Problem Set 6 - Waze Shiny Dashboard

Attaullah Abbasi

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:AA
2. "I have uploaded the names of anyone I worked with on the problem set **here**" AA (2 point)
3. Late coins used this pset: 1 Late coins left after submission: 0
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. Submit 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 the gradescope repo assignment (5 points).
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!*

```
def print_file_contents(file_path):
    """Print contents of a file."""
    try:
        with open(file_path, 'r') as f:
            content = f.read()
            print("```python")
            print(content)
            print("```")
    except FileNotFoundError:
        print("```python")
```

```
        print(f"Error: File '{file_path}' not found")
        print("```")
    except Exception as e:
        print("```python")
        print(f"Error reading file: {e}")
        print("```")

print_file_contents("./top_alerts_map_byhour/app.py") # Change accordingly
```

Background

Data Download and Exploration (20 points)

1.

```
import zipfile

# Step 1: Unzip the waze_data.zip file
zip_file_path = '/Users/attaullah/Documents/PS-6-/waze_data.zip'
extract_dir = '/Users/attaullah/Documents/PS-6-/waze_data/'

# Extract files
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall(extract_dir)
print(f"Files extracted to: {extract_dir}")

# Step 2: Load the waze_data_sample.csv file into a Pandas DataFrame
sample_file_path = '/Users/attaullah/Documents/PS-6-/waze_data/waze_data_sample.csv'
waze_sample_data = pd.read_csv(sample_file_path)

# Step 3: Report variable names and Altair-compatible data types
altair_dtypes = {
    'object': 'Nominal',
    'float64': 'Quantitative',
    'int64': 'Quantitative'
}
exclude_columns = ['ts', 'geo', 'geoWKT']

# Map the Altair-compatible types for the dataset
variable_types = {
    col: altair_dtypes[str(dtype)]
    for col, dtype in waze_sample_data.dtypes.items()
    if col not in exclude_columns
}

# Convert to DataFrame for better readability
variable_types_df = pd.DataFrame(variable_types.items(), columns=["Variable", "Altair
 ↪  Data Type"])

# Output results
```

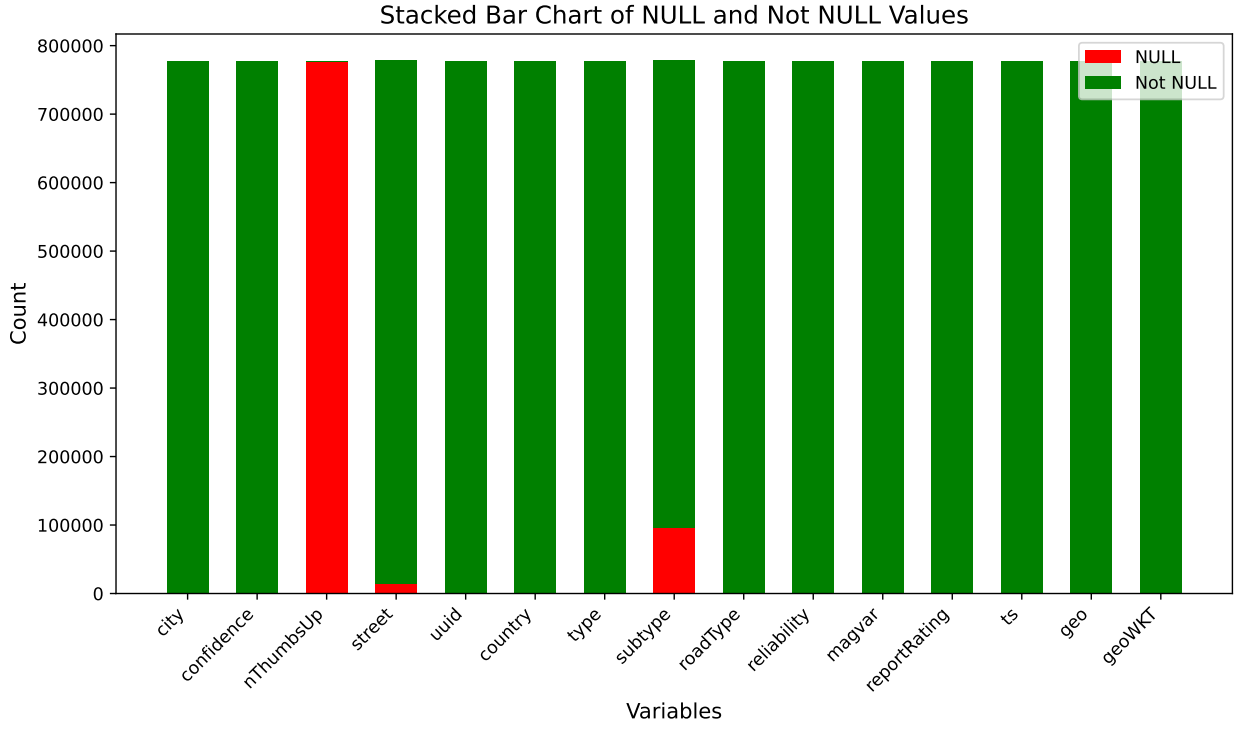
```
print("\nVariable Names and Altair Data Types:")
print(variable_types_df)
```

```
Files extracted to: /Users/attaullah/Documents/PS-6-/waze_data/

Variable Names and Altair Data Types:
        Variable Altair Data Type
0     Unnamed: 0     Quantitative
1           city          Nominal
2     confidence     Quantitative
3      nThumbsUp     Quantitative
4         street          Nominal
5           uuid          Nominal
6        country          Nominal
7           type          Nominal
8        subtype          Nominal
9       roadType     Quantitative
10   reliability     Quantitative
11        magvar     Quantitative
12  reportRating     Quantitative
```

2.

```
import matplotlib.pyplot as plt

# Step 1: Load the waze_data.csv file
full_file_path = '/Users/attaullah/Documents/PS-6-/waze_data/waze_data.csv'
waze_full_data = pd.read_csv(full_file_path)

# Step 2: Calculate NULL and non-NULL values for each column
null_counts = waze_full_data.isnull().sum()
not_null_counts = waze_full_data.notnull().sum()

# Create a DataFrame summarizing NULL and non-NULL counts
null_summary = pd.DataFrame({
    'Variable': null_counts.index,
    'NULL': null_counts.values,
    'Not NULL': not_null_counts.values
})

# Step 3: Identify variables with NULL values and the variable with the highest share of
 ↪  NULLs
null_summary['% NULL'] = (null_summary['NULL'] / len(waze_full_data)) * 100
variables_with_nulls = null_summary[null_summary['NULL'] > 0]
highest_null_variable = null_summary.loc[null_summary['% NULL'].idxmax()]

# Step 4: Create a stacked bar chart
fig, ax = plt.subplots(figsize=(10, 6))
bar_width = 0.6

# Plot the NULL and Not NULL counts
ax.bar(null_summary['Variable'], null_summary['NULL'], label='NULL', color='red',
 ↪  width=bar_width)
ax.bar(null_summary['Variable'], null_summary['Not NULL'], bottom=null_summary['NULL'],
 ↪  label='Not NULL', color='green', width=bar_width)
```

```
# Chart customization
ax.set_title('Stacked Bar Chart of NULL and Not NULL Values', fontsize=14)
ax.set_xlabel('Variables', fontsize=12)
ax.set_ylabel('Count', fontsize=12)
ax.set_xticks(range(len(null_summary['Variable'])))
ax.set_xticklabels(null_summary['Variable'], rotation=45, ha='right')
ax.legend()

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

# Step 5: Print analysis results
print("\nVariables with NULL Values:")
print(variables_with_nulls)
print("\nVariable with the Highest Share of NULLs:")
print(highest_null_variable)
```




```
Variables with NULL Values:
    Variable    NULL  Not NULL     % NULL
2  nThumbsUp  776723      1371  99.823800
3     street   14073    764021   1.808650
7    subtype   96086    682008  12.348894

Variable with the Highest Share of NULLs:
Variable    nThumbsUp
NULL           776723
```

```
Not NULL         1371
% NULL        99.8238
Name: 2, dtype: object
```

Analysis of NULL Values in `waze_data.csv`

Variables with NULL Values

The following variables have missing values: 1. **`nThumbsUp`** - Number of NULL values: 776,723 - Percentage of NULL values: 99.82% 2. **`street`** - Number of NULL values: 14,073 - Percentage of NULL values: 1.81% 3. **`subtype`** - Number of NULL values: 96,086 - Percentage of NULL values: 12.35%

Variable with the Highest Share of Observations Missing

- `nThumbsUp`
  - This variable has the highest proportion of missing values, with **99.82% of its observations NULL**.

Visualization

Refer to the stacked bar chart for a visual representation of NULL and non-NULL distributions across all variables.

3.

```
# Step 1: Analyze unique values in `type` and `subtype`
unique_type_subtype = waze_full_data[['type', 'subtype']].drop_duplicates()

# Count how many types have a subtype as NA
na_subtype_count = unique_type_subtype['subtype'].isna().sum()

# Print unique values for `type` and `subtype`
unique_types = unique_type_subtype['type'].unique()
unique_subtypes = unique_type_subtype['subtype'].unique()

# Identify types with subtypes that could potentially have sub-subtypes
types_with_subtypes = unique_type_subtype[~unique_type_subtype['subtype'].isna()]
potential_sub_subtypes = types_with_subtypes['type'].value_counts()

# Output results
print("Unique `type` values:")
print(unique_types)
print("\nUnique `subtype` values:")
print(unique_subtypes)
print(f"\nNumber of `type` entries with `subtype` as NA: {na_subtype_count}")
print("\nTypes with subtypes that have enough information to consider sub-subtypes:")
print(potential_sub_subtypes)
```

```
Unique `type` values:
['JAM' 'ACCIDENT' 'ROAD_CLOSED' 'HAZARD']

Unique `subtype` values:
[nan '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_OBJECT' 'HAZARD_ON_ROAD_POT_HOLE'
 'HAZARD_ON_ROAD_TRAFFIC_LIGHT_FAULT' 'HAZARD_ON_SHOULDER'
 'HAZARD_ON_SHOULDER_CAR_STOPPED' 'HAZARD_WEATHER' 'HAZARD_WEATHER_FLOOD'
 'JAM_HEAVY_TRAFFIC' 'JAM_MODERATE_TRAFFIC' 'JAM_STAND_STILL_TRAFFIC'
 'ROAD_CLOSED_EVENT' 'HAZARD_ON_ROAD_LANE_CLOSED' 'HAZARD_WEATHER_FOG'
 'ROAD_CLOSED_CONSTRUCTION' 'HAZARD_ON_ROAD_ROAD_KILL'
 'HAZARD_ON_SHOULDER_ANIMALS' 'HAZARD_ON_SHOULDER_MISSING_SIGN'
 'JAM_LIGHT_TRAFFIC' 'HAZARD_WEATHER_HEAVY_SNOW' 'ROAD_CLOSED_HAZARD'
 'HAZARD_WEATHER_HAIL']

Number of `type` entries with `subtype` as NA: 4

Types with subtypes that have enough information to consider sub-subtypes:
type
HAZARD         19
JAM             4
ROAD_CLOSED     3
ACCIDENT        2
Name: count, dtype: int64
```

Analysis of `type` and `subtype` Variables

a. Unique Values and Subtype Analysis

1. **Unique `type` values:**
   - JAM - ACCIDENT - ROAD_CLOSED - HAZARD
2. **Unique `subtype` values:**
   - `NA` - 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_OBJECT - HAZARD_ON_ROAD_POT_HOLE
   - HAZARD_ON_ROAD_TRAFFIC_LIGHT_FAULT - HAZARD_ON_SHOULDER
   - HAZARD_ON_SHOULDER_CAR_STOPPED - HAZARD_WEATHER
   - HAZARD_WEATHER_FLOOD - JAM_HEAVY_TRAFFIC - JAM_MODERATE_TRAFFIC
   - JAM_STAND_STILL_TRAFFIC - ROAD_CLOSED_EVENT
   - HAZARD_ON_ROAD_LANE_CLOSED - HAZARD_WEATHER_FOG
   - ROAD_CLOSED_CONSTRUCTION - HAZARD_ON_ROAD_ROAD_KILL
   - HAZARD_ON_SHOULDER_ANIMALS - HAZARD_ON_SHOULDER_MISSING_SIGN
   - JAM_LIGHT_TRAFFIC - HAZARD_WEATHER_HEAVY_SNOW
   - ROAD_CLOSED_HAZARD - HAZARD_WEATHER_HAIL
3. **Number of `type` entries with `subtype` as `NA`:**
   - 4 `type` entries have `subtype` as `NA`.
4. **Types with enough information to consider sub-subtypes:**
   - HAZARD: 19 subtypes - JAM: 4 subtypes
   - ROAD_CLOSED: 3 subtypes - ACCIDENT: 2 subtypes

b. Hierarchical Values for `type` and `subtype`

Here is the cleaned and user-friendly hierarchy:

- **Accident**
  - Major
  - Minor
- **Hazard**
  - On Road
    - Car Stopped - Construction - Emergency Vehicle - Ice - Object
    - Pot Hole - Lane Closed - Traffic Light Fault - Road Kill
  - On Shoulder
    - Car Stopped - Animals - Missing Sign
  - Weather
    - Flood - Fog - Heavy Snow - Hail
- **Jam**
  - Heavy Traffic - Moderate Traffic - Stand Still Traffic - Light Traffic
- **Road Closed**
  - Event - Construction - Hazard

---

c. Should We Keep the `NA` Subtypes?

- **Decision:** Yes, we should keep the `NA` subtypes but classify them as "Unclassified."
- **Reasoning:**
  - Keeping the `NA` subtypes ensures that we do not lose any meaningful `type` data.
  - Labeling them as "Unclassified" provides clarity and avoids confusion when displaying data in the Shiny Apps or dashboards.

Action:

- Replace all `NA` subtypes with "Unclassified."

4.

4.1

```
# Step 1: Create a crosswalk DataFrame from unique type and subtype combinations
crosswalk = unique_type_subtype.copy()

# Step 2: Update `type` to create `updated_type`
crosswalk['updated_type'] = crosswalk['type'].str.title()

# Step 3: Update `subtype` to create `updated_subtype` (replace NA with 'Unclassified'
 ↪  and clean formatting)
crosswalk['updated_subtype'] = crosswalk['subtype'].fillna('Unclassified')
crosswalk['updated_subtype'] = crosswalk['updated_subtype'].str.replace('_', '
 ↪  ').str.title()
```

```
# Step 4: Assign `updated_subsubtype` (optional for sub-subtypes)
# In this case, we'll leave it as `None` for now as sub-subtypes are not well-defined.
crosswalk['updated_subsubtype'] = None

# Step 5: Preview the crosswalk DataFrame
print("Crosswalk DataFrame:")
print(crosswalk.head())

# Save the crosswalk as a CSV file if needed
crosswalk.to_csv('/Users/attaullah/Documents/PS-6-/waze_data/crosswalk.csv', index=False)
print("\nCrosswalk saved to '/Users/attaullah/Documents/PS-6-/waze_data/crosswalk.csv'")
```

```
Crosswalk DataFrame:
            type         subtype updated_type updated_subtype  \
0            JAM             NaN          Jam    Unclassified
1       ACCIDENT             NaN     Accident    Unclassified
2    ROAD_CLOSED             NaN  Road_Closed    Unclassified
26        HAZARD             NaN       Hazard    Unclassified
122     ACCIDENT  ACCIDENT_MAJOR     Accident  Accident Major

    updated_subsubtype
0                 None
1                 None
2                 None
26                None
122               None

Crosswalk saved to '/Users/attaullah/Documents/PS-6-/waze_data/crosswalk.csv'
```

2.

```
# Step 1: Define the hierarchy for updated_subtype and updated_subsubtype
hierarchy = {
    "Accident": {
        "Major": "Accident Major",
        "Minor": "Accident Minor",
        "Unclassified": "Unclassified"
    },
    "Hazard": {
        "On Road": [
            "Car Stopped", "Construction", "Emergency Vehicle", "Ice",
            "Object", "Pot Hole", "Lane Closed", "Traffic Light Fault", "Road Kill"
        ],
        "On Shoulder": ["Car Stopped", "Animals", "Missing Sign"],
        "Weather": ["Flood", "Fog", "Heavy Snow", "Hail"],
        "Unclassified": "Unclassified"
    },
    "Jam": {
        "Traffic": [
            "Heavy Traffic", "Moderate Traffic", "Stand Still Traffic", "Light Traffic"
        ],
        "Unclassified": "Unclassified"
```

```
    },
    "Road Closed": {
        "Details": ["Event", "Construction", "Hazard"],
        "Unclassified": "Unclassified"
    }
}

# Step 2: Update the crosswalk DataFrame based on the hierarchy
def map_updated_columns(row):
    type_hierarchy = hierarchy.get(row['updated_type'], {})

    # Default subtype mappings
    subtype_mapping = {v: k for k, values in type_hierarchy.items() for v in (values if
 ↪  isinstance(values, list) else [values])}
    row['updated_subsubtype'] = subtype_mapping.get(row['updated_subtype'], None)

    return row

crosswalk = crosswalk.apply(map_updated_columns, axis=1)

# Step 3: Ensure NA subtypes are labeled as "Unclassified"
crosswalk['updated_subtype'] = crosswalk['updated_subtype'].fillna("Unclassified")
crosswalk['updated_subsubtype'] = crosswalk['updated_subsubtype'].fillna("Unclassified")

# Step 4: Preview the updated crosswalk DataFrame
print("Updated Crosswalk DataFrame:")
print(crosswalk)

# Save the updated crosswalk as a CSV file
crosswalk.to_csv('/Users/attaullah/Documents/PS-6-/waze_data/updated_crosswalk.csv',
 ↪  index=False)
print("\nUpdated crosswalk saved to
 ↪  '/Users/attaullah/Documents/PS-6-/waze_data/updated_crosswalk.csv'")
```

```
Updated Crosswalk DataFrame:
               type                              subtype updated_type  \
0               JAM                                  NaN          Jam
1          ACCIDENT                                  NaN     Accident
2       ROAD_CLOSED                                  NaN  Road_Closed
26           HAZARD                                  NaN       Hazard
122         ACCIDENT                     ACCIDENT_MAJOR     Accident
131         ACCIDENT                     ACCIDENT_MINOR     Accident
148           HAZARD                     HAZARD_ON_ROAD       Hazard
190           HAZARD          HAZARD_ON_ROAD_CAR_STOPPED       Hazard
240           HAZARD         HAZARD_ON_ROAD_CONSTRUCTION       Hazard
276           HAZARD    HAZARD_ON_ROAD_EMERGENCY_VEHICLE       Hazard
302           HAZARD                 HAZARD_ON_ROAD_ICE       Hazard
303           HAZARD              HAZARD_ON_ROAD_OBJECT       Hazard
355           HAZARD            HAZARD_ON_ROAD_POT_HOLE       Hazard
478           HAZARD  HAZARD_ON_ROAD_TRAFFIC_LIGHT_FAULT       Hazard
483           HAZARD                 HAZARD_ON_SHOULDER       Hazard
485           HAZARD      HAZARD_ON_SHOULDER_CAR_STOPPED       Hazard
854           HAZARD                     HAZARD_WEATHER       Hazard
857           HAZARD               HAZARD_WEATHER_FLOOD       Hazard
```

```
858             JAM                   JAM_HEAVY_TRAFFIC          Jam
1122            JAM                JAM_MODERATE_TRAFFIC          Jam
1184            JAM             JAM_STAND_STILL_TRAFFIC          Jam
1335    ROAD_CLOSED                   ROAD_CLOSED_EVENT  Road_Closed
1905         HAZARD          HAZARD_ON_ROAD_LANE_CLOSED       Hazard
5557         HAZARD                 HAZARD_WEATHER_FOG       Hazard
7331    ROAD_CLOSED            ROAD_CLOSED_CONSTRUCTION  Road_Closed
21443        HAZARD            HAZARD_ON_ROAD_ROAD_KILL       Hazard
21447        HAZARD          HAZARD_ON_SHOULDER_ANIMALS       Hazard
21940        HAZARD     HAZARD_ON_SHOULDER_MISSING_SIGN       Hazard
38546           JAM                   JAM_LIGHT_TRAFFIC          Jam
44216        HAZARD           HAZARD_WEATHER_HEAVY_SNOW       Hazard
54556   ROAD_CLOSED                 ROAD_CLOSED_HAZARD  Road_Closed
229005       HAZARD                HAZARD_WEATHER_HAIL       Hazard

                          updated_subtype updated_subsubtype
0                            Unclassified       Unclassified
1                            Unclassified       Unclassified
2                            Unclassified       Unclassified
26                           Unclassified       Unclassified
122                        Accident Major              Major
131                        Accident Minor              Minor
148                        Hazard On Road       Unclassified
190             Hazard On Road Car Stopped       Unclassified
240           Hazard On Road Construction       Unclassified
276       Hazard On Road Emergency Vehicle       Unclassified
302                     Hazard On Road Ice       Unclassified
303                 Hazard On Road Object       Unclassified
355               Hazard On Road Pot Hole       Unclassified
478     Hazard On Road Traffic Light Fault       Unclassified
483                     Hazard On Shoulder       Unclassified
485         Hazard On Shoulder Car Stopped       Unclassified
854                         Hazard Weather       Unclassified
857                   Hazard Weather Flood       Unclassified
858                      Jam Heavy Traffic       Unclassified
1122                 Jam Moderate Traffic       Unclassified
1184               Jam Stand Still Traffic       Unclassified
1335                     Road Closed Event       Unclassified
1905            Hazard On Road Lane Closed       Unclassified
5557                    Hazard Weather Fog       Unclassified
7331              Road Closed Construction       Unclassified
21443             Hazard On Road Road Kill       Unclassified
21447           Hazard On Shoulder Animals       Unclassified
21940      Hazard On Shoulder Missing Sign       Unclassified
38546                    Jam Light Traffic       Unclassified
44216            Hazard Weather Heavy Snow       Unclassified
54556                    Road Closed Hazard       Unclassified
229005                 Hazard Weather Hail       Unclassified

Updated crosswalk saved to
'/Users/attaullah/Documents/PS-6-/waze_data/updated_crosswalk.csv'
```

3.

```
# Step 1: Merge the crosswalk with the original dataset
merged_data = pd.merge(
    waze_full_data,             # Original dataset
    crosswalk,                  # Crosswalk DataFrame
    on=['type', 'subtype'],     # Merge on `type` and `subtype`
    how='left'                  # Keep all rows from the original dataset
)

# Step 2: Count rows for `Accident - Unclassified`
accident_unclassified_count = merged_data[
    (merged_data['updated_type'] == 'Accident') &
    (merged_data['updated_subtype'] == 'Unclassified')
].shape[0]

# Output the result
print(f"Number of rows for Accident - Unclassified: {accident_unclassified_count}")
```

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

4.

```
# Step 1: Extract unique type-subtype combinations from the crosswalk
crosswalk_combinations = crosswalk[['type', 'subtype']].drop_duplicates()

# Step 2: Extract unique type-subtype combinations from the merged dataset
merged_combinations = merged_data[['type', 'subtype']].drop_duplicates()

# Step 3: Compare the two sets of combinations
missing_in_merged = crosswalk_combinations.merge(merged_combinations, on=['type',
 ↪  'subtype'], how='left', indicator=True)
missing_in_merged = missing_in_merged[missing_in_merged['_merge'] == 'left_only']

missing_in_crosswalk = merged_combinations.merge(crosswalk_combinations, on=['type',
 ↪  'subtype'], how='left', indicator=True)
missing_in_crosswalk = missing_in_crosswalk[missing_in_crosswalk['_merge'] ==
 ↪  'left_only']

# Step 4: Output results
if missing_in_merged.empty and missing_in_crosswalk.empty:
    print("All type-subtype combinations match between the crosswalk and the merged
     ↪  dataset.")
else:
    print("Discrepancies found:")
    if not missing_in_merged.empty:
        print("\nCombinations in the crosswalk but not in the merged dataset:")
        print(missing_in_merged)
    if not missing_in_crosswalk.empty:
        print("\nCombinations in the merged dataset but not in the crosswalk:")
        print(missing_in_crosswalk)
```

```
All type-subtype combinations match between the crosswalk and the merged dataset.
```

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

1.

a.

```
import re
# Load the Waze data
waze_data =
 ↪  pd.read_csv('/Users/attaullah/Documents/PS-6-/waze_data/processed_waze_data.csv')

# Define a function to extract latitude and longitude with improved regex
def extract_coordinates(geo_string):
    """
    Extract latitude and longitude from WKT formatted geo string.
    Example WKT string: "POINT(-87.629798 41.878114)"
    """
    match = re.search(r'POINT\s*\(\s*(-?\d+\.\d+)\s+(-?\d+\.\d+)\s*\)', geo_string)
    if match:
        lon, lat = match.groups()  # Group 1 is longitude, Group 2 is latitude
        return float(lat), float(lon)
    return None, None

# Apply the function
waze_data['latitude'], waze_data['longitude'] =
 ↪  zip(*waze_data['geo'].map(extract_coordinates))

# Preview the updated DataFrame
print(waze_data[['geo', 'latitude', 'longitude']].head())

# Save the corrected data
waze_data.to_csv('/Users/attaullah/Documents/PS-6-/waze_data/processed_waze_data_fixed.csv',
 ↪  index=False)
print("\nCorrected data saved to
 ↪  '/Users/attaullah/Documents/PS-6-/waze_data/processed_waze_data_fixed.csv'")
```

```
                         geo   latitude  longitude
0  POINT(-87.676685 41.929692)  41.929692 -87.676685
1  POINT(-87.624816 41.753358)  41.753358 -87.624816
2  POINT(-87.614122 41.889821)  41.889821 -87.614122
3  POINT(-87.680139 41.939093)  41.939093 -87.680139
4   POINT(-87.735235 41.91658)  41.916580 -87.735235

Corrected data saved to
'/Users/attaullah/Documents/PS-6-/waze_data/processed_waze_data_fixed.csv'
```

b.

```
# Load the processed data
data_path = '/Users/attaullah/Documents/PS-6-/waze_data/processed_waze_data_fixed.csv'
waze_data = pd.read_csv(data_path)

# Step 1: Bin latitude and longitude into intervals of 0.01
waze_data['binned_latitude'] = np.floor(waze_data['latitude'] * 100) / 100
```

```
waze_data['binned_longitude'] = np.floor(waze_data['longitude'] * 100) / 100

# Step 2: Combine binned latitude and longitude into a single column for grouping
waze_data['binned_coordinates'] = list(zip(waze_data['binned_latitude'],
 ↪  waze_data['binned_longitude']))

# Step 3: Count occurrences for each binned coordinate
binned_counts = waze_data['binned_coordinates'].value_counts()

# Step 4: Find the binned coordinate with the highest number of observations
most_common_bin = binned_counts.idxmax()
max_count = binned_counts.max()

# Output the results
print(f"The binned latitude-longitude combination with the greatest number of
 ↪  observations is: {most_common_bin}")
print(f"Number of observations for this combination: {max_count}")
```

```
The binned latitude-longitude combination with the greatest number of observations is:
(41.96, -87.75)
Number of observations for this combination: 26540
```

c.

```
# Step 1: Filter for a specific type and subtype (replace with desired values)
chosen_type = "JAM"  # Example type
chosen_subtype = "JAM_HEAVY_TRAFFIC"  # Example subtype

filtered_data = waze_data[
    (waze_data['type'] == chosen_type) &
    (waze_data['subtype'] == chosen_subtype)
]

# Step 2: Group by binned latitude and longitude
collapsed_data = (
    filtered_data
    .groupby(['binned_latitude', 'binned_longitude'])
    .size()
    .reset_index(name='alert_count')  # Adds a column named 'alert_count'
)

# Step 3: Sort by alert count and select the top 10
top_10_bins = collapsed_data.nlargest(10, 'alert_count')

# Step 4: Save the result to a CSV file
output_path = '/Users/attaullah/Documents/PS-6-/waze_data/top_alerts_map.csv'
top_10_bins.to_csv(output_path, index=False)

# Output the results
print(top_10_bins)
print(f"\nTop 10 data saved to {output_path}")
print(f"\nLevel of aggregation: binned_latitude and binned_longitude")
print(f"Number of rows in the collapsed DataFrame: {collapsed_data.shape[0]}")
```

```
     binned_latitude  binned_longitude  alert_count
382            41.89            -87.66         4990
349            41.87            -87.65         4122
401            41.90            -87.67         3845
518            41.96            -87.75         3362
366            41.88            -87.65         3263
479            41.94            -87.72         3177
517            41.96            -87.76         3013
542            41.97            -87.77         2900
459            41.93            -87.71         2732
421            41.91            -87.67         2613

Top 10 data saved to /Users/attaullah/Documents/PS-6-/waze_data/top_alerts_map.csv

Level of aggregation: binned_latitude and binned_longitude
Number of rows in the collapsed DataFrame: 621
```

2.

```
import altair as alt
import pandas as pd

# Load the top 10 data
top_10_path = '/Users/attaullah/Documents/PS-6-/waze_data/top_alerts_map.csv'
top_10_bins = pd.read_csv(top_10_path)

# Create a scatter plot using Altair
scatter_plot = alt.Chart(top_10_bins).mark_circle(size=100).encode(
    x=alt.X('binned_longitude:Q', title='Longitude', scale=alt.Scale(domain=[-87.8,
 ↪  -87.6])),
    y=alt.Y('binned_latitude:Q', title='Latitude', scale=alt.Scale(domain=[41.85,
 ↪  42.0])),
    size=alt.Size('alert_count:Q', title='Number of Alerts'),
    tooltip=['binned_latitude', 'binned_longitude', 'alert_count']  # Add tooltips for
 ↪  interactivity
).properties(
    title="Top 10 Latitude-Longitude Bins for 'Jam - Heavy Traffic' Alerts",
    width=800,  # Customize plot width
    height=600   # Customize plot height
)

# Save the scatter plot as a PNG file
output_path = '/Users/attaullah/Documents/PS-6-/top_10_scatter_plot.png'
scatter_plot.save(output_path)

print(f"Scatter plot saved to {output_path}")
```

```
Scatter plot saved to /Users/attaullah/Documents/PS-6-/top_10_scatter_plot.png
```

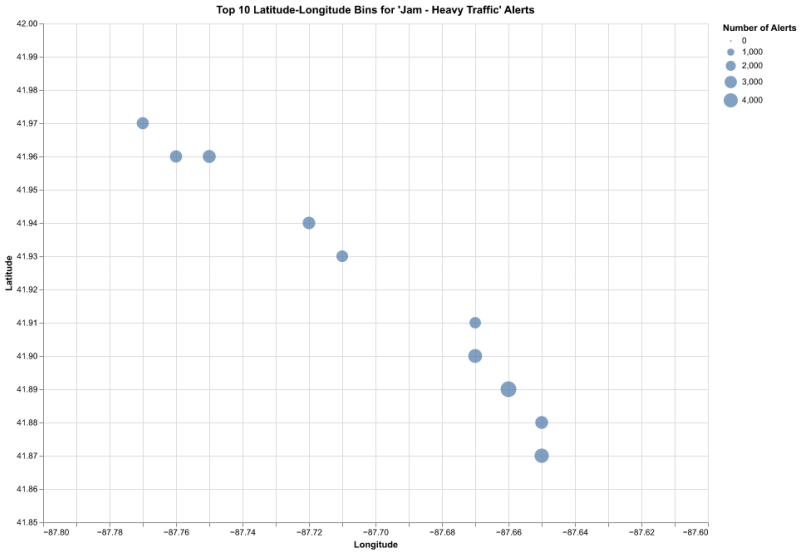


Figure 1: Top 10 Scatter Plot

3.

a.

```
# Define the path to the GeoJSON file
geojson_path = '/Users/attaullah/Documents/PS-6-/Boundaries - Neighborhoods.geojson'

# Load the GeoJSON data
with open(geojson_path, 'r') as f:
    chicago_geojson = json.load(f)

# Prepare GeoJSON data for Altair
geo_data = alt.Data(values=chicago_geojson["features"])

# Print success message
print("Chicago neighborhood boundaries loaded successfully!")
```

```
Chicago neighborhood boundaries loaded successfully!
```

b.

```
import altair as alt
import json

# Path to the GeoJSON file
file_path = '/Users/attaullah/Documents/PS-6-/Boundaries - Neighborhoods.geojson'

# Load the GeoJSON file
with open(file_path, 'r') as f:
    chicago_geojson = json.load(f)

# Prepare GeoJSON data for Altair
geo_data = alt.Data(values=chicago_geojson["features"])
```

```
# Create a basic map of Chicago using the GeoJSON data
chicago_map = alt.Chart(geo_data).mark_geoshape(
    fill='lightgray',  # Map fill color
    stroke='white'     # Map boundary stroke color
).project(
    type='equirectangular'  # Projection for Chicago area
).properties(
    title="Chicago Neighborhood Map",
    width=800,              # Map width
    height=600              # Map height
)

# Save the map as a PNG file
output_path = '/Users/attaullah/Documents/PS-6-/chicago_map.png'
chicago_map.save(output_path)

print(f"Map saved to {output_path}")
```

```
Map saved to /Users/attaullah/Documents/PS-6-/chicago_map.png
```


Figure 2: Chicago Map

4.

```
import altair as alt

# Combine the scatter plot with the Chicago map
final_map = (
    chicago_map +
    scatter_plot  # Reuse the previously defined scatter plot
).configure_view(
    stroke=None  # Remove any stroke around the map for a clean look
```

```
).properties(
    title="Top 10 Alert Locations for 'Jam - Heavy Traffic'",
    width=800,  # Consistent width for the map
    height=600  # Consistent height for the map
)

# Save the final layered map as a PNG file
output_path = '/Users/attaullah/Documents/PS-6-/final_map.png'
final_map.save(output_path)

print(f"Final layered map saved to {output_path}")
```

```
Final layered map saved to /Users/attaullah/Documents/PS-6-/final_map.png
```

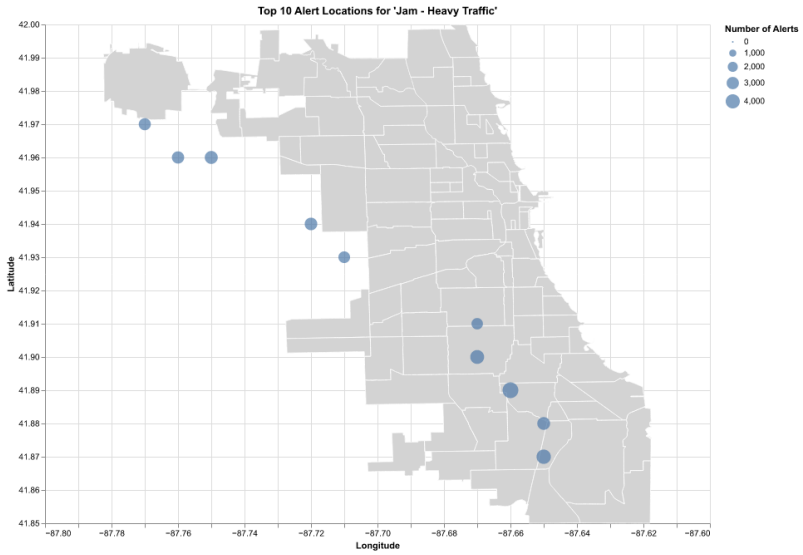


Figure 3: Final Layered Map

5.

a.

```
from shiny import ui, App, render, run_app
import os

# Load your processed data
data_path = '/Users/attaullah/Documents/PS-6-/waze_data/processed_waze_data_fixed.csv'
waze_data = pd.read_csv(data_path)

# Ensure valid coordinates
waze_data = waze_data[
    (waze_data['latitude'] >= 41.6) & (waze_data['latitude'] <= 42.1) &
    (waze_data['longitude'] >= -87.9) & (waze_data['longitude'] <= -87.5)
]

# Create unique type x subtype combinations
waze_data['type_subtype'] = waze_data['type'] + " x " +
 ↪  waze_data['subtype'].fillna("Unclassified")
```

```
unique_combinations = sorted(waze_data['type_subtype'].unique())

# Generate dropdown choices as a dictionary
dropdown_choices = {combo: combo for combo in unique_combinations}

# Create the dropdown UI
app_ui = ui.page_fluid(
    ui.h2("Top Alert Locations in Chicago"),
    ui.input_select(
        "type_subtype",
        "Select Type and Subtype",
        choices=dropdown_choices,
        multiple=False
    ),
    ui.output_image("top_10_plot")  # Render the plot as an image
)
# Server logic for the Shiny app
def server(input, output, session):
    @output
    @render.image
    def top_10_plot():
        # Get the selected type and subtype from the dropdown
        selected_type_subtype = input.type_subtype()

        # Filter the data for the selected type x subtype
        filtered_data = waze_data[waze_data['type_subtype'] == selected_type_subtype]

        # Group by latitude and longitude and calculate alert counts
        top_10 = (
            filtered_data.groupby(["latitude", "longitude"])
            .size()
            .reset_index(name="count")
            .nlargest(10, "count")
        )

        # Create a scatter plot using Altair
        scatter_plot = alt.Chart(top_10).mark_circle().encode(
            x=alt.X('longitude:Q', title='Longitude', scale=alt.Scale(domain=[-87.9,
 ↪  -87.5])),
            y=alt.Y('latitude:Q', title='Latitude', scale=alt.Scale(domain=[41.6,
 ↪  42.1])),
            size=alt.Size('count:Q', title='Number of Alerts'),
            tooltip=['latitude', 'longitude', 'count']
        ).properties(
            title=f"Top 10 Alert Locations for {selected_type_subtype}",
            width=800,
            height=600
        )

        # Save the plot as a PNG file
        output_path = "/tmp/top_10_plot.png"
        scatter_plot.save(output_path)
```

```
        # Return the path to the PNG file
        return {"src": output_path, "width": 800, "height": 600}

# Create the Shiny app
app = App(app_ui, server)

# Start the Shiny app server
if __name__ == "__main__":
    run_app(app)
print(f"Total type x subtype combinations: {len(unique_combinations)}")
```

Total type x subtype combinations: 32

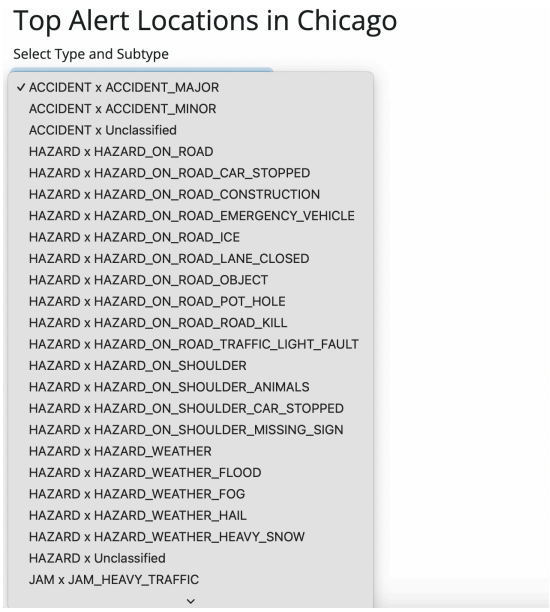


Figure 4: Dropdown Menu

b. **JAM Heavy Traffic:**

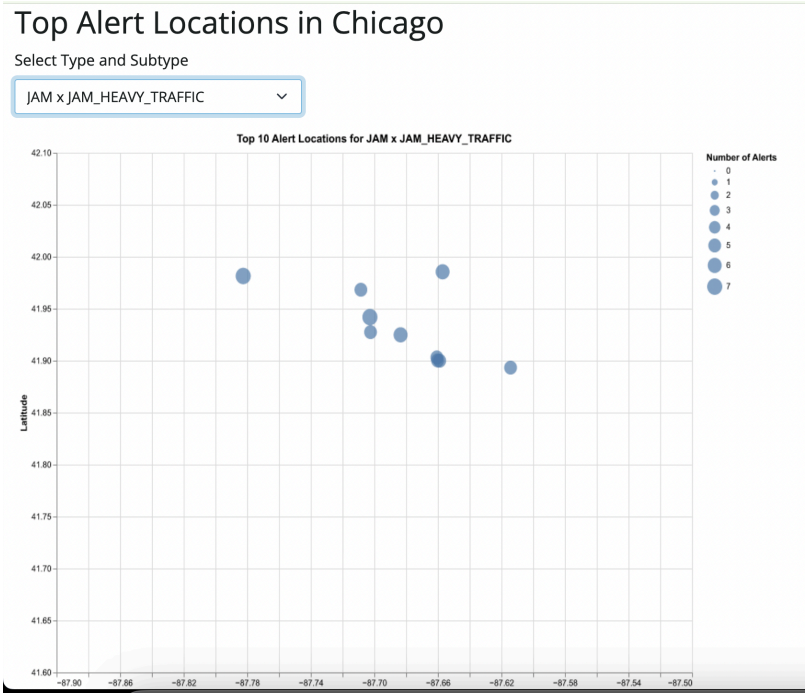


Figure 5: JAM_Heavy_Traffic

c. **Road Closed x Road Closed_Event :**

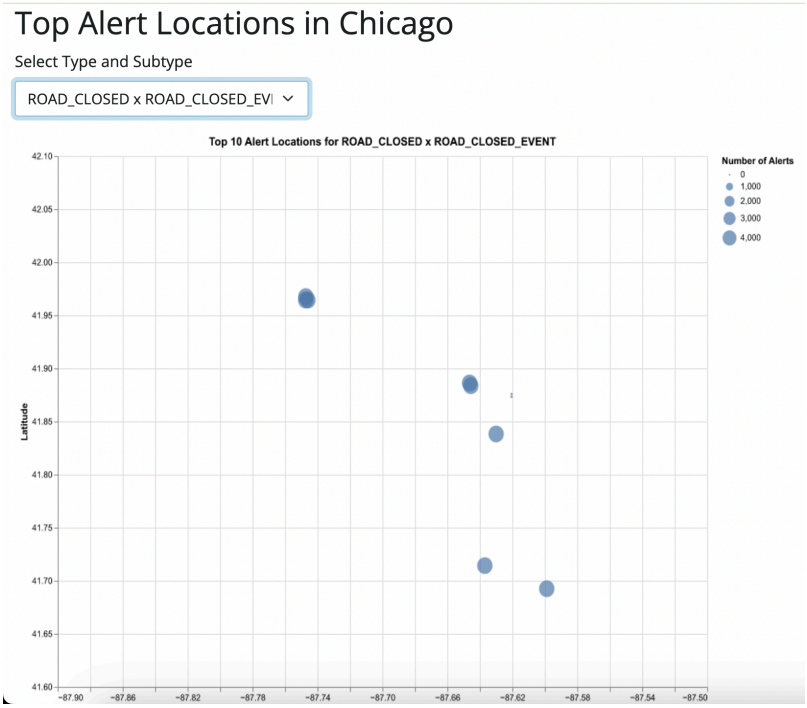


Figure 6: Road Closed_Event

d.

Question: "Where are the top locations in Chicago with the highest number of alerts for hazards caused

by weather conditions?"

Answer: Below is the resulting plot showcasing the top 10 locations for weather-related hazards:

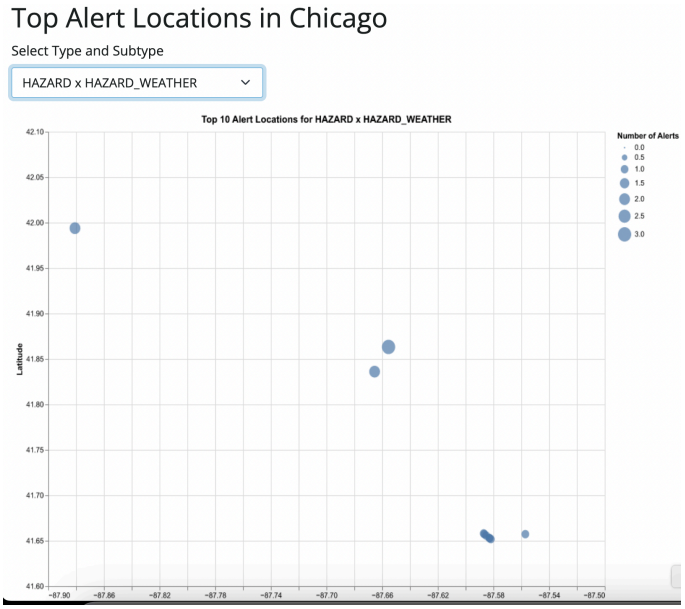


Figure 7: Hazard_Weather

e.

Adding a Time Dimension To improve the dashboard, I suggest adding a "Time of Alert" column. This would allow users to see when specific alerts, like road closures or accidents, are most frequent, offering insights into temporal patterns. For example, we could analyze peak traffic times or predict high-risk hours. This enhancement makes the dashboard more practical and actionable.

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

1.

a.

```
# Load the dataset
data_path = '/Users/attaullah/Documents/PS-6-/waze_data/processed_waze_data_fixed.csv'  #
 ↪  Adjust path if necessary
waze_data = pd.read_csv(data_path)

# Inspect the first few rows of the 'ts' column
print("First few rows of the 'ts' column:")
print(waze_data['ts'].head())

# Check the data type of the 'ts' column
print("\nData type of the 'ts' column:")
print(waze_data['ts'].dtype)

# Check for null values in the 'ts' column
```

```
print("\nNumber of null values in the 'ts' column:")
print(waze_data['ts'].isnull().sum())

# Get unique values and their count
print("\nNumber of unique values in the 'ts' column:")
print(waze_data['ts'].nunique())

# Check the first few unique values to assess granularity
print("\nSample unique values in the 'ts' column:")
print(waze_data['ts'].unique()[:10])

# Convert to datetime format and check the range
waze_data['ts'] = pd.to_datetime(waze_data['ts'], errors='coerce')  # Convert to
 ↪  datetime, handling errors
print("\nDatetime conversion - Check for errors (NaT values):")
print(waze_data['ts'].isnull().sum())

print("\nTime range in the 'ts' column:")
print(waze_data['ts'].min(), "to", waze_data['ts'].max())
```

```
First few rows of the 'ts' column:
0    2024-02-04 16:40:41 UTC
1    2024-02-04 20:01:27 UTC
2    2024-02-04 02:15:54 UTC
3    2024-02-04 00:30:54 UTC
4    2024-02-04 03:27:35 UTC
Name: ts, dtype: object

Data type of the 'ts' column:
object

Number of null values in the 'ts' column:
0

Number of unique values in the 'ts' column:
738674

Sample unique values in the 'ts' column:
['2024-02-04 16:40:41 UTC' '2024-02-04 20:01:27 UTC'
 '2024-02-04 02:15:54 UTC' '2024-02-04 00:30:54 UTC'
 '2024-02-04 03:27:35 UTC' '2024-02-04 19:19:40 UTC'
 '2024-02-04 00:07:40 UTC' '2024-02-04 06:49:29 UTC'
 '2024-02-04 12:50:43 UTC' '2024-02-04 16:20:04 UTC']

Datetime conversion - Check for errors (NaT values):
0

Time range in the 'ts' column:
2024-01-02 00:00:08+00:00 to 2024-10-12 23:58:54+00:00
```

Inspecting the `ts` Column and Collapsing the Dataset

After inspecting the `ts` column, I do not think it is a good idea to collapse the dataset solely based on this column. The `ts` column contains timestamps, which may have a high level of granularity (e.g., down to seconds or milliseconds). Collapsing the dataset by `ts` would likely result in many unique rows, making the dataset unnecessarily large and reducing its utility for analysis. Instead, aggregating data by broader time intervals (e.g., hourly or daily) would provide meaningful insights while keeping the dataset manageable.

b.

```
import os
# Load the full dataset
data_path = '/Users/attaullah/Documents/PS-6-/waze_data/processed_waze_data_fixed.csv'
waze_data = pd.read_csv(data_path)

# Step 1: Create a new variable `hour` by extracting the hour from the `ts` column
waze_data['hour'] = pd.to_datetime(waze_data['ts']).dt.floor('h')  # Use 'h' instead of
 ↪  'H'

# Step 2: Collapse the dataset by hour, type, subtype, latitude, and longitude
collapsed_data = (
    waze_data.groupby(['hour', 'type', 'subtype', 'latitude', 'longitude'])
    .size()
    .reset_index(name='alert_count')  # Adds a column named `alert_count`
)

# Step 3: Define the output directory and file path
output_dir = '/Users/attaullah/Documents/PS-6-/top_alerts_map_byhour'
output_path = os.path.join(output_dir, 'top_alerts_map_byhour.csv')

# Step 4: Ensure the directory exists
os.makedirs(output_dir, exist_ok=True)

# Step 5: Save the collapsed dataset
collapsed_data.to_csv(output_path, index=False)

# Step 6: Output the number of rows in the collapsed dataset
print(f"The collapsed dataset has {collapsed_data.shape[0]} rows.")
print(f"Collapsed dataset saved to: {output_path}")
```

```
The collapsed dataset has 648296 rows.
Collapsed dataset saved to:
/Users/attaullah/Documents/PS-6-/top_alerts_map_byhour/top_alerts_map_byhour.csv
```

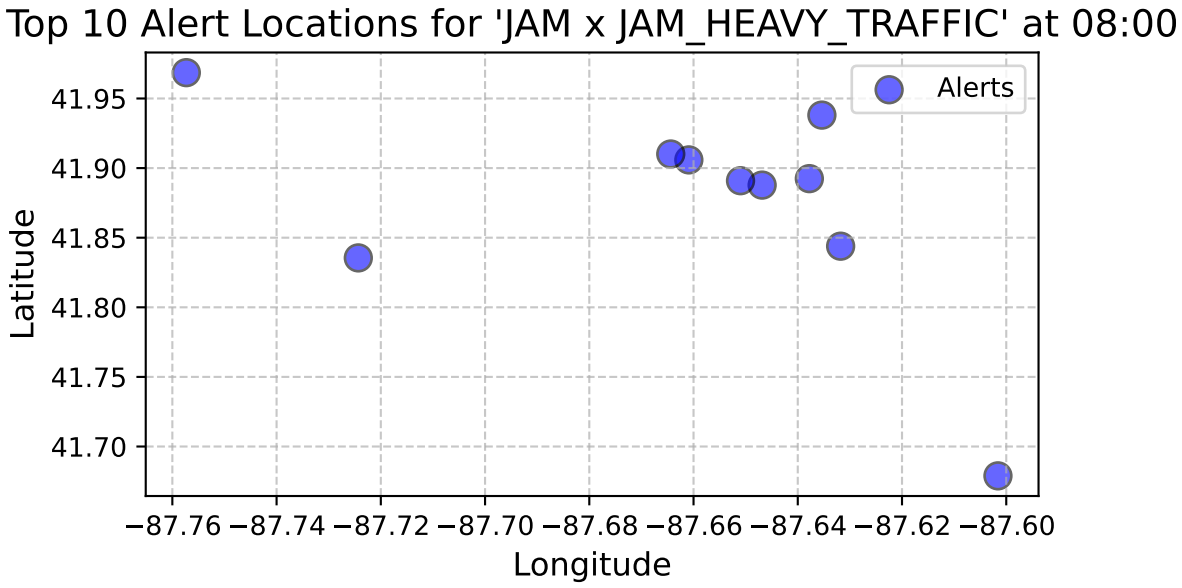
c.

```
import matplotlib.pyplot as plt

# Load the dataset
data_path =
 ↪  '/Users/attaullah/Documents/PS-6-/top_alerts_map_byhour/top_alerts_map_byhour.csv'
collapsed_data = pd.read_csv(data_path)

# Ensure the 'type_subtype' column exists
```

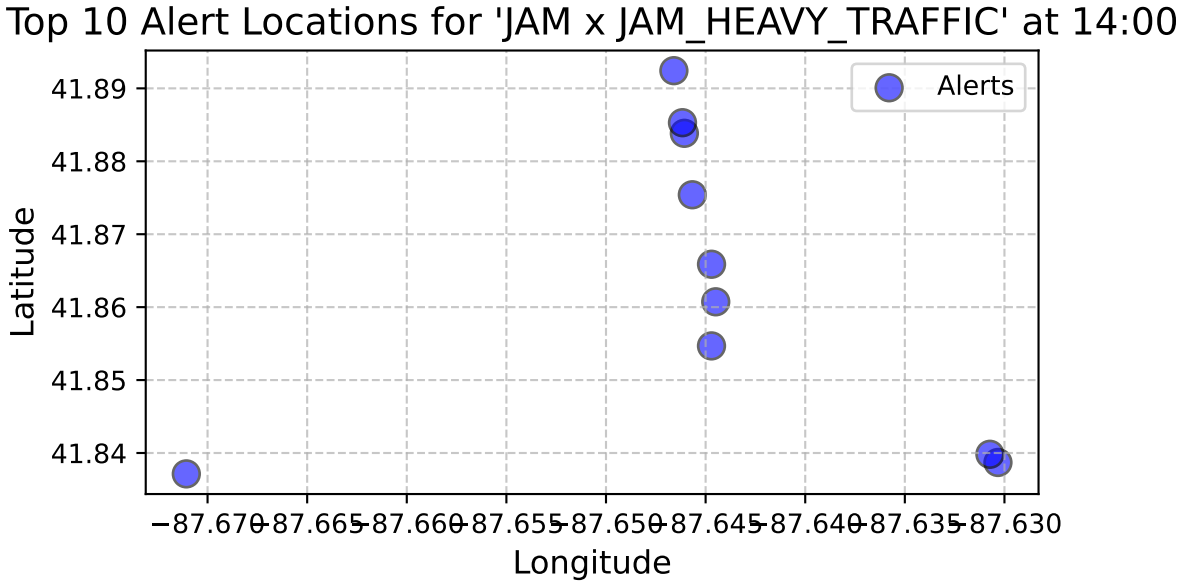
```
collapsed_data['type_subtype'] = collapsed_data['type'] + " x " +
 ↪  collapsed_data['subtype'].fillna("Unclassified")

# Extract only the hour portion from the 'hour' column
collapsed_data['hour_only'] = pd.to_datetime(collapsed_data['hour']).dt.strftime('%H:%M')

# Filter data for 'JAM x JAM_HEAVY_TRAFFIC'
alert_type = 'JAM x JAM_HEAVY_TRAFFIC'
filtered_data = collapsed_data[collapsed_data['type_subtype'] == alert_type]

# Check if data exists
if filtered_data.empty:
    print(f"No data available for '{alert_type}'.")
else:
    # Find available hours
    available_hours = filtered_data['hour_only'].unique()
    print("Available hours:", available_hours)

    # Use the specified time slots or dynamically select from available hours
    time_slots = ['08:00', '14:00', '20:00']

    # Create plots for the specified hours
    for time in time_slots:
        # Filter data for the current time slot
        time_data = filtered_data[filtered_data['hour_only'] == time]

        if time_data.empty:
            print(f"No data available for '{alert_type}' at {time}.")
            continue

        # Select the top 10 locations by alert count
        top_10 = time_data.nlargest(10, 'alert_count')

        # Plot the results
        plt.figure(figsize=(6, 3))
        plt.scatter(
            top_10['longitude'], top_10['latitude'],
            s=top_10['alert_count'] * 100, alpha=0.6,
            color='blue', edgecolors='black', label='Alerts'
        )
        plt.title(f"Top 10 Alert Locations for '{alert_type}' at {time}", fontsize=14)
        plt.xlabel('Longitude', fontsize=12)
        plt.ylabel('Latitude', fontsize=12)
        plt.grid(True, linestyle='--', alpha=0.7)
        plt.legend(fontsize=10)

        # Save the plot
        output_path = f"/Users/attaullah/Documents/PS-6-/jam_traffic_{time.replace(':',
 ↪  '')}.png"
        plt.savefig(output_path)
        plt.show()
        print(f"Plot for {time} saved at: {output_path}")
```

```
Available hours: ['10:00' '11:00' '12:00' '13:00' '14:00' '15:00' '16:00' '17:00' '18:00'
```

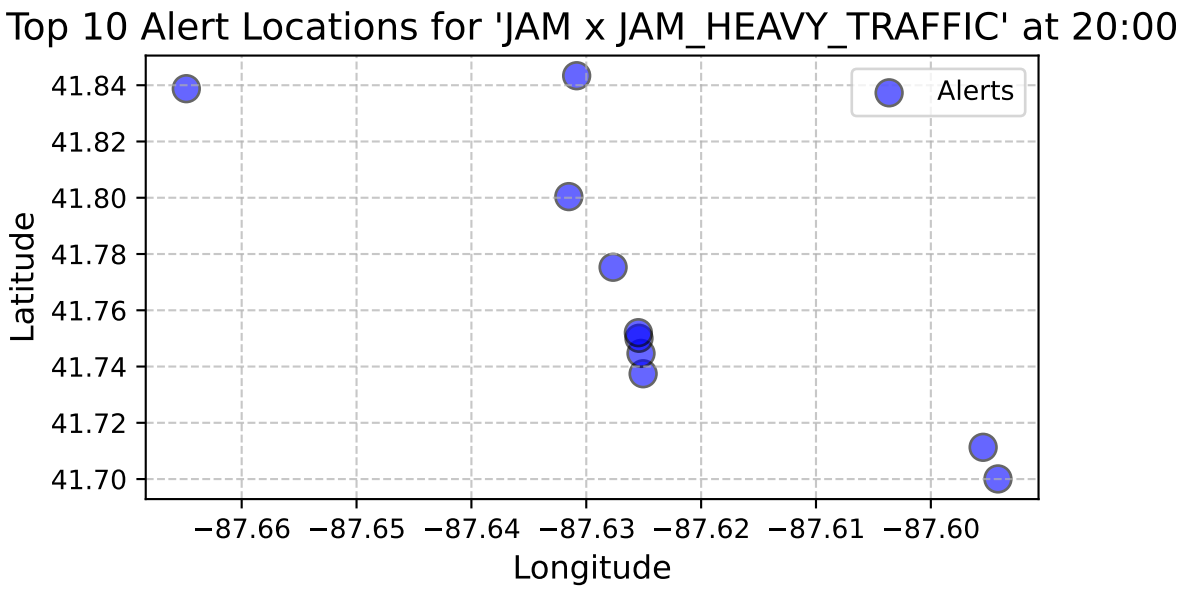
```
'19:00' '20:00' '21:00' '22:00' '23:00' '00:00' '01:00' '04:00' '05:00'
'06:00' '02:00' '07:00' '03:00' '08:00' '09:00']
```




```
Plot for 08:00 saved at: /Users/attaullah/Documents/PS-6-/jam_traffic_0800.png
```




```
Plot for 14:00 saved at: /Users/attaullah/Documents/PS-6-/jam_traffic_1400.png
```




```
Plot for 20:00 saved at: /Users/attaullah/Documents/PS-6-/jam_traffic_2000.png
```

2.

a.

```
from shiny import App, ui, render
import pandas as pd
import altair as alt
import os
import tempfile

# Load the collapsed dataset
data_path =
 ↪  "/Users/attaullah/Documents/PS-6-/top_alerts_map_byhour/top_alerts_map_byhour.csv"
collapsed_data = pd.read_csv(data_path)

# Add the 'type_subtype' column
collapsed_data["type_subtype"] = collapsed_data["type"] + " x " +
 ↪  collapsed_data["subtype"].fillna("Unclassified")

# Filter the data to ensure latitude and longitude are within valid bounds
latitude_bounds = [41.64, 42.02]
longitude_bounds = [-87.93, -87.55]

collapsed_data = collapsed_data[
    (collapsed_data["latitude"] >= latitude_bounds[0]) &
    (collapsed_data["latitude"] <= latitude_bounds[1]) &
    (collapsed_data["longitude"] >= longitude_bounds[0]) &
    (collapsed_data["longitude"] <= longitude_bounds[1])
]

# Ensure hour is in proper datetime format
collapsed_data["hour"] = pd.to_datetime(collapsed_data["hour"], errors="coerce")
```

```
# Get unique type_subtype combinations for the dropdown menu
unique_combinations = collapsed_data["type_subtype"].unique().tolist()

# Define the Shiny app UI
app_ui = ui.page_fluid(
    ui.h2("Top Alert Locations by Type, Subtype, and Hour"),
    ui.input_select(
        id="type_subtype",
        label="Select Type and Subtype",
        choices=unique_combinations,
        selected=unique_combinations[0] if unique_combinations else None
    ),
    ui.input_slider(
        id="hour",
        label="Select Hour",
        min=0,
        max=23,
        value=12,
        step=1,
        ticks=True
    ),
    ui.output_image("alert_plot")  # Render the plot as an image
)

# Define the server logic for the Shiny app
def server(input, output, session):
    @output
    @render.image
    def alert_plot():
        # Filter the data based on user input
        filtered_data = collapsed_data[
            (collapsed_data["type_subtype"] == input.type_subtype()) &
            (collapsed_data["hour"].dt.hour == input.hour())
        ]

        # Get the Top 10 locations by alert count
        top_10_data = filtered_data.nlargest(10, "alert_count")

        # Handle cases where no data is available
        if top_10_data.empty:
            return None

        # Create the scatter plot using Altair
        chart = alt.Chart(top_10_data).mark_circle().encode(
            x=alt.X("longitude:Q", title="Longitude",
 ↪  scale=alt.Scale(domain=longitude_bounds)),
            y=alt.Y("latitude:Q", title="Latitude",
 ↪  scale=alt.Scale(domain=latitude_bounds)),
            size="alert_count:Q",
            tooltip=["latitude", "longitude", "alert_count"]
        ).properties(
            width=700,
```

```
            height=500,
            title=f"Top 10 Alert Locations for '{input.type_subtype()}' at
↪   {input.hour()}:00"
        )

        # Save the chart as a temporary PNG file
        temp_file = tempfile.NamedTemporaryFile(suffix=".png", delete=False)
        chart.save(temp_file.name)

        return {
            "src": temp_file.name,
            "width": 700,
            "height": 500
        }

# Create the Shiny app
app = App(app_ui, server)

# Run the app
if __name__ == "__main__":
    app.run()
```

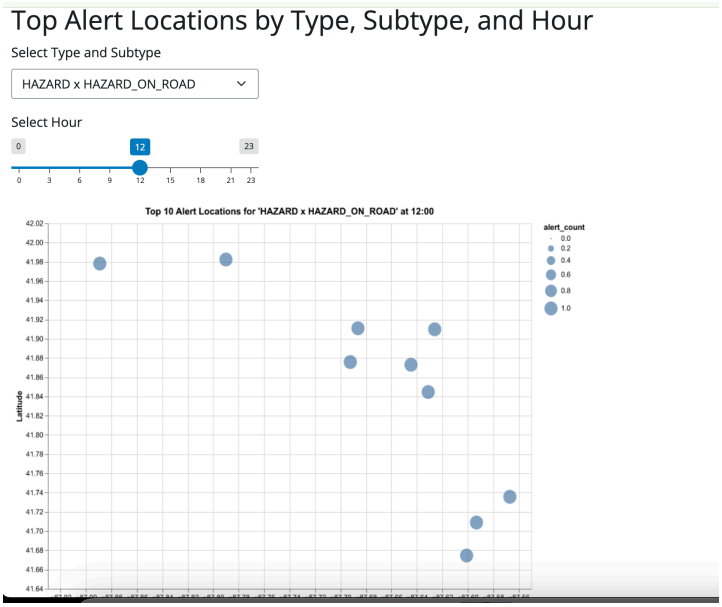
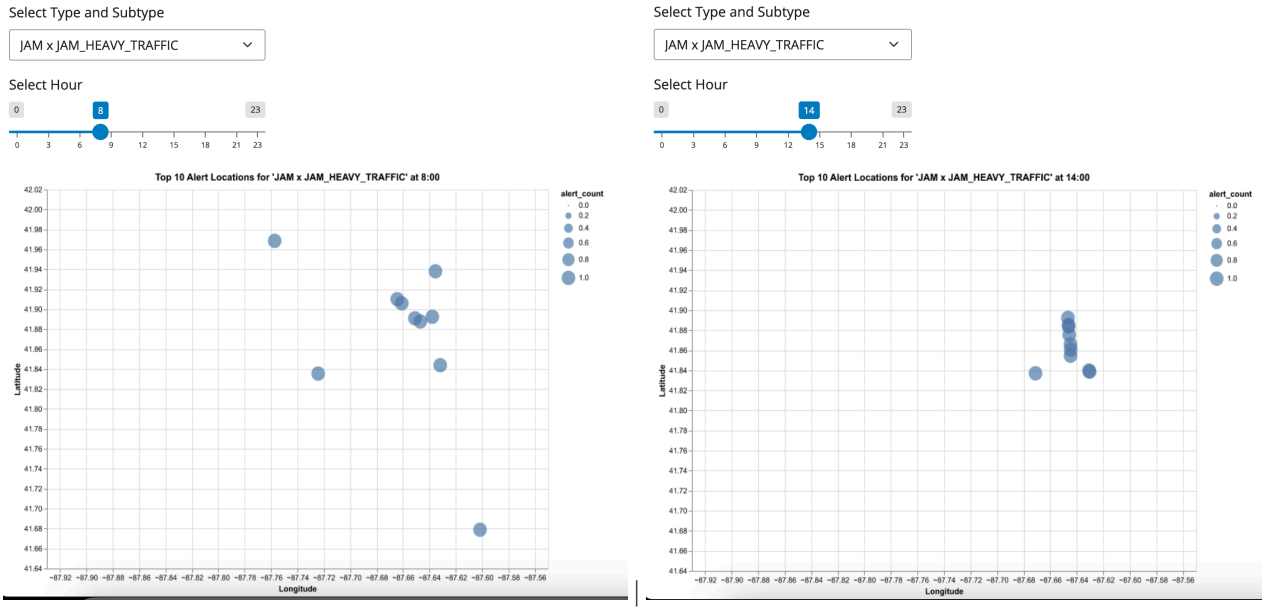


Figure 8: Slider

b.



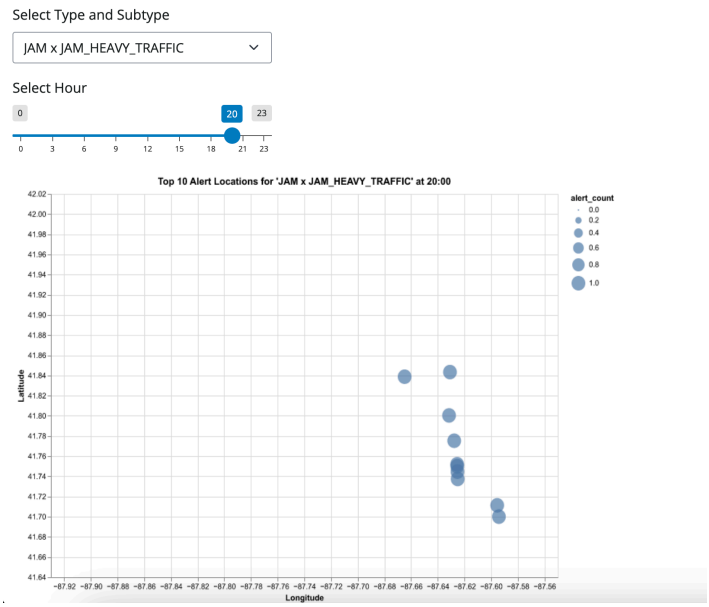
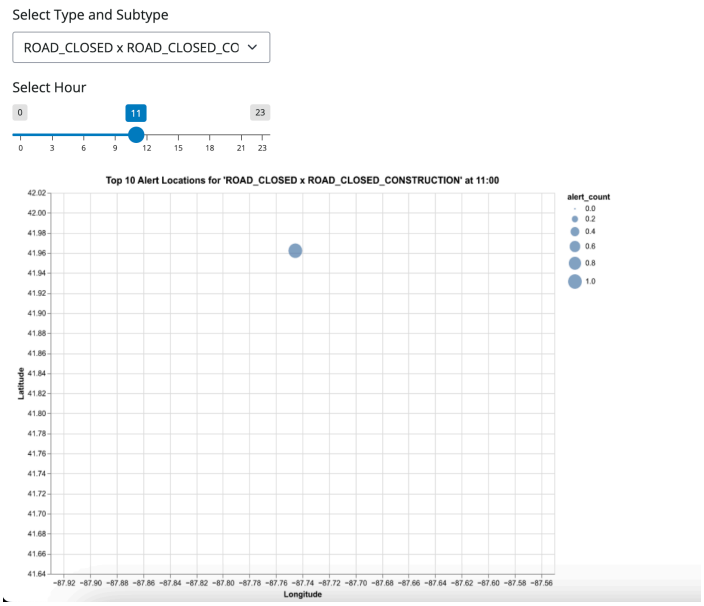
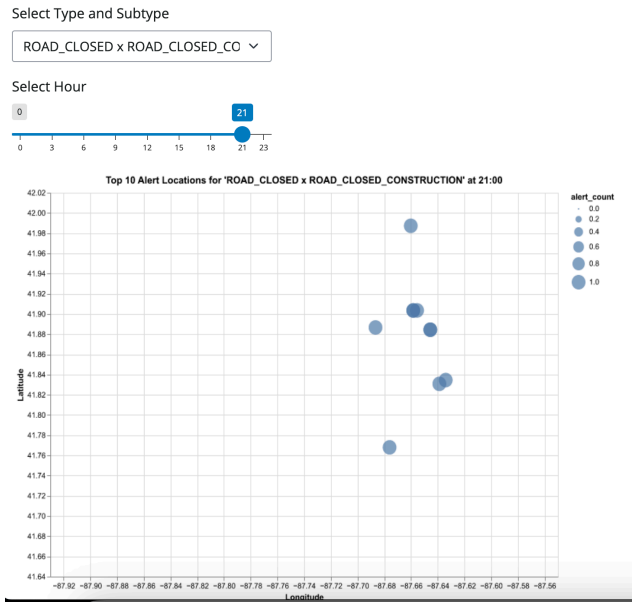


Figure 9: JAM_Heavy_Traffic_08pm

c.





Road construction appears to be more prevalent during night hours compared to morning hours.

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

1.

a.

Answer:

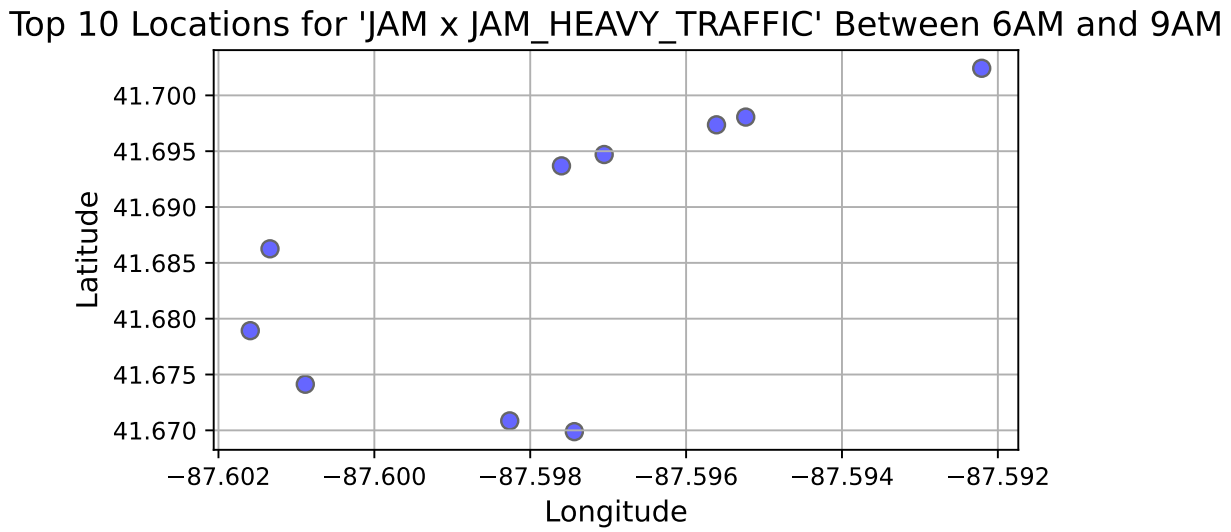
I don't think collapsing the dataset by range of hours is a good idea. Predefining ranges would limit the flexibility for users, as they wouldn't be able to select custom time ranges that fit their specific needs. It could also lead to data redundancy, as the same hour might fall into multiple ranges, making it harder to aggregate and interpret the data accurately.

Keeping the dataset at an hourly level, like in App 2, is better because it allows the app to dynamically filter for any range the user selects using the slider. This approach reduces the need for extra preprocessing while maintaining the app's interactivity and efficiency. It keeps the dataset clean and easy to handle.

b.

```
import pandas as pd
import matplotlib.pyplot as plt

# Load the collapsed dataset
data_path =
 ↪  "/Users/attaullah/Documents/PS-6-/top_alerts_map_byhour/top_alerts_map_byhour.csv"
collapsed_data = pd.read_csv(data_path)

# Ensure `type_subtype` column exists
collapsed_data["type_subtype"] = collapsed_data["type"] + " x " +
 ↪  collapsed_data["subtype"].fillna("Unclassified")
```

```
# Filter data for 'Jam - Heavy Traffic' and the time range 6 AM to 9 AM
alert_type = "JAM x JAM_HEAVY_TRAFFIC"
start_hour = 6
end_hour = 9

filtered_data = collapsed_data[
    (collapsed_data["type_subtype"] == alert_type) &
    (pd.to_datetime(collapsed_data["hour"]).dt.hour >= start_hour) &
    (pd.to_datetime(collapsed_data["hour"]).dt.hour < end_hour)
]

# Get the top 10 locations by alert count
top_10 = filtered_data.groupby(["latitude", "longitude"]).agg({"alert_count":
 ↪  "sum"}).reset_index()
top_10 = top_10.nlargest(10, "alert_count")

# Create the plot
plt.figure(figsize=(6, 3))
plt.scatter(
    top_10["longitude"],
    top_10["latitude"],
    s=top_10["alert_count"] * 50,  # Bubble size proportional to alert count
    c="blue", alpha=0.6, edgecolors="black"
)
plt.title(f"Top 10 Locations for '{alert_type}' Between 6AM and 9AM", fontsize=14)
plt.xlabel("Longitude", fontsize=12)
plt.ylabel("Latitude", fontsize=12)
plt.grid(True)
plt.show()
```



2.

a.

```
from shiny import App, ui, render
import pandas as pd
import altair as alt
import tempfile

# Load the collapsed dataset
data_path =
 ↪  "/Users/attaullah/Documents/PS-6-/top_alerts_map_byhour/top_alerts_map_byhour.csv"
collapsed_data = pd.read_csv(data_path)

# Add the 'type_subtype' column
collapsed_data["type_subtype"] = collapsed_data["type"] + " x " +
 ↪  collapsed_data["subtype"].fillna("Unclassified")

# Filter the data to ensure latitude and longitude are within valid bounds
latitude_bounds = [41.64, 42.02]
longitude_bounds = [-87.93, -87.55]

collapsed_data = collapsed_data[
    (collapsed_data["latitude"] >= latitude_bounds[0]) &
    (collapsed_data["latitude"] <= latitude_bounds[1]) &
    (collapsed_data["longitude"] >= longitude_bounds[0]) &
    (collapsed_data["longitude"] <= longitude_bounds[1])
]

# Ensure 'hour' is in datetime format
collapsed_data["hour"] = pd.to_datetime(collapsed_data["hour"], errors="coerce")

# Get unique type_subtype combinations for the dropdown menu
unique_combinations = collapsed_data["type_subtype"].unique().tolist()

# Define the Shiny app UI
app_ui = ui.page_fluid(
    ui.h2("Top Alert Locations by Type, Subtype, and Hour Range"),
    ui.input_select(
        id="type_subtype",
        label="Select Type and Subtype",
        choices=unique_combinations,
        selected=unique_combinations[0] if unique_combinations else None
    ),
    ui.input_slider(
        id="hour_range",
        label="Select Hour Range",
        min=0,
        max=23,
        value=[6, 9],  # Default range
        step=1,
        ticks=True
    ),
    ui.output_image("alert_plot")  # Render the plot as an image
)

# Define the server logic for the Shiny app
```

```python
def server(input, output, session):
    @output
    @render.image
    def alert_plot():
        # Filter the data based on user input
        filtered_data = collapsed_data[
            (collapsed_data["type_subtype"] == input.type_subtype()) &
            (collapsed_data["hour"].dt.hour >= input.hour_range()[0]) &
            (collapsed_data["hour"].dt.hour <= input.hour_range()[1])
        ]

        # Get the Top 10 locations by alert count
        top_10_data = filtered_data.nlargest(10, "alert_count")

        # Handle cases where no data is available
        if top_10_data.empty:
            return None

        # Create the scatter plot using Altair
        chart = alt.Chart(top_10_data).mark_circle().encode(
            x=alt.X("longitude:Q", title="Longitude",
 ↪  scale=alt.Scale(domain=longitude_bounds)),
            y=alt.Y("latitude:Q", title="Latitude",
 ↪  scale=alt.Scale(domain=latitude_bounds)),
            size="alert_count:Q",
            tooltip=["latitude", "longitude", "alert_count"]
        ).properties(
            width=700,
            height=500,
            title=f"Top 10 Alert Locations for '{input.type_subtype()}' Between
 ↪  {input.hour_range()[0]}:00 and {input.hour_range()[1]}:00"
        )

        # Save the chart as a temporary PNG file
        temp_file = tempfile.NamedTemporaryFile(suffix=".png", delete=False)
        chart.save(temp_file.name)

        return {
            "src": temp_file.name,
            "width": 700,
            "height": 500
        }

# Create the Shiny app
app = App(app_ui, server)

# Run the app
if __name__ == "__main__":
    app.run()
```

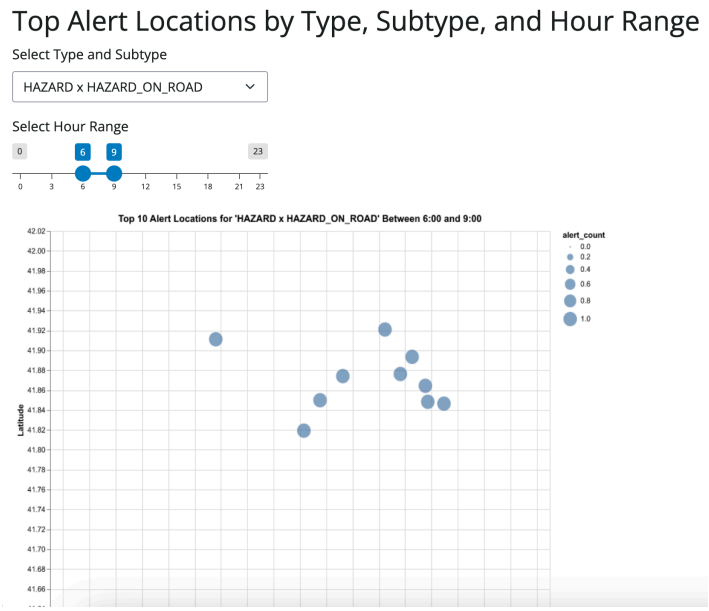


Figure 10: time_range

b.

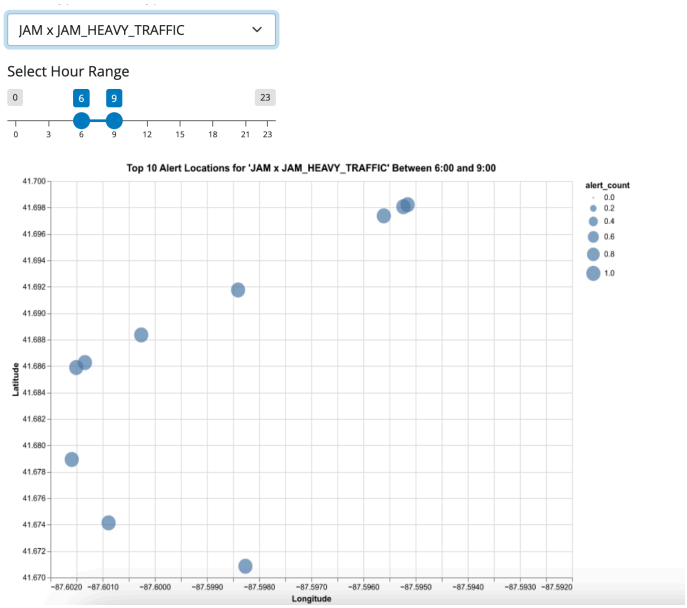


Figure 11: JAM_Heavy_Traffic

3.

a.

```
from shiny import App, ui, render

# Define the Shiny app UI
app_ui = ui.page_fluid(
```

```
    ui.h2("Top Alert Locations by Type, Subtype, and Hour"),
    ui.input_switch(
        id="switch_button",
        label="Toggle to switch to range of hours",
        value=False  # Default is OFF
    ),
    ui.output_text("display_switch_state")  # Display the state of the switch for testing
)

# Define the server logic
def server(input, output, session):
    @output
    @render.text
    def display_switch_state():
        # Show the current state of the switch button
        if input.switch_button():
            return "Switch is ON"
        else:
            return "Switch is OFF"

# Create the Shiny app
app = App(app_ui, server)

# Run the app
if __name__ == "__main__":
    app.run()
```

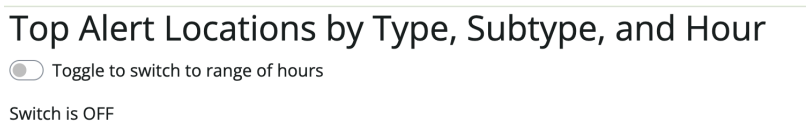


Figure 12: toggle

b.

```
# Load the collapsed dataset
data_path =
 ↪  "/Users/attaullah/Documents/PS-6-/top_alerts_map_byhour/top_alerts_map_byhour.csv"
collapsed_data = pd.read_csv(data_path)

# Add the 'type_subtype' column
collapsed_data["type_subtype"] = collapsed_data["type"] + " x " +
 ↪  collapsed_data["subtype"].fillna("Unclassified")

# Filter the data to ensure latitude and longitude are within valid bounds
latitude_bounds = [41.64, 42.02]
longitude_bounds = [-87.93, -87.55]

collapsed_data = collapsed_data[
    (collapsed_data["latitude"] >= latitude_bounds[0]) &
    (collapsed_data["latitude"] <= latitude_bounds[1]) &
```

```
    (collapsed_data["longitude"] >= longitude_bounds[0]) &
    (collapsed_data["longitude"] <= longitude_bounds[1])
]

# Ensure 'hour' is in datetime format
collapsed_data["hour"] = pd.to_datetime(collapsed_data["hour"], errors="coerce")

# Get unique type_subtype combinations for the dropdown menu
unique_combinations = collapsed_data["type_subtype"].unique().tolist()

# Define the Shiny app UI
app_ui = ui.page_fluid(
    ui.h2("Top Alert Locations by Type, Subtype, and Hour"),
    ui.input_switch("toggle_slider", "Toggle to switch to range of hours", value=False),
    ui.output_text_verbatim("toggle_state_display"),  # Display toggle state (for
 ↪  debugging)
    ui.panel_conditional(
        "input.toggle_slider == false",
        ui.input_slider(
            id="hour",
            label="Select Single Hour",
            min=0,
            max=23,
            value=12,
            step=1,
            ticks=True
        )
    ),
    ui.panel_conditional(
        "input.toggle_slider == true",
        ui.input_slider(
            id="hour_range",
            label="Select Hour Range",
            min=0,
            max=23,
            value=[6, 9],  # Default range
            step=1,
            ticks=True
        )
    )
)

# Define the server logic for the Shiny app
def server(input, output, session):
    @output
    @render.text
    def toggle_state_display():
        # Display the state of the toggle (for debugging purposes)
        return f"Switch is {'ON' if input.toggle_slider() else 'OFF'}"

# Create the Shiny app
app = App(app_ui, server)
```

```
# Run the app
if __name__ == "__main__":
    app.run()
```

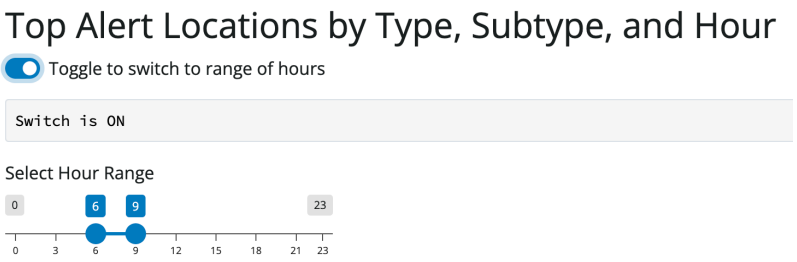


Figure 13: toggle_on

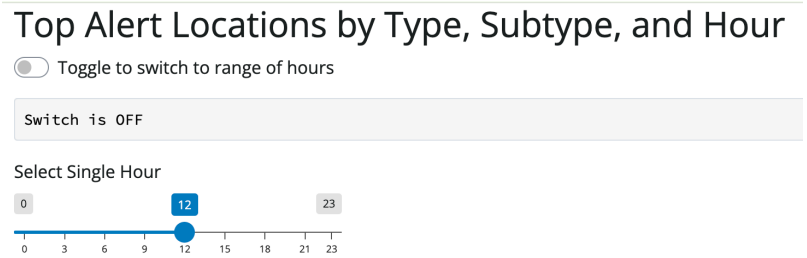


Figure 14: toggle_off

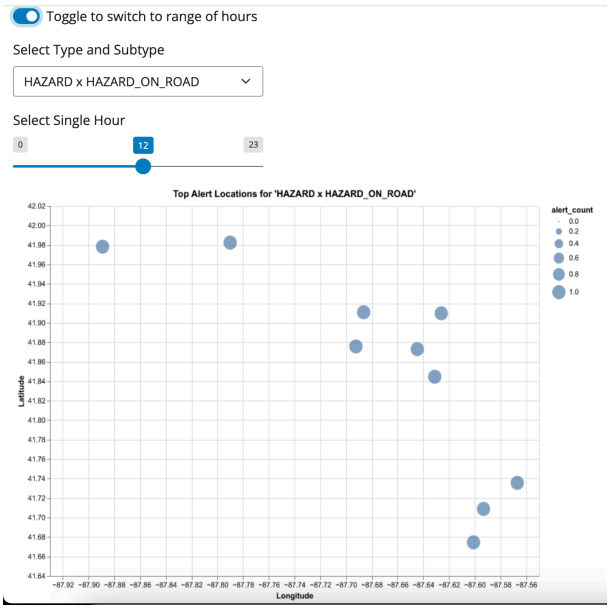
c.

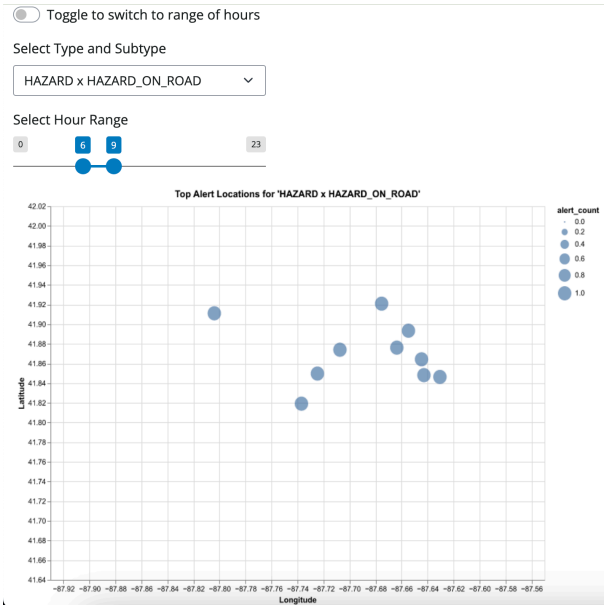
```
from shiny import App, ui, render
import pandas as pd
import altair as alt
import tempfile

# Load the collapsed dataset
data_path =
 ↪  "/Users/attaullah/Documents/PS-6-/top_alerts_map_byhour/top_alerts_map_byhour.csv"
collapsed_data = pd.read_csv(data_path)

# Add the 'type_subtype' column
collapsed_data["type_subtype"] = collapsed_data["type"] + " x " +
 ↪  collapsed_data["subtype"].fillna("Unclassified")

# Filter the data to ensure latitude and longitude are within valid bounds
latitude_bounds = [41.64, 42.02]
longitude_bounds = [-87.93, -87.55]

collapsed_data = collapsed_data[
    (collapsed_data["latitude"] >= latitude_bounds[0]) &
    (collapsed_data["latitude"] <= latitude_bounds[1]) &
    (collapsed_data["longitude"] >= longitude_bounds[0]) &
```

```
    (collapsed_data["longitude"] <= longitude_bounds[1])
]

# Ensure 'hour' is in datetime format
collapsed_data["hour"] = pd.to_datetime(collapsed_data["hour"], errors="coerce")

# Get unique type_subtype combinations for the dropdown menu
unique_combinations = collapsed_data["type_subtype"].unique().tolist()

# Define the Shiny app UI
app_ui = ui.page_fluid(
    ui.h2("Top Alert Locations by Type, Subtype, and Hour"),
    ui.input_switch(
        id="switch_button",
        label="Toggle to switch to range of hours",
        value=False  # Default is OFF
    ),
    ui.input_select(
        id="type_subtype",
        label="Select Type and Subtype",
        choices=unique_combinations,
        selected=unique_combinations[0] if unique_combinations else None
    ),
    ui.panel_conditional(
        "input.switch_button == false",
        ui.input_slider(
            id="hour_range",
            label="Select Hour Range",
            min=0,
            max=23,
            value=[6, 9],  # Default range
            step=1
        )
    ),
    ui.panel_conditional(
        "input.switch_button == true",
        ui.input_slider(
            id="single_hour",
            label="Select Single Hour",
            min=0,
            max=23,
            value=12,  # Default value
            step=1
        )
    ),
    ui.output_image("alert_plot")  # Render the plot as an image
)

# Define the server logic
def server(input, output, session):
    @output
    @render.image
    def alert_plot():
```

```
        # Get the selected type_subtype
        selected_type_subtype = input.type_subtype()

        if input.switch_button():
            # If the switch is ON, use the single hour slider
            selected_hour = input.single_hour()
            filtered_data = collapsed_data[
                (collapsed_data["type_subtype"] == selected_type_subtype) &
                (collapsed_data["hour"].dt.hour == selected_hour)
            ]
        else:
            # If the switch is OFF, use the hour range slider
            hour_range = input.hour_range()
            filtered_data = collapsed_data[
                (collapsed_data["type_subtype"] == selected_type_subtype) &
                (collapsed_data["hour"].dt.hour >= hour_range[0]) &
                (collapsed_data["hour"].dt.hour <= hour_range[1])
            ]

        # Get the Top 10 locations by alert count
        top_10_data = filtered_data.nlargest(10, "alert_count")

        # Handle cases where no data is available
        if top_10_data.empty:
            return None

        # Create the scatter plot using Altair
        chart = alt.Chart(top_10_data).mark_circle().encode(
            x=alt.X("longitude:Q", title="Longitude",
↪   scale=alt.Scale(domain=longitude_bounds)),
            y=alt.Y("latitude:Q", title="Latitude",
↪   scale=alt.Scale(domain=latitude_bounds)),
            size="alert_count:Q",
            tooltip=["latitude", "longitude", "alert_count"]
        ).properties(
            width=700,
            height=500,
            title=f"Top Alert Locations for '{selected_type_subtype}'"
        )

        # Save the chart as a temporary PNG file
        temp_file = tempfile.NamedTemporaryFile(suffix=".png", delete=False)
        chart.save(temp_file.name)

        return {
            "src": temp_file.name,
            "width": 700,
            "height": 500
        }

# Create the Shiny app
app = App(app_ui, server)
```

```
# Run the app
if __name__ == "__main__":
    app.run()
```





d. To achieve a plot like the one in the image, I'd need to make several updates to the app. First, I'd add a column to the dataset to classify hours into time periods like "Morning" (e.g., 6 AM to 11 AM) and "Afternoon" (e.g., 12 PM to 5 PM). This would allow us to group data by these categories instead of specific hours.

Next, I'd modify the app's logic to include a color legend for time periods. For example, I'd assign different colors, like red for "Morning" and blue for "Afternoon," to visually separate alerts based on time.

Finally, I'd integrate Chicago's neighborhood boundaries into the plot using a GeoJSON file. This would let the background map display clear, detailed outlines of the regions. The app would also need to dynamically adjust bubble sizes based on the total number of alerts during those time periods.

These changes would make the plot more detailed and provide clearer insights into spatial and temporal alert patterns.