Taxi Accidents analysis - EDA

August 15, 2025

1 Predicting Severity of Taxi Accidents in the UK - Part 1: EDA and pre-processing

Credit to the following people for working with me on the EDA part of this project:

1. Thu Trang Le
2. Xuan Nam Tram
3. Minh Hoang Vo

1.1 Problem definition

Taxi and ride hailing apps face the risk of accidents which can affect passenger satisfaction, claims against the company and pricing. Traditional operational strategies ignore pre-trip information such as weather, road conditions and other key details. This leads to missed opportunities for risk reduction, cost optimisation and safer fleet management

Goal of project

Predict severity of taxi collisions for ride hailing apps before trip begins, using pre-accident data available at time of booking

Potential Benefits - Insurance cost reduction - Proactive driver training - Better routing to avoid risky areas - Risk based pricing

1.2 Data importing and setup

[1]:
```
# importing relevant libraries

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import plotly.express as px
import seaborn as sns

# Hiding warnings for better presentation
import warnings
warnings.filterwarnings("ignore")
```

[2]:
```python
# Importing data files sourced from Department of Transportation - UK

collisions = pd.read_csv('collision-last-5-years.csv')
vehicles = pd.read_csv('vehicle-last-5-years.csv')
```

[3]:
```python
# Since the dataset is very large, we keep only relevant columns
# These were selected based on what information is available prior to a trip␣
↪starting, route that may be taken and time

collisions = collisions[[
    'accident_reference', 'longitude', 'latitude',
    'accident_severity', 'date', 'day_of_week', 'time',
    'local_authority_ons_district', 'first_road_class', 'road_type',
    'speed_limit', 'light_conditions', 'weather_conditions',
    'road_surface_conditions', 'special_conditions_at_site',
    'urban_or_rural_area'
]]
vehicles = vehicles[[
    'accident_year', 'accident_reference', 'vehicle_type', 'sex_of_driver',␣
↪'age_of_driver',
    'driver_imd_decile', 'propulsion_code'
]]
```

[4]:
```python
# Inspecting collisions tables

collisions.head()
```

[4]:
```
  accident_reference  longitude   latitude  accident_severity        date  \
0           10128300  -0.153842  51.508057                  3  18/02/2019
1           10152270  -0.127949  51.436208                  3  15/01/2019
2           10155191  -0.124193  51.526795                  3  01/01/2019
3           10155192  -0.191044  51.546387                  2  01/01/2019
4           10155194  -0.200064  51.541121                  3  01/01/2019

   day_of_week   time local_authority_ons_district  first_road_class  \
0            2  17:50                    E09000033                 3
1            3  21:45                    E09000022                 3
2            3  01:50                    E09000007                 4
3            3  01:20                    E09000007                 4
4            3  00:40                    E09000005                 3

   road_type  speed_limit  light_conditions  weather_conditions  \
0          1           30                 1                   1
1          2           30                 4                   1
2          6           30                 4                   1
3          6           20                 4                   1
4          6           30                 4                   1
```

```
   road_surface_conditions  special_conditions_at_site  urban_or_rural_area
0                        1                           0                    1
1                        1                           0                    1
2                        1                           0                    1
3                        1                           0                    1
4                        1                           0                    1
```

[5]:
```
# Inspecting vehicle table

vehicles.head()
```

[5]:
```
   accident_year accident_reference  vehicle_type  sex_of_driver  \
0           2019           10128300             9              1
1           2019           10128300             9              3
2           2019           10152270             9              2
3           2019           10152270             9              3
4           2019           10155191             9              1

   age_of_driver  driver_imd_decile  propulsion_code
0             58                  2               -1
1             -1                  2               -1
2             24                  3               -1
3             -1                  6               -1
4             45                  4               -1
```

1.3 Basic pre-processing

Note: Some pre-processing steps are be done towards the end of the file. These are only some basic pre-processing steps done to enable better EDA

[6]:
```
# We merge both tables based on the accident reference column

df = pd.merge(collisions, vehicles, on = 'accident_reference', how = 'inner')
df.head()
```

[6]:
```
  accident_reference  longitude   latitude  accident_severity        date  \
0           10128300  -0.153842  51.508057                  3  18/02/2019
1           10128300  -0.153842  51.508057                  3  18/02/2019
2           10152270  -0.127949  51.436208                  3  15/01/2019
3           10152270  -0.127949  51.436208                  3  15/01/2019
4           10155191  -0.124193  51.526795                  3  01/01/2019

   day_of_week   time local_authority_ons_district  first_road_class  \
0            2  17:50                    E09000033                 3
1            2  17:50                    E09000033                 3
2            3  21:45                    E09000022                 3
3            3  21:45                    E09000022                 3
```

```
4            3  01:50                    E09000007              4

   road_type  …  weather_conditions  road_surface_conditions  \
0          1  …                   1                        1
1          1  …                   1                        1
2          2  …                   1                        1
3          2  …                   1                        1
4          6  …                   1                        1

   special_conditions_at_site  urban_or_rural_area  accident_year  \
0                           0                    1           2019
1                           0                    1           2019
2                           0                    1           2019
3                           0                    1           2019
4                           0                    1           2019

   vehicle_type  sex_of_driver  age_of_driver  driver_imd_decile  \
0             9              1             58                  2
1             9              3             -1                  2
2             9              2             24                  3
3             9              3             -1                  6
4             9              1             45                  4

   propulsion_code
0               -1
1               -1
2               -1
3               -1
4               -1

[5 rows x 22 columns]
```

[7]:
```python
# We are interested in only taxis and private car hire accidents
# Accidents with this specification are recorded as '8' on the vehicle_type␣
 ↪column

df = df[df['vehicle_type'] == 8]
```

[8]:
```python
# changing date values to date type
df['date'] = pd.to_datetime(df['date'], format = '%d/%m/%Y')

# changing time values to time type
df['time'] = pd.to_datetime(df['time'], format = '%H:%M').dt.time

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 12808 entries, 13 to 769242
```

```
Data columns (total 22 columns):
 #   Column                         Non-Null Count  Dtype
---  ------                         --------------  -----
 0   accident_reference             12808 non-null  object
 1   longitude                      12807 non-null  float64
 2   latitude                       12807 non-null  float64
 3   accident_severity              12808 non-null  int64
 4   date                           12808 non-null  datetime64[ns]
 5   day_of_week                    12808 non-null  int64
 6   time                           12808 non-null  object
 7   local_authority_ons_district   12808 non-null  object
 8   first_road_class               12808 non-null  int64
 9   road_type                      12808 non-null  int64
 10  speed_limit                    12808 non-null  int64
 11  light_conditions               12808 non-null  int64
 12  weather_conditions             12808 non-null  int64
 13  road_surface_conditions        12808 non-null  int64
 14  special_conditions_at_site     12808 non-null  int64
 15  urban_or_rural_area            12808 non-null  int64
 16  accident_year                  12808 non-null  int64
 17  vehicle_type                   12808 non-null  int64
 18  sex_of_driver                  12808 non-null  int64
 19  age_of_driver                  12808 non-null  int64
 20  driver_imd_decile              12808 non-null  int64
 21  propulsion_code                12808 non-null  int64
dtypes: datetime64[ns](1), float64(2), int64(16), object(3)
memory usage: 2.2+ MB
```

1.4 Splitting training and testing data

[9]:

```python
# Importing library for train-test split
from sklearn.model_selection import train_test_split

# Specifying target variable (y) and predictor variables (X)
X = df.drop('accident_severity', axis = 1)
y = df['accident_severity']

# Splitting the training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size = 0.2, # 20% of␣
 ↪data will be used for testing
                                                    stratify = y, # Ensuring␣
 ↪test set has a good representation of all target values
                                                    random_state = 7
                                                   )

print(f'Training data size: {y_train.shape} rows')
```

```python
print(f'Test data size: {y_test.shape} rows')
```

```
Training data size: (10246,) rows
Test data size: (2562,) rows
```

[10]:
```python
# Combining training target and predictor variables

train_df = pd.concat([X_train, y_train], axis = 1)
test_df = pd.concat([X_test, y_test], axis = 1)

train_df.head()
```

[10]:
```
       accident_reference  longitude   latitude        date  day_of_week  \
560589          170L31182  -1.070140  54.618935  2022-10-08            7
408658          31C118021  -1.147784  52.955104  2021-09-12            1
520743           10410155  -0.382588  51.510418  2022-11-12            7
673997           52301874  -2.983360  53.460458  2023-08-10            5
674501           52302321  -2.842457  53.421571  2023-09-30            7

            time local_authority_ons_district  first_road_class  road_type  \
560589  22:00:00                    E06000003                 6          6
408658  01:30:00                    E06000018                 6          6
520743  10:40:00                    E09000009                 6          2
673997  17:15:00                    E08000014                 3          6
674501  16:24:00                    E08000011                 3          3

        speed_limit  …  road_surface_conditions  special_conditions_at_site  \
560589           30  …                        1                           0
408658           30  …                        1                           0
520743           20  …                        9                           0
673997           30  …                        1                           0
674501           20  …                        2                           0

        urban_or_rural_area  accident_year  vehicle_type  sex_of_driver  \
560589                    1           2022             8              1
408658                    1           2021             8              1
520743                    1           2022             8              3
673997                    1           2023             8              1
674501                    1           2023             8              1

        age_of_driver  driver_imd_decile  propulsion_code  accident_severity
560589             -1                 -1               -1                  3
408658             -1                  1                2                  3
520743             -1                  2                3                  3
673997             52                  3                8                  3
674501             37                  1                2                  3

[5 rows x 22 columns]
```

1.5 Exploratory Data Analysis (EDA)

This stage looks at any trends or particularly interesting insights in the data. This will allow us to also determine the best models that may work on the data.

1.5.1 1. Missing value analysis

[11]:
```
# Summarising values of all categorical values

categorical_cols = [
    'day_of_week', 'local_authority_ons_district', 'first_road_class',␣
  ↪'road_type',
    'light_conditions', 'weather_conditions', 'speed_limit',
    'road_surface_conditions', 'special_conditions_at_site',
    'urban_or_rural_area', 'accident_year', 'vehicle_type',
    'sex_of_driver', 'driver_imd_decile', 'propulsion_code','speed_limit'
]

for cat_cols in categorical_cols:
    print(df.value_counts(cat_cols))
```

```
day_of_week
7    2200
6    2132
5    1880
4    1804
1    1775
3    1600
2    1417
Name: count, dtype: int64
local_authority_ons_district
E09000033    872
E08000012    468
E09000020    334
E09000007    331
E08000025    326
            …
S12000023      1
E07000155      1
E07000229      1
E07000169      1
E07000031      1
Name: count, Length: 361, dtype: int64
first_road_class
3    6234
6    4282
4    1298
5     810
```

```
1     163
2      21
Name: count, dtype: int64
road_type
6    8737
3    1970
2     752
1     642
9     493
7     214
Name: count, dtype: int64
light_conditions
1    7523
4    4562
7     400
6     227
5      96
Name: count, dtype: int64
weather_conditions
1    9851
2    1625
8     505
9     437
5     170
4     126
3      45
7      43
6       6
Name: count, dtype: int64
speed_limit
 30    7773
 20    3439
 40     716
 60     370
 50     273
 70     234
-1        3
Name: count, dtype: int64
road_surface_conditions
 1    9231
 2    3116
 9     239
 4     109
-1      84
 3      20
 5       9
Name: count, dtype: int64
special_conditions_at_site
```

```
 0    11926
 9      480
 4      189
-1      101
 1       54
 3       21
 5       15
 2        8
 6        8
 7        6
Name: count, dtype: int64
urban_or_rural_area
1    11516
2     1291
3        1
Name: count, dtype: int64
accident_year
2019    3593
2021    2556
2022    2529
2023    2109
2020    2021
Name: count, dtype: int64
vehicle_type
8    12808
Name: count, dtype: int64
sex_of_driver
1    10313
3     2107
2      388
Name: count, dtype: int64
driver_imd_decile
-1     2908
 1     2028
 2     1980
 3     1549
 4     1095
 5      888
 6      689
 7      543
 8      461
 9      393
 10     274
Name: count, dtype: int64
propulsion_code
 2     5873
 8     3825
-1     2149
```

```
 1      572
 3      318
 7       44
 12      26
 6        1
Name: count, dtype: int64
speed_limit
 30    7773
 20    3439
 40     716
 60     370
 50     273
 70     234
-1        3
Name: count, dtype: int64
```

[12]:

```python
# Missing values, based on the above summary, are denoted with a -1 in the␣
↪above dataset
# We count its occurences and summarise
columns_to_analyse = [
    'day_of_week', 'first_road_class', 'road_type',
    'speed_limit', 'light_conditions', 'weather_conditions',
    'road_surface_conditions', 'special_conditions_at_site',
    'urban_or_rural_area', 'accident_year', 'vehicle_type', 'sex_of_driver',
    'age_of_driver', 'driver_imd_decile', 'propulsion_code',␣
↪'accident_severity',
    'date', 'time', 'local_authority_ons_district'
]

missing_summary = {
    col: (train_df[col] == -1).sum()
    for col in columns_to_analyse
}

# Convert to DataFrame for better visualization
missing_df = pd.DataFrame.from_dict(missing_summary, orient='index',␣
↪columns=['Missing (-1) Count'])
missing_df['Total'] = X_train.shape[0]
missing_df['Missing %'] = (missing_df['Missing (-1) Count'] /␣
↪missing_df['Total']) * 100
missing_df = missing_df.sort_values(by='Missing %', ascending=False)

missing_df[missing_df['Missing %'] != 0] # only display rows where missing␣
↪values are present
```

[12]:

	Missing (-1) Count	Total	Missing %
driver_imd_decile	2326	10246	22.701542

```
age_of_driver                                2011  10246  19.627172
propulsion_code                              1715  10246  16.738239
special_conditions_at_site                     81  10246   0.790552
road_surface_conditions                        69  10246   0.673434
speed_limit                                     2  10246   0.019520
```

The key missing values are the top 3: Driver IMD decile, age of driver and the propulsion code. We we will replace -1 with NaN for cleaner EDA, and later on impute blank values.

[13]:
```
# Making -1 values to be blank
train_df[columns_to_analyse] = train_df[columns_to_analyse].replace(-1, np.nan)

# The same is done to the test data
test_df[columns_to_analyse] = test_df[columns_to_analyse].replace(-1, np.nan)
```

1.5.2 2. Accidents by month, analysed by severity level

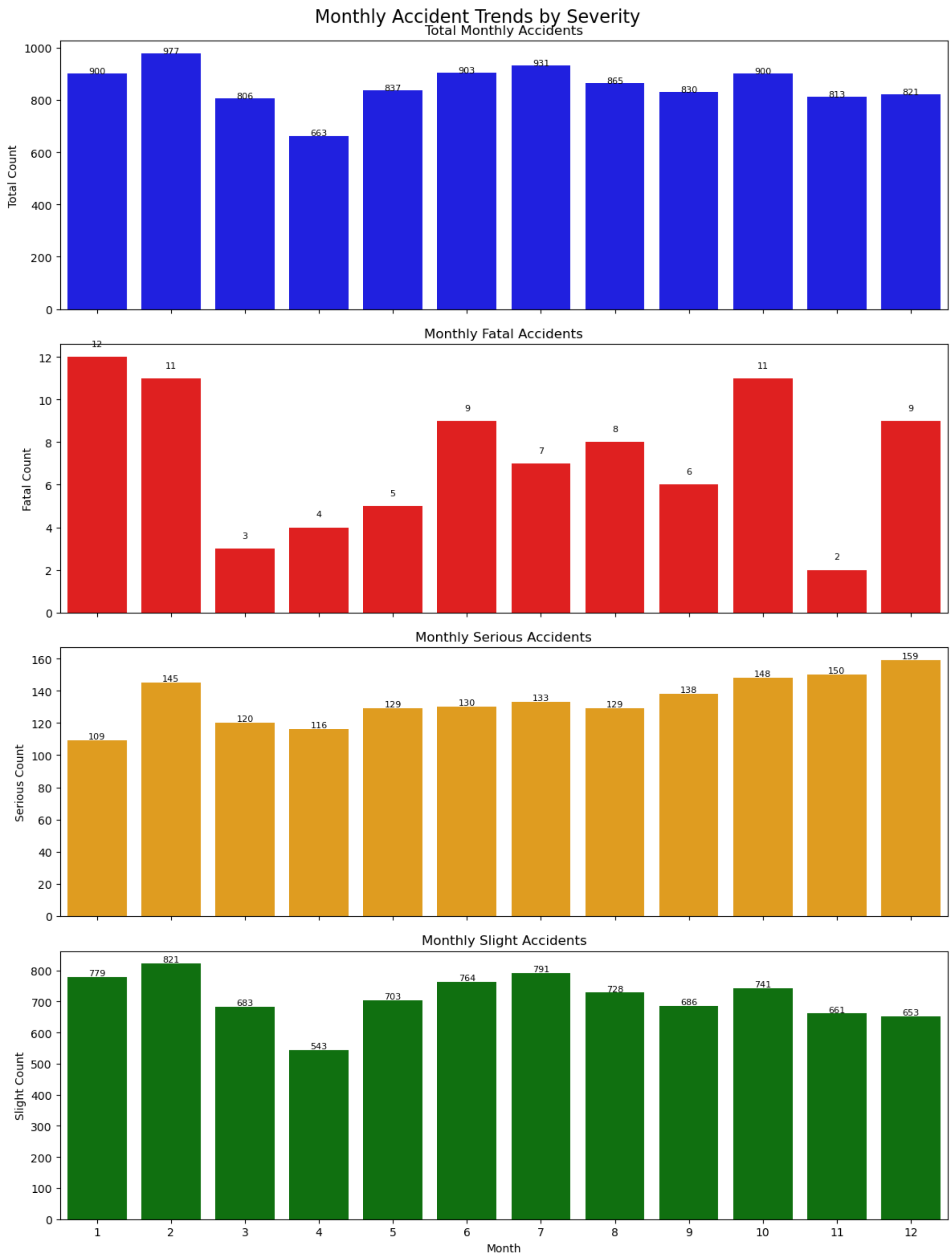
[14]:
```
# Using actual names instead of numbers for the casualty severity
train_df['accident_severity_label'] = train_df['accident_severity'].map({
    1: 'Fatal',
    2: 'Serious',
    3: 'Slight'
})
```

[15]:
```
# Getting month from 'date' column
train_df['month'] = train_df['date'].dt.month

# Get ordered list of available months
month_order = sorted(train_df['month'].dropna().unique())

# We also add the month column for the test set (required at later stages)
test_df['month'] = train_df['date'].dt.month
```

[16]:
```
# Grouping by severity level
all_accidents = train_df.groupby('month').size().reset_index(name='count')
fatal_acc = train_df[train_df['accident_severity'] == 1].groupby('month').
 ↪size().reset_index(name='count')
serious_acc = train_df[train_df['accident_severity'] == 2].groupby('month').
 ↪size().reset_index(name='count')
slight_acc = train_df[train_df['accident_severity'] == 3].groupby('month').
 ↪size().reset_index(name='count')
```

[17]:
```
# Setting up the plot
fig, axs = plt.subplots(4, 1, figsize=(12, 16), sharex=True)
fig.suptitle('Monthly Accident Trends by Severity', fontsize=16)

# Plotting Total accidents
```

```python
sns.barplot(data=all_accidents, x='month', y='count', ax=axs[0], color='blue')
axs[0].set_title('Total Monthly Accidents')
axs[0].set_ylabel('Total Count')
for bar in axs[0].patches:
    axs[0].text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.5,
                f'{int(bar.get_height())}', ha='center', fontsize=8)

# Plotting Fatal accidents
sns.barplot(data=fatal_acc, x='month', y='count', ax=axs[1], color='red')
axs[1].set_title('Monthly Fatal Accidents')
axs[1].set_ylabel('Fatal Count')
for bar in axs[1].patches:
    axs[1].text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.5,
                f'{int(bar.get_height())}', ha='center', fontsize=8)

# Plotting Serious accidents
sns.barplot(data=serious_acc, x='month', y='count', ax=axs[2], color='orange')
axs[2].set_title('Monthly Serious Accidents')
axs[2].set_ylabel('Serious Count')
for bar in axs[2].patches:
    axs[2].text(bar.get_x() + bar.get_width()/2, bar.get_height() + 1,
                f'{int(bar.get_height())}', ha='center', fontsize=8)

# Plotting Slight accidents
sns.barplot(data=slight_acc, x='month', y='count', ax=axs[3], color='green')
axs[3].set_title('Monthly Slight Accidents')
axs[3].set_ylabel('Slight Count')
axs[3].set_xlabel('Month')
for bar in axs[3].patches:
    axs[3].text(bar.get_x() + bar.get_width()/2, bar.get_height() + 5,
                f'{int(bar.get_height())}', ha='center', fontsize=8)

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



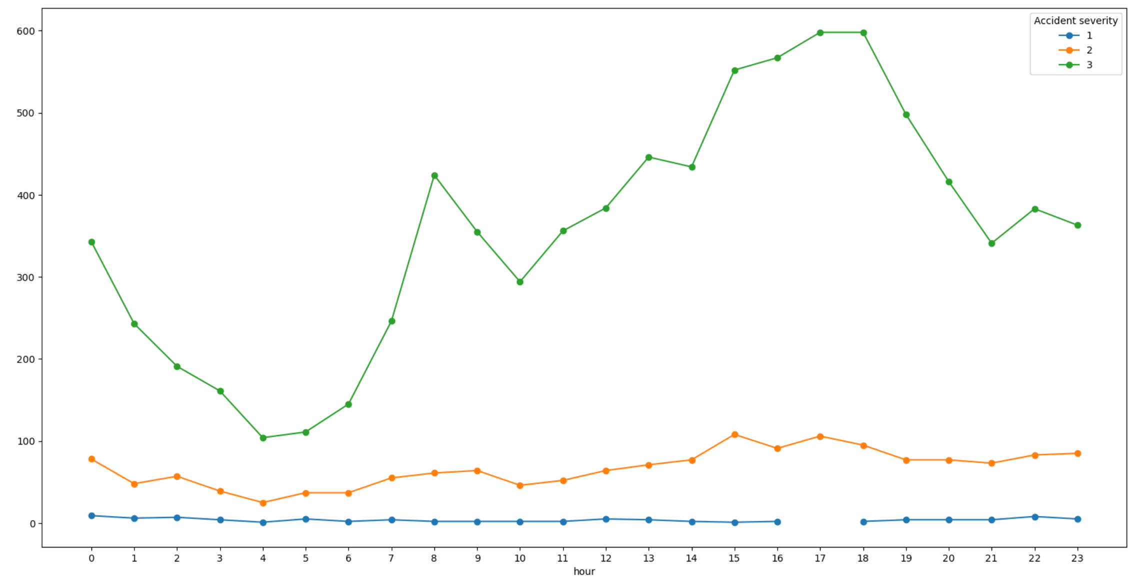
Key Insights

1. The number of accidents are almost similar across all months but February has the highest number of accidents.

2. January and February have the highest number of fatal accidents. This could be due to the poor weather conditions in these months (wet and slipery roads)
3. April - May marks the safest period of travel for taxis, possibly due to better visibility and normal road surfaces.
4. January to March represent the most risky periods for taxi drivers in terms of both frequency and severity

1.5.3 3. Accidents over time

[18]:
```
# Analysing accidents by time
train_df['time'] = pd.to_datetime(train_df['time'], format = '%H:%M:%S') #␣
↪Ensuring time is categorised correctly
train_df['hour'] = train_df['time'].dt.hour # Extracting the hour
accidents_by_time = train_df.groupby(['hour', 'accident_severity']).size().
↪unstack()

# Plotting the line chart

accidents_by_time.plot(kind = 'line', marker = 'o', figsize = (20, 10), label =␣
↪'Accidents by Hour')
plt.legend(title = 'Accident severity')
plt.xticks(range(0,24))
plt.show()
```

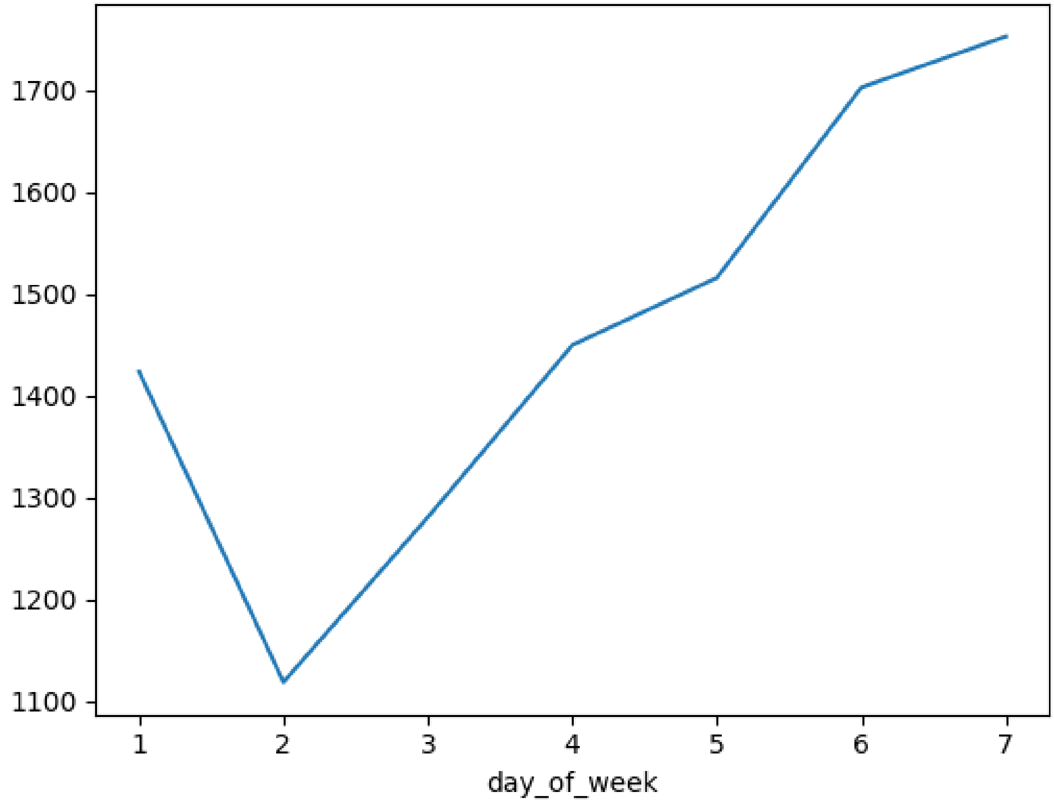


Accident timings have 3 key peaks 1. 8am - due to the morning rush hour hence more cars are on the road 2. 3pm-6pm - due to evening rush hour starting with students returning home, then workers coming back home 3. Midnight - due to lower visibility and possibly drunk driving cases which are common at night

1.5.4 4. Accidents by day of week

[19]:
```python
# grouping the accidents
accidents_by_day = train_df['day_of_week'].value_counts().sort_index()

# Making a simple plot to analyse
accidents_by_day.plot(kind = 'line')
```

[19]: <Axes: xlabel='day_of_week'>



Note: 1 = Sunday and 7 = Saturday, etc.

Accidents are highest on Fridays and Saturdays. This could be attributed to late night driving or drunk driving cases given that these are considered weekend days. To confirm this, we use the heatmap below

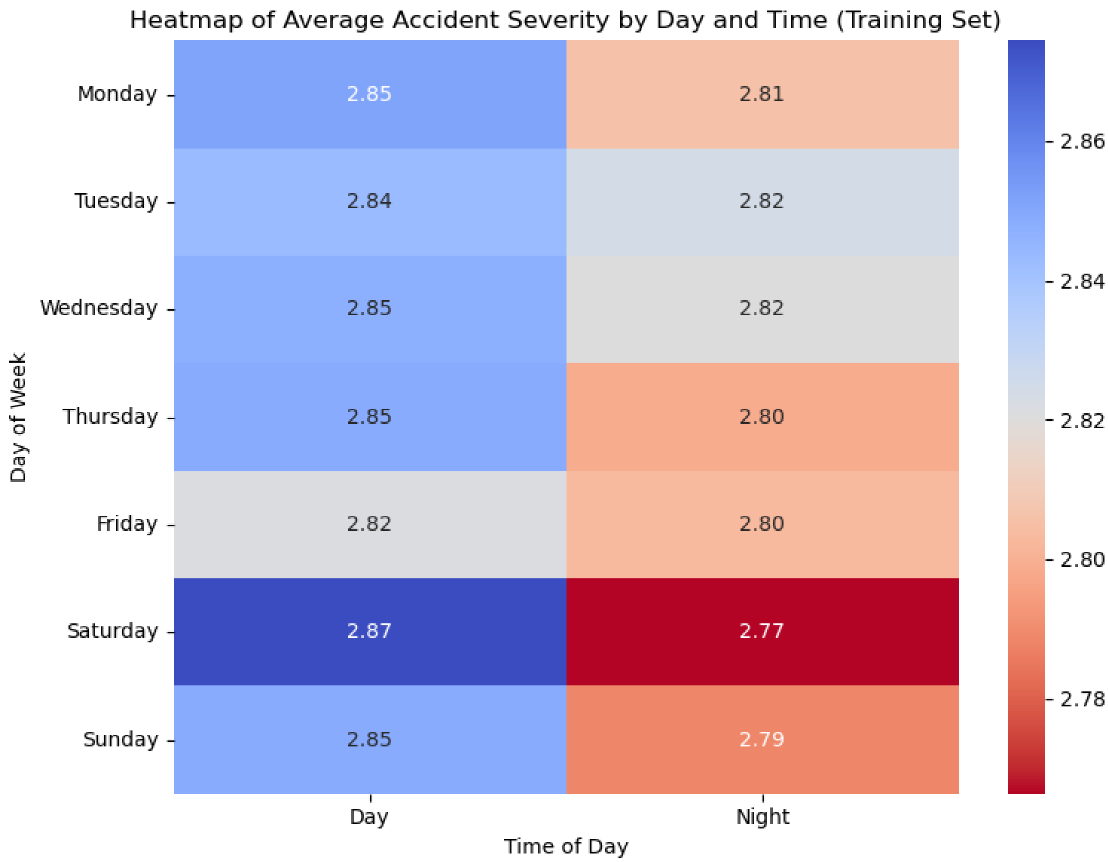
Analysing accidents by time of day and week

[20]:
```python
# Defining names of the days
train_df['day_name'] = train_df['day_of_week'].map({
    1: 'Sunday',
    2: 'Monday',
    3: 'Tuesday',
```

```
    4: 'Wednesday',
    5: 'Thursday',
    6: 'Friday',
    7: 'Saturday'
})
```

For analysis and model development purposes, the time between 7am to 7pm is classified as Day and 7pm onwards up to before 7am is classifed as night time. This allows us to reduce the dummy variables needed during model development.

[21]:
```
# Defining function and apply it to classify time
def classify_day_night(t):
    try:
        h = int(str(t).split(':')[0])
        # If the hour is earlier than 6am or later than 6PM, consider it a night
        return 'Night' if h < 7 or h >= 19 else 'Day'
    except:
        return 'Unknown'

# Applying time of day classification on training data
train_df['time_of_day'] = train_df['hour'].apply(classify_day_night)

# Order by day of week
day_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',␣
 ↪'Saturday', 'Sunday']
train_df['day_name'] = pd.Categorical(train_df['day_name'],␣
 ↪categories=day_order, ordered=True)

# Making a pivot table with average severity
severity_heatmap = train_df.pivot_table(
    index='day_name',
    columns='time_of_day',
    values='accident_severity',
    aggfunc='mean'
)

# Plot heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(severity_heatmap, cmap='coolwarm_r', annot=True, fmt=".2f")
plt.title('Heatmap of Average Accident Severity by Day and Time (Training Set)')
plt.xlabel('Time of Day')
plt.ylabel('Day of Week')
plt.tight_layout()
plt.show()
```



Note: A lower number (1) means a higher accident severity hence red

As expected, Friday - Sunday nights have more severe accidents due to potentially higher 'drinking and driving' cases, or fatigue from night outs by drivers. While taxi drivers themselves may not be drunk, they could still be involved in accidents because of the faults of others

Saturday mornings are much safer, possibly also due to fewer cars on the road which reduce chances of accidents, and hence reduces chances of severe accidents as well.

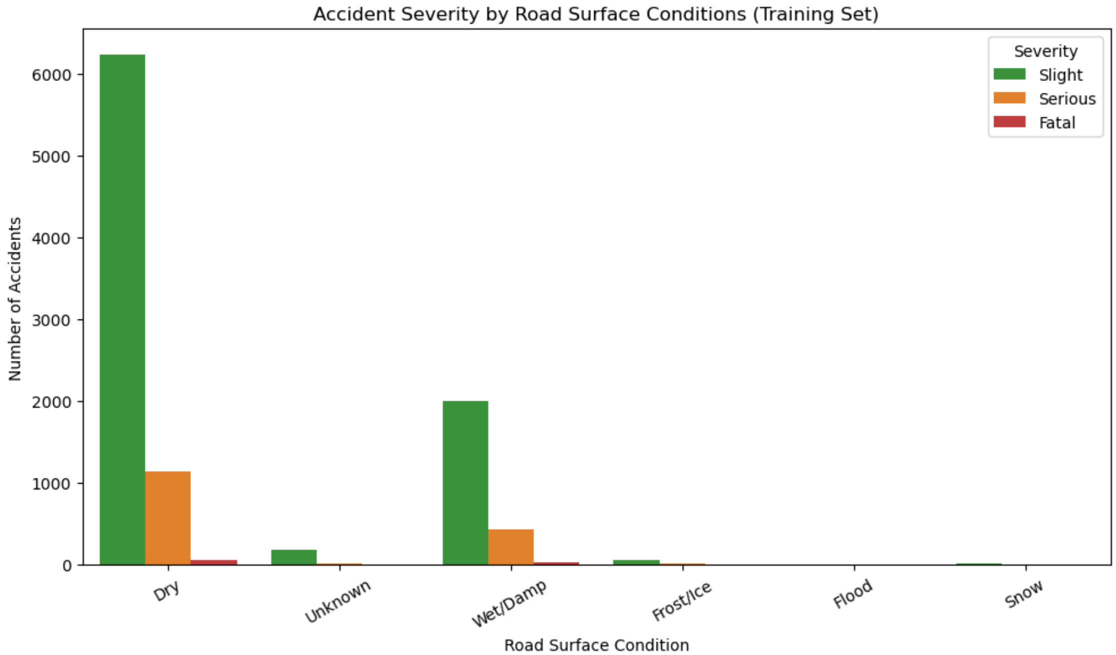
For taxi operators, weekend night shifts should therefore include: 1. Advanced safety checks of vehicles (tyres, lights etc) 2. Real time support systems e.g. dash cams 3. Higher caution and awareness from drivers

[22]:
```
# Applying time of day classification on test data
test_df['time'] = pd.to_datetime(test_df['time'], format = '%H:%M:%S') #␣
↪Ensuring time is categorised correctly
test_df['hour'] = test_df['time'].dt.hour # Extracting the hour
test_df['time_of_day'] = test_df['hour'].apply(classify_day_night)

# We convert this time to numeric as it will help us in outlier detection later␣
↪on
```

```python
train_df['time_of_day'] = train_df['time_of_day'].map({
    'Day': 1, 'Night': 2
})

test_df['time_of_day'] = test_df['time_of_day'].map({
    'Day': 1, 'Night': 2
})
```

1.5.5 5. Accident Severity by road surface conditions

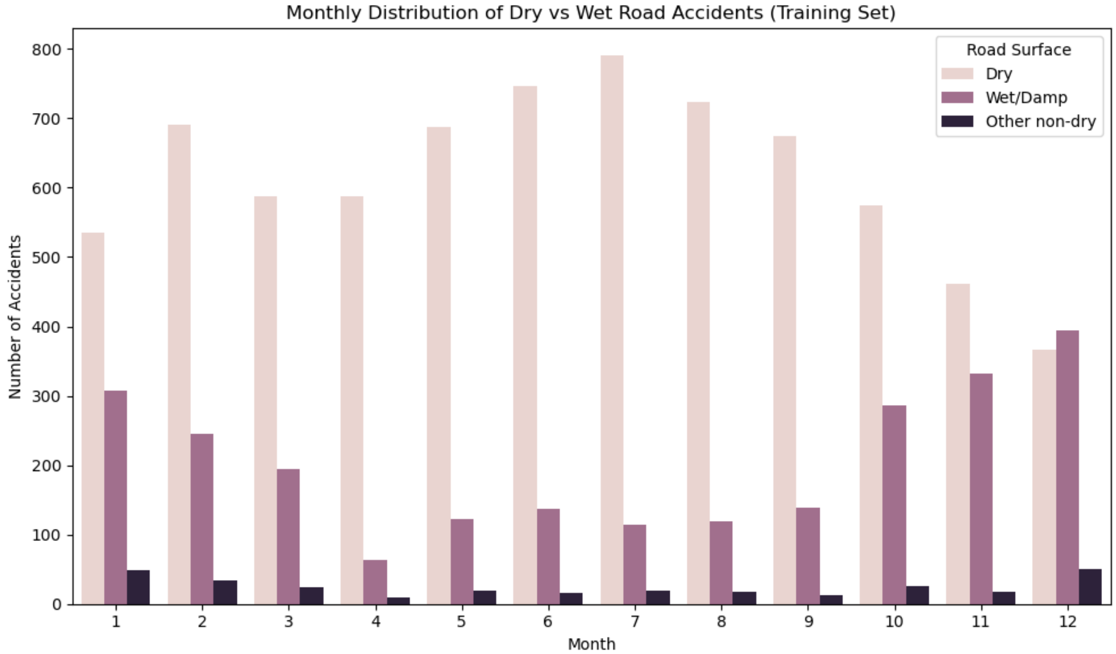
[23]:
```python
# Mapping number notation to text notation of road surface condition

train_df['road_surface_label'] = train_df['road_surface_conditions'].map({
    1: 'Dry',
    2: 'Wet/Damp',
    3: 'Snow', # Originally snow
    4: 'Frost/Ice',
    5: 'Flood',
    6: 'Oil or Diesel',
    7: 'Mud',
    9: 'Unknown'
})

# Setting colours for severity
# These colours will be used throughout for easier interpretation

severity_colors = {
    'Fatal': '#d62728',    # red
    'Serious': '#ff7f0e',  # orange
    'Slight': '#2ca02c'    # green
}

# Plotting the chart

plt.figure(figsize=(10, 6))
sns.countplot(data=train_df, x='road_surface_label',␣
 ↪hue='accident_severity_label', palette = severity_colors)
plt.title("Accident Severity by Road Surface Conditions (Training Set)")
plt.xlabel("Road Surface Condition")
plt.ylabel("Number of Accidents")
plt.xticks(rotation=30)
plt.legend(title="Severity")
plt.tight_layout()
plt.show()
```



[24]:
```
# Apart from Dry and Wet/Damp conditions, other categories are quite minor␣
↪hence will be merged.
# Note that we do this numerically so that we can use it for outlier detection␣
↪at a later stage

train_df['road_surface_label'] = train_df['road_surface_conditions'].map({
    # Key: 1 = Dry, 2 = Wet/Damp, 3 = Other non-dry
    1: 1,
    2: 2,
    3: 3, # Originally snow
    4: 3, # Originally Frost/ice
    5: 3, # Originally Flood
    6: 3, # Originally Oil or Diesel
    7: 3, # Originally Mud
    9: 3 # Originally unknown
})

# The same change is applied on the test data

test_df['road_surface_label'] = test_df['road_surface_conditions'].map({
    # Key: 1 = Dry, 2 = Wet/Damp, 3 = Other non-dry
    1: 1,
    2: 2,
    3: 3, # Originally snow
    4: 3, # Originally Frost/ice
```

```
    5: 3, # Originally Flood
    6: 3, # Originally Oil or Diesel
    7: 3, # Originally Mud
    9: 3 # Originally unknown
})
```

[25]:
```
# We analyse these on a monthly basis

# Defining label names for easier interpretability
custom_labels = ['Dry', 'Wet/Damp', 'Other non-dry']

# Making the plot
plt.figure(figsize=(10, 6))
sns.countplot(
    data=train_df[train_df['road_surface_label'].isin([1, 2, 3])],
    x='month',
    hue='road_surface_label'
)
plt.title("Monthly Distribution of Dry vs Wet Road Accidents (Training Set)")
plt.xlabel("Month")
plt.ylabel("Number of Accidents")
plt.legend(title="Road Surface", labels = custom_labels)
plt.tight_layout()
plt.show()
```



This graph shows consistency with the earlier claim that January and February have higher accidents (and most fatal ones) due to increased wet road conditions.

Although November and December have the highest number of accidents in wet/damp conditions, the number of accidents in dry conditions is low, hence fewer total accidents. This could be due to less people using taxis at this time as people go to their home towns or outside UK for the holidays.

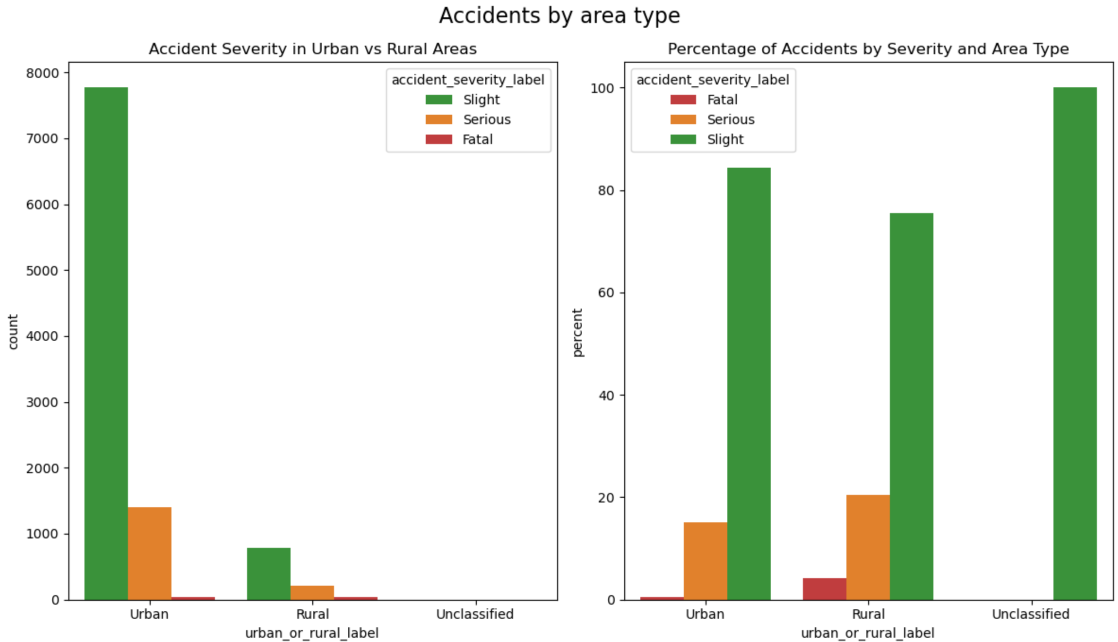
Nonetheless, the severity of wet/damp roads causing accidents is a concern and taxi companies must take extra steps to ensure passenger and driver safety between October - February. These can include regulating driver speeds, having standby emergency services and encouraging drivers to have well maintained cars

1.5.6 6. Urban vs Rural accidents

[26]:
```
# Mapping number notation to text notation of Urban and rural

train_df['urban_or_rural_label'] = train_df['urban_or_rural_area'].map({
    1: 'Urban',
    2: 'Rural',
    3: 'Unclassified'
})

# Because of the unequal distribution of accidents, mostly in urban areas due␣
↪to higher car numbers,
# we analyse by percentage as well

# Creating percentage-based DataFrame from training set
plot_df = train_df.groupby(['urban_or_rural_label', 'accident_severity_label']).
↪size().reset_index(name='count')
total_by_area = plot_df.groupby('urban_or_rural_label')['count'].
↪transform('sum')
plot_df['percent'] = plot_df['count'] / total_by_area * 100
```

[27]:
```
#Creating the plots
fig, axs = plt.subplots(1, 2, figsize=(12, 7), sharex=True)
fig.suptitle('Accidents by area type', fontsize=16)

# Plot of accidents by area type
sns.countplot(data=train_df, x='urban_or_rural_label',␣
↪hue='accident_severity_label',
              palette=severity_colors, ax=axs[0]
)
axs[0].set_title("Accident Severity in Urban vs Rural Areas")

# Plot of percentage of accidents by area type
sns.barplot(data=plot_df, x='urban_or_rural_label', y='percent',␣
↪hue='accident_severity_label',
    palette=severity_colors, ax=axs[1]
)
```

```
axs[1].set_title("Percentage of Accidents by Severity and Area Type")

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



Note: There is only 1 unclassified entry, whose severity is slight. This is an outlier that may be automatically removed during the outlier detection stage.

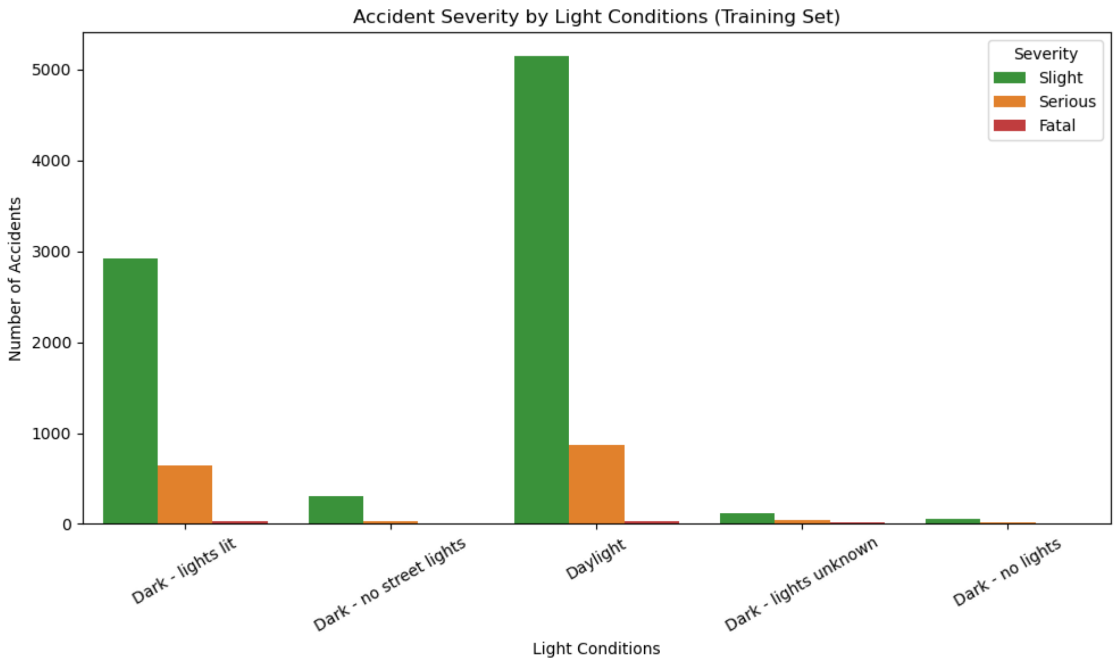
While most fatal accidents happen in urban areas, the percentage of fatal accidents ([fatal accidents/all accidents]*100) in rural areas is much higher. This may be because urban areas have lower speed limits hence lowering the chances of fatal accidents.

Taxi operators can therefore set their own speed limits for drivers driving in higher risk rural areas

1.5.7 7. Accidents by light conditions

[28]:
```
# Mapping number notation to text notation of Urban and rural
train_df['light_condition_label'] = train_df['light_conditions'].map({
    1: 'Daylight',
    4: 'Dark - lights lit',
    5: 'Dark - no lights',
    6: 'Dark - lights unknown',
    7: 'Dark - no street lights'
})

# Defining the plot
plt.figure(figsize=(10, 6))
```

```
sns.countplot(data=train_df, x='light_condition_label',␣
↪hue='accident_severity_label', palette = severity_colors)
plt.title("Accident Severity by Light Conditions (Training Set)")
plt.xlabel("Light Conditions")
plt.ylabel("Number of Accidents")
plt.xticks(rotation=30)
plt.legend(title="Severity")
plt.tight_layout()
plt.show()
```



While most accidents do happen in the day time when most cars are on the road, as a percentage, most serious and fatal accidents happen in the dark when visibility is lower.

Since the second and last 2 categories are minor, we combine them for simpler analysis during model development

[29]:
```
# Combining common values on train data
# We do this numerically so as to use in outlier prediction later on

train_df['light_condition_label'] = train_df['light_conditions'].map({
    # Key: 1 = Daylight, 2 = Dark - Lights lit, 3 = Dark - Bad lighting
    1: 1,
    4: 2,
    5: 3, # Originally 'Dark - no lights'
    6: 3, # Originally 'Dark - lights unknown'
    7: 3 # Originally 'Dark - no street lights'
```

```
})

# Combining common values on test data

test_df['light_condition_label'] = test_df['light_conditions'].map({
    1: 1,
    4: 2,
    5: 3, # Originally 'Dark - no lights'
    6: 3, # Originally 'Dark - lights unknown'
    7: 3 # Originally 'Dark - no street lights'
})
```

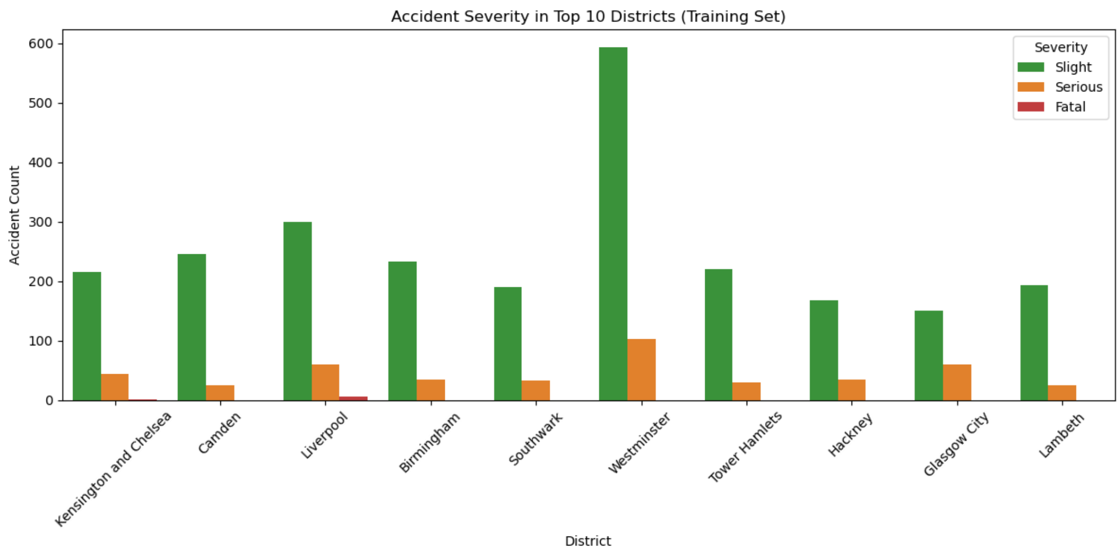
1.5.8 8. Location analysis

[30]:
```
# Finding top accident locations
# Loading data file with all location codes and names
# This was sourced from the lookup file provided by the Department of␣
↪Transportation, and grouped based on their region groupings
location_lookup = pd.read_excel('location_lookup.xlsx', usecols =␣
↪['local_authority_ons_district', 'Location_name','Region', 'region_number'])

# Joining location names
train_df = pd.merge(train_df,location_lookup,␣
↪on='local_authority_ons_district', how = 'left')


#The key for region and region number is summarised below
print(location_lookup.drop_duplicates(subset=['Region',␣
↪'region_number'])[['Region', 'region_number']])
```

```
                       Region  region_number
0                        East              1
55              East Midlands              2
92                     London              3
127                North East              4
141                North West              5
176                  Scotland              6
214                South East              7
304                South West              8
356                   Unknown              0
357                     Wales              9
378             West Midlands             10
408  Yorkshire and The Humber             11
```

[31]:
```
# Finding top 10 locations with most accidents
top10_districts = train_df['Location_name'].value_counts().nlargest(10).index
subset = train_df[train_df['Location_name'].isin(top10_districts)]
```

```
# Plot
plt.figure(figsize=(12, 6))
sns.countplot(data=subset, x='Location_name', hue='accident_severity_label',␣
 ↪palette = severity_colors)
plt.title("Accident Severity in Top 10 Districts (Training Set)")
plt.xlabel("District")
plt.ylabel("Accident Count")
plt.xticks(rotation=45)
plt.legend(title='Severity')
plt.tight_layout()
plt.show()
```



While Westminister has some of the highest accident numbers, it is more interesting to see how Liverpool has the highest number of fatal accidents. No other district comes close to this count which means Liverpool is a very high risk zone (it also has the second highest total accidents).

Taxi operators may have to price these areas accordingly due to possibly higher insurance rates in these areas.

Operators may also have to train drivers in these areas more to avoid fatal accidents

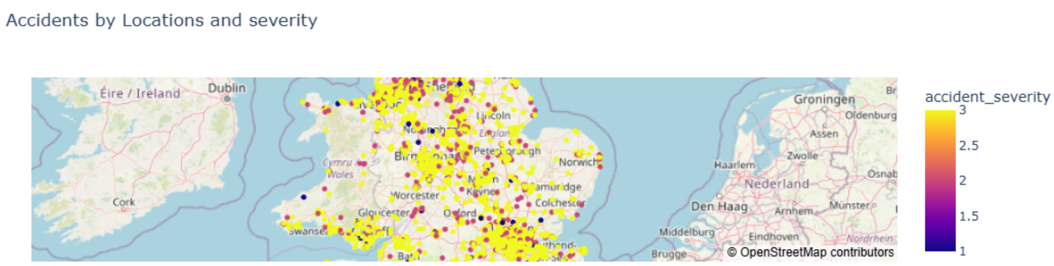
Note: We added the columns for location name and region. Since regions are fewer, there are simpler for categorisation. We add these regions to the test data as well (and eventually will drop ONS code and location name columns)

[32]:
```
# Making the same join with test data

# Loading data file with all location codes and names
```

```
location_lookup = pd.read_excel('location_lookup.xlsx', usecols =␣
↪['local_authority_ons_district','region_number'])

# Joining location names
test_df = pd.merge(test_df,location_lookup, on='local_authority_ons_district',␣
↪how = 'left')
```

1.5.9 Plotting accident locations

[33]:
```
fig = px.scatter_mapbox(
    train_df,
    lat="latitude",
    lon="longitude",
    color = "accident_severity",
    zoom=5,
    height=600,
    mapbox_style="open-street-map",
    title="Accidents by Locations and severity"
)

fig.show()
```



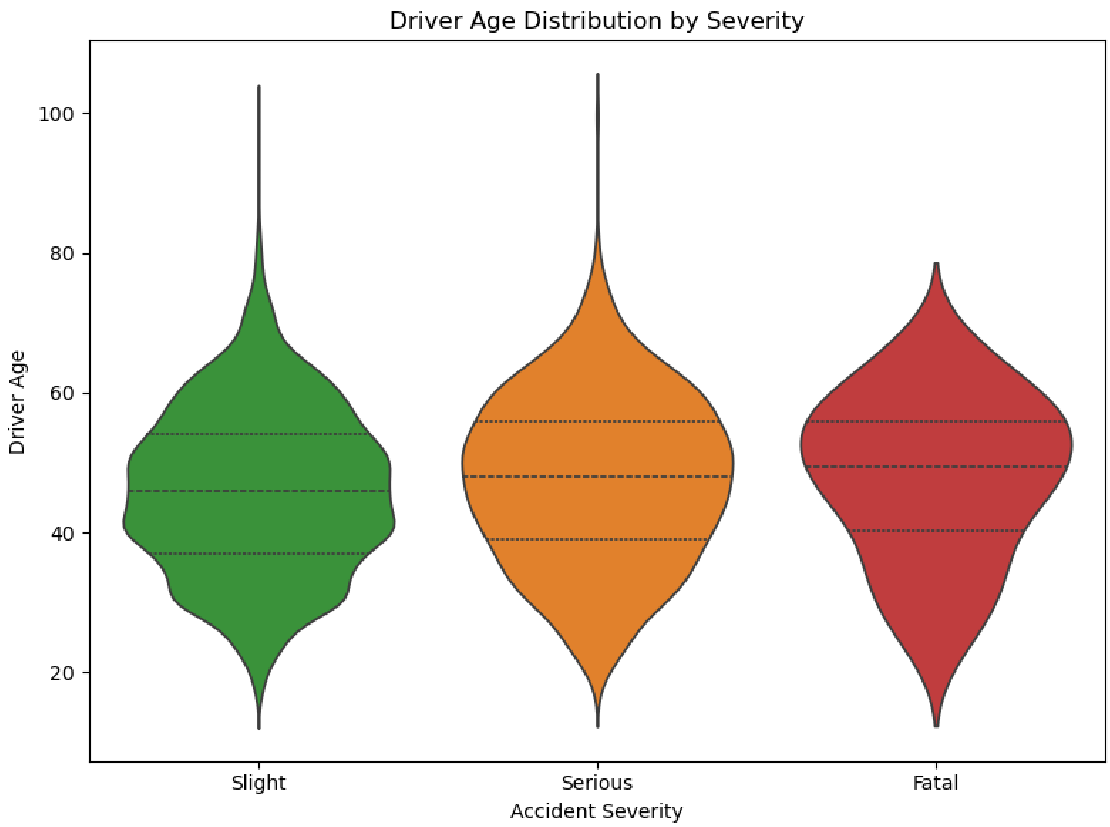
It appears that most severe accidents tend to occue on the outskirts of cities and less within the city. This could be due to higher traffic in the city, or lower speed limits within the city.

Operators can flag these areas and encourage drivers to use within city routes. They could also set up stricter monitoring such as speed alerts for drivers leaving the city to maximise safety

1.5.10 9. Driver Age distribution

[34]:
```
# Plotting a violin plot to analyse age

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

```
sns.violinplot(x='accident_severity_label', y='age_of_driver', data=train_df,␣
 ↪inner='quartile', palette = severity_colors)
plt.title('Driver Age Distribution by Severity')
plt.xlabel('Accident Severity')
plt.ylabel('Driver Age')
plt.tight_layout()
plt.show()
```



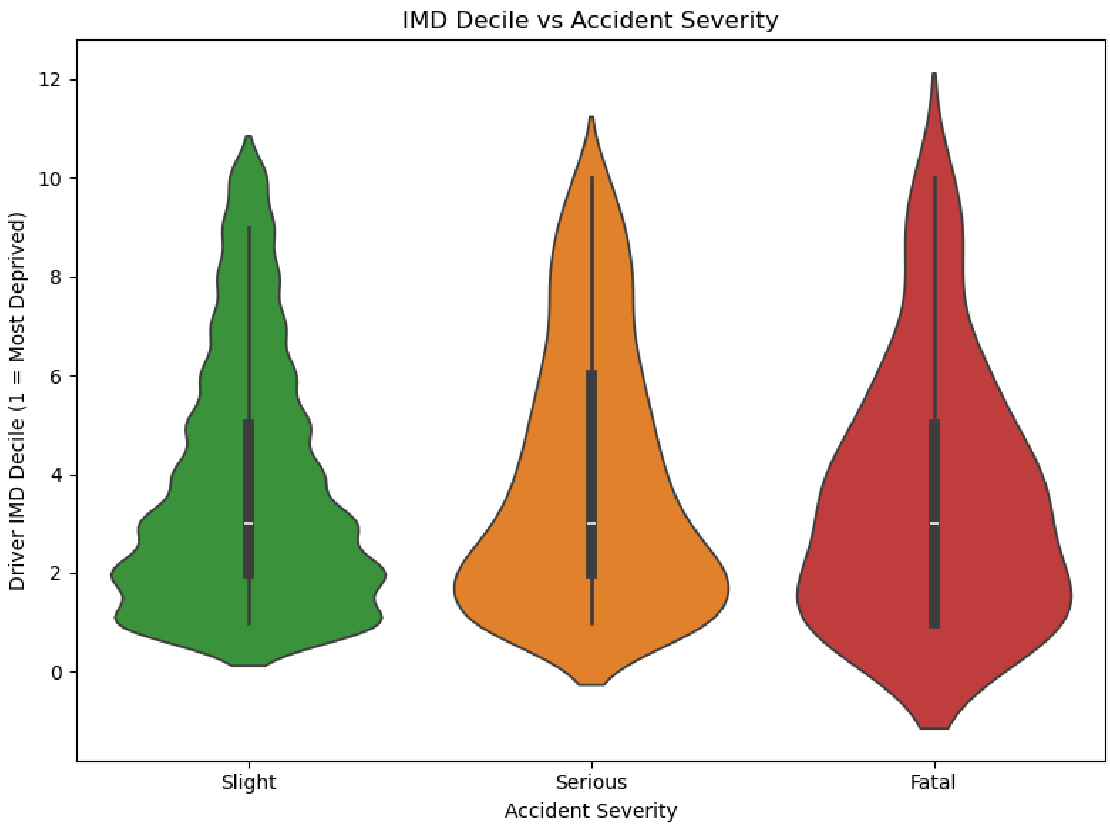
The median age of taxi drivers is around 45 across all severities.

Significant outliers are noted with slight and Serious accidents. There are much older drivers over the age of 80 still driving taxis, but they are less likely to get into fatal accidents (almost none in the past 5 years). However, fatal accidents often involve people aged between 50-60 possibly due to slower reaction times when driving and faced with emergencies.

1.5.11 10. Accident analysis by IMD Decile

[35]:
```
plt.figure(figsize=(8, 6))
sns.violinplot (x='accident_severity_label', y='driver_imd_decile',␣
 ↪data=train_df, inner='box',
                palette = severity_colors)
```

```
plt.title("IMD Decile vs Accident Severity")
plt.xlabel("Accident Severity")
plt.ylabel("Driver IMD Decile (1 = Most Deprived)")
plt.tight_layout()
plt.show()
```



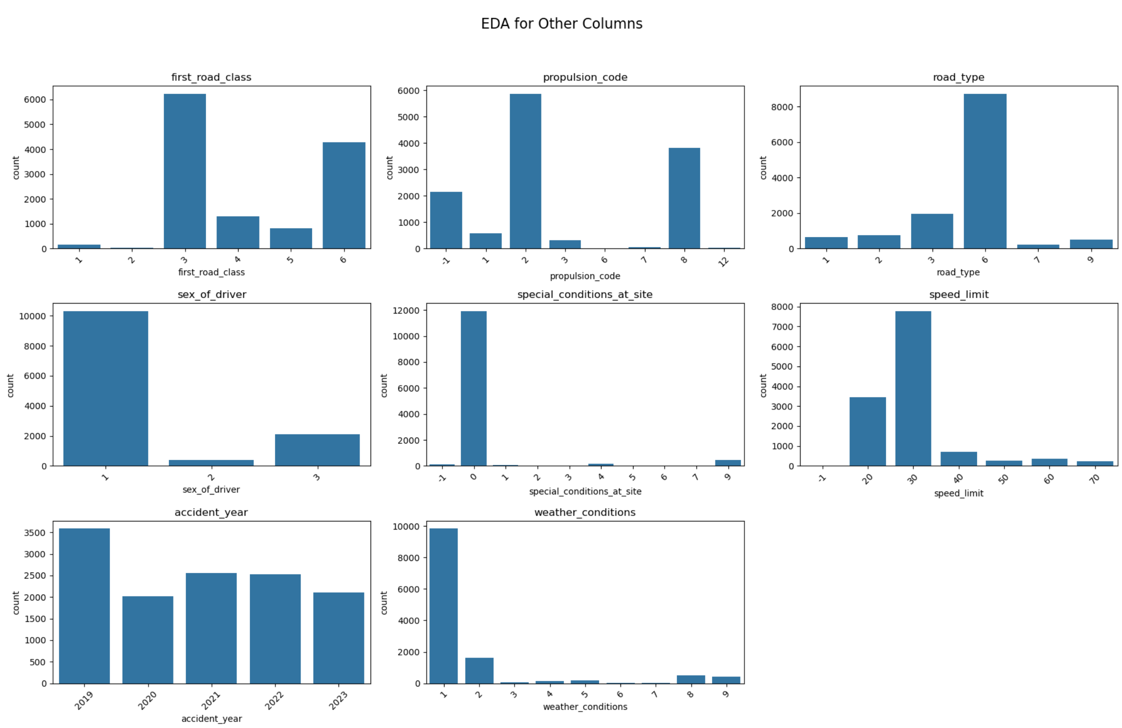
IMD Decile refers to the Index of Multiple Deprivation Deciles. This shows how deprived the drivers are (and can indicate their poverty levels).

All the plots show a right skew. This suggests that the lower the IMD decile of the taxi driver, the higher the chances of an accident. This could be due to: 1. Fatigue caused by driving longer hours (as a result of economic pressure) 2. Poorly maintained cars 3. Higher exposure to dangerous environments

Taxi potentially avoid accidents for lower IMD decile drivers by providing better work hours, and restricting number of hours worked to avoid fatigue which can result into accidents

1.6 Summarizing all other features

[36]:
```
# Other columns to visualize
cols = [
    'first_road_class', 'propulsion_code', 'road_type', 'sex_of_driver',
     'special_conditions_at_site', 'speed_limit', 'accident_year',␣
  ↪'weather_conditions'

]

# Create subplots
fig, axes = plt.subplots(3, 3, figsize=(18, 12))
axes = axes.flatten()

# Plot each column
for i, col in enumerate(cols):
    sns.countplot(x=df[col], ax=axes[i])
    axes[i].set_title(col)
    axes[i].tick_params(axis='x', rotation=45)

# Hide any unused axes
for j in range(i + 1, len(axes)):
    fig.delaxes(axes[j])

plt.suptitle("EDA for Other Columns", fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



1.7 Further pre-processing

From the above summary, any features which have categorical options with few values can be grouped to simplify the model.

Note that we leave the numeric nature of the values since outlier detection will require values to be in numeric form

[40]:
```
# Making the different maps to replace values with

weather_condition_map = {
    # Key: 1 = Fine, 2 = Raining, 3 = Snowing, 4 = Other
    1: 1, # Originally 'Fine no high winds'
    2: 2, # Originally 'Raining - no high winds'
    3: 3, # Originally 'Snowing'
    4: 1, # Originally 'Fine + High winds'
    5: 2, # Originally 'Raining + High Winds'
    6: 3, # Originally 'Snowing + High Winds'
    7: 4, # Originally 'Fog or mist'
    8: 4, # Originally 'Other'
    9: 4 # Originally 'Unknown'
}

speed_limit_map = {
    # Key: 1 = 20, 2 = 30, 3 = 40, 4 = 50+
    20: 1,
    30: 2,
    40: 3,
    50: 4,
    60: 4,
    70: 4
}

special_conditions_at_site_map = {
    # Key: 1 = None, 2 = Road Furniture Issue, 3 = Other
    0: 1, # Originally 'None'
    1: 2, # Originally 'Auto traffic signal - out'
    2: 2, # Originally 'Auto signal part defective'
    3: 2, # Originally 'Road sign or marking defective or obscured'
    4: 3, # Originally 'Road works'
    5: 3, # Originally 'Road surface defective'
    6: 3, # Originally 'Oil or Diesel'
    7: 3, # Originally 'Mud'
    9: 3 # Originally 'Unknown (self reported)'
}
```

```
propulsion_code_map = {
    # Key: 1 = Traditional (Gas/oil), 2 = Electric, 3 = Hybrid
    1: 1, # Originally 'Petrol'
    2: 1, # Originally 'Heavy Oil'
    3: 2, # Originally 'Electric'
    6: 1, # Petrol'Gas'
    7: 1, # Originally 'Gas/Bi-fuel'
    8: 3, # Originally 'Hybrid'
    12: 3 # Originally 'Electric/Diesel'
}
```

[41]:
```
# Function to make the replacements , to avoid duplication of map

def apply_mapping(df, source_col, new_col_name, new_values):
    """
    Maps codes to readable labels or merges values and adds them as a new␣
 ↪column.

    Parameters:
        df: The DataFrame to process
        source_col: Name of the column with current values
        new_col_name: Name of the new column to add mapped values
        new_values: Map with the new values

    Returns:
        pd.DataFrame: Updated DataFrame with new weather label column
    """
    df[new_col_name] = df[source_col].map(new_values)
```

[42]:
```
# Applying the transformation on train and test data

# note, for columns with missing values, the transformation will be done after␣
 ↪imputing missing values

apply_mapping(train_df, 'weather_conditions', 'weather_conditions_grouped',␣
 ↪weather_condition_map)
apply_mapping(test_df, 'weather_conditions', 'weather_conditions_grouped',␣
 ↪weather_condition_map)
```

1.7.1 Clearing out unwanted columns

We clear out any columns that have become redundant due to transformations made, or are not useful for analysis

[43]:

```
train_df.drop(['accident_reference', 'longitude',␣
↪'latitude','urban_or_rural_label',
             'day_name', 'time', 'vehicle_type', 'light_conditions',␣
↪'weather_conditions', 'Region',
            'road_surface_conditions', 'accident_severity_label', 'date',␣
↪'local_authority_ons_district', 'Location_name'], axis=1, inplace=True)
```

[44]:
```
# Applying the same on test_df
test_df.drop(['accident_reference', 'longitude', 'latitude',
             'time', 'vehicle_type', 'light_conditions', 'weather_conditions',
            'road_surface_conditions', 'date',␣
↪'local_authority_ons_district'], axis=1, inplace=True)
```

1.7.2 Dealing with blank values

Most of the data in this data set is categorical hence using KNN would be a good way to impute blank values

[45]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10293 entries, 0 to 10292
Data columns (total 19 columns):
 #   Column                      Non-Null Count  Dtype
---  ------                      --------------  -----
 0   day_of_week                 10293 non-null  int64
 1   first_road_class            10293 non-null  int64
 2   road_type                   10293 non-null  int64
 3   speed_limit                 10291 non-null  float64
 4   special_conditions_at_site  10212 non-null  float64
 5   urban_or_rural_area         10293 non-null  int64
 6   accident_year               10293 non-null  int64
 7   sex_of_driver               10293 non-null  int64
 8   age_of_driver               8281 non-null   float64
 9   driver_imd_decile           7936 non-null   float64
 10  propulsion_code             8572 non-null   float64
 11  accident_severity           10293 non-null  int64
 12  month                       10293 non-null  int32
 13  hour                        10293 non-null  int32
 14  time_of_day                 10293 non-null  int64
 15  road_surface_label          10224 non-null  float64
 16  light_condition_label       10293 non-null  int64
 17  region_number               10289 non-null  float64
 18  weather_conditions_grouped  10293 non-null  int64
dtypes: float64(7), int32(2), int64(10)
memory usage: 1.4 MB
```

[46]:
```python
#Using KNN imputer to deal with missing value
from sklearn.impute import KNNImputer, SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
import pandas as pd
import numpy as np
```

When imputing missing values, numerical columns will be scaled. The scaled columns will also be used for outlier detection. However, some models require unscaled variables hence the scaled variables will be saved as new columns and deleted before model creation

[47]:
```python
numeric_cols = ['age_of_driver', 'hour']

scaler = StandardScaler()

# We run the scaler on train set first
scaled_values = scaler.fit_transform(train_df[numeric_cols])

# We make new columns for scaled values to preserve original columns
scaled_col_names = [col + '_scaled' for col in numeric_cols]

# We assign scaled values to new columns
train_df[scaled_col_names] = scaled_values

# The same is done for the test set using the same scaler we ran for the test
test_df[scaled_col_names] = scaler.transform(test_df[numeric_cols])
```

[49]:
```python
# We can now perform the KNN

# 1.  defining columns we want to use for the imputation
numeric_cols = ['age_of_driver_scaled', 'hour_scaled']
categorical_cols = [
    'day_of_week', 'first_road_class', 'road_type',
    'speed_limit','special_conditions_at_site',
    'urban_or_rural_area', 'accident_year', 'sex_of_driver',
    'age_of_driver', 'driver_imd_decile', 'propulsion_code', 'month',
    'time_of_day', 'road_surface_label', 'light_condition_label',␣
  ↪'weather_conditions_grouped', 'region_number'
]

# 2.  Creating a pipeline for each group
numeric_pipeline = Pipeline([
    ('knn_imp', KNNImputer(n_neighbors=5)),   #  KNN imputation for numerical␣
  ↪features
])
```

```
categorical_pipeline = Pipeline([
    ('mode_imp', SimpleImputer(strategy='most_frequent'))  #  mode imputation␣
 ↪for categorical
])

# 3. Combine with ColumnTransformer
preprocessor = ColumnTransformer([
    ('num', numeric_pipeline,   numeric_cols),
    ('cat', categorical_pipeline, categorical_cols),
], remainder='passthrough') # Keep other columns

# 4. Fit on train and transform on test
trainset_imputed_arr= preprocessor.fit_transform(train_df)
testset_imputed_arr= preprocessor.transform(test_df)

# 5. Convert back to DataFrame with correct column names
# Get the column names after transformation
transformed_cols = []
for name, _, cols in preprocessor.transformers_:
    if name == 'num':
        transformed_cols.extend(cols)
    elif name == 'cat':
        transformed_cols.extend(cols)
    elif name == 'remainder':
        # Handle columns not in numeric_cols or categorical_cols
        # This assumes the order of columns in X_train and X_test is preserved␣
 ↪for remainder
        original_cols = train_df.columns.tolist()
        processed_cols = numeric_cols + categorical_cols
        remainder_cols = [col for col in original_cols if col not in␣
 ↪processed_cols]
        transformed_cols.extend(remainder_cols)


train_df = pd.DataFrame(trainset_imputed_arr, columns=transformed_cols,␣
 ↪index=train_df.index)
test_df = pd.DataFrame(testset_imputed_arr,  columns=transformed_cols,␣
 ↪index=test_df.index)


# 6. checking for missing value
print("Missing in train:\n", train_df.isna().sum())
print("Missing in test:\n",  test_df.isna().sum())
```

```
Missing in train:
 age_of_driver_scaled          0
hour_scaled                   0
```

```
day_of_week                  0
first_road_class             0
road_type                    0
speed_limit                  0
special_conditions_at_site   0
urban_or_rural_area          0
accident_year                0
sex_of_driver                0
age_of_driver                0
driver_imd_decile            0
propulsion_code              0
month                        0
time_of_day                  0
road_surface_label           0
light_condition_label        0
weather_conditions_grouped   0
region_number                0
accident_severity            0
hour                         0
dtype: int64
Missing in test:
 age_of_driver_scaled          0
hour_scaled                  0
day_of_week                  0
first_road_class             0
road_type                    0
speed_limit                  0
special_conditions_at_site   0
urban_or_rural_area          0
accident_year                0
sex_of_driver                0
age_of_driver                0
driver_imd_decile            0
propulsion_code              0
month                        0
time_of_day                  0
road_surface_label           0
light_condition_label        0
weather_conditions_grouped   0
region_number                0
accident_severity            0
hour                         0
dtype: int64
```

[50]:

```
# Now that there are no missing values, we can transform the remaining␣
↪categories
```

```
# Transforming speed limits
apply_mapping(train_df, 'speed_limit', 'speed_limit_grouped', speed_limit_map)
apply_mapping(test_df, 'speed_limit', 'speed_limit_grouped', speed_limit_map)

# Transforming speed limits
apply_mapping(train_df, 'speed_limit', 'speed_limit_grouped', speed_limit_map)
apply_mapping(test_df, 'speed_limit', 'speed_limit_grouped', speed_limit_map)

# Transforming special conditions at site
apply_mapping(train_df, 'special_conditions_at_site',␣
↪'special_conditions_at_site_grouped', special_conditions_at_site_map)
apply_mapping(test_df, 'special_conditions_at_site',␣
↪'special_conditions_at_site_grouped', special_conditions_at_site_map)

# Transforming propulsion code
propulsion_code_map
apply_mapping(train_df, 'propulsion_code', 'propulsion_code_grouped',␣
↪propulsion_code_map)
apply_mapping(test_df, 'propulsion_code', 'propulsion_code_grouped',␣
↪propulsion_code_map)

# Dropping the older columns
train_df.drop(['speed_limit', 'special_conditions_at_site', 'propulsion_code'],␣
↪axis=1, inplace=True)
test_df.drop(['speed_limit', 'special_conditions_at_site', 'propulsion_code'],␣
↪axis=1, inplace=True)
```

2 OUTLIER HANDLING

We used Isolation Forest for multivariate outlier detection, as it can capture anomalies across combinations of features (e.g., driver age and light condition). We used 3% contamination based on visual inspection of feature distributions.

[51]:
```
# Redefine X_train and y_train

X_train = train_df.drop(columns=['accident_severity'])
y_train = train_df['accident_severity']

X_test = test_df.drop(columns=['accident_severity'])
y_test = test_df['accident_severity']
```

[52]:
```
X_train.shape, X_test.shape
```

[52]:
```
((10293, 20), (2571, 20))
```

[53]:
```python
from sklearn.ensemble import IsolationForest

# Train Isolation Forest
clf = IsolationForest(n_estimators=100, random_state=7, contamination=0.03)
clf.fit(X_train)
```

[53]:
```
IsolationForest(contamination=0.03, random_state=7)
```

[54]:
```python
# Remove outliers from training set
yhat_train = clf.predict(X_train)
X_train = X_train[yhat_train == 1]
y_train = y_train[yhat_train == 1]

yhat_train.shape
```

[54]:
```
(10293,)
```

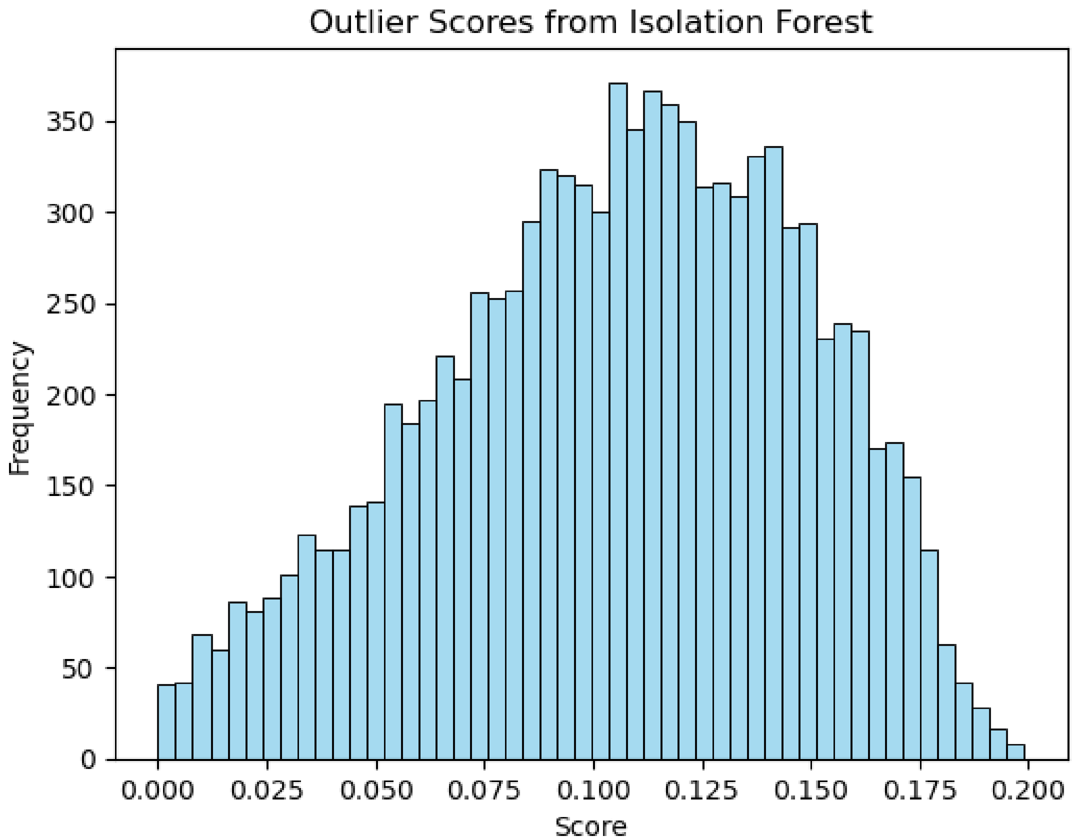
[55]:
```python
# Remove outliers from test set
yhat_test = clf.predict(X_test)
X_test = X_test[yhat_test == 1]
y_test = y_test[yhat_test == 1]

yhat_test.shape
```

[55]:
```
(2571,)
```

[56]:
```python
print("Train outliers removed:", sum(yhat_train == -1))
print("Test outliers removed:", sum(yhat_test == -1))
```

```
Train outliers removed: 309
Test outliers removed: 72
```

[57]:
```python
# Outlier Score Histogram

scores = clf.decision_function(X_train)

sns.histplot(scores, bins=50, color='skyblue')
plt.title("Outlier Scores from Isolation Forest")
plt.xlabel("Score")
plt.ylabel("Frequency")
plt.show()
```



The histogram shows a clean, symmetric score distribution after outlier removal, suggesting that Isolation Forest effectively filtered extreme anomalies.

3 Handling the imbalanced dataset

[58]:
```
y_train.value_counts()
```

[58]:
```
accident_severity
3.0    8328
2.0    1574
1.0      82
Name: count, dtype: int64
```

The data is clearly imbalanced. However, different modelling techniques may handle imbalanced values differently.

To reduce some imbalance without compromising results quality, category 1 and 2 will be combined as severe accidents (these are severe and fatal).

[59]:
```
# Convert labels to binary (1, 2 → 0 = Severe; 3 → 1 = Not Severe)
y_train = y_train.apply(lambda x: 0 if x in [1, 2] else 1)
y_test = y_test.apply(lambda x: 0 if x in [1, 2] else 1)

# This simplifies prediction
```

[60]:
```
y_train.value_counts()
```

[60]:
```
accident_severity
1    8328
0    1656
Name: count, dtype: int64
```

While it has been reduced, there is still some imbalance.

SMOTE was tested on this dataset to make it more balanced but was determined less useful as it creates decimal values for categorical variables. This made it an impractical choice

The portion of imbalanced handling is therefore done in the model development file.

4 CONCLUSION

In this project, we successfully prepared a comprehensive dataset of UK taxi accidents by combining collision and vehicle information. Through EDA, preprocessing, outlier and imbalance handling, and feature selection, we built a refined dataset ready for modeling.

Key insights for the taxi industry include:

- Fatal accidents are more likely to occur in rural areas
- Liverpool is quite a risky zone with the highest number of fatal accidents. Westminister has the highest accidents.
- The season from October to February is a high risk period with most accidents happening in this period due to wet roads
- Accidents peak during mornign rush hours, evening rush hours and around midnight. Weekends are more likely to have severe accidents.

In the next stage, a predictive model is developed to estimate the severity of taxi collisions before a trip begins, helping operators make safer and more cost-effective decisions.

[61]:
```
# Combining X and y values
train_set = pd.concat([X_train, y_train], axis=1)
test_set = pd.concat([X_test, y_test], axis=1)

# Exporting datasets to CSV for use in individual assignments
train_set.to_csv('processed_train_data.csv', index=False)
test_set.to_csv('processed_test_data.csv', index=False)
```