US Automobile Accidents 2018 to March 2023 - A Technical Report and Analysis

For your technical report document:

- You should include the following sections in your report:
  - Introduction
  - Data Cleaning/Preparation
  - Exploratory Data Analysis
  - Model Selection
  - Model Analysis
  - Conclusion and Recommendations.
- Include an appendix with the output of your code from a technical notebook (i.e., Jupyter Notebook).

Version_1

- Loaded data into Dataframe and performed some basic analysis and dropping of rows.

Version_2

- Added additional analysis on the data and corrected the layout of the report. Also changed the style of the Markdown Cells to be formatted correctly. Also added a picture at the top of the report showing the layout of the report required for the project. #### 10/01/2023

Section 1: Introduction

Section 2: Data Acquisition and Cleaning

Libraries used in this project:

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import math
import plotly.graph_objects as go
import scipy.stats as stats
```

In [ ]:
```
# From csv to pd dataframe
df = pd.read_csv(r'C:\Users\user\Desktop\School\MSAAI_500\Group_Project\US_Accidents_M

# col change
df['Start_Time'] = pd.to_datetime(df['Start_Time'])

# filtering out data before 2018
df_filtered = df[df['Start_Time'].dt.year >= 2018]

# sorting by time
df_sorted = df_filtered.sort_values(by='Start_Time')

# changin the df ref name
df = df_sorted

df.head()
```

Out[ ]:

	ID	Source	Severity	Start_Time	End_Time	Start_Lat	Start_Lng	End_Lat	End_
7576435	A-7625803	Source1	4	2018-01-01 00:07:06	2018-01-01 06:07:06	41.730490	-86.338150	41.704050	-86.625
7576436	A-7625804	Source1	2	2018-01-01 00:08:27	2018-01-01 00:38:27	39.976279	-83.116542	39.976279	-83.116
7576437	A-7625805	Source1	2	2018-01-01 00:08:27	2018-01-01 00:38:27	39.975830	-83.117170	39.975830	-83.117
7576469	A-7625837	Source1	4	2018-01-01 00:13:51	2018-01-01 06:13:51	33.823210	-84.129240	33.824210	-84.150
7576680	A-7626048	Source1	4	2018-01-01 00:14:30	2018-01-01 06:14:30	39.128510	-94.567830	39.130970	-94.567

5 rows × 46 columns

Getting all of the column names in the Data Frame and Explaining Them

In [ ]:
```
column_names_list = df.columns.values.tolist()
column_names_list
```

Out[ ]:
```
['ID',
 'Source',
 'Severity',
 'Start_Time',
 'End_Time',
 'Start_Lat',
 'Start_Lng',
 'End_Lat',
 'End_Lng',
 'Distance(mi)',
 'Description',
 'Street',
 'City',
 'County',
 'State',
 'Zipcode',
 'Country',
 'Timezone',
 'Airport_Code',
 'Weather_Timestamp',
 'Temperature(F)',
 'Wind_Chill(F)',
 'Humidity(%)',
 'Pressure(in)',
 'Visibility(mi)',
 'Wind_Direction',
 'Wind_Speed(mph)',
 'Precipitation(in)',
 'Weather_Condition',
 'Amenity',
 'Bump',
 'Crossing',
 'Give_Way',
 'Junction',
 'No_Exit',
 'Railway',
 'Roundabout',
 'Station',
 'Stop',
 'Traffic_Calming',
 'Traffic_Signal',
 'Turning_Loop',
 'Sunrise_Sunset',
 'Civil_Twilight',
 'Nautical_Twilight',
 'Astronomical_Twilight']
```

Data Columns

ID: The ID of the accident, unique key.

Source: Pulled from two diffrerent sources

Severity: On a scale of 1-4, how severe the accident was at the time of recording, four being the most severe.

Start_Time: Date the accident took place, in the format of YYYY-MM-DD HH:MM:SS (We have filtered out any dates that are older than 2018)

End_Time: Date the accident ended, in the format of YYYY-MM-DD HH:MM:SS

Start_Lat: Latitude of the accident from inital impact

Start_Lng: Longitude of the accident from inital impact

End_Lat: Latitude of the accident from ending place of the vehicle

End_Lng: Longitude of the accident from ending place of the vehicle

Distance(mi): distance in miles that the accident occured from the starting point

Description: Description of the accident and the incident that occured

Street: Street name of the accident

City: City name of the accident

County: County name of the accident

State: State name of the accident (49 states are listed in this data set)

Zipcode: Zipcode that the accident Occured in

Country: Country of the Accident (All accidents that occured happened in the United States)

Timezone: Timezone of the accident

Airport_Code: The closest Airport to the accident

Weather_Timestamp: the time-stamp of weather observation

Temperature(F): Temperature in Fahrenheit

Wind_Chill(F): Wind chilld in Fahrenheit

Humidity(%): Shows the humidity (in percentage).

Pressure(in): Shows the air pressure (in inches).

Visibility(mi): Shows visibility (in miles).

Wind_Direction: Shows wind direction.

Wind_Speed(mph): Shows wind speed (in miles per hour).

Precipitation(in): Shows precipitation amount in inches, if there is any.

Weather_Condition: Shows the weather condition (rain, snow, thunderstorm, fog, etc.)

Amenity: A POI annotation which indicates presence of amenity in a nearby location.

Bump: A POI annotation which indicates presence of speed bump or hump in a nearby location.

Crossing: A POI annotation which indicates presence of crossing in a nearby location.

Give_Way: A POI annotation which indicates presence of give_way in a nearby location.

Junction: A POI annotation which indicates presence of junction in a nearby location.

No_Exit: A POI annotation which indicates presence of no_exit in a nearby location.

Railway: A POI annotation which indicates presence of railway in a nearby location.

Roundabout: A POI annotation which indicates presence of roundabout in a nearby location.

Station: A POI annotation which indicates presence of station in a nearby location.

Stop: A POI annotation which indicates presence of stop in a nearby location.

Traffic_Calming: A POI annotation which indicates presence of traffic_calming in a nearby location.

Traffic_Signal: A POI annotation which indicates presence of traffic_signal in a nearby location.

Turning_Loop: A POI annotation which indicates presence of turning_loop in a nearby location.

Sunrise_Sunset: Shows the period of day (i.e. day or night) based on sunrise/sunset.

Civil_Twilight: Shows the period of day (i.e. day or night) based on civil twilight.

Nautical_Twilight: Shows the period of day (i.e. day or night) based on nautical twilight.

Astronomical_Twilight: hows the period of day (i.e. day or night) based on astronomical twilight.

Droping Columns that are Blank or Not Needed for analysis

In [ ]:
```
# Columns to drop
columns_to_drop = ['Country', 'Timezone', 'Airport_Code', 'Description', 'Street', 'As
df = df.drop(columns=columns_to_drop)

# Verify
df.head()
```

Out[ ]:

	ID	Source	Severity	Start_Time	End_Time	Start_Lat	Start_Lng	End_Lat	End_
7576435	A-7625803	Source1	4	2018-01-01 00:07:06	2018-01-01 06:07:06	41.730490	-86.338150	41.704050	-86.625
7576436	A-7625804	Source1	2	2018-01-01 00:08:27	2018-01-01 00:38:27	39.976279	-83.116542	39.976279	-83.116
7576437	A-7625805	Source1	2	2018-01-01 00:08:27	2018-01-01 00:38:27	39.975830	-83.117170	39.975830	-83.117
7576469	A-7625837	Source1	4	2018-01-01 00:13:51	2018-01-01 06:13:51	33.823210	-84.129240	33.824210	-84.150
7576680	A-7626048	Source1	4	2018-01-01 00:14:30	2018-01-01 06:14:30	39.128510	-94.567830	39.130970	-94.567

5 rows × 38 columns

Changing all of the Values that are True or False to 1 or 0

In [ ]:

```
# Define a function to convert boolean to integer
def convert_boolean(val):
    if isinstance(val, bool):
        return int(val)
    return val

# Apply the function to the entire DataFrame
df = df.applymap(convert_boolean)

# To verify the changes
df.head()
```

Out[ ]:

	ID	Source	Severity	Start_Time	End_Time	Start_Lat	Start_Lng	End_Lat	End_
7576435	A-7625803	Source1	4	2018-01-01 00:07:06	2018-01-01 06:07:06	41.730490	-86.338150	41.704050	-86.625
7576436	A-7625804	Source1	2	2018-01-01 00:08:27	2018-01-01 00:38:27	39.976279	-83.116542	39.976279	-83.116
7576437	A-7625805	Source1	2	2018-01-01 00:08:27	2018-01-01 00:38:27	39.975830	-83.117170	39.975830	-83.117
7576469	A-7625837	Source1	4	2018-01-01 00:13:51	2018-01-01 06:13:51	33.823210	-84.129240	33.824210	-84.150
7576680	A-7626048	Source1	4	2018-01-01 00:14:30	2018-01-01 06:14:30	39.128510	-94.567830	39.130970	-94.567

5 rows × 38 columns

Dropping all rows with missing values

In [ ]:
```
# Dropping all rows with missing vals
df = df.dropna()

print(df.shape)
```

```
(3551471, 38)
```

Adding the Weekday, Month, Day, Hour, and Minute Columns to the Data Frame

I am hoping that by using time, we can see if there are any trends in the data that we can use to predict the severity of the accident.

In [ ]:
```
# Cast Start_Time to datetime
df["Start_Time"] = pd.to_datetime(df["Start_Time"])

# Extract year, month, weekday and day
df["Year"] = df["Start_Time"].dt.year
df["Month"] = df["Start_Time"].dt.month
df["Weekday"] = df["Start_Time"].dt.weekday
df["Day"] = df["Start_Time"].dt.day

# Extract hour and minute
df["Hour"] = df["Start_Time"].dt.hour
df["Minute"] = df["Start_Time"].dt.minute

df.head()
```

Out[ ]:

	ID	Source	Severity	Start_Time	End_Time	Start_Lat	Start_Lng	End_Lat	En
7576435	A-7625803	Source1	4	2018-01-01 00:07:06	2018-01-01 06:07:06	41.730490	-86.338150	41.704050	-86.6
7576681	A-7626049	Source1	2	2018-01-01 01:17:30	2018-01-01 07:17:30	41.202430	-95.948110	41.214100	-95.9
7576589	A-7625957	Source1	2	2018-01-01 01:51:00	2018-01-01 02:21:00	40.369102	-78.433999	40.369471	-78.4
7576588	A-7625956	Source1	2	2018-01-01 01:51:00	2018-01-01 02:21:00	40.369469	-78.435852	40.369102	-78.4
7576780	A-7626148	Source1	4	2018-01-01 02:02:29	2018-01-01 08:02:29	38.855510	-110.920290	38.844746	-111.0

5 rows × 44 columns

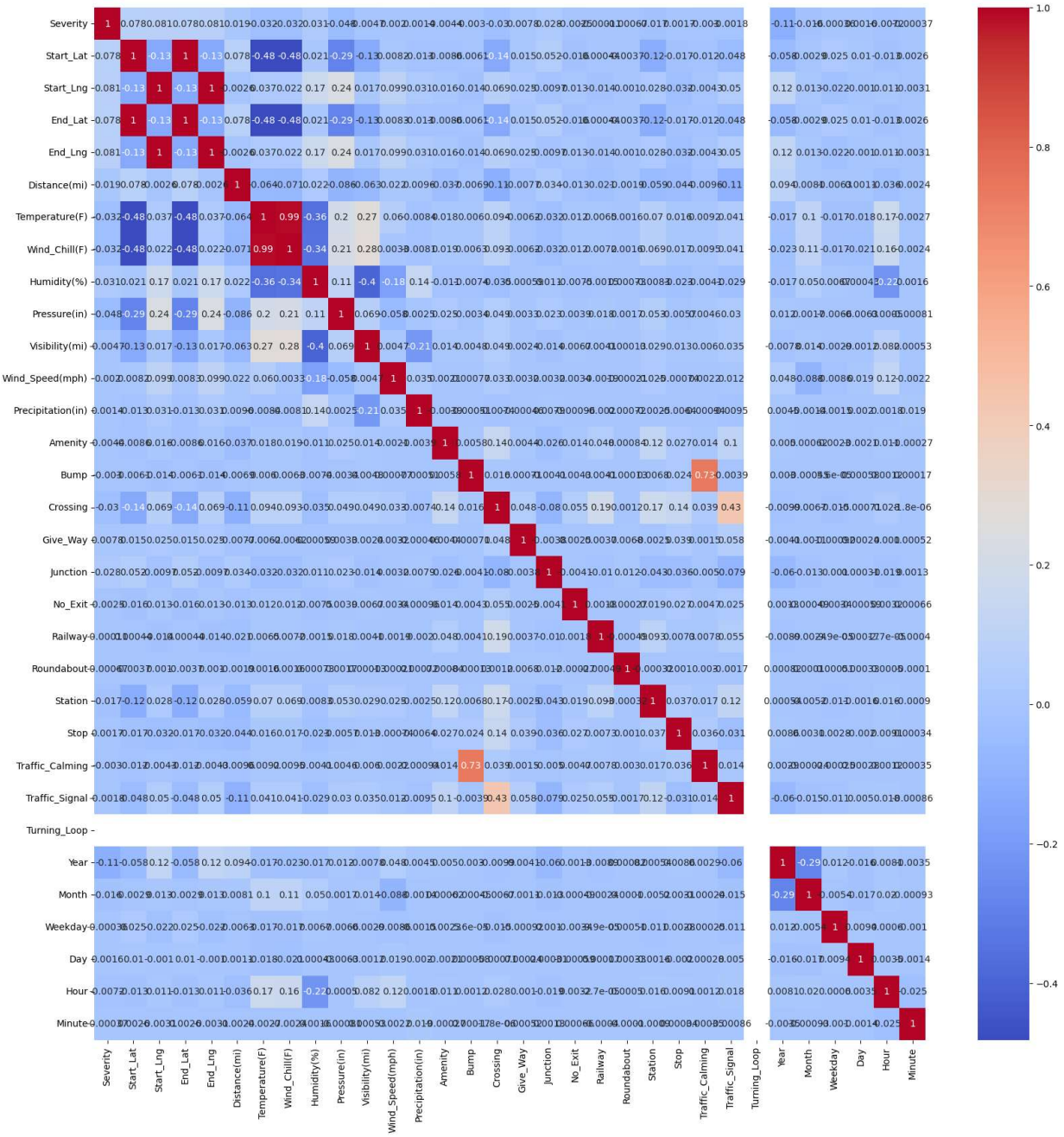
Section 3: Exploratory Data Analysis (EDA)

Correlation Matrix to see the correlation between all of the Variables

In [ ]:

```
Correlation_Matrix = df.corr()

# Correlation Heatmap
plt.figure(figsize=(20, 20))
sns.heatmap(Correlation_Matrix, annot=True, cmap='coolwarm')
plt.show()
```

```
C:\Users\user\AppData\Local\Temp\ipykernel_8144\876484934.py:1: FutureWarning: The de
fault value of numeric_only in DataFrame.corr is deprecated. In a future version, it
will default to False. Select only valid columns or specify the value of numeric_only
to silence this warning.
  Correlation_Matrix = df.corr()
```



Pivot table to get the count of accidents by State and the Severity of Accident counts

In [ ]:
```
pivot_table = pd.pivot_table(df, index='State', columns='Severity', values='ID', aggfu
pivot_table['Total'] = pivot_table.sum(axis=1)

print(pivot_table)
```

Severity	1	2	3	4	Total
State					
AL	114	25159	410	1152	26835
AR	11	16460	208	2022	18701
AZ	6195	64321	526	2233	73275
CA	4608	864589	5539	6638	881374
CO	513	25338	6047	4359	36257
CT	5	37902	792	2624	41323
DC	37	12751	251	520	13559
DE	7	5316	233	1048	6604
FL	2831	516850	4257	6246	530184
GA	374	46002	1906	5464	53746
IA	3	10449	230	1183	11865
ID	0	8363	39	356	8758
IL	237	29522	9212	1935	40906
IN	37	19575	1034	2846	23492
KS	5	10741	396	402	11544
KY	31	3970	251	386	4638
LA	1147	65466	966	988	68567
MA	166	3215	1474	300	5155
MD	228	59273	1794	5274	66569
ME	0	808	13	33	854
MI	28	49115	1464	3915	54522
MN	30	114230	1346	527	116133
MO	62	29300	1439	1091	31892
MS	4	6397	95	486	6982
MT	0	25866	28	335	26229
NC	1741	120919	1970	7620	132250
ND	0	2693	1	10	2704
NE	26	2906	139	321	3392
NH	1	1443	70	75	1589
NJ	66	67256	1779	3341	72442
NM	49	2136	30	268	2483
NV	1	7369	67	342	7779
NY	694	156364	4574	5945	167577
OH	508	29064	593	1474	31639
OK	54	10287	835	79	11255
OR	1261	118800	1479	4043	125583
PA	177	147905	2388	10837	161307
RI	67	2362	273	35	2737
SC	58	134466	419	1702	136645
SD	0	165	1	28	194
TN	1436	70802	549	1894	74681
TX	156	169055	6308	3451	178970
UT	374	44368	307	583	45632
VA	1697	160430	2011	12416	176554
VT	0	168	6	29	203
WA	495	26925	1412	2115	30947
WI	32	9444	676	1329	11481
WV	1	10708	68	313	11090
WY	0	2077	18	278	2373

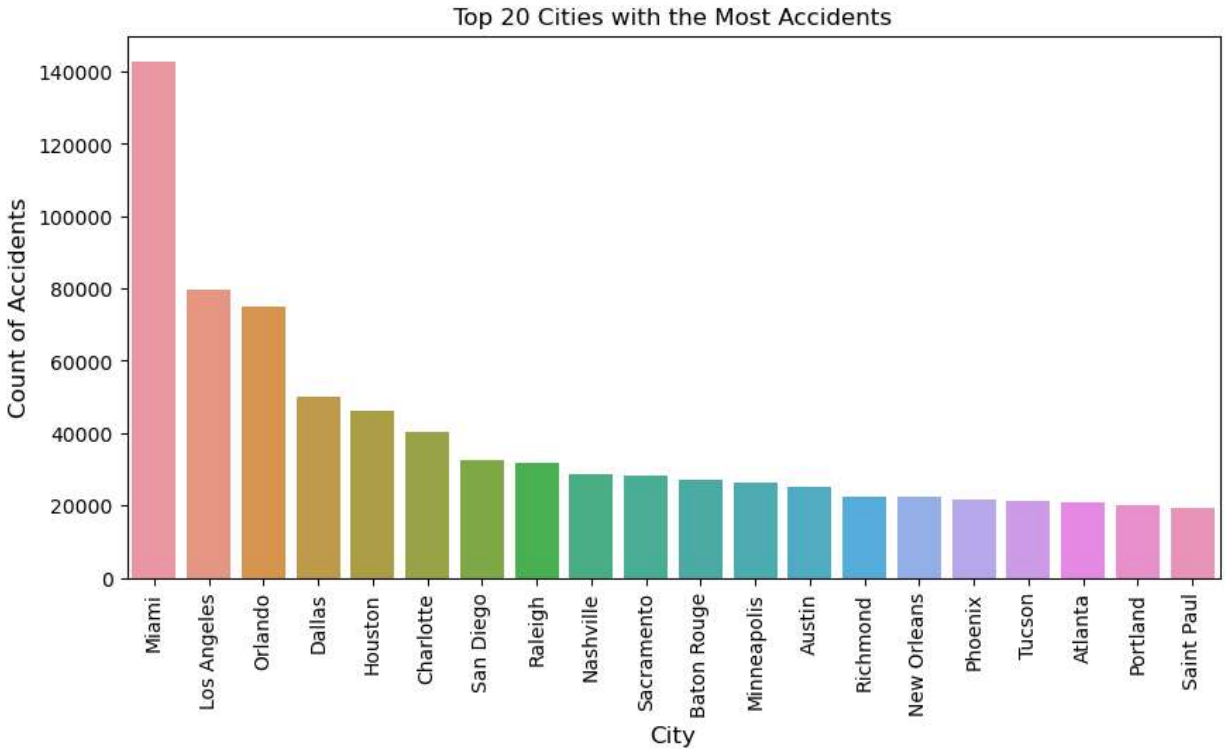
The Pivot Table shows that there are 49 states, The only State Missing from this data set is The State of Hawaii

Interactive Map of Accidents by The State

In [ ]:
```
state_counts = df["State"].value_counts()
fig = go.Figure(data=go.Choropleth(locations=state_counts.index, z=state_counts.values
fig.update_layout(title_text="Number of US Accidents for each State", geo_scope="usa")
fig.show()
```

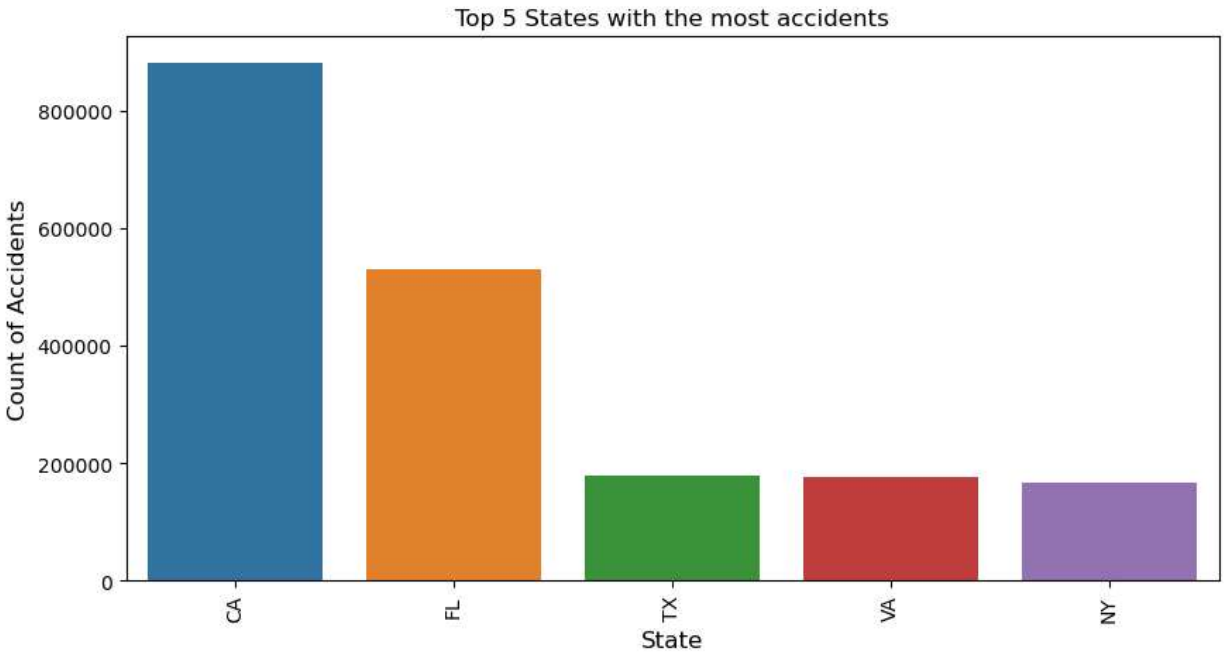
Getting the top 20 Cities with the most accidents

In [ ]:
```
# Get the top 20 cities with the most accidents
city_accident_counts = df['City'].value_counts().head(20)

plt.figure(figsize=(10, 5))
sns.barplot(x=city_accident_counts.index, y=city_accident_counts.values, alpha=1)
plt.title('Top 20 Cities with the Most Accidents')
plt.ylabel('Count of Accidents', fontsize=12)
plt.xlabel('City', fontsize=12)
plt.xticks(rotation=90)
plt.show()
```



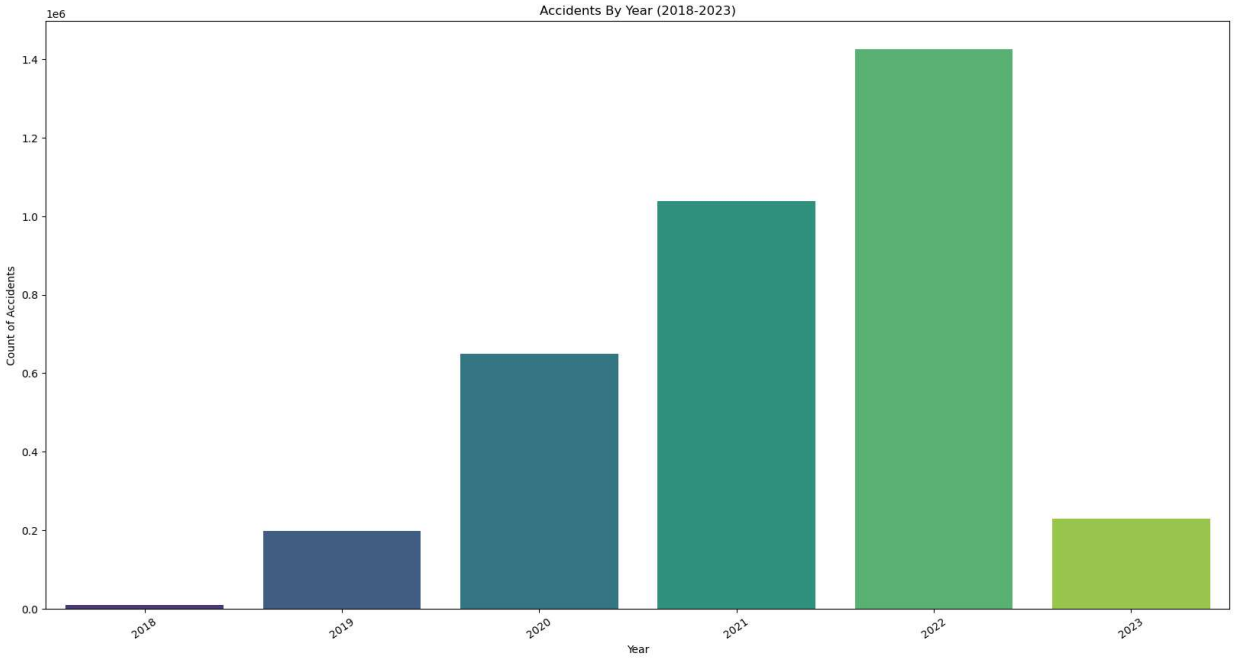
Getting the Top 5 States with the most Accidents

In [ ]:
```
# getting the top 5 States with the most accidents

top_5_states = state_counts[:5]

plt.figure(figsize=(10,5))
sns.barplot(x=top_5_states.index, y=top_5_states.values, alpha=1)
plt.title('Top 5 States with the most accidents')
plt.ylabel('Count of Accidents', fontsize=12)
plt.xlabel('State', fontsize=12)
plt.xticks(rotation=90)
plt.show()
```



Plotting the Amount of Accidents by Year

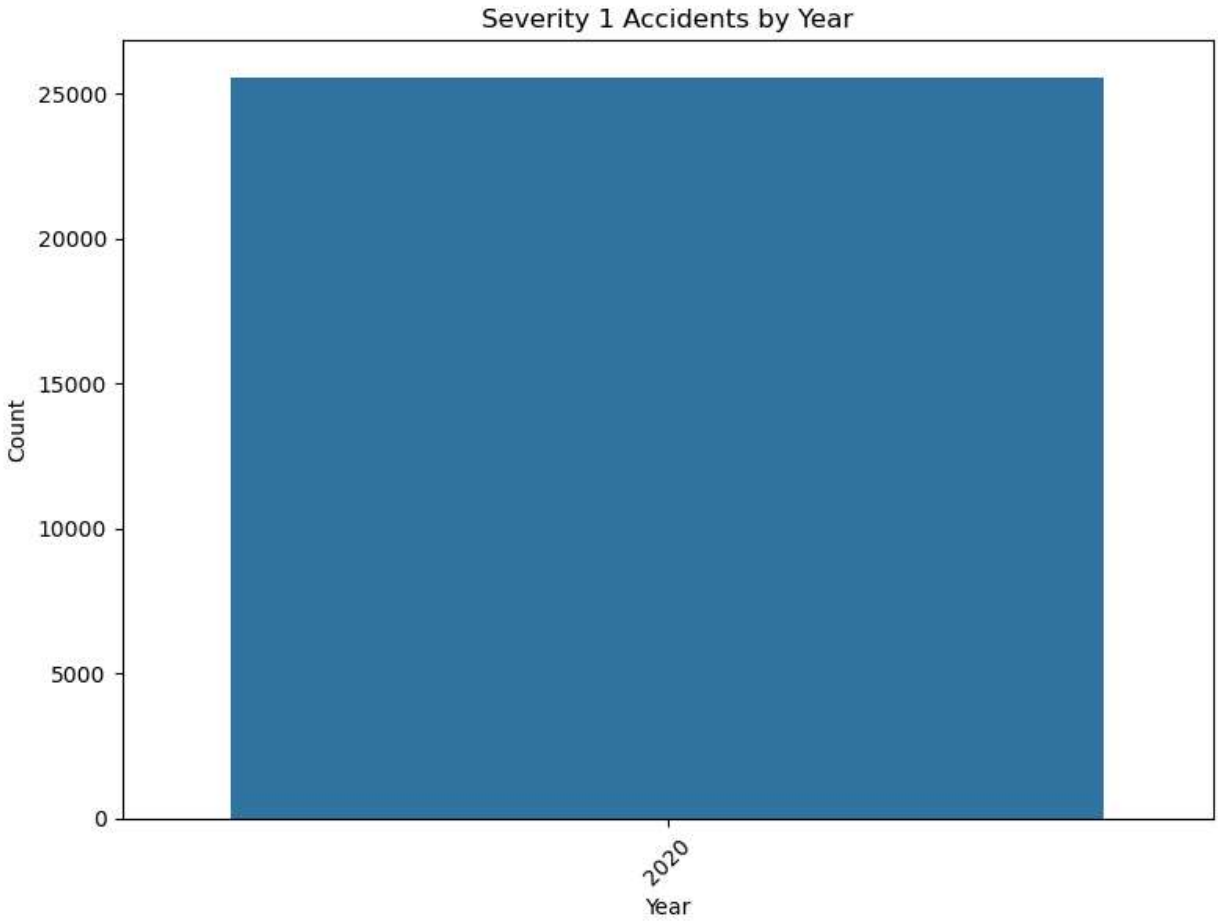
In [ ]:
```
# Creating a histogram of the accidents by year using Seaborn
df['Year'] = df['Start_Time'].dt.year
plt.figure(figsize=(20, 10))
sns.countplot(data=df, x='Year', palette='viridis')
plt.xlabel('Year')
plt.ylabel('Count of Accidents')
plt.title('Accidents By Year (2018-2023)')
plt.xticks(rotation=35)
plt.show()
```

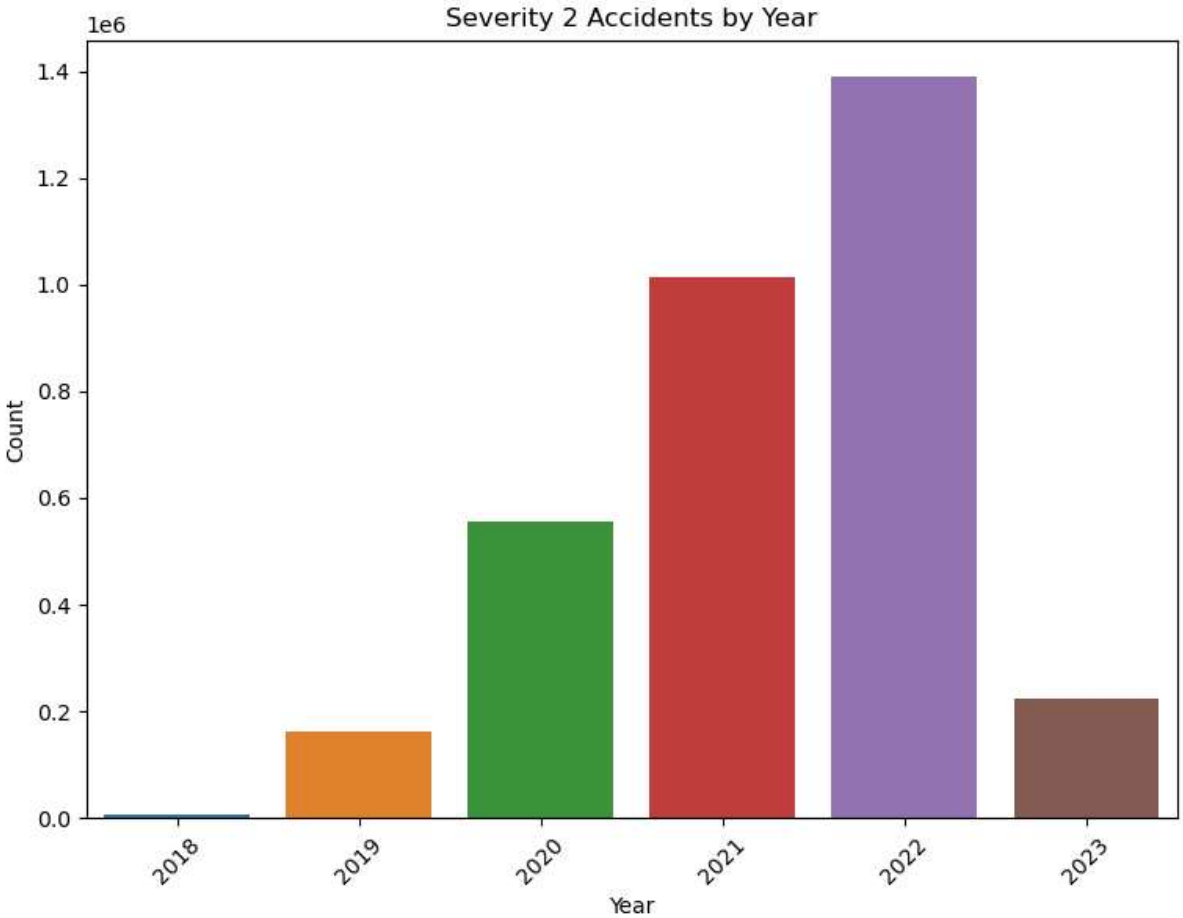


We only have data going up until March of 2023, We can either drop the data or keep it, because we are looking for variables that are correlated with the severity of the accident and date, doesn't matter as much as time of day.

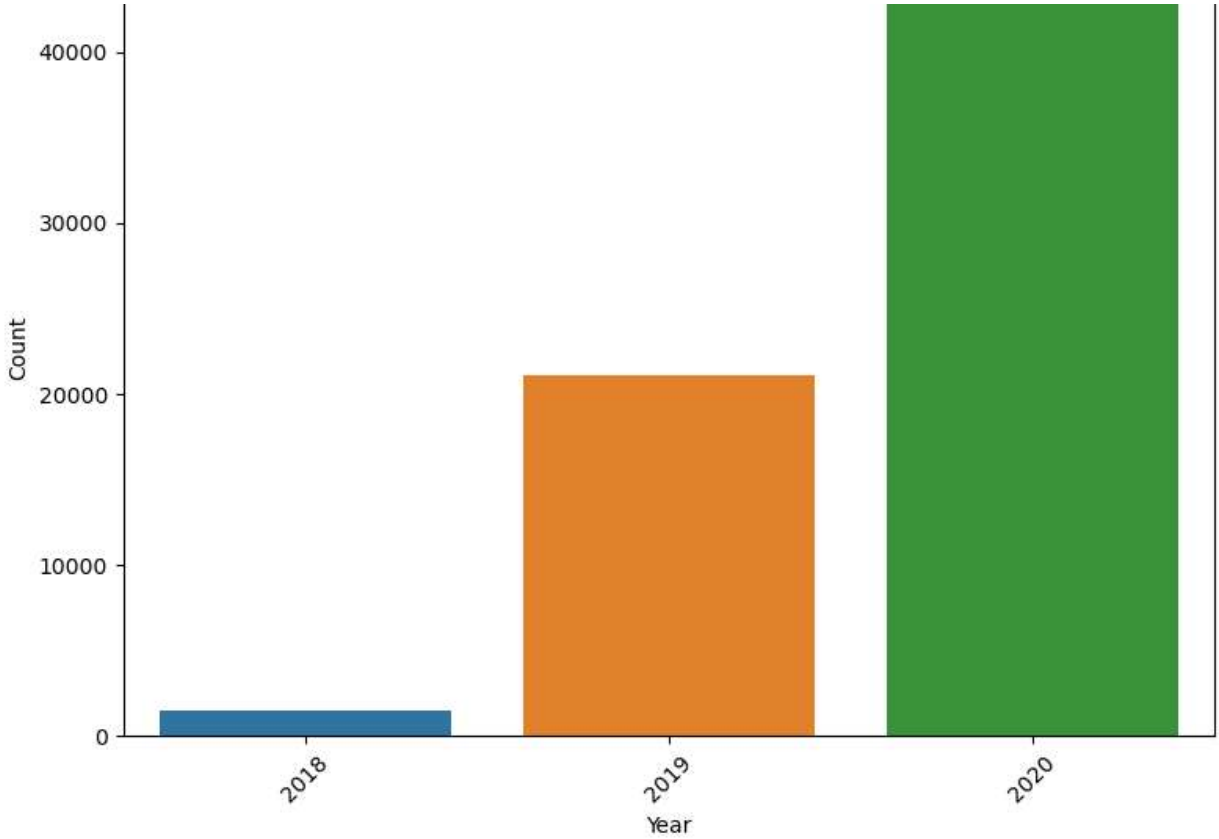
Severity Count of Accidents by Year

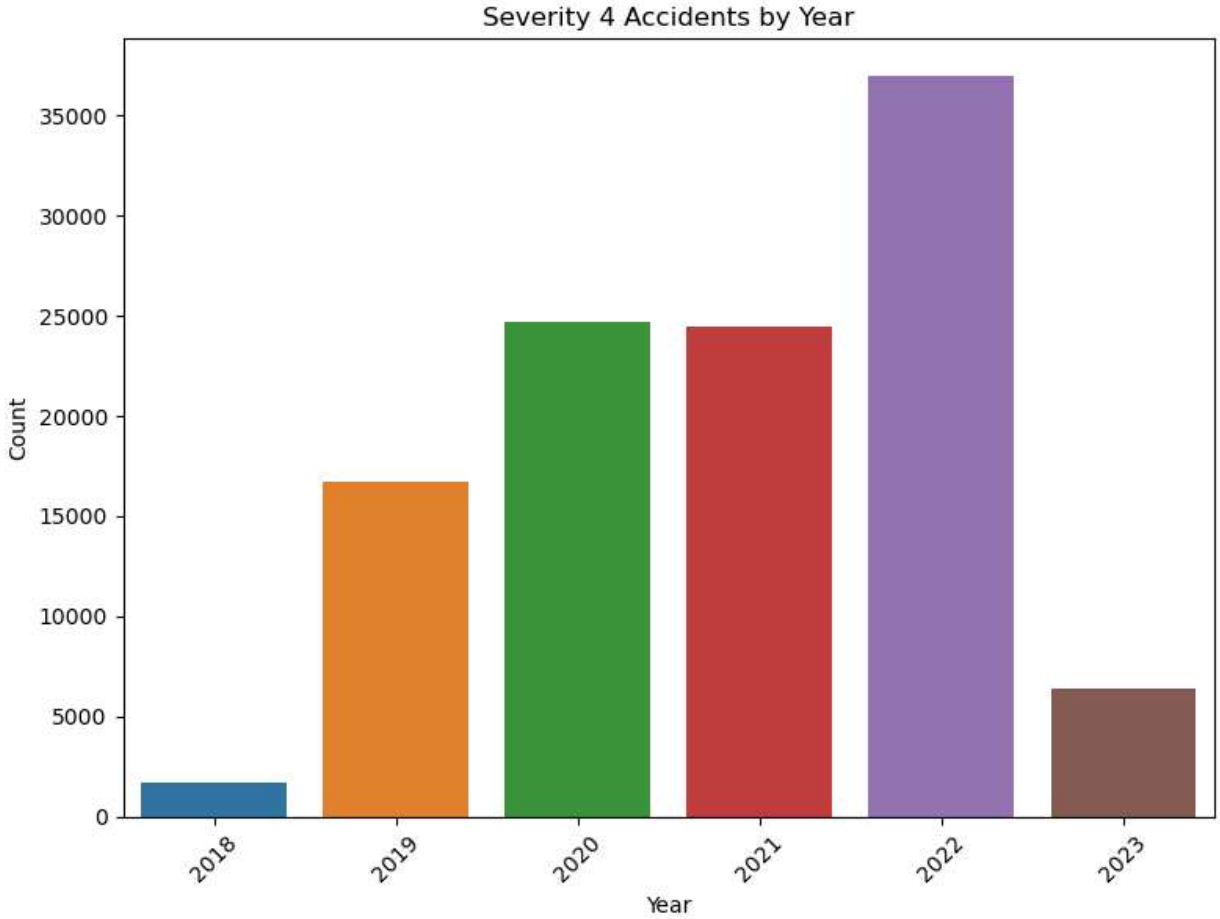
In [ ]:
```
# Extracting the year from the datetime column
df['Year'] = df['Start_Time'].dt.year

# Defining the severity
severity_levels = [1, 2, 3, 4]

fig, axes = plt.subplots(len(severity_levels), figsize=(8, 6 * len(severity_levels)))


for i, severity in enumerate(severity_levels):
    subset = df[df['Severity'] == severity]

    count_by_year = subset.groupby('Year').size()

    sns.barplot(x=count_by_year.index, y=count_by_year.values, ax=axes[i])
    axes[i].set_xlabel('Year')
    axes[i].set_ylabel('Count')
    axes[i].set_title(f'Severity {severity} Accidents by Year')
    axes[i].set_xticklabels(axes[i].get_xticklabels(), rotation=45)

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


Severity 1 Accidents by Year
25000
20000
15000
10000
5000
0
Count
2020
Year


Severity 2 Accidents by Year
1e6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Count
2018
2019
2020
2021
2022
2023
Year

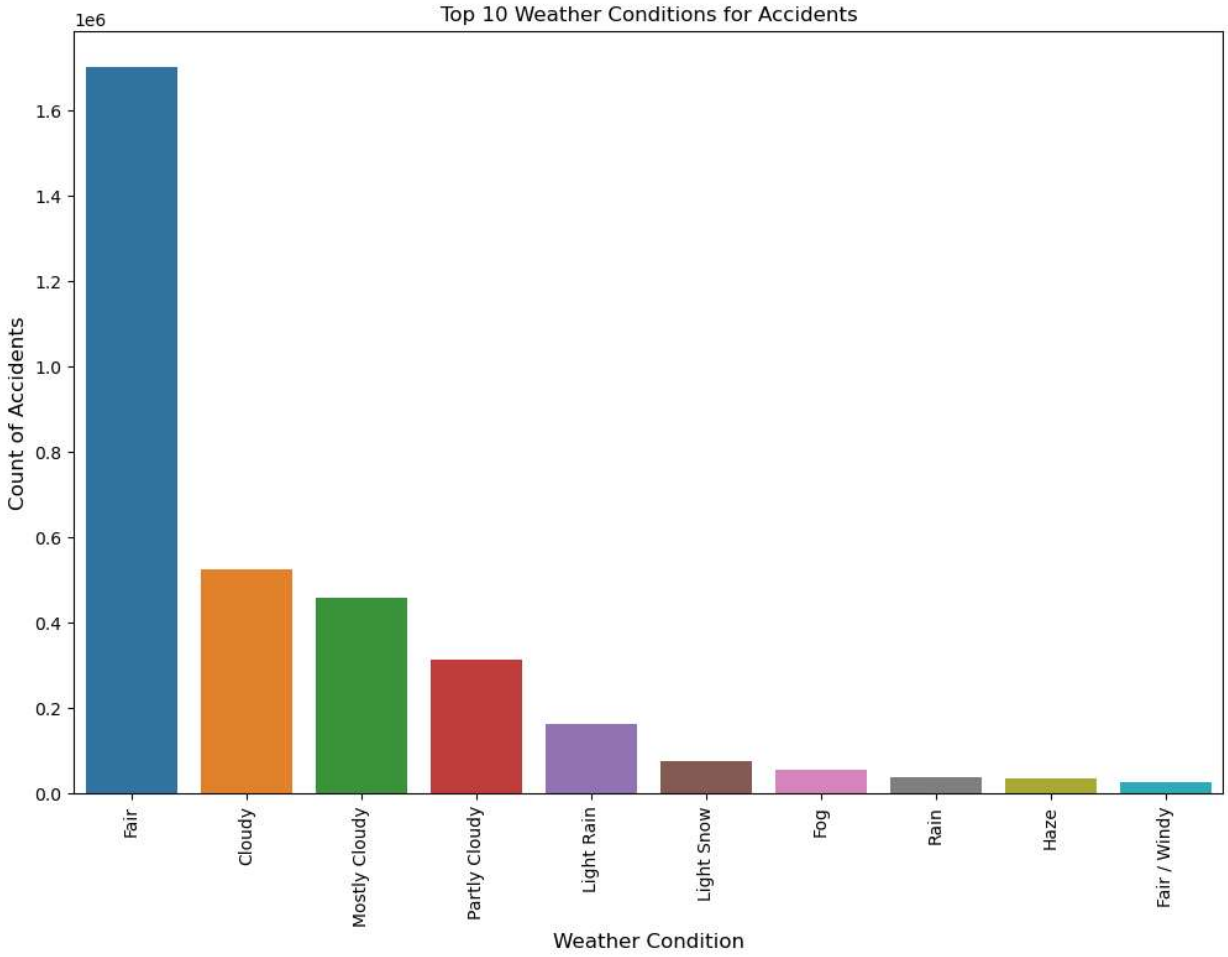

Severity 3 Accidents by Year





Getting the most common Weather Conditions at The Time of the Accident

In [ ]:
```
weather_conditions = df['Weather_Condition'].value_counts()[:10]
plt.figure(figsize=(12,8))
sns.barplot(x=weather_conditions.index, y=weather_conditions.values, alpha=1)
plt.title('Top 10 Weather Conditions for Accidents')
plt.ylabel('Count of Accidents', fontsize=12)
plt.xlabel('Weather Condition', fontsize=12)
plt.xticks(rotation=90)
plt.show()
```



Modeling the data

Getting Mean, Median, Standard Deviation, Variance of the Severity of Accidents

In [ ]:
```
# getting the stats of the severity column
mean = df['Severity'].mean()
median = df['Severity'].median()
std_dev = df['Severity'].std()
variance = df['Severity'].var()

print(f'The mean of the severity column is {mean}')
print(f'The median of the severity column is {median}')
print(f'The standard deviation of the severity column is {std_dev}')
print(f'The variance of the severity column is {variance}')
```

```
The mean of the severity column is 2.073811105313826
The median of the severity column is 2.0
The standard deviation of the severity column is 0.3810629489479216
The variance of the severity column is 0.1452089710608863
```

Getting the Stats Quickly

In [ ]:
```
df.describe()
```

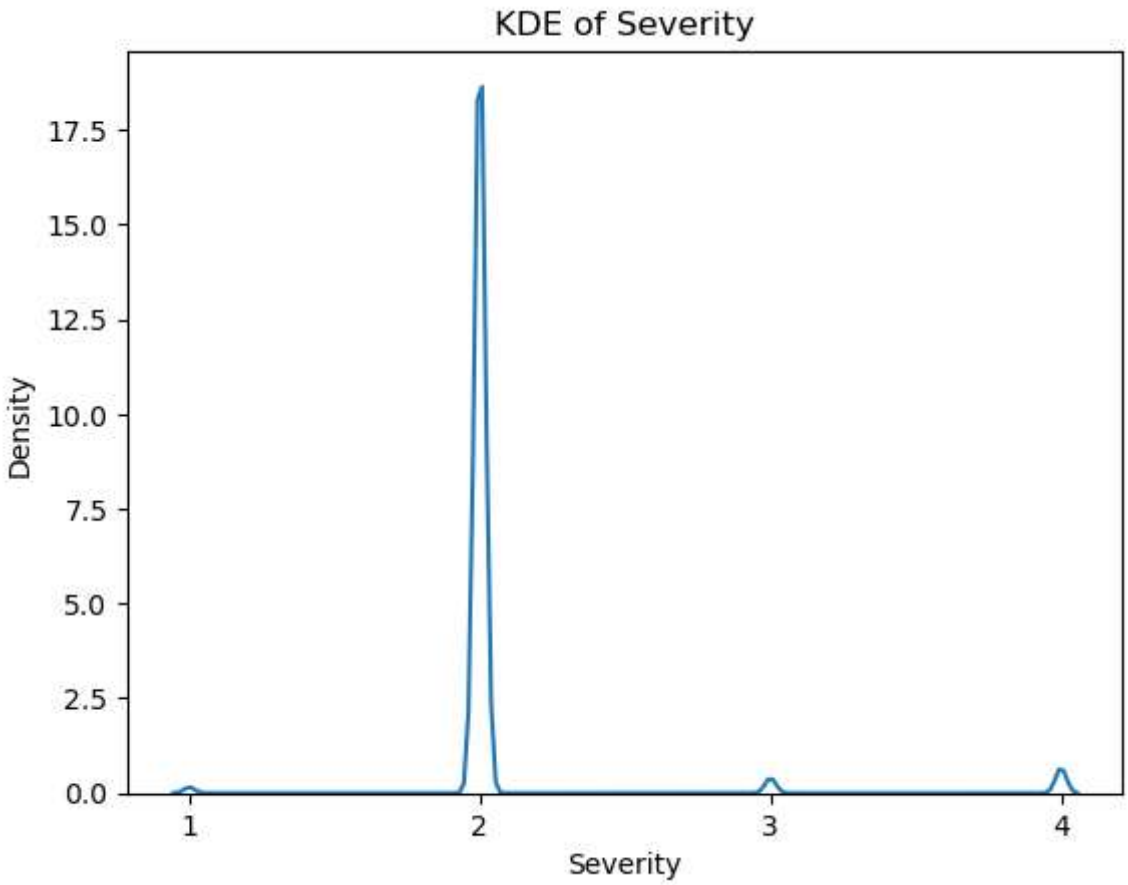
Out[ ]:

	Severity	Start_Lat	Start_Lng	End_Lat	End_Lng	Distance(mi)	Tempe
count	3.551471e+06	3.551471e+06	3.551471e+06	3.551471e+06	3.551471e+06	3.551471e+06	3.55
mean	2.073811e+00	3.610664e+01	-9.522731e+01	3.610684e+01	-9.522699e+01	8.464891e-01	6.11
std	3.810629e-01	5.348681e+00	1.801586e+01	5.348880e+00	1.801554e+01	1.816920e+00	1.93
min	1.000000e+00	2.456603e+01	-1.245481e+02	2.456601e+01	-1.245457e+02	0.000000e+00	-4.50
25%	2.000000e+00	3.316955e+01	-1.175579e+02	3.317011e+01	-1.175560e+02	6.700000e-02	4.80
50%	2.000000e+00	3.602911e+01	-8.731111e+01	3.603224e+01	-8.731098e+01	2.640000e-01	6.30
75%	2.000000e+00	4.012564e+01	-8.021014e+01	4.012570e+01	-8.020992e+01	9.230000e-01	7.60
max	4.000000e+00	4.900050e+01	-6.748413e+01	4.900222e+01	-6.748413e+01	1.551860e+02	1.96

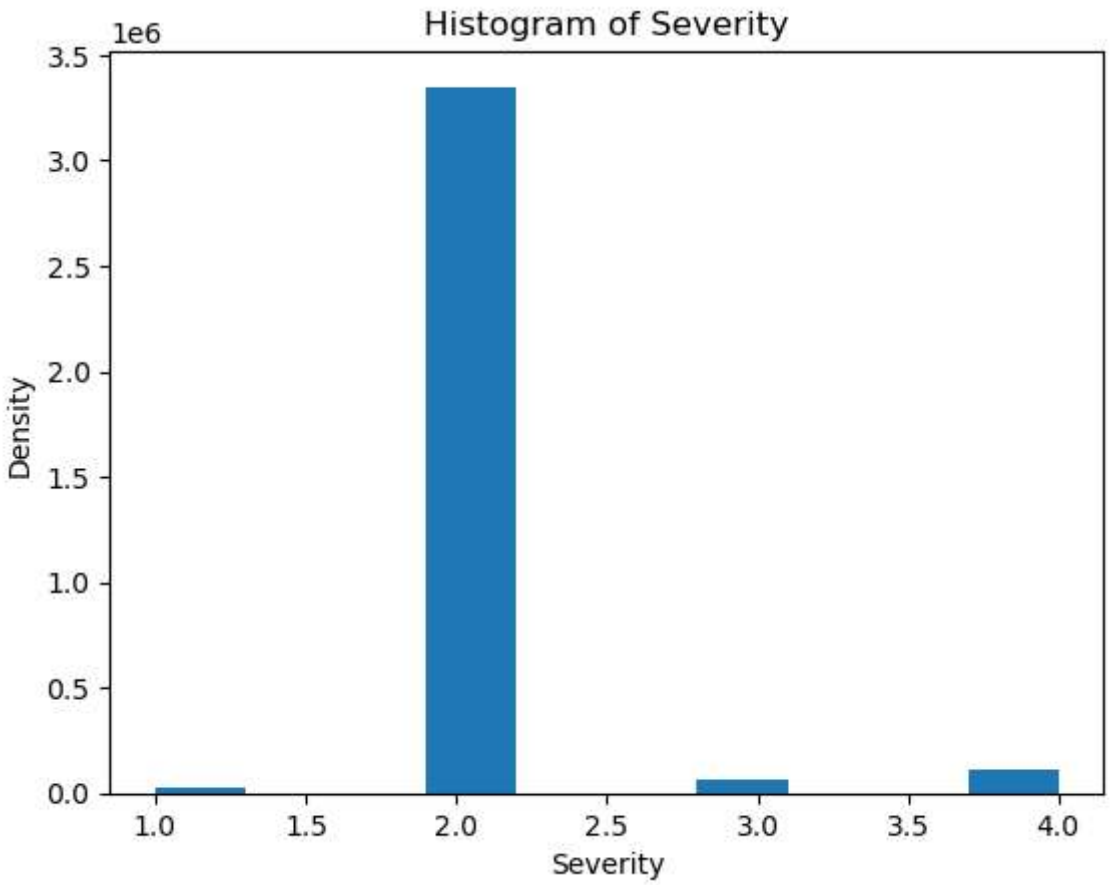
8 rows × 32 columns

Getting the KDE of the Severity of Accidents

In [ ]:
```
# Plot the KDE
sns.kdeplot(df['Severity'])
plt.title('KDE of Severity')
plt.xlabel('Severity')
plt.ylabel('Density')

xmin, xmax = plt.xlim()
plt.xticks(np.arange(np.ceil(xmin), np.floor(xmax) + 1),
           [str(int(val)) for val in np.arange(np.ceil(xmin), np.floor(xmax) + 1)])

plt.show()
```



In [ ]:
```
# Elan internal comments to group
# While we have mostly histograms above and I can see the value in trying to include d
# is basically non-existent, and it just ends up essentially being a histogram
plt.hist(df['Severity'])
plt.title('Histogram of Severity')
plt.xlabel('Severity')
plt.ylabel('Density')
plt.plot;
```



Data Manipluation

Changing some of the Values in the Data Frame to reduce the amount of unique values

Doing this makes our computational time much faster, and our system requirements much lower.

In [ ]:
```
weather_conditions = df['Weather_Condition'].unique()

print(weather_conditions)
print(len(weather_conditions))
```

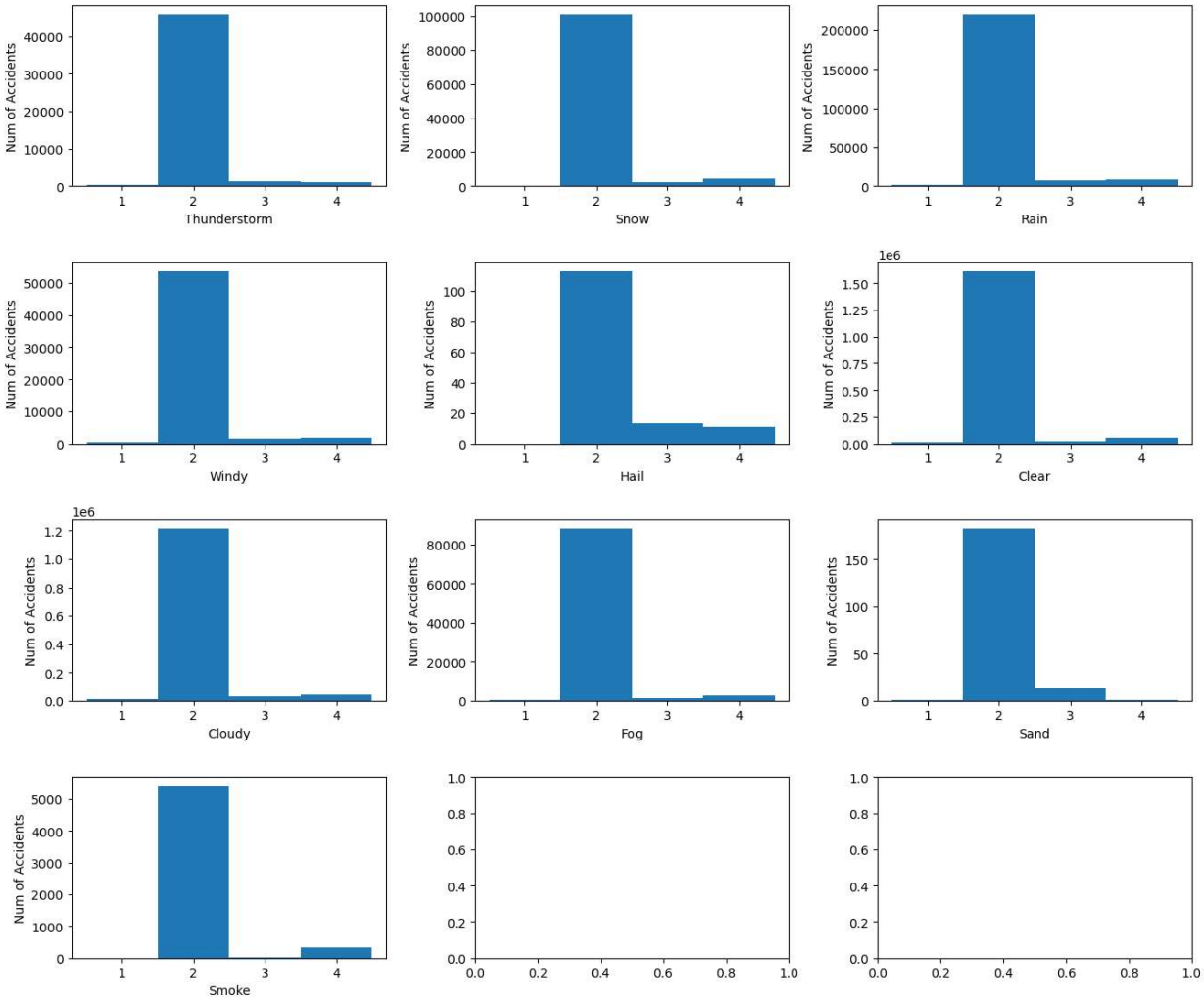
```
['Overcast' 'Clear' 'Cloudy' 'Light Rain' 'Partly Cloudy' 'Light Snow'
 'Fair' 'Mostly Cloudy' 'Heavy Snow' 'Snow' 'Light Freezing Rain' 'Fog'
 'Mist' 'Light Freezing Fog' 'Cloudy / Windy' 'Blowing Snow' 'Haze' 'Rain'
 'Light Freezing Drizzle' 'Heavy Snow / Windy' 'Light Drizzle'
 'Heavy Drizzle' 'Drizzle' 'Light Ice Pellets' 'N/A Precipitation'
 'Fair / Windy' 'Scattered Clouds' 'Light Freezing Rain / Windy'
 'Wintry Mix' 'Heavy Rain' 'Light Thunderstorms and Rain'
 'Partly Cloudy / Windy' 'Light Rain Showers' 'Snow / Windy'
 'Heavy Thunderstorms with Small Hail' 'Heavy Thunderstorms and Rain'
 'Thunderstorms and Rain' 'Light Rain with Thunder' 'Light Rain / Windy'
 'Shallow Fog' 'Ice Pellets' 'Thunderstorm' 'Drizzle and Fog'
 'T-Storm / Windy' 'Thunder' 'Heavy T-Storm' 'Thunder in the Vicinity'
 'T-Storm' 'Smoke' 'Light Blowing Snow' 'Heavy Rain / Windy' 'Small Hail'
 'Rain / Windy' 'Light Drizzle / Windy' 'Light Snow / Windy'
 'Haze / Windy' 'Mostly Cloudy / Windy' 'Showers in the Vicinity'
 'Patches of Fog' 'Fog / Windy' 'Snow and Sleet' 'Light Snow with Thunder'
 'Snow and Sleet / Windy' 'Blowing Snow / Windy' 'Heavy Snow with Thunder'
 'Wintry Mix / Windy' 'Heavy T-Storm / Windy' 'Light Rain Shower'
 'Thunder / Windy' 'Squalls / Windy' 'Sand / Dust Whirlwinds'
 'Thunder and Hail / Windy' 'Tornado' 'Blowing Dust / Windy'
 'Blowing Dust' 'Smoke / Windy' 'Partial Fog' 'Freezing Rain' 'Sleet'
 'Light Snow and Sleet' 'Drizzle / Windy' 'Drifting Snow'
 'Light Snow and Sleet / Windy' 'Light Sleet' 'Light Snow Shower'
 'Heavy Rain Shower' 'Rain Shower' 'Thunder and Hail'
 'Sand / Dust Whirls Nearby' 'Patches of Fog / Windy' 'Widespread Dust'
 'Thunder / Wintry Mix / Windy' 'Light Sleet / Windy' 'Sleet / Windy'
 'Freezing Drizzle' 'Blowing Sand' 'Heavy Freezing Drizzle' 'Snow Grains'
 'Thunder / Wintry Mix' 'Heavy Freezing Rain' 'Freezing Rain / Windy'
 'Heavy Sleet' 'Hail' 'Widespread Dust / Windy'
 'Sand / Dust Whirlwinds / Windy' 'Squalls' 'Heavy Rain Shower / Windy'
 'Sand / Windy' 'Snow and Thunder / Windy' 'Blowing Snow Nearby'
 'Snow and Thunder' 'Heavy Sleet and Thunder' 'Sleet and Thunder'
 'Shallow Fog / Windy' 'Light Snow Shower / Windy' 'Funnel Cloud'
 'Drifting Snow / Windy' 'Heavy Sleet / Windy' 'Light Rain Shower / Windy']
119
```

In [ ]:
```
df.loc[df["Weather_Condition"].str.contains("Thunder|T-Storm", na=False), "Weather_Cor
df.loc[df["Weather_Condition"].str.contains("Snow|Sleet|Wintry", na=False), "Weather_C
df.loc[df["Weather_Condition"].str.contains("Rain|Drizzle|Shower", na=False), "Weather
df.loc[df["Weather_Condition"].str.contains("Wind|Squalls", na=False), "Weather_Condit
df.loc[df["Weather_Condition"].str.contains("Hail|Pellets", na=False), "Weather_Condit
df.loc[df["Weather_Condition"].str.contains("Fair", na=False), "Weather_Condition"] =
df.loc[df["Weather_Condition"].str.contains("Cloud|Overcast", na=False), "Weather_Cond
df.loc[df["Weather_Condition"].str.contains("Mist|Haze|Fog", na=False), "Weather_Condi
df.loc[df["Weather_Condition"].str.contains("Sand|Dust", na=False), "Weather_Condition
df.loc[df["Weather_Condition"].str.contains("Smoke|Volcanic Ash", na=False), "Weather_
df.loc[df["Weather_Condition"].str.contains("N/A Precipitation", na=False), "Weather_C
```

Severity by Weather Condition

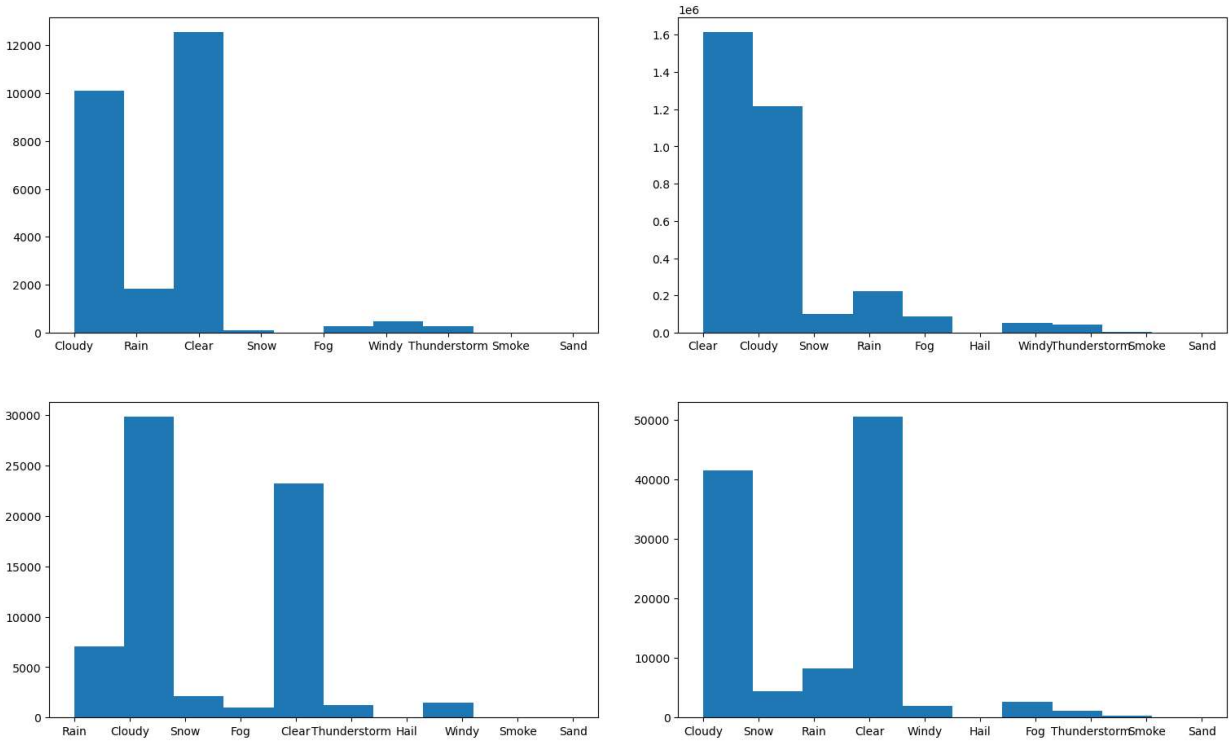
In [ ]:
```
weather_types = ['Thunderstorm','Snow','Rain','Windy','Hail','Clear','Cloudy','Fog','S
fig, axes = plt.subplots(nrows=4,ncols=3,figsize=(14,12))
fig.tight_layout(pad = 4.0)

i = 0
for weather in weather_types:
    row = i // 3
    col = i % 3
```

```
    weatherdf = df[df['Weather_Condition'] == weather]
    axes[row,col].hist(weatherdf['Severity'],bins=4,range=[0.5,4.5])
    axes[row,col].set_xlabel(weather)
    axes[row,col].set_ylabel('Num of Accidents')
    i += 1

plt.show;
```



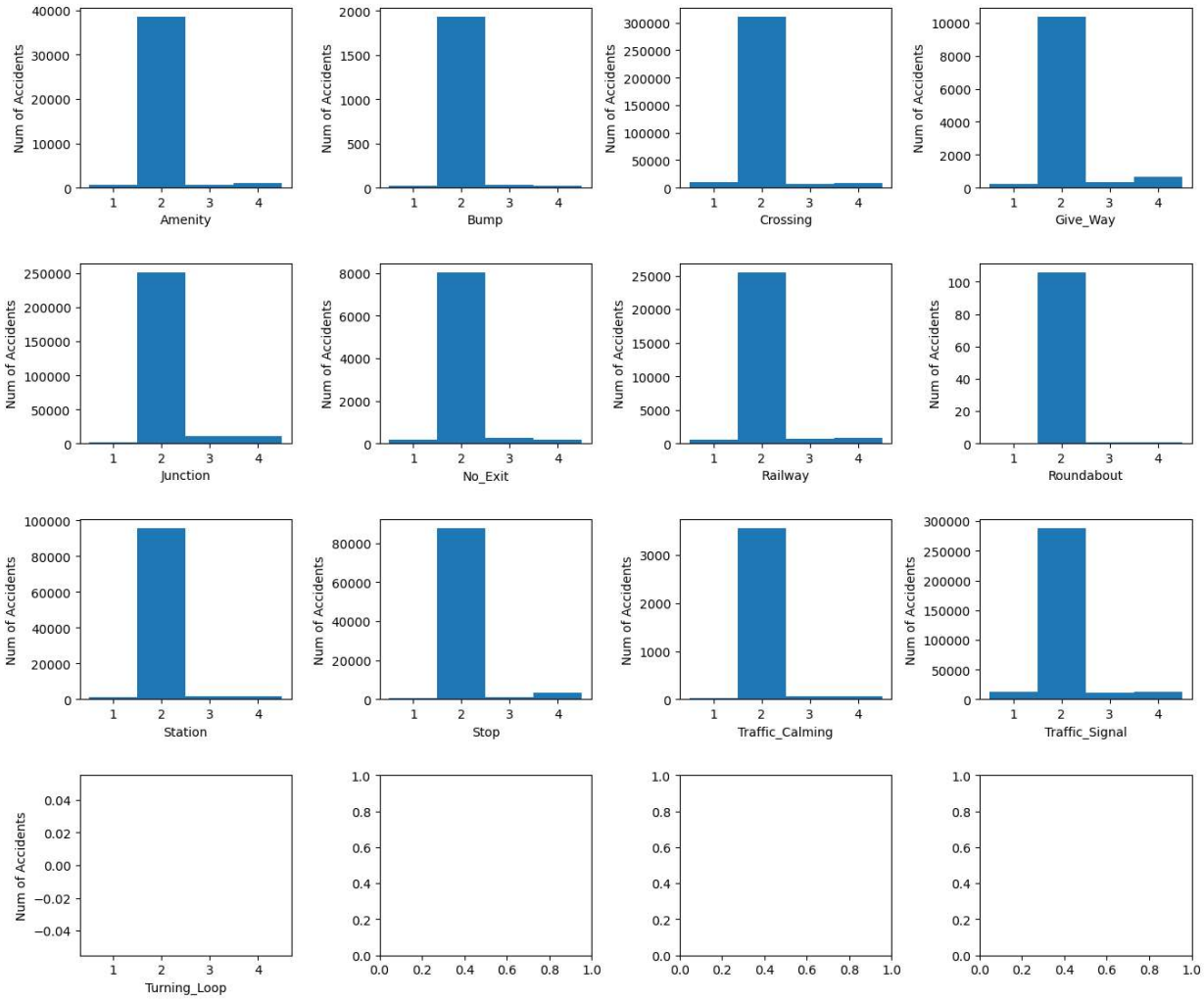
In [ ]:
```
severity_levels = [1,2,3,4]

fig, axes = plt.subplots(nrows=2,ncols=2,figsize=(16,10))
fig.tight_layout(pad = 4.0)
weather_types = ['Thunderstorm','Snow','Rain','Windy','Hail','Clear','Cloudy','Fog','S

i = 0
for severity in severity_levels:
    row = i // 2
    col = i % 2
    severityDF = df[df['Severity'] == severity]
    severityDF = severityDF[df['Weather_Condition'].isin(weather_types)]
    axes[row,col].hist(severityDF['Weather_Condition'],bins = 10)
    i += 1

plt.show;
```

```
C:\Users\user\AppData\Local\Temp\ipykernel_8144\598970995.py:12: UserWarning:

Boolean Series key will be reindexed to match DataFrame index.
```



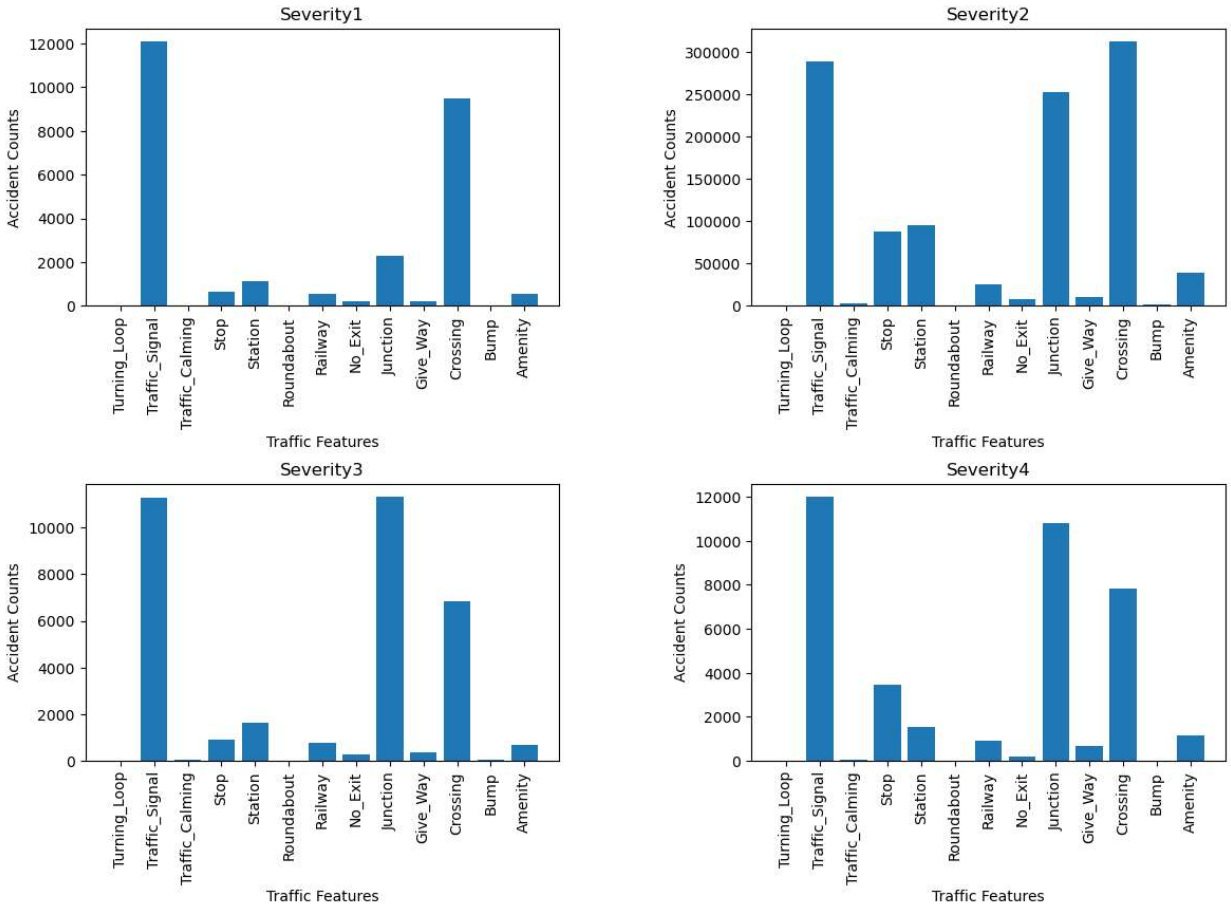
Severity by Environmental Features

In [ ]:
```
environmentalFeatures = ['Turning_Loop', 'Traffic_Signal','Traffic_Calming','Stop','St
environmentalFeatures =  environmentalFeatures[::-1]
fig, axes = plt.subplots(nrows=4,ncols=4,figsize=(14,12))
fig.tight_layout(pad = 4.0)

i = 0
for environmentFeature in environmentalFeatures:
    row = i // 4
    col = i % 4
    envFeatdf = df[df[environmentFeature] == 1]
    axes[row,col].hist(envFeatdf['Severity'],bins=4,range=[0.5,4.5])
    axes[row,col].set_xlabel(environmentFeature)
    axes[row,col].set_ylabel('Num of Accidents')
    i += 1

plt.show;
```



In [ ]:
```
severity_levels = [1,2,3,4]

fig, axes = plt.subplots(nrows=2,ncols=2,figsize=(14,10))
fig.tight_layout(pad = 10.0)

i = 0
for severity in severity_levels:
    row = i // 2
    col = i % 2
    severityDF = df[df['Severity'] == severity]
    labels = ['Turning_Loop','Traffic_Signal','Traffic_Calming','Stop','Station','Rour
    vals = []
    for label in labels:
        vals.append(severityDF[label].sum())
    axes[row,col].bar(labels,vals)
    axes[row,col].set_xticklabels(labels, rotation=90)
    axes[row,col].set_xlabel('Traffic Features')
    axes[row,col].set_ylabel('Accident Counts')
    axes[row,col].set_title(('Severity' + str(severity)))

    i += 1

plt.plot();
```

```
C:\Users\user\AppData\Local\Temp\ipykernel_8144\2145577616.py:16: UserWarning:

FixedFormatter should only be used together with FixedLocator
```



Checking the Wind Direction

In [ ]:
```
df["Wind_Direction"].unique()
```

Out[ ]:
```
array(['WNW', 'NW', 'CALM', 'North', 'VAR', 'WSW', 'SW', 'South', 'ENE',
       'N', 'SSW', 'Variable', 'West', 'NNW', 'NNE', 'S', 'SE', 'East',
       'SSE', 'ESE', 'W', 'E', 'NE'], dtype=object)
```

Changing the Wind Directions

In [ ]:
```
df.loc[df["Wind_Direction"] == "CALM", "Wind_Direction"] = "Calm"
df.loc[df["Wind_Direction"] == "VAR", "Wind_Direction"] = "Variable"
df.loc[df["Wind_Direction"] == "East", "Wind_Direction"] = "E"
df.loc[df["Wind_Direction"] == "North", "Wind_Direction"] = "N"
df.loc[df["Wind_Direction"] == "South", "Wind_Direction"] = "S"
df.loc[df["Wind_Direction"] == "West", "Wind_Direction"] = "W"

df["Wind_Direction"] = df["Wind_Direction"].map(lambda x : x if len(x) != 3 else x[1:]

df["Wind_Direction"].unique()
```

Out[ ]:
```
array(['NW', 'Calm', 'N', 'Variable', 'SW', 'S', 'NE', 'W', 'SE', 'E'],
      dtype=object)
```

Section 4: Model Selection

We are going to be deciding on which model to use to predict the Severity of the accident

Option 1: `Classification Tree`

Option 2: `Logistic Regression`

We are going to start with the `Classification Model` and see how it performs.

In [ ]:
```
## Importing all of the libraries needed for the Classification and log from sklearn t

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import learning_curve
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.calibration import CalibratedClassifierCV
from sklearn.calibration import calibration_curve
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
```

`Classification Model`

In [ ]:
```
# Dropping na values from the df
df = df.dropna()

# Printing all of the headers to see what we have to work with

print(df.columns)

# Printing the data types of the columns

print(df.dtypes)
```

```
Index(['ID', 'Source', 'Severity', 'Start_Time', 'End_Time', 'Start_Lat',
       'Start_Lng', 'End_Lat', 'End_Lng', 'Distance(mi)', 'City', 'County',
       'State', 'Zipcode', 'Weather_Timestamp', 'Temperature(F)',
       'Wind_Chill(F)', 'Humidity(%)', 'Pressure(in)', 'Visibility(mi)',
       'Wind_Direction', 'Wind_Speed(mph)', 'Precipitation(in)',
       'Weather_Condition', 'Amenity', 'Bump', 'Crossing', 'Give_Way',
       'Junction', 'No_Exit', 'Railway', 'Roundabout', 'Station', 'Stop',
       'Traffic_Calming', 'Traffic_Signal', 'Turning_Loop', 'Sunrise_Sunset',
       'Year', 'Month', 'Weekday', 'Day', 'Hour', 'Minute'],
      dtype='object')
ID                          object
Source                      object
Severity                     int64
Start_Time          datetime64[ns]
End_Time                    object
Start_Lat                  float64
Start_Lng                  float64
End_Lat                    float64
End_Lng                    float64
Distance(mi)               float64
City                        object
County                      object
State                       object
Zipcode                     object
Weather_Timestamp           object
Temperature(F)             float64
Wind_Chill(F)              float64
Humidity(%)                float64
Pressure(in)               float64
Visibility(mi)             float64
Wind_Direction              object
Wind_Speed(mph)            float64
Precipitation(in)          float64
Weather_Condition           object
Amenity                      int64
Bump                         int64
Crossing                     int64
Give_Way                     int64
Junction                     int64
No_Exit                      int64
Railway                      int64
Roundabout                   int64
Station                      int64
Stop                         int64
Traffic_Calming              int64
Traffic_Signal               int64
Turning_Loop                 int64
Sunrise_Sunset              object
Year                         int64
Month                        int64
Weekday                      int64
Day                          int64
Hour                         int64
Minute                       int64
dtype: object
```

One Hot Encoding the Weather Conditions column

In [ ]:
```
### One hot encoding the Weather_Condition column only
#
#df = pd.get_dummies(df, columns=['Weather_Condition'])
#
## Printing the columns to see if the one hot encoding worked
#
#print(df.column)
```

Rounding all of the integer values to the nearest whole number, and changing the data type to integer

In [ ]:
```
# changing the data type of the data types from floats to ints

for col in df.columns:
    if df[col].dtype == 'float64':
        df[col] = np.round(df[col]).astype('int64')

## Dropping all objects from the df
df = df.select_dtypes(exclude=['object'])
df = df.select_dtypes(exclude=['datetime64[ns]'])

# Printing the data types of the columns to verify
print(df.dtypes)
```

```
Severity                        int64
Start_Lat                       int64
Start_Lng                       int64
End_Lat                         int64
End_Lng                         int64
Distance(mi)                    int64
Temperature(F)                  int64
Wind_Chill(F)                   int64
Humidity(%)                     int64
Pressure(in)                    int64
Visibility(mi)                  int64
Wind_Speed(mph)                 int64
Precipitation(in)               int64
Amenity                         int64
Bump                            int64
Crossing                        int64
Give_Way                        int64
Junction                        int64
No_Exit                         int64
Railway                         int64
Roundabout                      int64
Station                         int64
Stop                            int64
Traffic_Calming                 int64
Traffic_Signal                  int64
Turning_Loop                    int64
Year                            int64
Month                           int64
Weekday                         int64
Day                             int64
Hour                            int64
Minute                          int64
Weather_Condition_Clear         uint8
Weather_Condition_Cloudy        uint8
Weather_Condition_Fog           uint8
Weather_Condition_Hail          uint8
Weather_Condition_Rain          uint8
Weather_Condition_Sand          uint8
Weather_Condition_Smoke         uint8
Weather_Condition_Snow          uint8
Weather_Condition_Thunderstorm  uint8
Weather_Condition_Tornado       uint8
Weather_Condition_Windy         uint8
dtype: object
```

Creating the Models Data Frames

Balancing the data to be equal in each category

In [ ]:
```
# Balacing the data ti have the same amount of each severity

df = df.groupby('Severity').apply(lambda x: x.sample(n=25567)).reset_index(drop = True

#Printing the counts of the severity to verify that they are all the same

print(df['Severity'].value_counts())
```

```
1    25567
2    25567
3    25567
4    25567
Name: Severity, dtype: int64
```

In [ ]:
```
# splitting the data into features and target

X = df.drop('Severity', axis=1)  # Features
y = df['Severity']               # Target
```

In [ ]:
```
# Setting up the Training Sets and Testing Sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=
```

Standardizing the Data

In [ ]:
```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In [ ]:
```
# Creating the classifier

clf = RandomForestClassifier(random_state=42)
clf.fit(X_train, y_train)
```

Out[ ]:
```
RandomForestClassifier
RandomForestClassifier(random_state=42)
```

Testing The Accuracy of the Model

In [ ]:
```
y_pred = clf.predict(X_test)

# Print Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.4f}")

# Print Classification Report
print(classification_report(y_test, y_pred))
```

```
Accuracy: 0.7929
              precision    recall  f1-score   support

           1       0.89      0.94      0.91      5096
           2       0.78      0.74      0.76      5143
           3       0.78      0.86      0.82      5169
           4       0.71      0.64      0.67      5046

    accuracy                           0.79     20454
   macro avg       0.79      0.79      0.79     20454
weighted avg       0.79      0.79      0.79     20454
```

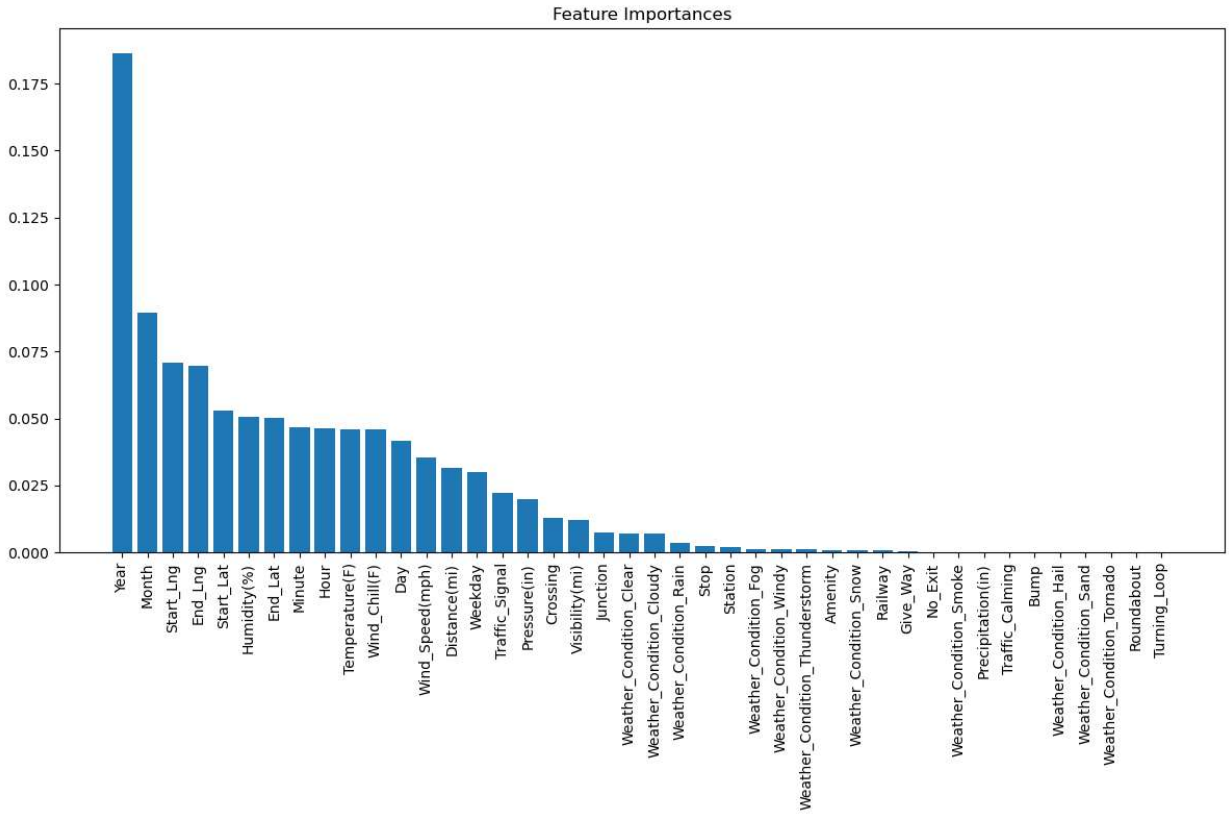
Confusion Matrix

In [ ]:
```
# compute Confusion Matrix
matrix = confusion_matrix(y_test, y_pred)

# plotting
plt.figure(figsize=(10,7))
sns.heatmap(matrix, annot=True, cmap='Blues', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

Feature Importance

In [ ]:
```
# Extract feature importances from the trained classifier
importances = clf.feature_importances_

# Sort the features based on importance
sorted_indices = np.argsort(importances)[::-1]

# Plot
plt.figure(figsize=(12, 8))
plt.title("Feature Importances")
plt.bar(range(X_train.shape[1]), importances[sorted_indices], align="center")
plt.xticks(range(X_train.shape[1]), X.columns[sorted_indices], rotation=90)
```

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

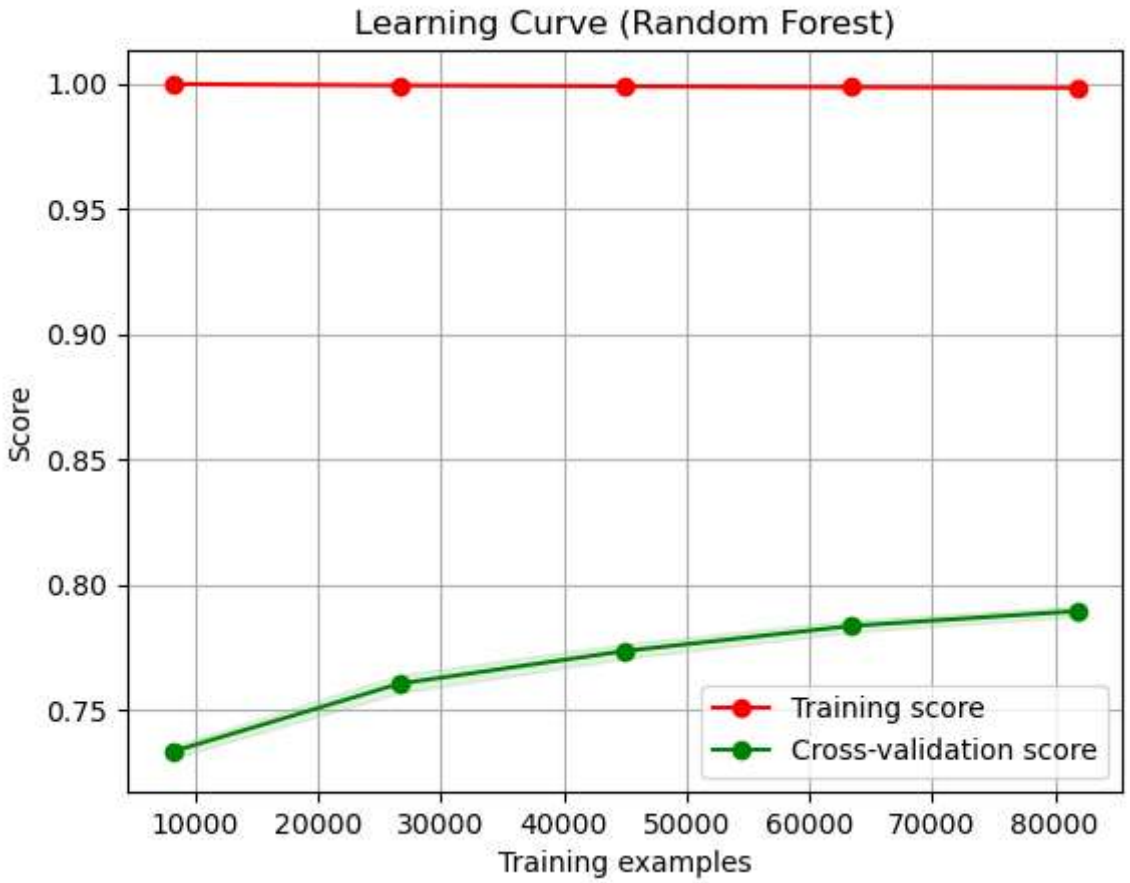


Learning Curve

In [ ]:

```
def plot_learning_curve(estimator, X, y, title="Learning Curves", ylim=None, cv=None,
    plt.figure()
    plt.title(title)
    if ylim is not None:
        plt.ylim(*ylim)
    plt.xlabel("Training examples")
    plt.ylabel("Score")

    train_sizes, train_scores, test_scores = learning_curve(estimator, X, y, cv=cv, n_

    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)

    plt.grid()

    plt.fill_between(train_sizes, train_scores_mean - train_scores_std, train_scores_m
    plt.fill_between(train_sizes, test_scores_mean - test_scores_std, test_scores_mear

    plt.plot(train_sizes, train_scores_mean, 'o-', color="r", label="Training score")
    plt.plot(train_sizes, test_scores_mean, 'o-', color="g", label="Cross-validation s

    plt.legend(loc="best")
    return plt

# Call the function to plot the learning curve
```

```
X_full = np.vstack((X_train, X_test))
y_full = np.hstack((y_train, y_test))
plot_learning_curve(clf, X_full, y_full, cv=5, n_jobs=-1, title="Learning Curve (Randc
plt.show()
```



Calibration and Reliability Diagram

In [ ]:
```
# splitting the data into Train and Temp
X_train_temp, X_temp, y_train_temp, y_temp = train_test_split(X, y, test_size=0.5, rar

# Splitting it even further
X_calib, X_test, y_calib, y_test = train_test_split(X_temp, y_temp, test_size=0.5, rar


# Train RandomForest on the training data
clf = RandomForestClassifier(random_state=42)
clf.fit(X_train_temp, y_train_temp)

# Calibrate on the calibration data
calibrator = CalibratedClassifierCV(clf, method='sigmoid', cv='prefit')
calibrator.fit(X_calib, y_calib)
```

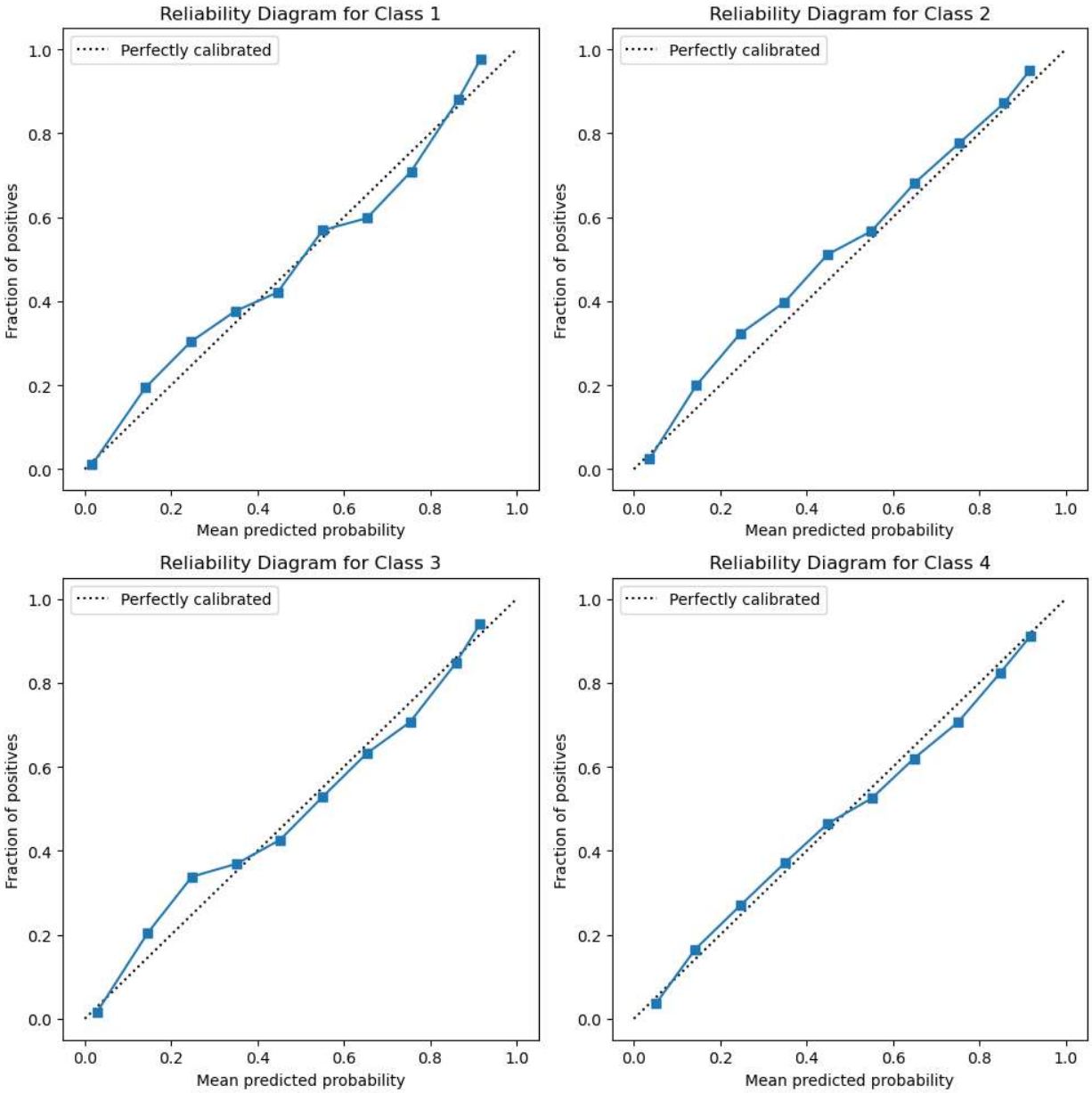
Out[ ]:

▸ **CalibratedClassifierCV**

▸ **estimator: RandomForestClassifier**

▸ RandomForestClassifier

In [ ]:
```
# 4 classes
n_classes = 4

# Create subplots for each class
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 10))

for i in range(n_classes):
    prob_pos = calibrator.predict_proba(X_test)[:, i]

    # Consider current class as positive and rest as negative
    y_class = np.where(y_test == i + 1, 1, 0)

    fraction_of_positives, mean_predicted_value = calibration_curve(y_class, prob_pos,

    ax = axes.flat[i]
    ax.plot([0, 1], [0, 1], "k:", label="Perfectly calibrated")
    ax.plot(mean_predicted_value, fraction_of_positives, "s-")
    ax.set_xlabel("Mean predicted probability")
    ax.set_ylabel("Fraction of positives")
    ax.set_title(f"Reliability Diagram for Class {i + 1}")
    ax.legend()

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



Boosting The Model

In [ ]:
```
gb_clf = GradientBoostingClassifier(n_estimators=100, random_state=42)
gb_clf.fit(X_train, y_train)
```

Out[ ]:
```
GradientBoostingClassifier
GradientBoostingClassifier(random_state=42)
```

In [ ]:
```
y_pred_gb = gb_clf.predict(X_test)
```

```
c:\Users\user\anaconda3\Lib\site-packages\sklearn\base.py:457: UserWarning:

X has feature names, but GradientBoostingClassifier was fitted without feature names
```

In [ ]:
```
# Print Accuracy
accuracy_gb = accuracy_score(y_test, y_pred_gb)
```

```
print(f"Gradient Boosting Accuracy: {accuracy_gb:.4f}")

# Print Classification Report
print(classification_report(y_test, y_pred_gb))

# Print Confusion Matrix
cm = confusion_matrix(y_test, y_pred_gb)
print("Confusion Matrix:")
print(cm)
```

```
Gradient Boosting Accuracy: 0.2508
              precision    recall  f1-score   support

           1       0.00      0.00      0.00      6235
           2       0.36      0.16      0.22      6519
           3       0.00      0.00      0.00      6455
           4       0.24      0.84      0.37      6358

    accuracy                           0.25     25567
   macro avg       0.15      0.25      0.15     25567
weighted avg       0.15      0.25      0.15     25567

Confusion Matrix:
[[   0  274    0 5961]
 [   0 1046    0 5473]
 [   0  564    0 5891]
 [   0  993    0 5365]]
```

```
c:\Users\user\anaconda3\Lib\site-packages\sklearn\metrics\_classification.py:1469: Un
definedMetricWarning:

Precision and F-score are ill-defined and being set to 0.0 in labels with no predicte
d samples. Use `zero_division` parameter to control this behavior.

c:\Users\user\anaconda3\Lib\site-packages\sklearn\metrics\_classification.py:1469: Un
definedMetricWarning:

Precision and F-score are ill-defined and being set to 0.0 in labels with no predicte
d samples. Use `zero_division` parameter to control this behavior.

c:\Users\user\anaconda3\Lib\site-packages\sklearn\metrics\_classification.py:1469: Un
definedMetricWarning:

Precision and F-score are ill-defined and being set to 0.0 in labels with no predicte
d samples. Use `zero_division` parameter to control this behavior.
```

Gradient Boosting Model Evaluation

Accuracy:

The model accuracy is **25.08%**. This means that only about 25.08% of the total predictions made by the model are correct. Given that we have four classes, a completely random guess would have an accuracy of 25%. This suggests that our model isn't performing significantly better than a random guess for this dataset.

Precision, Recall, and F1-score:

Class 1:

- **Precision**: 0.00% - Out of all the instances predicted as class 1, none were actually class 1.
- **Recall**: 0.00% - Out of all the actual class 1 instances, none were predicted correctly.
- **F1-score**: 0.00% - The harmonic mean of precision and recall for class 1, indicating very poor performance for this class.

Class 2:

- **Precision**: 36% - Out of all the instances predicted as class 2, 36% were actually class 2.
- **Recall**: 16% - Out of all the actual class 2 instances, 16% were predicted correctly.
- **F1-score**: 22% - Indicates a balance between precision and recall for class 2. However, this is not particularly high, suggesting room for improvement.

Class 3:

- **Precision, Recall, and F1-score**: All metrics are 0.00%, which indicates the model isn't predicting this class correctly at all.

Class 4:

- **Precision**: 24% - Out of all instances predicted as class 4, 24% were actually class 4.
- **Recall**: 84% - A significant majority (i.e., 84%) of actual class 4 instances were predicted correctly.
- **F1-score**: 37% - Represents the balance between precision and recall for class 4.

Confusion Matrix:

The confusion matrix provides a breakdown of predictions for each class. The rows of the matrix represent the actual class and the columns represent the predicted class.

From the matrix, it's evident that the model struggles to predict class 1 and 3 correctly, often misclassifying them as class 4. Class 2 and 4 are the most predicted, with class 4 having the highest recall.

Interpretation:

1. The model is performing only slightly better than a random guess.
2. The model struggles to distinguish between the classes, especially class 1 and class 3.
3. Class 4 is the most correctly predicted class, while class 2 predictions have the highest precision but low recall.
4. There's significant room for improvement. It might be beneficial to consider additional feature engineering, hyperparameter tuning, gathering more data, or trying different algorithms.

Lastly, it's worth noting that metrics such as precision, recall, and F1-score can sometimes be more informative than just accuracy, especially in cases where the classes are imbalanced or if the cost of misclassification varies between classes.