Introduction :

The purpose of this project is to assess the current issue and provide insights and preventative strategies for traffic accidents and road safety in the UK using Big Data analytical techniques. The dataset was obtained from Kaggle and contains more than 500 thousand traffic accident logs, and they were collected by the UK Department of Transport.

1. Background and Implementation :

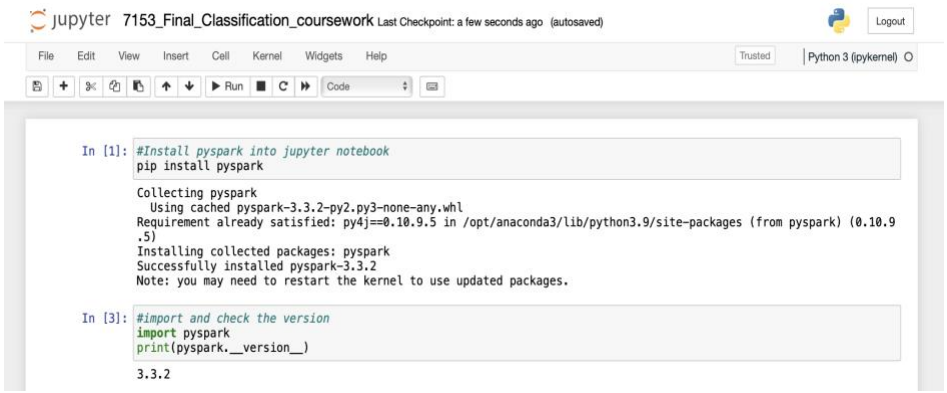
1.1 PySpark Installation

- It is necessary to have java version 8 or higher in the system to use Pyspark.

Figure 1
Check Java Version

- Pyspark can be installed in Jupyter Notebook using pip install command. This will download and install the PySpark package and its dependencies, which will enable us to use PySpark in the Jupyter Notebook. Once installed, we can import the pyspark module which provides an interface for working with Apache Spark using Python.

Figure 2
Install PySpark



- In order to verify that spark is operating well in Jupyter Notebook, we can now setup a spark session and a create a simple RDD.

master('local[*]'): This sets the master URL to run Spark in local mode with all available cores for local development and testing (Spark, n.d.).

appName('spark_coursework'): This sets the name of the Spark application to "spark_coursework".

getOrCreate(): This creates a new SparkSession or returns an existing one if one already exists.

Figure 3
Create Spark Session

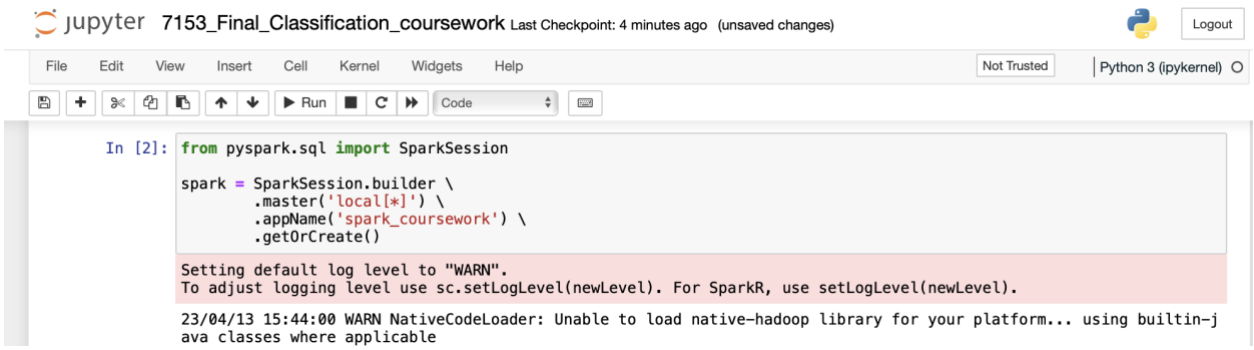
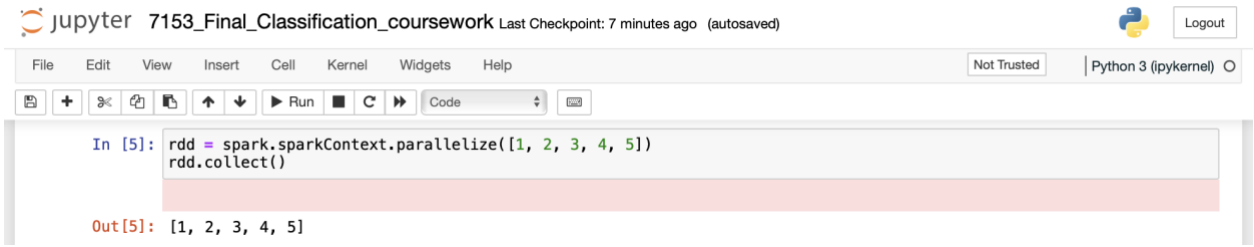


Figure 4
Test spark



1.2 Dataset

The dataset includes statistics on UK traffic accidents that occurred between 2016 and 2020. The information is sourced from the UK government's Open Data website, where it was made available by the Department of Transportation (data.gov.uk, 2022) and shared on Kaggle (Kaggle, 2022). The dataset comprises of three csv files:

1) **Accident**: This is the primary dataset and each line in the file represents a distinct traffic accident, which is identified by the accident_index column. It contains details like road condition, location of the accident, weather condition, did police attend the site etc.

The 3 types of accident severity are:

Figure 5
Accident Severity

Accident Severity	Description
1	Fatal
2	Serious
3	Slight

2) **Vehicle**: Each line in the file depicts the participation of a distinct vehicle in a distinct traffic accident, including columns for different vehicle and passenger features.

3) **Casualty Information**: Each line in the file includes information about the victims, such as whether they were pedestrians or passengers in a car, how seriously they were hurt, etc.

The categorical columns of the datasets are already encoded into digits, and the mapping codelist sheet has been shared by the Department of Transportation :Road-Safety-Open-Dataset-Data-Guide.xlsx (data.gov.uk, n.d.). Accident_index is the unique key used to inner join the three csv files.

Figure 6
Accidents

Column Name	Data Type	Description
accident_index	string	Unique identifier of the accident.
accident_year	integer	Year of the accident.
accident_reference	string	Used by the police to reference a collision. It is not unique outside of the year.
location_easting_osgr	geolocation	Ordnance Survey Grid Reference
location_northing_osgr	geolocation	Ordnance Survey Grid Reference
longitude	geolocation	Longitude location
latitude	geolocation	Latitude location
police_force	integer	Police force number of the area
accident_severity	integer	Severity of the accident
number_of_vehicles	integer	Number of vehicles involved in the accident
number_of_casualties	integer	Number of casualties
date	string	Date
day_of_week	integer	Day of the week
time	timestamp	Time
local_authority_district	integer	Local district number
local_authority_ons_district	integer	ONS district code
local_authority_highway	integer	Highway number
first_road_class	integer	important routes connecting large population centers, or for through traffic - Class I, and
first_road_number	integer	Class I road number
road_type	integer	Type of road - Dual carriageway, single carriageway, slip road etc.
speed_limit	integer	Speed limit
junction_detail	integer	Junction details

junction_control	integer	Authorized person, signal, stop sign etc.
second_road_class	integer	roads of lesser importance are designated as Class II.
second_road_number	integer	Class II road number
pedestrian_crossing_human_control	integer	Details of pedestrian control like school crossing patrol, control by person etc.
pedestrian_crossing_physical_facilities	integer	Zebra, Pelican, traffic signal etc.
light_conditions	integer	Light conditions
weather_conditions	integer	Weather conditions
road_surface_conditions	integer	Road surface conditions
carriageway_hazards	integer	Hazards on the road like Dog on road, vehicle on road, previous accident etc.
urban_or_rural_area	integer	Area type
did_police_officer_attend_scene_of_accident	integer	Yes/No
trunk_road_flag	integer	Trunk/ Non-trunk
lsoa_of_accident_location	string	England and Wales only. See Office for National Statistics (ONS) guidance: https://www.ons.gov.uk/methodology/geography/ukgeographies/censusgeography

Figure 7
Casualty

Column Name	Data Type	Description
accident_index	string	Unique identifier of the accident.
accident_year	integer	Year of the accident.
accident_reference	string	Used by the police to reference a collision. It is not unique outside of the year.
vehicle_reference	integer	Unique value for each vehicle in a singular accident.
casualty_reference	integer	unique value for each casualty in a singular accident.
casualty_class	integer	Description of type of casualty
sex_of_casualty	integer	Male/Female
age_of_casualty	integer	Age
age_band_of_casualty	integer	Age range of the casualty
casualty_severity	integer	Severity of the injuries
pedestrian_location	integer	Location of the injured pedestrian
pedestrian_movement	integer	Activity or movement of the pedestrian
car_passenger	integer	Yes/No
bus_or_coach_passenger	integer	Yes/No
pedestrian_road_maintenance_worker	integer	Yes/No/Maybe
casualty_type	integer	Was the injured a driver, passenger, cyclist etc.
casualty_home_area_type	integer	Urban/rural/town etc.
casualty_imd_decile	integer	Index of Multiple Deprivation

Figure 8
Vehicle

Column Name	Data Type	Description
accident_index	string	Unique identifier of the accident.
accident_year	integer	Year of the accident.
accident_reference	string	Used by the police to reference a collision. It is not unique outside of the year.
vehicle_reference	integer	Unique value for each vehicle in a singular accident.
vehicle_type	integer	Category of vehicle involved in the accident
towing_and_articulation	integer	Type of tow/articulation like caravan, single trailer etc.
vehicle_manoeuvre	integer	Slowing, stopping, parked reversed etc. while accident happened
vehicle_direction_from	integer	Direction like north, northwest etc.
vehicle_direction_to	integer	Direction like north, northwest etc.
vehicle_location_restricted_lane	integer	Bus lane, cycle lane, rail track etc.
junction_location	integer	Location of the junction like entering main road, leaving round about etc.
skidding_and_overturning	integer	Skidded, skidded and overturned, Jackknifed etc.
hit_object_in_carriageway	integer	Details about the objects that were hit like parked vehicle, bridge etc.
vehicle_leaving_carriageway	integer	Nearside, offside, Offside and rebounded etc.
hit_object_off_carriageway	integer	Details about the objects that were hit like tree, lamp post, wall etc.
first_point_of_impact	integer	The first point of impact like Front, backside, offside etc.
vehicle_left_hand_drive	integer	Yes/No/Unknown.
journey_purpose_of_driver	integer	Purpose of commute
sex_of_driver	integer	Male/Female
age_of_driver	integer	Age
age_band_of_driver	integer	Age range category
engine_capacity_cc	integer	Engine cc
propulsion_code	integer	Petrol, Diesel, gas, electric etc.
age_of_vehicle	integer	Age of the vehicle
generic_make_model	string	Model of the vehicle
driver_imd_decile	integer	Index of multiple deprivation
driver_home_area_type	integer	Rural, urban, small town etc.

1.3 Implementation

SparkSQL module is used for data processing as it simplifies the process of querying data stored in Spark Dataframe (Databricks, n.d.). The EDA visualizations were all completed in Tableau.

When all the 3 files are joined, there are more than 600 thousand rows available in the merged file. A challenge in this case was that the preprocessing of the merged file was resulting in a JVM error for inadequate memory due to the huge file size : `"java.lang.OutOfMemoryError: GC overhead limit exceeded"`. This error is caused as the data is run in local mode. In local mode, PySpark runs on a single machine, and all the workers are launched in separate threads or processes within that machine (Pyspark, n.d.).
To resolve this issue, each csv file was individually preprocessed to remove irrelevant columns and NULL values, and the resulting files were merged in the end.

The following steps were carried out for each csv file (accidents, vehicles and causalities) and was finally merged into a single file for machine learning.

1) Load the csv file.

2) Data Cleaning : Due to the size of the dataset and the percentage of rows with missing primary attributes, I chose to eliminate the associated vehicles and casualties rather than impute missing values in order to maintain the consistency of the data.

3) Perform EDA and eliminate additional irrelevant columns that doesn't contribute to the accident severity.

Casualty dataset：

We can use the spark.read method to read a CSV file into a Spark DataFrame.

Figure 9
Casualty dataset

```
In [7]: # read the first CSV file
casualty = spark.read.format("csv").option("header", "true").load('/Users/meeranair/Documents/Coventry/7153CEM/dft-
                                inferSchema=True).alias("casualty")
casualty.printSchema()

# get the number of rows
num_rows = casualty.count()
# get the number of columns
num_cols = len(casualty.columns)
print("Number of rows: ", num_rows)
print("Number of columns: ", num_cols)
```

```
root
 |-- accident_index: string (nullable = true)
 |-- accident_year: integer (nullable = true)
 |-- accident_reference: string (nullable = true)
 |-- vehicle_reference: integer (nullable = true)
 |-- casualty_reference: integer (nullable = true)
 |-- casualty_class: integer (nullable = true)
 |-- sex_of_casualty: integer (nullable = true)
 |-- age_of_casualty: integer (nullable = true)
 |-- age_band_of_casualty: integer (nullable = true)
 |-- casualty_severity: integer (nullable = true)
 |-- pedestrian_location: integer (nullable = true)
 |-- pedestrian_movement: integer (nullable = true)
 |-- car_passenger: integer (nullable = true)
 |-- bus_or_coach_passenger: integer (nullable = true)
 |-- pedestrian_road_maintenance_worker: integer (nullable = true)
 |-- casualty_type: integer (nullable = true)
 |-- casualty_home_area_type: integer (nullable = true)
 |-- casualty_imd_decile: integer (nullable = true)

Number of rows:  781716
Number of columns:  18
```

Each accident has a unique accident_index value and hence we drop duplicates, if any. Then we check for null, Nan, empty strings and missing values. According to the Department of Transportation's dataset guide, the dataset's missing values are denoted by the number -1. We therefore extract the list of columns and the total number of missing values in each column after loading the data. This is being done to get rid of the columns with thousands of missing values.

Figure 10
Preprocessing

```
: #Drop duplicates
casualty = casualty.dropDuplicates(['accident_index'])


# Check for null values
null_cols = [c for c in casualty.columns if casualty.filter((col(c).isNull()) | (col(c) == "NULL")).count() >

print("Columns with null values:", null_cols)

# check for NaN values
nan_cols = [c for c in casualty.columns if casualty.filter(isnan(col(c))).count() > 0]
print("Columns with NaN values:", nan_cols)

# check for empty or blank strings
blank_cols = [c for c in casualty.columns if casualty.filter((length(trim(col(c))) == 0)).count() > 0]
print("Columns with empty or blank strings:", blank_cols)

#check if any columns contain -1 (missing values)
cols_with_neg1 = []
for c in casualty.columns:
    count = casualty.filter(col(c) == -1).count()
    if count > 0:
        cols_with_neg1.append(c)
        print(f"Column {c} has {count} rows with a value of -1")
```

```
Columns with null values: []
```

```
Columns with NaN values: []
```

```
Columns with empty or blank strings: []
```

```
Column sex_of_casualty has 524 rows with a value of -1
Column age_of_casualty has 8373 rows with a value of -1
Column age_band_of_casualty has 8373 rows with a value of -1
```

```
Column pedestrian_location has 12 rows with a value of -1
Column pedestrian_movement has 16 rows with a value of -1
Column car_passenger has 564 rows with a value of -1
Column bus_or_coach_passenger has 159 rows with a value of -1
Column pedestrian_road_maintenance_worker has 419 rows with a value of -1
Column casualty_type has 10 rows with a value of -1
```

```
Column casualty_home_area_type has 71353 rows with a value of -1
Column casualty_imd_decile has 71501 rows with a value of -1
```

Figure 10 shows that while several integer columns have missing data, none of the columns have null or Nan values. We start by dropping the columns that have a large number of missing values since these columns can introduce biases into statistical studies that can affect the validity and accuracy of the results. We further filter out rows that have at least one missing value and remove them using the filter(), reduce(), count(), and subtract() functions from the PySpark SQL package. 1168 rows were deleted after this process.

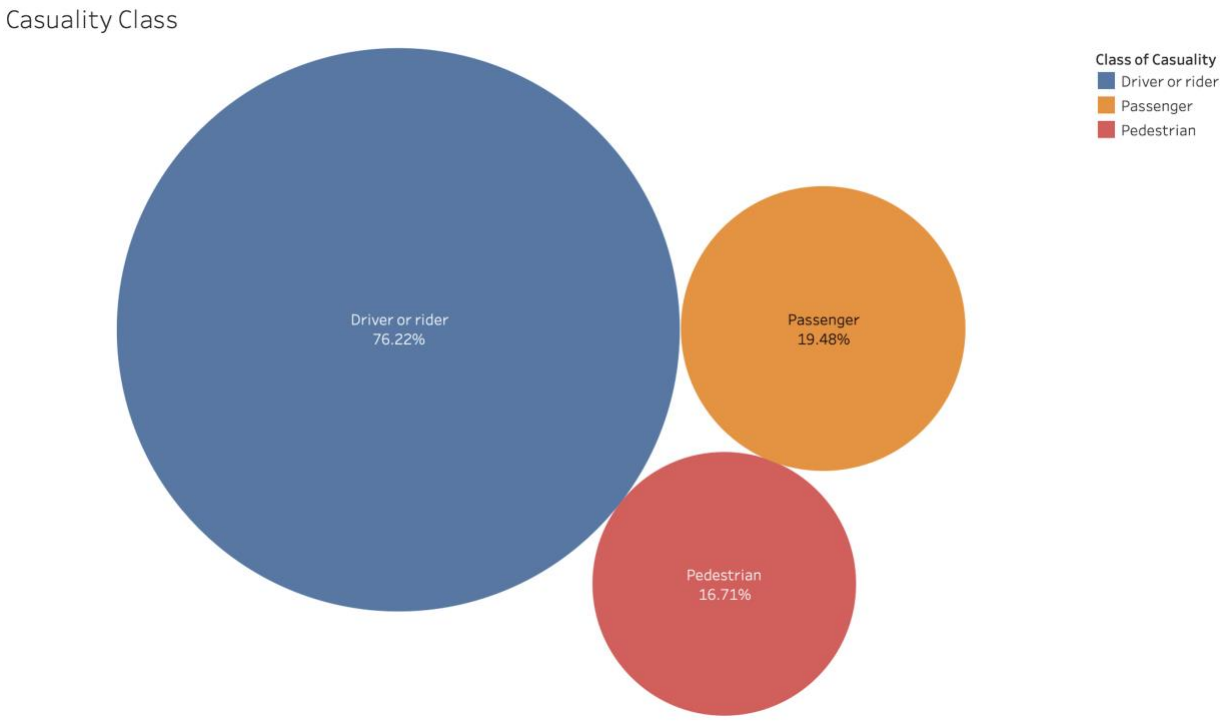
Figure 11
Clean the dataset

```
# Drop columns with too many missing values
casualty = casualty.drop('casualty_home_area_type','casualty_imd_decile','age_of_casualty','age_band_of_casual
                         'sex_of_casualty')

# Delete the rows containing -1 (missing values) and print number of rows that were deleted
conditions = [col(c) == -1 for c in casualty.columns]
count = casualty.filter(reduce(lambda x, y: x | y, conditions)).count()
accidents1_filtered = casualty.filter(reduce(lambda x, y: x | y, conditions))
print("Number of rows deleted with at least one -1 value:", count)

# Drop the filtered rows
accidents1_filtered = casualty.subtract(accidents1_filtered)
casualty = accidents1_filtered
```

```
Number of rows deleted with at least one -1 value: 1168
```

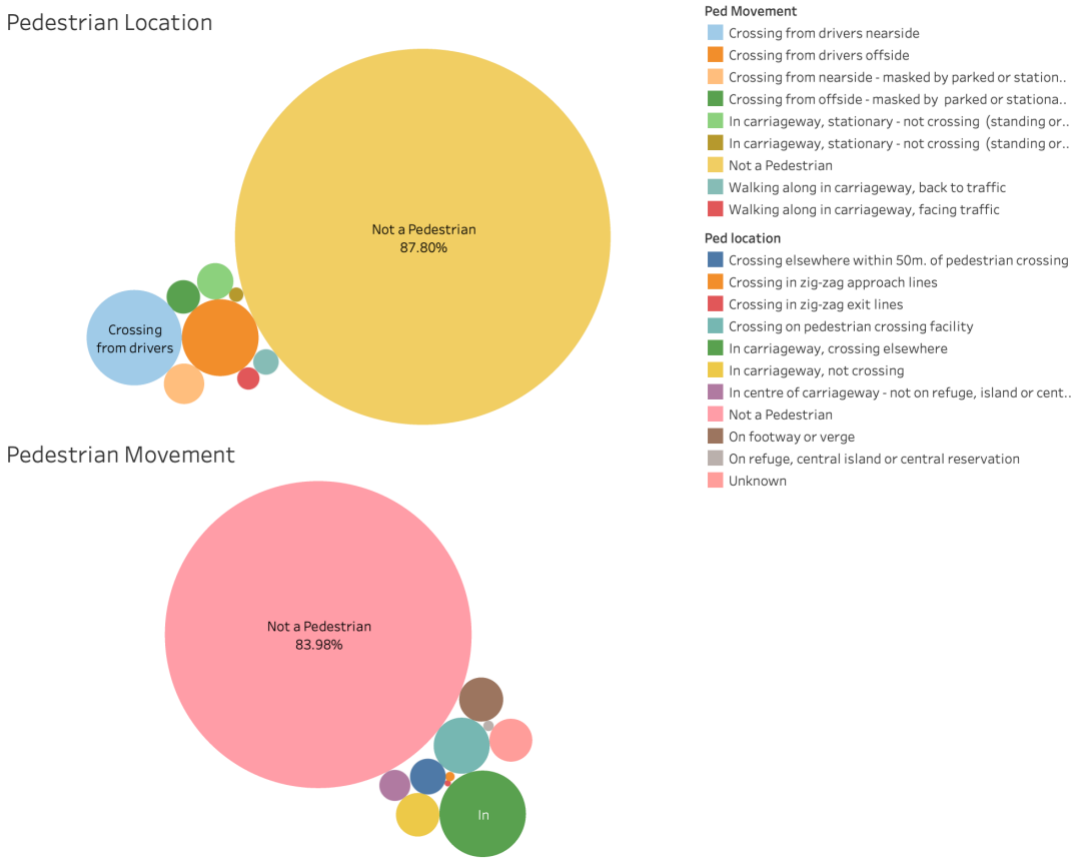
The majority of casualties, over 70%, are attributed to drivers.

Figure 12
EDA



Since most accidents don't involve pedestrians, we can fairly rule out pedestrian_movement and pedestrian_location because they don't tell us much about how serious the accident was.

Figure 13
EDA



Additionally, since the reference number columns are simply form numbers from the accident reports and are specific to each year, we may likewise eliminate them. The final casualty dataset details are shown in Figure 14.

Despite the fact that the majority of the columns have an integer datatype, these are encoded categorical variables and Road Safety Open Dataset Data Guide (data.gov.uk, n.d.) provides an explanation of the encoding.

Figure 14
Final casualty dataset

```
: #drop  irrelevant column
casualty = casualty.drop('pedestrian_location','pedestrian_movement', 'casualty_reference', 'accident_referenc
                         'vehicle_reference','accident_year')

casualty.printSchema()

# get the number of rows
num_rows = casualty.count()

# get the number of columns
num_cols = len(casualty.columns)

# print the results
print("Number of rows: ", num_rows)
print("Number of columns: ", num_cols)
```

```
root
 |-- accident_index: string (nullable = true)
 |-- casualty_class: integer (nullable = true)
 |-- casualty_severity: integer (nullable = true)
 |-- car_passenger: integer (nullable = true)
 |-- bus_or_coach_passenger: integer (nullable = true)
 |-- pedestrian_road_maintenance_worker: integer (nullable = true)
 |-- casualty_type: integer (nullable = true)

23/04/16 18:45:13 WARN package: Truncated the string representation of a plan since it was too large. This be
havior can be adjusted by setting 'spark.sql.debug.maxToStringFields'.
```

```
Number of rows:  596805
Number of columns:  7
```

Vehicle dataset： This dataset holds information about the vehicles involved in the accident. It provides fine details about the accident, like whether the vehicle hit a curb, was the first point of contact, etc., which can help us assess the severity of the accident.

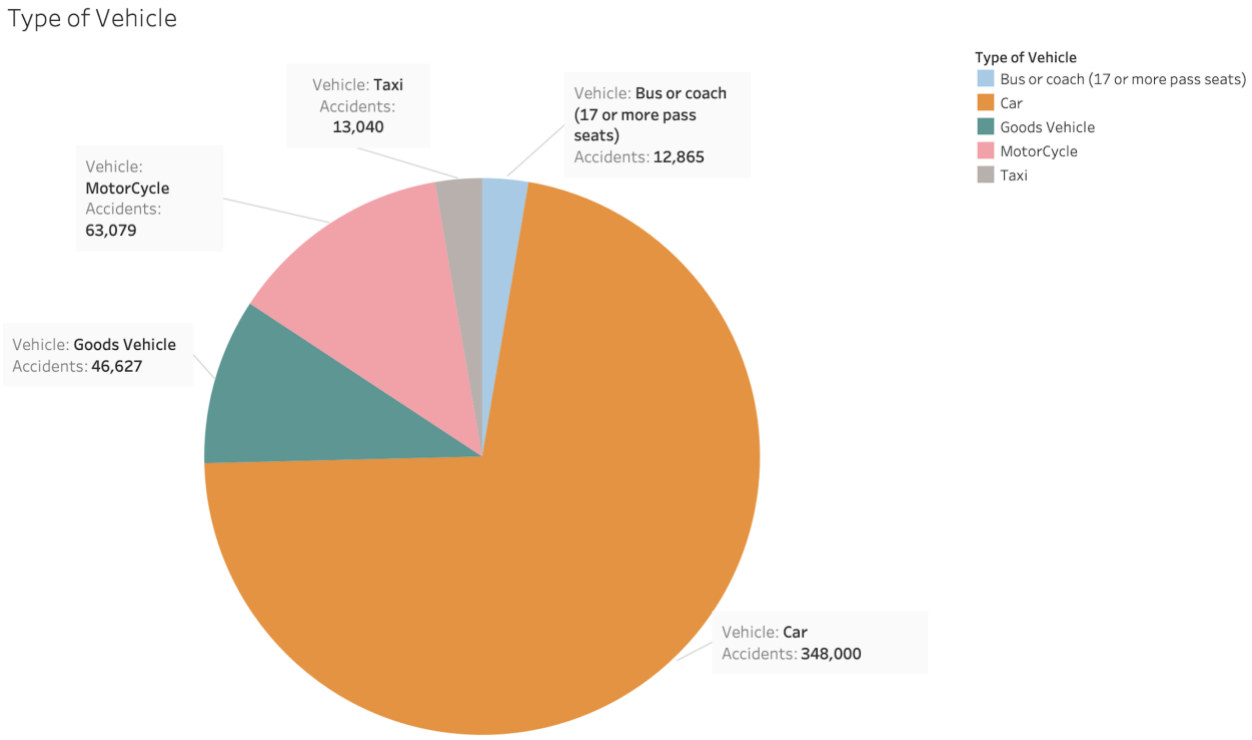
Figure 15
Vehicle dataset

```
In [13]: # read the second (vehicle) CSV file
vehicle = spark.read.format("csv").option("header", "true").load('/Users/meeranair/Documents/Coventry/7153CEM/dft-r
                                  inferSchema=True).alias("vehicle")
vehicle.printSchema()
# get the number of rows
num_rows = vehicle.count()
# get the number of columns
num_cols = len(vehicle.columns)
print("Number of rows: ", num_rows)
print("Number of columns: ", num_cols)
```

```
root
 |-- accident_index: string (nullable = true)
 |-- accident_year: integer (nullable = true)
 |-- accident_reference: string (nullable = true)
 |-- vehicle_reference: integer (nullable = true)
 |-- vehicle_type: integer (nullable = true)
 |-- towing_and_articulation: integer (nullable = true)
 |-- vehicle_manoeuvre: integer (nullable = true)
 |-- vehicle_direction_from: integer (nullable = true)
 |-- vehicle_direction_to: integer (nullable = true)
 |-- vehicle_location_restricted_lane: integer (nullable = true)
 |-- junction_location: integer (nullable = true)
 |-- skidding_and_overturning: integer (nullable = true)
 |-- hit_object_in_carriageway: integer (nullable = true)
 |-- vehicle_leaving_carriageway: integer (nullable = true)
 |-- hit_object_off_carriageway: integer (nullable = true)
 |-- first_point_of_impact: integer (nullable = true)
 |-- vehicle_left_hand_drive: integer (nullable = true)
 |-- journey_purpose_of_driver: integer (nullable = true)
 |-- sex_of_driver: integer (nullable = true)
 |-- age_of_driver: integer (nullable = true)
 |-- age_band_of_driver: integer (nullable = true)
 |-- engine_capacity_cc: integer (nullable = true)
 |-- propulsion_code: integer (nullable = true)
 |-- age_of_vehicle: integer (nullable = true)
 |-- generic_make_model: string (nullable = true)
 |-- driver_imd_decile: integer (nullable = true)
 |-- driver_home_area_type: integer (nullable = true)

Number of rows:  1101591
Number of columns:  27
```

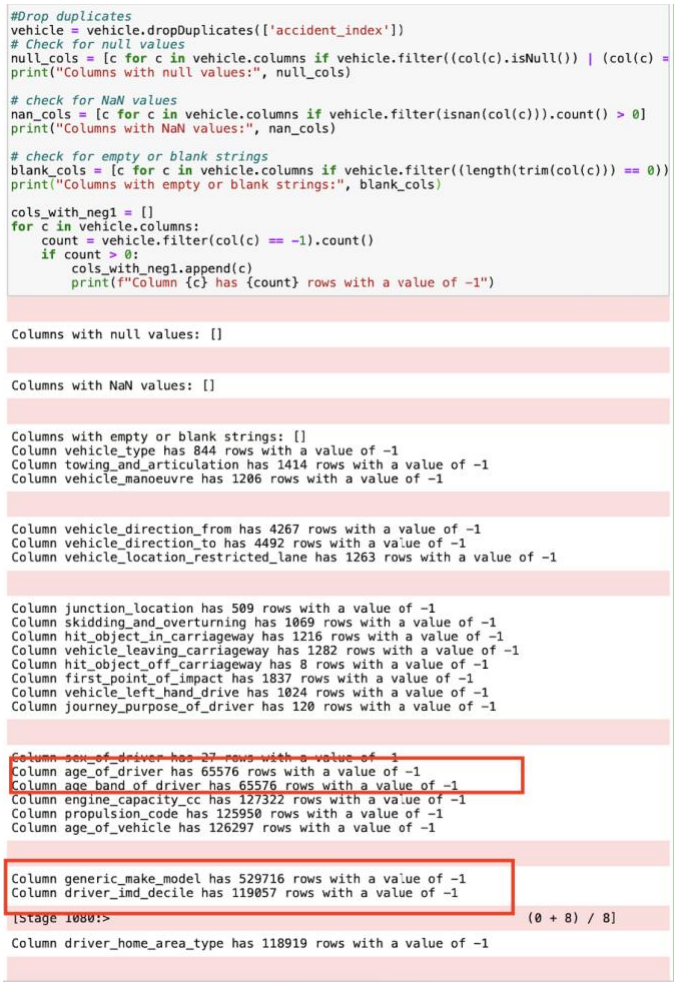
Cars are responsible for the highest number of accidents, with motorcycles coming in second.

Figure 16
EDA



Similar to the casualty dataset, we clean the dataset after loading the file to remove null, nan, and missing values. Despite the fact that this dataset has columns for storing driver information, the majority of the details are missing, as shown in Figure 17, thus we won't use any columns containing driver details except journey purpose of the driver.

Figure 17
Check for missing values



```
#Drop duplicates
vehicle = vehicle.dropDuplicates(['accident_index'])
# Check for null values
null_cols = [c for c in vehicle.columns if vehicle.filter((col(c).isNull()) | (col(c) =
print("Columns with null values:", null_cols)

# check for NaN values
nan_cols = [c for c in vehicle.columns if vehicle.filter(isnan(col(c))).count() > 0]
print("Columns with NaN values:", nan_cols)

# check for empty or blank strings
blank_cols = [c for c in vehicle.columns if vehicle.filter((length(trim(col(c))) == 0))
print("Columns with empty or blank strings:", blank_cols)

cols_with_neg1 = []
for c in vehicle.columns:
    count = vehicle.filter(col(c) == -1).count()
    if count > 0:
        cols_with_neg1.append(c)
        print(f"Column {c} has {count} rows with a value of -1")
```

```
Columns with null values: []
```

```
Columns with NaN values: []
```

```
Columns with empty or blank strings: []
Column vehicle_type has 844 rows with a value of -1
Column towing_and_articulation has 1414 rows with a value of -1
Column vehicle_manoeuvre has 1206 rows with a value of -1
```

```
Column vehicle_direction_from has 4267 rows with a value of -1
Column vehicle_direction_to has 4492 rows with a value of -1
Column vehicle_location_restricted_lane has 1263 rows with a value of -1
```

```
Column junction_location has 509 rows with a value of -1
Column skidding_and_overturning has 1069 rows with a value of -1
Column hit_object_in_carriageway has 1216 rows with a value of -1
Column vehicle_leaving_carriageway has 1282 rows with a value of -1
Column hit_object_off_carriageway has 8 rows with a value of -1
Column first_point_of_impact has 1837 rows with a value of -1
Column vehicle_left_hand_drive has 1024 rows with a value of -1
Column journey_purpose_of_driver has 120 rows with a value of -1
```

```
Column sex_of_driver has 27 rows with a value of -1
Column age_of_driver has 65576 rows with a value of -1
Column age_band_of_driver has 65576 rows with a value of -1
Column engine_capacity_cc has 127322 rows with a value of -1
Column propulsion_code has 125950 rows with a value of -1
Column age_of_vehicle has 126297 rows with a value of -1
```

```
Column generic_make_model has 529716 rows with a value of -1
Column driver_imd_decile has 119057 rows with a value of -1
[Stage 1080:>                                                 (0 + 8) / 8]
Column driver_home_area_type has 118919 rows with a value of -1
```

4627 rows were deleted due to missing values.

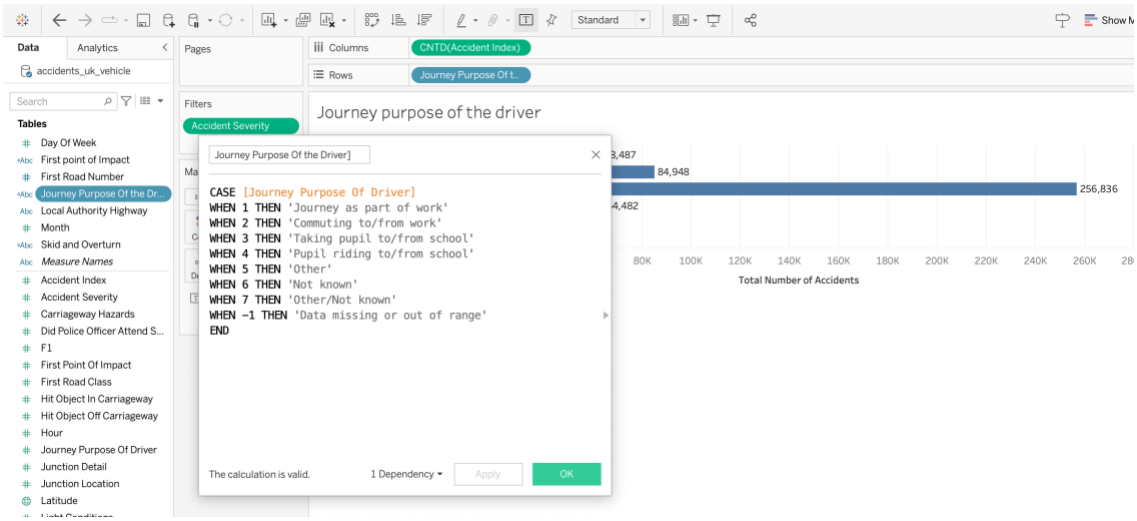
Figure 18
Clean the dataset

```
]: # Drop columns with too many missing values and also irrelevant columns
   vehicle = vehicle.drop('driver_home_area_type','driver_imd_decile','generic_make_model','age_of_vehicle',
                          'propulsion_code','engine_capacity_cc','age_band_of_driver','age_of_driver', 'accident_
                          'vehicle_direction_to','vehicle_direction_from','accident_reference','vehicle_reference

   # Delete the rows containing -1 (missing values) and print number of rows that were deleted
   conditions = [col(c) == -1 for c in vehicle.columns]
   count = vehicle.filter(reduce(lambda x, y: x | y, conditions)).count()
   accidents2_filtered = vehicle.filter(reduce(lambda x, y: x | y, conditions))
   print("Number of rows deleted with at least one -1 value:", count)

   # Drop the filtered rows
   accidents2_filtered = vehicle.subtract(accidents2_filtered)
   vehicle = accidents2_filtered
```

```
[Stage 1088:>                                                 (0 + 8) / 8]
Number of rows deleted with at least one -1 value: 4627
```

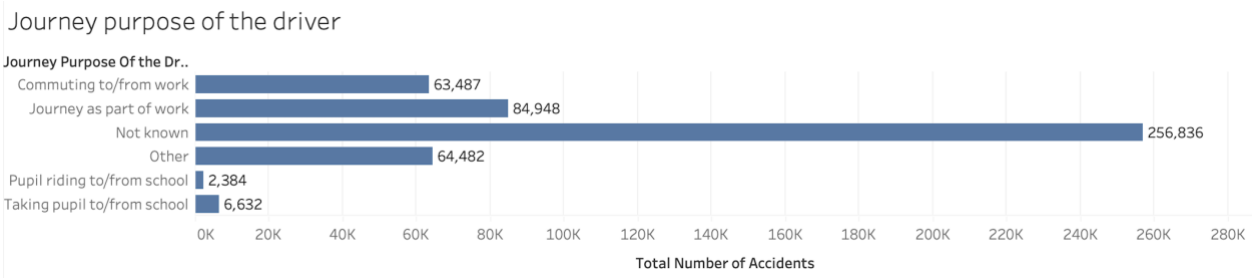
In tableau we can create a calculated field (Figure 19) and map the description of the encoded categorical variables to visualize the data.

Figure 19
Creating a new field



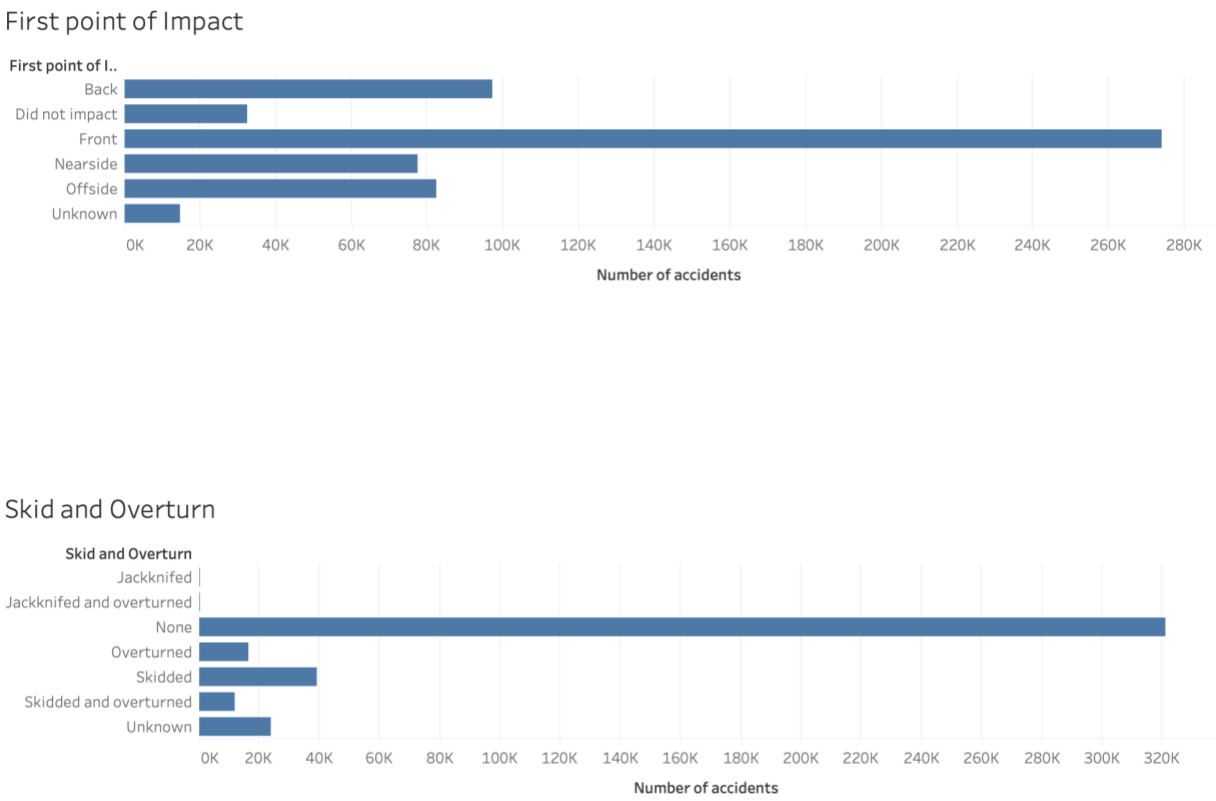
The majority of accidents have unknown travel objectives. Most accidents that have been documented happen during the commute to work.

Figure 20
EDA



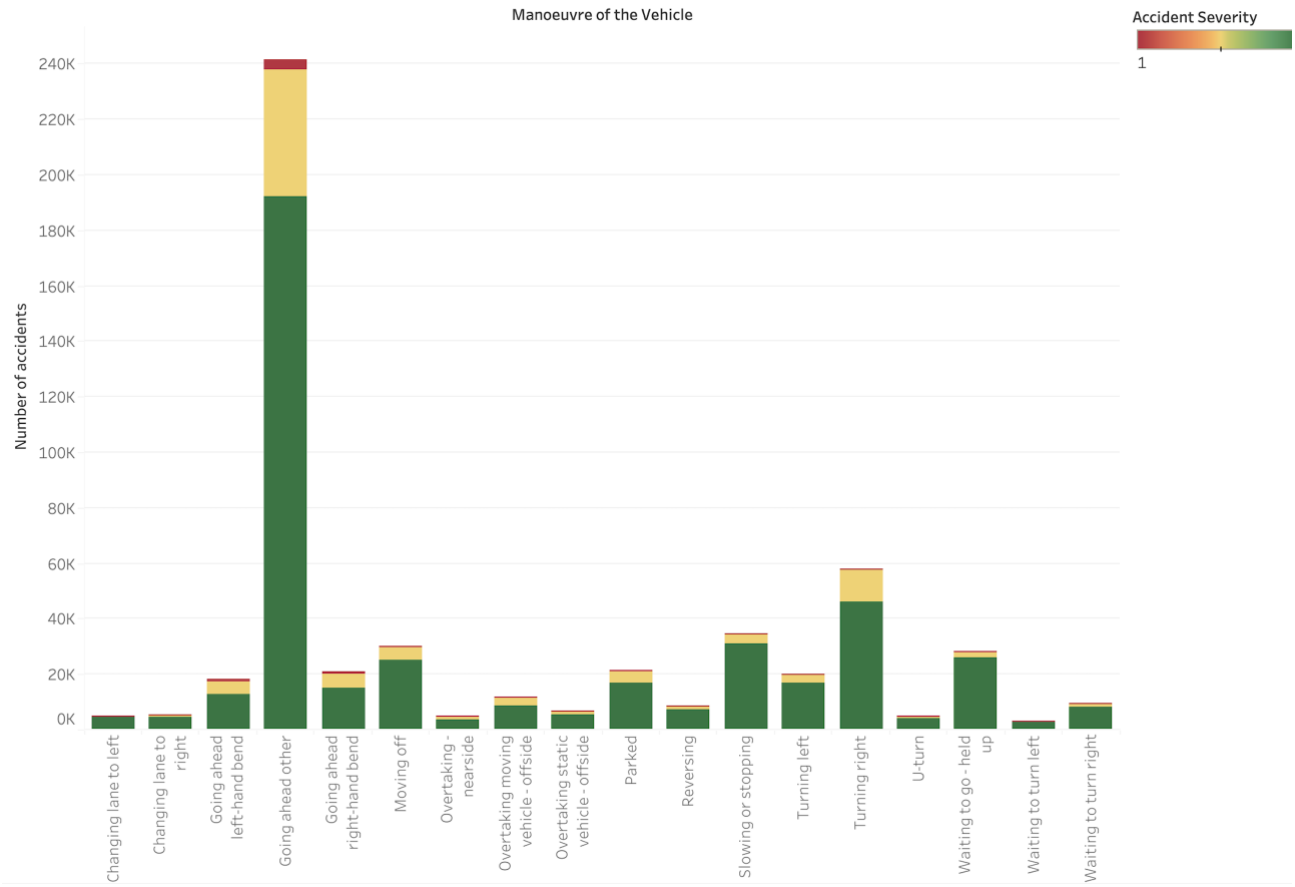
Most accidents have their first point of impact on the front of the vehicle (Fig 21), with sliding and overturning being less frequent occurrences. According to a study by the National Highway Traffic Safety Administration (NHTSA), only about 10% of all accidents involved skidding or overturning as a contributing factor (NHTSA, 2016). Skidding and overturning, on the other hand, are less common outcomes maybe because they often require specific conditions, such as high speeds, poor road conditions etc. And hence, we will further analyze these conditions from the third accidents file.

Figure 21
EDA



Analyzing the vehicle manoeuvre column reveals that the first point of impact and vehicle manoeuvre use different words to describe the same information.
As seen in figure 22, most accidents occur when vehicles collide with those moving in front of them, which is similar to the first point of impact occurring in the front (Figure 21). Hence, we can remove vehicle_manoeuvre column without losing any valuable information.

Figure22
EDA



The final vehicle dataset details are shown in Figure 23. Similar to casualty dataset, the integer columns are encoded categorical variables and Road Safety Open Dataset Data Guide (data.gov.uk, n.d.) provides an explanation of the encoding.

Figure 23
Final Vehicle dataset

```
vehicle = vehicle.drop('vehicle_manoeuvre')
vehicle.printSchema()

# get the number of rows
num_rows = vehicle.count()

# get the number of columns
num_cols = len(vehicle.columns)

# print the results
print("Number of rows: ", num_rows)
print("Number of columns: ", num_cols)
```

```
root
 |-- accident_index: string (nullable = true)
 |-- vehicle_type: integer (nullable = true)
 |-- towing_and_articulation: integer (nullable = true)
 |-- vehicle_location_restricted_lane: integer (nullable = true)
 |-- junction_location: integer (nullable = true)
 |-- skidding_and_overturning: integer (nullable = true)
 |-- hit_object_in_carriageway: integer (nullable = true)
 |-- vehicle_leaving_carriageway: integer (nullable = true)
 |-- hit_object_off_carriageway: integer (nullable = true)
 |-- first_point_of_impact: integer (nullable = true)
 |-- vehicle_left_hand_drive: integer (nullable = true)
 |-- journey_purpose_of_driver: integer (nullable = true)
```

```
Number of rows:  593346
Number of columns:  12
```

Accident dataset： This dataset contains all the pertinent facts about an accident. It has columns with several data types, including geolocation, timestamp, integer, and string.

Figure 24
Accident dataset

```
accidents3 = spark.read.format("csv").option("header", "true").load('/Users/meeranair/Documents/Coventry/7153C
                              inferSchema=True).alias("accidents3")

accidents3.printSchema()
# get the number of rows
num_rows = accidents3.count()
# get the number of columns
num_cols = len(accidents3.columns)
print("Number of rows: ", num_rows)
print("Number of columns: ", num_cols)
```

```
root
 |-- accident_index: string (nullable = true)
 |-- accident_year: integer (nullable = true)
 |-- accident_reference: string (nullable = true)
 |-- location_easting_osgr: string (nullable = true)
 |-- location_northing_osgr: string (nullable = true)
 |-- longitude: string (nullable = true)
 |-- latitude: string (nullable = true)
 |-- police_force: integer (nullable = true)
 |-- accident_severity: integer (nullable = true)
 |-- number_of_vehicles: integer (nullable = true)
 |-- number_of_casualties: integer (nullable = true)
 |-- date: string (nullable = true)
 |-- day_of_week: integer (nullable = true)
 |-- time: timestamp (nullable = true)
 |-- local_authority_district: integer (nullable = true)
 |-- local_authority_ons_district: string (nullable = true)
 |-- local_authority_highway: string (nullable = true)
 |-- first_road_class: integer (nullable = true)
 |-- first_road_number: integer (nullable = true)
 |-- road_type: integer (nullable = true)
 |-- speed_limit: string (nullable = true)
 |-- junction_detail: integer (nullable = true)
 |-- junction_control: integer (nullable = true)
 |-- second_road_class: integer (nullable = true)
 |-- second_road_number: integer (nullable = true)
 |-- pedestrian_crossing_human_control: integer (nullable = true)
 |-- pedestrian_crossing_physical_facilities: integer (nullable = true)
 |-- light_conditions: integer (nullable = true)
 |-- weather_conditions: integer (nullable = true)
 |-- road_surface_conditions: integer (nullable = true)
 |-- special_conditions_at_site: integer (nullable = true)
 |-- carriageway_hazards: integer (nullable = true)
 |-- urban_or_rural_area: integer (nullable = true)
 |-- did_police_officer_attend_scene_of_accident: integer (nullable = true)
 |-- trunk_road_flag: integer (nullable = true)
 |-- lsoa_of_accident_location: string (nullable = true)

Number of rows:  597973
Number of columns:  36
```

The columns indicating the location have a string type after the import. Therefore, we must convert this to double type.

Figure 25
Convert datatype

```
#convert into double type
accidents3 = accidents3.withColumn("latitude", col("latitude").cast("double"))
accidents3 = accidents3.withColumn("longitude", col("longitude").cast("double"))
accidents3 = accidents3.withColumn("location_easting_osgr", col("location_easting_osgr").cast("double"))
accidents3 = accidents3.withColumn("location_northing_osgr", col("location_northing_osgr").cast("double"))
```

Apart from latitude and longitude the dataset contains Easting OSGR coordinates. The British National Grid system, a coordinate reference system used to identify locations in Great Britain, uses the phrase "easting" to describe the process. The Easting coordinate, which measures the

distance eastward from a central meridian, is the horizontal part of the grid reference (Ordnancesurvey, n.d.).
The term "OSGR" refers to Ordnance Survey Grid Reference, which is the accepted system in Great Britain for utilizing a combination of letters and numbers to identify any point on a map (Wikipedia, 2023).

If we do a correlation check, we observe that the latitude, longitude, location_easting_osgr, and location_northing_osgr are identical with a correlation of almost 1. Therefore, we only need to utilize latitude and longitude and can remove location_easting_osgr and location_northing_osgr.

Figure 26
Correlation check

```
#check correlation of geolocation columns
corr_value = accidents3.limit(200).select(corr("longitude", "location_easting_osgr")).collect()[0][0]
print("Correlation between longitude and location_easting_osgr: ", corr_value)

corr_value = accidents3.limit(200).select(corr("latitude", "location_northing_osgr")).collect()[0][0]
print("Correlation between latitude and location_northing_osgr: ", corr_value)
```

```
Correlation between longitude and location_easting_osgr:  0.9998404965212871
Correlation between latitude and location_northing_osgr:  0.9993090059436579
```

The dataset also provides information about the date and time of the accident. To make analysis easier, we may add two new columns by separating the hours from the time column and the month from the date column.

Figure 27
Datetime processing

```
# Extract hour from time of accident
accidents3 = accidents3.withColumn("hour", date_format(col("time"), "HH").cast("int"))
# Get month ffrom date of accident
accidents3 = accidents3.withColumn("month", split("date", "/").getItem(1).cast("int"))
```

```
# drop the columns whose information has been extracted
accidents3 = accidents3.drop('date','time','location_easting_osgr','location_northing_osgr')
```

The time periods with the highest number of accidents on weekdays are during the morning from 8am to 10am and in the evening from 4pm to 6pm. These are usually rush hours when people are traveling to and from work or school.

Figure 28
EDA

Day of the Week

Hour Range	Sun	Mon	Tue	Wed	Thu	Fri	Sat
4 - 6	1,177	812	724	840	802	875	1,102
6 - 8	1,440	4,610	5,005	5,059	4,940	4,479	1,829
8 - 10	2,619	7,930	8,607	8,544	8,401	7,730	3,712
10 - 12	5,250	5,516	5,561	5,505	5,675	6,043	6,509
12 - 14	6,966	6,521	6,361	6,616	6,566	7,609	7,987
14 - 16	6,921	8,091	8,144	8,509	8,408	10,038	7,388
16 - 18	6,443	10,411	11,090	11,131	11,416	11,501	7,309
18 - 20	5,462	6,848	7,566	7,684	8,019	8,663	6,729
20 - 22	3,575	3,562	3,807	3,945	4,166	5,024	4,624

Number of Accidents
724 11,501

Our next step involves eliminating any null or missing values. Initially, we will identify all the columns that contain invalid data, and subsequently determine the appropriate method of handling them.

Figure 29
Preprocessing

```
#Drop duplicates
accidents3 = accidents3.dropDuplicates(['accident_index'])

# Check for null values
null_cols = [c for c in accidents3.columns if accidents3.filter((col(c).isNull()) | (co
print("Columns with null values:", null_cols)

# check for NaN values
nan_cols = [c for c in accidents3.columns if accidents3.filter(isnan(col(c))).count() >
print("Columns with NaN values:", nan_cols)

# check for empty or blank strings
blank_cols = [c for c in accidents3.columns if accidents3.filter((length(trim(col(c)))
print("Columns with empty or blank strings:", blank_cols)

# check for missing values (-1)
cols_with_neg1 = []
for c in accidents3.columns:
    count = accidents3.filter(col(c) == -1).count()
    if count > 0:
        cols_with_neg1.append(c)
        print(f"Column {c} has {count} rows with a value of -1")
```

```
Columns with null values: ['longitude', 'latitude', 'speed_limit']

Columns with NaN values: []

Columns with empty or blank strings: []

Column local_authority_district has 991 rows with a value of -1

Column local_authority_highway has 3154 rows with a value of -1
Column road_type has 1 rows with a value of -1

Column speed_limit has 92 rows with a value of -1
Column junction_detail has 7 rows with a value of -1
Column junction_control has 251177 rows with a value of -1
Column second_road_class has 212717 rows with a value of -1
Column second_road_number has 212773 rows with a value of -1
Column pedestrian_crossing_human_control has 743 rows with a value of -1
Column pedestrian_crossing_physical_facilities has 694 rows with a value of -1
Column light_conditions has 16 rows with a value of -1
Column weather_conditions has 34 rows with a value of -1
Column road_surface_conditions has 1786 rows with a value of -1

Column special_conditions_at_site has 713 rows with a value of -1
Column carriageway_hazards has 747 rows with a value of -1
Column urban_or_rural_area has 1 rows with a value of -1
Column did_police_officer_attend_scene_of_accident has 3 rows with a value of -1
Column trunk_road_flag has 52298 rows with a value of -1
Column lsoa_of_accident_location has 31492 rows with a value of -1
```

Handling Null Values : Figure 29 shows that latitude, longitude and speed_limit have Null values, so we next count the number of Null values.
Since there are only 133 and 35 Null values in location and speed limit respectively, we can drop them.

Figure 30
Check Java Version

```
# Get list of columns with null values
null_cols = [c for c in accidents3.columns if (accidents3.filter(col(c).isNull()).count() + accidents3.filter(


# Count the number of null values in each column
for col_name in null_cols:
    null_count = accidents3.filter((col(col_name).isNull()) | (col(col_name) == "NULL")).count()
    print(f"Column '{col_name}' has {null_count} null values.")
```

```
Column 'longitude' has 133 null values.
Column 'latitude' has 133 null values.
Column 'speed_limit' has 37 null values.
```

```
# Drop the NULL values
accidents3 = accidents3.replace("NULL", None)
accidents3 = accidents3.dropna()
```

Handling missing Values : In order to avoid bias, we can eliminate columns that contain an excessive number of missing values. Then we filter out the rows that contain -1 (missing value).

Figure 31
Preprocessing

```
# Drop columns with too many missing values
accidents3 = accidents3.drop('lsoa_of_accident_location','trunk_road_flag','second_road_number','second_road_c
                             'junction_control')


# Delete the rows containing -1 (missing values) and print number of rows that were deleted
conditions = [col(c) == -1 for c in accidents3.columns]
count = accidents3.filter(reduce(lambda x, y: x | y, conditions)).count()
accidents3_filtered = accidents3.filter(reduce(lambda x, y: x | y, conditions))
print("Number of rows deleted with at least one -1 value:", count)

# Drop the filtered rows
accidents3_filtered = accidents3.subtract(accidents3_filtered)
accidents3 = accidents3_filtered
```

```
[Stage 501:=======>                                                 (1 + 7) / 8]
Number of rows deleted with at least one -1 value: 6820
```

Handling latitude and longitude : The great_circle function is a part of the geopy library that can calculate the shortest distance (in kilometers) between two points on the Earth's surface, given their latitude and longitude coordinates (geopy, n.d.). We can calculate the average latitude and longitude of accident locations, and then adds a new column named “distance_km” to the data that shows the distance from each accident to the center point in kilometers.
In this code, the great_circle function is used to calculate the distance between the accident locations and average (mean) latitude and longitude values of the accident locations.

Figure 32
Distance calculation

```
: # Calculate the mean of the latitude and longitude columns
center_lat, center_lon = accidents3.agg(mean("latitude"), mean("longitude")).first()

# Define the UDF to calculate distance
def calculate_distance(lat, lon):
    return great_circle((lat, lon), (center_lat, center_lon)).kilometers

distance_udf = udf(calculate_distance, DoubleType())

# Apply the UDF to create a new column called "distance_km"
accidents3 = accidents3.withColumn("distance_km", distance_udf(col("latitude"), col("longitude")))

#Drop latitude and longitude
accidents3 = accidents3.drop('latitude','longitude')
```

After the initial cleaning of the data set is complete, we may thoroughly examine the remaining columns to verify the data they contain. This is being done to ensure that no two columns include information that is both redundant and irrelevant.

Figure 33
Schema

```
: accidents3.printSchema()
root
 |-- accident_index: string (nullable = true)
 |-- accident_year: integer (nullable = true)
 |-- accident_reference: string (nullable = true)
 |-- police_force: integer (nullable = true)
 |-- accident_severity: integer (nullable = true)
 |-- number_of_vehicles: integer (nullable = true)
 |-- number_of_casualties: integer (nullable = true)
 |-- day_of_week: integer (nullable = true)
 |-- local_authority_district: integer (nullable = true)
 |-- local_authority_ons_district: string (nullable = true)
 |-- local_authority_highway: string (nullable = true)
 |-- first_road_class: integer (nullable = true)
 |-- first_road_number: integer (nullable = true)
 |-- road_type: integer (nullable = true)
 |-- speed_limit: string (nullable = true)
 |-- junction_detail: integer (nullable = true)
 |-- pedestrian_crossing_human_control: integer (nullable = true)
 |-- pedestrian_crossing_physical_facilities: integer (nullable = true)
 |-- light_conditions: integer (nullable = true)
 |-- weather_conditions: integer (nullable = true)
 |-- road_surface_conditions: integer (nullable = true)
 |-- special_conditions_at_site: integer (nullable = true)
 |-- carriageway_hazards: integer (nullable = true)
 |-- urban_or_rural_area: integer (nullable = true)
 |-- did_police_officer_attend_scene_of_accident: integer (nullable = true)
 |-- hour: integer (nullable = true)
 |-- month: integer (nullable = true)
 |-- distance_km: double (nullable = true)
```

Both "local_authority_ons_district" and "local_authority_district" columns refer to administrative areas within the United Kingdom that are responsible for providing local government services. "Local_authority_ons_district" refers to the administrative areas defined by the Office for National Statistics (ONS) for the purpose of statistical analysis and reporting (Wikipedia, 2023). "Local_authority_district", on the other hand, is a more generic term that can refer to any district or area that is governed by a local authority.
Hence, we can drop local_authority_ons_district column.

Next, we can convert the “speed_limit” column into categories.

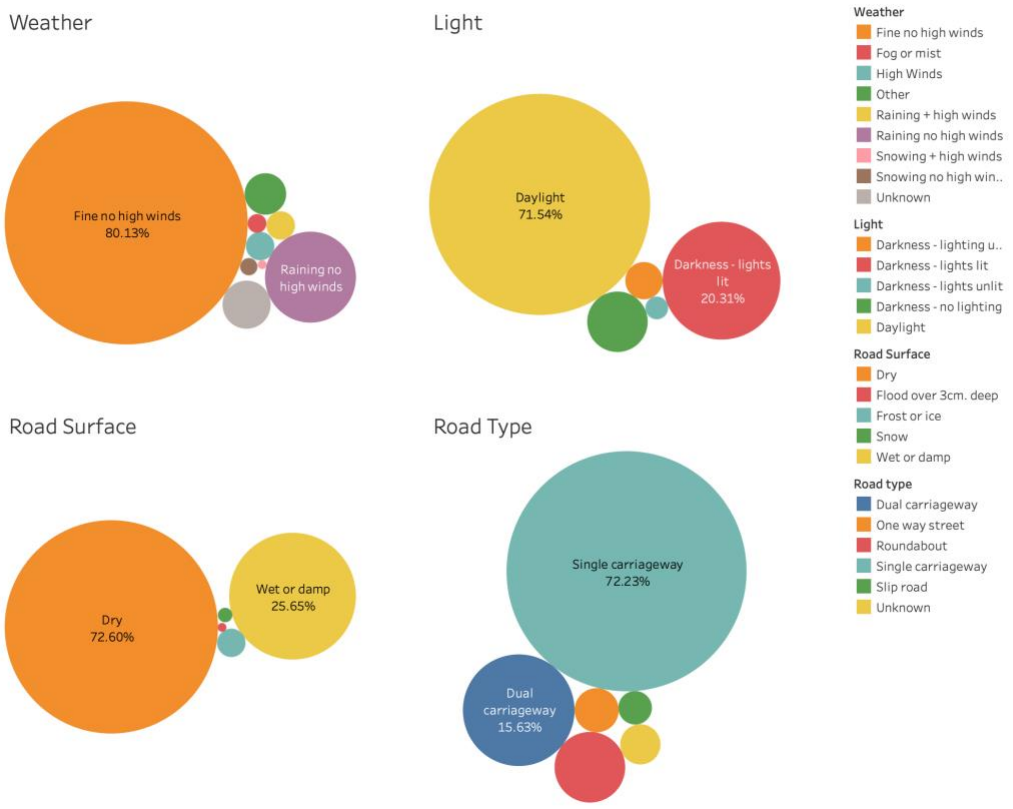
Figure 34
Categorizing speed_limit

```
accidents3 = accidents3.withColumn("road_type_limit", \
                                   when(col("speed_limit") == 70, "motorway/dual_carriageway") \
                                   .when(col("speed_limit") == 60, "single_carriageway_(60mph)") \
                                   .when(col("speed_limit") == 50, "single_carriageway_(50mph)") \
                                   .when(col("speed_limit") == 40, "urban_road") \
                                   .when(col("speed_limit") == 30, "urban_road") \
                                   .when(col("speed_limit") == 20, "urban_road") \
                                   .otherwise("other"))
```

Figure 21 shows that overturning and skidding rarely occur; thus, we can investigate the overall weather and road conditions to determine why.

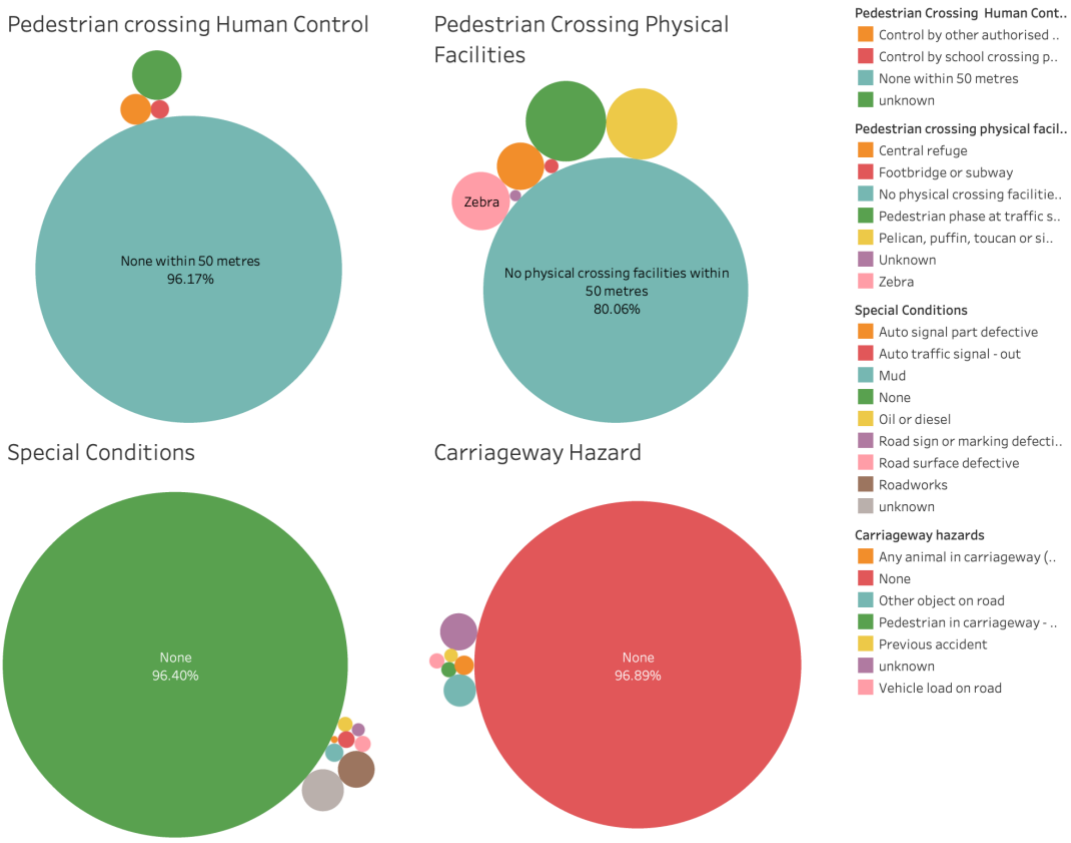
80% of accidents occurred during calm days without strong winds. This might be connected to the fact that 72% of accidents happened on dry surfaces of the roads. More than 70% of accidents happened on daylight and on a single carriageway. In general, it can be seen that the majority of incidents happened in good weather, such as daylight, with no severe winds or dry road surface.

Figure 35
EDA



As seen in Figure 36 , none of the accident-causing variables in the UK were any hazard in the carriageway (97%), any unusual state on the road (96%) or any pedestrian crossing with human control (96%). Therefore, it can be said that they weren't contributing factors to the occurrence of traffic accidents.

Figure 36
EDA



Lastly, since they have no bearing on the severity of the accident, we can eliminate accident_reference and accident_year columns.

Figure 37
Final accidents dataset

```
accidents3 =  accidents3.drop('local_authority_ons_district','speed_limit','accident_reference','accident_year

accidents3.printSchema()

# get the number of rows
num_rows = accidents3.count()

# get the number of columns
num_cols = len(accidents3.columns)

# print the results
print("Number of rows: ", num_rows)
print("Number of columns: ", num_cols)
```

```
root
 |-- accident_index: string (nullable = true)
 |-- police_force: integer (nullable = true)
 |-- accident_severity: integer (nullable = true)
 |-- number_of_vehicles: integer (nullable = true)
 |-- number_of_casualties: integer (nullable = true)
 |-- day_of_week: integer (nullable = true)
 |-- local_authority_district: integer (nullable = true)
 |-- local_authority_highway: string (nullable = true)
 |-- first_road_class: integer (nullable = true)
 |-- first_road_number: integer (nullable = true)
 |-- road_type: integer (nullable = true)
 |-- junction_detail: integer (nullable = true)
 |-- pedestrian_crossing_human_control: integer (nullable = true)
 |-- pedestrian_crossing_physical_facilities: integer (nullable = true)
 |-- light_conditions: integer (nullable = true)
 |-- weather_conditions: integer (nullable = true)
 |-- road_surface_conditions: integer (nullable = true)
 |-- special_conditions_at_site: integer (nullable = true)
 |-- carriageway_hazards: integer (nullable = true)
 |-- urban_or_rural_area: integer (nullable = true)
 |-- did_police_officer_attend_scene_of_accident: integer (nullable = true)
 |-- hour: integer (nullable = true)
 |-- month: integer (nullable = true)
 |-- distance_km: double (nullable = true)
 |-- road_type_limit: string (nullable = false)
```

```
[Stage 2575:>                                                       (0 + 8) / 9]
Number of rows:  590983
Number of columns:  25
```

After cleaning and preprocessing each of the three files separately, we can now combine them using the shared key accident_index.

Figure 38
Merged file

```
#merge the datasets
accidents_uk = casualty.join(vehicle, "accident_index", how='inner').join(accidents3, "accident_index", how='i
```

We can then double-check that the combined file doesn't contain any empty or missing values.

Figure 39
Sanity check

```
#check for missing values
cols_with_neg1 = []
for c in accidents_uk.columns:
    count = accidents_uk.filter(col(c) == -1).count()
    if count > 0:
        cols_with_neg1.append(c)
        print(f"Column {c} has {count} rows with a value of -1")
```

```
# Check for null values
null_cols = [c for c in accidents_uk.columns if accidents_uk.filter((col(c).isNull()) | (col(c) == "NULL")).co

print("Columns with null values:", null_cols)

# check for NaN values
nan_cols = [c for c in accidents_uk.columns if accidents_uk.filter(isnan(col(c))).count() > 0]
print("Columns with NaN values:", nan_cols)

# check for empty or blank strings
blank_cols = [c for c in accidents_uk.columns if accidents_uk.filter((length(trim(col(c))) == 0)).count() > 0]
print("Columns with empty or blank strings:", blank_cols)
```

```
Columns with null values: []
```

```
Columns with NaN values: []
```

```
[Stage 4582:>                                                       (0 + 8) / 8]
```

```
Columns with empty or blank strings: []
```

The final merged file has more than 550,000 rows and 41 columns.

Figure 40
Merged file

```
accidents_uk = accidents_uk.drop('accident_index')

# Find the size of the DataFrame
num_rows = accidents_uk.count()
num_cols = len(accidents_uk.columns)

# Print the size of the DataFrame
print(f"Accidents_uk has {num_rows} rows and {num_cols} columns.")
```

```
[Stage 7660:>                                                       (0 + 8) / 8]
```

```
Accidents_uk has 586216 rows and 41 columns.
```

2. Machine Learning :

Classification :

The aim is to classify the accident severity.

1 – Fatal accidents
2 – Serious accidents
3 – Slight/minor accidents

This part covers data preparation, testing the model by predicting the test set, and, finally, estimating the model's performance and tweaking the hyper-parameters to enhance the model.

***3.1 Class Imbalance**:* For minor accidents, there are over 400,000 rows, and for serious accidents, there are over 100,000 rows. Only 7951 occurrences comprise the "1" category, which represents the most serious incidents. One way to handle the imbalanced data is to downsample so that the training will be on a low subset of the majority class examples.

Figure 41
Count unique values

```
from pyspark.sql.functions import count
severity_counts = accidents_uk.groupBy('accident_severity').agg(count('*').alias('count'))
severity_counts.show()
```

```
[Stage 202:>                                                        (0 + 8) / 8]
```

```
+-----------------+------+
|accident_severity| count|
+-----------------+------+
|                1|  7951|
|                3|471834|
|                2|106431|
+-----------------+------+
```

After downsampling, we get relatively equal proportion of each accident severity class.

Figure 42
Downsample

```
fractions = {1: 1, 2: .15, 3: 0.035}
downsampled_df = accidents_uk.sampleBy("accident_severity", fractions=fractions, seed=42)

# group by the column and count the number of occurrences of each value
counts = downsampled_df.groupBy('accident_severity').agg(count('*').alias('count'))

# show the results
counts.show()
```

```
[Stage 258:=================================================>       (7 + 1) / 8]
```

```
+-----------------+-----+
|accident_severity|count|
+-----------------+-----+
|                1| 7951|
|                3|16733|
|                2|15964|
+-----------------+-----+
```

***3.2 Encoding categorical features and feature selection**:* Since math is typically done using numbers, categorical data is encoded using numbers. If the data is not numerical, machine learning algorithms cannot run and process it. There are 2 string values in the dataset.

Figure 43
Schema

```
: downsampled_df.printSchema()

root
 |-- casualty_class: integer (nullable = true)
 |-- casualty_severity: integer (nullable = true)
 |-- car_passenger: integer (nullable = true)
 |-- bus_or_coach_passenger: integer (nullable = true)
 |-- pedestrian_road_maintenance_worker: integer (nullable = true)
 |-- casualty_type: integer (nullable = true)
 |-- vehicle_type: integer (nullable = true)
 |-- towing_and_articulation: integer (nullable = true)
 |-- vehicle_location_restricted_lane: integer (nullable = true)
 |-- junction_location: integer (nullable = true)
 |-- skidding_and_overturning: integer (nullable = true)
 |-- hit_object_in_carriageway: integer (nullable = true)
 |-- vehicle_leaving_carriageway: integer (nullable = true)
 |-- hit_object_off_carriageway: integer (nullable = true)
 |-- first_point_of_impact: integer (nullable = true)
 |-- vehicle_left_hand_drive: integer (nullable = true)
 |-- journey_purpose_of_driver: integer (nullable = true)
 |-- police_force: integer (nullable = true)
 |-- accident_severity: integer (nullable = true)
 |-- number_of_vehicles: integer (nullable = true)
 |-- number_of_casualties: integer (nullable = true)
 |-- day_of_week: integer (nullable = true)
 |-- local_authority_district: integer (nullable = true)
 |-- local_authority_highway: string (nullable = true)
 |-- first_road_class: integer (nullable = true)
 |-- first_road_number: integer (nullable = true)
 |-- road_type: integer (nullable = true)
 |-- junction_detail: integer (nullable = true)
 |-- pedestrian_crossing_human_control: integer (nullable = true)
 |-- pedestrian_crossing_physical_facilities: integer (nullable = true)
 |-- light_conditions: integer (nullable = true)
 |-- weather_conditions: integer (nullable = true)
 |-- road_surface_conditions: integer (nullable = true)
 |-- special_conditions_at_site: integer (nullable = true)
 |-- carriageway_hazards: integer (nullable = true)
 |-- urban_or_rural_area: integer (nullable = true)
 |-- did_police_officer_attend_scene_of_accident: integer (nullable = true)
 |-- hour: integer (nullable = true)
 |-- month: integer (nullable = true)
 |-- distance_km: double (nullable = true)
 |-- road_type_limit: string (nullable = false)
```

a) Indexer: StringIndexer transforms the string columns into numerical indices.

b) OneHotEncoder: This stage uses the OneHotEncoder transformer to convert the numerical indices from the previous stage into one-hot encoded vectors. The dataset will have 210 new columns added after this phase (Figure 44), which is a sizable quantity, so we further do a feature selection to choose the top 10 features.

Figure 44
Count categories

```
: unique_values = downsampled_df.select('road_type_limit').distinct().count()
  print("Number of unique values for column 'road_type_limit':", unique_values)

  unique_values = downsampled_df.select('local_authority_highway').distinct().count()
  print("Number of unique values for column 'local_authority_highway':", unique_values)

Number of unique values for column 'road_type_limit': 4
[Stage 519:>                                                        (0 + 8) / 8]
Number of unique values for column 'local_authority_highway': 206
```

c) Assembler: This stage uses the VectorAssembler transformer to combine the two one-hot encoded vectors from the previous stage into a single feature vector.

d) Feature selection : This stage uses the ChiSqSelector selector to select the top 10 features from the 'encoded_features' column based on their chi-squared statistics with respect to the target column 'accident_severity'.

Figure 45
Encoding

```
#Encoding categorical features and feature selection:
indexer = StringIndexer(inputCols=('local_authority_highway','road_type_limit'),
                        outputCols=('local_authority_highway_idx','road_type_limit_idx'),
                        handleInvalid='keep')

onehot = OneHotEncoder(inputCols=('local_authority_highway_idx','road_type_limit_idx'),
                       outputCols=('local_authority_highway_idx_en','road_type_limit_en'))

assembler = VectorAssembler(inputCols=['local_authority_highway_idx_en','road_type_limit_en'], outputCol='enco

selector = ChiSqSelector(numTopFeatures=10, featuresCol="encoded_features",
                        outputCol="selected_features", labelCol="accident_severity")
```

3.3 Handling numerical columns :

Scaling : The machine learning model used in this study is a decision tree. Decision trees does not require feature scaling because features are not sensitive to data variance (TowardsDataScience, 2020).

No other special processing is required for numerical columns.

3.4 Vectorizing: VectorAssembler transformer is used to combine all the columns (Numerical and vectorized categorical columns)into a single vector column.

Figure 46
Vectorize

```
numeric_cols = [col for col in downsampled_df.columns if col not in
                ['accident_severity', 'local_authority_highway', 'local_authority_highway_idx','road_type_limi
                 'road_type_limit_idx','local_authority_highway_idx_en','local_authority_highway_idx_en']]


assembler2 = VectorAssembler(inputCols=numeric_cols+['selected_features'], outputCol="assembled_features")
```

3.4 Classifier : Decision tree is used to predict the target variable.

Hypertuning parameters: A ParamGridBuilder is used to construct a grid of hyperparameters for the DecisionTreeClassifier. The maximum tree depth, the maximum number of discretization bins, and the impurity metric are some of these parameters.Performance of the model is evaluated using three folds of k-fold cross-validation using a CrossValidator for hyperparameter adjustments.

3.5 Pipeline : All the defined steps are passed in sequence to the pipeline now. The pipeline transforms the input data as it moves through each stage, and the output of one stage becomes the input of the following stage. The pipeline then uses the altered data to apply the estimator (DecisionTreeClassifier) and provides a model.

Figure 47
Pipeline

```
: # Define the stages of the pipeline

  #Encoding categorical features and feature selection:
  indexer = StringIndexer(inputCols=('local_authority_highway','road_type_limit'),
                          outputCols=('local_authority_highway_idx','road_type_limit_idx'),
                          handleInvalid='keep')

  onehot = OneHotEncoder(inputCols=('local_authority_highway_idx','road_type_limit_idx'),
                         outputCols=('local_authority_highway_idx_en','road_type_limit_en'))

  assembler = VectorAssembler(inputCols=['local_authority_highway_idx_en','road_type_limit_en'], outputCol='enco

  selector = ChiSqSelector(numTopFeatures=10, featuresCol="encoded_features",
                          outputCol="selected_features", labelCol="accident_severity")



  numeric_cols = [col for col in downsampled_df.columns if col not in
                  ['accident_severity', 'local_authority_highway', 'local_authority_highway_idx','road_type_limi
                   'road_type_limit_idx','local_authority_highway_idx_en','local_authority_highway_idx_en']]


  assembler2 = VectorAssembler(inputCols=numeric_cols+['selected_features'], outputCol="assembled_features")


  classifier = DecisionTreeClassifier(featuresCol='assembled_features', labelCol='accident_severity')


  # Define the pipeline
  pipeline = Pipeline(stages=[indexer, onehot, assembler, selector, assembler2, classifier])


  # Define the parameter grid
  param_grid = ParamGridBuilder() \
               .addGrid(classifier.maxDepth, [10, 15, 20]) \
               .addGrid(classifier.maxBins, [10,20,30]) \
               .addGrid(classifier.impurity, ['entropy', 'gini']) \
               .build()

  # Define the cross validator
  cv = CrossValidator(estimator=pipeline,
                      estimatorParamMaps=param_grid,
                      evaluator=MulticlassClassificationEvaluator(labelCol='accident_severity'),
                      numFolds=3)
```

3.6 Train/Test: The data is randomly split into a training set and a testing set after the preceding procedures. 70% of the data in this project are used for training, while the remaining 30% are utilized for testing.

3.7 Fit and predict: Finally, we fit and predict the model. Accuracy and F1 score are used to validate the results.

Figure 48
Model

```
acc_train, acc_test = downsampled_df.randomSplit([0.7,0.3],seed=10)

# Fit the pipeline with the cross validator
cv_pipeline = cv.fit(acc_train)
```

```
00]]
```

```
# Transform the new data using the pipeline and calculate metrics
predictions = cv_pipeline.transform(acc_test)
accuracy_evaluator = MulticlassClassificationEvaluator(labelCol='accident_severity', me
f1_evaluator = MulticlassClassificationEvaluator(labelCol='accident_severity', metricNa
accuracy = accuracy_evaluator.evaluate(predictions)
f1_score = f1_evaluator.evaluate(predictions)
print("Accuracy: %f" % accuracy)
print("F1 Score: %f" % f1_score)

# Show the confusion matrix
conf_matrix = predictions.groupBy('accident_severity').pivot('prediction', [1,2,3]).cou
conf_matrix.show()
```

```
Accuracy: 0.932994
F1 Score: 0.932697
[Stage 25847:================================================>        (8 + 1) / 9]
23/04/16 23:17:20 WARN RowBasedKeyValueBatch: Calling spill() on RowBasedKeyValueBatch
. Will not spill but return 0.
23/04/16 23:17:20 WARN RowBasedKeyValueBatch: Calling spill() on RowBasedKeyValueBatch
. Will not spill but return 0.
23/04/16 23:17:20 WARN RowBasedKeyValueBatch: Calling spill() on RowBasedKeyValueBatch
. Will not spill but return 0.
23/04/16 23:17:20 WARN RowBasedKeyValueBatch: Calling spill() on RowBasedKeyValueBatch
. Will not spill but return 0.
23/04/16 23:17:20 WARN RowBasedKeyValueBatch: Calling spill() on RowBasedKeyValueBatch
. Will not spill but return 0.
23/04/16 23:17:20 WARN RowBasedKeyValueBatch: Calling spill() on RowBasedKeyValueBatch
. Will not spill but return 0.
23/04/16 23:17:20 WARN RowBasedKeyValueBatch: Calling spill() on RowBasedKeyValueBatch
. Will not spill but return 0.
23/04/16 23:17:20 WARN RowBasedKeyValueBatch: Calling spill() on RowBasedKeyValueBatch
. Will not spill but return 0.
[Stage 25858:================================================>        (8 + 1) / 9]
+-----------------+----+----+----+
|accident_severity|   1|   2|   3|
+-----------------+----+----+----+
|                1|2112| 160| 100|
|                3|  42|  88|4931|
|                2|  97|4333| 330|
+-----------------+----+----+----+
```

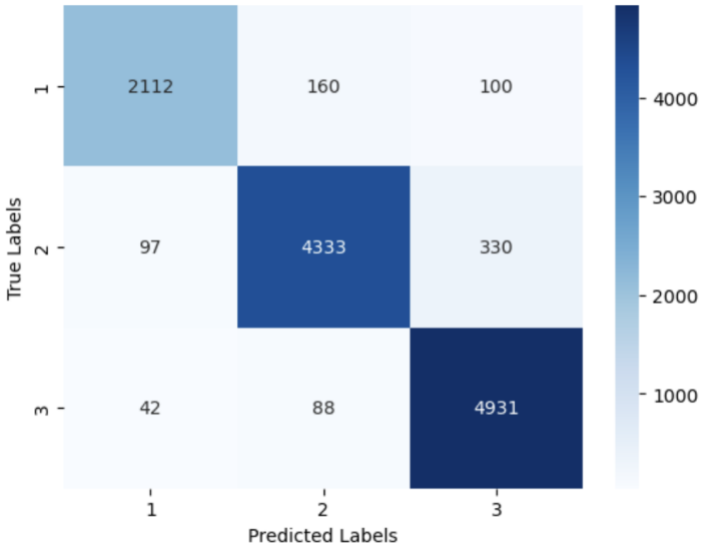
Results : The summary of the results are shown below.

Figure 49
Results

Overall Accuracy	Overall F1 Score
93.2994%	93.2697%

Accident Type	Recall	Precision
Minor [3]	0.974313	0.919791
Serious [2]	0.910294	0.947933
Fatal [1]	0.893456	0.938740

Figure 50
Confusion Matrix



Clustering :

To highlight accident hotspots, we can cluster (in Tableau) based on the total number of accidents. 7 clusters were produced by tableau. Birmingham has the most accidents overall (Figure 52) and has been assigned into a separate cluster.
A study by auto insurance specialists who examined Department of Transport data in all local authorities throughout the West Midlands, reveals that Birmingham has more car accidents than anywhere else in the region with 2.26 accidents per 1,000 people (Forte, 2022).

Figure 51
Accident Hotspots