Task 7 - Traffic Accident Analysis

Description: City collects traffic accident data including location (lat,lon), vehicle_type, cause, severity (numeric or categorical), timestamp, traffic_volume, vehicle_speed, traffic_density, and citizen reports (text). Authorities want to identify accident hotspots and improve road safety.

DATASET:

	A	B	C	D	E	F	G	H	I	J	K	L
1	lat	lon	vehicle_typ	cause	severity	timestamp	traffic_volu	vehicle_sp	traffic_den	zone	intersectio	report_text
2	17.40618	78.42777	Truck	Distracted	3	06-09-2024 14:09	575	75.38	0.76	Central	Junction_1	Road construction without sign
3	17.49261	78.48129	Bus	Speeding	3	15-06-2024 11:18	182	71.3	0.86	Central	Junction_4	Pedestrian crossing issue
4	17.4598	78.53094	Truck	Bad Weath	3	31-07-2024 09:28	209	48.14	0.59	North	Junction_7	Road construction without sign
5	17.4398	78.50983	Pedestrian	Mechanica	2	11-06-2024 10:45	289	102.22	0.18	South	Junction_1	Poor lighting
6	17.3734	78.52098	Bike	Mechanica	1	10-06-2024 18:19	899	95.28	0.31	West	Junction_1	Road construction without sign
7	17.3734	78.49882	Pedestrian	Mechanica	1	01-03-2024 08:24	350	93.29	0.25	West	Junction_8	Over speeding vehicles
8	17.35871	78.50384	Bike	Drunk Drivi	4	24-02-2024 00:44	823	107.62	0.21	South	Junction_2	Road construction without sign
9	17.47993	78.52738	Car	Speeding	4	06-10-2024 07:42	390	92.28	0.43	East	Junction_1	Over speeding vehicles
10	17.44017	78.43745	Auto	Bad Weath	2	11-07-2024 23:19	744	43.18	0.31	North	Junction_1	Pedestrian crossing issue
11	17.45621	78.47341	Pedestrian	Drunk Drivi	5	08-03-2024 17:51	300	90.56	0.3	South	Junction_1	Pedestrian crossing issue
12	17.35309	78.43318	Truck	Signal Jum	1	25-10-2024 01:57	896	50.77	0.15	West	Junction_2	Pedestrian crossing issue
13	17.49549	78.54815	Car	Bad Weath	4	22-10-2024 20:01	425	44.22	0.12	Central	Junction_1	Road construction without sign
14	17.47487	78.54161	Auto	Distracted	1	28-02-2024 17:11	610	27.3	0.19	West	Junction_1	Pedestrian crossing issue
15	17.38185	78.40591	Truck	Bad Weath	2	24-04-2024 08:31	508	94.4	0.25	Central	Junction_1	Road construction without sign
16	17.37727	78.50584	Bus	Bad Weath	2	15-02-2024 10:54	706	91.1	0.49	North	Junction_1	Road construction without sign
17	17.37751	78.53879	Truck	Mechanica	5	01-05-2024 04:14	889	43.45	0.56	Central	Junction_9	Signal not working
18	17.39564	78.42709	Bike	Bad Weath	5	06-09-2024 13:47	936	94.94	0.23	South	Junction_3	Signal not working
19	17.42871	78.48519	Bike	Distracted	2	09-09-2024 02:37	607	111.78	0.48	Central	Junction_1	Road construction without sign
20	17.41479	78.53732	Bus	Distracted	4	28-01-2024 19:56	830	30.8	0.41	East	Junction_9	Over speeding vehicles
21	17.39368	78.40509	Car	Speeding	1	04-08-2024 13:13	700	43.26	0.27	South	Junction_1	Poor lighting
22	17.44178	78.50461	Auto	Bad Weath	3	10-04-2024 17:13	171	95.03	0.11	North	Junction_1	Pothole issue
23	17.37092	78.4446	Truck	Mechanica	5	20-06-2024 02:12	346	47.45	0.52	South	Junction_1	Signal not working
24	17.39382	78.53866	Bike	Bad Weath	3	10-08-2024 01:06	740	82.47	0.72	East	Junction_1	Road construction without sign

Questions:

1) How color schemes can indicate accident severity

Visualization guidance / explanation:

Use a sequential → diverging → categorical strategy depending on data:

If severity is ordinal/numeric (e.g., 0–5): use a sequential palette from light (low severity) → dark (high severity) (e.g., YlOrRd or Turbo/viridis), so intensity maps to danger.
If you want to emphasize a critical threshold (e.g., fatal vs non-fatal): use a diverging palette with neutral center and strong warning color for the high-severity side (e.g., gray → yellow → red).

For multiple nominal severity classes (minor, major, fatal): use distinct categorical colors but keep high-severity color perceptually dominant (e.g., grey, orange, deep red).

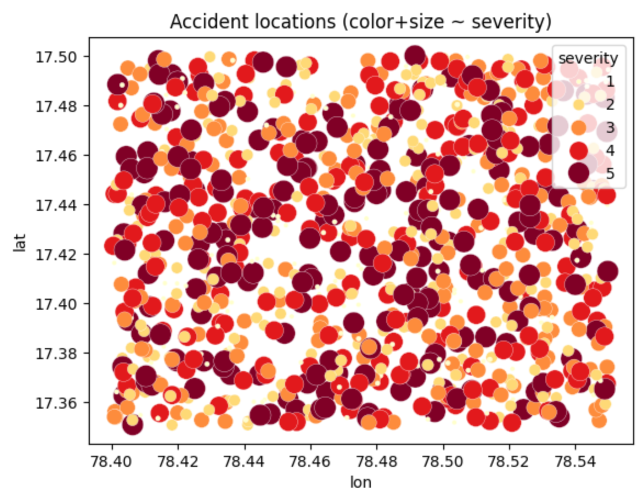
Use consistent color semantics across maps, charts, legends, and interactive tooltips.Include accessibility: colorblind-safe palettes and redundant encodings (size or icon) for critical # severity: 0-5 numeric

code:

```
df = pd.read_csv('/content/Traffic_Accident_Analysis_Dataset.csv')
```

```
import seaborn as sns
import matplotlib.pyplot as plt

# severity: 0-5 numeric
sns.scatterplot(x='lon', y='lat', hue='severity', palette='YlOrRd', size='severity', sizes=(10,200), data=df)
plt.title('Accident locations (color+size ~ severity)')
```

Visuvalization:



Inference: Color intensity + size quickly directs attention to the most dangerous locations and reduces cognitive load when scanning maps/dashboards.

2) Visualization pipeline from raw accident data to dashboards

Pipeline steps (end-to-end):

Data ingestion

→ Validation → Cleaning

→ Enrichment → Feature engineering

→ Spatial aggregation → Analytics

→ Visualization generation → Dashboard assembly

→ Delivery.

Inference: A reproducible ETL + enrichment pipeline ensures dashboards are trustworthy and supports model training.

3) Apply Gestalt principles to quickly identify high-risk zones

Proximity, Similarity, Continuity, Figure-Ground, Enclosure, Common Fate principles guide visualization.

- **Proximity:** group nearby accident points into clusters.
- **Similarity:** use consistent color/shape for same severity.
- **Continuity:** connect accidents along major roads to show continuous high-risk corridors.
- **Figure-Ground:** highlight hotspots by bright color; fade background map.
- **Enclosure:** draw cluster boundaries or district polygons.
- **Common Fate:** animate accidents over time to show temporal motion.

Combine color, size, opacity cues for fast recognition.

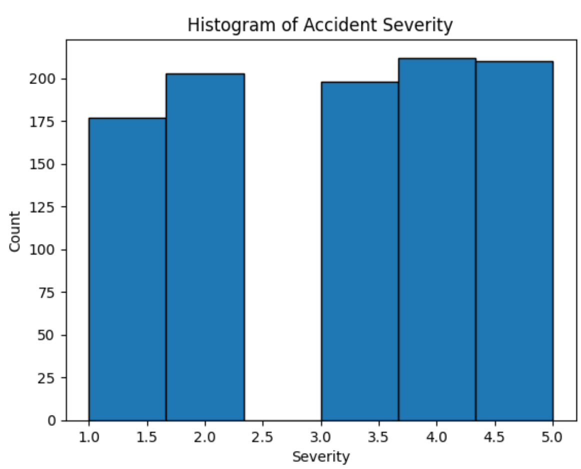
4) Univariate analysis

a. Histogram of accident severity

CODE:

```
plt.hist(df['severity'], bins=6, edgecolor='black')
plt.title('Histogram of Accident Severity')
plt.xlabel('Severity')
plt.ylabel('Count')
```

Visuvalization:



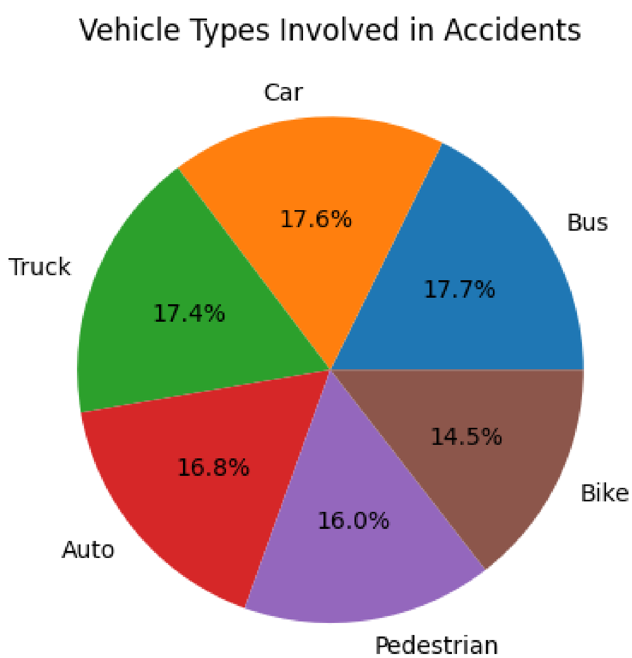
Inference: Shows distribution of severity levels.

b. Pie chart of vehicle types involved

CODE:

```
plt.pie(df['vehicle_type'].value_counts(), labels=df['vehicle_type'].value_counts().index, autopct='%1.1f%%')
plt.title('Vehicle Types Involved in Accidents')
```

Visuvalization:



Inference: Identifies dominant vehicle categories (e.g., motorcycles, cars).

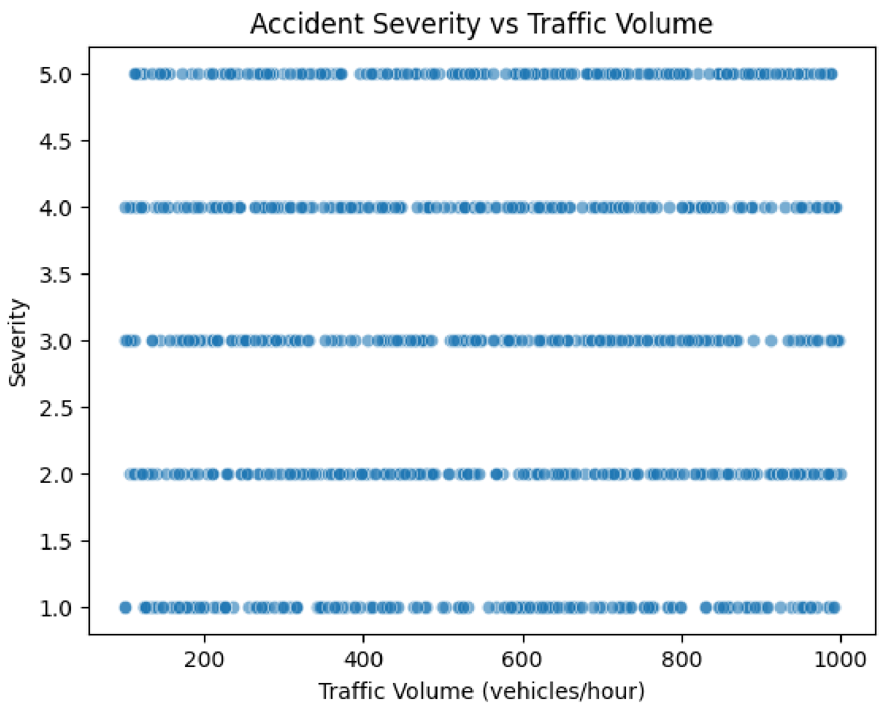
5) Bivariate analysis

a. Scatterplot of accidents vs. traffic volume

CODE:

```
sns.scatterplot(x='traffic_volume', y='severity', data=df, alpha=0.6)

plt.title('Accident Severity vs Traffic Volume')

plt.xlabel('Traffic Volume (vehicles/hour)')

plt.ylabel('Severity')
```

Visuvalization:



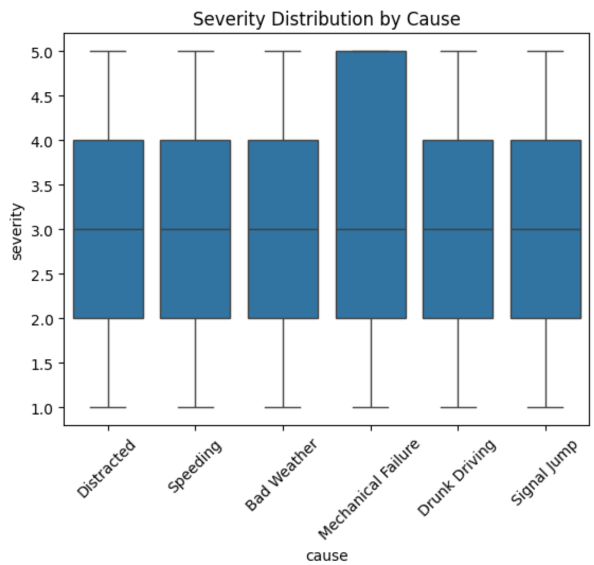
Inference: Checks whether severity correlates with volume; may reveal thresholds where severity spikes.

b. Box plot of severity by cause

CODE:

```
sns.boxplot(x='cause', y='severity', data=df)
plt.title('Severity Distribution by Cause')
plt.xticks(rotation=45)
```

Visuvalization:



Inference: Shows which causes (speeding, poor visibility, intoxication) have larger median severity and spread — useful for targeted enforcement.

6) Multivariate analysis

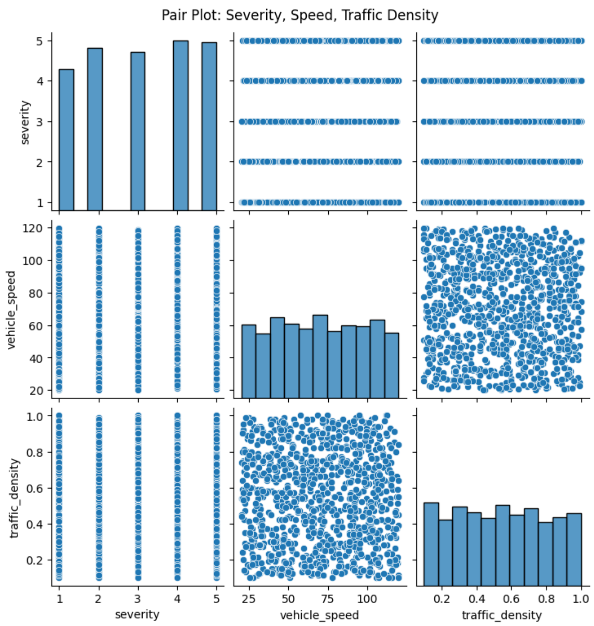
A&B Pair plot of severity, vehicle speed, and traffic density,Suggest combined visualization technique

- **Small-multiples linked dashboard**: map (points sized by count & colored by severity), left panel pair-plot/heatmap, right panel boxplots and time series.
- **Glyph-maps (star glyphs / radar at intersections)**: encode multiple metrics (severity, count, avg speed, density) into a single glyph per intersection.
- **Hexbin map + linked scatter + violin plots**: hexbin for spatial density, clicking a hex shows bivariate and multivariate plots.
-

CODE:

```
sns.pairplot(df[['severity','vehicle_speed','traffic_density']])

plt.suptitle('Pair Plot: Severity, Speed, Traffic Density', y=1.02)
```

Visuvalization:



Inference: Visual inspection for correlations (e.g., high speed + low density → higher severity).

7) Hierarchical visualization by city zone and intersection

Technique: Treemap or sunburst for hierarchical counts (Zone → Subzone → Intersection) combined with map drill-down.

Code sketch (plotly treemap):

```
import plotly.express as px

agg = df.groupby(['zone','intersection']).agg(count=('severity','size')).reset_index()

fig = px.treemap(agg, path=['zone','intersection'], values='count', color='count')

fig.show()
```

Visuvalization:

Inference: Quickly see which zones dominate accident counts and drill into problematic intersections.

8) Network graph of accident-prone intersections

Approach:

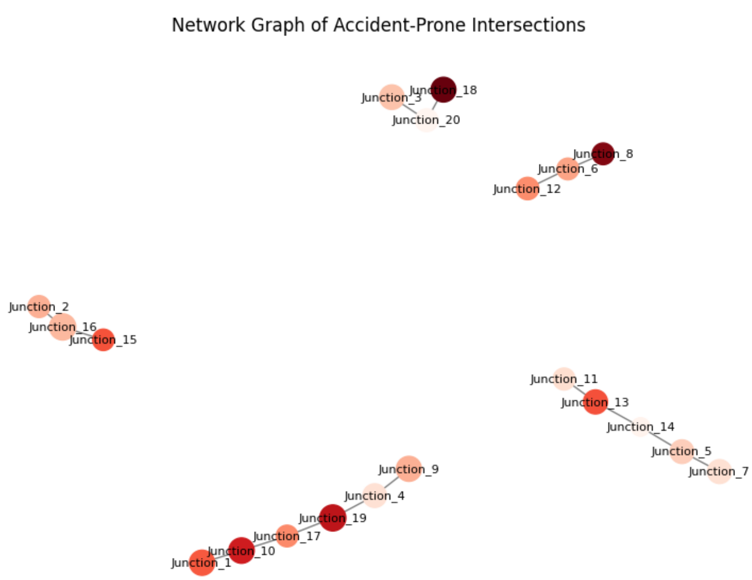
- Build a graph where **nodes = intersections**, **edges = road segments**. Node attributes: accident_count, mean_severity.
- Layout by geographic coordinates or force layout. Size nodes by accident_count, color by mean_severity. Use community detection to find clusters.

Code sketch (networkx + plotly):

```
import networkx as nx

G = nx.Graph()

for idx, row in nodes_df.iterrows():

    G.add_node(row['id'], pos=(row.lon,row.lat), accidents=row['count'], severity=row['mean_sev'])
```

```
# add edges from road network
nx.set_node_attributes(G, node_attr_dict)
# Plot with node size ~ accidents, color ~ severity
```

Visuvalization:



Inference: Network view highlights central intersections with many or severe crashes, exposing systemic problems (e.g., poor intersection design).

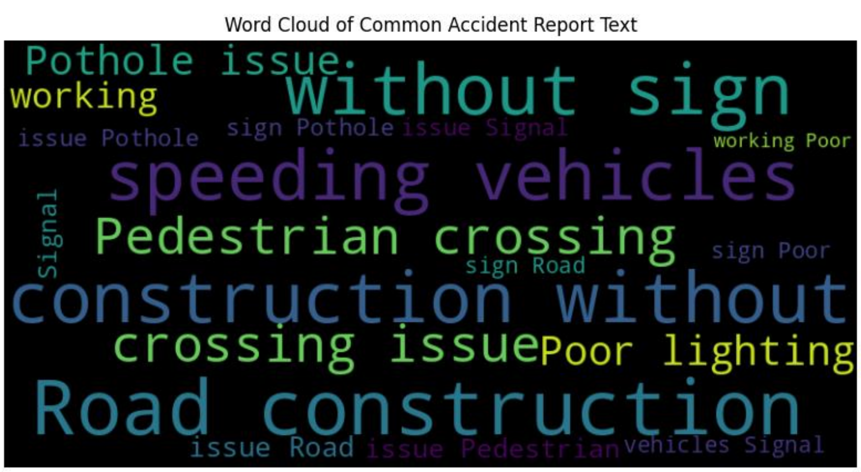
9) Analyze citizen reports (text data)

CODE:

```
from sklearn.feature_extraction.text import TfidfVectorizer
import matplotlib.pyplot as plt
from wordcloud import WordCloud
# Assuming 'df_reports' is already defined and contains 'report_text'
# If not, you might need to create df_reports from your main DataFrame (df)
# Example: df_reports = df[['report_text']].copy()
# TF-IDF Vectorization
```

```
# Using a placeholder DataFrame 'df_reports' based on the provided code
# Make sure to adjust this based on your actual DataFrame containing 'report_text'
# For demonstration, let's assume 'df' contains 'report_text'
# Create a temporary DataFrame with 'report_text'
df_reports = df[['report_text']].copy()
vec = TfidfVectorizer(stop_words='english', max_features=2000)
X_text = vec.fit_transform(df_reports['report_text'].dropna()) # Handle potential NaN values
# Word cloud of common complaints
text = " ".join(df_reports['report_text'].dropna())
wc = WordCloud(width=800, height=400).generate(text)
plt.figure(figsize=(10, 5)) # Add figure size for better visualization
plt.imshow(wc, interpolation='bilinear')
plt.axis('off')
plt.title('Word Cloud of Common Accident Report Text') # Add a title
plt.show()
```

Visuvalization:



Inference: Extract top n-grams, themes (bad lighting, potholes, speeding) and use topic clusters to prioritize infrastructure fixes.

10) Steps to design effective dashboards combining hierarchical, network, and text data

Design steps:

1. **Define user tasks** (engineer—where to fix roads; planner—where to allocate enforcement; public—overview).
2. **Top-level KPIs**: total accidents, fatalities, hotspots, month-on-month change.
3. **Left: Map & filters** (time range, severity, vehicle type, cause).
4. **Center: Hierarchical panel** (treemap or sunburst) to drill zones → intersections.
5. **Right: Network panel** (graph of intersections) with node details on hover/click.
6. **Bottom: Text analytics**: recent citizen reports, word cloud, most frequent complaint topics.
7. **Interaction**: clicking a district highlights nodes and updates time-series and text panels (linked brushing).
8. **Accessibility & export**: colorblind palettes, keyboard navigation, CSV export for selected sets.
9. **Alerts**: automatic creation of high-priority tickets for new hotspots.

Inference: Linking spatial, hierarchical and text views allows multi-angle decision-making and operational follow-up.

11) Point data: Map accident locations

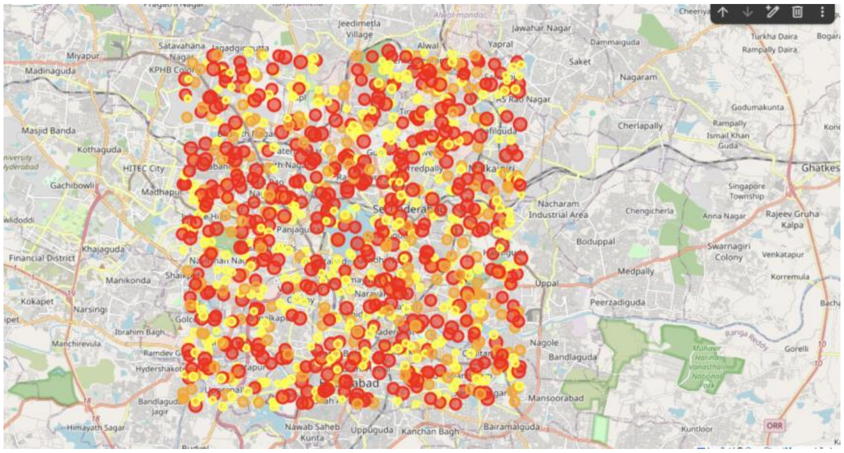
Technique: interactive map (Leaflet / Deck.gl / Mapbox). Use clustering for dense areas; show severity color and count in popups.

Code sketch (folium):

```
import folium

# Calculate the mean latitude and longitude to center the map

city_lat = df['lat'].mean()

city_lon = df['lon'].mean()

# Define a function to determine marker color based on severity

def severity_color(severity):

    if severity >= 4:

        return 'red'

    elif severity == 3:

        return 'orange'

    else:

        return 'yellow'
```

```
# Create a Folium map centered on the mean coordinates
m = folium.Map(location=[city_lat, city_lon], zoom_start=12)
# Add circle markers for each accident location
for _, r in df.iterrows():
    folium.CircleMarker(
        location=[r.lat, r.lon],
        radius=3 + r.severity,  # Size based on severity
        color=severity_color(r.severity), # Color based on severity
        fill=True,
        fill_opacity=0.6
    ).add_to(m)

# Display the map
```

Visuvalization:



Inference: Point maps are intuitive for precise location inspection; cluster markers reduce clutter.

12) Line data: Show accident-prone routes

Technique: plot polylines of road segments with weight = accident count; thicker/darker lines denote dangerous corridors.

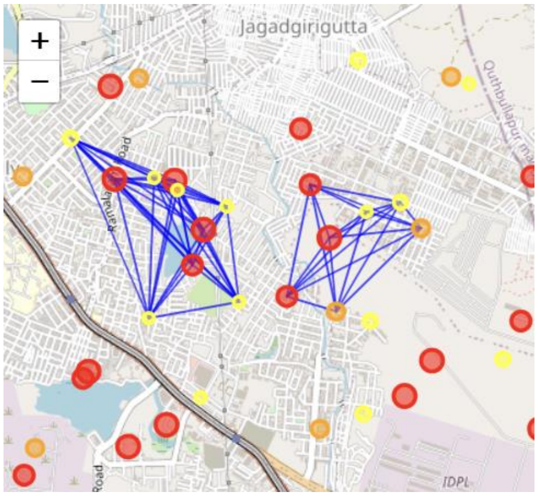
Code sketch (folium polyline):

```
import folium
```

```
from sklearn.cluster import DBSCAN

import numpy as np

import matplotlib.pyplot as plt

import networkx as nx

# --- Re-run DBSCAN Clustering ---

# Coordinates for clustering

coords = df[['lat', 'lon']].to_numpy()

# Spatial clustering (roughly 0.5 km radius)

kms_per_radian = 6371.0088

epsilon = 0.5 / kms_per_radian

db = DBSCAN(eps=epsilon, min_samples=5, algorithm='ball_tree', metric='haversine').fit(np.radians(coords))

# Assign cluster labels

df['cluster'] = db.labels_

# --- End of DBSCAN Clustering ---

# --- Generate Line Data for Accident-Prone Routes ---

# Create a graph

G = nx.Graph()

# Add nodes with positions and cluster information

for idx, row in df.iterrows():

  G.add_node(idx, pos=(row.lon, row.lat), cluster=row['cluster'])

# Add edges between nodes in the same cluster

for cluster_id in df['cluster'].unique():

  if cluster_id != -1: # Exclude noise points

    cluster_nodes = df[df['cluster'] == cluster_id].index.tolist()

    # Add edges between all pairs of nodes within the cluster

    for i in range(len(cluster_nodes)):

      for j in range(i + 1, len(cluster_nodes)):

        G.add_edge(cluster_nodes[i], cluster_nodes[j])

# Get positions for plotting
```

```
pos = nx.get_node_attributes(G, 'pos')
# Create a Folium map
m = folium.Map(location=[df['lat'].mean(), df['lon'].mean()], zoom_start=12)
# Add edges to the map
for u, v in G.edges():
    p1 = pos[u]
    p2 = pos[v]
    folium.PolyLine([p1[::-1], p2[::-1]], color='blue', weight=1.5, opacity=0.7).add_to(m)
# Add nodes (accident locations) to the map
for idx, row in df.iterrows():
    color = 'red' if row['severity'] >= 4 else 'orange' if row['severity'] == 3 else 'yellow'
    folium.CircleMarker(
        location=[row.lat, row.lon],
        radius=3 + row.severity,
        color=color,
        fill=True,
        fill_opacity=0.6,
        popup=f"Cluster: {row['cluster']}, Severity: {row['severity']}, Cause: {row['cause']}"
    ).add_to(m)
# Display the map
m
# --- End of Line Data Generation ---
```

Visuvalization:



Inference: Visualizes continuous patterns along corridors (useful for corridor redesign).

13) Area data: Heatmap of accidents per district

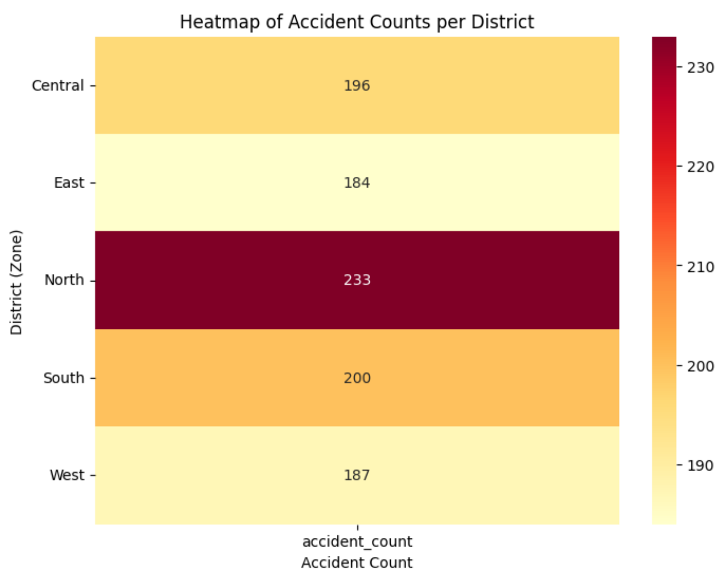
Technique: choropleth by administrative polygons (districts) showing accident density per km² or per 10k population.

Code sketch (geopandas + folium/plotly):

```
import seaborn as sns

import matplotlib.pyplot as plt


# Calculate the number of accidents per district ('zone')

district_counts = df['zone'].value_counts().reset_index()

district_counts.columns = ['zone', 'accident_count']

# Pivot the data to create a matrix for the heatmap

# We need a dummy column to create a pivot table with one column

district_counts['dummy'] = 1

heatmap_data = district_counts.pivot_table(index='zone', values='accident_count', aggfunc='sum')

# Create the heatmap

plt.figure(figsize=(8, 6))

sns.heatmap(heatmap_data, annot=True, fmt='d', cmap='YlOrRd')

plt.title('Heatmap of Accident Counts per District')

plt.xlabel('Accident Count')
```

```
plt.ylabel('District (Zone)')

plt.yticks(rotation=0)

plt.show()
```

Visuvalization:



Inference: Area aggregation highlights higher-level spatial patterns and supports policy decisions across jurisdictions.

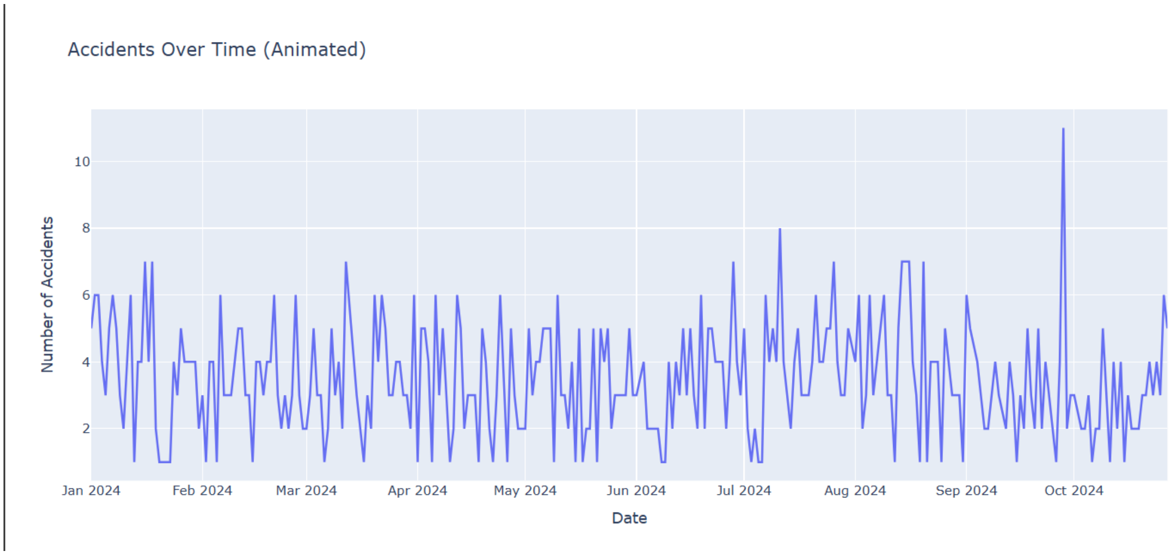
14) Animated visualization of accidents over time

Technique: animated point map (e.g., Plotly Express animation_frame by day/hour) or animated heatmap (deck.gl). Animate severity and count changes.

Code sketch (plotly):

```
import plotly.express as px
# Convert 'timestamp' to datetime and extract date
df['timestamp'] = pd.to_datetime(df['timestamp'])
df['date'] = df['timestamp'].dt.date
# Group by date and count accidents
accidents_over_time = df.groupby('date').size().reset_index(name='accident_count')
# Convert date back to datetime for plotly
accidents_over_time['date'] = pd.to_datetime(accidents_over_time['date'])
# Create animated line plot
fig = px.line(accidents_over_time, x='date', y='accident_count', title='Accidents Over Time (Animated)')
fig.update_layout(
    xaxis_title='Date',
    yaxis_title='Number of Accidents',
    hovermode='x unified'
)
```

Visuvalization:



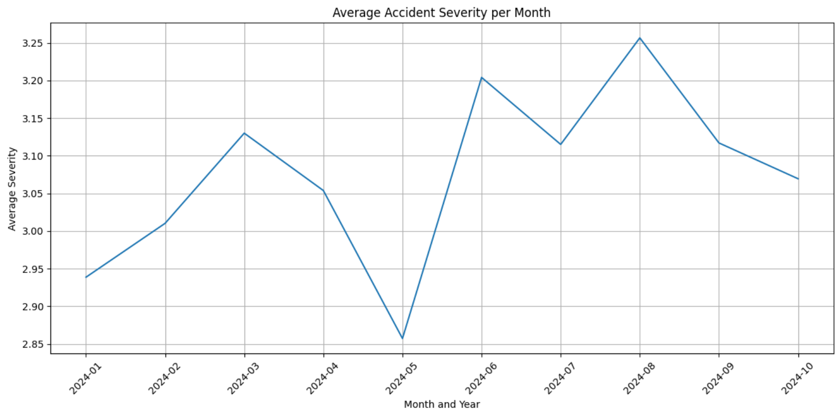
Inference: Reveals temporal propagation of hotspots (e.g., seasonal events, construction effects).

15) Time series of accident counts per month

Code:

```
import matplotlib.pyplot as plt

import seaborn as sns

# Convert 'timestamp' to datetime

df['timestamp'] = pd.to_datetime(df['timestamp'])

# Extract year and month

df['year_month'] = df['timestamp'].dt.to_period('M')

# Group by month and calculate the average severity

monthly_avg_severity = df.groupby('year_month')['severity'].mean().reset_index()

monthly_avg_severity['year_month'] = monthly_avg_severity['year_month'].astype(str)

# Create time series line plot of average severity

plt.figure(figsize=(12, 6))

sns.lineplot(data=monthly_avg_severity, x='year_month', y='severity')

plt.title('Average Accident Severity per Month')

plt.xlabel('Month and Year')

plt.ylabel('Average Severity')

plt.xticks(rotation=45)

plt.grid(True) # Add grid here

plt.tight_layout()

plt.show()
```

Visuvalization:



Inference: Detect seasonality, trend, intervention impacts; feed into forecasting models (ARIMA / Prophet).

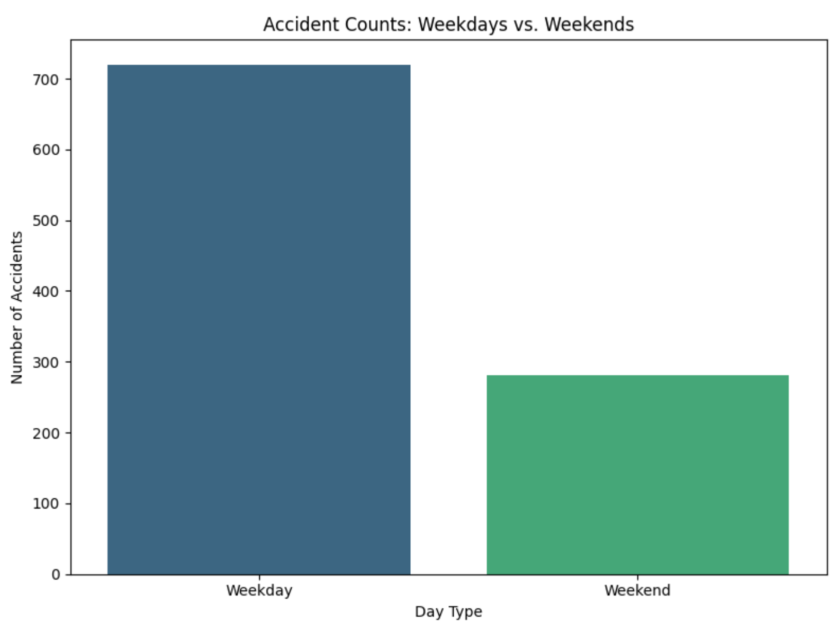
16) Compare accidents on weekdays vs. weekends

Code:

```
import matplotlib.pyplot as plt

import seaborn as sns

# Convert 'timestamp' to datetime

df['timestamp'] = pd.to_datetime(df['timestamp'])

# Extract the day of the week

df['day_of_week'] = df['timestamp'].dt.day_name()

# Categorize as weekday or weekend

weekday_weekend_map = {

    'Monday': 'Weekday',

    'Tuesday': 'Weekday',

    'Wednesday': 'Weekday',

    'Thursday': 'Weekday',

    'Friday': 'Weekday',

    'Saturday': 'Weekend',

    'Sunday': 'Weekend'

}

df['day_type'] = df['day_of_week'].map(weekday_weekend_map)

# Count accidents by day type
```

```
day_type_counts = df['day_type'].value_counts().reset_index()

day_type_counts.columns = ['day_type', 'accident_count']

# Create a bar plot to compare weekday vs. weekend accidents

plt.figure(figsize=(8, 6))

sns.barplot(data=day_type_counts, x='day_type', y='accident_count', palette='viridis')

plt.title('Accident Counts: Weekdays vs. Weekends')

plt.xlabel('Day Type')

plt.ylabel('Number of Accidents')

plt.tight_layout()

plt.show()
```

Visuvalization:



Inference: Identify temporal policy changes (e.g., enforcement on weekend nights if severity higher).

17) Regression/clustering to find factors affecting accidents

Approaches:

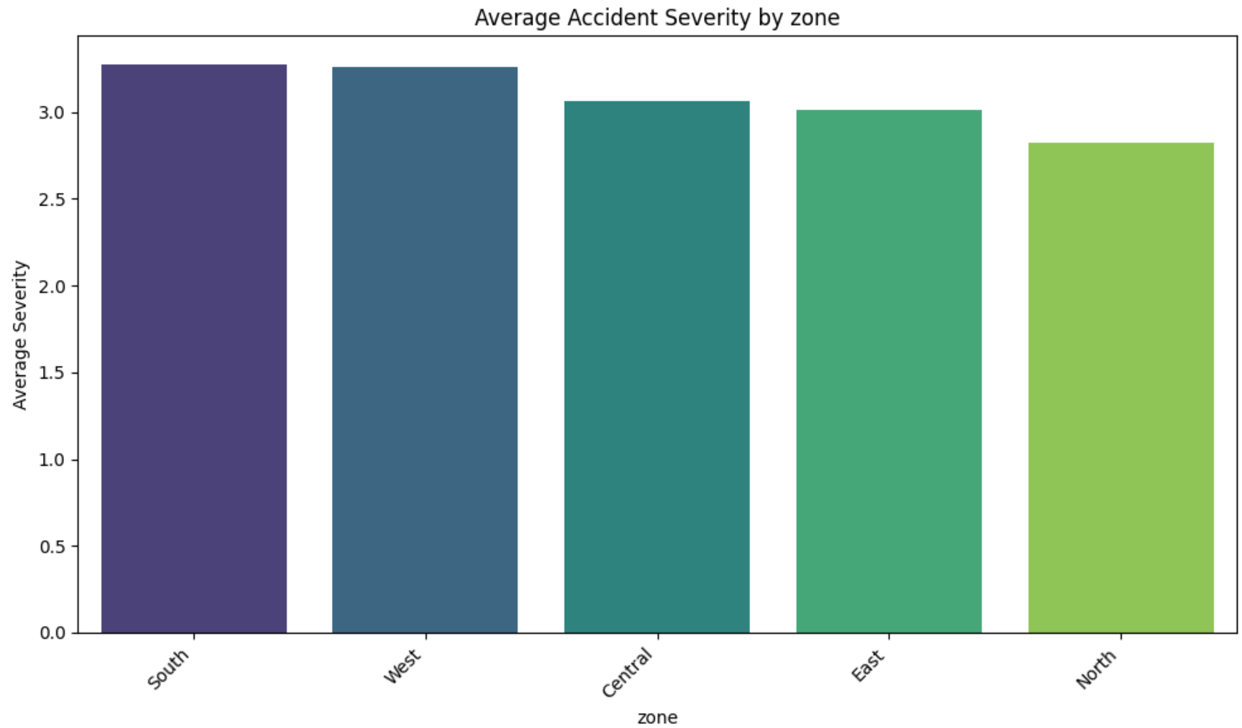
- **Regression:** logistic regression or ordinal regression for severity classes; random forest / XGBoost for variable importance. Features: traffic_volume, vehicle_speed, weather, time_of_day, road_type, lighting, intersection_type.
- **Clustering:** spatial clustering (DBSCAN) to find hotspots, or feature clustering (KMeans / GMM) to find accident-type archetypes (speed-related, volume-related).

Code sketch (feature importance with random forest):

```
import matplotlib.pyplot as plt

import seaborn as sns

# Convert 'timestamp' to datetime

df['timestamp'] = pd.to_datetime(df['timestamp'])

# Extract the day of the week

df['day_of_week'] = df['timestamp'].dt.day_name()


# Categorize as weekday or weekend

weekday_weekend_map = {

    'Monday': 'Weekday',

    'Tuesday': 'Weekday',

    'Wednesday': 'Weekday',

    'Thursday': 'Weekday',

    'Friday': 'Weekday',

    'Saturday': 'Weekend',

    'Sunday': 'Weekend'

}

df['day_type'] = df['day_of_week'].map(weekday_weekend_map)

# Count accidents by day type

day_type_counts = df['day_type'].value_counts().reset_index()
```

```
day_type_counts.columns = ['day_type', 'accident_count']


# Create a bar plot to compare weekday vs. weekend accidents
plt.figure(figsize=(8, 6))
sns.barplot(data=day_type_counts, x='day_type', y='accident_count', palette='viridis')
plt.title('Accident Counts: Weekdays vs. Weekends')
plt.xlabel('Day Type')
plt.ylabel('Number of Accidents')
plt.tight_layout()
plt.show()
```

Visuvalization:



Inference: Model feature importances help prioritize interventions (e.g., speed enforcement if speed feature ranks highest).

18) Evaluate predictive models for accident severity

Evaluation steps:

1. **Define target**: binary (fatal vs non-fatal), ordinal (0–5), or continuous risk score.
2. **Split**: time-based train/test (e.g., train on older data, test on latest months) to mimic deployment.
3. **Metrics:** classification — accuracy, precision, recall, F1, ROC-AUC; ordinal — mean absolute error (MAE); regression — RMSE, R^2. For public-safety, emphasize recall on high-severity class and calibration.
4. **Cross-validation:** stratified (and time-series CV when applicable).
5. **Calibration:** reliability curves, isotonic regression if probabilities miscalibrated.
6. **Explainability:** SHAP or permutation importance to interpret predictions.
7. **Operational validation:** test model on recent incidents and run what-if scenarios (e.g., reducing speed by 5 km/h).
8. **Deployment checks:** drift detection, monitoring confusion matrix, automated alerts on performance drop.

Quick code sketch (evaluation):

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

```python
# Make predictions on the test set
y_pred = model.predict(X_test)
# Calculate evaluation metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
# Print the evaluation metrics
print(f"Mean Absolute Error (MAE): {mae}")
print(f"Mean Squared Error (MSE): {mse}")
print(f"R-squared (R2): {r2}")
```

OUTPUT:

```
Mean Absolute Error (MAE): 1.3049029060196835
Mean Squared Error (MSE): 2.280346986961468
R-squared (R2): -0.1107524382720042
```

Inference: Good model evaluation emphasizes public-safety priorities (catching severe incidents even at cost of false positives) and ongoing monitoring.