Problem Set 6 - Waze Shiny Dashboard

Peter Ganong, Maggie Shi, and Andre Oviedo

2024-11-23

1. **ps6:** Due Sat 23rd at 5:00PM Central. Worth 100 points (80 points from questions, 10 points for correct submission and 10 points for code style) + 10 extra credit.

We use (*) to indicate a problem that we think might be time consuming.

Steps to submit (10 points on PS6)

1. "This submission is my work alone and complies with the 30538 integrity policy." Add your initials to indicate your agreement: **SH**
2. "I have uploaded the names of anyone I worked with on the problem set **here**" **SH** (2 point)
3. Late coins used this pset: **00** Late coins left after submission: **04**
4. Before starting the problem set, make sure to read and agree to the terms of data usage for the Waze data here.
5. Knit your `ps6.qmd` as a pdf document and name it `ps6.pdf`.
6. Push your `ps6.qmd`, `ps6.pdf`, `requirements.txt`, and all created folders (we will create three Shiny apps so you will have at least three additional folders) to your Github repo (5 points). It is fine to use Github Desktop.
7. Submit `ps6.pdf` and also link your Github repo via Gradescope (5 points)
8. Tag your submission in Gradescope. For the Code Style part (10 points) please tag the whole correspondingsection for the code style rubric.

Notes: see the Quarto documentation (link) for directions on inserting images into your knitted document.

IMPORTANT: For the App portion of the PS, in case you can not arrive to the expected functional dashboard we will need to take a look at your `app.py` file. You can use the following

code chunk template to "import" and print the content of that file. Please, don"t forget to also tag the corresponding code chunk as part of your submission!

Background

Data Download and Exploration (20 points)

1. Using the zipfile package ...

```
# Load the csv file into a pandas DataFrame
waze_data_sample = pd.read_csv("waze_data_sample.csv")

# Show summary
print("DataFrame Summary:")
print(waze_data_sample.info())

# Show first 5 rows
print("\nFirst 5 rows of the DataFrame:")
print(waze_data_sample.head())

# Check suspicious columns
print("\nUnique values in subtype:")
print(sorted(waze_data_sample["subtype"].dropna().unique()))
print("\nUnique values in confidence:")
print(sorted(waze_data_sample["confidence"].unique()))
print("\nUnique values in reliability:")
print(sorted(waze_data_sample["reliability"].unique()))
print("\nUnique values in reportRating:")
print(sorted(waze_data_sample["reportRating"].unique()))
```

```
DataFrame Summary:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7781 entries, 0 to 7780
Data columns (total 16 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   Unnamed: 0    7781 non-null   int64
 1   city          7781 non-null   object
 2   confidence    7781 non-null   int64
 3   nThumbsUp     10 non-null     float64
 4   street        7630 non-null   object
```

```
 5   uuid          7781 non-null   object
 6   country       7781 non-null   object
 7   type          7781 non-null   object
 8   subtype       6777 non-null   object
 9   roadType      7781 non-null   int64
 10  reliability   7781 non-null   int64
 11  magvar        7781 non-null   int64
 12  reportRating  7781 non-null   int64
 13  ts            7781 non-null   object
 14  geo           7781 non-null   object
 15  geoWKT        7781 non-null   object
dtypes: float64(1), int64(6), object(9)
memory usage: 972.8+ KB
None

First 5 rows of the DataFrame:
   Unnamed: 0         city  confidence  nThumbsUp        street  \
0      584358  Chicago, IL           0        NaN           NaN
1      472915  Chicago, IL           0        NaN        I-90 E
2      550891  Chicago, IL           0        NaN        I-90 W
3      770659  Chicago, IL           0        NaN           NaN
4      381054  Chicago, IL           0        NaN  N Pulaski Rd

                                   uuid country         type  \
0  c9b88a12-79e8-44cb-aadd-a75855fc4bcb      US          JAM
1  7c634c0a-099c-4262-b57f-e893bdebce73      US  ROAD_CLOSED
2  7aa3c61a-f8dc-4fe8-bbb0-db6b9e0dc53b      US       HAZARD
3  3b95dd2f-647c-46de-b4e1-8ebc73aa9221      US       HAZARD
4  13a5e230-a28a-4bf4-b928-bc1dd38850e0      US          JAM

                         subtype  roadType  reliability  magvar  \
0                            NaN        17            5     116
1              ROAD_CLOSED_EVENT         3            6     173
2  HAZARD_ON_SHOULDER_CAR_STOPPED         3            5     308
3                 HAZARD_ON_ROAD        20            5     155
4              JAM_HEAVY_TRAFFIC         7            5     178

   reportRating                        ts                          geo  \
0             5  2024-07-02 18:27:40 UTC   POINT(-87.64577 41.892743)
1             0  2024-06-16 10:13:19 UTC  POINT(-87.646359 41.886295)
2             5  2024-05-02 19:01:47 UTC   POINT(-87.695982 41.93272)
3             2  2024-03-25 18:53:24 UTC  POINT(-87.669253 41.904497)
4             2  2024-06-03 21:17:33 UTC  POINT(-87.728322 41.978769)
```

```
                        geoWKT
0   Point(-87.64577 41.892743)
1  Point(-87.646359 41.886295)
2   Point(-87.695982 41.93272)
3  Point(-87.669253 41.904497)
4  Point(-87.728322 41.978769)

Unique values in subtype:
['ACCIDENT_MAJOR', 'ACCIDENT_MINOR', 'HAZARD_ON_ROAD',
'HAZARD_ON_ROAD_CAR_STOPPED', 'HAZARD_ON_ROAD_CONSTRUCTION',
'HAZARD_ON_ROAD_EMERGENCY_VEHICLE', 'HAZARD_ON_ROAD_LANE_CLOSED',
'HAZARD_ON_ROAD_OBJECT', 'HAZARD_ON_ROAD_POT_HOLE',
'HAZARD_ON_ROAD_ROAD_KILL', 'HAZARD_ON_ROAD_TRAFFIC_LIGHT_FAULT',
'HAZARD_ON_SHOULDER_CAR_STOPPED', 'HAZARD_WEATHER', 'HAZARD_WEATHER_FLOOD',
'HAZARD_WEATHER_FOG', 'HAZARD_WEATHER_HEAVY_SNOW', 'JAM_HEAVY_TRAFFIC',
'JAM_MODERATE_TRAFFIC', 'JAM_STAND_STILL_TRAFFIC', 'ROAD_CLOSED_EVENT']

Unique values in confidence:
[np.int64(0), np.int64(1), np.int64(2), np.int64(3), np.int64(4),
np.int64(5)]

Unique values in reliability:
[np.int64(5), np.int64(6), np.int64(7), np.int64(8), np.int64(9),
np.int64(10)]

Unique values in reportRating:
[np.int64(0), np.int64(1), np.int64(2), np.int64(3), np.int64(4),
np.int64(5)]
```

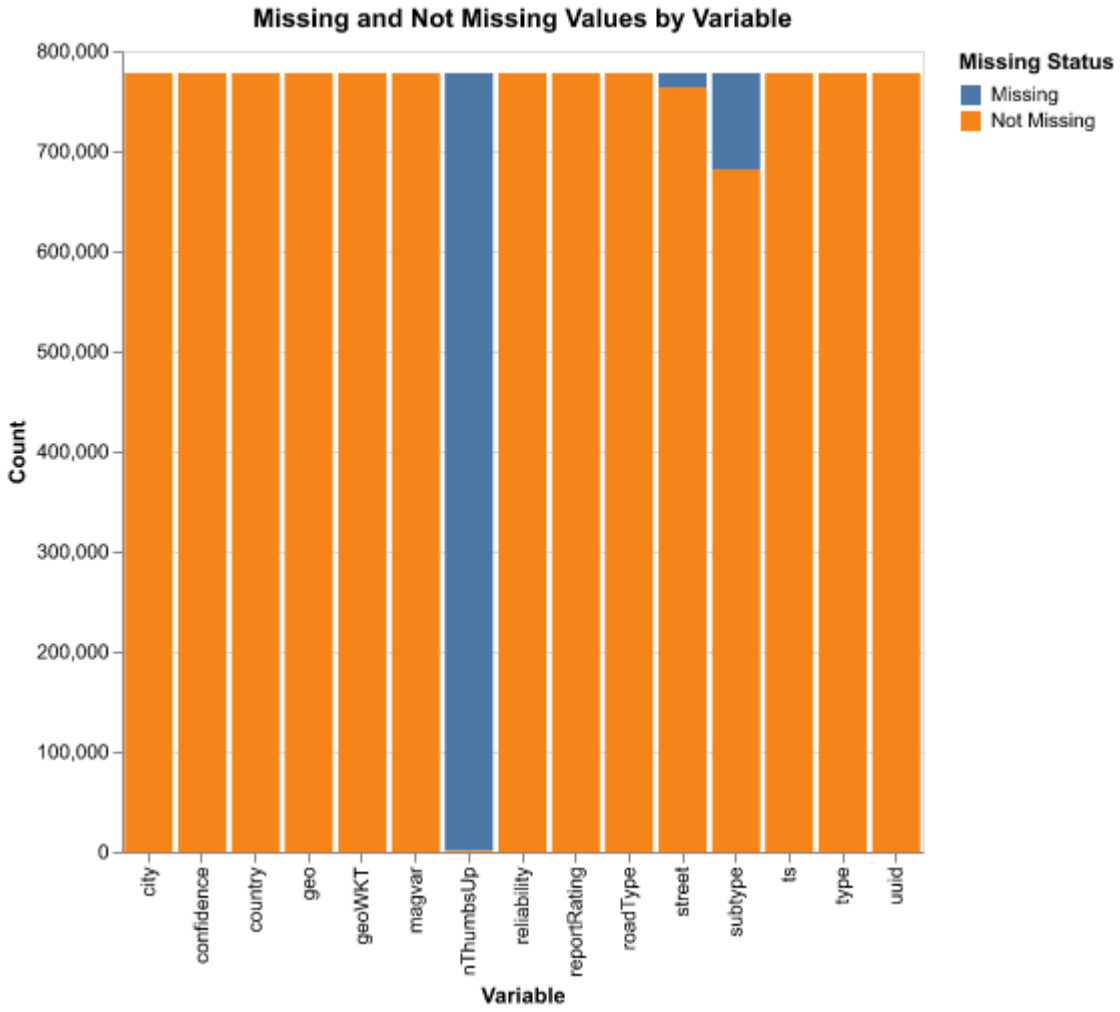
Variable Name	Altair Data Type	Rationale
city	Nominal	Categorical data without order
confidence	Ordinal	Discrete values from 0 to 5 suggest an ordered metric, but cannot be compared using ratios.
nThumbsUp	Quantitative	The count is inherently numeric
street	Nominal	Categorical data without order

Variable Name	Altair Data Type	Rationale
uuid	Nominal	Unique identifier without inherent order
country	Nominal	Categorical data without order
type	Nominal	Categorical data without order
subtype	Nominal (but Ordinal in specific cases)	Primarily categorical, but in cases like "ACCIDENT" or "JAM," there could be an inherent order (i.e., Major/Minor Accident and Light/Moderate/Heavy/Standstill Jam)
roadType	Nominal	Categories of roads without a hierarchical order
reliability	Ordinal	Discrete values from 4 to 10 could indicate an ordered ranking, but cannot be compared using ratios.
magvar	Quantitative	Depicts direction in degrees (0-359). Even though it is not comparable using ratios due to its circular nature, it is still measurable on a continuous numerical (circular) scale.
reportRating	Ordinal	Discrete user rank from 0 to 5, suggesting an ordinal quality due to ranking.

2. Now load the waze_data.csv …

```
# Load the csv file into a pandas DataFrame
waze_data = pd.read_csv("waze_data.csv")

# Calculate missing and non-missing values for each column
missing_data = pd.DataFrame({
    "Variable": waze_data.columns,
    "Missing": waze_data.isnull().sum(),
    "Not Missing": waze_data.notnull().sum()
```

```
})

# Melt the DataFrame to have "Missing" and "Not Missing" as values in a
 ↪  single column
missing_data = missing_data.melt(id_vars="Variable",
                                 var_name="Status",
                                 value_name="Count")

# Stacked bar chart
alt.Chart(missing_data).mark_bar().encode(
    x=alt.X("Variable", title="Variable"),
    y=alt.Y("Count", title="Count"),
    color=alt.Color("Status", legend=alt.Legend(title="Missing Status")),
    tooltip=["Variable", "Status", "Count"]
).properties(
    title="Missing and Not Missing Values by Variable",
    width=400,
    height=400
)
```




```
# Sanity check
print(missing_data[(missing_data["Status"] == "Missing") &
 ↪  (missing_data["Count"] > 0)])
```

```
    Variable   Status   Count
2  nThumbsUp  Missing  776723
3     street  Missing   14073
7    subtype  Missing   96086
```

- Variables with NULL values: `nThumbsUp`, `street`, `subtype`
- Variable with the highest share of missing observations: `nThumbsUp`

3. Take a look at the variables type and subtype. …

a. Print the unique values for the columns type and subtype …

```
# Print unique values for "type" and "subtype" columns
print("Type unique values:\n", sorted(waze_data["type"].unique()))
print("\nSubtype unique values:\n",
 ↪  sorted(waze_data["subtype"].dropna().unique()))
```

```
Type unique values:
 ['ACCIDENT', 'HAZARD', 'JAM', 'ROAD_CLOSED']

Subtype unique values:
 ['ACCIDENT_MAJOR', 'ACCIDENT_MINOR', 'HAZARD_ON_ROAD',
 'HAZARD_ON_ROAD_CAR_STOPPED', 'HAZARD_ON_ROAD_CONSTRUCTION',
 'HAZARD_ON_ROAD_EMERGENCY_VEHICLE', 'HAZARD_ON_ROAD_ICE',
 'HAZARD_ON_ROAD_LANE_CLOSED', 'HAZARD_ON_ROAD_OBJECT',
 'HAZARD_ON_ROAD_POT_HOLE', 'HAZARD_ON_ROAD_ROAD_KILL',
 'HAZARD_ON_ROAD_TRAFFIC_LIGHT_FAULT', 'HAZARD_ON_SHOULDER',
 'HAZARD_ON_SHOULDER_ANIMALS', 'HAZARD_ON_SHOULDER_CAR_STOPPED',
 'HAZARD_ON_SHOULDER_MISSING_SIGN', 'HAZARD_WEATHER', 'HAZARD_WEATHER_FLOOD',
 'HAZARD_WEATHER_FOG', 'HAZARD_WEATHER_HAIL', 'HAZARD_WEATHER_HEAVY_SNOW',
 'JAM_HEAVY_TRAFFIC', 'JAM_LIGHT_TRAFFIC', 'JAM_MODERATE_TRAFFIC',
 'JAM_STAND_STILL_TRAFFIC', 'ROAD_CLOSED_CONSTRUCTION', 'ROAD_CLOSED_EVENT',
 'ROAD_CLOSED_HAZARD']
```

```
# Types with NA in subtype
print(waze_data[waze_data["subtype"].isna()]["type"].unique())
```

```
['JAM' 'ACCIDENT' 'ROAD_CLOSED' 'HAZARD']
```

- All four type values have NA in their subtype.

```
# Identify types with sub-subtypes
print(f"JAM's subtypes:
 ↪  {waze_data[waze_data["type"]=="JAM"]["subtype"].sort_values().unique()}")
print(f"ACCIDENT's subtypes:
 ↪  {waze_data[waze_data["type"]=="ACCIDENT"]["subtype"].sort_values().unique()}")
print(f"ROAD_CLOSED's subtypes:
 ↪  {waze_data[waze_data["type"]=="ROAD_CLOSED"]["subtype"].sort_values().unique()}")
print(f"HAZARD's subtypes:
 ↪  {waze_data[waze_data["type"]=="HAZARD"]["subtype"].sort_values().unique()}")
```

```
JAM's subtypes: ['JAM_HEAVY_TRAFFIC' 'JAM_LIGHT_TRAFFIC'
'JAM_MODERATE_TRAFFIC'
 'JAM_STAND_STILL_TRAFFIC' nan]
ACCIDENT's subtypes: ['ACCIDENT_MAJOR' 'ACCIDENT_MINOR' nan]
ROAD_CLOSED's subtypes: ['ROAD_CLOSED_CONSTRUCTION' 'ROAD_CLOSED_EVENT'
'ROAD_CLOSED_HAZARD' nan]
HAZARD's subtypes: ['HAZARD_ON_ROAD' 'HAZARD_ON_ROAD_CAR_STOPPED'
 'HAZARD_ON_ROAD_CONSTRUCTION' 'HAZARD_ON_ROAD_EMERGENCY_VEHICLE'
 'HAZARD_ON_ROAD_ICE' 'HAZARD_ON_ROAD_LANE_CLOSED' 'HAZARD_ON_ROAD_OBJECT'
 'HAZARD_ON_ROAD_POT_HOLE' 'HAZARD_ON_ROAD_ROAD_KILL'
 'HAZARD_ON_ROAD_TRAFFIC_LIGHT_FAULT' 'HAZARD_ON_SHOULDER'
 'HAZARD_ON_SHOULDER_ANIMALS' 'HAZARD_ON_SHOULDER_CAR_STOPPED'
 'HAZARD_ON_SHOULDER_MISSING_SIGN' 'HAZARD_WEATHER' 'HAZARD_WEATHER_FLOOD'
 'HAZARD_WEATHER_FOG' 'HAZARD_WEATHER_HAIL' 'HAZARD_WEATHER_HEAVY_SNOW'
 nan]
```

- JAM is divided into light, moderate, heavy, and stand still traffic – no more subdivision of each subtype.
- ACCIDENT is divided into major and minor – no more subdivision of each subtype.
- ROAD_CLOSED is divided into event, construction, and hazard – no more subdivision of each subtype.
- HAZARD divided into on road, on shoulder, and weather – those three subtypes may branch to more specific sub-subtypes such as on road car stopped, on shoulder animals, weather hail, and more.
- **Thus, the only type that has sub-subtypes is HAZARD.**

b. Write out a bulleted listed with the values at each layer given this hierarchy …

- **Jam**
  - Light Traffic
  - Moderate Traffic
  - Heavy Traffic
  - Standstill Traffic
  - Unclassified
- **Accident**
  - Minor
  - Major
  - Unclassified
- **Road Closed**
  - Event
  - Construction

  - Hazard
  - Unclassified

- **Hazard**
  - On Road
    * Car Stopped
    * Construction
    * Emergency Vehicle
    * Ice
    * Object
    * Pothole
    * Traffic Light Fault
    * Lane Closed
    * Road Kill
    * Unclassified
  - On Shoulder
    * Car Stopped
    * Animals
    * Missing Sign
    * Unclassified
  - Weather
    * Flood
    * Fog
    * Heavy Snow
    * Hail
    * Unclassified
  - Unclassified

c. Finally, do you consider that we should keep the NA subtypes? …

```
print(f"Number of records with NA subtypes:
 ↪  {len(waze_data[waze_data["subtype"].isna()])}")
```

```
Number of records with NA subtypes: 96086
```

We should keep the records with NA values in their subtype because they represent a significant portion of the data (96,086 rows). By coding these as "Unclassified," we retain the data for analysis while indicating that specific subtype information is missing. Assuming that these "Unclassified" entries still belong to their respective type categories, we can maintain a complete dataset even if they lack detailed subtype classification.

4. We want to assign this newly created hierarchy …

5. a. To create a crosswalk …

```
# Define a pandas df which has 5 columns
data = []
crosswalk_df = pd.DataFrame(data,
 ↪  columns=["type","subtype","updated_type","updated_subtype","updated_subsubtype"])
```

2. b. Let each row of this DataFrame be a unique …

```
# Extract unique combinations of type and subtype
unique_combos = waze_data[["type", "subtype"]].drop_duplicates()

# Create new column "subtype1" based on condition
unique_combos["subtype1"] = np.where(unique_combos["subtype"].isna(), "",
 ↪  unique_combos["subtype"])

unique_combos["combo"] = unique_combos["type"] + unique_combos["subtype1"]

# Bullet list into list in list (with combo as primary key)
bullet_list = [
    ["JAM", "Jam", "Unclassified"],
    ["ROAD_CLOSED", "Road Closed", "Unclassified"],
    ["ACCIDENT", "Accident", "Unclassified"],
    ["HAZARD", "Hazard", "Unclassified"],
    ["ACCIDENTACCIDENT_MAJOR", "Accident", "Major"],
    ["ACCIDENTACCIDENT_MINOR", "Accident", "Minor"],
    ["HAZARDHAZARD_ON_ROAD", "Hazard", "On Road"],
    ["HAZARDHAZARD_ON_ROAD_CAR_STOPPED", "Hazard", "On Road", "Car Stopped"],
    ["HAZARDHAZARD_ON_ROAD_CONSTRUCTION", "Hazard", "On Road",
 ↪  "Construction"],
    ["HAZARDHAZARD_ON_ROAD_EMERGENCY_VEHICLE", "Hazard", "On Road",
 ↪  "Emergency Vehicle"],
    ["HAZARDHAZARD_ON_ROAD_ICE", "Hazard", "On Road", "Ice"],
    ["HAZARDHAZARD_ON_ROAD_OBJECT", "Hazard", "On Road", "Object"],
    ["HAZARDHAZARD_ON_ROAD_POT_HOLE", "Hazard", "On Road", "Pothole"],
    ["HAZARDHAZARD_ON_ROAD_TRAFFIC_LIGHT_FAULT", "Hazard", "On Road",
 ↪  "Traffic Light Fault"],
    ["HAZARDHAZARD_ON_ROAD_LANE_CLOSED", "Hazard", "On Road", "Lane Closed"],
    ["HAZARDHAZARD_ON_ROAD_ROAD_KILL", "Hazard", "On Road", "Road Kill"],
    ["HAZARDHAZARD_ON_ROAD_UNCLASSIFIED", "Hazard", "On Road",
 ↪  "Unclassified"],
    ["HAZARDHAZARD_ON_SHOULDER", "Hazard", "On Shoulder"],
```

```
    ["HAZARDHAZARD_ON_SHOULDER_CAR_STOPPED", "Hazard", "On Shoulder", "Car
 ↪  Stopped"],
    ["HAZARDHAZARD_ON_SHOULDER_ANIMALS", "Hazard", "On Shoulder", "Animals"],
    ["HAZARDHAZARD_ON_SHOULDER_MISSING_SIGN", "Hazard", "On Shoulder",
 ↪  "Missing Sign"],
    ["HAZARDHAZARD_ON_SHOULDER_UNCLASSIFIED", "Hazard", "On Shoulder",
 ↪  "Unclassified"],
    ["HAZARDHAZARD_WEATHER", "Hazard", "Weather"],
    ["HAZARDHAZARD_WEATHER_FLOOD", "Hazard", "Weather", "Flood"],
    ["HAZARDHAZARD_WEATHER_FOG", "Hazard", "Weather", "Fog"],
    ["HAZARDHAZARD_WEATHER_HEAVY_SNOW", "Hazard", "Weather", "Heavy Snow"],
    ["HAZARDHAZARD_WEATHER_HAIL", "Hazard", "Weather", "Hail"],
    ["HAZARDHAZARD_WEATHER_UNCLASSIFIED", "Hazard", "Weather",
 ↪  "Unclassified"],
    ["JAMJAM_HEAVY_TRAFFIC", "Jam", "Heavy Traffic"],
    ["JAMJAM_MODERATE_TRAFFIC", "Jam", "Moderate Traffic"],
    ["JAMJAM_STAND_STILL_TRAFFIC", "Jam", "Standstill Traffic"],
    ["JAMJAM_LIGHT_TRAFFIC", "Jam", "Light Traffic"],
    ["ROAD_CLOSEDROAD_CLOSED_EVENT", "Road Closed", "Event"],
    ["ROAD_CLOSEDROAD_CLOSED_CONSTRUCTION", "Road Closed", "Construction"],
    ["ROAD_CLOSEDROAD_CLOSED_HAZARD", "Road Closed", "Hazard"],
    ["ROAD_CLOSEDROAD_CLOSED_UNCLASSIFIED", "Road Closed", "Unclassified"]
]

# Convert bullet_list into a DataFrame
bullet_df = pd.DataFrame(bullet_list, columns=["combo", "updated_type",
 ↪  "updated_subtype", "updated_subsubtype"])

# Merge unique_combos and bullet_df on "combo"
merged_df = unique_combos.merge(bullet_df, on="combo", how="left")

# Populate crosswalk_df with the merged data
crosswalk_df["type"] = merged_df["type"]
crosswalk_df["subtype"] = merged_df["subtype"]
crosswalk_df["updated_type"] = merged_df["updated_type"]
crosswalk_df["updated_subtype"] = merged_df["updated_subtype"]
crosswalk_df["updated_subsubtype"] = merged_df["updated_subsubtype"]

print(crosswalk_df.head())
```

```
          type         subtype updated_type updated_subtype
          updated_subsubtype
```

```
0          JAM             NaN          Jam    Unclassified
None
1     ACCIDENT             NaN     Accident    Unclassified
None
2  ROAD_CLOSED             NaN  Road Closed    Unclassified
None
3       HAZARD             NaN       Hazard    Unclassified
None
4     ACCIDENT  ACCIDENT_MAJOR     Accident           Major
None
```

```
# Ensure there are 32 observations
print(f"Number of obserrvations: {len(crosswalk_df)}")
```

```
Number of obserrvations: 32
```

3. c. Merge the crosswalk with the original data using type and subtype …

```
# Merge crosswalk_df with waze_data
waze_data_merged = waze_data.merge(crosswalk_df, on=["type", "subtype"],
 ↪  how="inner")
```

```
# Print rows of waze_data with specific conditions Accident - Unclassified
print("Number of rows for Accident - Unclassified:",
 ↪  len(waze_data_merged[(waze_data_merged["updated_type"]=="Accident")&(waze_data_merged["u
```

```
Number of rows for Accident - Unclassified: 24359
```

There are 24,359 rows of data with `type`=`Accident` and `subtype`=`Unclassified` in the complete Waze dataset.

4. d. EXTRA CREDIT/OPTIONAL: After merging the crosswalk …

```
# EXTRA CREDIT
# Compare "type" column
crosswalk_df_types = crosswalk_df["type"].dropna().unique()
waze_data_merged_types = waze_data_merged["type"].dropna().unique()

# Create a DataFrame to store comparison results for "type"
type_comparison = pd.DataFrame({
    "crosswalk": crosswalk_df_types,
    "merged": waze_data_merged_types,
```

```
    "similarity": [cw == wm for cw, wm in zip(crosswalk_df_types,
     ↪  waze_data_merged_types)]
})

# Compare "subtype" column
crosswalk_df_subtypes = crosswalk_df["subtype"].dropna().unique()
waze_data_merged_subtypes = waze_data_merged["subtype"].dropna().unique()

# Create a DataFrame to store comparison results for "subtype"
subtype_comparison = pd.DataFrame({
    "crosswalk": crosswalk_df_subtypes,
    "merged": waze_data_merged_subtypes,
    "similarity": [cw == wm for cw, wm in zip(crosswalk_df_subtypes,
     ↪  waze_data_merged_subtypes)]
})

# Display the comparison tables
print("Type Comparison Table:")
print(type_comparison)

print("\nSubtype Comparison Table:")
print(subtype_comparison)
```

```
Type Comparison Table:
     crosswalk       merged  similarity
0          JAM          JAM        True
1     ACCIDENT     ACCIDENT        True
2  ROAD_CLOSED  ROAD_CLOSED        True
3       HAZARD       HAZARD        True

Subtype Comparison Table:
                            crosswalk                            merged  \
0                      ACCIDENT_MAJOR                      ACCIDENT_MAJOR
1                      ACCIDENT_MINOR                      ACCIDENT_MINOR
2                      HAZARD_ON_ROAD                      HAZARD_ON_ROAD
3           HAZARD_ON_ROAD_CAR_STOPPED          HAZARD_ON_ROAD_CAR_STOPPED
4          HAZARD_ON_ROAD_CONSTRUCTION         HAZARD_ON_ROAD_CONSTRUCTION
5     HAZARD_ON_ROAD_EMERGENCY_VEHICLE    HAZARD_ON_ROAD_EMERGENCY_VEHICLE
6                   HAZARD_ON_ROAD_ICE                  HAZARD_ON_ROAD_ICE
7               HAZARD_ON_ROAD_OBJECT              HAZARD_ON_ROAD_OBJECT
8             HAZARD_ON_ROAD_POT_HOLE             HAZARD_ON_ROAD_POT_HOLE
9   HAZARD_ON_ROAD_TRAFFIC_LIGHT_FAULT  HAZARD_ON_ROAD_TRAFFIC_LIGHT_FAULT
```

10	HAZARD_ON_SHOULDER	HAZARD_ON_SHOULDER
11	HAZARD_ON_SHOULDER_CAR_STOPPED	HAZARD_ON_SHOULDER_CAR_STOPPED
12	HAZARD_WEATHER	HAZARD_WEATHER
13	HAZARD_WEATHER_FLOOD	HAZARD_WEATHER_FLOOD
14	JAM_HEAVY_TRAFFIC	JAM_HEAVY_TRAFFIC
15	JAM_MODERATE_TRAFFIC	JAM_MODERATE_TRAFFIC
16	JAM_STAND_STILL_TRAFFIC	JAM_STAND_STILL_TRAFFIC
17	ROAD_CLOSED_EVENT	ROAD_CLOSED_EVENT
18	HAZARD_ON_ROAD_LANE_CLOSED	HAZARD_ON_ROAD_LANE_CLOSED
19	HAZARD_WEATHER_FOG	HAZARD_WEATHER_FOG
20	ROAD_CLOSED_CONSTRUCTION	ROAD_CLOSED_CONSTRUCTION
21	HAZARD_ON_ROAD_ROAD_KILL	HAZARD_ON_ROAD_ROAD_KILL
22	HAZARD_ON_SHOULDER_ANIMALS	HAZARD_ON_SHOULDER_ANIMALS
23	HAZARD_ON_SHOULDER_MISSING_SIGN	HAZARD_ON_SHOULDER_MISSING_SIGN
24	JAM_LIGHT_TRAFFIC	JAM_LIGHT_TRAFFIC
25	HAZARD_WEATHER_HEAVY_SNOW	HAZARD_WEATHER_HEAVY_SNOW
26	ROAD_CLOSED_HAZARD	ROAD_CLOSED_HAZARD
27	HAZARD_WEATHER_HAIL	HAZARD_WEATHER_HAIL

	similarity
0	True
1	True
2	True
3	True
4	True
5	True
6	True
7	True
8	True
9	True
10	True
11	True
12	True
13	True
14	True
15	True
16	True
17	True
18	True
19	True
20	True
21	True
22	True

```
23        True
24        True
25        True
26        True
27        True
```

All type values are similar in two dataset.

App #1: Top Location by Alert Type Dashboard (30 points)

1. Let's begin by by developing our output outside of Shiny …

a. The geo variable holds coordinates data …

```
import re

# Define txt (list of text to process)
txt = waze_data_merged["geo"]

# Function to split latitude and longitude
def split_coordinates(geo_str):
    parts = re.split(r"[( )]", geo_str)
    return parts[1], parts[2]

# Apply the splitting to each row
coordinates = [split_coordinates(geo_str) for geo_str in
 ↪  waze_data_merged["geo"]]

# Separate longitude and latitude explicitly
longitudes = [coord[0] for coord in coordinates]
latitudes = [coord[1] for coord in coordinates]

# Assign the separate lists to new columns in the DataFrame
waze_data_merged["longitude"] = longitudes
waze_data_merged["latitude"] = latitudes
```

GPT Prompt:
teach me re.split() text between two characters python
e.g.: POINT(-87.615862 41.887432)
get "-87.615862"

b. Bin the latitude and longitude variables into bins of step size 0.01 …

```
# Bin the latitude and longitude using rounding
waze_data_merged["binned_longitude"] =
 ↪  waze_data_merged["longitude"].astype(float).round(2)
waze_data_merged["binned_latitude"] =
 ↪  waze_data_merged["latitude"].astype(float).round(2)

# Count each binned latitude-longitude combination
combination_counts = waze_data_merged.groupby(["binned_latitude",
 ↪  "binned_longitude"]).size().reset_index(name="count")

# Show the combinations with the biggest count
print(combination_counts.sort_values(by="count", ascending=False).head(1))
```

```
     binned_latitude  binned_longitude  count
396            41.88            -87.65  21325
```

A binned latitude-longitude combination that has the greatest number of observations in the overall dataset is 41.88, -87.65 with 21,325 records.

c. Collapse the data down to the level of aggregation needed …

```
# Aggregate by latitude, longitude, type, and subtype
combination_counts2 = waze_data_merged.groupby(
    ["updated_type", "updated_subtype", "binned_latitude",
 ↪  "binned_longitude"]
).size().reset_index(name="count")

# Save to CSV using pandas
top_alerts_map = combination_counts2.copy()
top_alerts_map.to_csv("top_alerts_map/top_alerts_map.csv", index=False)
```

The level of aggregation is at binned_latitude, binned_longitude, type, and subtype. The data is grouped by these four fields to count alerts separately for each unique type-subtype combination within each latitude-longitude bin. We don't need to filter out the top 10 of each type-subtype yet, let the shiny app do that.

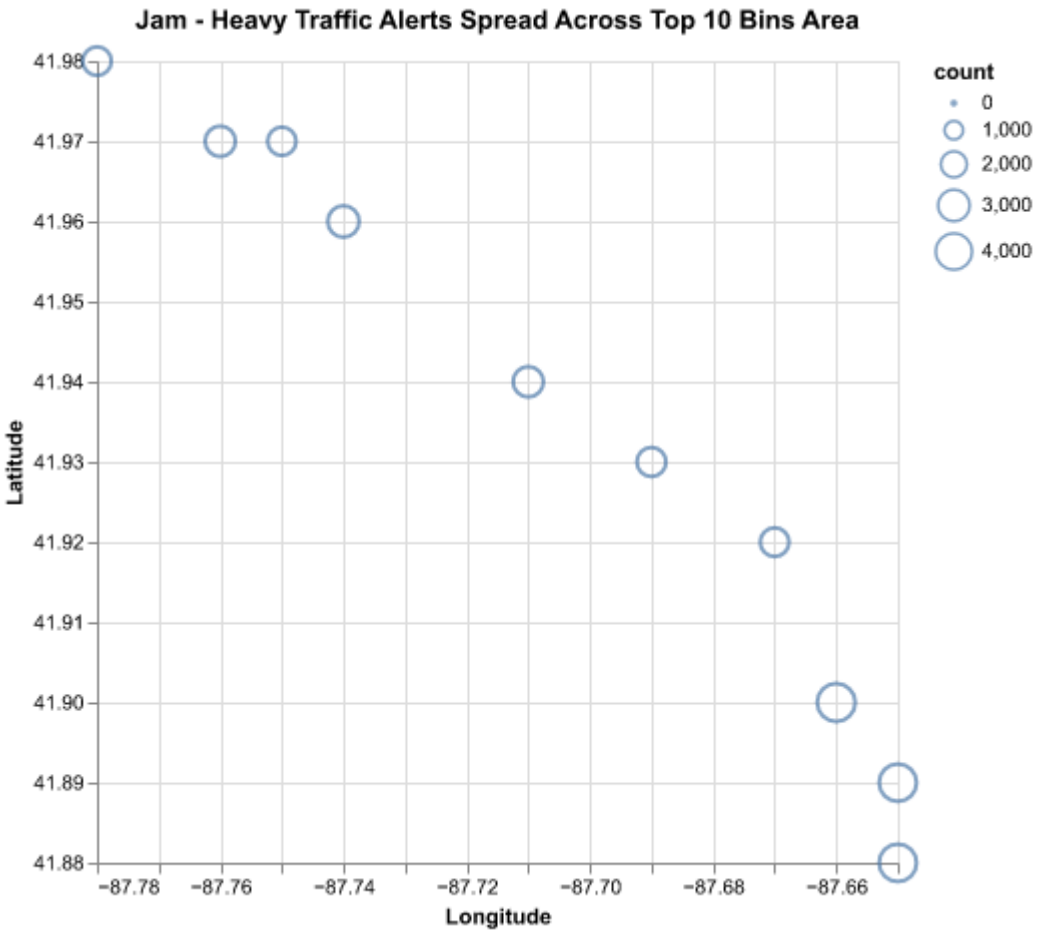
```
len(top_alerts_map)
```

```
6675
```

There are 6,675 rows in the top_alerts_map dataframe.

2. Using altair, plot a scatter plot ...

```
# Initialize Heavy Traffic Jam alerts data in top 10 bins area
jam_heavy = top_alerts_map[(top_alerts_map["updated_type"] == "Jam") & (
    top_alerts_map["updated_subtype"] == "Heavy Traffic")]

# Get the top 10 bins by alert count
top_10_jam_heavy = jam_heavy.sort_values(
    by="count", ascending=False).head(10)

# Max and min of longitude and latitude to set plot domain
min_lat = top_10_jam_heavy["binned_latitude"].min()
max_lat = top_10_jam_heavy["binned_latitude"].max()
min_long = top_10_jam_heavy["binned_longitude"].min()
max_long = top_10_jam_heavy["binned_longitude"].max()

# Plot the data
alt.Chart(top_10_jam_heavy).mark_point().encode(
    alt.X("binned_longitude", scale=alt.Scale(
        domain=[min_long, max_long]), title="Longitude"),
    alt.Y("binned_latitude", scale=alt.Scale(
        domain=[min_lat, max_lat]), title="Latitude"),
    size="count",
    tooltip=["binned_latitude", "binned_longitude", "count"]
).properties(
    title="Jam - Heavy Traffic Alerts Spread Across Top 10 Bins Area",
    height=400,
    width=400
)
```



3. Next, we will layer the scatter plot on top of a map of Chicago …

a. Download the neighborhood boundaries as a GeoJSON …

```
import requests

# EXTRA CREDIT: Download the file using the reequests package
# Download and save file
url =
 ↪  "https://data.cityofchicago.org/api/geospatial/bbvz-uum9?method=export&format=GeoJSON"
response = requests.get(url)
file_path = "D:/UCHICAGO/DATA ANALYSIS PYTHON
 ↪  II/problem-set-6-suryahardiansyah/top_alerts_map/chicago-boundaries.geojson"

with open(file_path, "wb") as f:
  f.write(response.content)
```

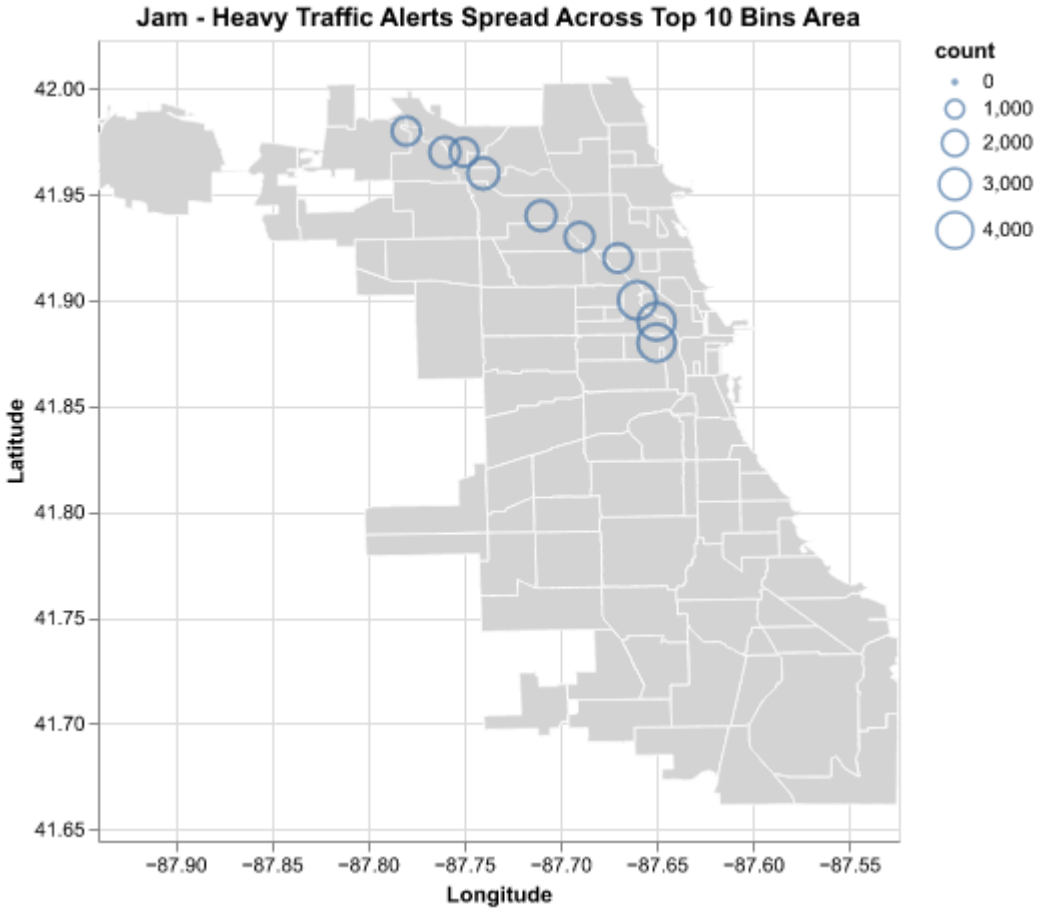
b. Load it into Python using the json package …

```
# Load geojson and prepare it for Altair
with open(file_path) as f:
    chicago_geojson = json.load(f)

geo_data = alt.Data(values=chicago_geojson["features"])
```

4. Layer the scatter plot from step 2 on top of a plot of the map using the information you loaded in step 3 and geo_data …

```
# Get max min long lat from chicago_geojson to readjust scatter plot domain
# Extract and flatten coordinates from the Chicago Boundaries GeoJSON
flat_coords = []

for feature in chicago_geojson["features"]:
    geometry = feature["geometry"]
    if geometry["type"] == "Polygon":
        for ring in geometry["coordinates"]:
            flat_coords.extend(ring)
    elif geometry["type"] == "MultiPolygon":
        for polygon in geometry["coordinates"]:
            for ring in polygon:
                flat_coords.extend(ring)

# Get min/max longitude and latitude
min_long_chi = min(coord[0] for coord in flat_coords)
max_long_chi = max(coord[0] for coord in flat_coords)
min_lat_chi = min(coord[1] for coord in flat_coords)
max_lat_chi = max(coord[1] for coord in flat_coords)

# Readjust scatter plot domain
scatter_plot = alt.Chart(top_10_jam_heavy).mark_point().encode(
    alt.X("binned_longitude", scale=alt.Scale(
        domain=[min_long_chi, max_long_chi]), title="Longitude"),
    alt.Y("binned_latitude", scale=alt.Scale(
        domain=[min_lat_chi, max_lat_chi]), title="Latitude"),
    size="count",
    tooltip=["binned_latitude", "binned_longitude", "count"]
).project(type="identity").properties(
    title="Jam - Heavy Traffic Alerts Spread Across Top 10 Bins Area",
    height=400,
```

```
    width=400
)

# Prepare the background plot
background = alt.Chart(geo_data).mark_geoshape(
    fill="lightgray",
    stroke="white"
).project(type="identity", reflectY=True).properties(
    width=400,
    height=400
)

# Show the combined plot
background + scatter_plot
```



5. Now, we are ready to make our data and plot into the Shiny dashboard …

a. For the UI component, create a single dropdown menu for type and subtype …

```
# Prepare unique type-subtype combinations for the dropdown
top_alerts_map["type_subtype"] = top_alerts_map["updated_type"] + " - " +
 ↪  top_alerts_map["updated_subtype"]
unique_combinations = sorted(top_alerts_map["type_subtype"].unique())
unique_combinations
```

```
['Accident - Major',
 'Accident - Minor',
 'Accident - Unclassified',
 'Hazard - On Road',
 'Hazard - On Shoulder',
 'Hazard - Unclassified',
 'Hazard - Weather',
 'Jam - Heavy Traffic',
 'Jam - Light Traffic',
 'Jam - Moderate Traffic',
 'Jam - Standstill Traffic',
 'Jam - Unclassified',
 'Road Closed - Construction',
 'Road Closed - Event',
 'Road Closed - Hazard',
 'Road Closed - Unclassified']
```

Figure 1: App Screenshot

There are 16 combinations of type-subtype in the dropdown menu.

b. Recreate the "Jam - Heavy Traffic" plot from above by using the dropdown menu and insert a screenshot of the graph below.

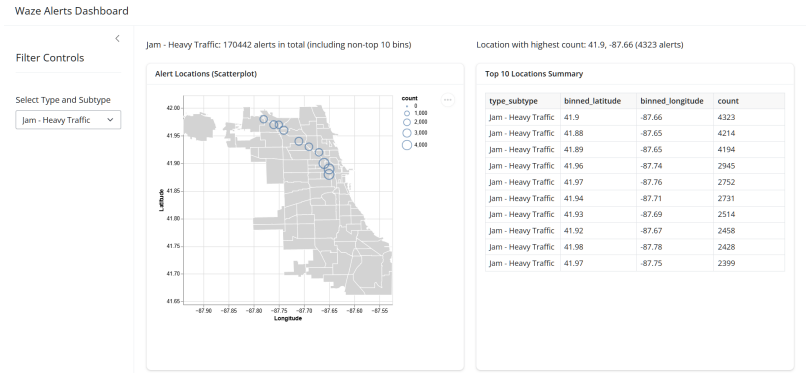


Figure 2: App Screenshot

c. Use your dashboard to answer the following question: where are alerts for road closures due to events most common? Insert a screenshot as your answer below.

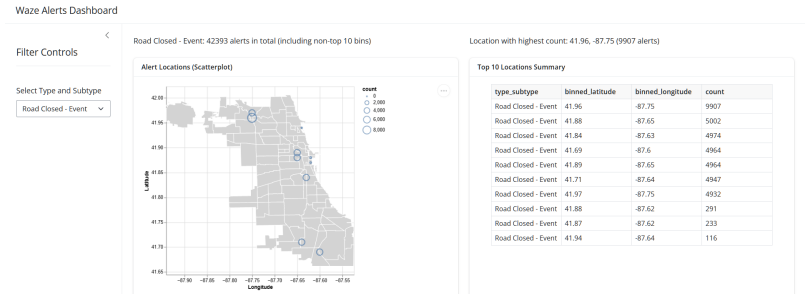
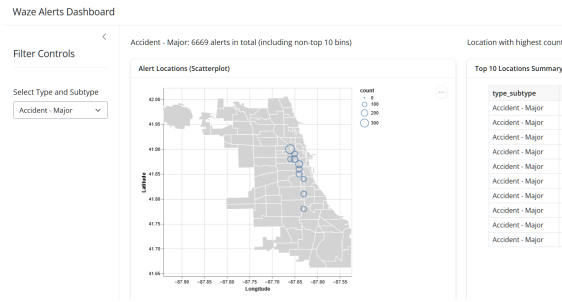


Figure 3: App Screenshot

Road Closed - Event alerts were reported most frequent at the 41.96 -87.75 bin with 9,907 reports.

d. Other than the examples above, give an example of a question this dashboard could be used to answer. Formulate the question, take a screenshot of the selection and resulting plot in the dashboard, and then provide the answer.



Q: Which highway is the most reported with major accident report?
A: Based on the coordinate bins, the major accident reports are concentrated along the stretch of the I-90/94 Expressway (Dan Ryan Expressway), particularly from the area around 41.9, -87.66 (near Downtown Chicago and the South Loop) extending southward towards 41.78, -87.63, which aligns with areas around Bridgeport and Fuller Park. This corridor is one of the busiest highways in Chicago, known for heavy traffic and complex interchanges, and it is prone to incidents like major accidents.

e. Can you suggest adding another column …

- From my perspective, the current columns—type-subtype, lat_bin, lon_bin, and count—are sufficient for delivering a clear and focused analysis. Adding more columns risks cluttering the dashboard and detracting from the user experience. The goal is to keep the interface simple and actionable. However, if further detail is needed, an optional toggle or filter could allow users to access additional information, like time of day or road type, without overwhelming the main view.

App #2: Top Location by Alert Type and Hour Dashboard (20 points)

1. We will now create a new App folder called top_alerts_map_byhour …

a. Take a look at the whole dataset we are working with …

Collapsing the dataset by the `ts` column is not ideal because second-level granularity creates too many unique groups, which can burden the computation process without adding meaningful insights. Instead, aggregating by the hour extracted from `ts` is a better approach, as it provides actionable time-based insights while maintaining clarity and computational efficiency in the data.

b. Create a new variable called hour that extracts the hour from the ts column …

```
# Convert ts column to datetime
waze_data_merged['ts'] = pd.to_datetime(waze_data_merged['ts'])

# Extract the hour part as a string in the format HH:00
waze_data_merged['hour'] = waze_data_merged['ts'].dt.strftime('%H:00')

# Collapse the dataset
top_alerts_map_byhour = waze_data_merged.groupby(
    ['hour', 'updated_type', 'updated_subtype', 'binned_latitude',
 ↪  'binned_longitude']
).size().reset_index(name="count")

# Save the collapsed dataset
top_alerts_map_byhour.to_csv("top_alerts_map_byhour/top_alerts_map_byhour.csv",
 ↪  index=False)
```

```
print(f"The collapsed dataset has {len(top_alerts_map_byhour)} rows.")
```

```
The collapsed dataset has 62825 rows.
```

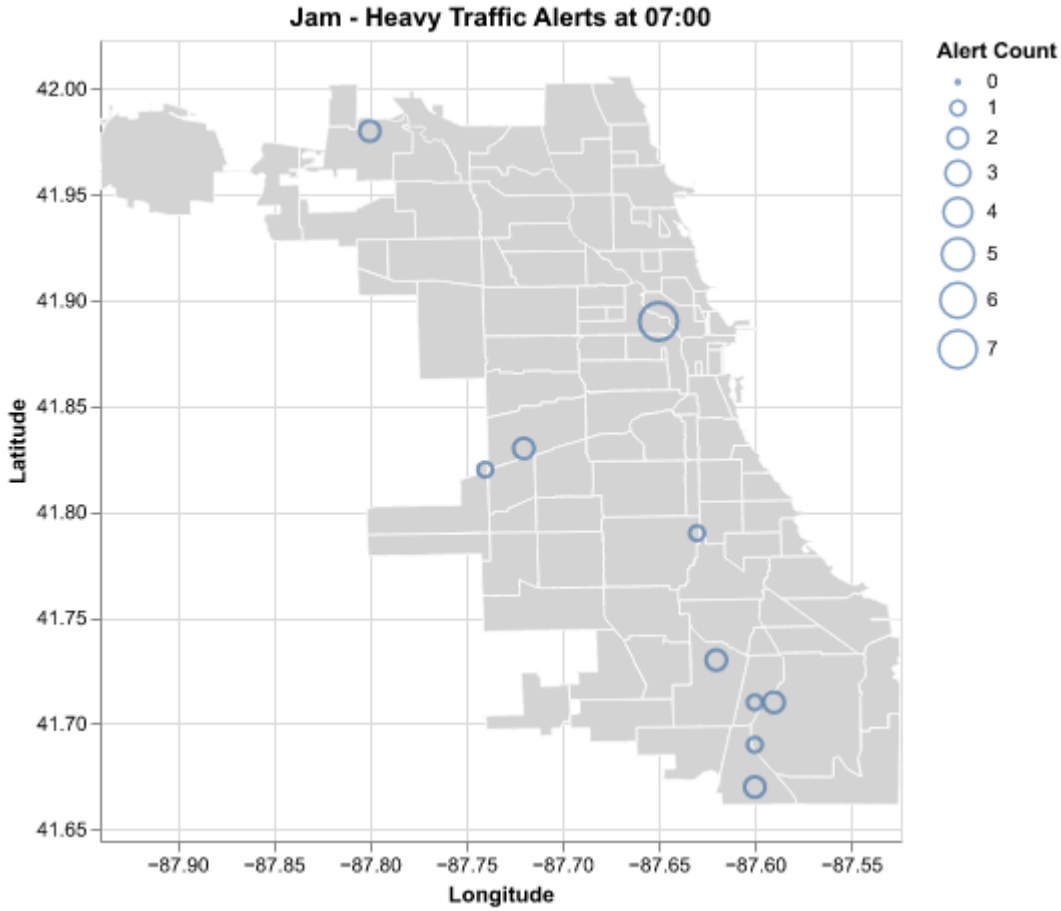
The `top_alerts_map_byhour` dataset has 68,892 rows.

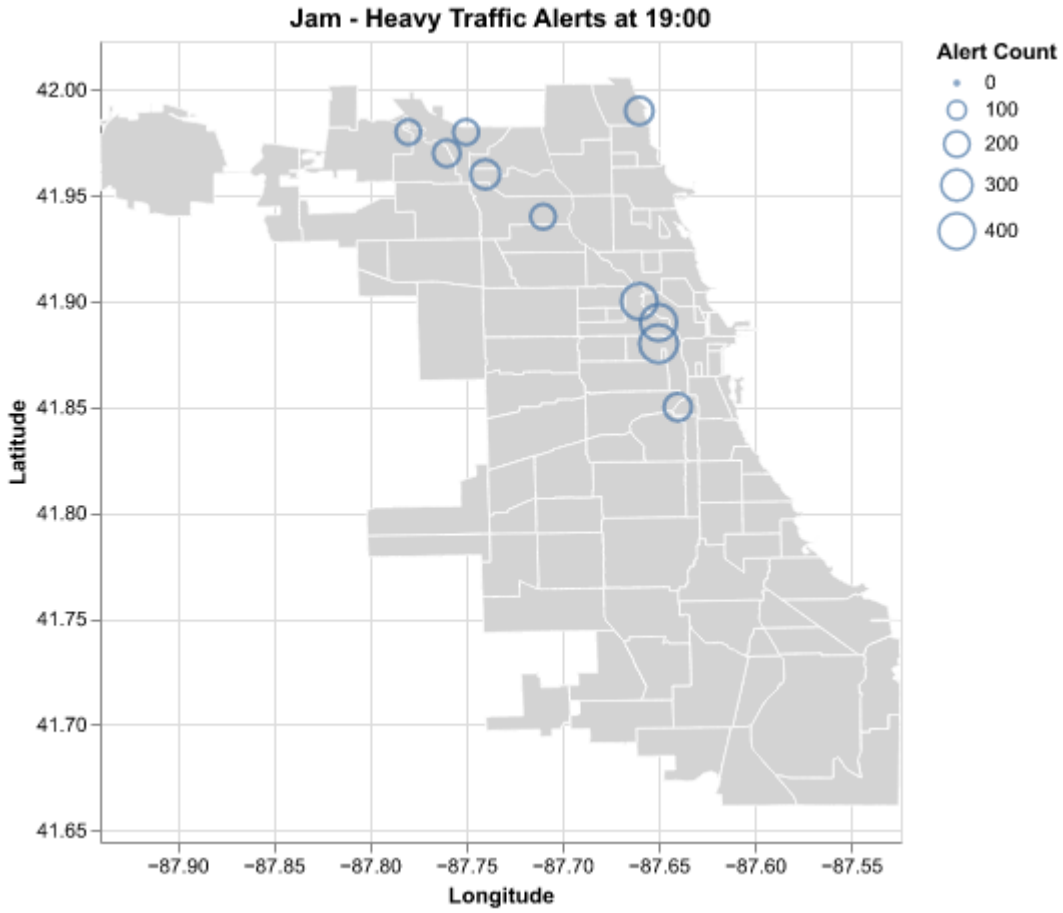
c. Generate an individual plot of the top 10 locations by hour …

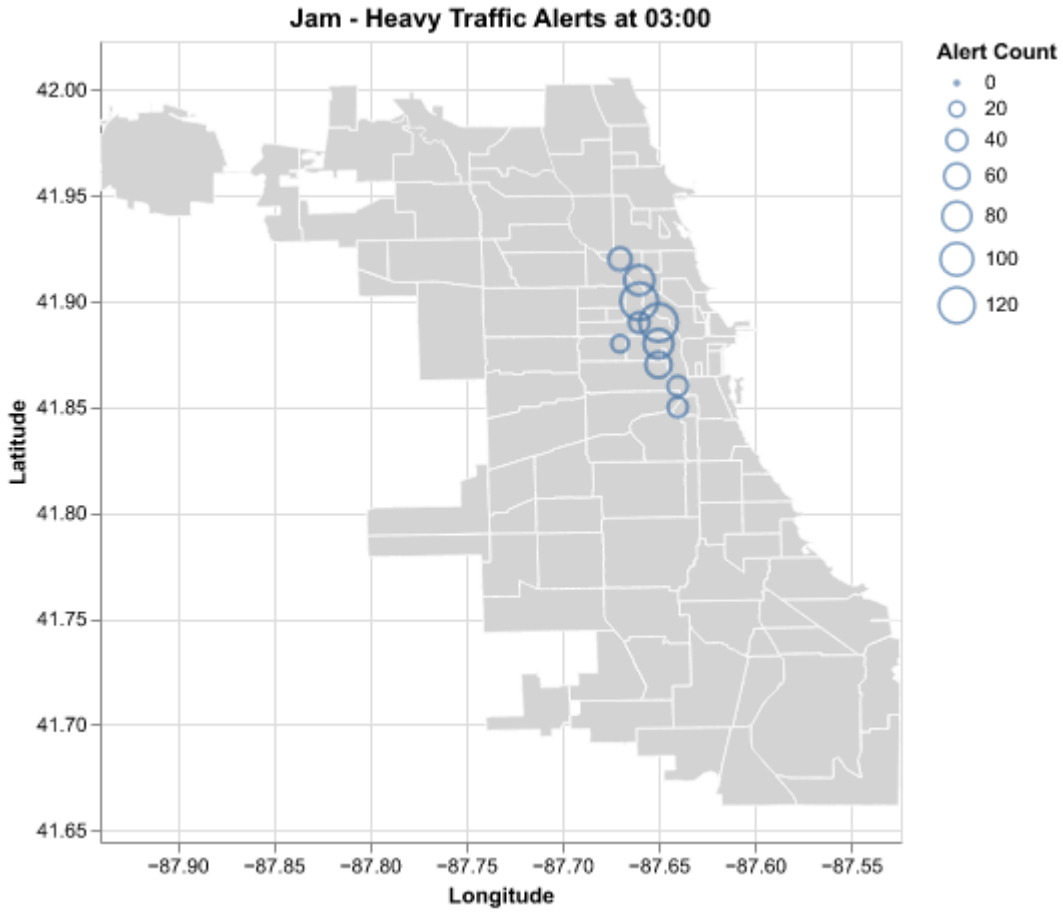
```
# Load the collapsed dataset
top_alerts_map_byhour =
 ↪  pd.read_csv("top_alerts_map_byhour/top_alerts_map_byhour.csv")

# Load the Chicago GeoJSON file
with open("top_alerts_map_byhour/chicago-boundaries.geojson") as f:
    chicago_geojson = json.load(f)

geo_data = alt.Data(values=chicago_geojson["features"])

# Filter data for 'Jam - Heavy Traffic'
jam_heavy = top_alerts_map_byhour[top_alerts_map_byhour["updated_type"] ==
 ↪  "Jam"]
jam_heavy = jam_heavy[jam_heavy["updated_subtype"] == "Heavy Traffic"]

# Define times of interest to test function
times_of_day = ["07:00", "19:00", "03:00"]
```

```
def create_hourly_plot(hour):
    # Filter data for the given hour
    hourly_data = jam_heavy[jam_heavy["hour"] == hour]

    # Get top 10 locations
    top_10 = hourly_data.sort_values(by="count", ascending=False).head(10)

    # Fixed latitude and longitude domains based on Chicago boundaries
    flat_coords = []
    for feature in chicago_geojson["features"]:
        geometry = feature["geometry"]
        if geometry["type"] == "Polygon":
            for ring in geometry["coordinates"]:
                flat_coords.extend(ring)
        elif geometry["type"] == "MultiPolygon":
            for polygon in geometry["coordinates"]:
                for ring in polygon:
                    flat_coords.extend(ring)

    min_long_chi = min(coord[0] for coord in flat_coords)
    max_long_chi = max(coord[0] for coord in flat_coords)
    min_lat_chi = min(coord[1] for coord in flat_coords)
    max_lat_chi = max(coord[1] for coord in flat_coords)

    # Create the background map
    background = alt.Chart(geo_data).mark_geoshape(
        fill="lightgray",
        stroke="white"
    ).project("identity", reflectY=True).properties(
        width=400,
        height=400
    )

    # Create the scatter plot for the top 10 locations
    scatter = alt.Chart(top_10).mark_point().encode(
        alt.X("binned_longitude:Q", scale=alt.Scale(domain=[min_long_chi,
↪   max_long_chi]), title="Longitude"),
        alt.Y("binned_latitude:Q", scale=alt.Scale(domain=[min_lat_chi,
↪   max_lat_chi]), title="Latitude"),
        size=alt.Size("count:Q", title="Alert Count"),
        tooltip=["binned_latitude", "binned_longitude", "count"]
    )
```

```
    # Combine the background map and scatter plot
    return (background + scatter).properties(
        title=f"Jam - Heavy Traffic Alerts at {hour}"
    )

# Generate plots for the specified times
plots = [create_hourly_plot(hour) for hour in times_of_day]

# Display the plots
for plot in plots:
    plot.display()
```




Jam - Heavy Traffic Alerts at 19:00
Alert Count
0
100
200
300
400
Latitude
42.00
41.95
41.90
41.85
41.80
41.75
41.70
41.65
−87.90
−87.85
−87.80
−87.75
−87.70
−87.65
−87.60
−87.55
Longitude



2. We will now turn into creating the Shiny app …

a. Create the UI for the app, which should have the dropdown menu to choose a combination of type and subtype, and a slider to pick the hour. Insert a screenshot of the UI below.

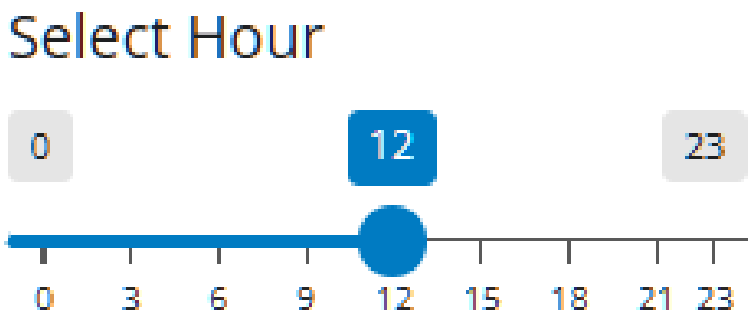


Figure 4: App Screenshot

```
ui.input_slider(id="hour", label="Select Hour",
            min=0, max=23, value=12, step=1,
            ticks=True)
```

b. Recreate the “Jam - Heavy Traffic” plot from above …

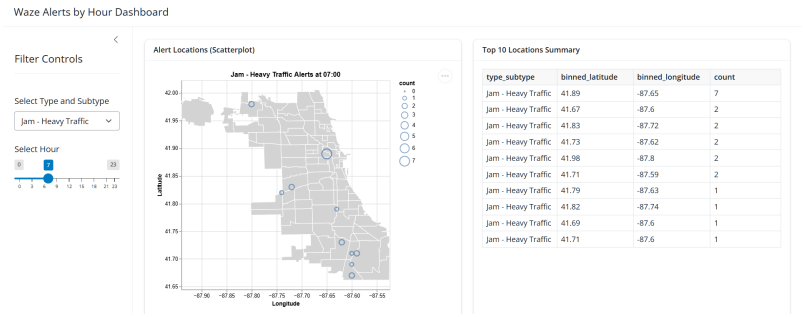


Figure 5: App Screenshot

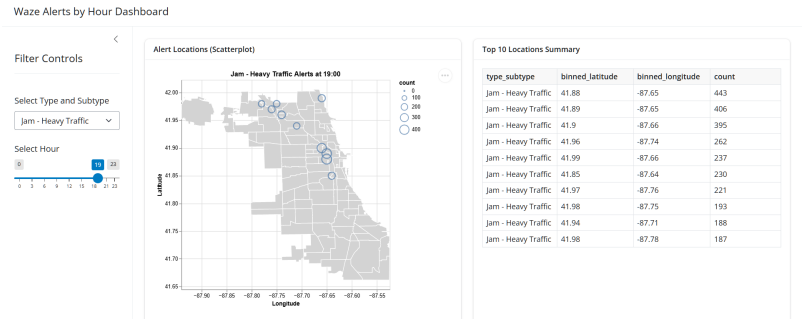


Figure 6: App Screenshot

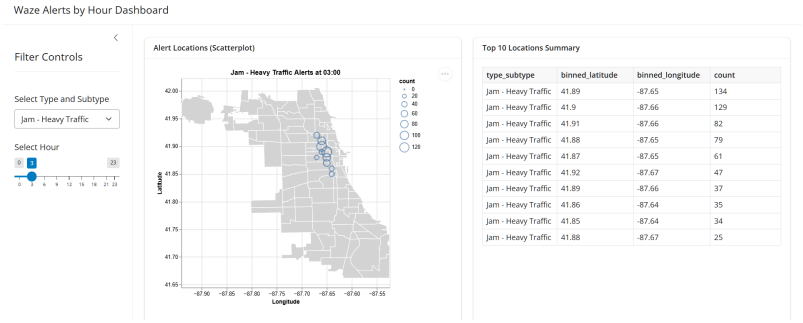


Figure 7: App Screenshot

The three plots made by the dashboard precisely appear the same as ones made by coding before in the previous section above.

c. Use your dashboard to answer the following question …

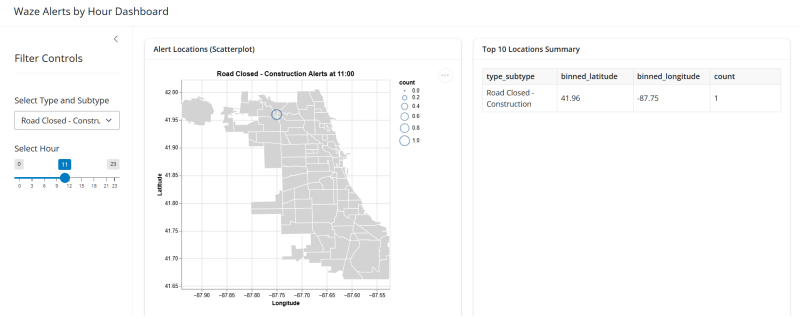


Figure 8: App Screenshot

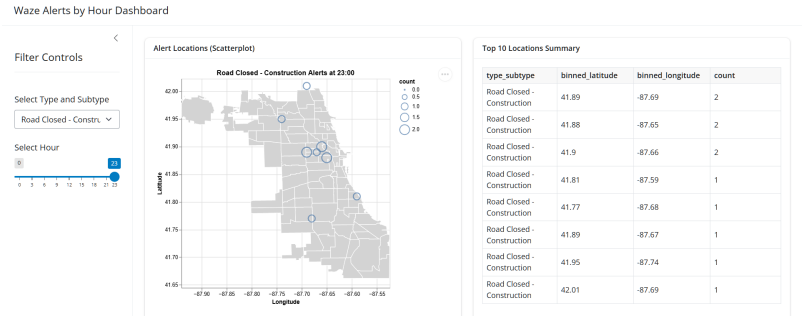


Figure 9: App Screenshot

Based on the two plots, it appears that there are more road construction reports in the morning than at night (e.g., 9 at 11 PM (night) versus only 1 at 11 AM (morning)). This discrepancy

might be attributed to actual construction activity schedules, as roadwork is often planned during low-traffic hours, or due to underreporting by users in the morning when they are typically occupied with indoor tasks. However, to draw a more reliable and generalizable conclusion, a broader analysis covering more ponits of time of the day is necessary.

App #3: Top Location by Alert Type and Hour Dashboard (20 points)

1. As choosing a single hour might not the best way …

a. Think about what we did in App 1 and 2 …

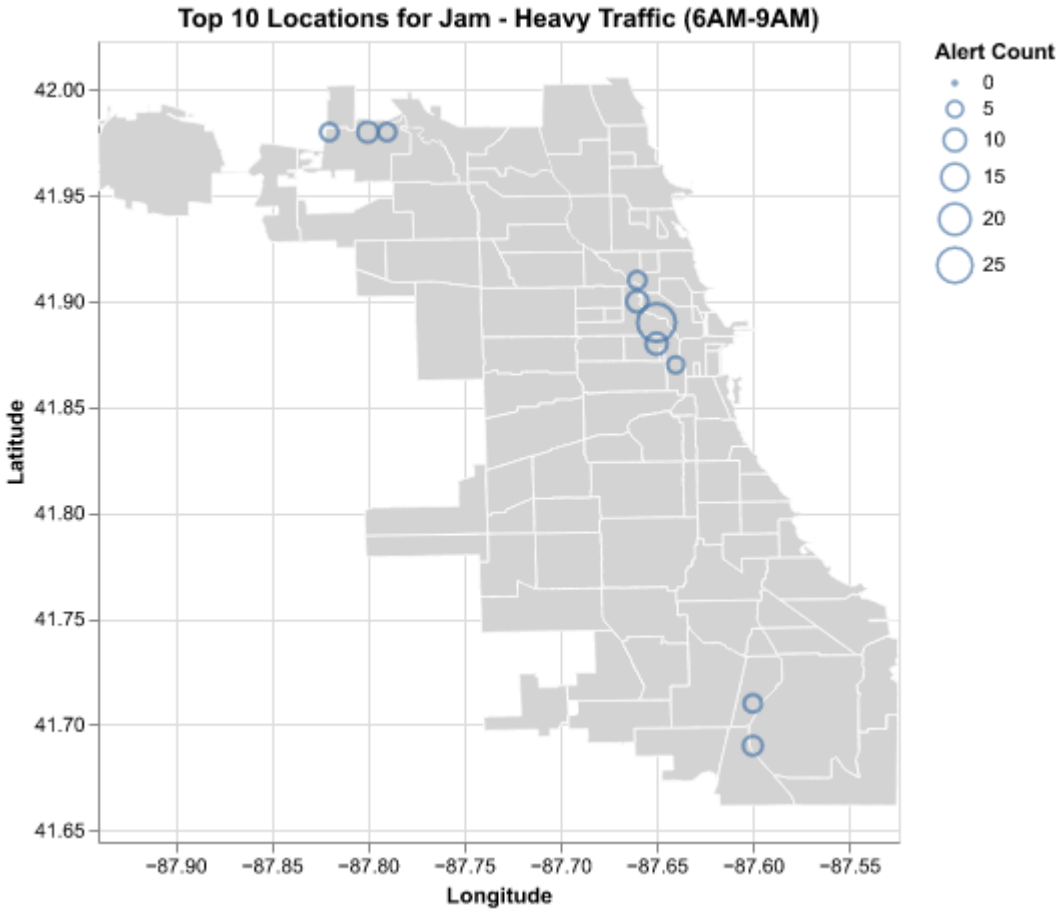
- No, that is not a good idea. Pre-aggregating data for all possible ranges would create a large, complex dataset with overlapping rows, which would be computationally inefficient and hard to manage. Instead, it would be better to collapse the dataset by individual hours (as in App 2) and dynamically filter and aggregate the data in the Shiny app based on the user's selected range.

b. Before going into the Shiny app …

```
# Load the previous collapsed dataset
top_alerts_map_byhour =
 ↪  pd.read_csv("top_alerts_map_byhour/top_alerts_map_byhour.csv")

# Load the Chicago GeoJSON file
with open("top_alerts_map_byhour/chicago-boundaries.geojson") as f:
    chicago_geojson = json.load(f)

geo_data = alt.Data(values=chicago_geojson["features"])

# Filter data for 'Jam - Heavy Traffic' between 6AM and 9AM
jam_heavy = top_alerts_map_byhour[
    (top_alerts_map_byhour["updated_type"] == "Jam") &
    (top_alerts_map_byhour["updated_subtype"] == "Heavy Traffic") &
    (top_alerts_map_byhour["hour"].isin(["06:00", "07:00", "08:00",
 ↪  "09:00"]))
]

# Aggregate counts across the selected range of hours
jam_heavy_aggregated = jam_heavy.groupby(
    ["binned_latitude", "binned_longitude"]
```

```
).agg({"count": "sum"}).reset_index()

# Get top 10 locations by count
top_10_jam_heavy = jam_heavy_aggregated.sort_values(by="count",
 ↪  ascending=False).head(10)

# Fixed latitude and longitude domains based on Chicago boundaries
flat_coords = []
for feature in chicago_geojson["features"]:
    geometry = feature["geometry"]
    if geometry["type"] == "Polygon":
        for ring in geometry["coordinates"]:
            flat_coords.extend(ring)
    elif geometry["type"] == "MultiPolygon":
        for polygon in geometry["coordinates"]:
            for ring in polygon:
                flat_coords.extend(ring)

min_long_chi = min(coord[0] for coord in flat_coords)
max_long_chi = max(coord[0] for coord in flat_coords)
min_lat_chi = min(coord[1] for coord in flat_coords)
max_lat_chi = max(coord[1] for coord in flat_coords)

# Create the background map
background = alt.Chart(geo_data).mark_geoshape(
    fill="lightgray",
    stroke="white"
).project("identity", reflectY=True).properties(
    width=400,
    height=400
)

# Create the scatter plot for the top 10 locations
scatter = alt.Chart(top_10_jam_heavy).mark_point().encode(
    alt.X("binned_longitude:Q", scale=alt.Scale(domain=[min_long_chi,
 ↪  max_long_chi]), title="Longitude"),
    alt.Y("binned_latitude:Q", scale=alt.Scale(domain=[min_lat_chi,
 ↪  max_lat_chi]), title="Latitude"),
    size=alt.Size("count:Q", title="Alert Count"),
    tooltip=["binned_latitude", "binned_longitude", "count"]
)
```

```
# Combine the background map and scatter plot
plot = (background + scatter).properties(
    title="Top 10 Locations for Jam - Heavy Traffic (6AM-9AM)"
)

# Display the plot
plot.display()
```



2. We will now create our new Shiny app …

a. Create the required UI for the App …

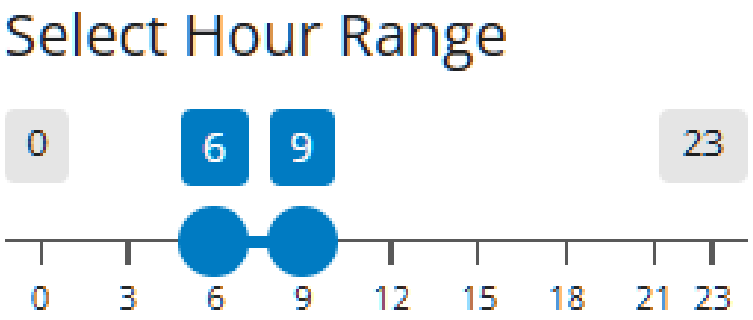


Figure 10: App Screenshot

```
ui.input_slider(id="hour", label="Select Hour",
            min=0, max=23, value=[6, 9], step=1, ticks=True)
```

b. Recreate the "Jam - Heavy Traffic" plot …

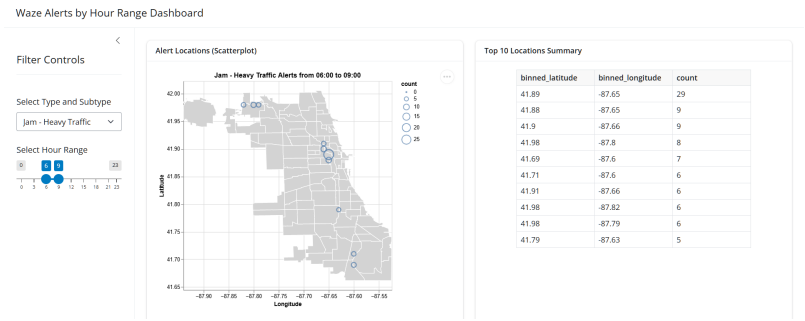


Figure 11: App Screenshot

3. We will now add a conditional panel to the app …

a. Read the documentation on switch buttons …

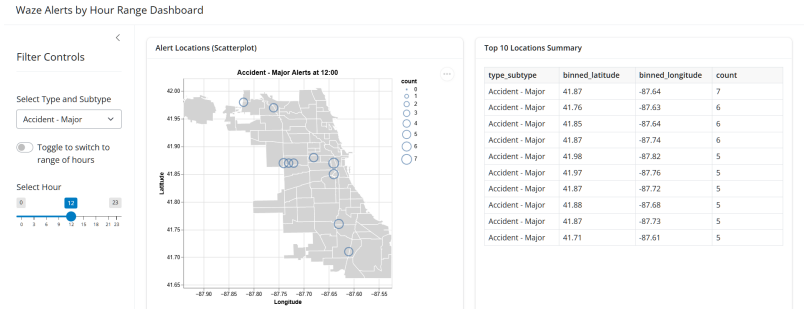


Figure 12: App Screenshot

- The possible values for a ui.input_switch are:
  - **True**: When the switch is toggled "on" or in the active state.
  - **False**: When the switch is toggled "off" or in the inactive state.

b. Modify the UI to add a conditional panel …

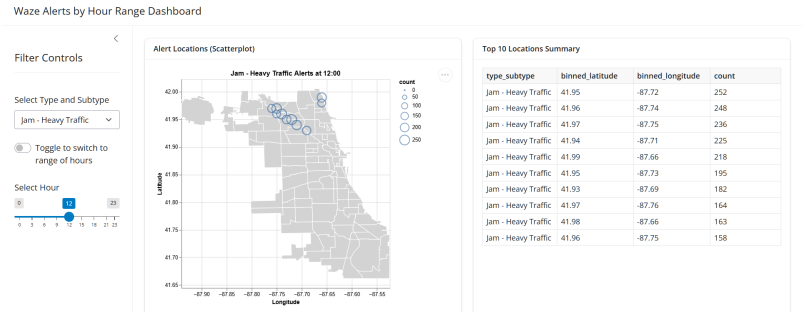


Figure 13: App Screenshot

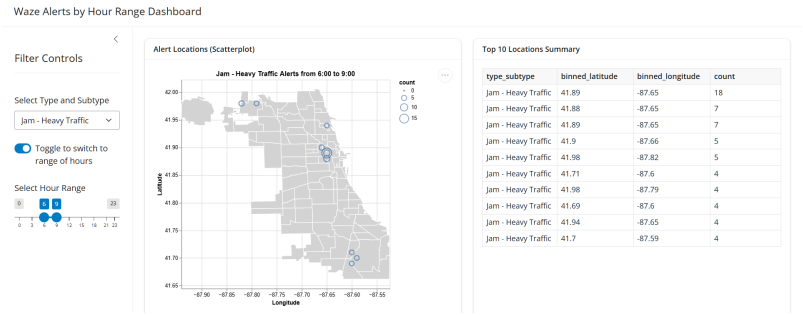


Figure 14: App Screenshot

c. Lastly, modify the UI and server logic …

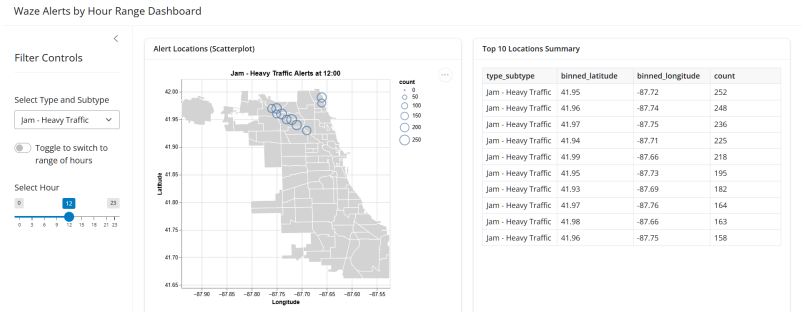


Figure 15: App Screenshot

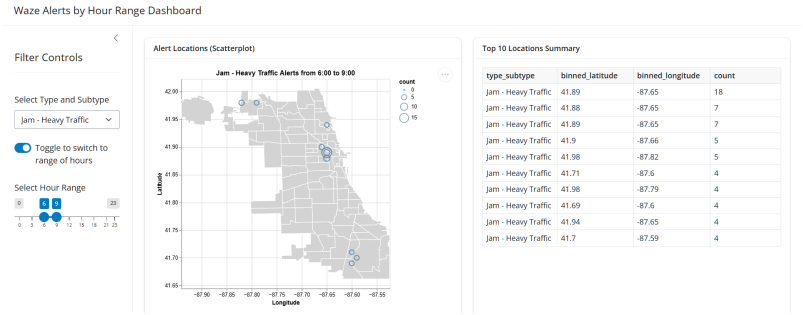


Figure 16: App Screenshot

d. EXTRA CREDIT: No need to code this part …

The plot incorporates three layers: the background map, a scatter plot for morning alerts, and another for afternoon alerts, with the dataset predefined based on "Time Period." To achieve this, we exclude the dynamic hour selection logic, such as the hour sliders and toggle switch, and instead use a dataset with a "Time Period" column grouping alerts into "Morning" (6AM-12PM) and "Afternoon" (12PM-6PM). The scatter plot is updated to include two distinct layers, color-coded for each time period, with size representing the number of alerts. This approach eliminates user input for toggling or filtering by hours, allowing both time periods to be visualized simultaneously in an intuitive and streamlined manner.