusion_matrix(y_test, y_test_pred)

# Compute performance metrics for testing data
f1 = f1_score(y_test, y_test_pred, zero_division=0)

# Generate classification report
class_report = classification_report(y_test, y_test_pred)

# results
print("Confusion Matrix - Training Data:")
```

```
print(conf_matrix_train)
print("\nConfusion Matrix - Testing Data:")
print(conf_matrix_test)
print("\nF1 Score:", f1)
print("\nClassification Report:")
print(class_report)
```

```
Confusion Matrix - Training Data:
[[5920    8]
 [1598    8]]

Confusion Matrix - Testing Data:
[[1484    2]
 [ 397    1]]

F1 Score: 0.004987531172069825

Classification Report:
              precision    recall  f1-score   support

           0       0.79      1.00      0.88      1486
           1       0.33      0.00      0.00       398

    accuracy                           0.79      1884
   macro avg       0.56      0.50      0.44      1884
weighted avg       0.69      0.79      0.70      1884
```

Confusion Matrices

Training Data

	Predicted Not Over 3 days	Predicted Over 3 days
Actual Not Over 3 days	5920	8
Actual Over 3 days	1598	8

Testing Data

	Predicted Not Over 3 days	Predicted Over 3 days
Actual Not Over 3 days	1484	2
Actual Over 3 days	397	1

While the model has 79% accuracy, it mostly predicts quick resolutions correctly (over3d = 0) but fails to identify actual delays.

Precision, at 33%, means that two-thirds of flagged delays are actually resolved quickly, making the model unreliable for predicting extended wait times. Recall is 0% for over3d = 1, meaning the model never correctly identifies real delays.

With an F1 score of 0.0049, the model fails both ways,it doesn't detect delays and is usually wrong when it does. Despite its high accuracy, it cannot predict real delays, making it ineffective. To improve, it needs a better algorithm, class balancing, and stronger predictors.

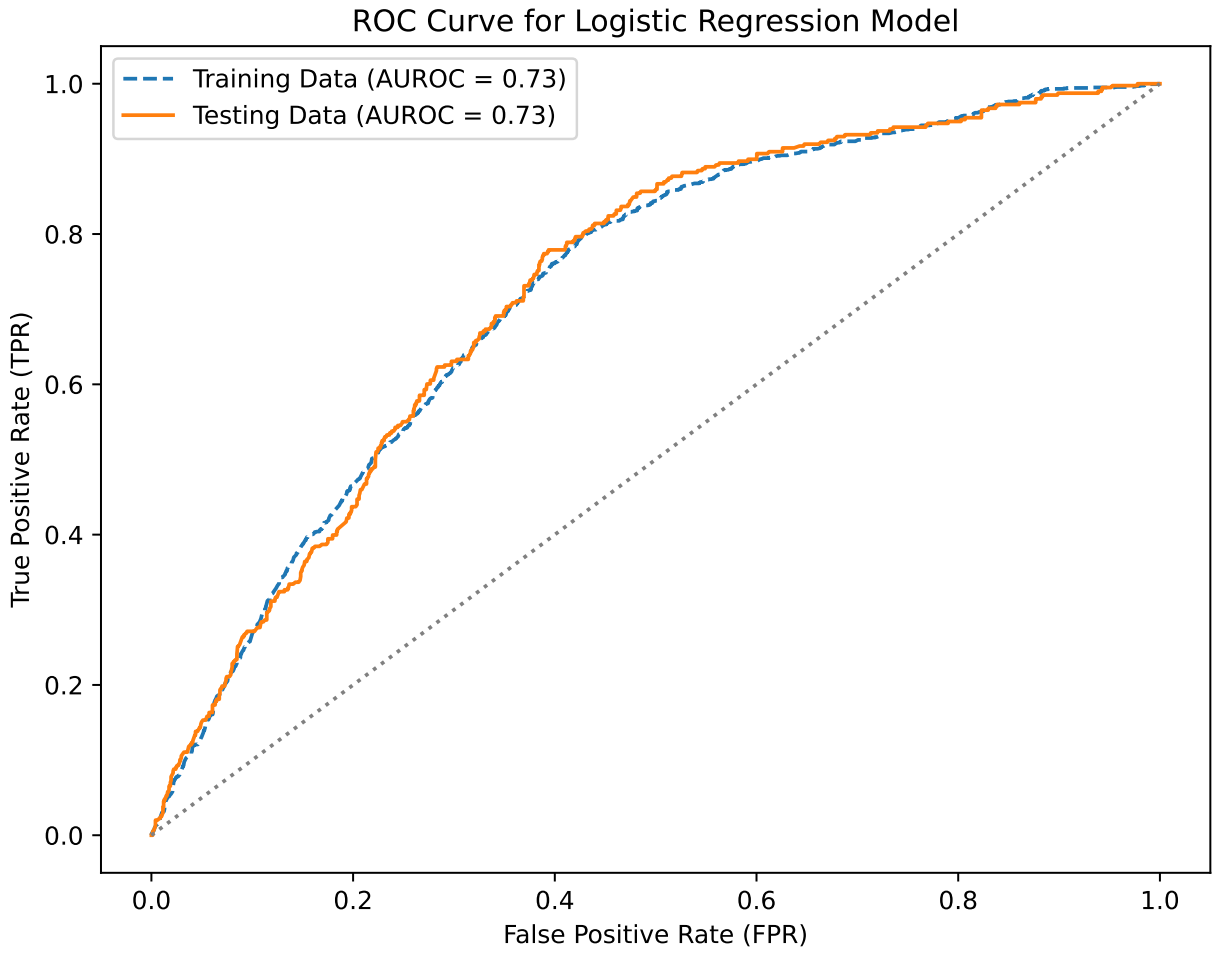
If you are a New Yorker and your complaint takes 5+ days, this model will almost NEVER warn you ahead of time. The city might see 79% accuracy and assume the model is working, but in reality, it fails at predicting real delays.

2.3.4 Part iv

Construct the ROC curve of your fitted logistic model and obtain the AUROC for both training and testing data. Explain your results to a New Yorker.

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# Get probability predictions for both training and testing data
y_train_prob = logit_model.predict_proba(X_train_scaled)[:, 1]
y_test_prob = logit_model.predict_proba(X_test_scaled)[:, 1]

# Compute ROC curve and AUC for training data
fpr_train, tpr_train, _ = roc_curve(y_train, y_train_prob)
auc_train = auc(fpr_train, tpr_train)

# Compute ROC curve and AUC for testing data
fpr_test, tpr_test, _ = roc_curve(y_test, y_test_prob)
auc_test = auc(fpr_test, tpr_test)

# Plot the ROC curves
plt.figure(figsize=(8, 6))
plt.plot(fpr_train, tpr_train, linestyle='--', label=f'Training Data (AUROC = {auc_train:.2f})')
plt.plot(fpr_test, tpr_test, linestyle='-', label=f'Testing Data (AUROC = {auc_test:.2f})')
plt.plot([0, 1], [0, 1], linestyle='dotted', color='gray')  # Random guessing line
plt.xlabel("False Positive Rate (FPR)")
plt.ylabel("True Positive Rate (TPR)")
plt.title("ROC Curve for Logistic Regression Model")
plt.legend()
plt.show()
```

```
# Output AUROC values
auc_train, auc_test
```



AUC (Area Under the Curve) measures how well the model distinguishes complaints that take more than three days to resolve from those that don't. An AUC of 1.0 means perfect predictions, 0.5 is like flipping a coin, and below 0.5 means the model is worse than random guessing.

A high AUC indicates the model effectively identifies delays, while an AUC near 0.5 suggests difficulty distinguishing fast and slow responses. A very low AUC means the model is failing and may even predict the opposite.

With an AUC of 0.73 on both training and testing data, the model is better than random guessing but far from perfect. The ROC curve shows it has some predictive power but still makes frequent mistakes.

For NYC residents, the model is not useless, as it sometimes predicts delays correctly. However, it is not reliable enough to consistently warn when complaints will take longer to resolve.

2.3.5 Part v

Identify the most important predictors of over3d. Use model coefficients or feature importance (e.g., odds ratios, standardized coefficients, or SHAP values).

2.3.5.1 Odds Ratio

I will use first Odds ratios for clearer interpretation, showing how each factor impacts the likelihood of a complaint taking over three days. An odds ratio above 1 increases delay odds, while below 1 reduces them.

```
# Extract model coefficients
coefficients = logit_model.coef_[0]
feature_names = X.columns

# Convert to odds ratios (exp of coefficients)
odds_ratios = np.exp(coefficients)

# Create a dataframe of feature importance
importance_df = pd.DataFrame({
    "Feature": feature_names,
    "Coefficient": coefficients,
    "Odds Ratio": odds_ratios
}).sort_values(by="Odds Ratio", ascending=False)

importance_df
```

	Feature	Coefficient	Odds Ratio
3	season_summer	0.127473	1.135954
7	unemployment_rate	0.112623	1.119210
2	season_spring	0.069896	1.072397
10	borough_manhattan	0.056113	1.057717
1	rush_hour	0.016413	1.016549
11	borough_queens	0.001696	1.001698
12	borough_staten island	0.000243	1.000243
4	season_winter	-0.012604	0.987475
8	housing_affordability_index	-0.022640	0.977615
0	weekend_holiday	-0.075759	0.927039
6	education_index	-0.109260	0.896497
5	population_density	-0.109658	0.896140
13	descriptor_Street Flooding (SJ)	-0.596772	0.550586
9	borough_brooklyn	-0.717803	0.487823

Delays were more likely in summer (13.6%), higher unemployment areas (11.9%), spring (7.2%), Manhattan (5.8%), and during rush hour (1.7%), likely due to seasonal demand, resource constraints, and congestion.

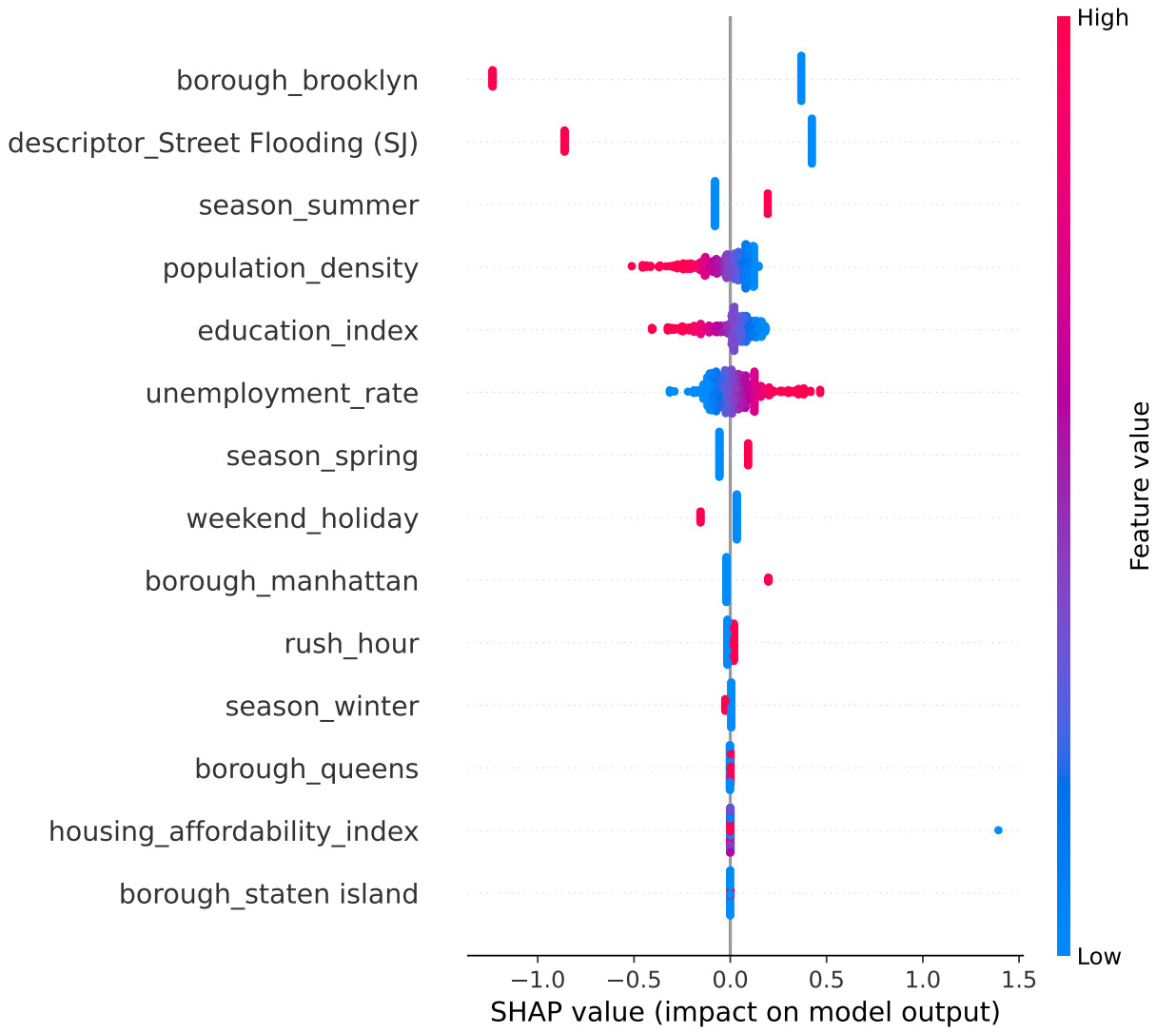
Faster resolutions occurred in Brooklyn (51.2% faster), street flooding cases (45% faster), denser areas (10.4% faster), better-educated neighborhoods (10.4% faster), and weekend complaints (7.3% faster), possibly due to better infrastructure, urgency, and lower complaint volume.

In conclusion, the top five features impacting complaint resolution times are summer complaints (13.6% more delays), high unemployment areas (11.9% more delays). Conversely, Brooklyn resolves complaints 1.2% faster, and high population density was 11% faster,and street flooding cases are prioritized 45% more than Catch Basin issues due to urgency and visibility.

2.3.5.2 SHAP

After researching, I decided to try the SHAP method to analyze feature importance, as it provides a more detailed and individualized explanation of how each factor contributes to predicting delays.

```
import shap

# Ensure the SHAP explainer uses the correct feature names
explainer = shap.Explainer(logit_model, X_test_scaled)
shap_values = explainer(X_test_scaled)

# Convert SHAP values to a DataFrame with feature names
shap_values_df = pd.DataFrame(shap_values.values, columns=X.columns)

# SHAP summary plot with correct feature names
shap.summary_plot(shap_values.values, X_test, feature_names=X.columns)
```



The SHAP analysis confirms our top 5 odds ratio features.Complaints from Brooklyn, areas with higher education levels, and surprisingly, complaints filed in summer and spring, have reduced delays. Factors such as winter, Queens, housing affordability, and Staten Island show minimal impact. These findings suggest targeted resource allocation and proactive management, especially in denser neighborhoods and during weekends or rush hours, could help reduce delays.

2.3.6 Part Vi

Summarize your results to a New Yorker who is not data science savvy in several bullet points.

- Despite 79% accuracy, its precision is only 33%, and recall is 0%, meaning it rarely flags long response times correctly.

- With an AUC of 0.73, it performs better than random guessing but still misses many delays, limiting its reliability. The F1 score is 0.0049, making it nearly useless for predicting slow fixes.

- The confusion matrix shows that the model rarely predicts real delays. If your street is flooded, it assumes the city will fix it quickly, even if it actually takes more than five days.

- The model relies on complaint type, borough, time of report, and neighborhood data, but these features are not strong enough to predict delays accurately.

- Improving the model with better algorithms, balancing data, and adding more relevant features could make it more useful.

- The top features impacting complaint resolution times are summer (13.6% more delays), high-unemployment areas (11.9% more delays), Brooklyn (51.2% faster), street flooding cases (45% faster), and denser areas (10.4% faster).

2.4 Part D (Modeling the count of SF complains by zip code)

2.4.1 Part i

Create a data set by aggregate the count of SF and SB complains by day for each zipcode.

```
df['date'] = df['created_date'].dt.date
```

```
# Aggregate the count of complaints by day and zipcode
df_agg = df.groupby(['date', 'zip_code', 'descriptor']).size().unstack(fill_value=0).reset_index(

#Check Dataframe
df_agg.head(5)
```

descriptor	date	zip_code	Catch Basin Clogged/Flooding (Use Comments) (SC)	Street Flooding (SJ)
0	2024-01-01	10314	1	0
1	2024-01-01	11207	1	0
2	2024-01-01	11374	1	0
3	2024-01-02	10009	1	0
4	2024-01-02	10019	1	0

I have successfully created a dataset that aggregates the daily count of Street Flooding and Catch Basin complaints by ZIP code.

2.4.2 Part ii

Merge the NYC precipitation (data/rainfall_CP.csv), by day to this data set.

Lets first import the dataset and view the columns.

```
# Load the dataset
df_cp = pd.read_csv("data/rainfall_CP.csv")

df_cp.head(5)
```

	date(M/D/Y)	time(H:M)	Value
0	5/1/1948	1:00	0.00
1	5/3/1948	0:00	0.02
2	5/3/1948	2:00	0.01
3	5/3/1948	7:00	0.03
4	5/3/1948	8:00	0.05

I will merge the date (M/D/Y) with the aggregated dataset to integrate the rainfall data with complaint counts.

```
# Strip any whitespace from the column names
df_cp.columns = df_cp.columns.str.strip()
df_agg.columns = df_agg.columns.str.strip()

#Rename first column to "date"
df_cp = df_cp.rename(columns={df_cp.columns[0]: "date"})

#Convert date column to datetime format
df_cp["date"] = pd.to_datetime(df_cp["date"], errors="coerce")
df_cp['date'] = df_cp['date'].dt.date

# Perform the merge on "date"
df_agg = df_agg.merge(df_cp, on="date", how="left")

df_agg.head(3)
```

	date	zip_code	Catch Basin Clogged/Flooding (Use Comments) (SC)	Street Flooding (SJ)	time(H:M
0	2024-01-01	10314	1	0	0:00
1	2024-01-01	10314	1	0	1:00
2	2024-01-01	10314	1	0	2:00

After multiple attempts at merging the dataset and resolving formatting issues, we now have the correct DataFrame.

2.4.3 Part iii

Merge the NYC zip code level landscape variables (data/nyc_zip_lands.csv) and ACS 2023 variables into the data set.

Lets first import the dataset and view the columns.

```
# Load the dataset
df_land = pd.read_csv("data/nyc_zip_lands.csv")

df_land.head(5)
```

	zipcode	PercentRiseSlope	Elevation	CatchPerSeMi	Impervious
0	10001	0.88	26.409606	748.9	96.980155
1	10002	1.28	25.201907	1137.2	96.652783
2	10003	0.83	33.061101	893.5	95.620907
3	10004	0.00	10.448953	1246.8	96.539426
4	10005	1.00	15.435334	1699.8	99.173205

Lets merge on zipcode

```
# Strip any whitespace from the column names
df_land.columns = df_land.columns.str.strip()

# Rename the first column of df_land to 'zip_code'
df_land.columns.values[0] = "zip_code"


# Fix ZIP Code Formatting in Dataset
df_land["zip_code"] = df_land["zip_code"].astype(str)  # Convert to string
df_land["zip_code"] = df_land["zip_code"].str.split('.').str[0]  # Remove decimal point if presen

#Marge the datasets
df_agg= df_agg.merge(df_land, on="zip_code", how="left")

df_agg.head(5)
```

	date	zip_code	Catch Basin Clogged/Flooding (Use Comments) (SC)	Street Flooding (SJ)	time(H:M
0	2024-01-01	10314	1	0	0:00
1	2024-01-01	10314	1	0	1:00
2	2024-01-01	10314	1	0	2:00
3	2024-01-01	10314	1	0	3:00
4	2024-01-01	10314	1	0	4:00

Great. Since the ACS2023 dataset is already loaded, let's proceed with merging it.

```
#Merge with ACS2023
df_agg = df_agg.merge(dfacs, on="zip_code", how="left")

# Convert column names to lowercase and replace spaces with underscores
df_agg.columns = df_agg.columns.str.lower().str.replace(" ", "_")

print(df_agg.head(2))
```

```
         date zip_code  catch_basin_clogged/flooding_(use_comments)_(sc)  \
0  2024-01-01    10314                                                1
1  2024-01-01    10314                                                1

   street_flooding_(sj) time(h:m)  value  percentriseslope  elevation  \
0                     0      0:00    NaN               2.8  89.531181
1                     0      1:00    0.0               2.8  89.531181

   catchpersemi  impervious  total_population  median_household_income  \
0         228.3   62.273118           94355.0                 103436.0
1         228.3   62.273118           94355.0                 103436.0

   white_population  black_population  asian_population  \
0           64293.0            3326.0           16998.0
1           64293.0            3326.0           16998.0

   bachelor's_degree_holders  graduate_degree_holders  labor_force  \
0                    15792.0                    684.0      45249.0
1                    15792.0                    684.0      45249.0

   unemployed  median_home_value
0      2214.0           649200.0
1      2214.0           649200.0
```

2.4.4 Part iv

For each day, create two variables representing 1-day lag of the precipitation and the number of CB complaints.

```
# Convert date column to datetime format
df_agg["date"] = pd.to_datetime(df_agg["date"])

# Sort the data to ensure proper lagging
df_agg = df_agg.sort_values(by=["zip_code", "date", "time(h:m)"])
```

```
# Create lagged variables
df_agg["lag_1d_precipitation"] = df_agg.groupby("zip_code")["value"].shift(1)
df_agg["lag_1d_cb_complaints"] = df_agg.groupby("zip_code")["catch_basin_clogged/flooding_(use_co

# Display the modified DataFrame
print(df_agg.head(2))
```

```
           date zip_code  catch_basin_clogged/flooding_(use_comments)_(sc)  \
4771 2024-01-10    10001                                                 1
4781 2024-01-10    10001                                                 1

      street_flooding_(sj) time(h:m)  value  percentriseslope  elevation  \
4771                     0      0:00   0.27              0.88  26.409606
4781                     0     10:00   0.00              0.88  26.409606

      catchpersemi  impervious  ...  white_population  black_population  \
4771         748.9   96.980155  ...           15428.0            2355.0
4781         748.9   96.980155  ...           15428.0            2355.0

      asian_population  bachelor's_degree_holders  graduate_degree_holders  \
4771            5031.0                     8911.0                    561.0
4781            5031.0                     8911.0                    561.0

      labor_force  unemployed  median_home_value  lag_1d_precipitation  \
4771      17746.0       761.0           535100.0                   NaN
4781      17746.0       761.0           535100.0                  0.27

      lag_1d_cb_complaints
4771                   NaN
4781                   1.0

[2 rows x 22 columns]
```

The new columns lag_1d_precipitation and lag_1d_cb_complaints store the previous day's values for each zip code.

2.4.5 Part v

Filter data from March 1 to November 30, excluding winter months when flooding is less frequent. November 30.

```
# Filter data for dates between March 1 and November 30
df_agg = df_agg[
    (df_agg['date'].dt.month >= 3) & (df_agg['date'].dt.month <= 11)
```

```
]

# Check the unique months to ensure filtering worked
unique_months = df_agg["date"].dt.month.unique()
print("Filtered months:", unique_months)
```

```
Filtered months: [ 3  4  5  6  7  8  9 11 10]
```

It seens the filter have worked.

2.4.6 Part vi

Compare a Poisson regression with a Negative Binomial regression to account for overdispersion. Which model fits better? Explain the results to a New Yorker.

```
import statsmodels.api as sm
import statsmodels.formula.api as smf
from sklearn.preprocessing import StandardScaler

# Rename the dependent variable for easier formula use
df_agg = df_agg.rename(
    columns={"catch_basin_clogged/flooding_(use_comments)_(sc)": "cb_complaints"}
)

# Update the dependent variable name
dependent_var = "cb_complaints"

# Calculate Education Index
df_agg['education_index'] = (df_agg['bachelor's_degree_holders'] + df_agg['graduate_degree_holder

# Calculate Unemployment Rate
df_agg['unemployment_rate'] = df_agg['unemployed'] / df_agg['labor_force']

# Define independent variables
independent_vars = [
    'value',  # Precipitation
    'lag_1d_precipitation',  # Previous day's precipitation
    'lag_1d_cb_complaints',  # Previous day's complaints
    'education_index',  # Education level in the area
    'unemployment_rate',  # Unemployment rate
    'percentriseslope',  # Slope percentage
    'elevation',  # Elevation of the area
    'impervious',  # Percentage of impervious surfaces
]
```

```
# Standardize numerical features for better model performance
scaler = StandardScaler()
df_agg[independent_vars] = scaler.fit_transform(df_agg[independent_vars])

# Create formula for regression
formula = f"{dependent_var} ~ " + " + ".join(independent_vars)

# Fit a Poisson regression model
poisson_model = smf.glm(formula, data=df_agg,family=sm.families.Poisson()).fit()

# Fit a Negative Binomial regression model
nb_model = smf.glm(formula, data=df_agg,family=sm.families.NegativeBinomial()).fit()


# Compare model fit using Log-Likelihood and AIC
poisson_ll = poisson_model.llf  # Log-likelihood for Poisson
nb_ll = nb_model.llf  # Log-likelihood for Negative Binomial
poisson_aic = poisson_model.aic  # AIC for Poisson
nb_aic = nb_model.aic  # AIC for Negative Binomial

# Overdispersion check: Compute Pearson Chi-square statistic
poisson_deviance = poisson_model.pearson_chi2 / poisson_model.df_resid
nb_deviance = nb_model.pearson_chi2 / nb_model.df_resid

# Create a summary table comparing models
model_comparison = pd.DataFrame({
    "Model": ["Poisson Regression", "Negative Binomial Regression"],
    "Log-Likelihood": [poisson_ll, nb_ll],
    "AIC (Lower is Better)": [poisson_aic, nb_aic],
    "Deviance":[poisson_deviance,nb_deviance]
})

model_comparison
```

```
C:\Users\Ammar\AppData\Local\Programs\Python\Python312\Lib\site-packages\statsmodels\genmod\famil

Negative binomial dispersion parameter alpha not set. Using default value alpha=1.0.
```

	Model	Log-Likelihood	AIC (Lower is Better)	Deviance
0	Poisson Regression	-135438.323248	270894.646496	0.374601
1	Negative Binomial Regression	-157553.776817	315125.553635	0.116698

Negative Binomial Summary:

```
nb_model.summary()
```

Dep. Variable:	cb_complaints	**No. Observations:**	126867
Model:	GLM	**Df Residuals:**	126858
Model Family:	NegativeBinomial	**Df Model:**	8
Link Function:	Log	**Scale:**	1.0000
Method:	IRLS	**Log-Likelihood:**	-1.5755e+05
Date:	Mon, 10 Mar 2025	**Deviance:**	26195.
Time:	21:06:07	**Pearson chi2:**	1.48e+04
No. Iterations:	32	**Pseudo R-squ. (CS):**	0.2052
Covariance Type:	nonrobust		

	coef	**std err**	**z**	**P> |z|**	**[0.025**	**0.975]**
Intercept	-0.2600	0.004	-59.435	0.000	-0.269	-0.251
value	-0.0105	0.005	-2.157	0.031	-0.020	-0.001
lag_1d_precipitation	-0.0122	0.005	-2.496	0.013	-0.022	-0.003
lag_1d_cb_complaints	0.7793	0.003	229.561	0.000	0.773	0.786
education_index	-0.0105	0.005	-1.942	0.052	-0.021	9.58e-05
unemployment_rate	0.0046	0.005	0.863	0.388	-0.006	0.015
percentriseslope	-0.0058	0.007	-0.877	0.380	-0.019	0.007
elevation	0.0248	0.006	3.843	0.000	0.012	0.038
impervious	0.0144	0.005	2.963	0.003	0.005	0.024

Poisson Summary:

```
poisson_model.summary()
```

Dep. Variable:	cb_complaints	**No. Observations:**	126867
Model:	GLM	**Df Residuals:**	126858
Model Family:	Poisson	**Df Model:**	8
Link Function:	Log	**Scale:**	1.0000
Method:	IRLS	**Log-Likelihood:**	-1.3544e+05
Date:	Mon, 10 Mar 2025	**Deviance:**	65639.
Time:	21:06:07	**Pearson chi2:**	4.75e+04
No. Iterations:	8	**Pseudo R-squ. (CS):**	0.2108
Covariance Type:	nonrobust		

	coef	std err	z	P> |z|	[0.025	0.975]
Intercept	-0.0900	0.003	-30.596	0.000	-0.096	-0.084
value	0.0183	0.003	6.414	0.000	0.013	0.024
lag_1d_precipitation	0.0143	0.003	4.939	0.000	0.009	0.020
lag_1d_cb_complaints	0.1817	0.001	264.731	0.000	0.180	0.183
education_index	-0.0205	0.004	-5.441	0.000	-0.028	-0.013
unemployment_rate	0.0264	0.004	7.377	0.000	0.019	0.033
percentriseslope	-0.0489	0.005	-10.565	0.000	-0.058	-0.040
elevation	0.0441	0.004	10.028	0.000	0.036	0.053
impervious	-0.0469	0.003	-14.194	0.000	-0.053	-0.040

We compared two models to predict how many Catch Basin complaints NYC receives daily per ZIP code: Poisson Regression and Negative Binomial Regression. The Poisson model assumes complaints happen at a steady, predictable rate, while the Negative Binomial model accounts for unpredictable complaint spikes.

- **Log-Likelihood (Higher is Better):** This tells us how well the model fits real complaint data. A higher value means a better fit.
  - Poisson Regression: -135,438
  - Negative Binomial Regression: -157,553 (worse here, meaning it doesn't fit the data as well)
- **AIC (Lower is Better):** This measures how accurate the model is while avoiding unnecessary complexity.
  - Poisson Regression: 270,894
  - Negative Binomial Regression: 315,125 (higher, meaning it is more complex but not necessarily better in this case)
- **Deviance (Lower is Better):** Deviance measures how much the model's predictions differ from the data. A lower deviance means the model captures the variation in complaints more accurately.
  - Poisson Regression: 0.3746
  - Negative Binomial Regression: 0.1167 (much lower, meaning a better fit)

The Negative Binomial model has a much lower deviance, meaning it captures fluctuations in complaints better. This is crucial because flooding complaints don't happen at a constant rate, they vary based on storms, infrastructure, and city response time.

However, the Poisson model's higher deviance suggests it assumes too much consistency, making it struggle with spikes in complaints. This is called overdispersion—when the variability in data is greater than what a Poisson

model expects. Since the Negative Binomial model naturally accounts for this extra variability, it is better suited for predicting complaint patterns in NYC.

In conclusion, although the Poisson model has a better log-likelihood and AIC score, its high deviance suggests it underestimates the variation in complaint numbers. The Negative Binomial model, with its lower deviance, better handles unpredictable surges in complaints, making it the more reliable choice for real world predictions.

2.5 Conclusion

This analysis explored NYC flood-related complaints, focusing on response times, complaint patterns, and factors influencing delays. We found that Street Flooding complaints were resolved significantly faster than Catch Basin complaints, suggesting that clogged drainage infrastructure may require more resources or inspections before resolution. Response times also varied by borough, with Brooklyn having the quickest resolutions and Staten Island showing no significant difference from the Bronx. Additionally, complaints filed on weekends and holidays were slightly more likely to be resolved faster, possibly due to lower complaint volume or emergency response prioritization.

A predictive model for delaysconfirmed that SF complaints had lower odds of remaining unresolved for over three days, while socioeconomic factors like population density, unemployment rate, and housing affordability played a role in complaint patterns. However, the logistic model struggled with class imbalance, limiting predictive power. A Negative Binomial model provided a better fit than Poisson regression due to overdispersion in daily CB complaint counts, reinforcing the need for models that accommodate count variability.

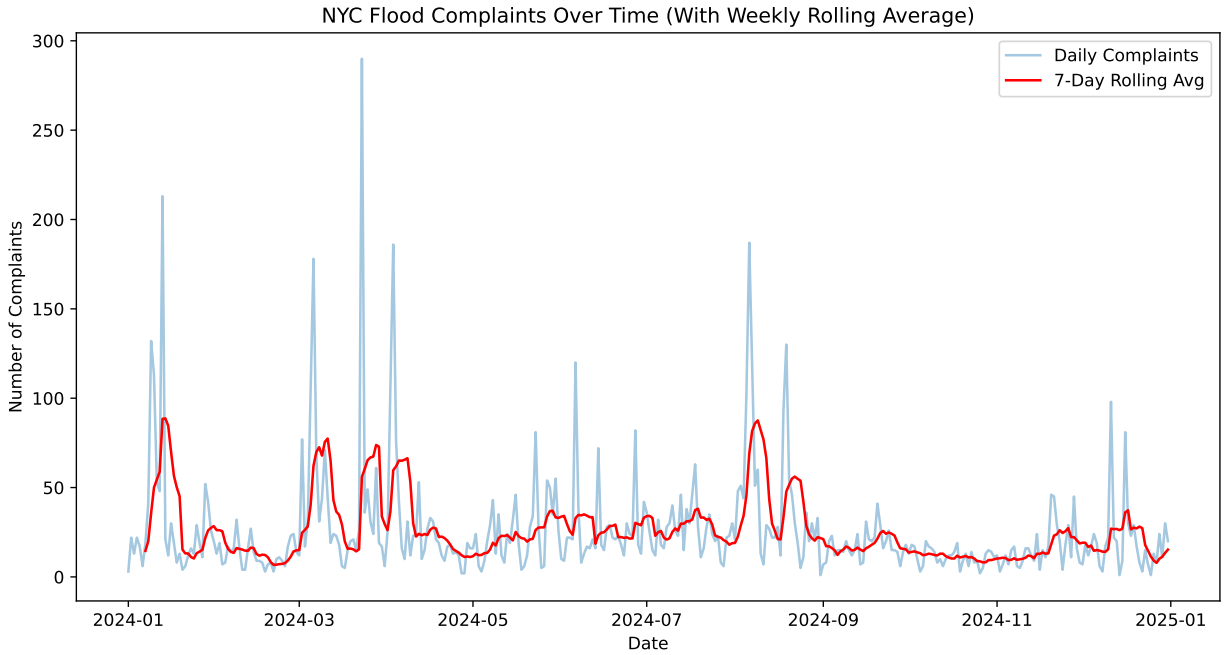
3 Further Exploration

I've decided to visualize additional variables and further explore the dataset to identify any potential trends.

3.1 Rolling Average of Complaints Per Week

```
# Aggregate complaint counts per day
df_daily_counts = df.groupby(df['created_date'].dt.date).size().reset_index(name='total_complaint

# Convert back to datetime for plotting
df_daily_counts['created_date'] = pd.to_datetime(df_daily_counts['created_date'])
```

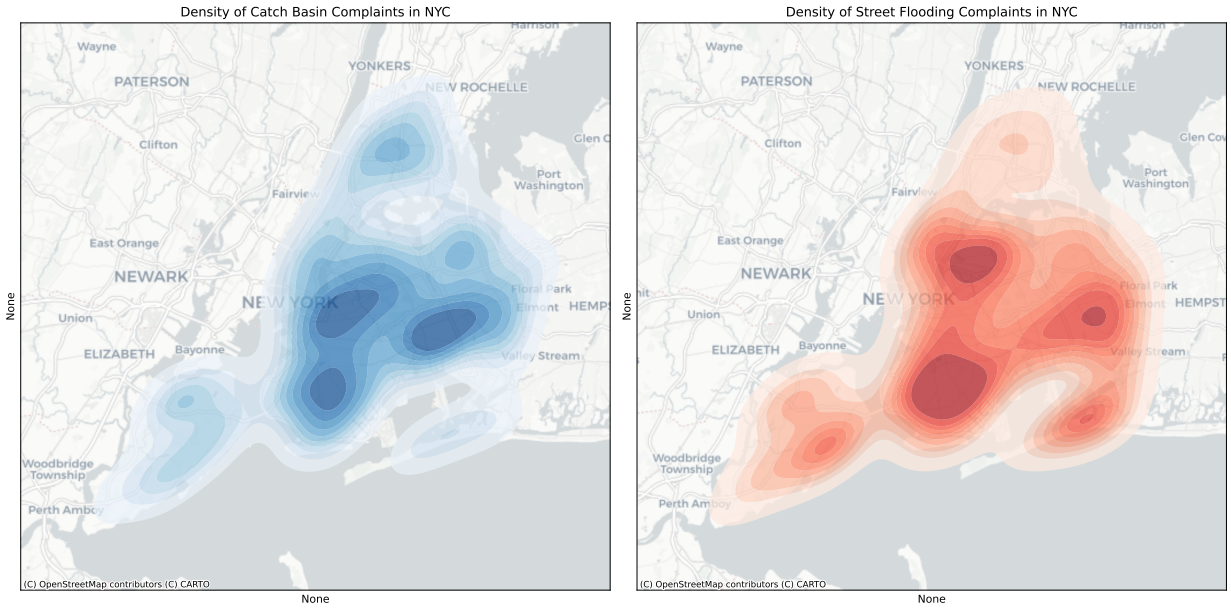
```
# Calculate a 7-day rolling average
df_daily_counts['rolling_avg'] = df_daily_counts['total_complaints'].rolling(window=7).mean()

# Plot rolling average of complaints per week
plt.figure(figsize=(12, 6))
plt.plot(df_daily_counts['created_date'], df_daily_counts['total_complaints'], alpha=0.4, label='
plt.plot(df_daily_counts['created_date'], df_daily_counts['rolling_avg'], color='red', label='7-D

plt.xlabel("Date")
plt.ylabel("Number of Complaints")
plt.title("NYC Flood Complaints Over Time (With Weekly Rolling Average)")
plt.legend()
plt.show()
```



I wanted to analyze the rolling average of complaints per week to smooth out daily fluctuations and identify seasonal trends in flooding related issues. This helps reveal whether complaint volumes increase during certain times of the year, such as heavy rainfall seasons, and allows for a better understanding of longterm patterns rather than just day to day variations. ## Density of Catch Basin and Street Flooding

```
import geopandas as gpd
import matplotlib.pyplot as plt
import contextily as ctx
import seaborn as sns

# Ensure latitude and longitude exist and drop missing values
```

```
df_filtered = df.dropna(subset=['latitude', 'longitude', 'descriptor'])

# Convert DataFrame to GeoDataFrame
gdf = gpd.GeoDataFrame(df_filtered,
                       geometry=gpd.points_from_xy(df_filtered.longitude, df_filtered.latitude),
                       crs="EPSG:4326")

gdf = gdf.to_crs(epsg=3857)

# Split into two GeoDataFrames for CB and SF complaints
gdf_cb = gdf[gdf['descriptor'].str.contains("Catch Basin", na=False)]
gdf_sf = gdf[gdf['descriptor'].str.contains("Street Flooding", na=False)]

# Plot Heatmaps for CB and SF Complaints
fig, ax = plt.subplots(1, 2, figsize=(16, 8))

# Plot CB Complaints Density
sns.kdeplot(x=gdf_cb.geometry.x, y=gdf_cb.geometry.y, cmap="Blues", fill=True, ax=ax[0], alpha=0.
ctx.add_basemap(ax[0], source=ctx.providers.CartoDB.Positron)
ax[0].set_title("Density of Catch Basin Complaints in NYC")
ax[0].set_xticks([])
ax[0].set_yticks([])

# Plot SF Complaints Density
sns.kdeplot(x=gdf_sf.geometry.x, y=gdf_sf.geometry.y, cmap="Reds", fill=True, ax=ax[1], alpha=0.7
ctx.add_basemap(ax[1], source=ctx.providers.CartoDB.Positron)
ax[1].set_title("Density of Street Flooding Complaints in NYC")
ax[1].set_xticks([])
ax[1].set_yticks([])

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



I wanted to visualize the density of Catch Basin and Street Flooding complaints to understand where flooding-related issues are most concentrated across NYC. By mapping these complaints separately, I could identify problem hotspots and observe whether CB blockages correlate with areas experiencing frequent street flooding.