ETL Project (Extract, Transform, Load)

Traffic Collisions Analysis

Group: DSC 71

Members:

Komal Patil : komal2015patil@gmail.com

Nishant Gaurav : nishantgaurav@gmail.com

Sakshi Sihag : sakshisihag20@gmail.com

Sayan Debnath : sayandebnath1994@gmail.com

Contents

- Business Value.
- Objective.
- Data Preparation.
- Data Cleaning.
- Exploratory Data Analysis (Finding Patterns).
- Data Querying.
- Key Insights and Recommendations.
- Conclusion.
- Appendix : AWS Environment Setup.

Business Value

Traffic collisions pose significant risks to public safety, requiring continuous monitoring and analysis to enhance road safety measures. Government agencies, city planners and policymakers must leverage data-driven insights to improve infrastructure, optimise traffic management and implement preventive measures. In this case study we have been asked to analyse California traffic collision data to uncover patterns related to accident severity, location-based risks and key contributing factors.

The task will be to analyse historical crash data to derive actionable insights that can drive policy improvements and safety interventions. The analysis will help identify high-risk areas, categorise accidents by severity and contributing factors and store the processed data for scalable and long-term storage. By leveraging big data analytics and cloud-based storage, urban planners and traffic authorities can enhance road safety strategies, reduce accident rates and improve public transportation planning.

Objective

Using the tech stack below, process the given large data on traffic collison.

- Apache PySpark,
- Amazon S3
- Amazon EMR Cluster

- Traffic collisions pose significant risks to public safety, requiring continuous monitoring and analysis to enhance road safety measures
- Government agencies, city planners, and policymakers must leverage data-driven insights to improve infrastructure, optimize traffic management, and implement preventive measures
- This assignment involves creating an ETL pipeline to analyse California Traffic Collision data obtained from State-wide Integrated Traffic Records System (SWITRS)
- As an analyst examining traffic safety trends, our task is to analyse historical crash data to derive actionable insights that can drive policy improvements and uncover patterns related to accident severity, location-based risks, and key contributing factors
- The ETL pipeline we are building will help identify high-risk areas, categorise accidents by severity and contributing factors, and store the processed data

Data Preparation

1. Per instruction, dataset was downloaded from

https://kh3-ls-storage.s3.us-east-1.amazonaws.com/UPGrad/Crash_Data_Analysis_Dataset.zip

2. Upload the zipped csv files to Amazon S3.

3. Download to EMR master node and unzip them.

4. Push the csv files to Hadoop filesystem under /user/livy

```
download: s3://collison-crash-data/Crash_Data_Analysis_Dataset.zip to ./Crash_Data_Analysis_Dataset.zip
Archive:  Crash_Data_Analysis_Dataset.zip
  inflating: sample_case_ids.csv
  inflating: sample_victims.csv
  inflating: sample_parties.csv
  inflating: sample_collisions.csv
total 1091320
-rw-r--r--. 1 hadoop hadoop  15742289 Mar  2 18:20 sample_case_ids.csv
-rw-r--r--. 1 hadoop hadoop 419377774 Mar  2 18:20 sample_collisions.csv
-rw-r--r--. 1 hadoop hadoop 429599180 Mar  2 18:21 sample_parties.csv
-rw-r--r--. 1 hadoop hadoop 113985422 Mar  2 18:21 sample_victims.csv
-rw-r--r--. 1 hadoop hadoop 138792969 Jun 11 16:23 Crash_Data_Analysis_Dataset.zip
Found 5 items
drwxr-xr-x   - livy   livy          0 2025-06-14 16:38 /user/livy/.sparkStaging
-rw-r--r--   1 hadoop livy   15742289 2025-06-14 16:41 /user/livy/sample_case_ids.csv
-rw-r--r--   1 hadoop livy  419377774 2025-06-14 16:41 /user/livy/sample_collisions.csv
-rw-r--r--   1 hadoop livy  429599180 2025-06-14 16:41 /user/livy/sample_parties.csv
-rw-r--r--   1 hadoop livy  113985422 2025-06-14 16:41 /user/livy/sample_victims.csv
```

Read the .csv files from Hadoop filesystem

```
1 # Write code to load the data and check the schema
2 victims_df = spark.read.load("/user/livy/sample_victims.csv", format = "csv", header = "true", inferSchema = "true")
```

Read the inferred schema

```
1 print("Victims Schema as read by PySpark:\n")
2 victims_df.printSchema()
```

```
VBox()
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layout(height='25px', width='50%'),…
Victims Schema as read by PySpark:

root
 |-- id: integer (nullable = true)
 |-- case_id: double (nullable = true)
 |-- party_number: integer (nullable = true)
 |-- victim_role: string (nullable = true)
 |-- victim_sex: string (nullable = true)
 |-- victim_age: double (nullable = true)
 |-- victim_degree_of_injury: string (nullable = true)
 |-- victim_seating_position: string (nullable = true)
 |-- victim_safety_equipment_1: string (nullable = true)
 |-- victim_safety_equipment_2: string (nullable = true)
 |-- victim_ejected: string (nullable = true)
```

Data Cleaning

1. Inspect the cols in the data-frame for incorrect data types.

2. Define a schema using StructType taking the target cols as StringType. This is needed as a direct map to target data type results into NULL values.

3. Cast from StringType to target correct data type.

4. Special case of case_id: The max length of this col is 22 digits. A decimalType number was defined

large_case_id_decimal_type with 0 precision.

Inspect cols (example: victims file)

```
1 victims_df.select("case_id").distinct().show()
```

```
VBox()
FloatProgress(value=0.0, bar_style='info', descri
+--------------------+
|             case_id|
+--------------------+
|           8474612.0|
|           7130714.0|
|           4714551.0|
|           5866798.0|
|           1304870.0|
|            990175.0|
|           5368293.0|
|           6233401.0|
|3.711010106104204...|
|            470926.0|
|           4845793.0|
|           8044503.0|
|           7071072.0|
|3.404010717121100...|
|         9.0017906E7|
|           8574868.0|
|           5392691.0|
|           1567280.0|
|           6304776.0|
|           2504632.0|
+--------------------+
only showing top 20 rows
```

```
1 victims_df.select("victim_age").distinct().show()
```

```
VBox()
FloatProgress(value=0.0, bar_style='info', descripti
+----------+
|victim_age|
+----------+
|      NULL|
|      29.0|
|     107.0|
|      47.0|
|       1.0|
|      94.0|
|     122.0|
|     103.0|
|     106.0|
|      63.0|
|      82.0|
|      66.0|
|      61.0|
|      46.0|
|      28.0|
|      13.0|
|      26.0|
|      76.0|
|      69.0|
|      98.0|
+----------+
only showing top 20 rows
```

Columns identified for data type correction

Some of the data types in Victims schema need to change

- 'case_id': double --> long int
- 'victim_age': double --> int

Define schema with 'case_id' and 'victim_age' as String and after Spark read, cast to the target data type explicitly. This is intermediate steps is introduced as a direct schema read into target data type is resulting into NULL values in the table.

```
victimSchema = StructType([StructField('id', IntegerType(),True),
                           StructField('case_id', StringType(),True),
                           StructField('party_number', IntegerType(),True),
                           StructField('victim_role', StringType(),True),
                           StructField('victim_sex', StringType(),True),
                           StructField('victim_age', StringType(),True),
                           StructField('victim_degree_of_injury', StringType(),True),
                           StructField('victim_seating_position', StringType(),True),
                           StructField('victim_safety_equipment_1', StringType(),True),
                           StructField('victim_safety_equipment_2', StringType(),True),
                           StructField('victim_ejected', StringType(),True),
                           ])
```

Define a StructType Schema

Special case of ‘case_id’ ➔ 22 digit integer

```
1 victims_df2 = spark.read.load("/user/livy/sample_victims.csv", format = "csv", header = "true", schema = victimSchema)
2 victims_df2.select(max(length(col('case_id')))).show()
```

```
VBox()
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layout(height='25px', width='50%'),…
+--------------------+
|max(length(case_id))|
+--------------------+
|                  22|
+--------------------+
```

'case_id' is an integer of max 22 digits. Define a large decimal type with 22 digits and 0 scale and use it for casting

```
1 large_case_id_decimal_type = DecimalType(22, 0)
```

Define DecimalType(22,0)

Cast the column and update the schema

```
1 victims_df1 = spark.read.load("/user/livy/sample_victims.csv", format = "csv", header = "true", schema = victimSchema)
2
3 victims_df1 = victims_df1.withColumn("case_id", col("case_id").cast(large_case_id_decimal_type))
4
5 victims_df1 = victims_df1.withColumn("victim_age", col("victim_age").cast("int"))
6 print("Victims Schema after modification:\n")
7 victims_df1.printSchema()
8 print("\nSample data from Victims Schema:\n")
9 victims_df1.show(5)
```

```
Victims Schema after modification:

root
 |-- id: integer (nullable = true)
 |-- case_id: decimal(22,0) (nullable = true)
 |-- party_number: integer (nullable = true)
 |-- victim_role: string (nullable = true)
 |-- victim_sex: string (nullable = true)
 |-- victim_age: integer (nullable = true)
 |-- victim_degree_of_injury: string (nullable = true)
 |-- victim_seating_position: string (nullable = true)
 |-- victim_safety_equipment_1: string (nullable = true)
 |-- victim_safety_equipment_2: string (nullable = true)
 |-- victim_ejected: string (nullable = true)
```

Function to identify Missing Values

```
4 # Function to count missing values for each column
5 def find_missing_vals(df, table):
6     print(f"\nMissing values in table {table}:")
7     df.select([sum(when(col(c).isNull() | (col(c) == ''), 1).otherwise(0)).alias(c) for c in df.columns]).show()
```

```
Missing values in table victims:
+---+-------+-----------+-----------+----------+----------+----------------------+----------------------+-----------------
-------+------------------------+-------------+
| id|case_id|party_number|victim_role|victim_sex|victim_age|victim_degree_of_injury|victim_seating_position|victim_safety_equi
pment_1|victim_safety_equipment_2|victim_ejected|
+---+-------+-----------+-----------+----------+----------+----------------------+----------------------+-----------------
-------+------------------------+-------------+
|  0|      0|          0|          1|     24041|     32145|                     0|                  1913|
55215|                  310220|         4272|
+---+-------+-----------+-----------+----------+----------+----------------------+----------------------+-----------------
-------+------------------------+-------------+
```

Missing values in Victims

```
Missing values in table case_ids:
+-------+--------+
|case_id|db_year|
+-------+--------+
|      0|       0|
+-------+--------+
```

Missing values in case_id

Drop Sparse Cols

```python
from pyspark.sql.functions import col, sum, when

def drop_sparse_cols(df, table, threshold=0.5):
    table_count = df.count()
    print(f"\nCount of records in {table}: {table_count}")

    # Obtain count of missing values in each column
    miss_cnt = df.select([
        sum(when(col(c).isNull() | (col(c) == ''), 1)).alias(c) for c in df.columns
    ]).collect()[0].asDict()

    # Handle None (treat as 0)
    sparse_cols = [
        c_name for c_name, miss in miss_cnt.items()
        if (miss or 0) / table_count > threshold
    ]

    print(f"Sparse Cols to drop in table {table} (>{threshold*100}% miss): {sparse_cols}")

    # Drop sparse columns
    return df.drop(*sparse_cols)
```

Define a threshold of 50% for sparse values

```
Count of records in case_ids: 942433
Sparse Cols to drop in table case_ids (>50.0% miss): []

Count of records in collisions: 935791
Sparse Cols to drop in table collisions (>50.0% miss): ['reporting_district', 'caltrans_county', 'caltrans_district', 'state_r
oute', 'postmile', 'location_type', 'side_of_highway', 'pcf_violation_subsection', 'latitude', 'longitude']

Count of records in parties: 1866917
Sparse Cols to drop in table parties (>50.0% miss): []

Count of records in victims: 963933
Sparse Cols to drop in table victims (>50.0% miss): []
```

Output for 4 .csv files

Convert date type

```
3  from pyspark.sql.functions import col, to_date
4  from pyspark.sql.types import IntegerType, DoubleType
5
6  # Step 1: Convert known date columns in PySpark
7  def convert_date_col(df, col_name, date_format="yyyy-MM-dd"):
8      if col_name in df.columns:
9          df = df.withColumn(col_name, to_date(col(col_name), date_format))
10         print(f"Converted column '{col_name}' to date.")
11     return df
12
13 # Apply to relevant columns
14 collisions_df1 = convert_date_col(collisions_df1, 'collision_date')
15 collisions_df1 = convert_date_col(collisions_df1, 'process_date')
16
```

```
VBox()
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=La
Converted column 'collision_date' to date.
Converted column 'process_date' to date.
```

Function to convert into date type format.

Handle Missing Values

```python
#Handle Missing Values

from pyspark.sql.functions import col, min, to_date
from pyspark.sql.types import IntegerType, FloatType, StringType, DateType

def impute_missing_vals(df, table):
    print(f"\nHandling missing values for Table {table}:")

    # Replace missing (int and double) values to 0
    num_cols = [c_name for c_name, dtype in df.dtypes if dtype in ['int', 'double', 'long', 'decimal(22,0)']]
    for c_name in num_cols:
        df = df.fillna({c_name: 0})
    print(f"Replaced missing numbers with 0.")

    # Replace NULL string values with 'Unknown'
    obj_cols = [c_name for c_name, dtype in df.dtypes if dtype == 'string']
    for c_name in obj_cols:
        df = df.fillna({c_name: 'Unknown'})
    print(f"Replaced NULL columns with 'Unknown'.")

    # Fill datetime columns with the earliest available date
    date_cols = [c_name for c_name, dtype in df.dtypes if dtype == 'timestamp']
    for c_name in date_cols:
        earliest_date = df.select(min(col(c_name))).collect()[0][0]  # Get the earliest date
        if earliest_date:
            df = df.withColumn(c_name, when(col(c_name).isNull(), earliest_date).otherwise(col(c_name)))
            print(f"Replaced missing values in '{c_name}' with earliest date: {earliest_date}")

    return df

# Apply to all datasets
case_ids_df1 = impute_missing_vals(case_ids_df1, "caseids__df")
collisions_df1 = impute_missing_vals(collisions_df1, "collisions_df")
parties_df1 = impute_missing_vals(parties_df1, "parties_df")
victims_df1 = impute_missing_vals(victims_df1, "victims_df")
```

1. Missing numbers → 0
2. NULL strings → ‘Unknown’
3. Blank dates to the earliest datetime

Fixing Cols: Remove Duplicates

```
#Remove Duplicates

# Function to remove duplicates and print the count of duplicates removed
def remove_dupes(df, table):

    df_cleaned = df.dropDuplicates()
    print(f"Removed {df.count() - df_cleaned.count()} duplicate rrecords from {table}.")
    return df_cleaned

case_ids_df1 = remove_dupes(case_ids_df1, "case_df")
collisions_df1 = remove_dupes(collisions_df1, "collisions_df")
parties_df1 = remove_dupes(parties_df1, "parties_df")
victims_df1 = remove_dupes(victims_df1, "victims_df")
```

```
VBox()
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layout(height='2
Removed 35 duplicate rrecords from case_df.
Removed 0 duplicate rrecords from collisions_df.
Removed 0 duplicate rrecords from parties_df.
Removed 0 duplicate rrecords from victims_df.
```

35 duplicates were removed from case if DataFrame

Outlier Analysis

```python
from pyspark.sql import functions as fn

def remove_outliers_iqr(df, table, num_cols, ex_cols=None):
    print(f"\nRemoving outliers from {table}...")
    clean_df = df
    total_removed = 0

    for c in num_cols:
        # Calculate Q1 and Q3
        q1, q3 = clean_df.approxQuantile(c, [0.25, 0.75], 0.05)
        iqr = q3 - q1
        lower = q1 - 1.5 * iqr
        upper = q3 + 1.5 * iqr

        before_count = clean_df.count()
        clean_df = clean_df.filter((fn.col(c) >= lower) & (fn.col(c) <= upper))
        after_count = clean_df.count()

        removed = before_count - after_count
        total_removed += removed

        if removed > 0:
            print(f" - {removed} rows removed based on column '{c}'")

    if total_removed == 0:
        print(" - No outliers removed.")

    return clean_df

# Define columns to exclude from outlier removal
ex_case_id = ['case_id']
ex_collison = ['case_id', 'id']
ex_party = ['case_id', 'id', 'party_number']
ex_victim = ['case_id', 'id', 'party_number']
```

Function to remove outliers from the DataFrame

Relative Error threshold set to 5% for speed and accuracy

Remove records below 25 %tile and above 75 %tile

Retain key fields from outlier analysis as these may have extreme values

EDA: Finding Patterns

Challenge with Rendering images in PySpark

1. We cannot render images in a Jupyter notebook using PySpark with Livy on EMR.
2. Livy is a REST based service and does not support rick media like images or plots directly in its response.

Workaround to display the EDA graph plots

1. We can render images in a Jupyter notebook using Python 3 Kernel.
2. Save the plots from matplotlib an seaborn into a local Hadoop path '/tmp/<file_name>.png
3. Push the file to an S3 location using boto library. Print the S3 URL
4. Open another notebook with Python Kernel.
5. Download the above saved image from S3 location.
6. Use Display function from IPython to display in inline in the notebook.
7. Submit both the notebooks with PySpark and Python kernels

```python
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import boto3
# Univariate Analysis

# Collision Severity Distribution
import matplotlib
matplotlib.use('Agg')  # Use non-interactive backend

# Convert to Pandas
severity_cnt_df = collisions_df1.groupBy("collision_severity").count().orderBy("count", ascending=False)
severity_cnt_pd = severity_cnt_df.toPandas()

# Plot
plt.figure(figsize=(10, 6))
sns.barplot(
    data=severity_cnt_pd,
    x='collision_severity',
    y='count',
    hue='collision_severity',
    legend=False,  # Hide redundant legend
    palette='viridis'
)
plt.title("Collision Severity Distribution")
plt.xlabel("Collision Severity")
plt.ylabel("Count")
plt.xticks(rotation=45)
plt.tight_layout()
#plt.show()

# Save plot locally on EMR
local_path = "/tmp/collision_severity_dist.png"
plt.savefig(local_path)
print(f"✅ Plot saved locally at: {local_path}")

# Upload to S3
bucket_name = "trafficds71"
s3_key = "plots/collision_severity_dist.png"  # You can customize folder/key structure

# Upload
s3 = boto3.client("s3")
s3.upload_file(local_path, bucket_name, s3_key)

# Generate viewable URL (if public or accessible)
s3_url = f"https://{bucket_name}.s3.amazonaws.com/{s3_key}"
print(f"📊 Plot uploaded to: {s3_url}")
```

Non-Interactive matplotlib

Aggregate data in DataFrame and convert to Panda

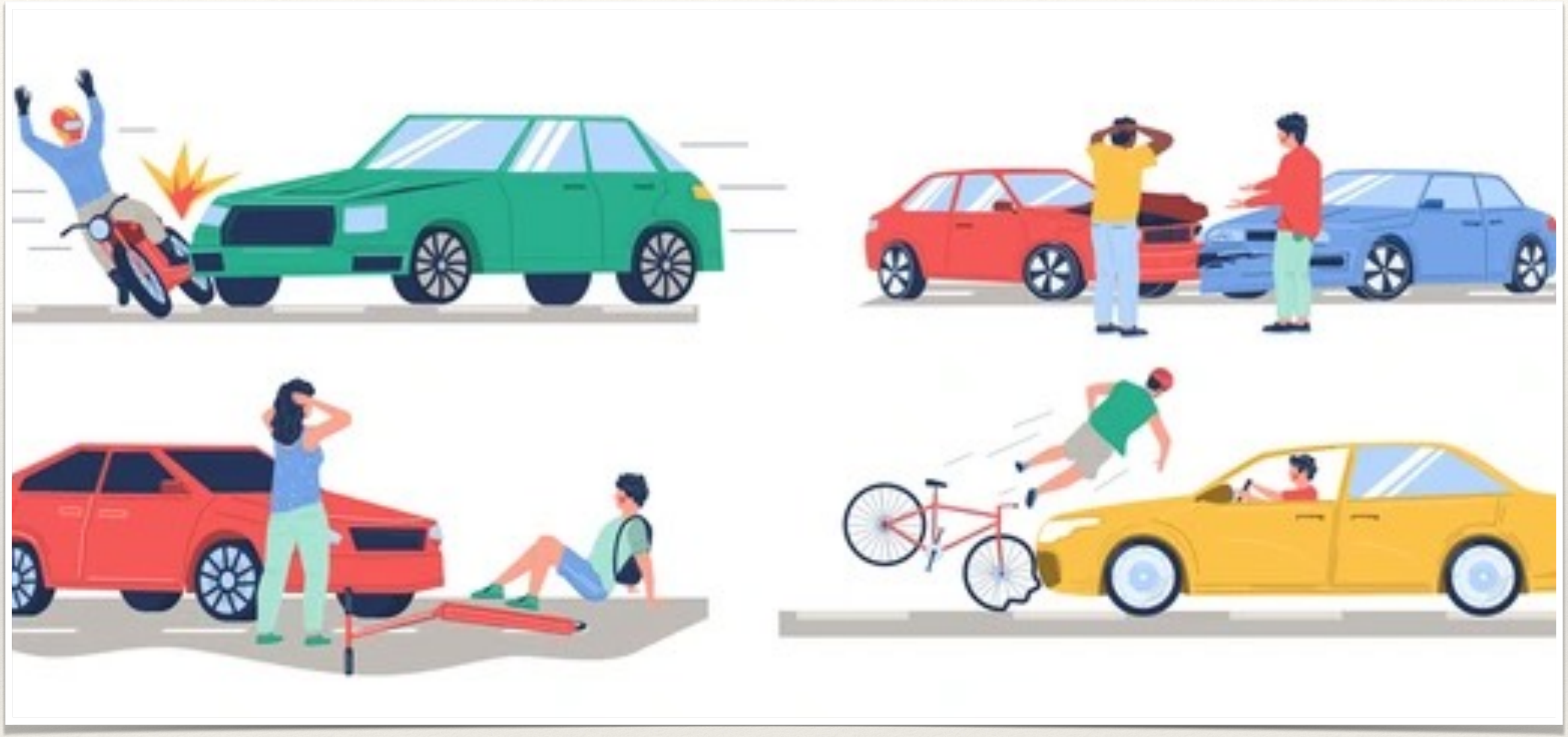
Save the image to hadoop

Upload to S3

```python
import boto3
from IPython.display import Image, display

# S3 details
bucket_name = 'trafficds71'
object_key = 'plots/weather_conditions.png'
local_path = '/tmp/weather_conditions.png'

# Download image from S3
s3 = boto3.client('s3')
s3.download_file(bucket_name, object_key, local_path)

# Display the image
display(Image(filename=local_path))
```

Download the image from S3 using boto3

Render the graph plot image using IPython



Univariate Analysis

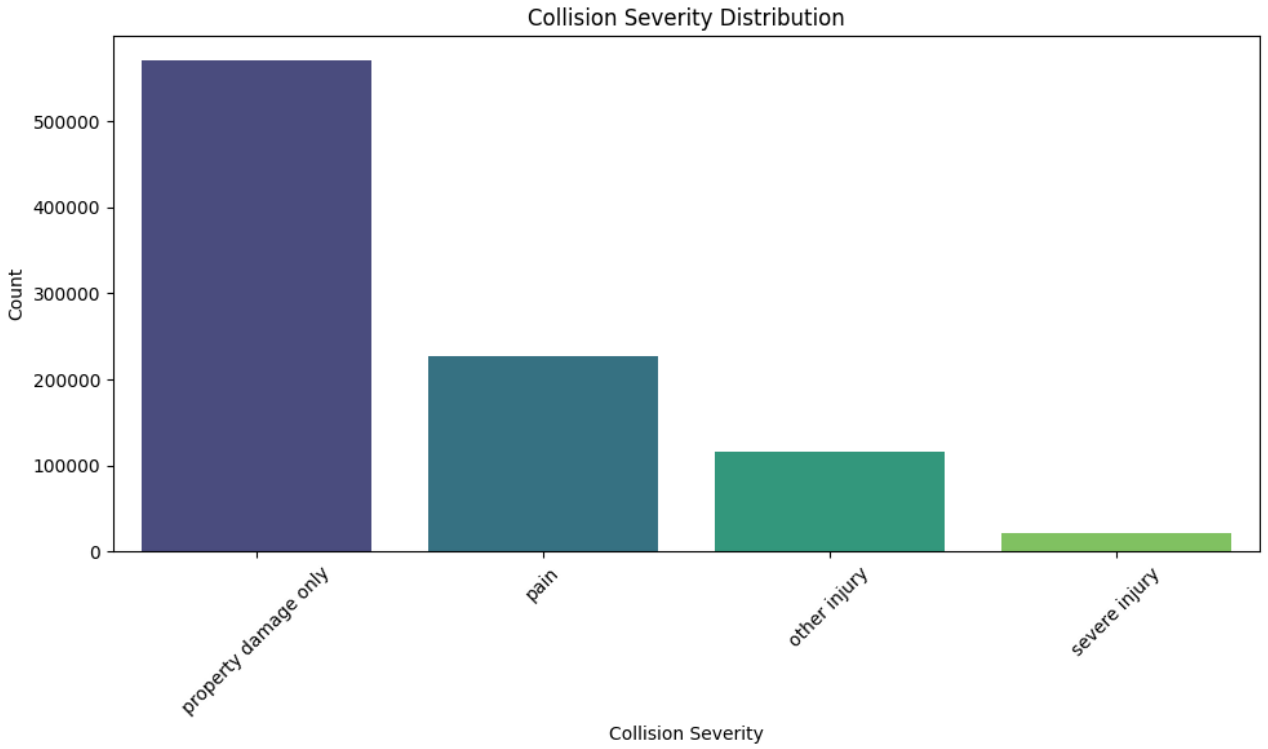
Collision Severity Distribution

X-axis represents severity types.

Y-axis represents their respective count.

“Property damage only” has the highest count with over 500,000 incidents.

Second highest count is “pain” with over 200,000 incidents.



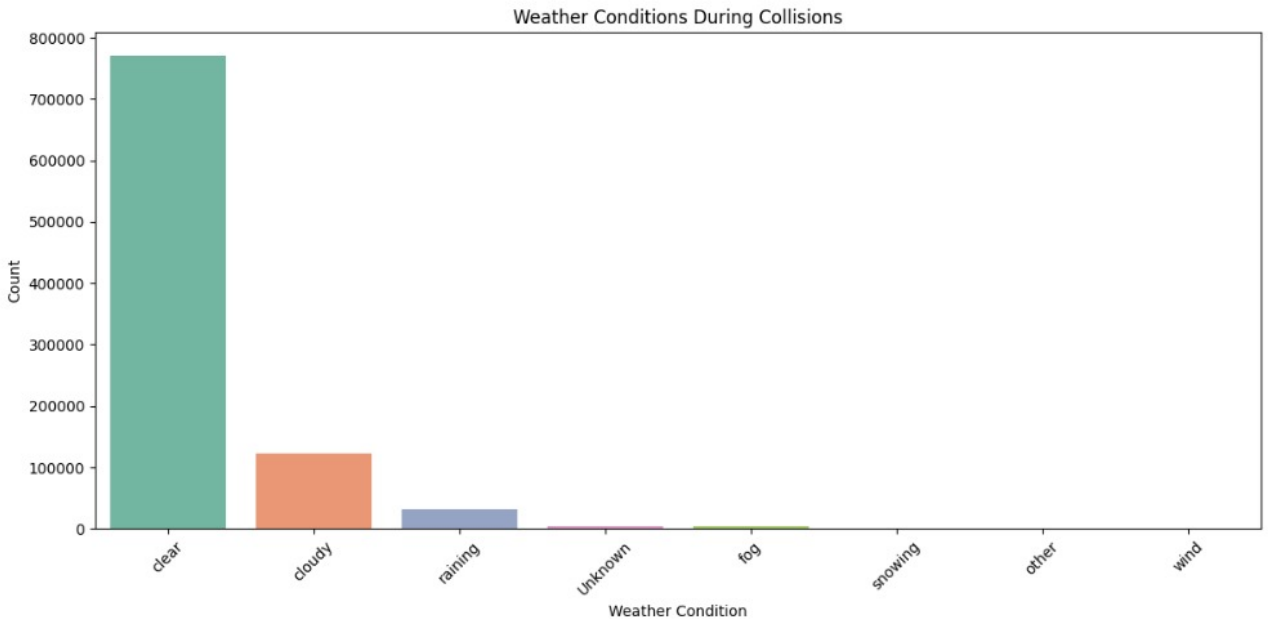
Weather condition during collision

X-axis represents “Weather Conditions”

Y-axis represents “Count”.

Most days have “clear” weather condition and so the count of collisions is also the highest during “clear” weather conditions.

We see high collision rates in “cloudy” and “raining” conditions amongst other weather conditions.



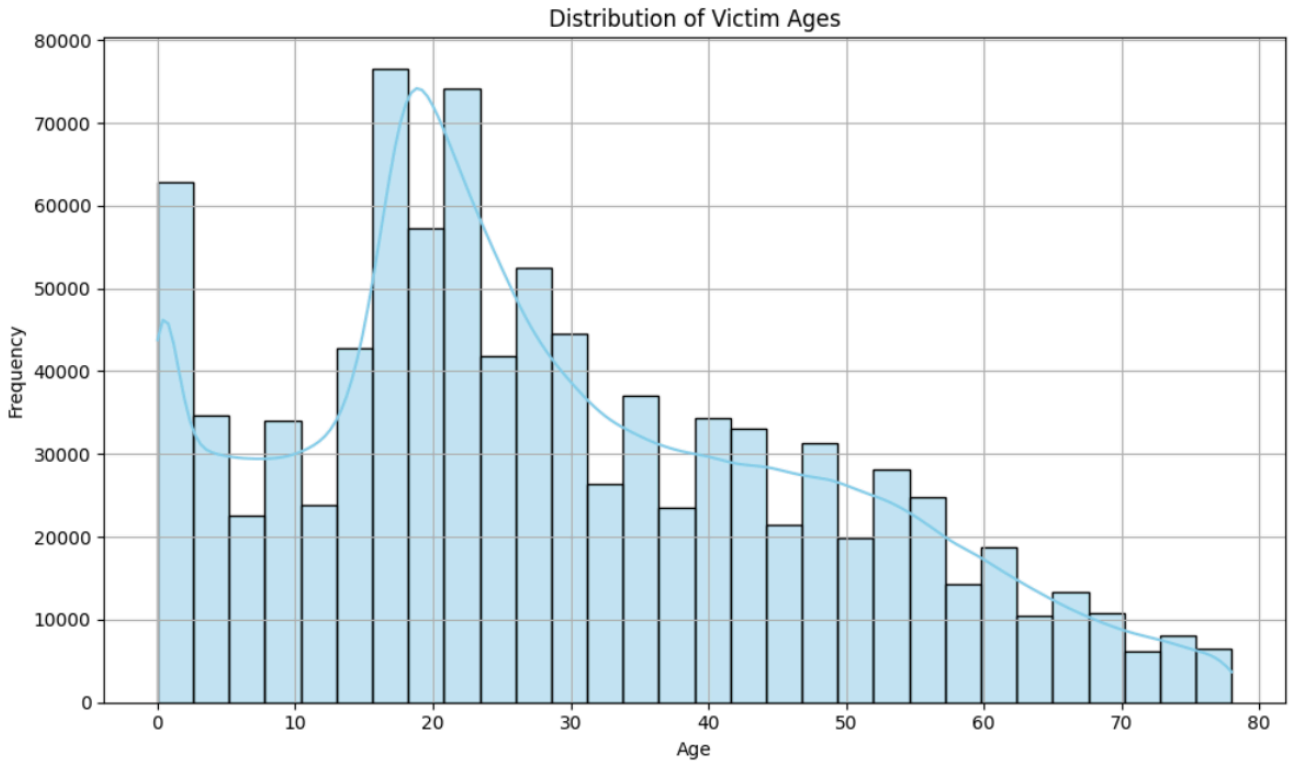
Distribution of victim ages

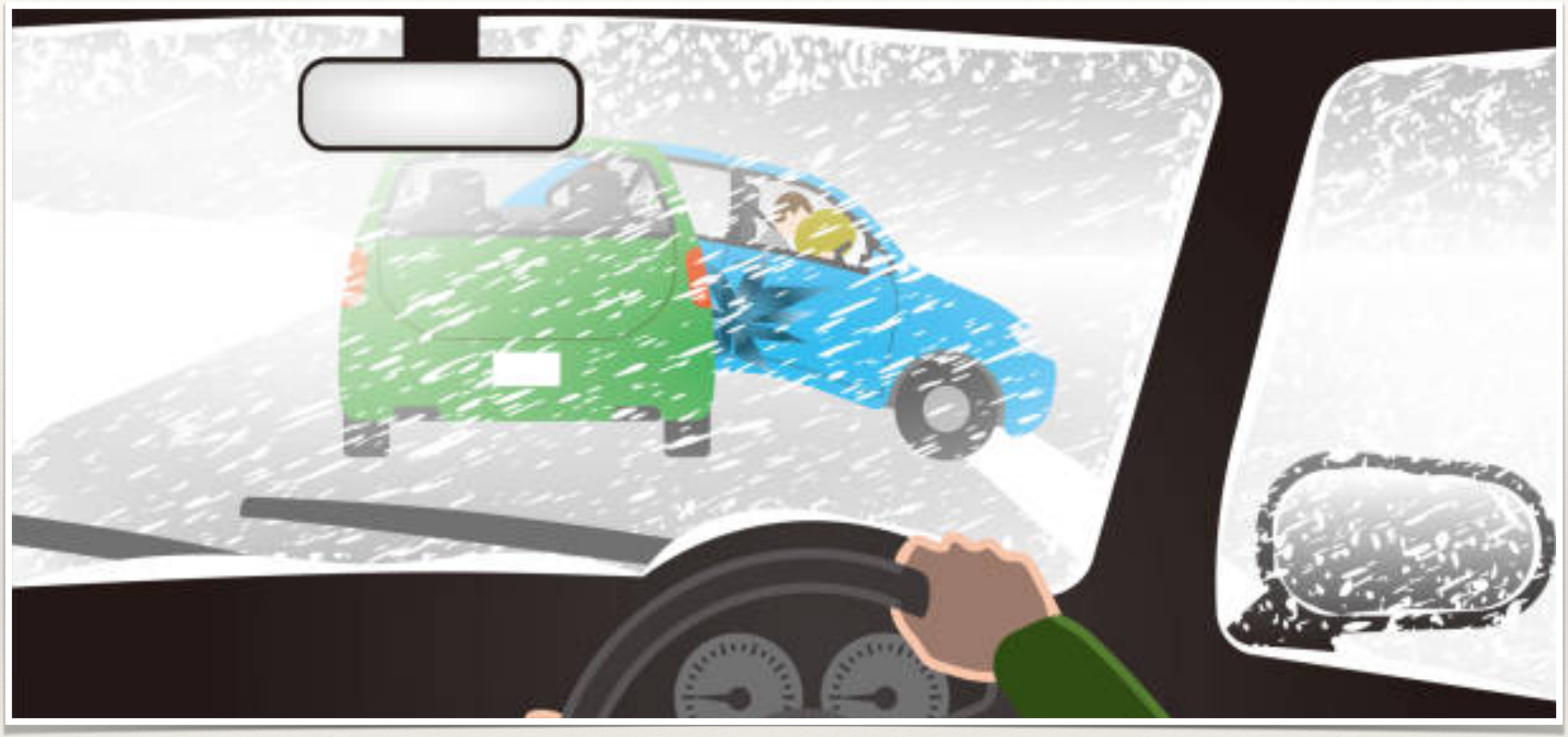
X-axis represents victim age.

Y-axis represents the frequency.

We observe that the frequency of collision is highest for victims between the age 16 and 24 (above 70,000).

We also observe that there is a high frequency of victims between the ages 0 and 4 (above 60,000).





Bivariate and Multivariate Analysis

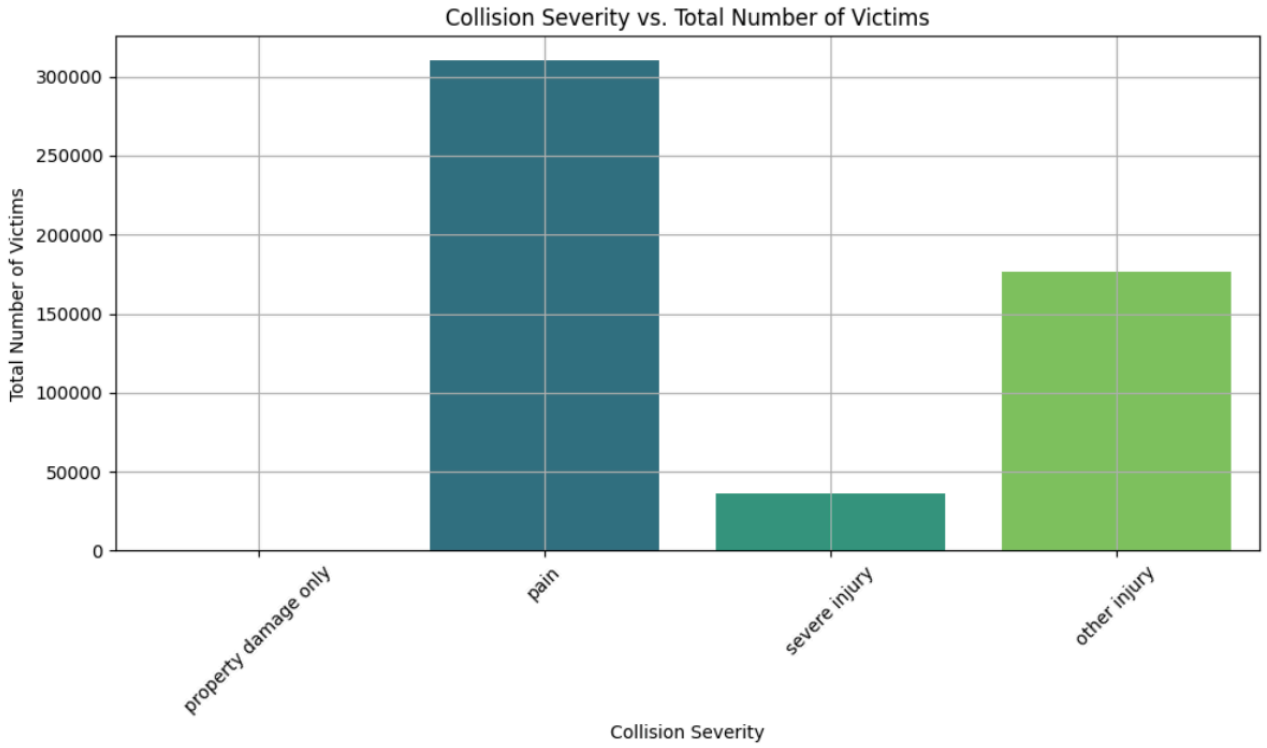
Collision Severity vs Number of Victims

X-axis represents the Collision Severity.

Y-axis represents the Total Number of Victims.

Most victims suffered from complaint-of-pain injuries (above 300,000).

This is followed by visible injuries.



Weather condition vs collision severity

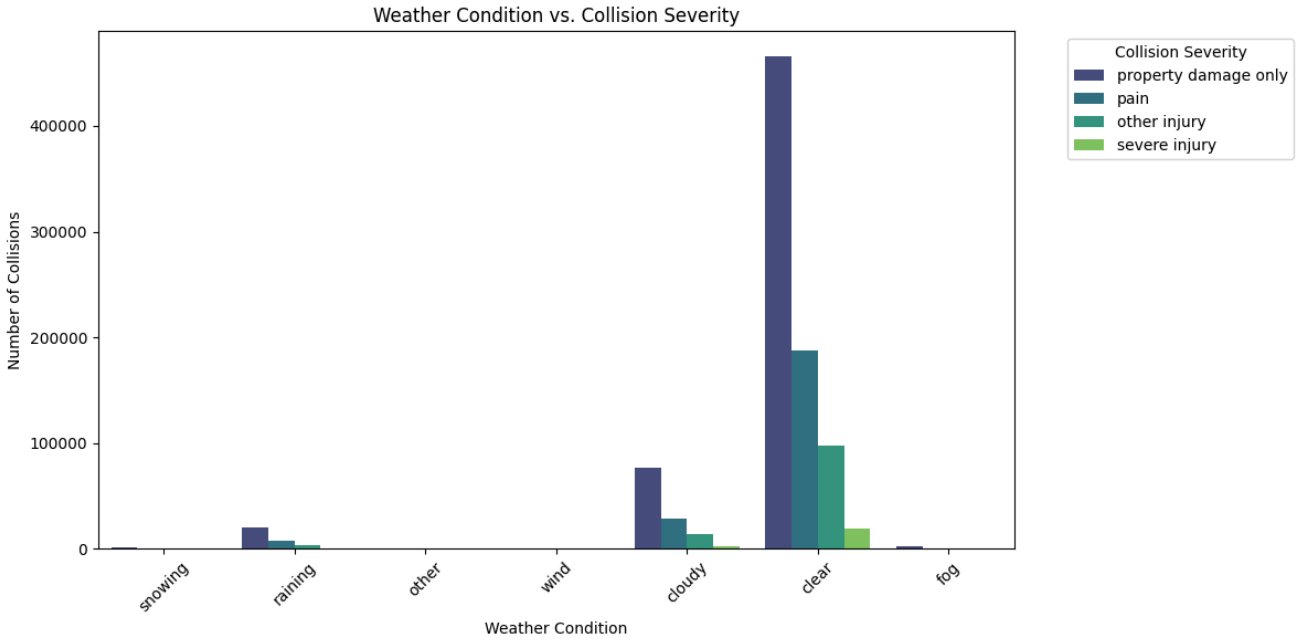
X-axis represents weather conditions.

Y-axis represents number of collisions.

We observe that during most collisions, severity wise, the highest severity is “property damage only”, followed by “pain” being the second highest category.

Most collisions occur during “clear” weather conditions as the amount of traffic would also be high during “clear weather conditions”.

There’s a high rate of collision possibility during “cloudy” and “raining” weather condition.



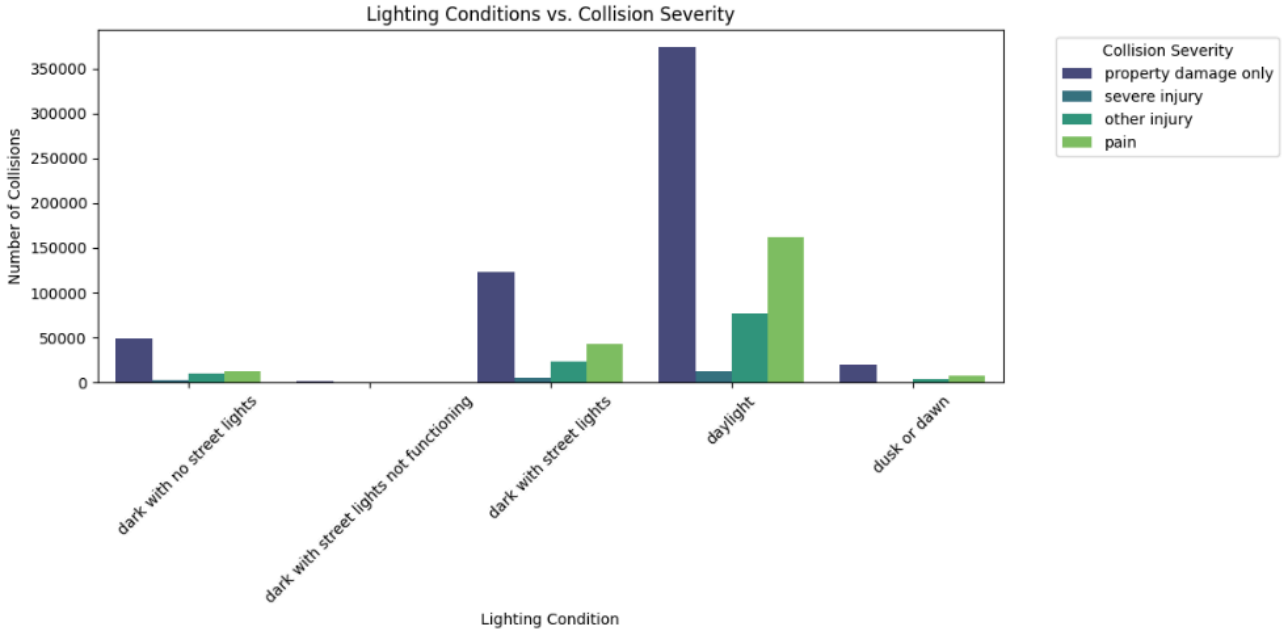
Lighting Condition vs Collision Severity

X-axis represents lighting conditions.

Y-axis represents the number of collisions.

Most collisions occur during “daylight” lighting condition as the number of vehicles are the highest at this time.

Collisions during daylight results in maximum cases of “property damage only”.



Hourly trend of collision

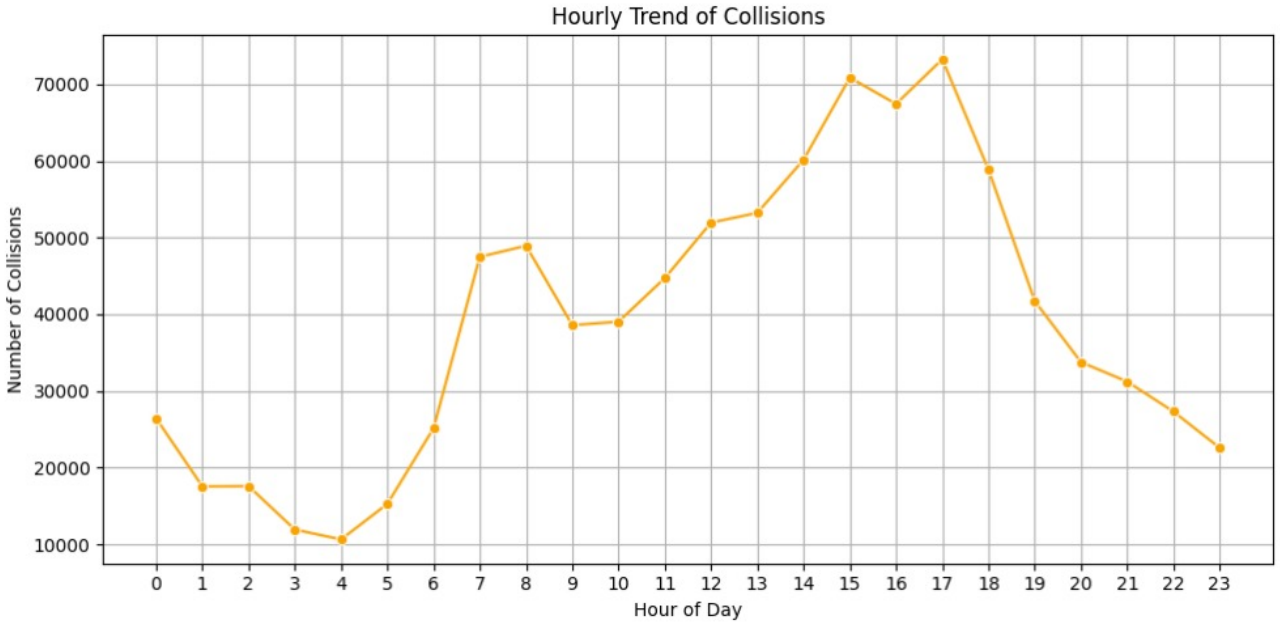
X-axis represents the Hour of the day.

Y-axis represents the Number of collisions.

We observe that the highest frequency of collisions is between the hours 15:00 and 17:00. The most possible reason for this could be because traffic is highest during this time as this falls within most office hours (above 70,000).

We observe above 26,000 frequency of collision during midnight which then reduces to approximately 10,000 by 04:00 but then increases steeply till 08:00 which is the start of most working hours.

This keeps increasing till 17:00 and then reduces.



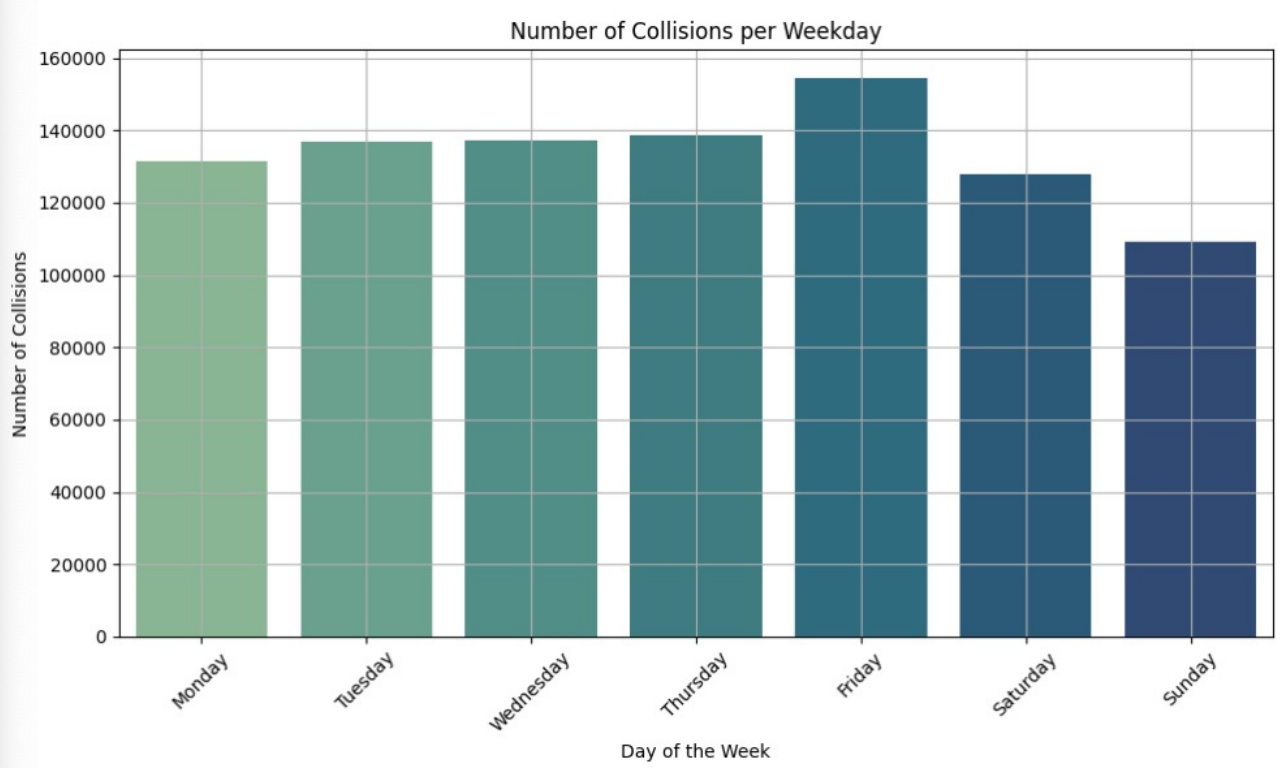
Daily Trend of Collision

X-axis represents “Day of the week”.

Y-axis represents “Number of collisions”.

We observe that the number of collisions is the highest on “Friday” approximately 150,000.

We observe that the number of collisions is comparatively lower on “Sunday”. Possible reason being weekend and so there’s less rush in traffic.



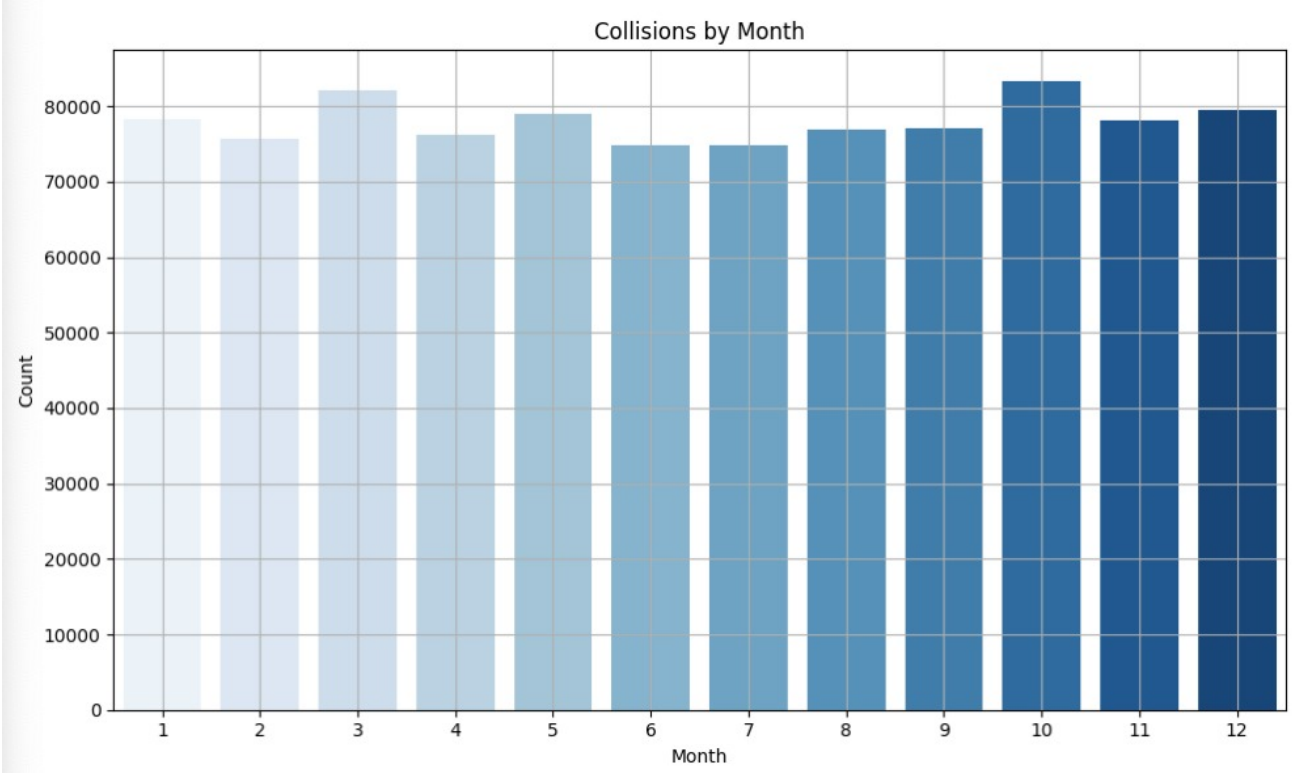
Monthly trend of collision

X-axis represents the Month.

Y-axis represents Count of collisions.

We Observe that collision rate is highest during the 3rd, 10th and 12th Month of the year.

Possible reason being that these months have festive celebrations and holidays leading to higher traffic and hence a higher rate of collision(Easter, Halloween, Thanksgiving and Christmas).



Yearly trend of collision

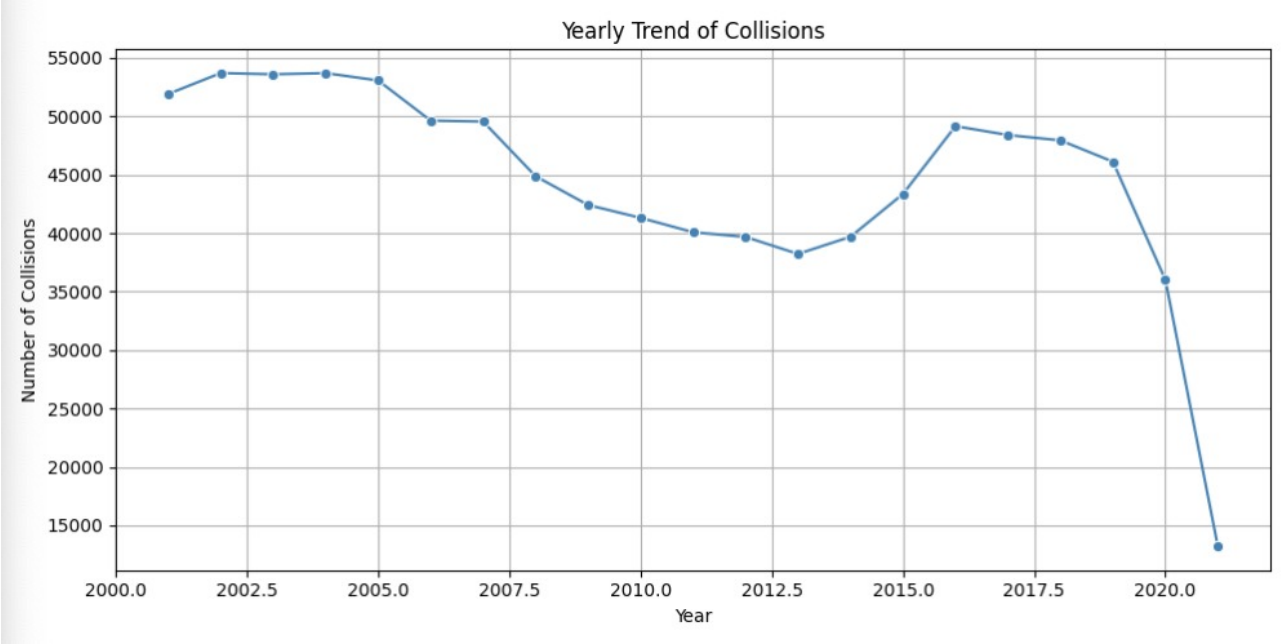
X-axis represents the Year.

Y-axis represents the Number of collisions.

We observe a gradual decrease in the number of collision between the year 2000 and 2013.

We then see a steep rise in the number of collision between 2013 and 2016 followed by a gradual decrease till 2019.

Post 2019 we see a steep fall in the number of collisions. A major reason for which could be due to the pandemic conditions because of Covid-19.



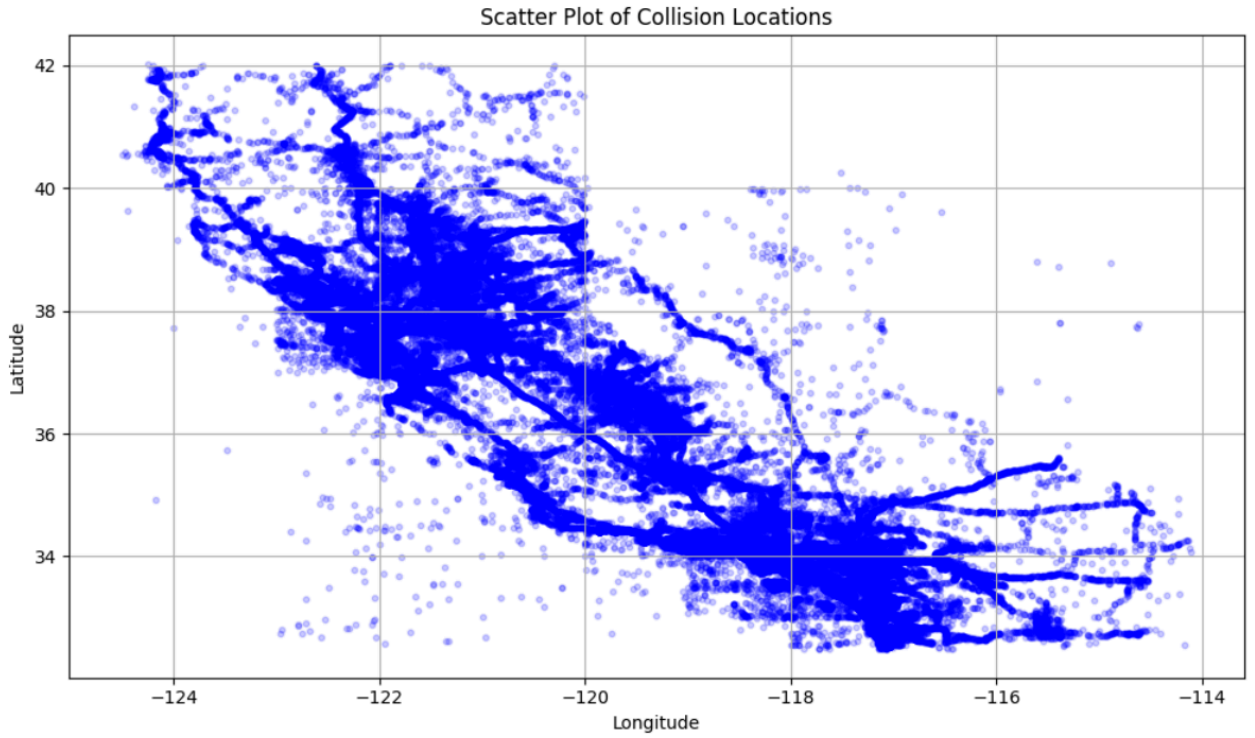
Collision location observation

This is a scatter plot with X-axis representing the Longitude and Y-axis representing the Latitude.

There is a high concentration of collisions between latitudes 33–39 and longitudes -122 to -118, aligning with:

Los Angeles, San Francisco Bay Area, Central Valley corridor (e.g., Highway 99, I-5). These areas are known for dense population and heavy traffic, which often correlates with higher collision rates.

Sparse Areas: Far east (longitude > -116) and far north (latitude > 41) have much sparser points, indicating: Fewer roads/highways, Lower population density (e.g., deserts or rural mountain regions).



Data Querying

Save the cleaned DataFrames to csv files in S3 and SQL query

1. Coalesce to 1 file before writing the file into S3
2. Save the file with header
3. The output is saved as part files.
4. Loop through each file rename to victims/collisions/parties/case_ids files
5. Delete the part file
6. Write the SQL to query the data in CSV and generate output

```
victims_df1.coalesce(1) \
  .write \
  .option("header", "true") \
  .mode("overwrite") \
  .csv("s3://trafficds71/temp_output/")
```

Coalesce into 1 single file

S3 temp location

```
VBox()
FloatProgress(value=0.0, bar_style='info', description='Progress:'
```

```
import boto3

s3 = boto3.client('s3')
bucket = 'trafficds71'
prefix = 'temp_output/'

# List objects to find the actual CSV file
response = s3.list_objects_v2(Bucket=bucket, Prefix=prefix)

for obj in response.get('Contents', []):
    key = obj['Key']
    if key.endswith(".csv"):
        print(f"Renaming {key} to output/sample_victims.csv")

        # Copy with new name
        s3.copy_object(
            Bucket=bucket,
            CopySource={'Bucket': bucket, 'Key': key},
            Key='output/sample_victims.csv'
        )

        # Optionally delete original part file
        s3.delete_object(Bucket=bucket, Key=key)
        break
```

S3 output location

Clean the part file

Top 5 Countries

Q: Identify the top 5 counties with the highest number of collisions.

```
# Query: Identify the top 5 counties with the most collisions
from pyspark.sql.functions import col, desc, month, count

top5_counties = collisions_df1.groupBy("county_location") \
    .count() \
    .orderBy(desc("count")) \
    .limit(5)

top5_counties.show()
```

```
VBox()
FloatProgress(value=0.0, bar_style='info', description='Progress:
+---------------+------+
|county_location| count|
+---------------+------+
|    los angeles|284100|
|         orange| 72042|
| san bernardino| 56737|
|      san diego| 53105|
|      riverside| 48686|
+---------------+------+
```

Month with Highest Collisions

Q. Identify the month with the highest number of collisions.

```
# Query: Find the month with the highest number of collisions
# Extract month with highest collisions
month_with_high_collisions = collisions_df1 \
    .withColumn("collision_month", month("collision_date")) \
    .groupBy("collision_month") \
    .agg(count("*").alias("collision_count")) \
    .orderBy(desc("collision_count")) \
    .limit(1)

month_with_high_collisions.show()
```

```
VBox()
FloatProgress(value=0.0, bar_style='info', description='Progress:
+---------------+---------------+
|collision_month|collision_count|
+---------------+---------------+
|             10|          83274|
+---------------+---------------+
```

October has the highest number of collisions.

Weather Conditions with Highest Collisions

Q. Determine the most common weather condition during collisions.

```
1 # Query: Find the most common weather condition during collisions
2
3 weather_with_most_col = collisions_df1 \
4     .groupBy("weather_1") \
5     .agg(count("*").alias("collision_count")) \
6     .orderBy(desc("collision_count")) \
7     .limit(1)
8
9 weather_with_most_col .show()
```

```
VBox()
FloatProgress(value=0.0, bar_style='info', description='Progress:',
+---------+---------------+
|weather_1|collision_count|
+---------+---------------+
|    clear|         769927|
+---------+---------------+
```

Clear weather has the highest collision.

Fatal Collisions

Q. Calculate the percentage of collisions that resulted in fatalities.

```
# Query: Determine the percentage of collisions that resulted in fatalities
from pyspark.sql.functions import col, count, when
# Total number of collisions
total_col = collisions_df1.count()

# Count of Collisions with fatalities
fatal_col = collisions_df1.filter(col("killed_victims") > 0).count()

# percentage Calculation
fatal_percent = (fatal_col / total_col) * 100

print(f"Percentage of collisions with fatalities: {fatal_percent:.2f}%")
```

```
VBox()
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layou
Percentage of collisions with fatalities: 0.00%
```

Dangerous Time for Collisions

Q. Find the most dangerous time of day for collisions.

```
# Query: Find the most dangerous time of day for collisions
from pyspark.sql.functions import hour, count, desc

# Extract hour and rsepective count collisions
dangerous_hour = collisions_df1 \
    .withColumn("collision_hour", hour("collision_time")) \
    .groupBy("collision_hour") \
    .agg(count("*").alias("collision_count")) \
    .orderBy(desc("collision_count")) \
    .limit(1)

dangerous_hour.show()
```

```
VBox()
FloatProgress(value=0.0, bar_style='info', description='Progress
+-------------+---------------+
|collision_hour|collision_count|
+-------------+---------------+
|           17|          73255|
+-------------+---------------+
```

Highest collisions observed at 5:00 PM

Road Surface Conditions

Q. Identify the top 5 road surface conditions with the highest collision frequency.

```
# Query: List the top 5 road types with the highest collision frequency
top5_road_types = collisions_df1 \
    .groupBy("road_surface") \
    .agg(count("*").alias("collision_count")) \
    .orderBy(desc("collision_count")) \
    .limit(5)

top5_road_types.show()
```

```
VBox()
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout
+------------+---------------+
|road_surface|collision_count|
+------------+---------------+
|         dry|         845637|
|         wet|          76833|
|     Unknown|           8141|
|       snowy|           4122|
|    slippery|           1048|
+------------+---------------+
```

Lighting Conditions

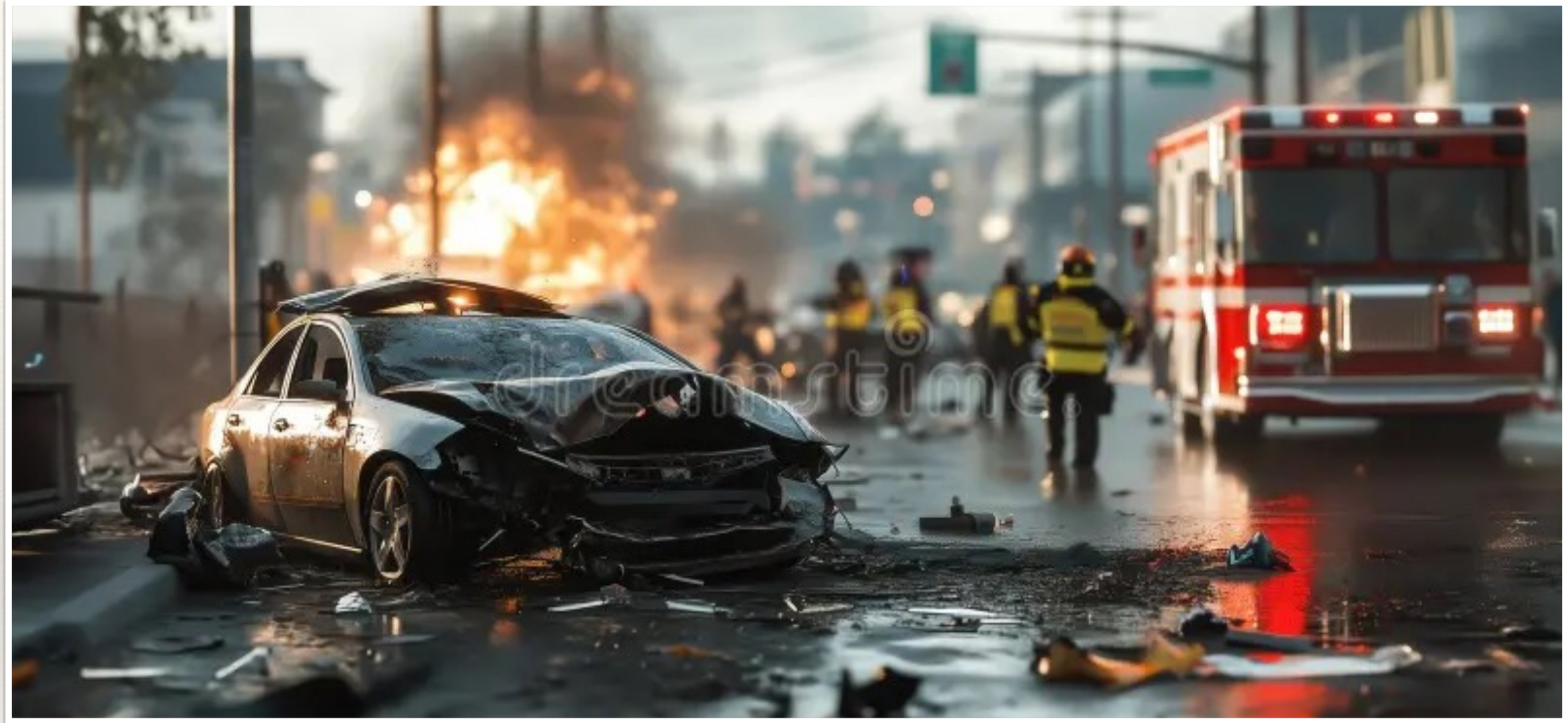
Q. Analyze lighting conditions that contribute to the highest number of collisions.

```
# Query: Find the top 3 lighting conditions that lead to the most collisions

top3_lighting_conditions = collisions_df1 \
    .groupBy("lighting") \
    .agg(count("*").alias("collision_count")) \
    .orderBy(desc("collision_count")) \
    .limit(3)

top3_lighting_conditions.show()
```

```
VBox()
FloatProgress(value=0.0, bar_style='info', description='Progress:', layout=Layo
+--------------------+---------------+
|            lighting|collision_count|
+--------------------+---------------+
|            daylight|         625574|
|dark with street ...|         195867|
|dark with no stre...|          74634|
+--------------------+---------------+
```

Conclusion

- This project effectively leveraged Apache Spark and AWS S3 to process and analyse large-scale California traffic collision data.
- Through detailed ETL operations and rich visual analysis, we uncovered crucial patterns in accident severity, timing, locations, and contributing factors.
- These insights can help traffic authorities, city planners, and policymakers make improvements.



Key Insights and Recommendations

Key Insights from Collision Data

- **Collision Severity:**
  - Property damage only is the most common, with over 500,000 cases.
  - Pain-related injuries are the second highest, over 200,000 cases.

- **Weather Conditions**
  - Most collisions occurred in clear weather, likely due to high traffic.
  - Cloudy and raining conditions also show significant collision counts.

- **Weather vs Severity**
  - Majority of severe collisions happen during clear weather.
  - Cloudy and rainy conditions also contribute to higher severity.

Key Insights from Collision Data

- **Victim Age Distribution:**
  - Highest collision involvement is in the 16–24 age group.
  - Notably high involvement also seen in 0–4 age group.

- **Lighting Conditions:**
  - Most collisions occur during daylight.
  - Dark with street lights is the second most common lighting condition.

- **Hourly Collision Trend:**
  - Peak hours: 15:00–17:00, likely due to office traffic.
  - Collisions drop post-midnight and rise sharply from 08:00 onward.

Key Insights from Collision Data

- **Daily Trend:**
  - Friday sees the most collisions; Sunday the least.
- **Monthly Trend:**
  - Highest collisions in March, October, and December, likely due to holidays and festive traffic.
- **Yearly Trend:**
  - Decrease from 2000–2013, spike till 2016, then a drop.
  - Post-2019 decline aligns with COVID-19 pandemic and reduced mobility.
- **Geographical Trends:**
  - High collision density in Los Angeles, San Francisco Bay Area, and Central Valley.
  - Sparse data in eastern deserts and northern rural regions due to low population density.

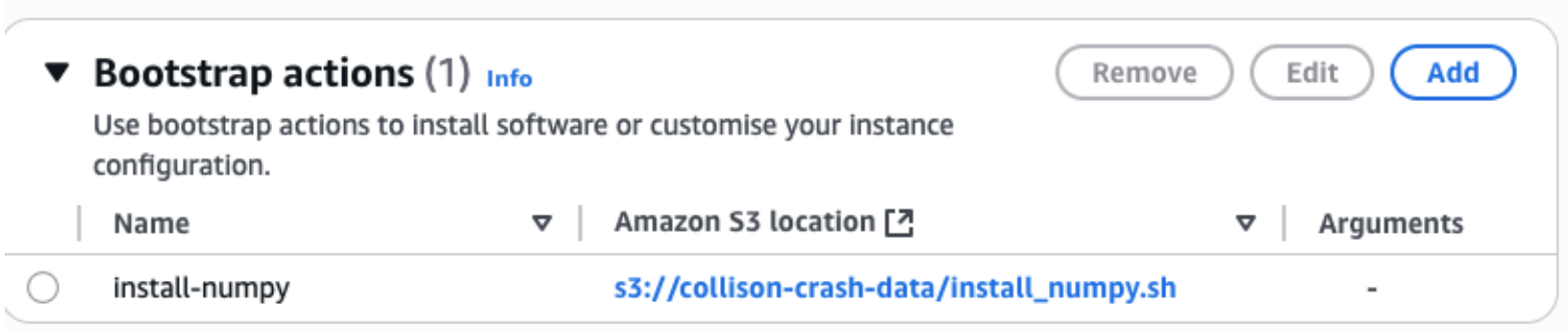
Recommendations

- **Focus on High-Risk Areas & Time Slots.**
  - Improve infrastructure and traffic control in regions with high collision density (e.g., Los Angeles, Bay Area).
  - Increase monitoring and enforcement during peak hours (15:00–17:00) and high-risk days (Fridays).

- **Implement Targeted Safety Campaigns.**
  - Launch awareness programs for vulnerable age groups (16–24 and 0–4 years).
  - Enforce child safety seat usage and promote defensive driving education.

- **Leverage Technology & Seasonal Planning.**
  - Use predictive analytics and smart traffic systems for weather-based and real-time alerts.
  - Plan special safety measures during festive months (March, October, December) when collisions spike.

Appendix : AWS Environment Setup



Bootstrap shell script to install below.

Required for EDA and generating csv files

- numpy
- matplotlib
- seaborn
- boto3
- Ipython
- Pillow

```
#!/bin/bash

# Install pip for Python 3 (Amazon Linux 2023 compatible)
sudo dnf install -y python3-pip

# Upgrade pip and install numpy
sudo python3 -m pip install --upgrade pip
sudo python3 -m pip install numpy
sudo python3 -m pip install Pillow
sudo python3 -m pip install --quiet matplotlib seaborn
sudo python3 -m pip install boto3
sudo python3 -m pip install IPython
```