SC1015 Lab REP2 Team 4 DSAI Mini-Project

Sricharan Balasubramanian, Travis Tan Hai Shuo, Joanna Wu Haoyue

In light of our upcoming exchange in UC Berkeley, the 3 of us are excited to go on many roadtrips to explore the beautiful state of California. However, we are concerned about the road safety there and want to know which streets are most prone to accidents, so we can be more cautious.

Upon finding the Kaggle dataset of "US Accidents" that contained data on road accidents all across US, we decided to scope down this dataset to only accidents that happened in the state of California. Coincidentally, with a quick visualisation of the dataset, we realised California is the state with the highest number of road accidents, which inspired a greater goal and use case.

Thus, in this project we will be using this dataset to find out the times of day or night where particular streets are most accident prone. We also want to find correlation between weather conditions and accidents. With these insights, emergency services like firefighters and hospitals can have an early warning system to help them allocate rescue resources optimally, reducing the fatality of road accidents. Additionally, these insights can be integrated into GPS systems to alert drivers when they are driving on certain streets at accident-prone times or weather conditions, minimising road accidents.

In [4]:
```
!pip install geopandas
!pip install contextily
!pip install shapely
!pip install folium
!pip install imbalanced-learn xgboost
!pip install pandoc
```

```
Requirement already satisfied: geopandas in c:\users\joanna\anaconda3\lib\si
te-packages (1.0.1)
Requirement already satisfied: numpy>=1.22 in c:\users\joanna\anaconda3\lib
\site-packages (from geopandas) (1.26.4)
Requirement already satisfied: pyogrio>=0.7.2 in c:\users\joanna\anaconda3\l
ib\site-packages (from geopandas) (0.10.0)
Requirement already satisfied: packaging in c:\users\joanna\anaconda3\lib\si
te-packages (from geopandas) (24.1)
Requirement already satisfied: pandas>=1.4.0 in c:\users\joanna\anaconda3\li
b\site-packages (from geopandas) (2.2.2)
Requirement already satisfied: pyproj>=3.3.0 in c:\users\joanna\anaconda3\li
b\site-packages (from geopandas) (3.7.1)
Requirement already satisfied: shapely>=2.0.0 in c:\users\joanna\anaconda3\l
ib\site-packages (from geopandas) (2.1.0)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\joanna\ana
conda3\lib\site-packages (from pandas>=1.4.0->geopandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in c:\users\joanna\anaconda3\lib
\site-packages (from pandas>=1.4.0->geopandas) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in c:\users\joanna\anaconda3\l
ib\site-packages (from pandas>=1.4.0->geopandas) (2023.3)
Requirement already satisfied: certifi in c:\users\joanna\anaconda3\lib\site
-packages (from pyogrio>=0.7.2->geopandas) (2025.1.31)
Requirement already satisfied: six>=1.5 in c:\users\joanna\anaconda3\lib\sit
e-packages (from python-dateutil>=2.8.2->pandas>=1.4.0->geopandas) (1.16.0)
Requirement already satisfied: contextily in c:\users\joanna\anaconda3\lib\s
ite-packages (1.6.2)
Requirement already satisfied: geopy in c:\users\joanna\anaconda3\lib\site-p
ackages (from contextily) (2.4.1)
Requirement already satisfied: matplotlib in c:\users\joanna\anaconda3\lib\s
ite-packages (from contextily) (3.9.2)
Requirement already satisfied: mercantile in c:\users\joanna\anaconda3\lib\s
ite-packages (from contextily) (1.2.1)
Requirement already satisfied: pillow in c:\users\joanna\anaconda3\lib\site-
packages (from contextily) (10.4.0)
Requirement already satisfied: rasterio in c:\users\joanna\anaconda3\lib\sit
e-packages (from contextily) (1.4.3)
Requirement already satisfied: requests in c:\users\joanna\anaconda3\lib\sit
e-packages (from contextily) (2.32.3)
Requirement already satisfied: joblib in c:\users\joanna\anaconda3\lib\site-
packages (from contextily) (1.4.2)
Requirement already satisfied: xyzservices in c:\users\joanna\anaconda3\lib
\site-packages (from contextily) (2022.9.0)
Requirement already satisfied: geographiclib<3,>=1.52 in c:\users\joanna\ana
conda3\lib\site-packages (from geopy->contextily) (2.0)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\joanna\anaconda3
\lib\site-packages (from matplotlib->contextily) (1.2.0)
Requirement already satisfied: cycler>=0.10 in c:\users\joanna\anaconda3\lib
\site-packages (from matplotlib->contextily) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\joanna\anaconda
3\lib\site-packages (from matplotlib->contextily) (4.51.0)
Requirement already satisfied: kiwisolver>=1.3.1 in c:\users\joanna\anaconda
3\lib\site-packages (from matplotlib->contextily) (1.4.4)
Requirement already satisfied: numpy>=1.23 in c:\users\joanna\anaconda3\lib
\site-packages (from matplotlib->contextily) (1.26.4)
Requirement already satisfied: packaging>=20.0 in c:\users\joanna\anaconda3
\lib\site-packages (from matplotlib->contextily) (24.1)
```

```
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\joanna\anaconda3
\lib\site-packages (from matplotlib->contextily) (3.1.2)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\joanna\anaco
nda3\lib\site-packages (from matplotlib->contextily) (2.9.0.post0)
Requirement already satisfied: click>=3.0 in c:\users\joanna\anaconda3\lib\s
ite-packages (from mercantile->contextily) (8.1.7)
Requirement already satisfied: affine in c:\users\joanna\anaconda3\lib\site-
packages (from rasterio->contextily) (2.4.0)
Requirement already satisfied: attrs in c:\users\joanna\anaconda3\lib\site-p
ackages (from rasterio->contextily) (23.1.0)
Requirement already satisfied: certifi in c:\users\joanna\anaconda3\lib\site
-packages (from rasterio->contextily) (2025.1.31)
Requirement already satisfied: cligj>=0.5 in c:\users\joanna\anaconda3\lib\s
ite-packages (from rasterio->contextily) (0.7.2)
Requirement already satisfied: click-plugins in c:\users\joanna\anaconda3\li
b\site-packages (from rasterio->contextily) (1.1.1)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\joanna\a
naconda3\lib\site-packages (from requests->contextily) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in c:\users\joanna\anaconda3\lib
\site-packages (from requests->contextily) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\joanna\anacond
a3\lib\site-packages (from requests->contextily) (2.2.3)
Requirement already satisfied: colorama in c:\users\joanna\anaconda3\lib\sit
e-packages (from click>=3.0->mercantile->contextily) (0.4.6)
Requirement already satisfied: six>=1.5 in c:\users\joanna\anaconda3\lib\sit
e-packages (from python-dateutil>=2.7->matplotlib->contextily) (1.16.0)
Requirement already satisfied: shapely in c:\users\joanna\anaconda3\lib\site
-packages (2.1.0)
Requirement already satisfied: numpy>=1.21 in c:\users\joanna\anaconda3\lib
\site-packages (from shapely) (1.26.4)
Requirement already satisfied: folium in c:\users\joanna\anaconda3\lib\site-
packages (0.19.5)
Requirement already satisfied: branca>=0.6.0 in c:\users\joanna\anaconda3\li
b\site-packages (from folium) (0.8.1)
Requirement already satisfied: jinja2>=2.9 in c:\users\joanna\anaconda3\lib
\site-packages (from folium) (3.1.4)
Requirement already satisfied: numpy in c:\users\joanna\anaconda3\lib\site-p
ackages (from folium) (1.26.4)
Requirement already satisfied: requests in c:\users\joanna\anaconda3\lib\sit
e-packages (from folium) (2.32.3)
Requirement already satisfied: xyzservices in c:\users\joanna\anaconda3\lib
\site-packages (from folium) (2022.9.0)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\joanna\anaconda3
\lib\site-packages (from jinja2>=2.9->folium) (2.1.3)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\joanna\a
naconda3\lib\site-packages (from requests->folium) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in c:\users\joanna\anaconda3\lib
\site-packages (from requests->folium) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\joanna\anacond
a3\lib\site-packages (from requests->folium) (2.2.3)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\joanna\anacond
a3\lib\site-packages (from requests->folium) (2025.1.31)
Requirement already satisfied: imbalanced-learn in c:\users\joanna\anaconda3
\lib\site-packages (0.12.3)
Requirement already satisfied: xgboost in c:\users\joanna\anaconda3\lib\site
-packages (3.0.0)
```

```
Requirement already satisfied: numpy>=1.17.3 in c:\users\joanna\anaconda3\li
b\site-packages (from imbalanced-learn) (1.26.4)
Requirement already satisfied: scipy>=1.5.0 in c:\users\joanna\anaconda3\lib
\site-packages (from imbalanced-learn) (1.13.1)
Requirement already satisfied: scikit-learn>=1.0.2 in c:\users\joanna\anacon
da3\lib\site-packages (from imbalanced-learn) (1.5.1)
Requirement already satisfied: joblib>=1.1.1 in c:\users\joanna\anaconda3\li
b\site-packages (from imbalanced-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in c:\users\joanna\anaco
nda3\lib\site-packages (from imbalanced-learn) (3.5.0)
Requirement already satisfied: pandoc in c:\users\joanna\anaconda3\lib\site-
packages (2.4)
Requirement already satisfied: plumbum in c:\users\joanna\anaconda3\lib\site
-packages (from pandoc) (1.9.0)
Requirement already satisfied: ply in c:\users\joanna\anaconda3\lib\site-pac
kages (from pandoc) (3.11)
Requirement already satisfied: pywin32 in c:\users\joanna\anaconda3\lib\site
-packages (from plumbum->pandoc) (305.1)
```

In [207…

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sb
from sklearn.preprocessing import MinMaxScaler
import time
import geopandas as gpd
import contextily as ctx
from shapely.geometry import Point
from matplotlib.ticker import MaxNLocator
import folium
from folium.plugins import HeatMap
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegresso
from sklearn.linear_model import Ridge, LogisticRegression
from tqdm import tqdm
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_scor
from imblearn.over_sampling import SMOTE
from sklearn.preprocessing import LabelEncoder
from xgboost import XGBClassifier
```

In [6]:

```
df = pd.read_csv('US_Accidents_March23_Cleaned.csv')
```

In [7]:

```
df.head(500)
```

Out[7]:

	ID	Source	Severity	Start_Time	End_Time	Start_Lat	Start_Lng	E
0	A-729	Source2	3	2016-06-21 10:34:40	2016-06-21 11:04:40	38.085300	-122.233017	
1	A-730	Source2	3	2016-06-21 10:30:16	2016-06-21 11:16:39	37.631813	-122.084167	
2	A-731	Source2	2	2016-06-21 10:49:14	2016-06-21 11:19:14	37.896564	-122.070717	
3	A-732	Source2	3	2016-06-21 10:41:42	2016-06-21 11:11:42	37.334255	-122.032471	
4	A-733	Source2	2	2016-06-21 10:16:26	2016-06-21 11:04:16	37.250729	-121.910713	
...	...	...	...	...	...	...	...	
495	A-1224	Source2	2	2016-06-25 04:58:12	2016-06-25 05:43:12	38.227760	-122.095024	
496	A-1225	Source2	3	2016-06-25 05:25:30	2016-06-25 06:10:30	37.847965	-122.027657	
497	A-1226	Source2	2	2016-06-25 04:57:48	2016-06-25 05:42:48	37.364319	-121.901840	
498	A-1227	Source2	3	2016-06-25 05:52:57	2016-06-25 06:52:57	37.836437	-122.011444	
499	A-1228	Source2	2	2016-06-25 06:17:03	2016-06-25 07:02:03	37.145508	-121.984970	

500 rows × 46 columns

In [8]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1741433 entries, 0 to 1741432
Data columns (total 46 columns):
 #   Column                 Dtype  
---  ------                 -----  
 0   ID                     object 
 1   Source                 object 
 2   Severity               int64  
 3   Start_Time             object 
 4   End_Time               object 
 5   Start_Lat              float64
 6   Start_Lng              float64
 7   End_Lat                float64
 8   End_Lng                float64
 9   Distance(mi)           float64
 10  Description            object 
 11  Street                 object 
 12  City                   object 
 13  County                 object 
 14  State                  object 
 15  Zipcode                object 
 16  Country                object 
 17  Timezone               object 
 18  Airport_Code           object 
 19  Weather_Timestamp      object 
 20  Temperature(F)         float64
 21  Wind_Chill(F)          float64
 22  Humidity(%)            float64
 23  Pressure(in)           float64
 24  Visibility(mi)         float64
 25  Wind_Direction         object 
 26  Wind_Speed(mph)        float64
 27  Precipitation(in)      float64
 28  Weather_Condition      object 
 29  Amenity                bool   
 30  Bump                   bool   
 31  Crossing               bool   
 32  Give_Way               bool   
 33  Junction               bool   
 34  No_Exit                bool   
 35  Railway                bool   
 36  Roundabout             bool   
 37  Station                bool   
 38  Stop                   bool   
 39  Traffic_Calming        bool   
 40  Traffic_Signal         bool   
 41  Turning_Loop           bool   
 42  Sunrise_Sunset         object 
 43  Civil_Twilight         object 
 44  Nautical_Twilight      object 
 45  Astronomical_Twilight  object 
dtypes: bool(13), float64(12), int64(1), object(20)
memory usage: 460.0+ MB
```

In [9]:
```
df.nunique()
```

Out[9]:
```
ID                        1741433
Source                          3
Severity                        4
Start_Time                1394898
End_Time                  1555839
Start_Lat                  498509
Start_Lng                  501661
End_Lat                    350772
End_Lng                    356821
Distance(mi)                13409
Description                811862
Street                      67370
City                         1268
County                         58
State                           1
Zipcode                    129022
Country                         1
Timezone                        2
Airport_Code                  142
Weather_Timestamp          422496
Temperature(F)                570
Wind_Chill(F)                 402
Humidity(%)                   100
Pressure(in)                  834
Visibility(mi)                 63
Wind_Direction                 24
Wind_Speed(mph)               121
Precipitation(in)              91
Weather_Condition              87
Amenity                         2
Bump                            2
Crossing                        2
Give_Way                        2
Junction                        2
No_Exit                         2
Railway                         2
Roundabout                      2
Station                         2
Stop                            2
Traffic_Calming                 2
Traffic_Signal                  2
Turning_Loop                    1
Sunrise_Sunset                  2
Civil_Twilight                  2
Nautical_Twilight               2
Astronomical_Twilight           2
dtype: int64
```

In [10]:
```
df.describe(include='all')
```

Out[10]:

	ID	Source	Severity	Start_Time	End_Time	Start_Lat
count	1741433	1741433	1.741433e+06	1741433	1741433	1.741433e+06
unique	1741433	3	NaN	1394898	1555839	NaN
top	A-729	Source1	NaN	2022-04-26 16:14:30	2019-10-17 18:07:45	NaN
freq	1	1104102	NaN	54	31	NaN
mean	NaN	NaN	2.165688e+00	NaN	NaN	3.563026e+01
std	NaN	NaN	4.068822e-01	NaN	NaN	2.093458e+00
min	NaN	NaN	1.000000e+00	NaN	NaN	3.254259e+01
25%	NaN	NaN	2.000000e+00	NaN	NaN	3.397552e+01
50%	NaN	NaN	2.000000e+00	NaN	NaN	3.423644e+01
75%	NaN	NaN	2.000000e+00	NaN	NaN	3.770239e+01
max	NaN	NaN	4.000000e+00	NaN	NaN	4.200542e+01

11 rows × 46 columns

Data Pre-processing

We will prepare the data by performing data cleaning to suit our problem statement. This begins with feature reduction on the California dataset, to improve model performance and reduce complexity.

We will eliminate columns that are not necessary for our analysis, such as columns with only 1 unique value (e.g. 'country' and 'state'columns, because all accidents happen in the same country and state). Likewise, for Turning_Loop where every value is False, there is no value added with purely negative data. We will also eliminate columns that are irrelevant to the problem, like timezone and the various twilights, as they are just alternative ways to tell time.

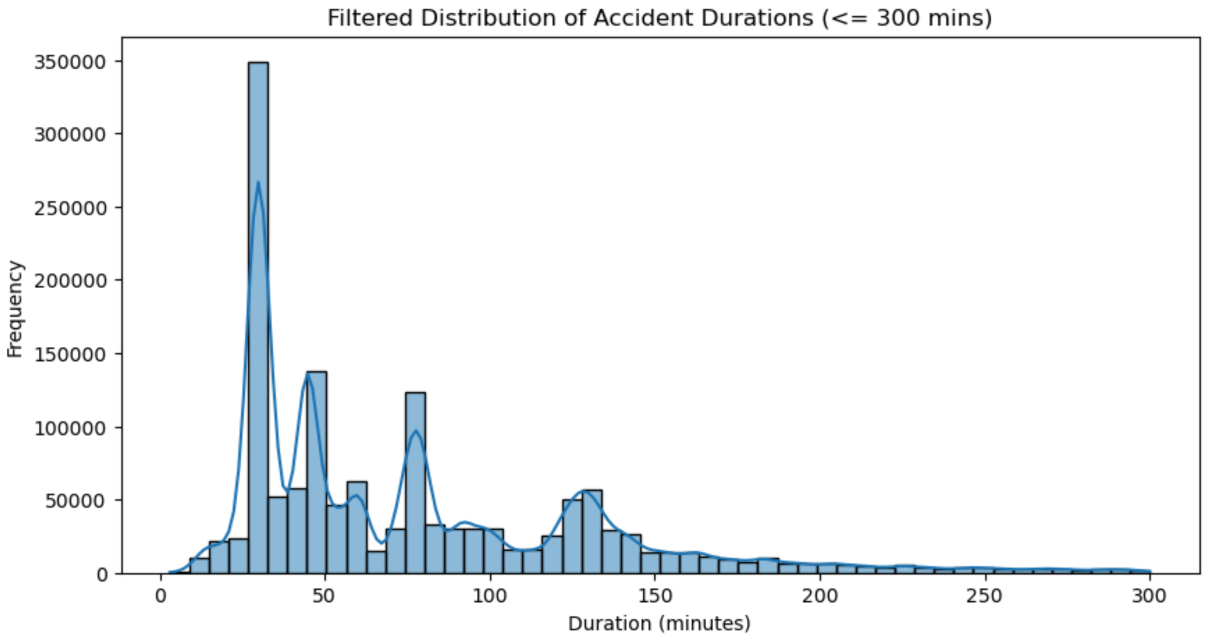
We will also analyse the duration of accident using (end_time - start_time) to see if end_time can be dropped.

In [14]:
```
df['Start_Time'] = pd.to_datetime(df['Start_Time'], errors='coerce')
df['End_Time'] = pd.to_datetime(df['End_Time'], errors='coerce')
```

In [15]:
```
df['Duration'] = df['End_Time'] - df['Start_Time']
df['Duration_Minutes'] = df['Duration'].dt.total_seconds() / 60
```

Since some of the accident durations are days, we will remove them from this visualisation as the focus of our project is not on how accidents affect traffic.

In [17]:
```
filtered_df = df[df['Duration_Minutes'] <= 300]

plt.figure(figsize=(10, 5))
sb.histplot(filtered_df['Duration_Minutes'], bins=50, kde=True)
plt.title("Filtered Distribution of Accident Durations (<= 300 mins)")
plt.xlabel("Duration (minutes)")
plt.ylabel("Frequency")
plt.show()
```



We observe that most accident durations is about 25 minutes, which we deem to be too short to be relevant to our dataset and problem statement. Hence, we justify removing End_Time (and therefore End_Lat, End_Lng too) from our dataset.

In [19]:
```
columns_to_drop = [
    'ID', 'Source', 'End_Time','End_Lat', 'End_Lng', 'Country', 'State', 'Ti
    'Sunrise_Sunset', 'Civil_Twilight', 'Nautical_Twilight', 'Astronomical_T
]
df.drop(columns=columns_to_drop, inplace=True)
```

In [20]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1741433 entries, 0 to 1741432
Data columns (total 31 columns):
 #   Column             Dtype
---  ------             -----
 0   Severity           int64
 1   Start_Time         datetime64[ns]
 2   Start_Lat          float64
 3   Start_Lng          float64
 4   Distance(mi)       float64
 5   Description        object
 6   Street             object
 7   City               object
 8   County             object
 9   Zipcode            object
 10  Temperature(F)     float64
 11  Wind_Chill(F)      float64
 12  Humidity(%)        float64
 13  Pressure(in)       float64
 14  Visibility(mi)     float64
 15  Wind_Direction     object
 16  Wind_Speed(mph)    float64
 17  Precipitation(in)  float64
 18  Weather_Condition  object
 19  Amenity            bool
 20  Bump               bool
 21  Crossing           bool
 22  Give_Way           bool
 23  Junction           bool
 24  No_Exit            bool
 25  Railway            bool
 26  Roundabout         bool
 27  Station            bool
 28  Stop               bool
 29  Traffic_Calming    bool
 30  Traffic_Signal     bool
dtypes: bool(12), datetime64[ns](1), float64(10), int64(1), object(7)
memory usage: 272.4+ MB
```

We will now convert this dataset's features into their appropriate types, making it easier to handle the data and help the ML models better understand the features. This is important for datetime and categorical features.

We realise an important step is to split the Start_Time into date and time separately, because our problem statement considers time to be more important than date.

In [22]:

```
# Convert object columns to category
object_cols = df.select_dtypes(include=['object']).columns
df[object_cols] = df[object_cols].astype('category')

# Specifically convert 'Description' to string
df["Description"] = df["Description"].astype('string')
```

```python
# Start_Time has already been converted into datetime earlier, but here we w

# Split Start_Time into separate features
df['Start_Date'] = df['Start_Time'].dt.date
df['Start_Hour'] = df['Start_Time'].dt.hour
df['Start_Minute'] = df['Start_Time'].dt.minute
df['Accident_Time'] = df['Start_Hour'] + df['Start_Minute'] / 60

# Update column type lists AFTER cleaning
datetime_cols = df.select_dtypes(include=['datetime64[ns]']).columns.tolist(
cat_cols = df.select_dtypes(include=['category']).columns.tolist()
bool_cols = df.select_dtypes(include=['bool']).columns.tolist()
num_cols = df.select_dtypes(include=['float64', 'int64']).columns.tolist()
string_cols = df.select_dtypes(include=['string']).columns.tolist()
```

These are the converted data types.

In [24]:
```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1741433 entries, 0 to 1741432
Data columns (total 35 columns):
 #   Column             Dtype         
---  ------             -----         
 0   Severity           int64         
 1   Start_Time         datetime64[ns]
 2   Start_Lat          float64       
 3   Start_Lng          float64       
 4   Distance(mi)       float64       
 5   Description        string        
 6   Street             category      
 7   City               category      
 8   County             category      
 9   Zipcode            category      
 10  Temperature(F)     float64       
 11  Wind_Chill(F)      float64       
 12  Humidity(%)        float64       
 13  Pressure(in)       float64       
 14  Visibility(mi)     float64       
 15  Wind_Direction     category      
 16  Wind_Speed(mph)    float64       
 17  Precipitation(in)  float64       
 18  Weather_Condition  category      
 19  Amenity            bool          
 20  Bump               bool          
 21  Crossing           bool          
 22  Give_Way           bool          
 23  Junction           bool          
 24  No_Exit            bool          
 25  Railway            bool          
 26  Roundabout         bool          
 27  Station            bool          
 28  Stop               bool          
 29  Traffic_Calming    bool          
 30  Traffic_Signal     bool          
 31  Start_Date         object        
 32  Start_Hour         float64       
 33  Start_Minute       float64       
 34  Accident_Time      float64       
dtypes: bool(12), category(6), datetime64[ns](1), float64(13), int64(1), obj
ect(1), string(1)
memory usage: 275.0+ MB
```

After feature reduction and keeping only columns we deemed as relevant, now we will handle any NULL values present in the dataset. Let's look at the spread of NULL values within the dataset.

In [26]:

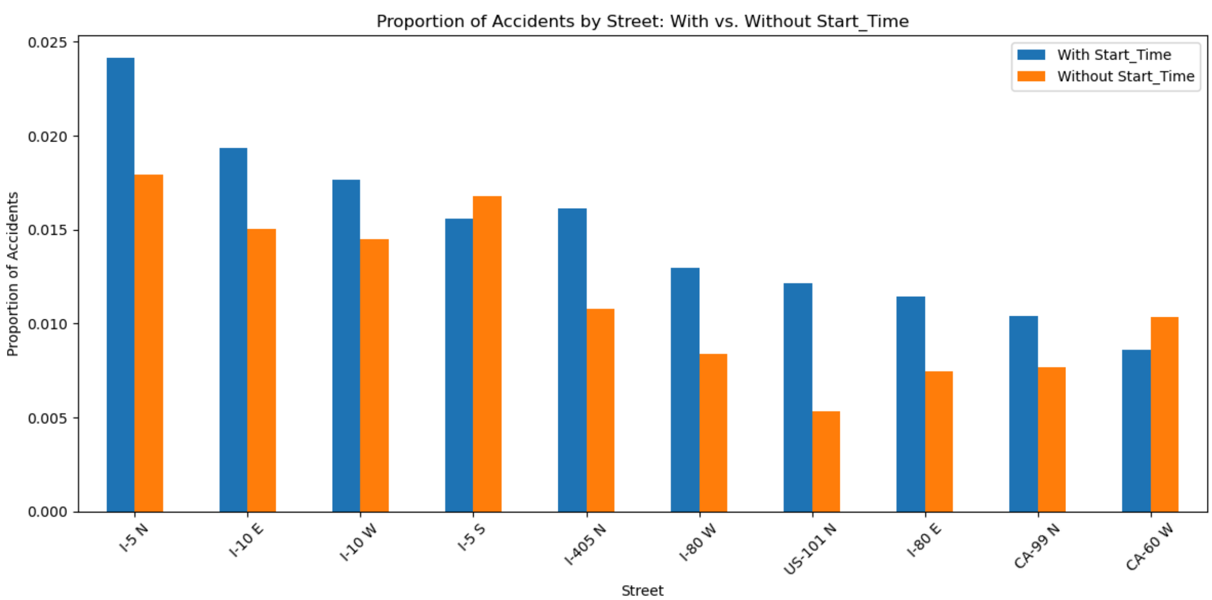
```
null_count = df.isnull().sum()

total_rows = len(df)
null_percentage = (null_count / total_rows) * 100

null_df = pd.DataFrame({'Null Count': null_count, 'Null Percentage': null_pe
print(null_df)
```

	Null Count	Null Percentage
Severity	0	0.000000
Start_Time	174297	10.008826
Start_Lat	0	0.000000
Start_Lng	0	0.000000
Distance(mi)	0	0.000000
Description	3	0.000172
Street	2442	0.140229
City	11	0.000632
County	0	0.000000
Zipcode	597	0.034282
Temperature(F)	45969	2.639723
Wind_Chill(F)	510965	29.341640
Humidity(%)	48341	2.775932
Pressure(in)	37126	2.131922
Visibility(mi)	40125	2.304137
Wind_Direction	46189	2.652356
Wind_Speed(mph)	162891	9.353848
Precipitation(in)	566204	32.513683
Weather_Condition	39778	2.284211
Amenity	0	0.000000
Bump	0	0.000000
Crossing	0	0.000000
Give_Way	0	0.000000
Junction	0	0.000000
No_Exit	0	0.000000
Railway	0	0.000000
Roundabout	0	0.000000
Station	0	0.000000
Stop	0	0.000000
Traffic_Calming	0	0.000000
Traffic_Signal	0	0.000000
Start_Date	174297	10.008826
Start_Hour	174297	10.008826
Start_Minute	174297	10.008826
Accident_Time	174297	10.008826

We noticed approximately 10% of the dataset contains missing values in the Start_Time column, which might be a result of improper date time format that failed to convert, resulting in NaT (Not a time) NULL value replacing the data point. Since Start_Time is a critical variable used to derive several temporal features such as accident time, rows lacking this information are unsuitable for time-based analysis, but 10% is a considerable amount of data to be dropped.

We decided to visualise the distribution of streeets across the dataset to observe if the rows with NULL accident time will skew the results heavily if removed. For example, if a street has disproportionately higher accidents with no time than ones recorded with time, then removing the NULL times would impact the ML results of that street. We will select the streets with the most number of accidents to plot.

In [28]:
```python
df_with_time = df[df['Start_Time'].notnull()]
df_without_time = df[df['Start_Time'].isnull()]

# Count accidents per street for each group
street_counts_with_time = df_with_time['Street'].value_counts()
street_counts_without_time = df_without_time['Street'].value_counts()

# Combine counts into a DataFrame
street_counts = pd.DataFrame({
    'With Start_Time': street_counts_with_time,
    'Without Start_Time': street_counts_without_time
}).fillna(0)

# Normalise to get proportions
street_counts_norm = street_counts.div(street_counts.sum(axis=0), axis=1)

# Select top N streets by total accidents
top_n = 10
top_streets = street_counts.sum(axis=1).nlargest(top_n).index
street_counts_top = street_counts_norm.loc[top_streets]

# Plot
street_counts_top.plot(kind='bar', figsize=(12, 6))
plt.title('Proportion of Accidents by Street: With vs. Without Start_Time')
plt.xlabel('Street')
plt.ylabel('Proportion of Accidents')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Since the proportions of accidents across streets are similar between the two groups, dropping rows with missing Start_Time is unlikely to bias our analysis. We also believe that imputing the NULL times with an average time would not be the best way to handle the data, and thus decided on dropping rows with missing Start_Time.

In [30]:
```
df.dropna(subset=['Start_Time'], inplace=True)
```

In [31]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1567136 entries, 0 to 1741432
Data columns (total 35 columns):
 #   Column             Non-Null Count    Dtype
---  ------             --------------    -----
 0   Severity           1567136 non-null  int64
 1   Start_Time         1567136 non-null  datetime64[ns]
 2   Start_Lat          1567136 non-null  float64
 3   Start_Lng          1567136 non-null  float64
 4   Distance(mi)       1567136 non-null  float64
 5   Description        1567133 non-null  string
 6   Street             1565225 non-null  category
 7   City               1567127 non-null  category
 8   County             1567136 non-null  category
 9   Zipcode            1566637 non-null  category
 10  Temperature(F)     1526011 non-null  float64
 11  Wind_Chill(F)      1063180 non-null  float64
 12  Humidity(%)        1523810 non-null  float64
 13  Pressure(in)       1534257 non-null  float64
 14  Visibility(mi)     1531542 non-null  float64
 15  Wind_Direction     1526613 non-null  category
 16  Wind_Speed(mph)    1409998 non-null  float64
 17  Precipitation(in)  1014985 non-null  float64
 18  Weather_Condition  1531733 non-null  category
 19  Amenity            1567136 non-null  bool
 20  Bump               1567136 non-null  bool
 21  Crossing           1567136 non-null  bool
 22  Give_Way           1567136 non-null  bool
 23  Junction           1567136 non-null  bool
 24  No_Exit            1567136 non-null  bool
 25  Railway            1567136 non-null  bool
 26  Roundabout         1567136 non-null  bool
 27  Station            1567136 non-null  bool
 28  Stop               1567136 non-null  bool
 29  Traffic_Calming    1567136 non-null  bool
 30  Traffic_Signal     1567136 non-null  bool
 31  Start_Date         1567136 non-null  object
 32  Start_Hour         1567136 non-null  float64
 33  Start_Minute       1567136 non-null  float64
 34  Accident_Time      1567136 non-null  float64
dtypes: bool(12), category(6), datetime64[ns](1), float64(13), int64(1), obj
ect(1), string(1)
memory usage: 260.2+ MB
```

To address the other NULL values, they will be broken down into 3 groups:

1. Crticial columns with low NULLs/NULL percentages: These rows will be dropped as the change is negligible but will greatly improve quality of dataset

2. Numerical weather features (2-9% missing values): These rows will be filled with median values to reflect the central tendency of these weather conditions
3. Columns with high NULLs/NULL percentages (Wind_Chill and Precipitation): Wind_Chill will be dropped as with such a high NULL percentage, it is often redundant and comes hand in hand with attribute from another column such as Temperature.

Precipitation NULLs will be inputed with 0 but not dropped, because rain is an important weather condition. If precipiation were a factor of the accident, we assume it would have been documented. Thus, empty cells for precipiation implies rain was not a contributive factor and we assume NULL cells will be replaced with 0. Disclaimer: precipitation is often not measured precisely as snow may not be counted as Precipitation.

We will ignore Description column.

In [33]:
```
print("Missing values BEFORE cleaning:")
print(df.isnull().sum()[df.isnull().sum() > 0].sort_values(ascending=False))

# dealing with group 1
df.dropna(subset=['Street', 'City'], inplace=True)

# dealing with group 2
weather_num_cols = ['Temperature(F)', 'Humidity(%)', 'Pressure(in)', 'Visibi
for col in weather_num_cols:
    df[col].fillna(df[col].median(), inplace=True)

# handling missing Zipcode
if 'Unknown' not in df['Zipcode'].cat.categories:
    df['Zipcode'] = df['Zipcode'].cat.add_categories('Unknown')
df['Zipcode'] = df['Zipcode'].fillna('Unknown')


# Fill categorical/object-based weather columns with 'Unknown'
if 'Wind_Direction' in df.columns and df['Wind_Direction'].dtype.name == 'ca
    if 'Unknown' not in df['Wind_Direction'].cat.categories:
        df['Wind_Direction'] = df['Wind_Direction'].cat.add_categories('Unkn
    df['Wind_Direction'] = df['Wind_Direction'].fillna('Unknown')

if 'Weather_Condition' in df.columns and df['Weather_Condition'].dtype.name
    if 'Unknown' not in df['Weather_Condition'].cat.categories:
        df['Weather_Condition'] = df['Weather_Condition'].cat.add_categories
    df['Weather_Condition'] = df['Weather_Condition'].fillna('Unknown')

df.drop(columns=['Wind_Chill(F)'], inplace=True)
df['Precipitation(in)'].fillna(0, inplace=True)  # assumes most missing = no


# Optional: Fill less important columns with default (for modeling)
optional_bool_cols = df.select_dtypes(include='bool').columns
df[optional_bool_cols] = df[optional_bool_cols].fillna(False)
```

```python
# Check again after cleaning
print("\nMissing values AFTER cleaning:")
print(df.isnull().sum()[df.isnull().sum() > 0])
```

```
Missing values BEFORE cleaning:
Precipitation(in)    552151
Wind_Chill(F)        503956
Wind_Speed(mph)      157138
Humidity(%)           43326
Temperature(F)        41125
Wind_Direction        40523
Visibility(mi)        35594
Weather_Condition     35403
Pressure(in)          32879
Street                 1911
Zipcode                 499
City                      9
Description               3
dtype: int64
```

```
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\3493830457.py:10: FutureW
arning: A value is trying to be set on a copy of a DataFrame or Series throu
gh chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behave
s as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'd
f.method({col: value}, inplace=True)' or df[col] = df[col].method(value) ins
tead, to perform the operation inplace on the original object.


  df[col].fillna(df[col].median(), inplace=True)
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\3493830457.py:30: FutureW
arning: A value is trying to be set on a copy of a DataFrame or Series throu
gh chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behave
s as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'd
f.method({col: value}, inplace=True)' or df[col] = df[col].method(value) ins
tead, to perform the operation inplace on the original object.


  df['Precipitation(in)'].fillna(0, inplace=True)  # assumes most missing =
no rain
```

```
Missing values AFTER cleaning:
Description    3
dtype: int64
```

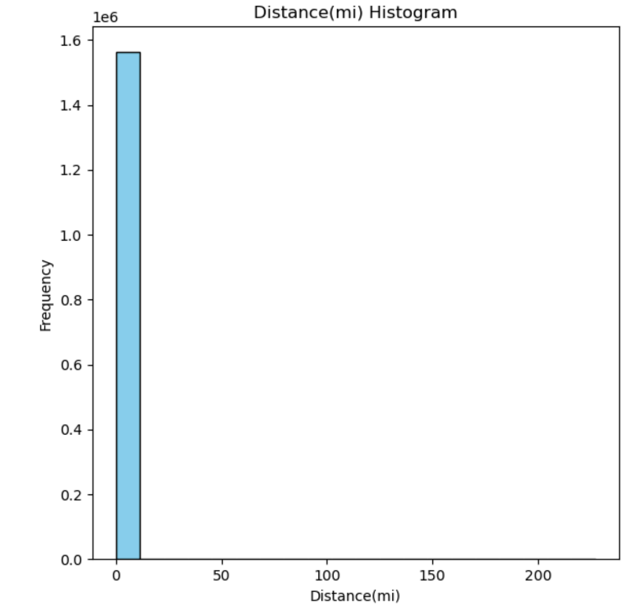
We have successfully handled NULL values in our dataset!

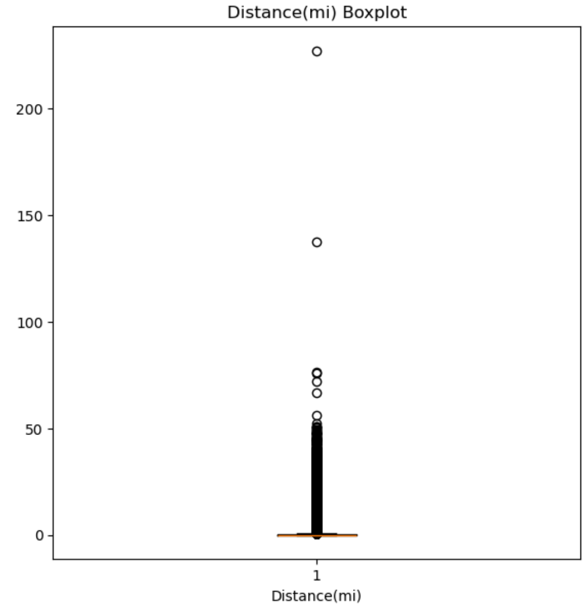
Distribution Analysis & Normalisation

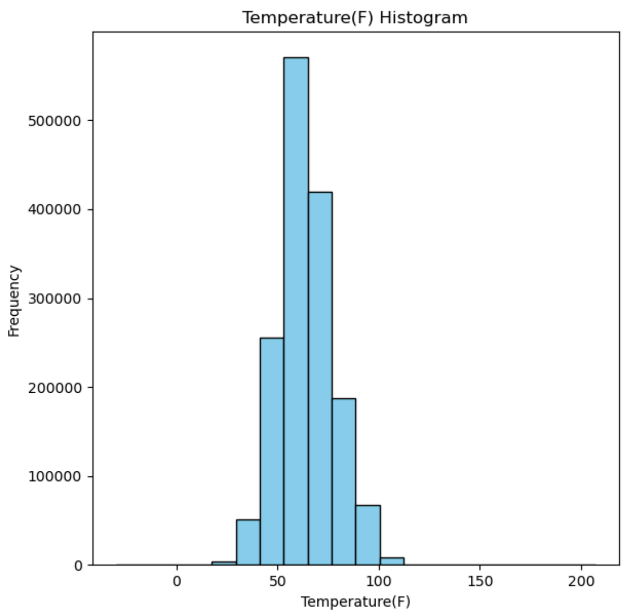
Now, looking at how some columns have many unique values (continuous numericial), we will normalise the data to remove outliers as they can skew data analysis and affect the reliability of ML models like KNN and clustering.

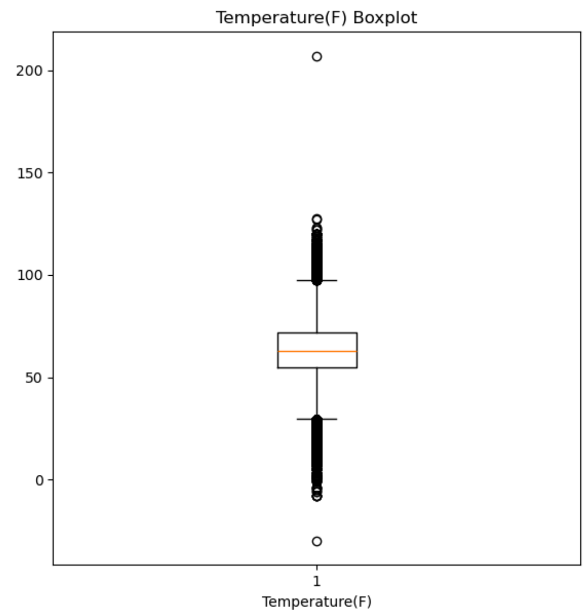
We will begin with a visualisation of potential outliers in the dataset.

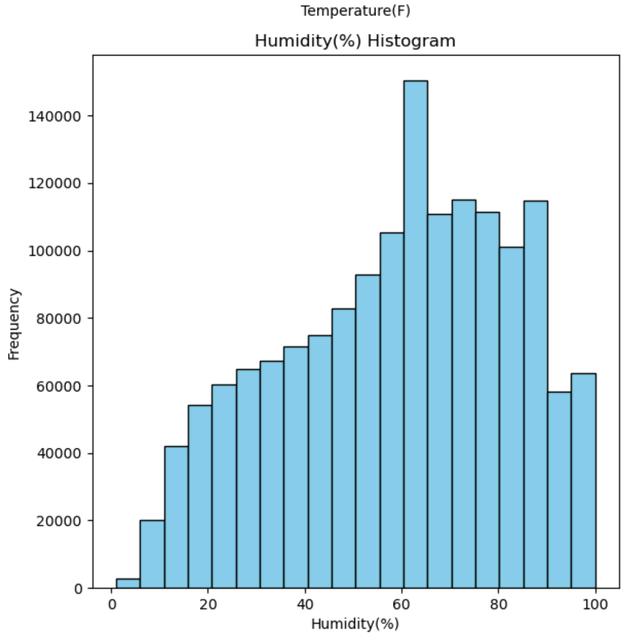
In [37]:
```
def plot_numeric_features(df, numeric_features):
    df[numeric_features] = df[numeric_features].apply(pd.to_numeric, errors=
    df[numeric_features] = df[numeric_features].fillna(0)
    num_cols = len(numeric_features)
    fig, axs = plt.subplots(num_cols, 2, figsize=(12, num_cols*6))

    for i, feature in enumerate(numeric_features):
        # Histogram
        axs[i, 0].hist(df[feature], bins=20, color='skyblue', edgecolor='bla
        axs[i, 0].set_title(f'{feature} Histogram')
        axs[i, 0].set_xlabel(feature)
        axs[i, 0].set_ylabel('Frequency')

        # Boxplot
        axs[i, 1].boxplot(df[feature], vert=True)
        axs[i, 1].set_title(f'{feature} Boxplot')
        axs[i, 1].set_xlabel(feature)

    plt.tight_layout()
    plt.show()

numeric_features = ['Distance(mi)', 'Temperature(F)', 'Humidity(%)', 'Pressu
                    'Visibility(mi)', 'Wind_Speed(mph)', 'Precipitation(in)'
plot_numeric_features(df, numeric_features)
```

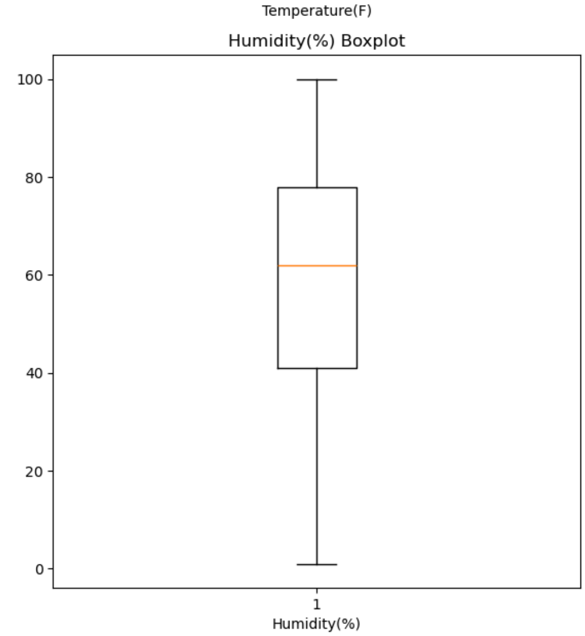

Distance(mi) Histogram
1e6
Frequency
Distance(mi)

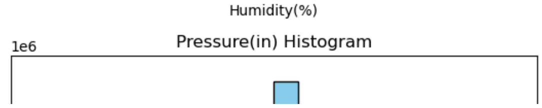

Distance(mi) Boxplot
Distance(mi)

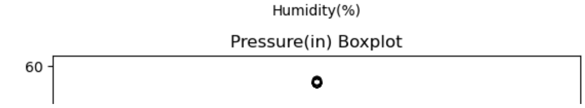

Temperature(F) Histogram
Frequency
Temperature(F)

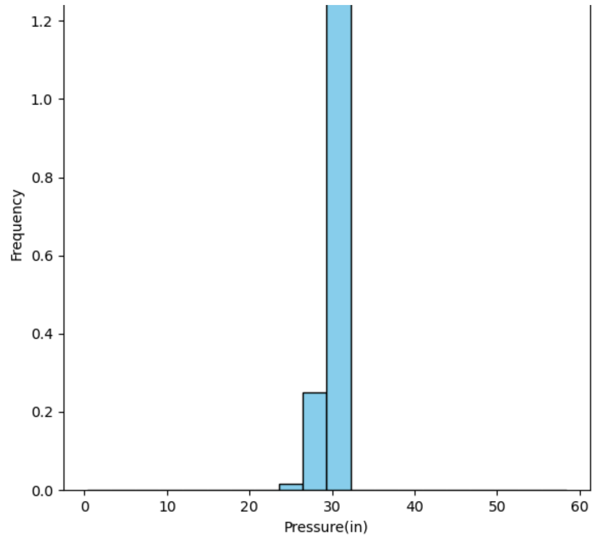

Temperature(F) Boxplot
Temperature(F)

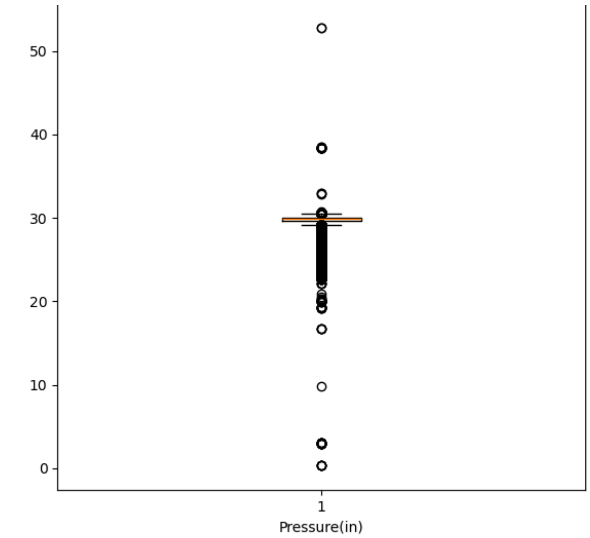

Humidity(%) Histogram
Frequency
Humidity(%)

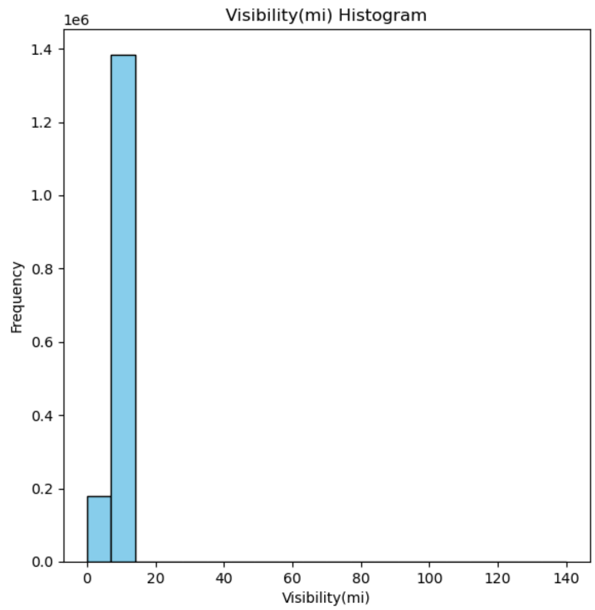

Humidity(%) Boxplot
Humidity(%)

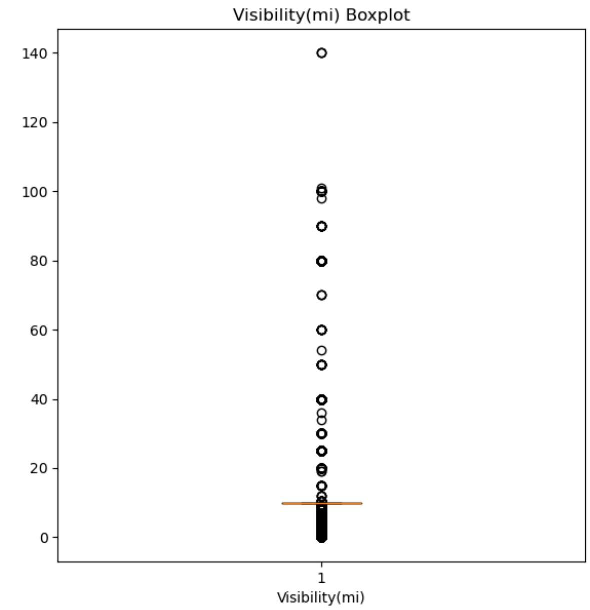

Pressure(in) Histogram
1e6

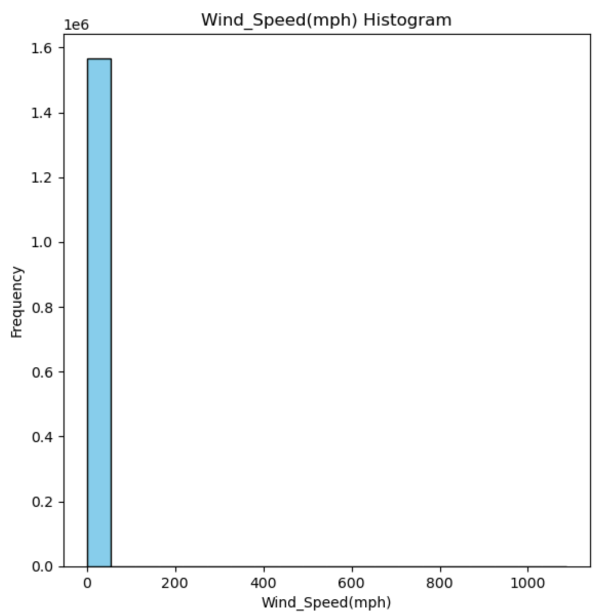

Pressure(in) Boxplot

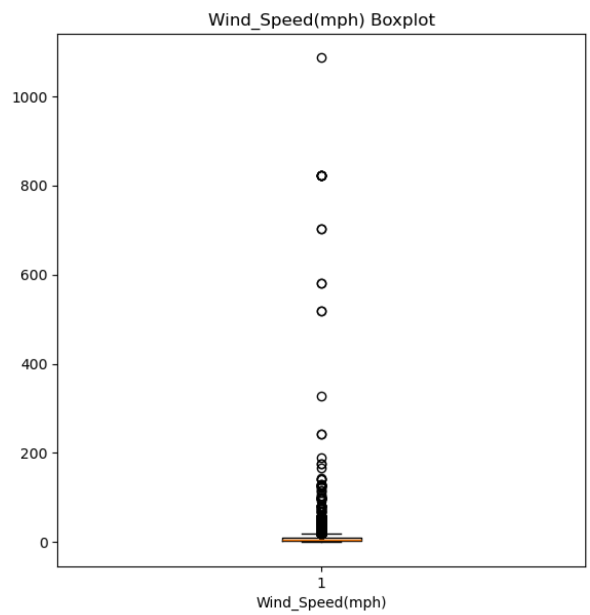

Frequency
0.0 0.2 0.4 0.6 0.8 1.0 1.2
0 10 20 30 40 50 60
Pressure(in)

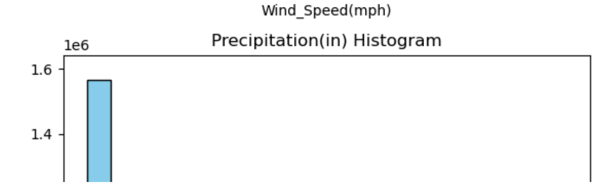

0 10 20 30 40 50
1
Pressure(in)

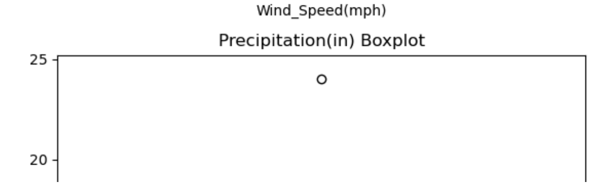

Visibility(mi) Histogram
1e6
Frequency
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4
0 20 40 60 80 100 120 140
Visibility(mi)

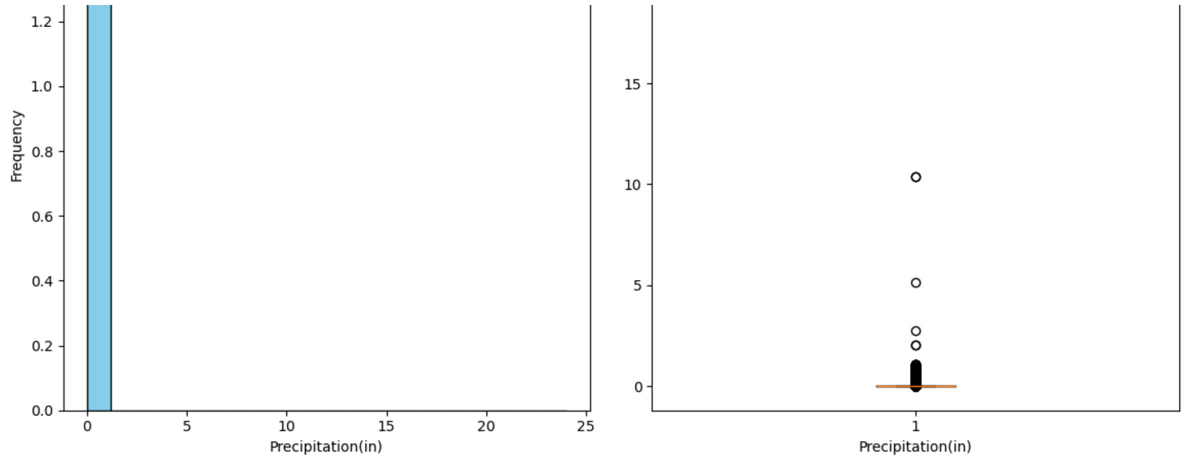

Visibility(mi) Boxplot
0 20 40 60 80 100 120 140
1
Visibility(mi)


Wind_Speed(mph) Histogram
1e6
Frequency
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6
0 200 400 600 800 1000
Wind_Speed(mph)


Wind_Speed(mph) Boxplot
0 200 400 600 800 1000
1
Wind_Speed(mph)

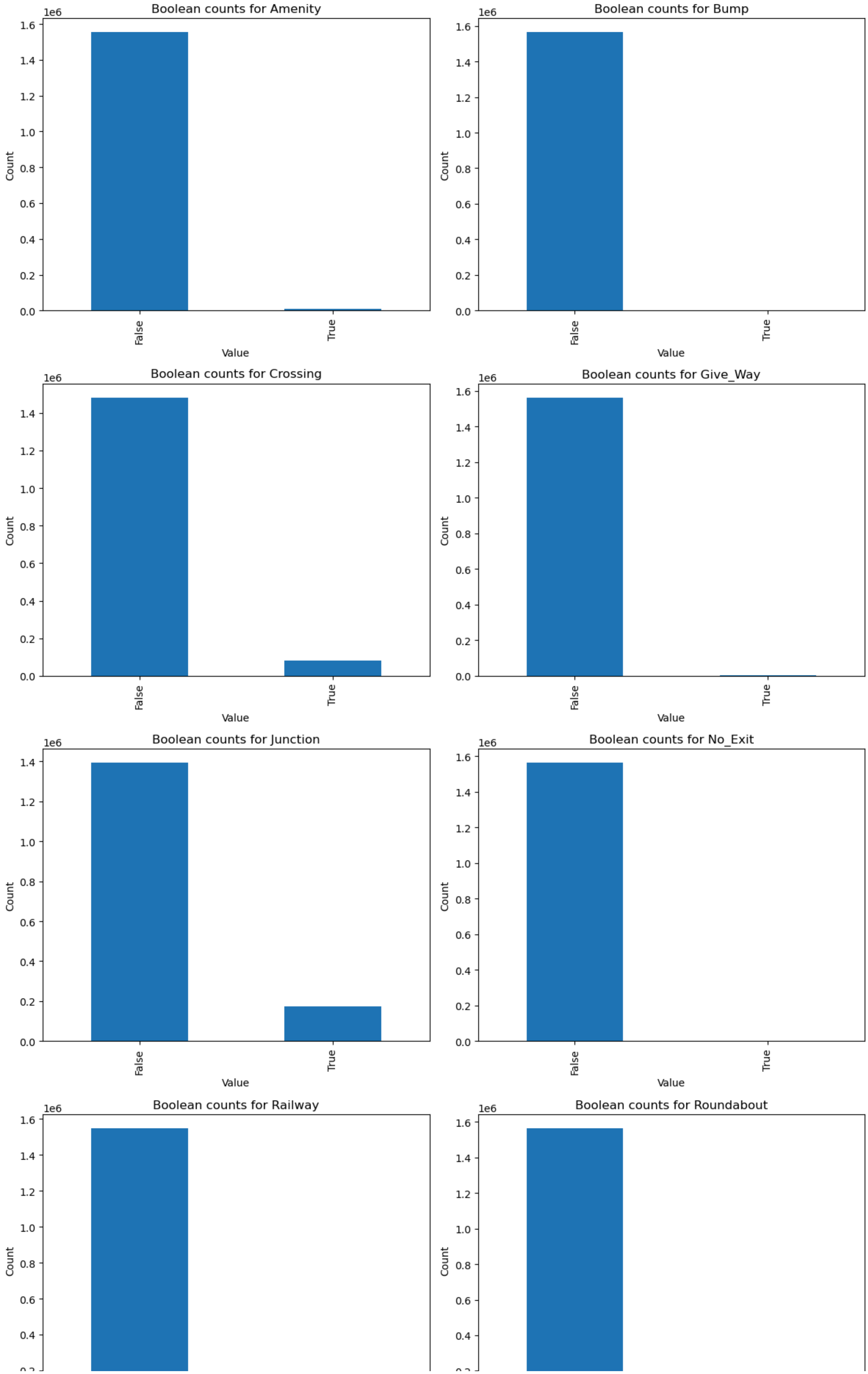

Precipitation(in) Histogram
1e6
1.4 1.6


Precipitation(in) Boxplot
20 25

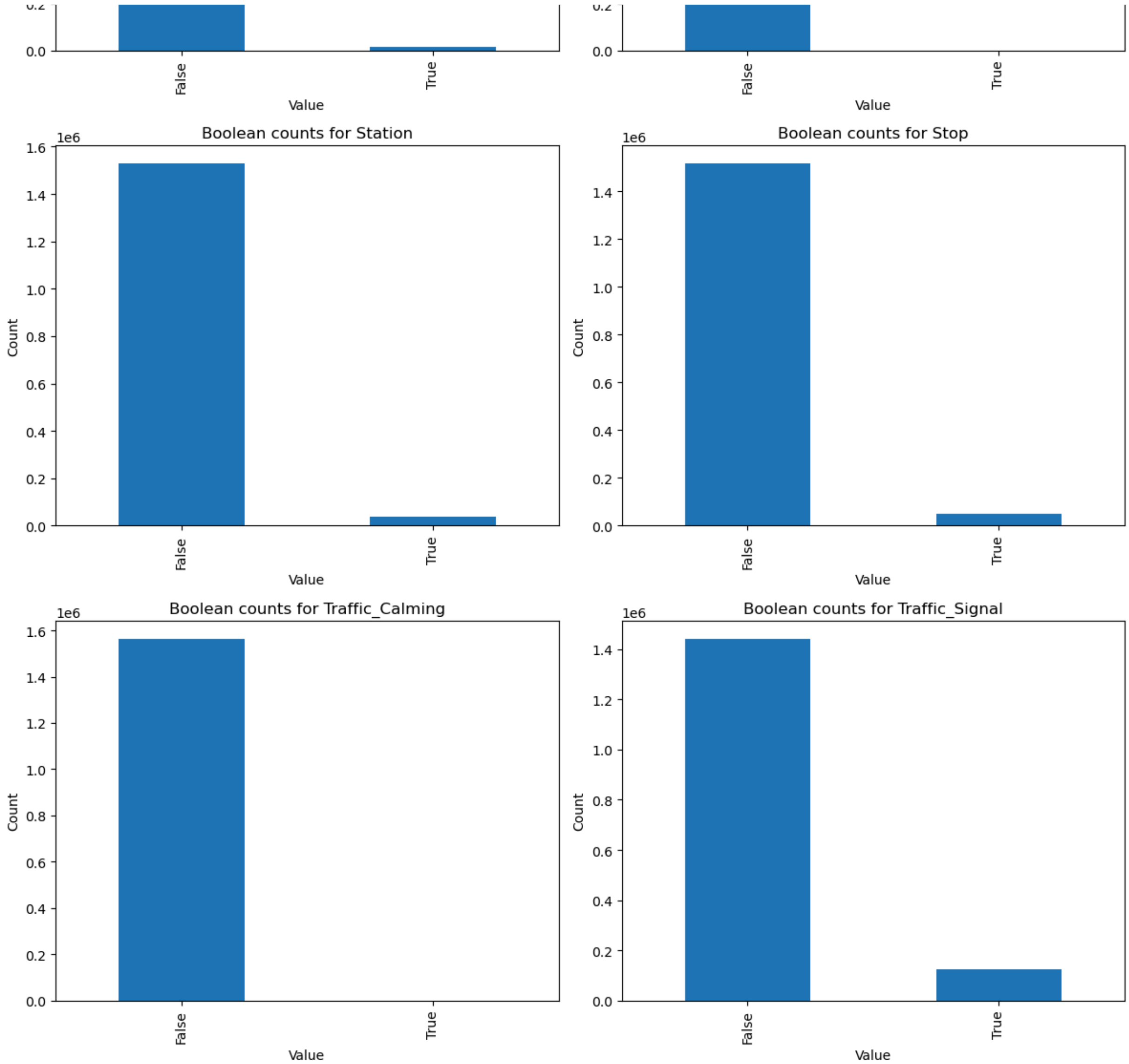


Outliers are detected in precipitation, wind_speed, visbility and distance. Possible explanations would be that most days go with 0 rain, and crashes often occur nearby. Either ways, the data can be normalised to minimise the skew.

In [39]:
```
def plot_boolean_features(df, bool_features):
    # Calculate the number of rows and columns for subplots
    num_features = len(bool_features)
    num_rows = num_features // 2 + num_features % 2
    num_cols = 2

    # Create subplots with rectangular shape
    fig, axes = plt.subplots(num_rows, num_cols, figsize=(12, 30))

    # Flatten the axes array to make it easier to iterate
    axes = axes.flatten()

    # Plot boolean counts for specified features
    for i, feature in enumerate(bool_features):
        counts = df[feature].value_counts()
        counts.plot(kind='bar', ax=axes[i])
        axes[i].set_title(f'Boolean counts for {feature}')
        axes[i].set_xlabel('Value')
        axes[i].set_ylabel('Count')

    # Hide empty subplots
    for j in range(num_features, num_rows * num_cols):
        axes[j].axis('off')

    plt.tight_layout()
    plt.show()

plot_boolean_features(df, bool_cols)
```



Boolean counts for Amenity
1e6
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Count
False
True
Value
Boolean counts for Bump
1e6
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Count
False
True
Value
Boolean counts for Crossing
1e6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Count
False
True
Value
Boolean counts for Give_Way
1e6
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Count
False
True
Value
Boolean counts for Junction
1e6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Count
False
True
Value
Boolean counts for No_Exit
1e6
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Count
False
True
Value
Boolean counts for Railway
1e6
1.6
1.4
1.2
1.0
0.8
0.6
0.4
Count
Boolean counts for Roundabout
1e6
1.6
1.4
1.2
1.0
0.8
0.6
0.4
Count



In all the above graphs, false values are more frequent than true values.

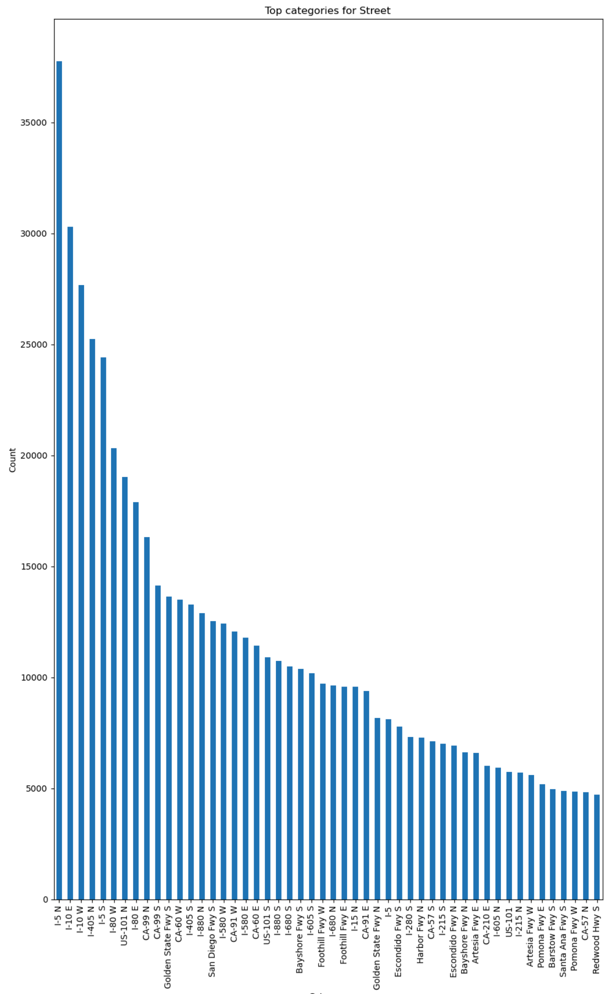
In [41]:
```
for column in df[bool_cols].columns:
    true_percentage = df[column].mean() * 100
    false_percentage = 100 - true_percentage

    print(column)
    print(f"Percentage of True: {true_percentage:.2f}%")
    print(f"Percentage of False: {false_percentage:.2f}%")
    print()
```

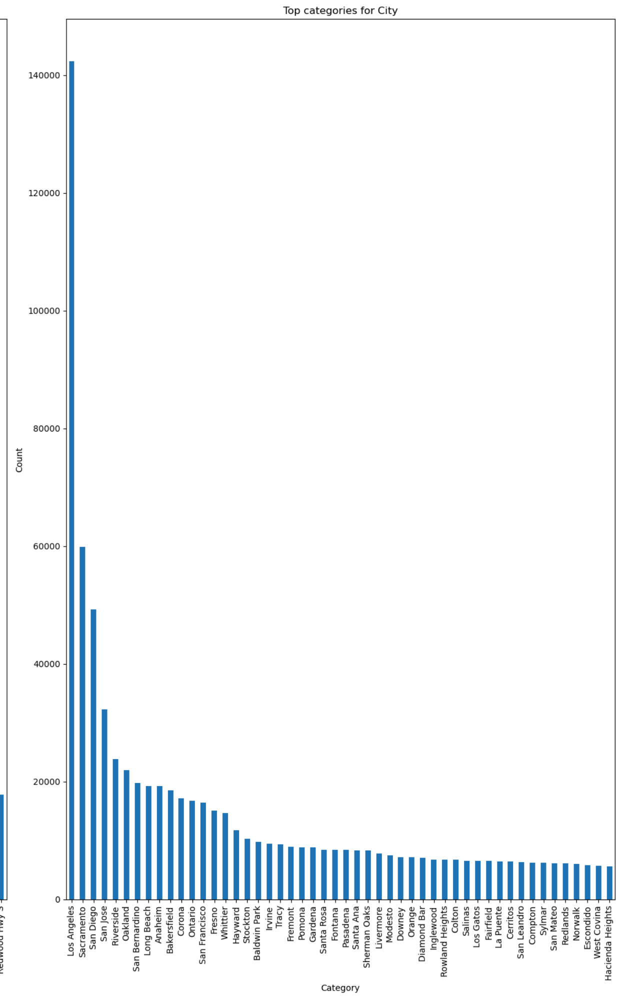
```
Amenity
Percentage of True: 0.72%
Percentage of False: 99.28%

Bump
Percentage of True: 0.06%
Percentage of False: 99.94%

Crossing
Percentage of True: 5.30%
Percentage of False: 94.70%

Give_Way
Percentage of True: 0.13%
Percentage of False: 99.87%

Junction
Percentage of True: 11.03%
Percentage of False: 88.97%

No_Exit
Percentage of True: 0.11%
Percentage of False: 99.89%

Railway
Percentage of True: 1.05%
Percentage of False: 98.95%

Roundabout
Percentage of True: 0.00%
Percentage of False: 100.00%

Station
Percentage of True: 2.37%
Percentage of False: 97.63%

Stop
Percentage of True: 3.15%
Percentage of False: 96.85%

Traffic_Calming
Percentage of True: 0.08%
Percentage of False: 99.92%

Traffic_Signal
Percentage of True: 7.92%
Percentage of False: 92.08%
```

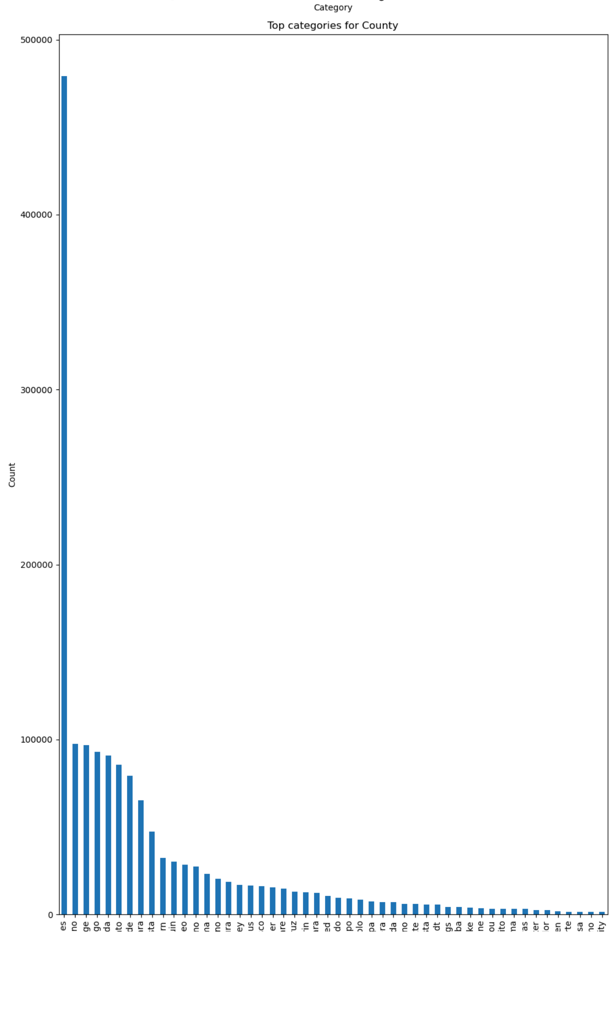
Only Traffic_Signal, Junction, and Crossing have True values over 3% of all values.

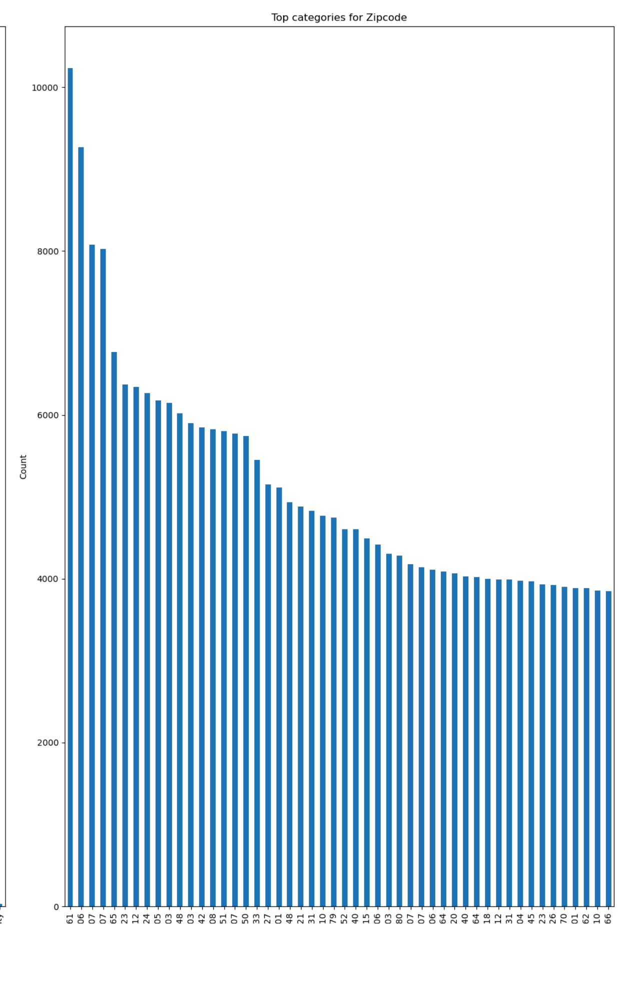
In [43]:
```
def plot_top_categories(df, cat_features):
    # Calculate the number of rows and columns for subplots
    num_features = len(df[cat_features].columns)
    num_rows = num_features // 2 + num_features % 2
    num_cols = 2
```

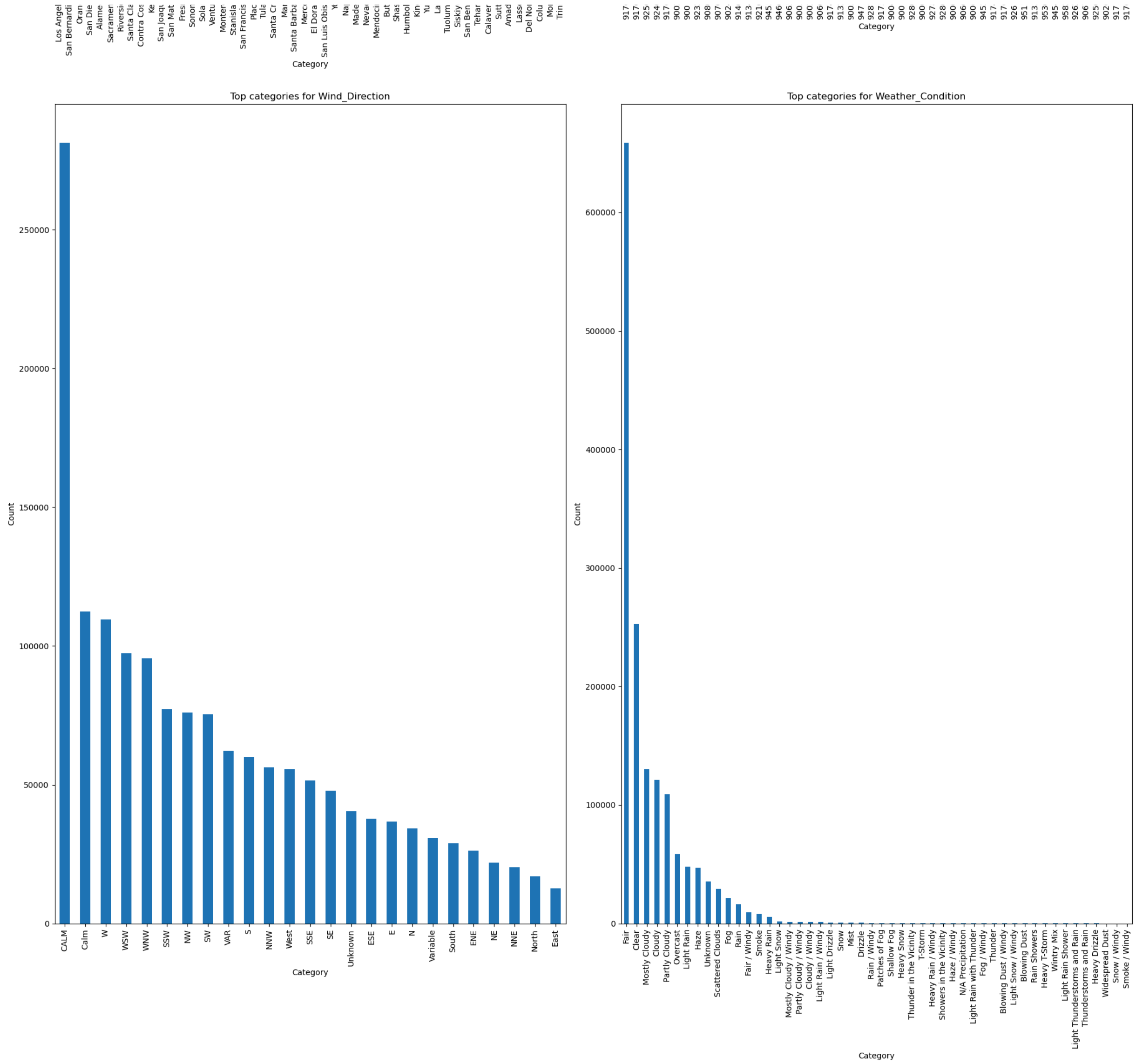
```
    # Create subplots
    fig, axes = plt.subplots(num_rows, num_cols, figsize=(20, 50))

    # Flatten the axes array to make it easier to iterate
    axes = axes.flatten()

    # Plot top categories for each categorical feature
    for i, column in enumerate(df[cat_features].columns):
        top_categories = df[column].value_counts().nlargest(50)
        top_categories.plot(kind='bar', ax=axes[i])
        axes[i].set_title(f'Top categories for {column}')
        axes[i].set_xlabel('Category')
        axes[i].set_ylabel('Count')

    # Hide empty subplots
    for j in range(num_features, num_rows * num_cols):
        axes[j].axis('off')

    plt.tight_layout()
    plt.show()

plot_top_categories(df, cat_cols)
```


Top categories for Street
Count
Category
I-5 N
I-10 E
I-10 W
I-405 N
I-5 S
I-80 W
US-101 N
I-80 E
CA-99 N
CA-99 S
Golden State Fwy S
CA-60 W
I-405 S
I-880 N
San Diego Fwy S
I-580 W
CA-91 W
I-580 E
CA-60 E
US-101 S
I-880 S
I-680 S
Bayshore Fwy S
I-605 S
Foothill Fwy W
I-680 N
Foothill Fwy E
I-15 N
CA-91 E
Golden State Fwy N
I-5
Escondido Fwy S
I-280 S
Harbor Fwy N
CA-57 S
I-215 S
Escondido Fwy N
Bayshore Fwy N
Artesia Fwy E
CA-210 E
I-605 N
US-101
I-215 N
Artesia Fwy W
Pomona Fwy E
Barstow Fwy S
Santa Ana Fwy S
Pomona Fwy W
CA-57 N
Redwood Hwy S


Top categories for City
Count
Category
Los Angeles
Sacramento
San Diego
San Jose
Riverside
Oakland
San Bernardino
Long Beach
Anaheim
Bakersfield
Corona
Ontario
San Francisco
Fresno
Whittier
Hayward
Stockton
Baldwin Park
Irvine
Tracy
Fremont
Pomona
Gardena
Santa Rosa
Fontana
Pasadena
Santa Ana
Sherman Oaks
Livermore
Modesto
Downey
Orange
Diamond Bar
Inglewood
Rowland Heights
Colton
Salinas
Los Gatos
Fairfield
La Puente
Cerritos
San Leandro
Compton
Sylmar
San Mateo
Redlands
Norwalk
Escondido
West Covina
Hacienda Heights


Top categories for County
Count


Top categories for Zipcode
Count



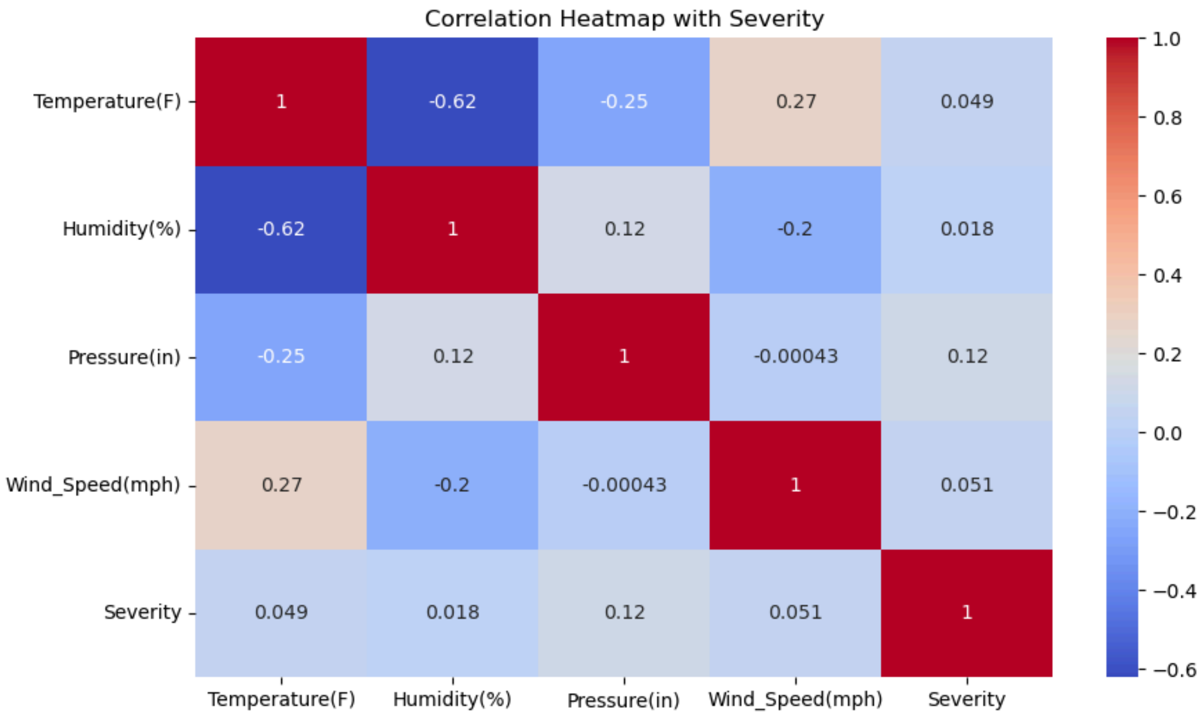
Now we normalise and remove outliers.

In [45]:
```
# Select only numeric columns excluding 'Severity'
numeric_cols = df.select_dtypes(include=['float64', 'int64']).columns
numeric_cols = [col for col in numeric_cols if col != 'Severity']  # Remove

# Normalize the remaining numeric columns
scaler = MinMaxScaler()
df[numeric_cols] = scaler.fit_transform(df[numeric_cols])

# Outlier removal
Q1 = df[numeric_cols].quantile(0.25)
Q3 = df[numeric_cols].quantile(0.75)
IQR = Q3 - Q1

# Filter out rows outside 1.5 * IQR range, excluding 'Severity' column
df = df[~((df[numeric_cols] < (Q1 - 1.5 * IQR)) | (df[numeric_cols] > (Q3 +
```

Correlation Analysis - Weather Conditions

We will perform correlation analysis to identify the most correlated weather conditions with severity, and illustrate the relationships between various conditions

In [48]:
```
num_cols = ['Temperature(F)', 'Humidity(%)', 'Pressure(in)', 'Wind_Speed(mph

scaler = MinMaxScaler()
df[num_cols] = scaler.fit_transform(df[num_cols])

plt.figure(figsize=(10, 6))
sb.heatmap(df[num_cols + ['Severity']].corr(), annot=True, cmap='coolwarm')
plt.title("Correlation Heatmap with Severity")
plt.show()
```



We will now perform feature merging to combine similar features. For example, we will combine Weather_Conditions as there are too many unique values inside, amany of which are overly-specified and semantically similar. This will reduce the complexity of our ML model, and improve its efficiency.

To address this, we will map specific weather conditions into broader categories, and create binary flags for important weather conditions (e.g. Is_Windy, Is_Wet) to improve model interpretability and performance.

In [50]:
```
def simplify_weather(condition):
    condition = condition.lower()
    if 'rain' in condition or 'drizzle' in condition or 'shower' in conditic
        return 'Rain'
    elif 'snow' in condition or 'sleet' in condition:
        return 'Snow'
    elif 'fog' in condition or 'mist' in condition or 'haze' in condition or
```

```
        return 'Fog'
    elif 'storm' in condition or 'thunder' in condition or 'tstorm' in condi
        return 'Storm'
    elif 'clear' in condition or 'sun' in condition:
        return 'Clear'
    elif 'cloud' in condition or 'overcast' in condition:
        return 'Cloudy'
    elif 'wind' in condition or 'breezy' in condition or 'gusty' in conditic
        return 'Windy'
    else:
        return 'Other' # how many are classified under other?

df['Weather_Simple'] = df['Weather_Condition'].astype(str).apply(simplify_we

#to see other
# Count how many rows were classified as 'Other'
other_count = df[df['Weather_Simple'] == 'Other'].shape[0]
print(f"Number of rows classified as 'Other': {other_count}")

# See the unique Weather_Condition values that ended up as 'Other'
other_conditions = df[df['Weather_Simple'] == 'Other']['Weather_Condition'].
print("Unique Weather_Condition values classified as 'Other':")
print(other_conditions)
```

```
Number of rows classified as 'Other': 393686
Unique Weather_Condition values classified as 'Other':
['Unknown', 'Fair', 'Volcanic Ash', 'Widespread Dust', 'Blowing Dust', 'N/A
Precipitation']
Categories (88, object): ['Blowing Dust', 'Blowing Dust / Windy', 'Blowing S
and', 'Blowing Snow', ..., 'Widespread Dust / Windy', 'Wintry Mix', 'Wintry
Mix / Windy', 'Unknown']
```

Creating binary flags to capture each weather effect, which helps with readability and interpretability admist mixed weather conditions.

In [52]:
```
df['Is_Windy'] = df['Weather_Condition'].str.contains('Wind|Breezy|Gusty', c
df['Is_Stormy'] = df['Weather_Condition'].str.contains('Storm|Thunder|Tstorm
df['Is_Rainy'] = df['Weather_Condition'].str.contains('Rain|Drizzle|Shower',
df['Is_Foggy'] = df['Weather_Condition'].str.contains('Fog|Mist|Haze|Smoke',
df['Is_Snowy'] = df['Weather_Condition'].str.contains('Snow|Sleet', case=Fal
df['Is_Clear'] = df['Weather_Condition'].str.contains('Clear|Sunny', case=Fa
```

In [53]:
```
df.nunique()
```

Out[53]:
```
Severity                    4
Start_Time             767175
Start_Lat              315148
Start_Lng              314165
Distance(mi)              973
Description            473306
Street                  47597
City                     1217
County                     58
Zipcode                 86293
Temperature(F)            405
Humidity(%)               100
Pressure(in)              132
Visibility(mi)              1
Wind_Direction             25
Wind_Speed(mph)            31
Precipitation(in)           1
Weather_Condition          37
Amenity                     2
Bump                        2
Crossing                    2
Give_Way                    2
Junction                    2
No_Exit                     2
Railway                     2
Roundabout                  2
Station                     2
Stop                        2
Traffic_Calming             2
Traffic_Signal              2
Start_Date               2497
Start_Hour                 24
Start_Minute               60
Accident_Time            1440
Weather_Simple              7
Is_Windy                    1
Is_Stormy                   2
Is_Rainy                    2
Is_Foggy                    2
Is_Snowy                    2
Is_Clear                    2
dtype: int64
```

After normalisation and outlier removal, we realised Visibility (mi) and Precipitation (in) have been reduced to only 1 unique value, which does not provide any valuable insights. Hence, these 2 columns will be dropped from the dataset.

In [55]:
```
features_to_drop = [
    'Visibility(mi)', 'Precipitation(in)'
]
df.drop(columns=features_to_drop, inplace=True)
```

Thus, the data has been properly processed through feature reduction, feature merging, NULL-value and outlier handling. Let's create the new DataFrame to be used for the EDA and model training.

In [57]:
```
df_cleaned = df.copy()
```

In [58]:
```
df_cleaned.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 889962 entries, 0 to 1741431
Data columns (total 39 columns):
 #   Column             Non-Null Count   Dtype
---  ------             --------------   -----
 0   Severity           889962 non-null  int64
 1   Start_Time         889962 non-null  datetime64[ns]
 2   Start_Lat          889962 non-null  float64
 3   Start_Lng          889962 non-null  float64
 4   Distance(mi)       889962 non-null  float64
 5   Description        889960 non-null  string
 6   Street             889962 non-null  category
 7   City               889962 non-null  category
 8   County             889962 non-null  category
 9   Zipcode            889962 non-null  category
 10  Temperature(F)     889962 non-null  float64
 11  Humidity(%)        889962 non-null  float64
 12  Pressure(in)       889962 non-null  float64
 13  Wind_Direction     889962 non-null  category
 14  Wind_Speed(mph)    889962 non-null  float64
 15  Weather_Condition  889962 non-null  category
 16  Amenity            889962 non-null  bool
 17  Bump               889962 non-null  bool
 18  Crossing           889962 non-null  bool
 19  Give_Way           889962 non-null  bool
 20  Junction           889962 non-null  bool
 21  No_Exit            889962 non-null  bool
 22  Railway            889962 non-null  bool
 23  Roundabout         889962 non-null  bool
 24  Station            889962 non-null  bool
 25  Stop               889962 non-null  bool
 26  Traffic_Calming    889962 non-null  bool
 27  Traffic_Signal     889962 non-null  bool
 28  Start_Date         889962 non-null  object
 29  Start_Hour         889962 non-null  float64
 30  Start_Minute       889962 non-null  float64
 31  Accident_Time      889962 non-null  float64
 32  Weather_Simple     889962 non-null  object
 33  Is_Windy           889962 non-null  int32
 34  Is_Stormy          889962 non-null  int32
 35  Is_Rainy           889962 non-null  int32
 36  Is_Foggy           889962 non-null  int32
 37  Is_Snowy           889962 non-null  int32
 38  Is_Clear           889962 non-null  int32
dtypes: bool(12), category(6), datetime64[ns](1), float64(10), int32(6), int
64(1), object(2), string(1)
memory usage: 157.8+ MB
```

Exploratory Data Analysis

We have successfully cleaned our data and will now perform exploratory data analysis. We have a few hypotheses that we wish to explore, and will do them sequentially to validate potential trends, and subsequently feed into our ML algorithms for predictions. The items we wish to explore are the following:

1. Which regions in California have the most accidents, and which

2. What trends can we identify in the frequency of accidents against the time of the day, and the day of the week?

3. Do accidents occur in all weather conditions or only in certain weather conditions?

In [64]:
```
df_cleaned.head(500000)
```

Out[64]:

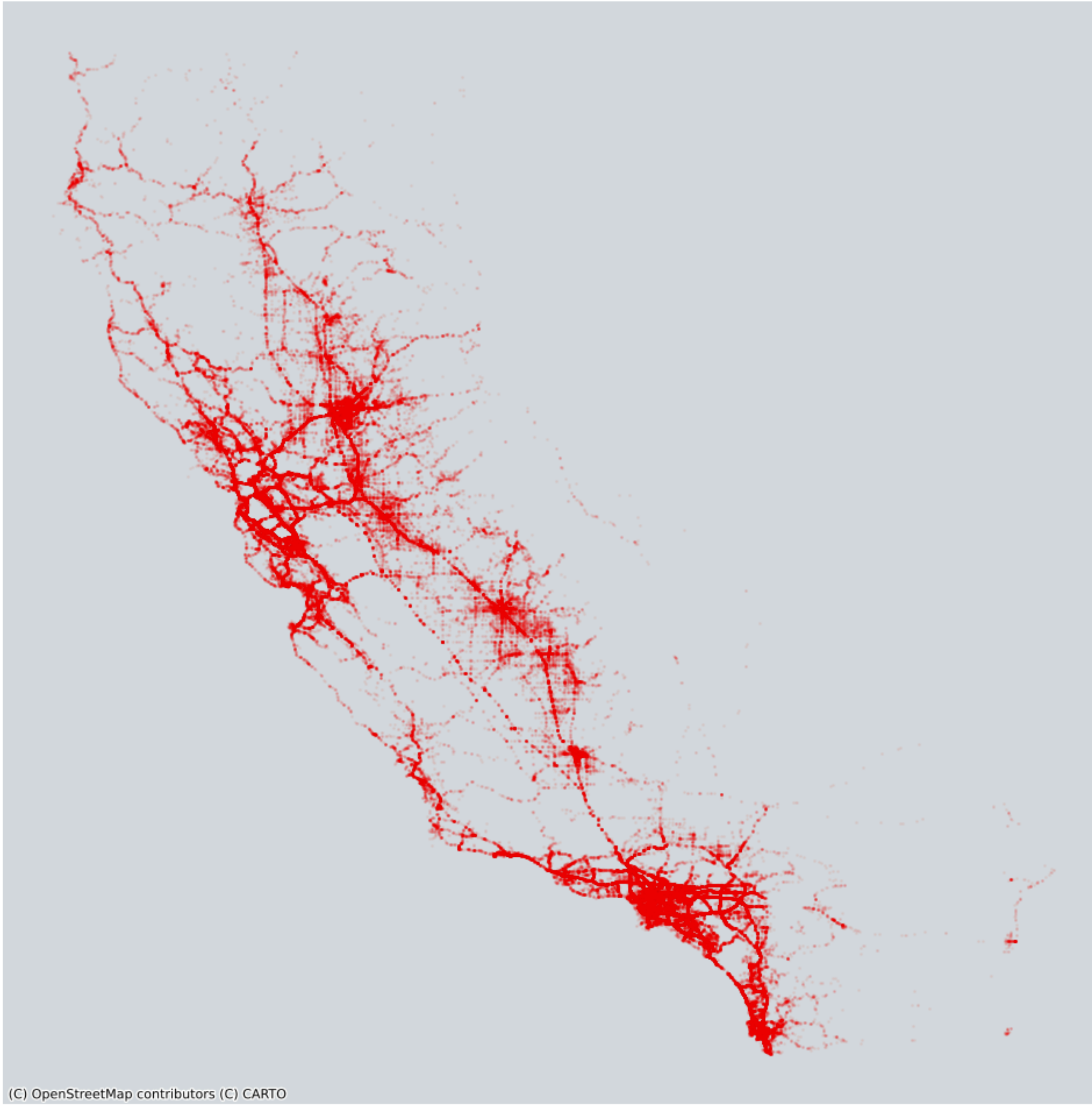
	Severity	Start_Time	Start_Lat	Start_Lng	Distance(mi)	Description
0	3	2016-06-21 10:34:40	0.585735	0.209256	0.000000	Right hand shoulder blocked due to accident on...
1	3	2016-06-21 10:30:16	0.537812	0.223798	0.000000	Accident on I-880 Northbound at Exit 26 Tennys...
2	2	2016-06-21 10:49:14	0.565790	0.225111	0.000000	Right lane blocked due to accident on CA-24 We...
3	3	2016-06-21 10:41:42	0.506367	0.228848	0.000000	#4 & HOV lane blocked due to accident on I...
4	2	2016-06-21 10:16:26	0.497540	0.240743	0.000000	Right hand shoulder blocked due to accident on...
...	...	...	...	...	...	...
796848	2	2022-03-23 22:27:27	0.027924	0.699416	0.000264	Incident on GARNET AVE near I-5 Drive with cau...
796849	2	2022-11-19 15:48:00	0.640625	0.296878	0.000238	Incident on FAIR OAKS BLVD near GARFIELD AVE E...
796852	2	2022-03-23 07:35:30	0.699910	0.136583	0.002940	Slow traffic on CA-20 from Scotts Valley Rd (C...
796853	2	2022-08-21 12:02:00	0.183023	0.569768	0.001857	Stationary traffic on CA-118 W - Ronald Reagan...
796858	2	2022-09-23 18:12:56	0.171099	0.607295	0.002548	Slow traffic on I-210 E - Foothill Fwy E from ...

500000 rows × 39 columns

We first start with a general plot of the entire California map with the accidents displayed, to be able to pick up some general trends.

In [67]:
```
# Set plotting styles
plt.rcParams['figure.figsize'] = (10, 6)

# sample for general trend
sample_df = df_cleaned[['Start_Lat', 'Start_Lng']].dropna().sample(500000, r

# convert to GeoDataFrame in WGS84
geometry = [Point(xy) for xy in zip(sample_df['Start_Lng'], sample_df['Start
gdf = gpd.GeoDataFrame(sample_df, geometry=geometry, crs='EPSG:4326')

# project to Web Mercator
gdf = gdf.to_crs(epsg=3857)

# plot the accidents
fig, ax = plt.subplots(figsize=(12, 10))
gdf.plot(ax=ax, markersize=1, alpha=0.05, color='red')

# after plotting gdf and adding basemap, get the map bounds
xmin, xmax = ax.get_xlim()
ymin, ymax = ax.get_ylim()

# add basemap and format
ctx.add_basemap(ax, source=ctx.providers.CartoDB.Positron)
ax.set_title('Accident Density Map: California', fontsize=14)
ax.set_axis_off()
plt.tight_layout()
plt.show()
```

Accident Density Map: California

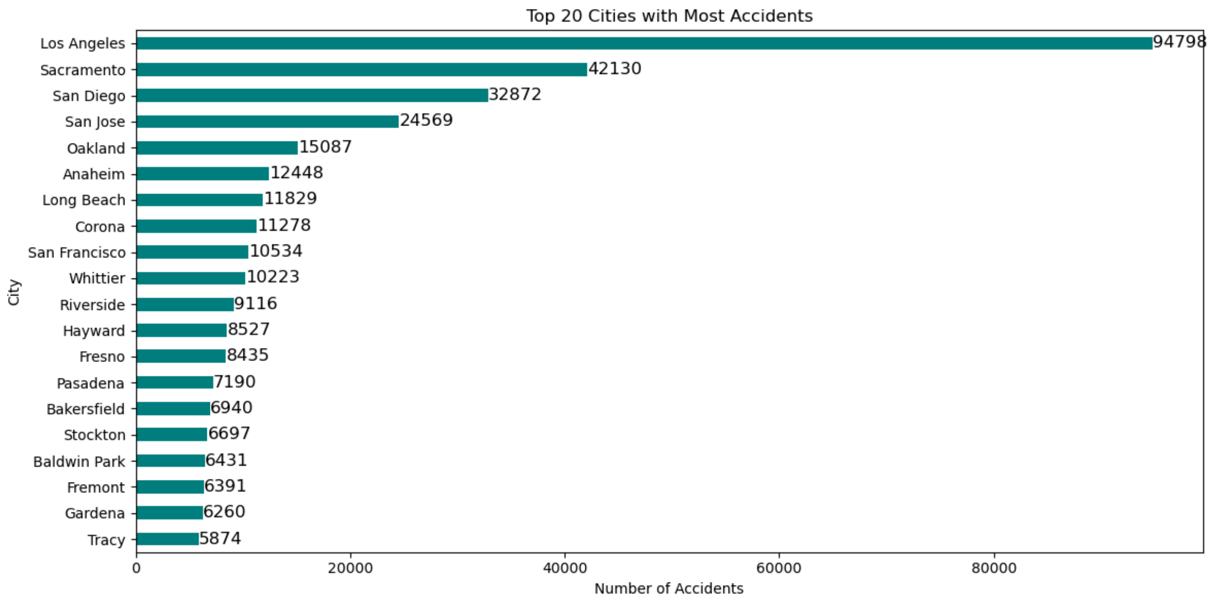


It is clear that the accidents are not distributed homogeneously distributed in California. We see that they are concentrated in specific regions. For example, Los Angeles and the Bay Area show clear spikes. Moreover, we see accidents along lines through the state, probably indicating long roads. We confirm this with further visualisations.

In [69]:

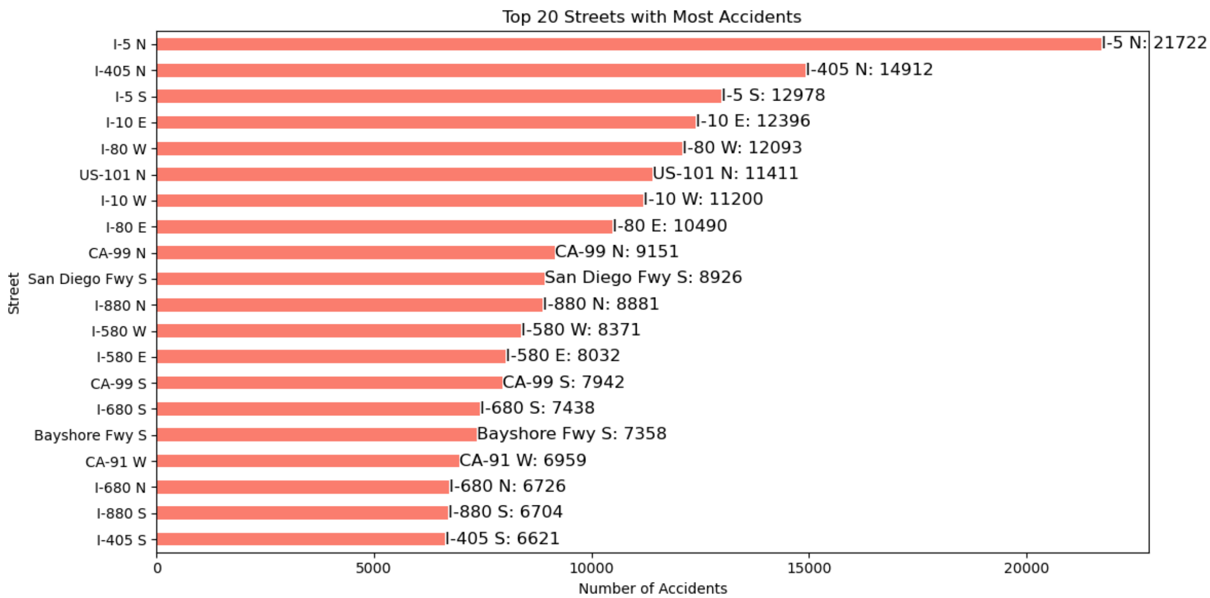
```
# get top 20 cities by accident count and turn it into a proper DataFrame
top_cities_df = df_cleaned['City'].value_counts().head(20).reset_index()
top_cities_df.columns = ['City', 'Accident_Count']

top_cities_df = top_cities_df.sort_values('Accident_Count', ascending=True)

ax = top_cities_df.plot(kind='barh', x='City', y='Accident_Count', legend=Fa

# annotate each bar with the exact accident count
for index, value in enumerate(top_cities_df['Accident_Count']):
```

```
    ax.text(value, index, str(value), va='center', ha='left', color='black',

plt.title("Top 20 Cities with Most Accidents")
plt.xlabel("Number of Accidents")
plt.ylabel("City")
plt.tight_layout()

plt.show()
```



The city distribution shows clear imbalance in distributions, and this can be used as a foundation for our analysis.

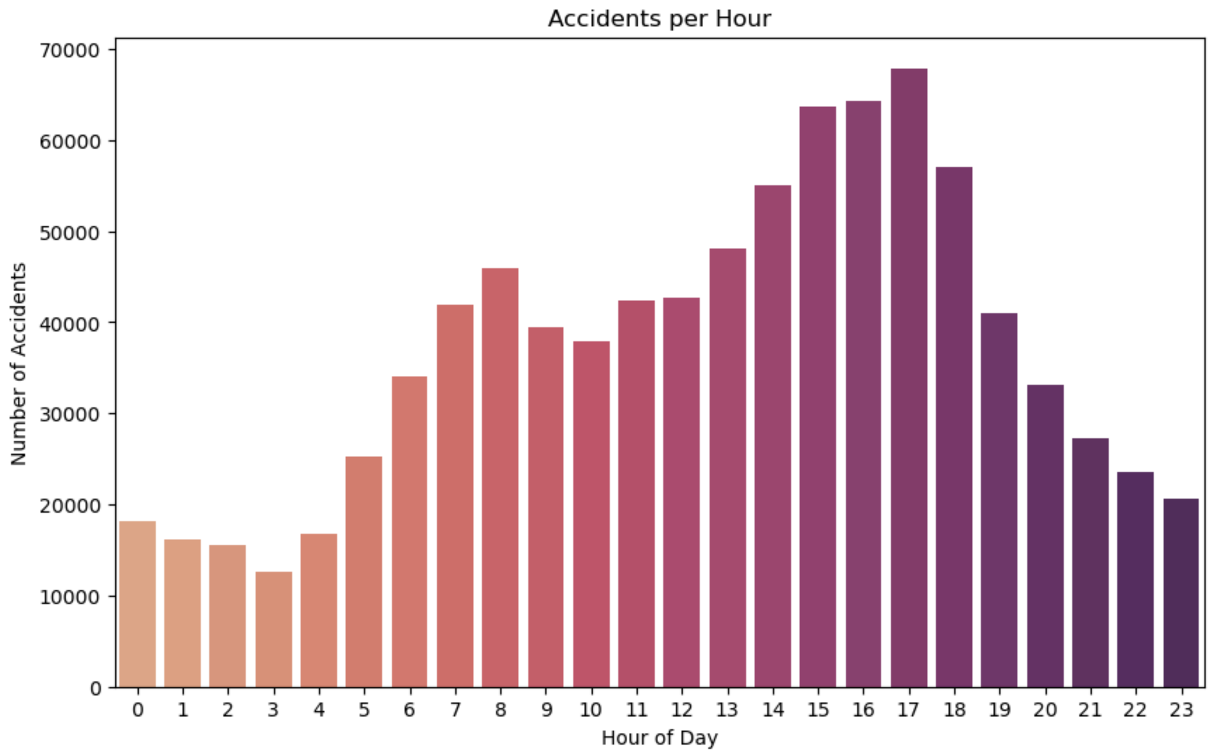
In [71]:
```
# get top 20 streets by accident count and turn it into a proper DataFrame
top_streets_df = df_cleaned['Street'].value_counts().head(20).reset_index()
top_streets_df.columns = ['Street', 'Accident_Count']

# sort by 'Accident_Count' in descending order
top_streets_df = top_streets_df.sort_values('Accident_Count', ascending=True

# slot the bar chart (in ascending order, so the highest will be at the top)
ax = top_streets_df.plot(kind='barh', x='Street', y='Accident_Count', legend

# annotate each bar with the street name and the exact accident count
for index, value in enumerate(top_streets_df['Accident_Count']):
    street_name = top_streets_df['Street'].iloc[index]
    ax.text(value, index, f'{street_name}: {value}', va='center', ha='left',

plt.title("Top 20 Streets with Most Accidents")
plt.xlabel("Number of Accidents")
plt.ylabel("Street")
plt.tight_layout()

plt.show()
```



The split by street is a bit more homogeneous, and this can potentially be a weakness as there might be very high cardinality, inhibiting trend detection in our model. We have 60000 streets, and this will pose a problem in our model later.

In [73]:
```
# plot: accidents per hour (ensure hour is 0–23, sorted)
plt.figure(figsize=(10, 6))
sb.countplot(
    x='Start_Hour',
    data=df_cleaned,
    order=sorted(df_cleaned['Start_Hour'].unique()),
    palette='flare'
)
plt.title("Accidents per Hour")
plt.xlabel("Hour of Day")
plt.ylabel("Number of Accidents")
plt.xticks(ticks=range(24), labels=[str(i) for i in range(24)])  # force 0–2
plt.show()
```

```
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\3386055065.py:3: FutureWa
rning:

Passing `palette` without assigning `hue` is deprecated and will be removed
in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the
same effect.

  sb.countplot(
```

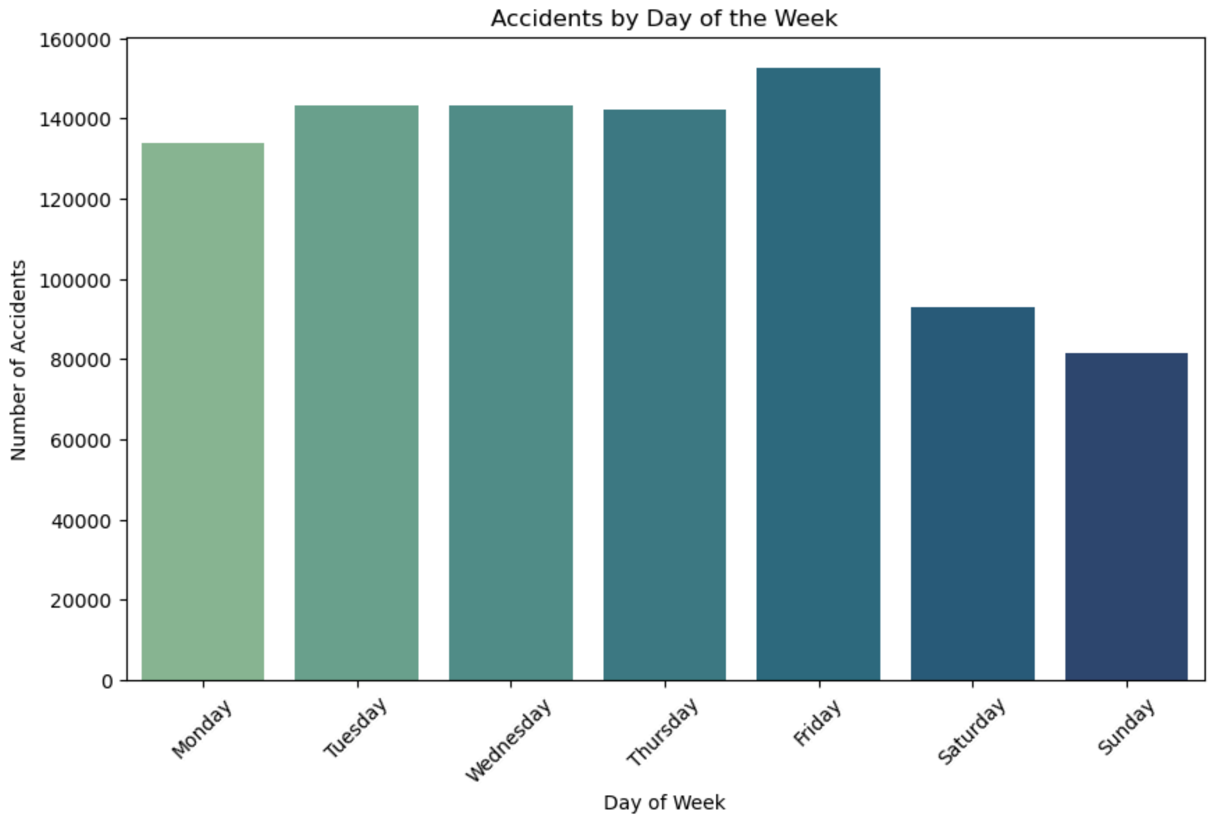


We see a local spike between 7AM and 9AM, and a global spike between 4PM and 7PM. These are peak hours, and clearly this is another dimension for our analysis.

In [76]:

```
# extract day of week from Start_Time
df_cleaned['Day_of_Week'] = df_cleaned['Start_Time'].dt.day_name()

# plot: Accidents by Day of the Week
plt.figure(figsize=(10, 6))

sb.countplot(
    x='Day_of_Week',
    data=df_cleaned,
    order=['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday
    palette='crest'
)
plt.title("Accidents by Day of the Week")
plt.xlabel("Day of Week")
plt.ylabel("Number of Accidents")
plt.xticks(rotation=45)
plt.show()
```

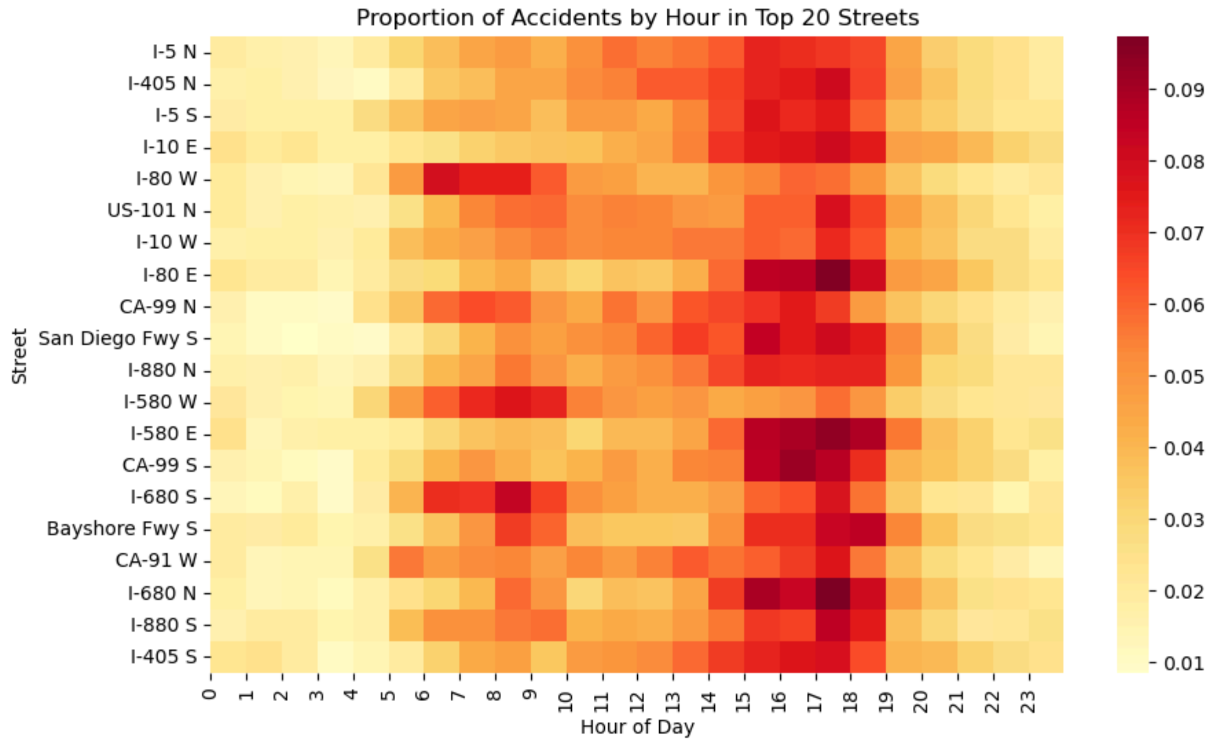
```
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\3276437083.py:7: FutureWa
rning:

Passing `palette` without assigning `hue` is deprecated and will be removed
in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the
same effect.

  sb.countplot(
```



We also see a clear distinction between weekdays and weekends. A specific spike on Fridays, possibly an indicator of more haphazard driving. However, since we are trying to identify trends across both time and location, we will cross validate these trends.

In [78]:
```
# group by Street and Start_Hour and calculate the count of accidents
time_location_street = df.groupby(['Street', 'Start_Hour']).size().unstack(f

# calculate the total accidents in each street
street_totals = time_location_street.sum(axis=1)

# normalize the counts to proportions by dividing by the total accidents in
time_location_proportion_street = time_location_street.div(street_totals, ax

# use top 20 streets
top20_streets = df['Street'].value_counts().head(20).index
sb.heatmap(time_location_proportion_street.loc[top20_streets], cmap="YlOrRd"
plt.title("Proportion of Accidents by Hour in Top 20 Streets")
plt.xlabel("Hour of Day")
plt.ylabel("Street")
plt.xticks(ticks=range(0, 24), labels=[str(i) for i in range(24)])  # Ensure
plt.show()
```

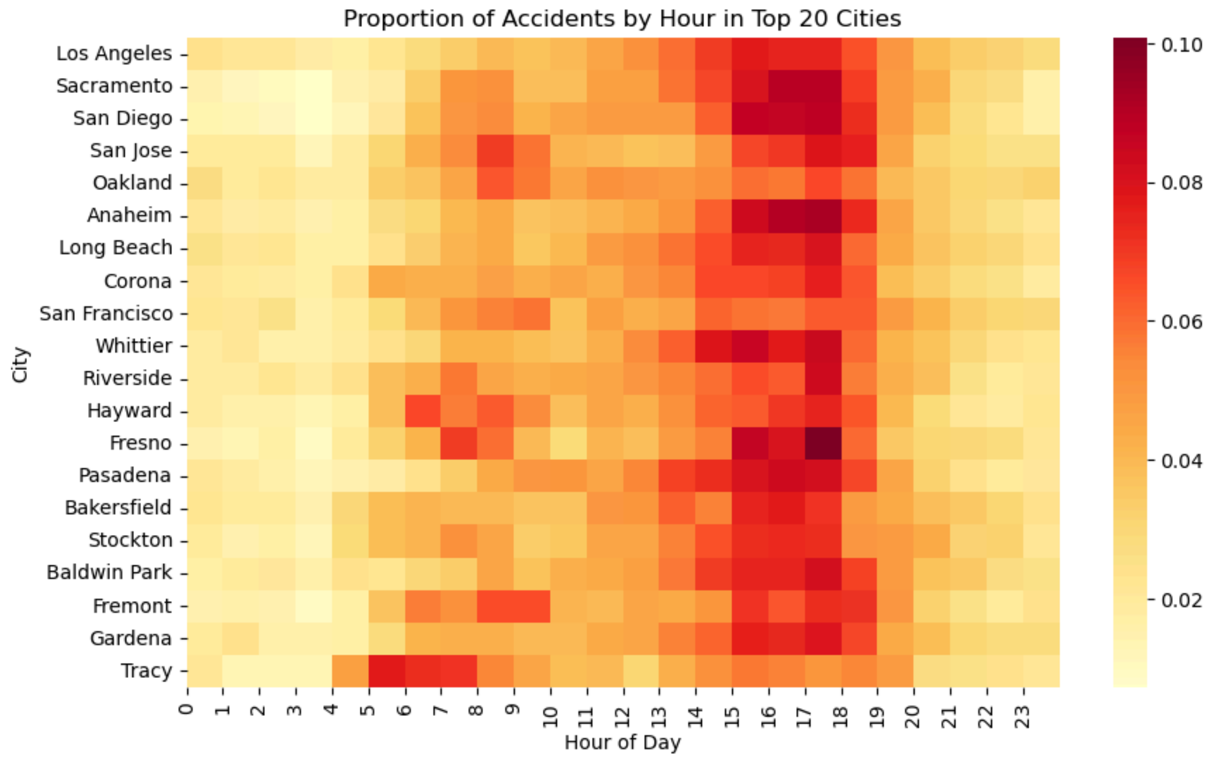
```
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\1582233670.py:2: FutureWa
rning: The default of observed=False is deprecated and will be changed to Tr
ue in a future version of pandas. Pass observed=False to retain current beha
vior or observed=True to adopt the future default and silence this warning.
  time_location_street = df.groupby(['Street', 'Start_Hour']).size().unstack
(fill_value=0)
```



In [79]:

```
# group by City and Start_Hour and calculate the count of accidents
time_location = df.groupby(['City', 'Start_Hour']).size().unstack(fill_value

# calculate the total accidents in each city
city_totals = time_location.sum(axis=1)

# normalize the counts to proportions by dividing by the total accidents in
time_location_proportion = time_location.div(city_totals, axis=0)

# use top 20 cities
top20 = df['City'].value_counts().head(20).index
sb.heatmap(time_location_proportion.loc[top20], cmap="YlOrRd", annot=False)
plt.title("Proportion of Accidents by Hour in Top 20 Cities")
plt.xlabel("Hour of Day")
plt.ylabel("City")
plt.xticks(ticks=range(0, 24), labels=[str(i) for i in range(24)])  # Ensure
plt.show()
```

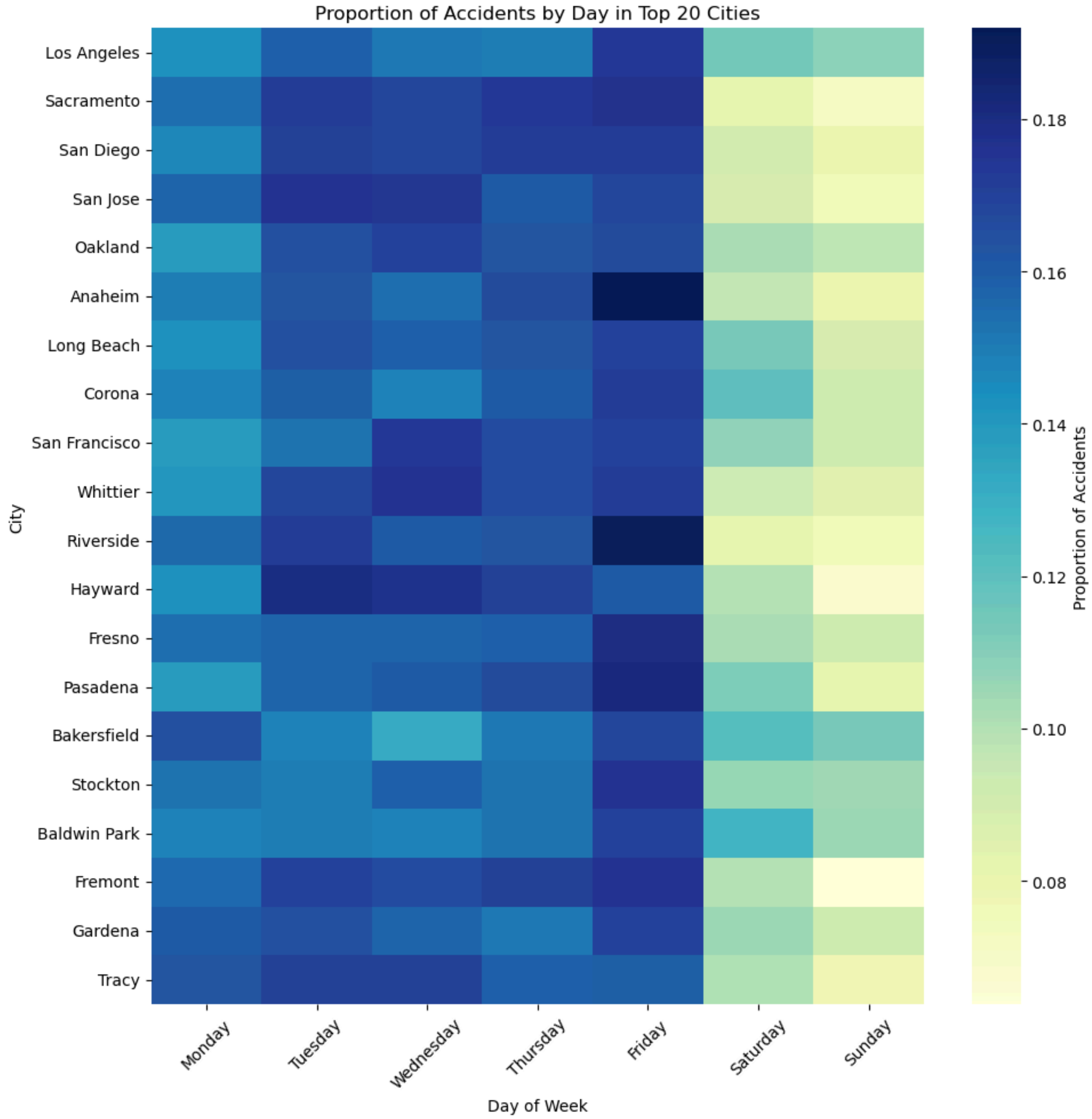
```
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\835226610.py:2: FutureWar
ning: The default of observed=False is deprecated and will be changed to Tru
e in a future version of pandas. Pass observed=False to retain current behav
ior or observed=True to adopt the future default and silence this warning.
  time_location = df.groupby(['City', 'Start_Hour']).size().unstack(fill_val
ue=0)
```



Both heatmaps validate there is a trend across all the different cities and streets, justifying using these as dimensions for our core analysis.

In [81]:
```
# group by city and day of week, count accidents
df_cleaned['Day_of_Week'] = df_cleaned['Start_Time'].dt.day_name()
city_day = df_cleaned.groupby(['City', 'Day_of_Week']).size().unstack(fill_v

# normalize row-wise to get proportions per city
city_day_prop = city_day.div(city_day.sum(axis=1), axis=0)

# reorder columns to match actual day order
ordered_days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Sat
city_day_prop = city_day_prop[ordered_days]

# take top 20 cities by total accidents
top20_cities = df_cleaned['City'].value_counts().head(20).index
city_day_top20 = city_day_prop.loc[top20_cities]

# plot heatmap
plt.figure(figsize=(10, 10))
sb.heatmap(city_day_top20, cmap='YlGnBu', cbar_kws={'label': 'Proportion of
plt.title("Proportion of Accidents by Day in Top 20 Cities")
plt.xlabel("Day of Week")
plt.ylabel("City")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

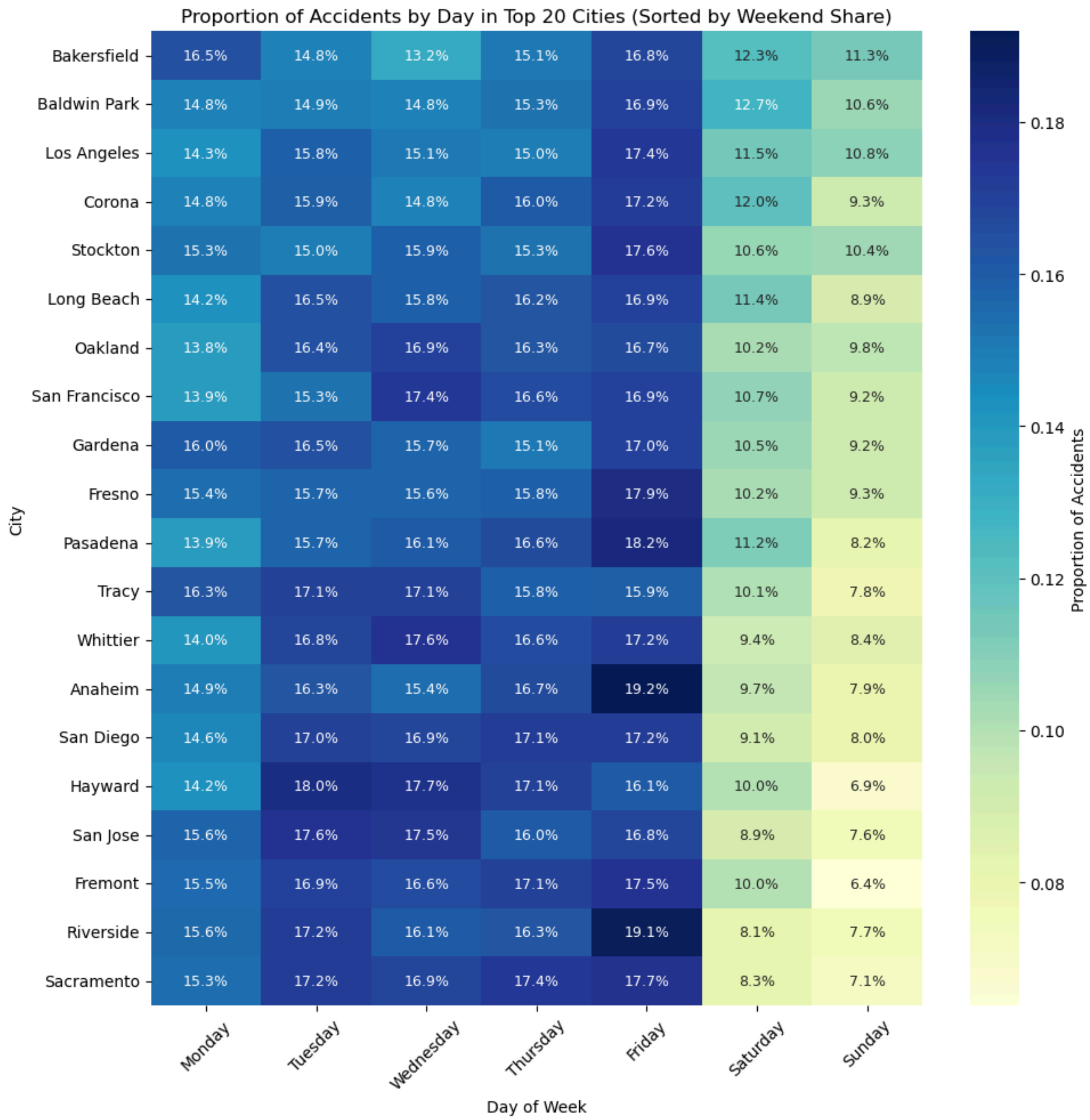
```
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\550648925.py:3: FutureWar
ning: The default of observed=False is deprecated and will be changed to Tru
e in a future version of pandas. Pass observed=False to retain current behav
ior or observed=True to adopt the future default and silence this warning.
  city_day = df_cleaned.groupby(['City', 'Day_of_Week']).size().unstack(fill
_value=0)
```



The day trends are relevant to the cities with largest number of accidents, validating our focus on it.

In [83]:
```
# sort by total weekend proportion
city_day_top20['Weekend'] = city_day_top20['Saturday'] + city_day_top20['Sur
city_day_top20_sorted = city_day_top20.sort_values(by='Weekend', ascending=F

# create annotation DataFrame with formatted strings
annot = (city_day_top20_sorted * 100).round(1).astype(str) + '%'
annot = annot.values  # Ensure it's a NumPy array
```

```
plt.figure(figsize=(10, 10))
sb.heatmap(city_day_top20_sorted, cmap='YlGnBu', cbar_kws={'label': 'Proport
           annot=annot, fmt='', annot_kws={"size": 9})
plt.title("Proportion of Accidents by Day in Top 20 Cities (Sorted by Weeker
plt.xlabel("Day of Week")
plt.ylabel("City")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

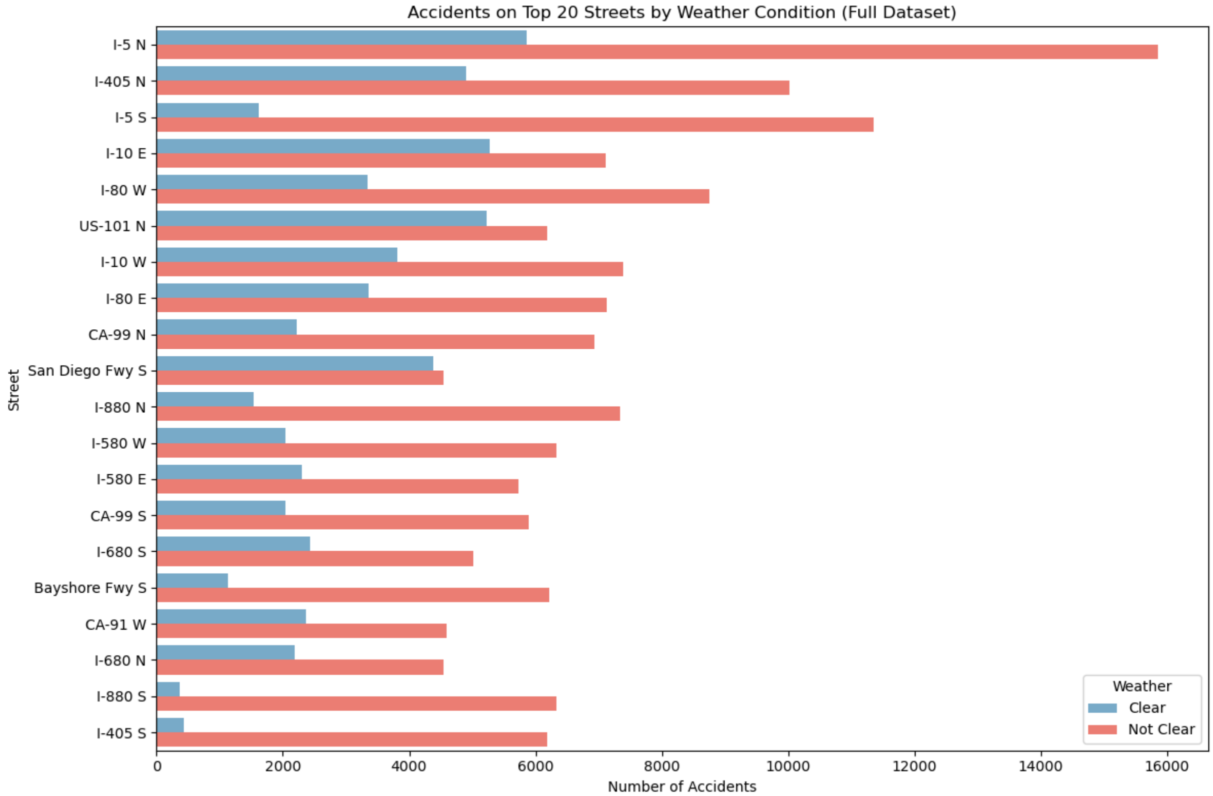


The above also validates that the trend is seen across all cities. Bakersfield has the most even distribution, but even it shows some differentiation. We will continue to look at this trend in our analysis.

The below shows us the weather analysis. We wish to see if all the accidents in specific streets occur in adverse conditions. This will assist in classifying if the streets are dangerous only in adverse weather, or if they are dangerous just in general.

In [86]:
```
# create Weather_Category based on Is_Clear flag
df['Weather_Category'] = df['Is_Clear'].apply(lambda x: 'Clear' if x == 1 el

# get top 20 streets by accident count
top_streets = df['Street'].value_counts().head(20).index

# filter to those streets
df_top_streets = df[df['Street'].isin(top_streets)].copy()

# createountplot
plt.figure(figsize=(12, 8))
ax = sb.countplot(
    data=df_top_streets,
    y='Street',
    hue='Weather_Category',
    order=top_streets,
    palette=['#6baed6', '#ff6f61']  # Clear = blue, Not Clear = red
)

# add percentages to the bars
# get counts grouped by street and weather
counts = (
    df_top_streets.groupby(['Street', 'Weather_Category'])
    .size()
    .unstack(fill_value=0)
)

# add annotation on bars
for i, street in enumerate(top_streets):
    total = counts.loc[street].sum()
    clear_count = counts.loc[street].get('Clear', 0)
    not_clear_count = counts.loc[street].get('Not Clear', 0)

    # get bar coordinates and add text
    bar_clear = ax.patches[i * 2]  # Each street has two bars (hue)
    bar_not_clear = ax.patches[i * 2 + 1]

plt.title("Accidents on Top 20 Streets by Weather Condition (Full Dataset)")
plt.xlabel("Number of Accidents")
plt.ylabel("Street")
plt.legend(title="Weather")
plt.tight_layout()
plt.show()
```

```
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\3915172420.py:23: FutureW
arning: The default of observed=False is deprecated and will be changed to T
rue in a future version of pandas. Pass observed=False to retain current beh
avior or observed=True to adopt the future default and silence this warning.
  df_top_streets.groupby(['Street', 'Weather_Category'])
```



We see that unclear weather contributes heavily to the occurrence of accidents, making it a key factor that we wish to consider. The different factors of weather can be analysed to make predictions. For streets that show more evenness, we can exclude them.

To conclude, our three dimensions of region, time+day and weather are all validated. We narrow down specifically to investigate cities and streets for region, peak hour trends and weekday-weekend splits for time+day, and indicators of adverse weather.

Machine Learning

Now that we have performed our EDA on the dataset, we understand a few key trends better. It is evident that there are a few key cities and regions that experience significantly higher volumes of accidents than others. We've also identified a trend throughout the day where accidents spike during peak hours and seem to be higher during the peak hours in the evening as opposed to the morning.

Understanding these trends, we now want to apply our dataset into some Machine Learning applications. There were a few ideas that we considered as outputs from our algorithms.

1. Binary classification on whether a condition would result in accidents

2. Time series forecast to predict the rate of accidents in the next hour at a given hour, region and weather
3. Severity prediction for an accident based on time, region and weather

Unfortunately, as we are working with a positives-only dataset, it would be impossible to create an algorithm that could perform the binary classification we would want in 1 unless we merged our dataset with another. Furthermore, as seen above, only one unique value of severity of accident exists making it impossible to conduct classification on the severity. Thus, we will be going ahead to create algorithms for objectives 2 only.

Thus, let us tackle algorithm 2, predicting the rates of accidents in the upcoming hour based on the location, weather, time and day of the week.

To make the dataset easier to work with for our models, let us first remove all unnecessary columns. We will also work with cities first as aggregating by city would be easier to work with for the localised prediction.

In [93]:
```
columns_to_keep = [
    'Start_Time',  # Time information
    'City', #City
    'Temperature(F)',  # Weather data
    'Humidity(%)',  # Weather data
    'Pressure(in)', #Pressure data
    'Wind_Speed(mph)',  # Weather data
    'Start_Hour',  # Extracted from Start_Time
    'Day_of_Week',  # Extracted from Start_Time
    'Is_Windy',
    'Is_Stormy',
    'Is_Rainy',
    'Is_Foggy',
    'Is_Snowy',
    'Is_Clear'
]

# Drop all other columns
df_algo1 = df_cleaned[columns_to_keep]
```

In [94]:
```
df_algo1.head()
```

Out[94]:

	Start_Time	City	Temperature(F)	Humidity(%)	Pressure(in)	Wind_Spe
0	2016-06-21 10:34:40	Vallejo	0.669118	0.474747	0.633588	
1	2016-06-21 10:30:16	Hayward	0.669118	0.474747	0.679389	
2	2016-06-21 10:49:14	Walnut Creek	0.785294	0.303030	0.610687	
3	2016-06-21 10:41:42	Cupertino	0.682353	0.474747	0.664122	
4	2016-06-21 10:16:26	San Jose	0.672059	0.404040	0.679389	

After reducing the dataset to focus on our key predictors, we now need to create the target variable for predicting the rate of accidents in the next hour.

In [153…

```
# Step 1: Floor to hour
df_algo1['Start_Hourly'] = df_algo1['Start_Time'].dt.floor('h')

# Step 2: Create "Next Hour"
df_algo1['Next_Hourly'] = df_algo1['Start_Hourly'] + pd.Timedelta(hours=1)

# Step 3: Count future accidents per city per hour
city_hourly_counts = df_algo1.groupby(['City', 'Start_Hourly']).size().renam

# Step 4: Shift within each city to get *next hour's* count
city_hourly_next = city_hourly_counts.groupby(level=0).shift(-1).rename('Acc

# Step 5: Combine the City/Hour index with the shifted counts
target_df = pd.concat([city_hourly_counts, city_hourly_next], axis=1).reset_

# Step 6: Merge back into main df
df_algo1 = df_algo1.merge(target_df[['City', 'Start_Hourly', 'Accident_Next_
              on=['City', 'Start_Hourly'], how='left')


print(df_algo1.head(3))
```

```
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\3585540051.py:2: SettingW
ithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/
stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_algo1['Start_Hourly'] = df_algo1['Start_Time'].dt.floor('h')
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\3585540051.py:5: SettingW
ithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/
stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_algo1['Next_Hourly'] = df_algo1['Start_Hourly'] + pd.Timedelta(hours=1)
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\3585540051.py:8: FutureWa
rning: The default of observed=False is deprecated and will be changed to Tr
ue in a future version of pandas. Pass observed=False to retain current beha
vior or observed=True to adopt the future default and silence this warning.
  city_hourly_counts = df_algo1.groupby(['City', 'Start_Hourly']).size().ren
ame('Accident_Count')
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\3585540051.py:11: FutureW
arning: The default of observed=False is deprecated and will be changed to T
rue in a future version of pandas. Pass observed=False to retain current beh
avior or observed=True to adopt the future default and silence this warning.
  city_hourly_next = city_hourly_counts.groupby(level=0).shift(-1).rename('A
ccident_Next_Hour')
           Start_Time          City  Temperature(F)  Humidity(%)  \
0 2016-06-21 10:34:40       Vallejo        0.669118     0.474747
1 2016-06-21 10:30:16       Hayward        0.669118     0.474747
2 2016-06-21 10:49:14  Walnut Creek        0.785294     0.303030

   Pressure(in)  Wind_Speed(mph)  Start_Hour Day_of_Week  Is_Windy  Is_Storm
y  \
0      0.633588         0.322222    0.434783     Tuesday         0
0
1      0.679389         0.255556    0.434783     Tuesday         0
0
2      0.610687         0.255556    0.434783     Tuesday         0
0

   Is_Rainy  Is_Foggy  Is_Snowy  Is_Clear        Start_Hourly  \
0         0         0         0         1 2016-06-21 10:00:00
1         0         0         0         1 2016-06-21 10:00:00
2         0         0         0         1 2016-06-21 10:00:00

          Next_Hourly  Accident_Next_Hour
0 2016-06-21 11:00:00                 0.0
1 2016-06-21 11:00:00                 0.0
2 2016-06-21 11:00:00                 0.0
```

In [155…
```
df_algo1.head()
```

Out[155…

	Start_Time	City	Temperature(F)	Humidity(%)	Pressure(in)	Wind_Spe
0	2016-06-21 10:34:40	Vallejo	0.669118	0.474747	0.633588	
1	2016-06-21 10:30:16	Hayward	0.669118	0.474747	0.679389	
2	2016-06-21 10:49:14	Walnut Creek	0.785294	0.303030	0.610687	
3	2016-06-21 10:41:42	Cupertino	0.682353	0.474747	0.664122	
4	2016-06-21 10:16:26	San Jose	0.672059	0.404040	0.679389	

In [157…

```
# Step 1: Make sure your time is rounded to the hour
df_algo1['Start_Hourly'] = df_algo1['Start_Time'].dt.floor('H')

# Step 2: Count number of accidents per city per hour
city_hourly_counts = df_algo1.groupby(['City', 'Start_Hourly']).size().renam

# Step 3: Create the lag (previous hour's accident count)
city_hourly_counts_lag = city_hourly_counts.groupby(level=0).shift(1).rename

# Step 4: Combine the counts into a DataFrame for merging
lag_features = pd.concat([city_hourly_counts, city_hourly_counts_lag], axis=

# Step 5: Merge lag features into main DataFrame
df_algo1 = df_algo1.merge(lag_features, how='left', on=['City', 'Start_Hourl
```

```
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\951553649.py:2: FutureWar
ning: 'H' is deprecated and will be removed in a future version, please use
'h' instead.
  df_algo1['Start_Hourly'] = df_algo1['Start_Time'].dt.floor('H')
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\951553649.py:5: FutureWar
ning: The default of observed=False is deprecated and will be changed to Tru
e in a future version of pandas. Pass observed=False to retain current behav
ior or observed=True to adopt the future default and silence this warning.
  city_hourly_counts = df_algo1.groupby(['City', 'Start_Hourly']).size().ren
ame('Accident_Current_Hour')
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\951553649.py:8: FutureWar
ning: The default of observed=False is deprecated and will be changed to Tru
e in a future version of pandas. Pass observed=False to retain current behav
ior or observed=True to adopt the future default and silence this warning.
  city_hourly_counts_lag = city_hourly_counts.groupby(level=0).shift(1).rena
me('Accident_Prev_Hour')
```

Since there is the cardinality of the City variable is very high, it would not be possible to convert it with One-Hot Encoding. Additionally, if label encoding is used, patterns about the distance between the numbers could be identified when in reality there is no pattern between the integers given to each city. Thus, we will use a different method called Target Encoding instead.

In [159…

```
# Calculate the mean of the target variable for each city
city_target_mean = df_algo1.groupby('City')['Accident_Next_Hour'].mean()

# Map the mean target value to the 'City' column
df_algo1['City_Encoded'] = df_algo1['City'].map(city_target_mean)

# Check the result
print(df_algo1.head(20))
```

	Start_Time	City	Temperature(F)	Humidity(%)	\
0	2016-06-21 10:34:40	Vallejo	0.669118	0.474747	
1	2016-06-21 10:30:16	Hayward	0.669118	0.474747	
2	2016-06-21 10:49:14	Walnut Creek	0.785294	0.303030	
3	2016-06-21 10:41:42	Cupertino	0.682353	0.474747	
4	2016-06-21 10:16:26	San Jose	0.672059	0.404040	
5	2016-06-21 10:31:06	Pleasanton	0.785294	0.232323	
6	2016-06-21 10:17:17	San Jose	0.639706	0.525253	
7	2016-06-21 10:51:31	San Francisco	0.611765	0.545455	
8	2016-06-21 10:56:00	Orinda	0.785294	0.303030	
9	2016-06-21 10:57:01	Livermore	0.785294	0.232323	
10	2016-06-21 11:02:59	Cotati	0.751471	0.383838	
11	2016-06-21 11:06:50	Concord	0.785294	0.303030	
12	2016-06-21 11:11:50	Livermore	0.785294	0.232323	
13	2016-06-21 11:13:06	Sunnyvale	0.682353	0.474747	
14	2016-06-21 11:19:24	Dublin	0.785294	0.232323	
15	2016-06-21 11:16:14	Sacramento	0.830882	0.252525	
16	2016-06-21 11:16:55	Sacramento	0.830882	0.191919	
17	2016-06-21 11:20:47	Sunnyvale	0.682353	0.474747	
18	2016-06-21 11:37:40	Oakland	0.652941	0.494949	
19	2016-06-21 11:33:06	Alviso	0.772059	0.282828	

	Pressure(in)	Wind_Speed(mph)	Start_Hour	Day_of_Week	Is_Windy	\
0	0.633588	0.322222	0.434783	Tuesday	0	
1	0.679389	0.255556	0.434783	Tuesday	0	
2	0.610687	0.255556	0.434783	Tuesday	0	
3	0.664122	0.255556	0.434783	Tuesday	0	
4	0.679389	0.322222	0.434783	Tuesday	0	
5	0.633588	0.322222	0.434783	Tuesday	0	
6	0.664122	0.255556	0.434783	Tuesday	0	
7	0.664122	0.255556	0.434783	Tuesday	0	
8	0.610687	0.255556	0.434783	Tuesday	0	
9	0.633588	0.322222	0.434783	Tuesday	0	
10	0.664122	0.322222	0.478261	Tuesday	0	
11	0.610687	0.255556	0.478261	Tuesday	0	
12	0.633588	0.322222	0.478261	Tuesday	0	
13	0.664122	0.255556	0.478261	Tuesday	0	
14	0.633588	0.322222	0.478261	Tuesday	0	
15	0.633588	0.511111	0.478261	Tuesday	0	
16	0.641221	0.511111	0.478261	Tuesday	0	
17	0.664122	0.255556	0.478261	Tuesday	0	
18	0.656489	0.450000	0.478261	Tuesday	0	
19	0.656489	0.322222	0.478261	Tuesday	0	

	Is_Stormy	Is_Rainy	Is_Foggy	Is_Snowy	Is_Clear	Start_Hourly	\
0	0	0	0	0	1	2016-06-21 10:00:00	
1	0	0	0	0	1	2016-06-21 10:00:00	
2	0	0	0	0	1	2016-06-21 10:00:00	
3	0	0	0	0	1	2016-06-21 10:00:00	
4	0	0	0	0	1	2016-06-21 10:00:00	
5	0	0	0	0	1	2016-06-21 10:00:00	
6	0	0	0	0	0	2016-06-21 10:00:00	
7	0	0	0	0	0	2016-06-21 10:00:00	
8	0	0	0	0	1	2016-06-21 10:00:00	
9	0	0	0	0	1	2016-06-21 10:00:00	
10	0	0	0	0	1	2016-06-21 11:00:00	

```
11          0          0          0          0          1 2016-06-21 11:00:00
12          0          0          0          0          1 2016-06-21 11:00:00
13          0          0          0          0          1 2016-06-21 11:00:00
14          0          0          0          0          1 2016-06-21 11:00:00
15          0          0          0          0          1 2016-06-21 11:00:00
16          0          0          0          0          1 2016-06-21 11:00:00
17          0          0          0          0          1 2016-06-21 11:00:00
18          0          0          0          0          0 2016-06-21 11:00:00
19          0          0          0          0          1 2016-06-21 11:00:00

           Next_Hourly  Accident_Next_Hour  Accident_Current_Hour  \
0  2016-06-21 11:00:00                 0.0                      1
1  2016-06-21 11:00:00                 0.0                      1
2  2016-06-21 11:00:00                 0.0                      2
3  2016-06-21 11:00:00                 0.0                      1
4  2016-06-21 11:00:00                 2.0                      2
5  2016-06-21 11:00:00                 1.0                      1
6  2016-06-21 11:00:00                 2.0                      2
7  2016-06-21 11:00:00                 0.0                      1
8  2016-06-21 11:00:00                 0.0                      1
9  2016-06-21 11:00:00                 1.0                      1
10 2016-06-21 12:00:00                 0.0                      1
11 2016-06-21 12:00:00                 0.0                      1
12 2016-06-21 12:00:00                 1.0                      1
13 2016-06-21 12:00:00                 0.0                      2
14 2016-06-21 12:00:00                 0.0                      1
15 2016-06-21 12:00:00                 0.0                      2
16 2016-06-21 12:00:00                 0.0                      2
17 2016-06-21 12:00:00                 0.0                      2
18 2016-06-21 12:00:00                 0.0                      1
19 2016-06-21 12:00:00                 0.0                      1

    Accident_Prev_Hour  City_Encoded
0                  0.0      0.153049
1                  0.0      0.360971
2                  0.0      0.253292
3                  0.0      0.133208
4                  0.0      0.906101
5                  0.0      0.185723
6                  0.0      0.906101
7                  2.0      0.412664
8                  0.0      0.133943
9                  0.0      0.251729
10                 0.0      0.028391
11                 0.0      0.155421
12                 1.0      0.251729
13                 0.0      0.156306
14                 0.0      0.094972
15                 0.0      1.967814
16                 0.0      1.967814
17                 0.0      0.156306
18                 0.0      0.565056
19                 0.0      0.012987
```

```
C:\Users\Joanna\AppData\Local\Temp\ipykernel_15908\1661754449.py:2: FutureWarning: The default of observed=False is deprecated and will be changed to True in a future version of pandas. Pass observed=False to retain current behavior or observed=True to adopt the future default and silence this warning.
  city_target_mean = df_algo1.groupby('City')['Accident_Next_Hour'].mean()
```

Since the Day_of_Week currently exists as strings, we shall apply a simple ordinal encoding to allow it to be processed by the algorithms.

In [161…

```
day_of_week_map = {
    'Monday': 0,
    'Tuesday': 1,
    'Wednesday': 2,
    'Thursday': 3,
    'Friday': 4,
    'Saturday': 5,
    'Sunday': 6
}

# Apply the mapping to the 'Day_of_Week' column
df_algo1['Day_of_Week'] = df_algo1['Day_of_Week'].map(day_of_week_map)

# Check the result
print(df_algo1[['Day_of_Week']].head())
```

```
   Day_of_Week
0            1
1            1
2            1
3            1
4            1
```

In [163…

```
# Drop rows where target is NaN
df_algo1 = df_algo1.dropna(subset=['Accident_Prev_Hour'])
df_algo1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 889958 entries, 0 to 889961
Data columns (total 20 columns):
 #   Column                 Non-Null Count   Dtype
---  ------                 --------------   -----
 0   Start_Time             889958 non-null  datetime64[ns]
 1   City                   889958 non-null  category
 2   Temperature(F)         889958 non-null  float64
 3   Humidity(%)            889958 non-null  float64
 4   Pressure(in)           889958 non-null  float64
 5   Wind_Speed(mph)        889958 non-null  float64
 6   Start_Hour             889958 non-null  float64
 7   Day_of_Week            889958 non-null  int64
 8   Is_Windy               889958 non-null  int32
 9   Is_Stormy              889958 non-null  int32
 10  Is_Rainy               889958 non-null  int32
 11  Is_Foggy               889958 non-null  int32
 12  Is_Snowy               889958 non-null  int32
 13  Is_Clear               889958 non-null  int32
 14  Start_Hourly           889958 non-null  datetime64[ns]
 15  Next_Hourly            889958 non-null  datetime64[ns]
 16  Accident_Next_Hour     889956 non-null  float64
 17  Accident_Current_Hour  889958 non-null  int64
 18  Accident_Prev_Hour     889958 non-null  float64
 19  City_Encoded           889958 non-null  float64
dtypes: category(1), datetime64[ns](3), float64(8), int32(6), int64(2)
memory usage: 117.2 MB
```

In [165…

```
df_algo1.head()
```

Out[165…

	Start_Time	City	Temperature(F)	Humidity(%)	Pressure(in)	Wind_Spe
0	2016-06-21 10:34:40	Vallejo	0.669118	0.474747	0.633588	
1	2016-06-21 10:30:16	Hayward	0.669118	0.474747	0.679389	
2	2016-06-21 10:49:14	Walnut Creek	0.785294	0.303030	0.610687	
3	2016-06-21 10:41:42	Cupertino	0.682353	0.474747	0.664122	
4	2016-06-21 10:16:26	San Jose	0.672059	0.404040	0.679389	

Finally, since the hours of the day are cyclical, let's convert our start_hour to represent the cyclical nature instead.

In [168…

```
# Hour of the day (0 to 23)
df_algo1['Start_Hour'] = df_algo1['Start_Time'].dt.hour

# Convert hour into sine and cosine to capture cyclic nature
df_algo1['Hour_Sin'] = np.sin(2 * np.pi * df_algo1['Start_Hour'] / 24)
df_algo1['Hour_Cos'] = np.cos(2 * np.pi * df_algo1['Start_Hour'] / 24)
```

```python
df_algo1.head()
```

Out[168…

	Start_Time	City	Temperature(F)	Humidity(%)	Pressure(in)	Wind_Sp
0	2016-06-21 10:34:40	Vallejo	0.669118	0.474747	0.633588	
1	2016-06-21 10:30:16	Hayward	0.669118	0.474747	0.679389	
2	2016-06-21 10:49:14	Walnut Creek	0.785294	0.303030	0.610687	
3	2016-06-21 10:41:42	Cupertino	0.682353	0.474747	0.664122	
4	2016-06-21 10:16:26	San Jose	0.672059	0.404040	0.679389	

5 rows × 22 columns

In [170…

```python
df_algo1.describe()
```

Out[170…

	Start_Time	Temperature(F)	Humidity(%)	Pressure(in)	Win
count	889958	889958.000000	889958.000000	889958.000000	
mean	2019-10-01 16:53:32.761748480	0.519540	0.560466	0.551692	
min	2016-03-22 20:00:34	0.000000	0.000000	0.000000	
25%	2018-01-22 21:57:40	0.404412	0.404040	0.488550	
50%	2019-12-10 18:34:10	0.507353	0.595960	0.564885	
75%	2021-05-24 13:37:53.500000	0.625000	0.727273	0.648855	
max	2023-03-31 23:25:30	1.000000	1.000000	1.000000	
std	NaN	0.176642	0.215994	0.157582	

8 rows × 21 columns

In [171…

```python
X = df_algo1[['Temperature(F)', 'Humidity(%)', 'Wind_Speed(mph)', 'Pressure(
    'Is_Stormy',
    'Is_Rainy',
    'Is_Foggy',
    'Is_Snowy',
    'Is_Clear',
    'Hour_Sin', 'Hour_Cos',
    'Accident_Prev_Hour'] ]
y = df_algo1['Accident_Next_Hour']
```

```
from sklearn.model_selection import train_test_split
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, shu
```

We will be testing 3 different algorithms to determine which will perform the best.

1. Random Forest Regressor
2. Gradient Boosting Regressor
3. Linear Regression

We will define hyperparameter grids to tune the hyperparameters for each algorithm.

In [177…
```
# Define the hyperparameters for Random Forest
rf_param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}
```

In [179…
```
# Define the hyperparameters for Gradient Boosting
gb_param_grid = {
    'n_estimators': [100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7],
    'subsample': [0.8, 1.0],
    'min_samples_split': [2, 5],
    'min_samples_leaf': [1, 2]
}
```

In [181…
```
# Define the hyperparameters for Ridge Regression (regularized Linear Regres
lr_param_grid = {
    'alpha': [0.1, 1, 10, 100]
}
```

In [183…
```
def tune_model(model, param_grid, X_train, y_train, X_test, y_test, n_iter=1
    search = RandomizedSearchCV(model, param_distributions=param_grid, n_ite
                                random_state=42, n_jobs=-1)

    with tqdm(total=n_iter, desc=f"Training {model.__class__.__name__}") as
        search.fit(X_train, y_train)
        pbar.update(n_iter)

    best_model = search.best_estimator_
    y_pred = best_model.predict(X_test)

    mse = mean_squared_error(y_test, y_pred)
    mae = mean_absolute_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)
```

```python
    return best_model, mse, mae, r2, y_pred
```

In [191…

```python
# --- Check and Handle NaN values in y_train ---
# This section is added to fix the "Input y contains NaN" error
print("--- NaN Handling ---")
initial_y_nans = y_train.isna().sum()
# Handle case where y_train might be a DataFrame (multi-output)
if isinstance(y_train, pd.DataFrame):
    initial_y_nans = initial_y_nans.sum()
print(f"Number of NaNs initially in y_train: {initial_y_nans}")

if initial_y_nans > 0:
    print(f"Original shape X_train: {X_train.shape}, y_train: {y_train.shape
    # Create a boolean mask for rows where y_train IS NOT NaN
    # If y_train is a DataFrame, .all(axis=1) keeps rows where *all* outputs
    # Adjust to .any(axis=1) if needed, depending on your multi-output strat
    if isinstance(y_train, pd.DataFrame):
         mask = y_train.notna().all(axis=1)
    else: # Assumes y_train is a Pandas Series
         mask = y_train.notna()

    # Apply mask to remove rows with NaN in y_train from BOTH X_train and y_
    X_train = X_train[mask]
    y_train = y_train[mask]

    # Confirm removal
    final_y_nans = y_train.isna().sum()
    if isinstance(y_train, pd.DataFrame): final_y_nans = final_y_nans.sum()
    print(f"New shape after removing NaN y_train rows: X_train: {X_train.sha
    print(f"Number of NaNs remaining in y_train: {final_y_nans}")
else:
    print("No NaNs found in y_train.")
print("--- End NaN Handling ---")

# --- Check for NaNs in X_train (Good Practice) ---
# Note: Handling X_train NaNs (e.g., imputation) should ideally happen earli
initial_x_nans = X_train.isna().sum().sum()
print(f"Total NaNs found in X_train: {initial_x_nans}")
if initial_x_nans > 0:
    print("Warning: NaNs detected in X_train. Ensure they are handled approp
print("-" * 20)

# --- Initialise models ---
# (Your original initialization)
rf_model = RandomForestRegressor(random_state=42)
gb_model = GradientBoostingRegressor(random_state=42)
lr_model = Ridge()

# --- Train and collect metrics + predictions ---
# Added n_iter parameter (adjust value as needed for RandomizedSearchCV)
tuning_iterations = 10 # Example: test 10 random parameter combinations per

# Ensure your tune_model function accepts and uses the n_iter argument
rf_best_model, rf_mse, rf_mae, rf_r2, rf_pred = tune_model(rf_model, rf_para
gb_best_model, gb_mse, gb_mae, gb_r2, gb_pred = tune_model(gb_model, gb_para
```

```python
# Note: RandomizedSearch might be less common for Ridge, but using for consi
# Adjust n_iter or use GridSearchCV via tune_model if lr_param_grid is small
lr_best_model, lr_mse, lr_mae, lr_r2, lr_pred = tune_model(lr_model, lr_para

# --- Create a results table ---
# Added checks to ensure models trained successfully before adding results
# (Assumes tune_model returns None or np.inf/nan on failure - adjust if need
results = {}
if rf_best_model is not None and np.isfinite(rf_mse):
     results['Random Forest'] = {'MSE': rf_mse, 'MAE': rf_mae, 'R²': rf_r2}
if gb_best_model is not None and np.isfinite(gb_mse):
     results['Gradient Boosting'] = {'MSE': gb_mse, 'MAE': gb_mae, 'R²': gb_
if lr_best_model is not None and np.isfinite(lr_mse):
    # Changed key slightly for clarity
    results['Linear Regression (Ridge)'] = {'MSE': lr_mse, 'MAE': lr_mae, 'R

# Only create and print DataFrame if results dictionary is not empty
if results:
    results_df = pd.DataFrame(results).T
    print("\n--- Model Performance ---")
    print(results_df.round(4))
else:
    print("\nNo models trained successfully or yielded valid metrics.")
```

```
--- NaN Handling ---
Number of NaNs initially in y_train: 2
Original shape X_train: (711966, 15), y_train: (711966,)
New shape after removing NaN y_train rows: X_train: (711964, 15), y_train:
(711964,)
Number of NaNs remaining in y_train: 0
--- End NaN Handling ---
Total NaNs found in X_train: 0
--------------------
```

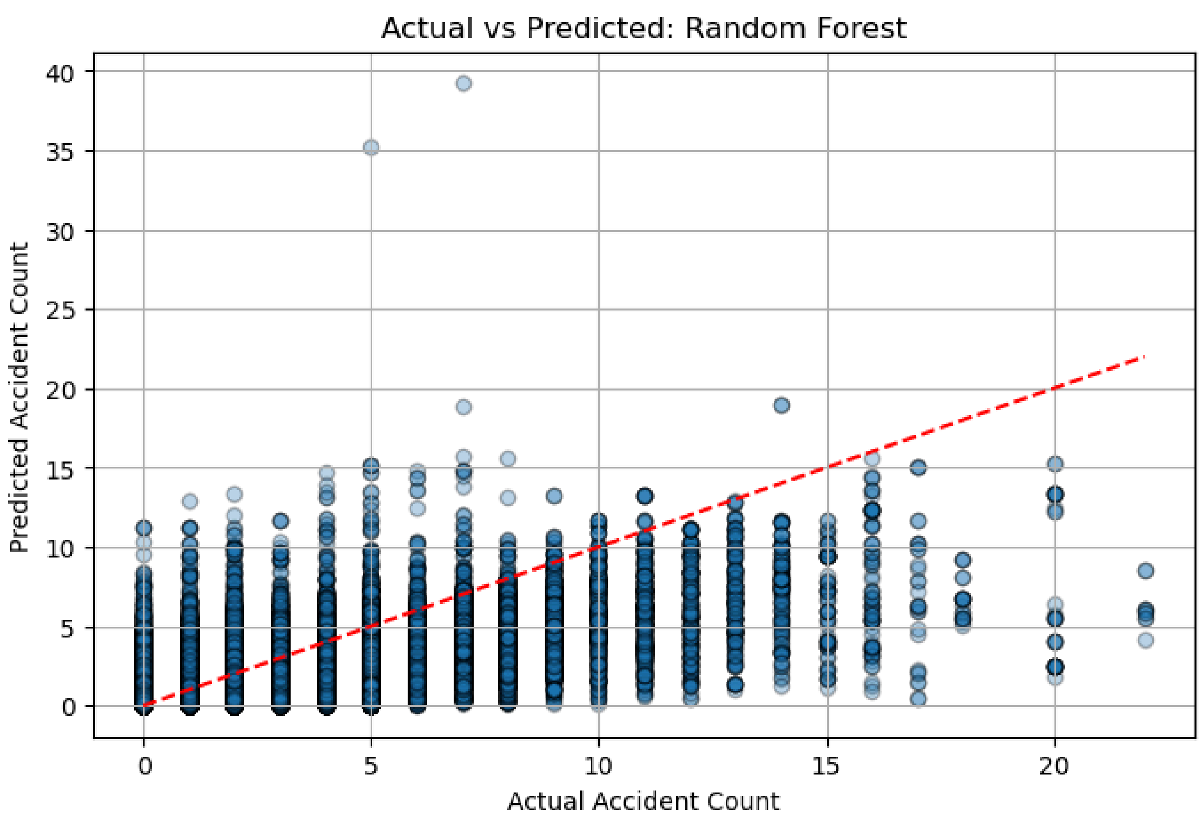
```
Training RandomForestRegressor: 100%|██████████| 10/10 [16:12<00:00, 97.29s/
it]
Training GradientBoostingRegressor: 100%|██████████| 10/10 [08:46<00:00, 52.
64s/it]
Training Ridge:   0%|          | 0/10 [00:00<?, ?it/s]C:\Users\Joanna\anacon
da3\Lib\site-packages\sklearn\model_selection\_search.py:320: UserWarning: T
he total space of parameters 4 is smaller than n_iter=10. Running 4 iteratio
ns. For exhaustive searches, use GridSearchCV.
  warnings.warn(
Training Ridge: 100%|██████████| 10/10 [00:01<00:00,  9.50it/s]
```

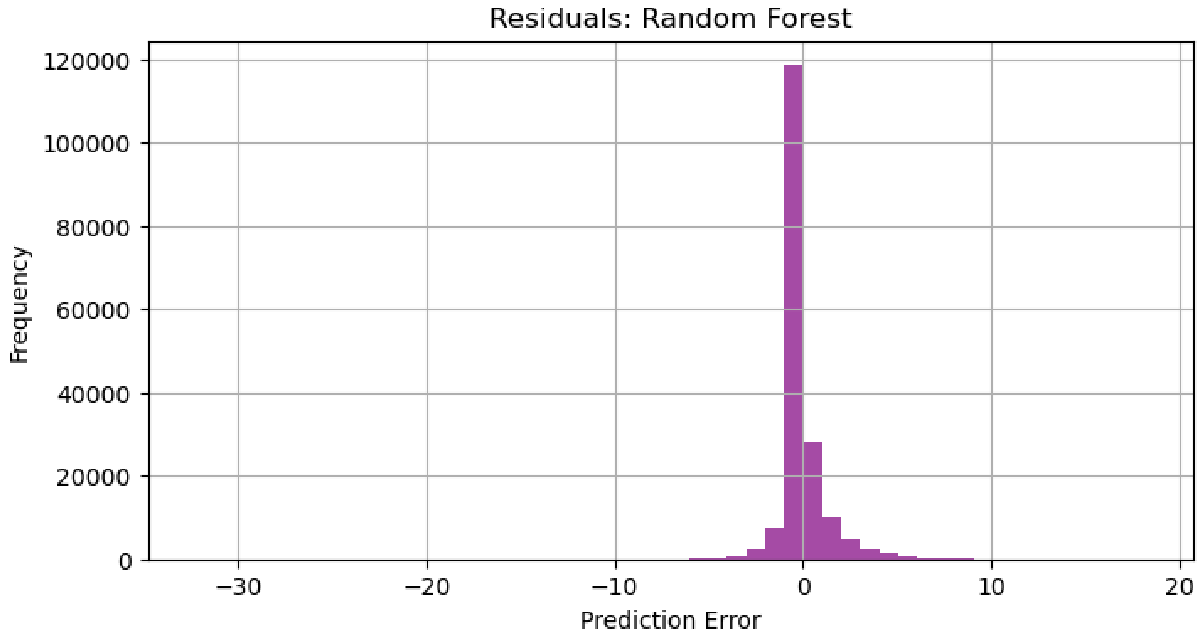
```
--- Model Performance ---
                              MSE     MAE      R²
Random Forest              1.7232  0.6696  0.5569
Gradient Boosting          2.0410  0.7428  0.4752
Linear Regression (Ridge)  2.3621  0.8207  0.3926
```

In [193…

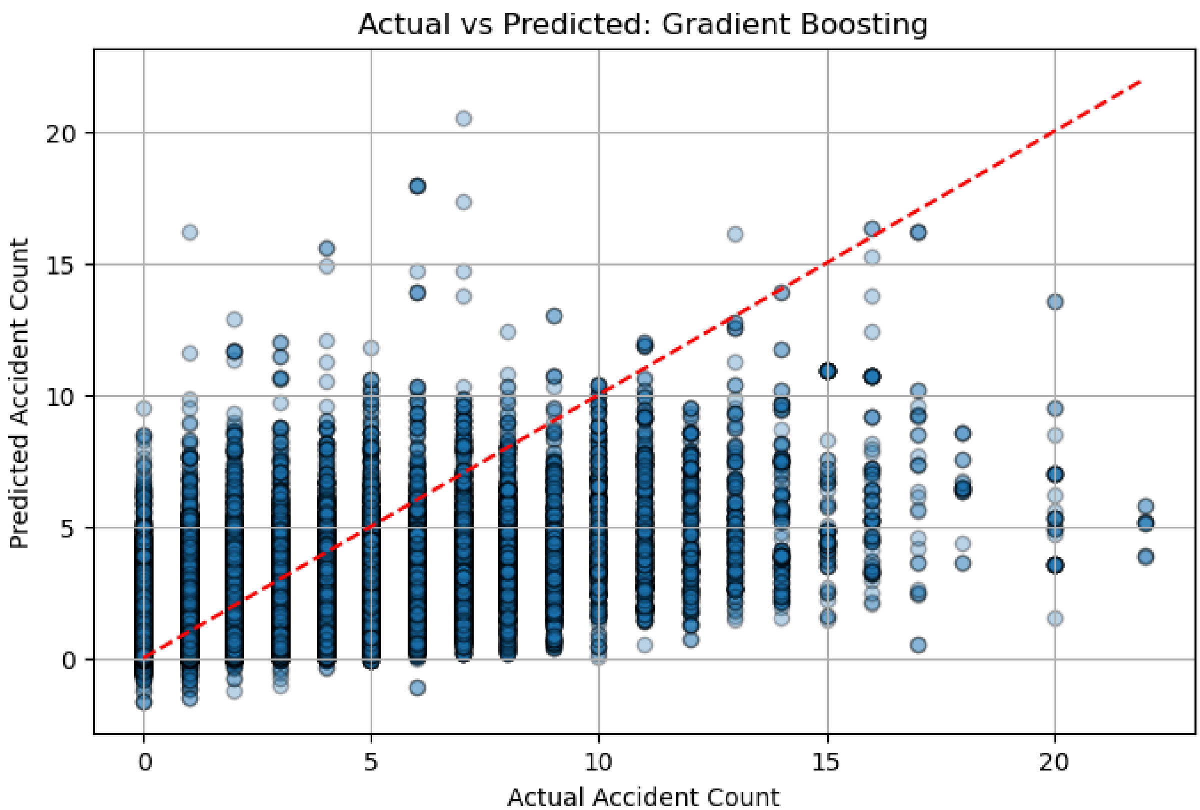
```python
# Plot Actual vs Predicted
def plot_actual_vs_pred(y_true, y_pred, title):
    plt.figure(figsize=(8, 5))
    plt.scatter(y_true, y_pred, alpha=0.3, edgecolor='k')
    plt.xlabel("Actual Accident Count")
    plt.ylabel("Predicted Accident Count")
    plt.title(f"Actual vs Predicted: {title}")
```

```
    plt.plot([y_true.min(), y_true.max()], [y_true.min(), y_true.max()], 'r-
    plt.grid(True)
    plt.show()

# Residuals
def plot_residuals(y_true, y_pred, title):
    residuals = y_true - y_pred
    plt.figure(figsize=(8, 4))
    plt.hist(residuals, bins=50, alpha=0.7, color='purple')
    plt.title(f"Residuals: {title}")
    plt.xlabel("Prediction Error")
    plt.ylabel("Frequency")
    plt.grid(True)
    plt.show()

# Run for each model
plot_actual_vs_pred(y_test, rf_pred, "Random Forest")
plot_residuals(y_test, rf_pred, "Random Forest")

plot_actual_vs_pred(y_test, gb_pred, "Gradient Boosting")
plot_residuals(y_test, gb_pred, "Gradient Boosting")

plot_actual_vs_pred(y_test, lr_pred, "Linear Regression")
plot_residuals(y_test, lr_pred, "Linear Regression")
```

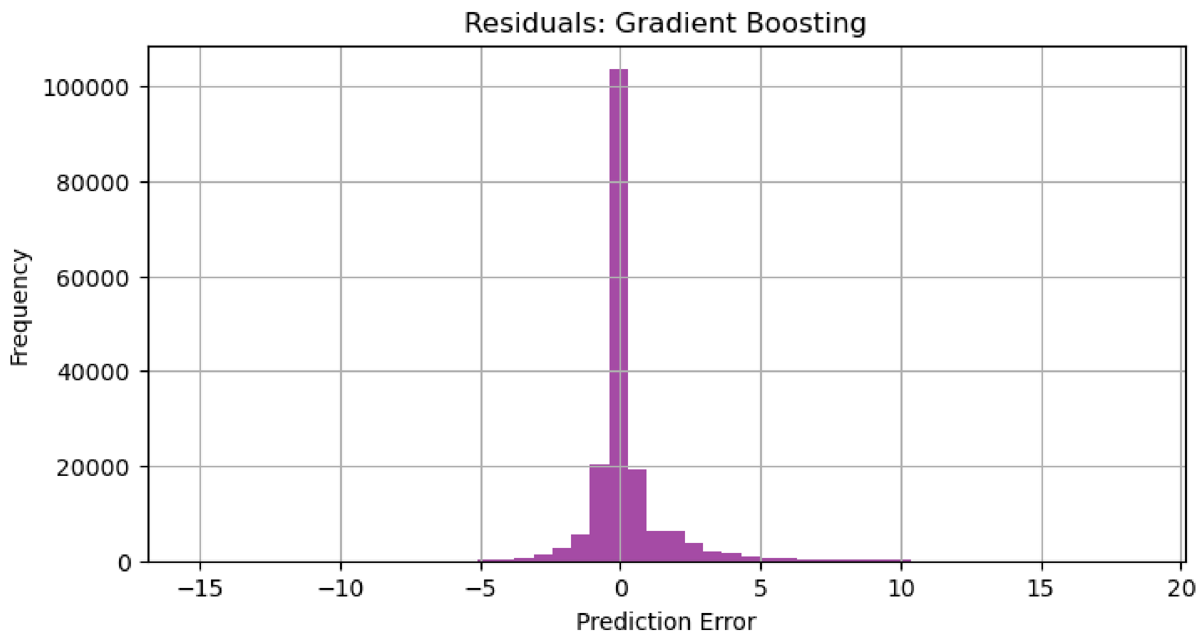




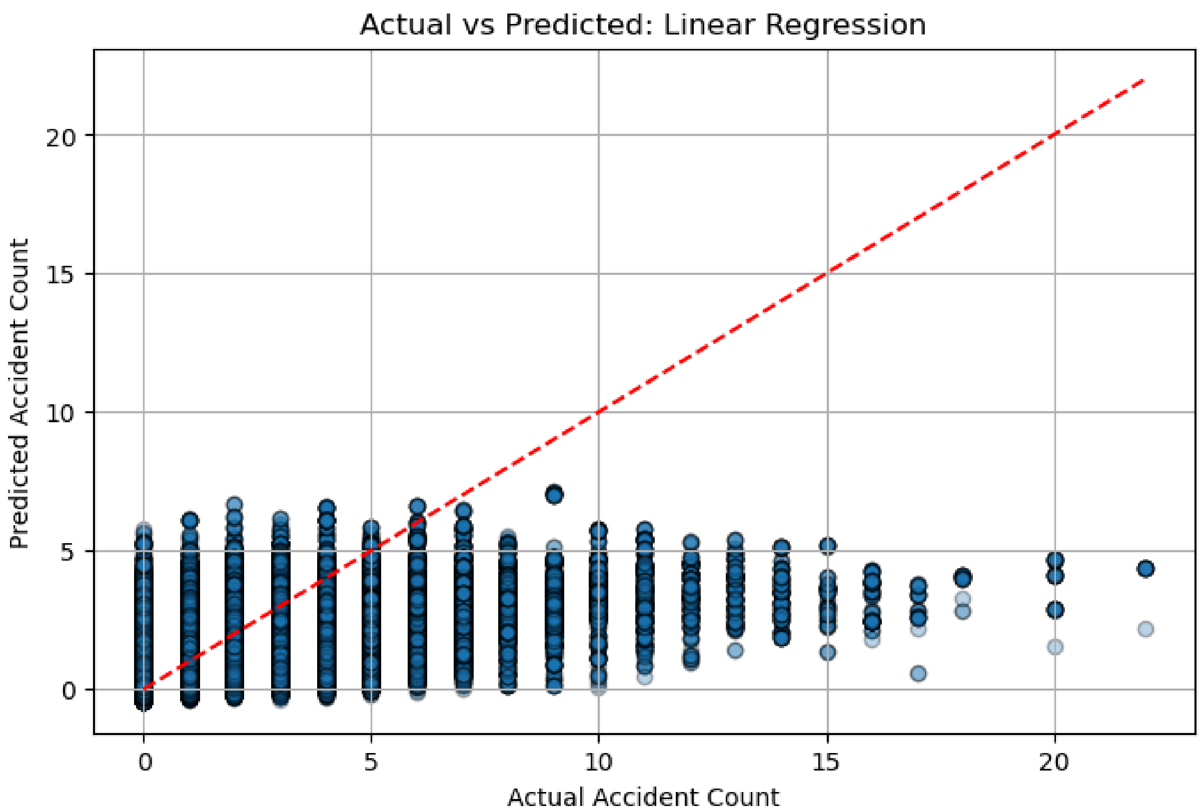
Residuals: Random Forest
Frequency
Prediction Error



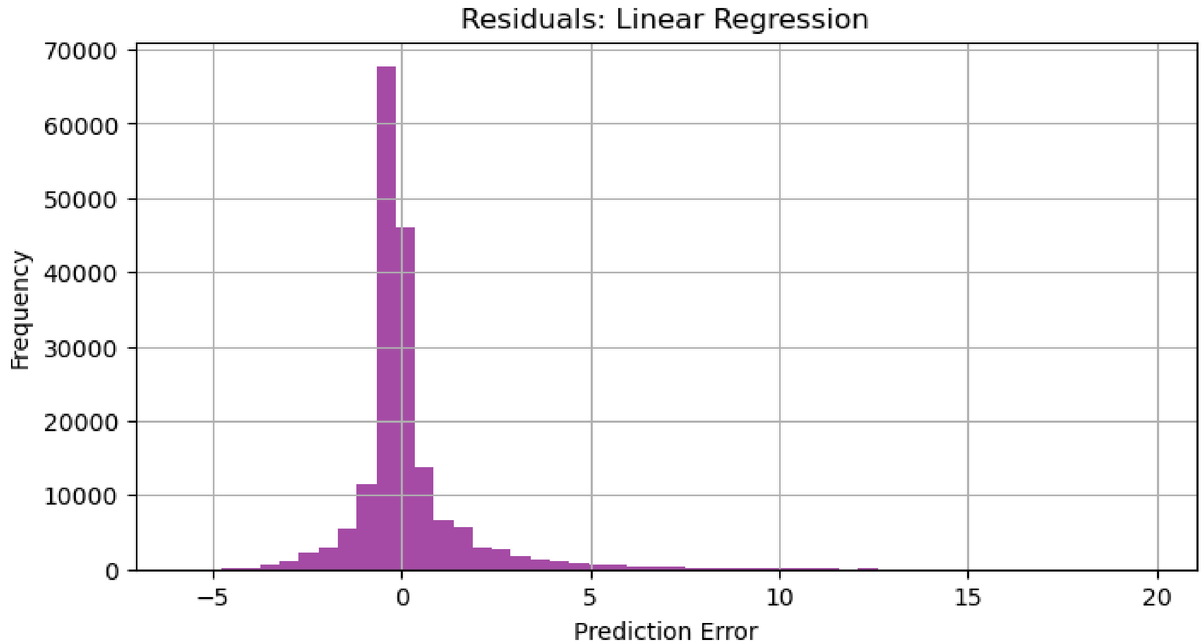
Actual vs Predicted: Gradient Boosting
Predicted Accident Count
Actual Accident Count



Residuals: Gradient Boosting
Frequency
Prediction Error
100000
80000
60000
40000
20000
0
−15
−10
−5
0
5
10
15
20



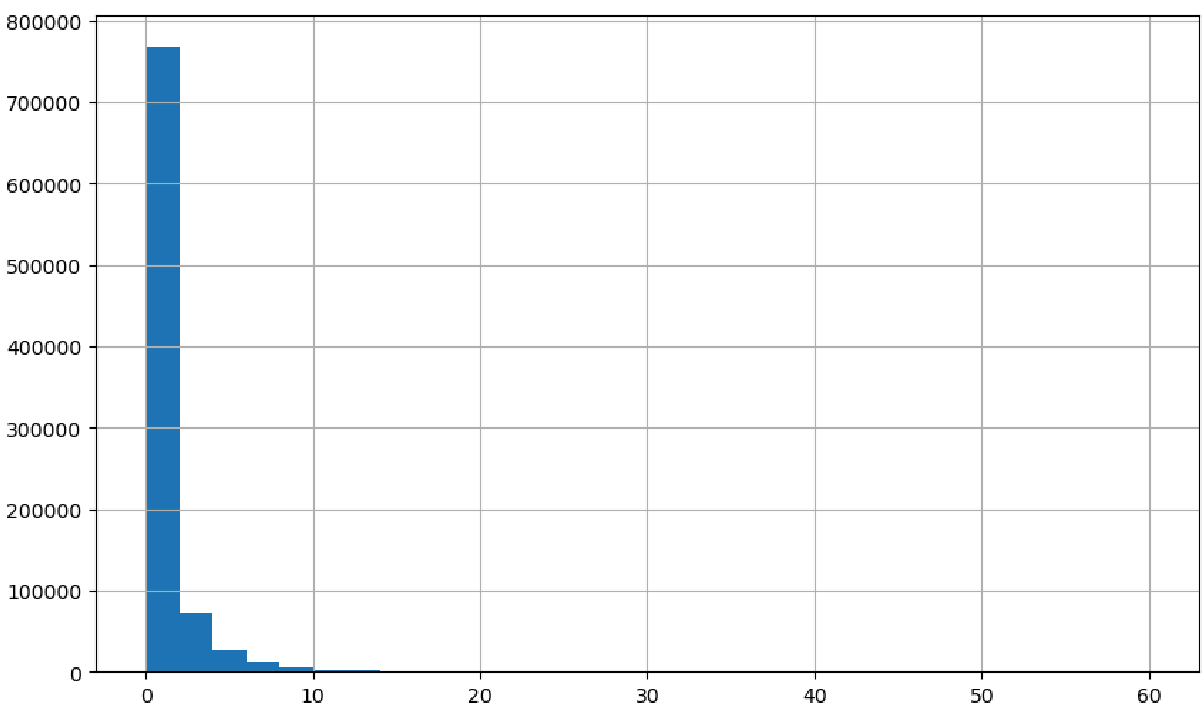
Actual vs Predicted: Linear Regression
Predicted Accident Count
Actual Accident Count
20
15
10
5
0
0
5
10
15
20



In [195…

```
df_algo1['Accident_Next_Hour'].hist(bins=30)
```

Out[195… <Axes: >



In [197…

```
def categorize_accidents(x):
    if x == 0:
        return 'none'
    elif x <= 2:
        return 'low'
    elif x <= 5:
        return 'medium'
    else:
        return 'high'
```

```
df_algo1['Accident_Class'] = df_algo1['Accident_Next_Hour'].apply(categorize
df_algo1.head()
```

Out[197…

	Start_Time	City	Temperature(F)	Humidity(%)	Pressure(in)	Wind_Spe
0	2016-06-21 10:34:40	Vallejo	0.669118	0.474747	0.633588	
1	2016-06-21 10:30:16	Hayward	0.669118	0.474747	0.679389	
2	2016-06-21 10:49:14	Walnut Creek	0.785294	0.303030	0.610687	
3	2016-06-21 10:41:42	Cupertino	0.682353	0.474747	0.664122	
4	2016-06-21 10:16:26	San Jose	0.672059	0.404040	0.679389	

5 rows × 23 columns

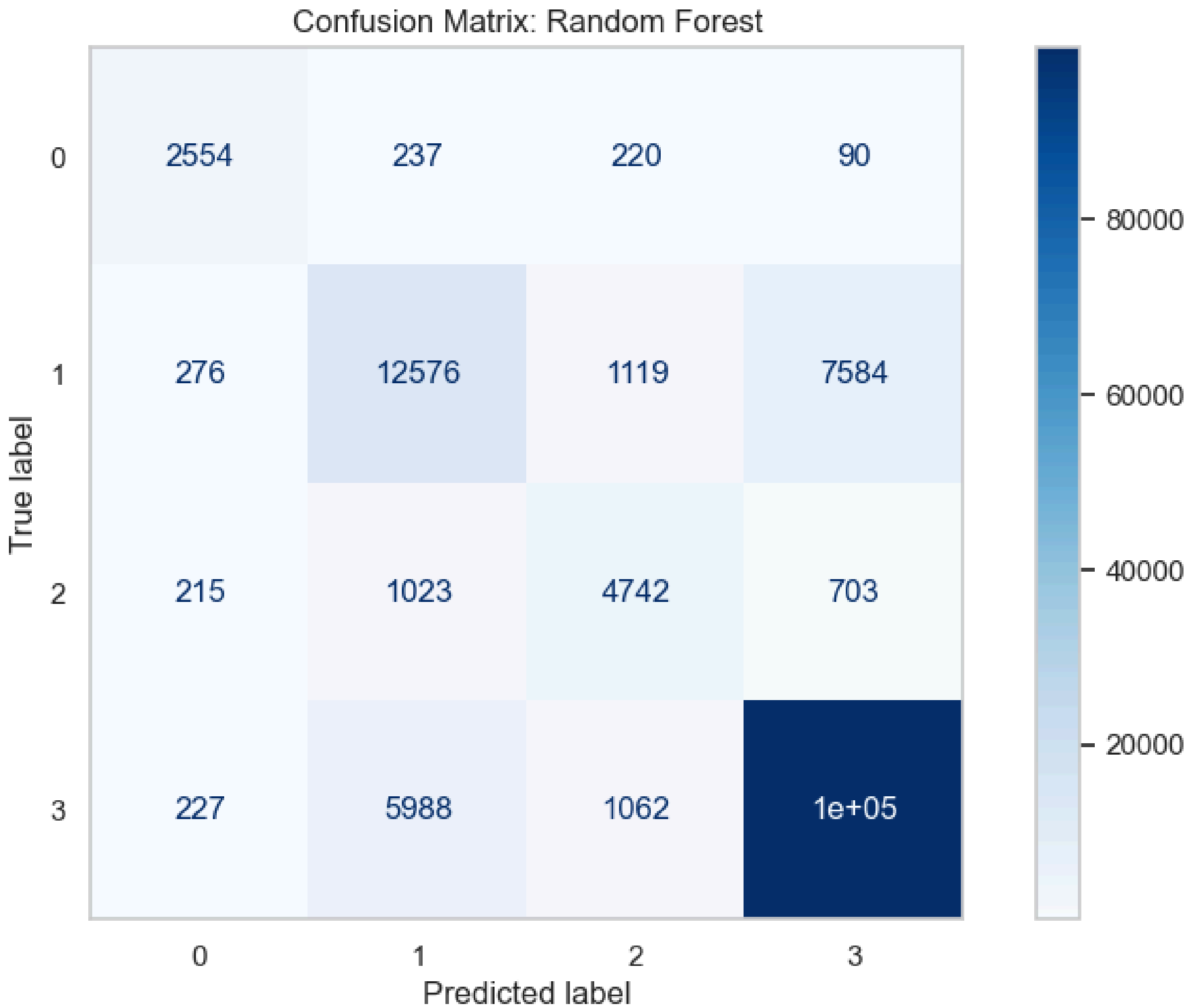
In [199…

```
# Define feature columns
feature_cols = [
    'Temperature(F)', 'Humidity(%)', 'Wind_Speed(mph)', 'Pressure(in)',
    'City_Encoded', 'Is_Windy', 'Is_Stormy', 'Is_Rainy',
    'Is_Foggy', 'Is_Snowy', 'Is_Clear',
    'Hour_Sin', 'Hour_Cos', 'Accident_Prev_Hour'
]

# Encode target classes
le = LabelEncoder()
y_encoded = le.fit_transform(df_algo1['Accident_Class'])
# Define features and target
X = df_algo1[feature_cols]
y = y_encoded

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y, test_s

# Apply SMOTE to the training data
sm = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = sm.fit_resample(X_train, y_train)

# print("Before SMOTE:", y_train.value_counts())
# print("After SMOTE:", pd.Series(y_train_resampled).value_counts())
```

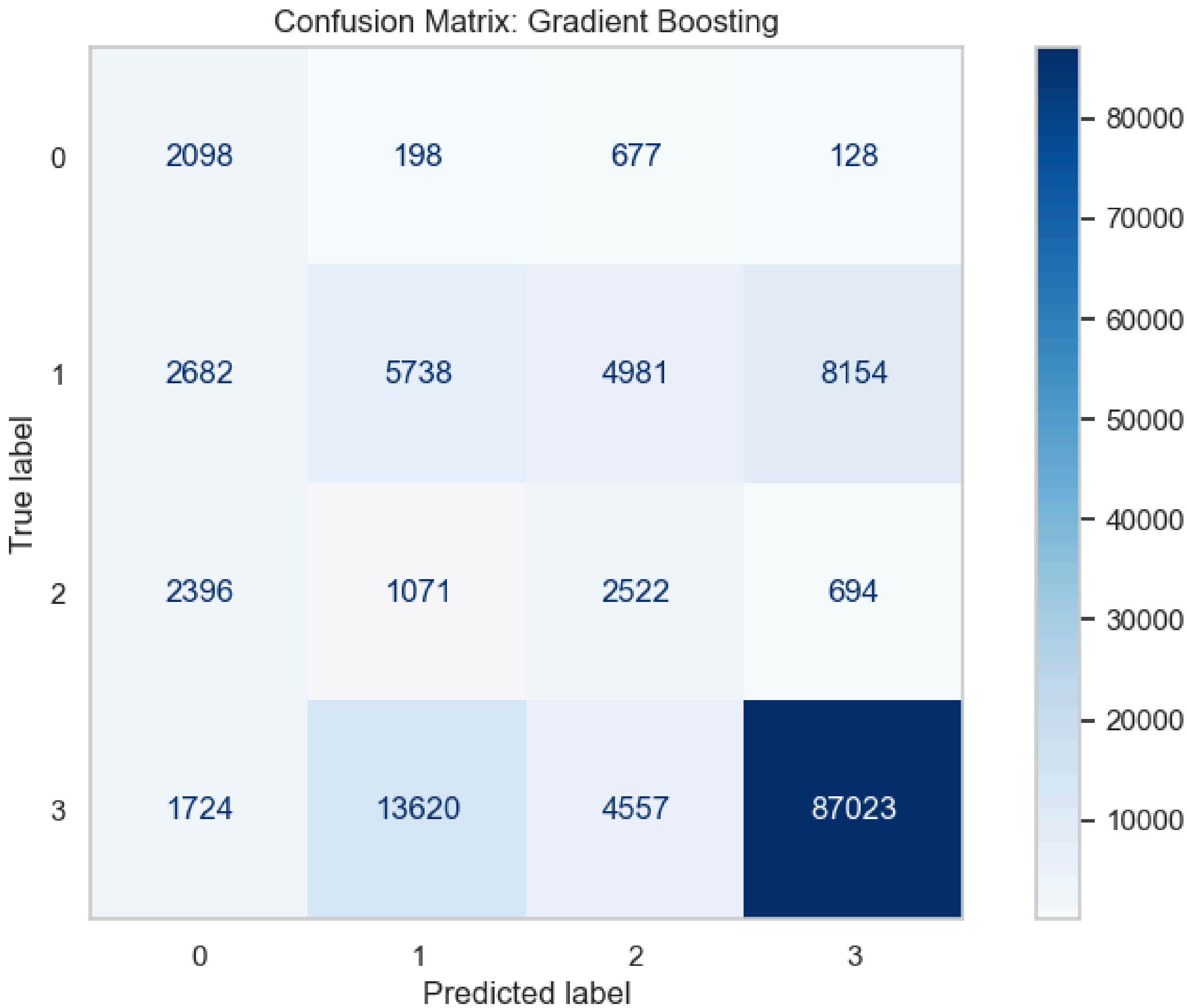
In [291…

```
# Define classifiers
models = {
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=4
    'Gradient Boosting': GradientBoostingClassifier(n_estimators=100, random
    'Logistic Regression': LogisticRegression(max_iter=1000, class_weight='b
    'XGBoost': XGBClassifier(use_label_encoder=False, eval_metric='mlogloss'
    'HistGradientBoosting': HistGradientBoostingClassifier(random_state=42)
}

results = {}

for name, model in models.items():
    print(f"\n🧪 Training {name}...")
    model.fit(X_train_resampled, y_train_resampled)
    y_pred = model.predict(X_test)

    print(classification_report(y_test, y_pred))

    # Confusion Matrix
    cm = confusion_matrix(y_test, y_pred, labels=model.classes_)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=model.
    disp.plot(cmap='Blues')
    plt.title(f"Confusion Matrix: {name}")
    plt.grid(False)
    plt.show()

    # Save overall accuracy and macro F1
    from sklearn.metrics import accuracy_score, f1_score
    results[name] = {
        "Accuracy": accuracy_score(y_test, y_pred),
        "F1 (macro)": f1_score(y_test, y_pred, average='macro')
    }
```

```
🧪 Training Random Forest...
              precision    recall  f1-score   support

           0       0.78      0.82      0.80      3101
           1       0.63      0.58      0.61     21555
           2       0.66      0.71      0.69      6683
           3       0.92      0.93      0.93    106924

    accuracy                           0.86    138263
   macro avg       0.75      0.76      0.76    138263
weighted avg       0.86      0.86      0.86    138263
```

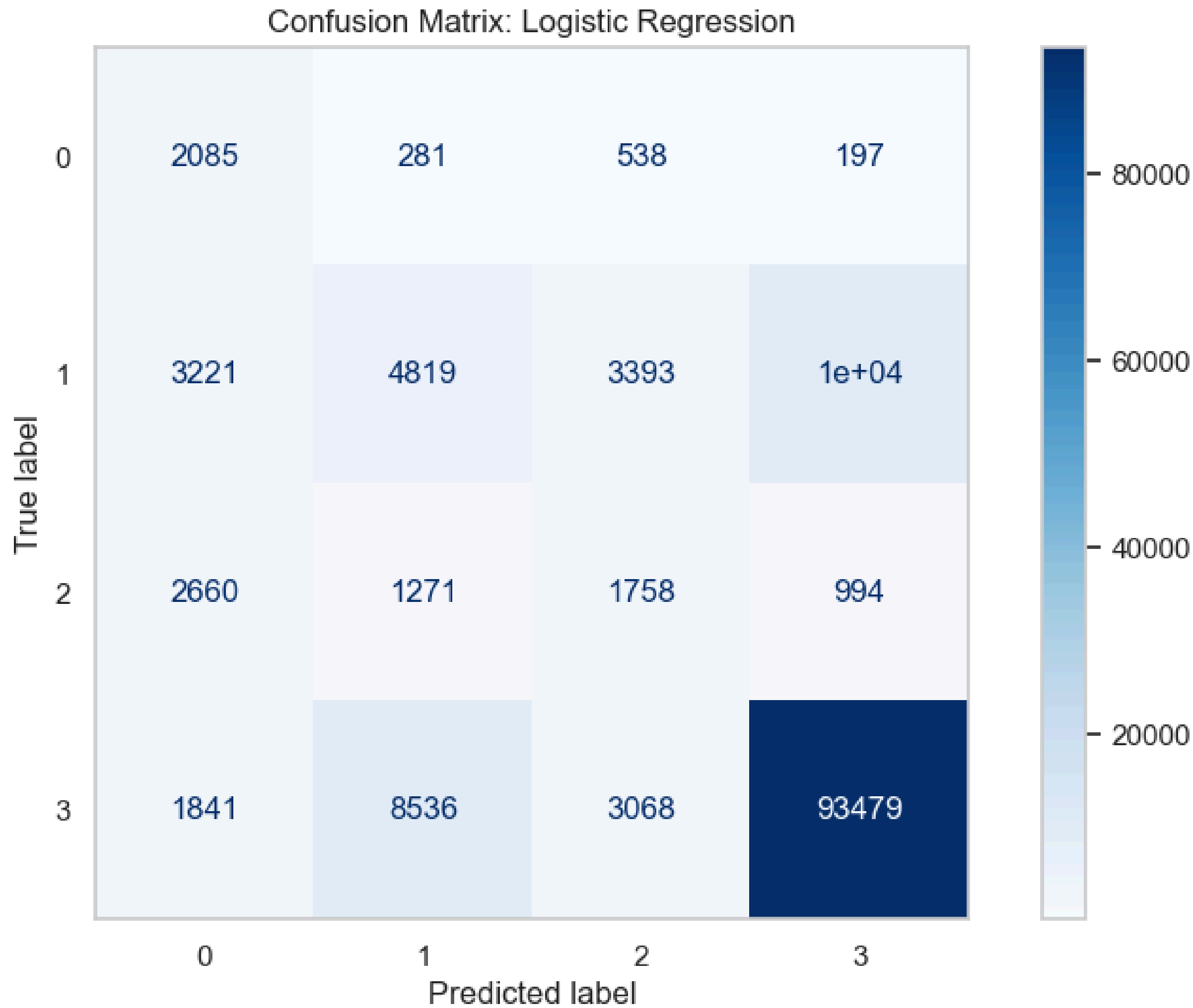



```
🧪 Training Gradient Boosting...
              precision    recall  f1-score   support

           0       0.24      0.68      0.35      3101
           1       0.28      0.27      0.27     21555
           2       0.20      0.38      0.26      6683
           3       0.91      0.81      0.86    106924

    accuracy                           0.70    138263
   macro avg       0.40      0.53      0.43    138263
weighted avg       0.76      0.70      0.73    138263
```



🧪 Training Logistic Regression...

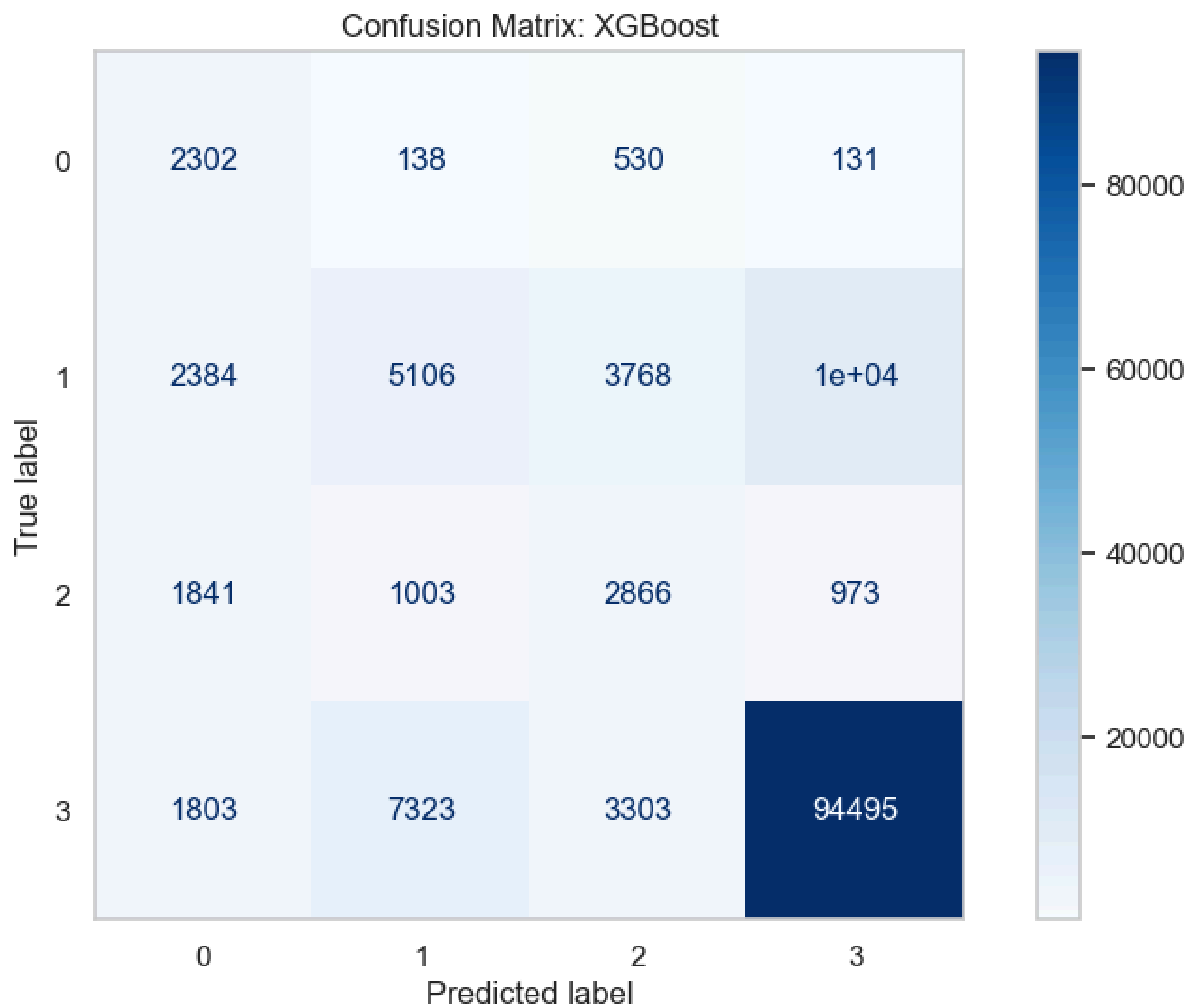
	precision	recall	f1-score	support
0	0.21	0.67	0.32	3101
1	0.32	0.22	0.26	21555
2	0.20	0.26	0.23	6683
3	0.89	0.87	0.88	106924
accuracy			0.74	138263
macro avg	0.41	0.51	0.42	138263
weighted avg	0.75	0.74	0.74	138263



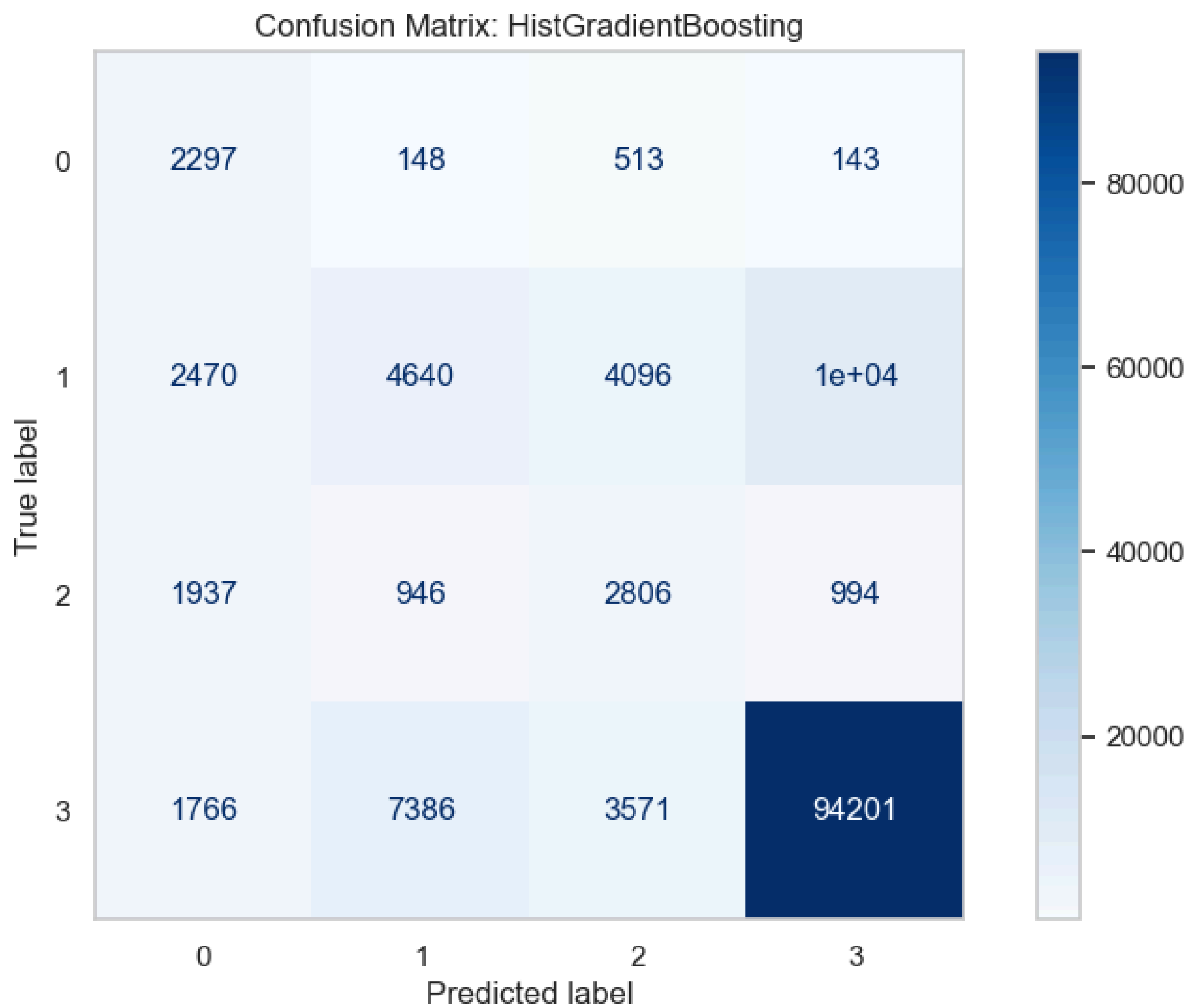
🧪 Training XGBoost...

```
C:\Users\ASUS\anaconda3\Lib\site-packages\xgboost\training.py:183: UserWarni
ng: [12:52:38] WARNING: C:\actions-runner\_work\xgboost\xgboost\src\learner.
cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
```

```
              precision    recall  f1-score   support

           0       0.28      0.74      0.40      3101
           1       0.38      0.24      0.29     21555
           2       0.27      0.43      0.33      6683
           3       0.89      0.88      0.89    106924

    accuracy                           0.76    138263
   macro avg       0.45      0.57      0.48    138263
weighted avg       0.77      0.76      0.76    138263
```




```
🧪 Training HistGradientBoosting...
              precision    recall  f1-score   support

           0       0.27      0.74      0.40      3101
           1       0.35      0.22      0.27     21555
           2       0.26      0.42      0.32      6683
           3       0.89      0.88      0.89    106924

    accuracy                           0.75    138263
   macro avg       0.44      0.56      0.47    138263
weighted avg       0.76      0.75      0.75    138263
```



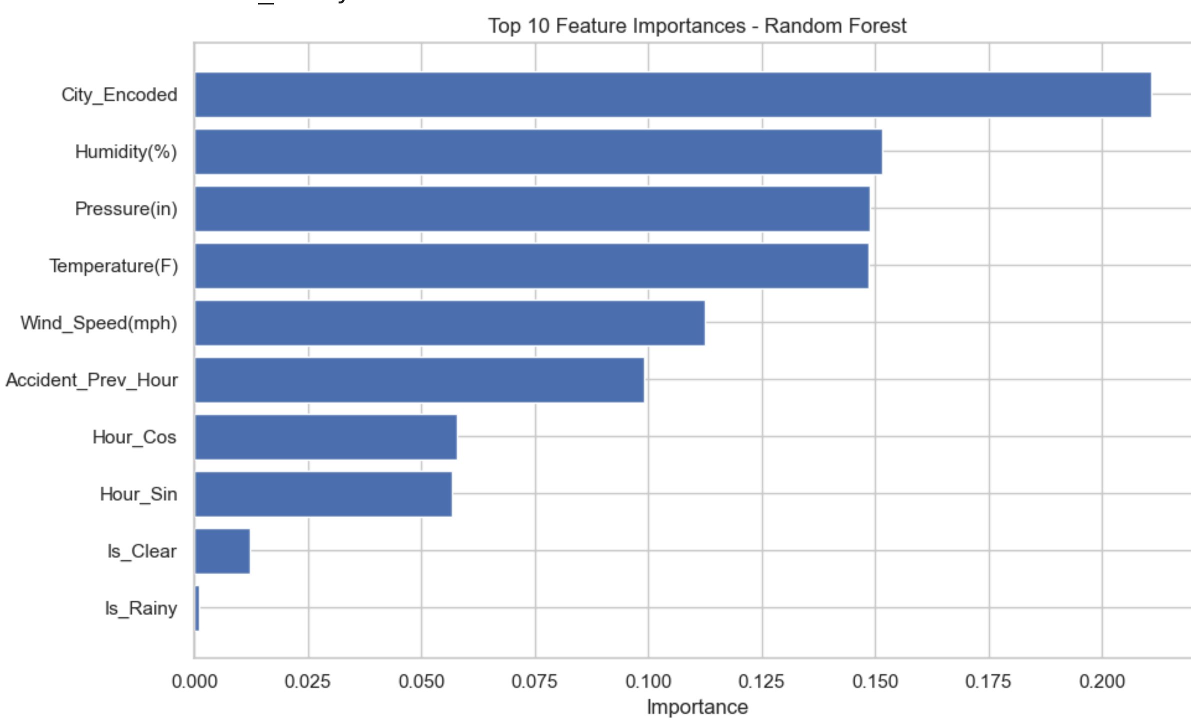
Here we can see that the shift to the classification approach works much better. The model can quite accurately discern between high rates of accident and no accidents over the next hour. Giving us much more meaningful data than the linear regression.

In [295…

```
# Make sure to use the same feature set you trained on
feature_names = X.columns  # assuming X was the feature DataFrame used

# Access the already-trained Random Forest model
rf_model = models['Random Forest']

# Check if the model supports feature importances
if hasattr(rf_model, 'feature_importances_'):
    importances = rf_model.feature_importances_

    # Create a DataFrame of features and their importance
    importance_df = pd.DataFrame({
        'Feature': feature_names,
        'Importance': importances
    }).sort_values(by='Importance', ascending=False)

    # Display top 10
    print("🔍 Top 10 Most Important Features:\n")
```

```
    print(importance_df.head(10))

    # Plot
    plt.figure(figsize=(10, 6))
    plt.barh(importance_df['Feature'][:10][::-1], importance_df['Importance'
    plt.xlabel("Importance")
    plt.title("Top 10 Feature Importances - Random Forest")
    plt.grid(True)
    plt.tight_layout()
    plt.show()
else:
    print("❌ This model does not support feature importances.")
```

```
🔍 Top 10 Most Important Features:

              Feature  Importance
4        City_Encoded    0.211020
1         Humidity(%)    0.151643
3        Pressure(in)    0.148997
0      Temperature(F)    0.148530
2      Wind_Speed(mph)   0.112412
13  Accident_Prev_Hour   0.099151
12           Hour_Cos    0.057844
11           Hour_Sin    0.056679
10           Is_Clear    0.012098
7            Is_Rainy    0.001053
```



Data-Driven Insights

With the classification model, we understand that ‘City’ has the highest prediction importance at 0.211, followed by ‘Humidity’ (0.151) , ‘Pressure’ (0.149), ‘Temperature’ (0.148), ‘Windspeed’ (0.112), ‘Accident In Previous Hour’ (0.099) and ‘Hour’ (0.058).

This supports that Los Angeles, Sacramento, San Diego and San Jose would have a significantly higher possibility of accidents relative to other cities. The number of accidents drops sharply after these four cities, making it harder to make definitive predictions for the other cities.

Next, the series of climate-related metrics that are all interrelated, allude to adverse weather. High humidity is possibly an indicator of precipitation, but the correlation to temperature possibly links to driver comfort as well.

‘Accidents in Previous Hour’ shows high clustering of accidents, and ‘Hour’ supports our peak hour hypothesis.

Therefore, to interpret the outputs exactly as an example, we would be at our highest caution when driving in Los Angeles at 5PM on a Friday, especially when it is raining in the summer and there have been other accidents in the vicinity.

The model also warns that it shows strong predictive accuracy for low and moderate severity accidents, especially severity 1 and 2 — with recall scores of 0.83 and 0.74 respectively. However, it struggles to distinguish between severity 3 and 4 accidents due to overlapping environmental features. This is evident in the confusion matrix where 47% of severity 4 accidents are misclassified.

Therefore, we have to remember the caveat that while these predictions will help us with general accidents, freak catastrophic accidents can still very much occur in any situation, without following these trends. This keeps us alert and prevents any complacency in driving.