Predicting Accident Severity Using Spark Big Data

Project Report for Course IS 789 (Big Data Fundamentals and Techniques), Spring 2025

Team Gryffindor: Praveesha Gongura, Tejaswini Jaladurgam, Prathamesh Bhosale

Department of Information Systems, University of Maryland, Baltimore County

Abstract

Every day, road accidents impact countless lives, and knowing how serious an accident might be, even before it happens, can help save time, lives, and resources. In this project, we used Apache Spark to process a massive dataset of over 2 million UK traffic accident records and built a machine learning model that predicts how severe an accident is likely to be. We focused on classifying accidents into two groups: "Slight" or "Serious_to_Fatal". By giving more importance to identifying serious accidents (recall), our goal is to help emergency teams respond faster and smarter. Our models include Logistic Regression, Random Forest, and Gradient Boosting, which were chosen for their performance and ability to deal with imbalanced data. We also tested different Spark configurations and adjusted the number of partitions. This allowed us to analyse the change in execution times and improve the scalability.

1. Introduction

Car accidents can range from a minor collision to a more serious one that results in serious injuries or even death. There are thousands of incidents in the United Kingdom alone each year, and knowing how serious an accident is before it happens or even during it might literally save your life. Emergency crews would be able to respond more efficiently, and city planners would be able to implement more safety precautions if they could predict accident severity. But with many variables at play, such as weather, lighting, road conditions, and time of day. It is difficult to make these predictions by hand.

We used Apache Spark to work through a massive dataset of over 2 million records of UK traffic accidents and applied machine learning in order to classify accidents as "Slight" or "Serious_to_Fatal". The hope is to transform raw accident data into useful information that saves lives. We wanted to create a great system for identifying serious accidents, even if it captures a few false positives along the way. This focus on recall guarantees that no significant cases are left out. Our goal is to make wiser planning, faster response, and ultimately safer roads possible.

2. Objectives

- Building a smart system using Apache Spark that can look at past traffic data and predict how serious a road accident might be.
- Clean and pre-process a large UK road accident data set with time, location, weather, and road conditions data.
- Since serious accidents are much less common than slight ones, the dataset is highly imbalanced. To make predictions more reliable, we grouped the accidents into two categories: "Slight" and "Serious_to_Fatal". This not only simplifies the prediction task but also helps the model focus better on identifying the more critical cases.
- Use models like Logistic Regression, Random Forest, and Gradient Boosting to make predictions.

- Compare the classification performance of each model, especially where the dataset is unbalanced.
- Prioritize weighted recall and F1 scores so that we record as many serious accidents as possible, even at the cost of tolerating some false alarms.
- Experiment with different combinations of executor memory, executor cores, total executors, and number of nodes to analyze scalability.
- Also, partitioning should be implemented to optimize parallel processing and reduce execution time.

3. Related Work

Yang et al. used a Random Forest model to predict traffic accident severity based on a large national dataset, focusing on feature importance and model optimization to improve prediction accuracy [4]. A broader study published in *PLOS ONE* compared various machine learning methods and found that Random Forest consistently performed well in classifying accident severity and identifying the most influential factors [3].

Zhang et al. explored ensemble models like XGBoost and Random Forest to predict accident severity in large datasets, especially where class imbalance was challenging. Their study demonstrated that ensemble methods can be used to enhance prediction accuracy [5]. To prove that comprehensible models are not excluded from generating useful information, Abellán et al. used decision tree-based methods to find rule-based patterns in accident severity [1].

To address the issue of imbalanced data in our own dataset, we referred to the SMOTE technique introduced by Chawla et al., which generates synthetic examples for the minority class to help the model learn more evenly [2]. While we ultimately used downsampling instead of SMOTE, the underlying principle guided our efforts to improve the model's ability to detect serious accidents.

4. Problem Statement

Road accidents result in severe injuries daily, and knowing beforehand how severe an accident could be saves lives. Numerous variables determine how severe an accident will be, including road conditions, weather, lighting conditions, seasons, and location. These factors are complex and hard to predict. With over 2 million records available, we aim to use this historical accident data to identify the features that lead to more severe accidents and build a model that can forecast accident severity. The model will classify the level of the accident that has taken place into "Slight" or "Serious_to_Fatal".

The resulting model will be a severity-predicting model that predicts the accidents into categories, which will allow the authorities to plan and allocate better. We will stress the importance of using recall together with the F1 score as the primary evaluation measure, since it is important to predict "Serious_to_Fatal" accidents to provide an on-time response. By minimizing false negatives, the model can help authorities prioritize critical incidents and save lives.

5. Dataset

Link to dataset: https://www.kaggle.com/datasets/tsiaras/uk-road-safety-accidents-and-vehicles

The Accident Information dataset, downloaded from Kaggle, contains traffic accident information in the UK from 2005 to 2017. It is in CSV format and consists of 34 columns and over 2 million records of accident severity, weather, road types, and casualties. Kaggle provides two datasets: Accident_Information.csv and Vehicle_Information.csv. We used the Accident_Information.csv file for this project. It has detailed information about an accident, like the Accident_Index location, time, severity, and road conditions, etc. We shall use this data to predict the severity of accidents and improve safety measures.

5.1 Variables and Value Types

The following table categorizes the variables based on their data types to understand the dataset better. This classification helps identify the type of analysis that can be performed on each variable.

Temporal Variables: Represent time-based information such as dates and times.

1. Categorical Variables: Represent different categories or classes.
2. Numeric Variables: Represent continuous numerical values.
3. Spatial Variables: Represent geographical or location-related data.

Below is a table that categorizes all variables in the dataset according to their type:

Temporal	Categorical	Numeric	Spatial
Date	Accident_Severity	Number_of_Vehicles	Location_Easting_OSGR
Time	Accident_Index	Number_of_Casualties	Location_Northing_OSGR
	LSOA_of_Accident_Location	1st_Road_Number	Longitude
	Local_Authority_(Highway)	Speed_limit	Latitude
	Day_of_Week	2nd_Road_Number	
	Local_Authority_(District)	Year	
	1st_Road_Class		
	Road_Type		
	Junction_Detail		
	Junction_Control		
	2nd_Road_Class		
	Pedestrian_Crossing-Human_Control		
	Pedestrian_Crossing-Physical_Facilities		
	Light_Conditions		
	Weather_Conditions		
	Road_Surface_Conditions		
	Special_Conditions_at_Site		
	Carriageway_Hazards		

	Urban_or_Rural_Area Did_Police_Officer_Attend_Scene_of_Accident Police_Force InScotland		

Table 1: The types of variables in the dataset

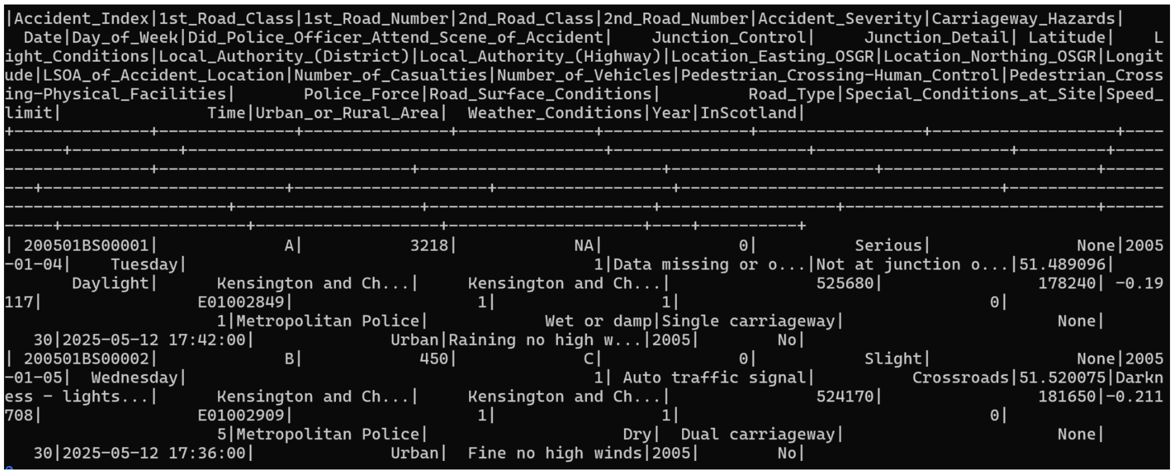
5.2 Data Property

This data set is identically distributed since every row has the same pattern (same attributes), and this data is collected in similar ways by the UK government (provided by Kaggle). Second, we consider the data as independent in the sense that a particular accident won't have any direct cause from another. Although there are trends, especially concerning time and location (e.g., more during rush hour or at specific spots), these trends are indirect dependencies, with the accidents being more likely to happen under specific circumstances but not violating the independence condition for any given data point.

This data has spatial as well as temporal attributes:

- Spatial: Latitude, Longitude, Location_Easting_OSGR, and Location_Northing_OSGR specify the geographical locations of the accidents.
- Temporal: Time and Date variables record accidents at specific times, and thus allow analysis by time.

5.3 Sample of the Dataset



```
|Accident_Index|1st_Road_Class|1st_Road_Number|2nd_Road_Class|2nd_Road_Number|Accident_Severity|Carriageway_Hazards|
  Date|Day_of_Week|Did_Police_Officer_Attend_Scene_of_Accident|    Junction_Control|     Junction_Detail| Latitude|    L
ight_Conditions|Local_Authority_(District)|Local_Authority_(Highway)|Location_Easting_OSGR|Location_Northing_OSGR|Longit
ude|LSOA_of_Accident_Location|Number_of_Casualties|Number_of_Vehicles|Pedestrian_Crossing-Human_Control|Pedestrian_Cross
ing-Physical_Facilities|       Police_Force|Road_Surface_Conditions|         Road_Type|Special_Conditions_at_Site|Speed_
limit|               Time|Urban_or_Rural_Area|  Weather_Conditions|Year|InScotland|
+--------------+--------------+---------------+--------------+---------------+-----------------+-------------------+----
------+-----------+-------------------------------------------+--------------------+--------------------+---------+-----
--------------+--------------------------+-------------------------+---------------------+----------------------+------
---+-------------------------+--------------------+------------------+---------------------------------+---------------
-----------------------+-------------------+-----------------------+------------------+--------------------------+------
-----+-------------------+-------------------+--------------------+----+----------+
| 200501BS00001|             A|           3218|            NA|              0|          Serious|               None|2005
-01-04|    Tuesday|                                          1|Data missing or o...|Not at junction o...|51.489096|
       Daylight|      Kensington and Ch...|     Kensington and Ch...|               525680|                178240| -0.19
117|               E01002849|                   1|                 1|                                0|
                      1|Metropolitan Police|            Wet or damp|Single carriageway|                      None|
   30|2025-05-12 17:42:00|              Urban|Raining no high w...|2005|        No|
| 200501BS00002|             B|            450|             C|              0|           Slight|               None|2005
-01-05|  Wednesday|                                          1| Auto traffic signal|          Crossroads|51.520075|Darkn
ess - lights...|      Kensington and Ch...|     Kensington and Ch...|               524170|                181650|-0.211
708|               E01002909|                   1|                 1|                                0|
                      5|Metropolitan Police|                    Dry|  Dual carriageway|                      None|
   30|2025-05-12 17:36:00|              Urban|  Fine no high winds|2005|        No|
```

Figure 1: The first five rows of the Accident_Information dataset

6. Pre-Processing

6.1 Handling Missing Values:

The first step was to address the missing data. We replaced missing values with NaN to get the data in a consistent format. Then, we looked at how much data was missing in each column. Some columns had so much missing data that they would not be useful to us in our analysis, so we dropped those. For example, columns like 2nd_Road_Class, Carriageway_Hazards, LSOA_of_Accident_Location, and Special_Conditions_at_Site were dropped because they had too many missing values.

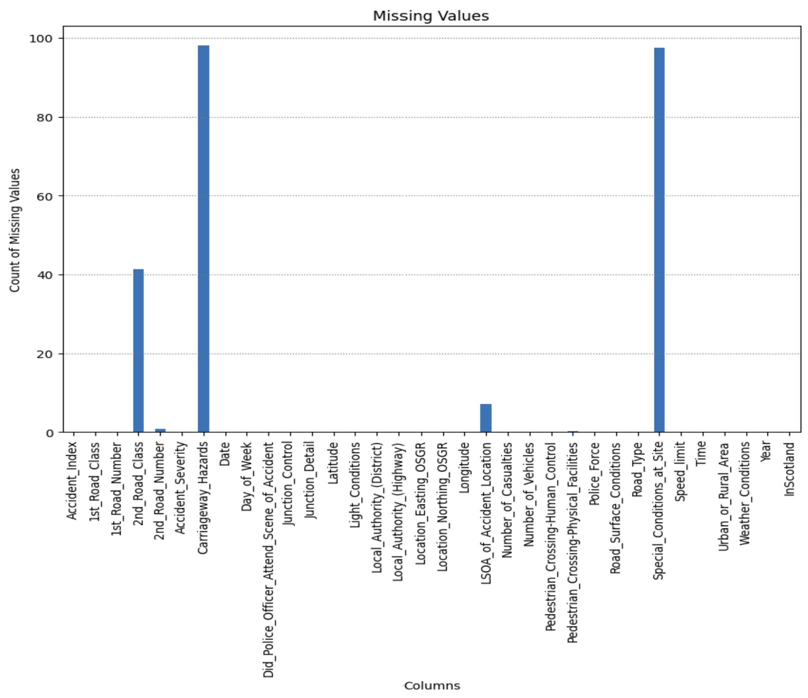


Figure 2: A bar chart showing the percentage of missing values in each column of the dataset

6.2 Dropped Irrelevant Features:

Some columns in the dataset were removed because they didn't contribute to predicting accident severity:

- **Accident_Index:** This column was dropped because it is a unique identifier for each accident. Encoding it would create numerous columns with alphanumeric values, which would not help with the prediction and would make the dataset unnecessarily large.
- **Police_Force, Did_Police_Officer_Attend_Scene_of_Accident**: These columns were removed because they refer to the police attendance report made after the accident. They do not provide helpful information for predicting the accident's severity.
- **Pedestrian_Crossing-Physical_Facilities, Pedestrian_Crossing-Human_Control:** These columns were highly imbalanced, with more than 80% of the instances showing "No facilities within 50m," and did not contribute to the prediction, so they were removed.
- **LSOA_of_Accident_Location, Local_Authority_(Highway), Local_Authority_(District):** These columns contained many unique values. Encoding them would have added too many features to the dataset, increasing complexity and the risk of overfitting. Therefore, they were removed to keep the dataset simple.

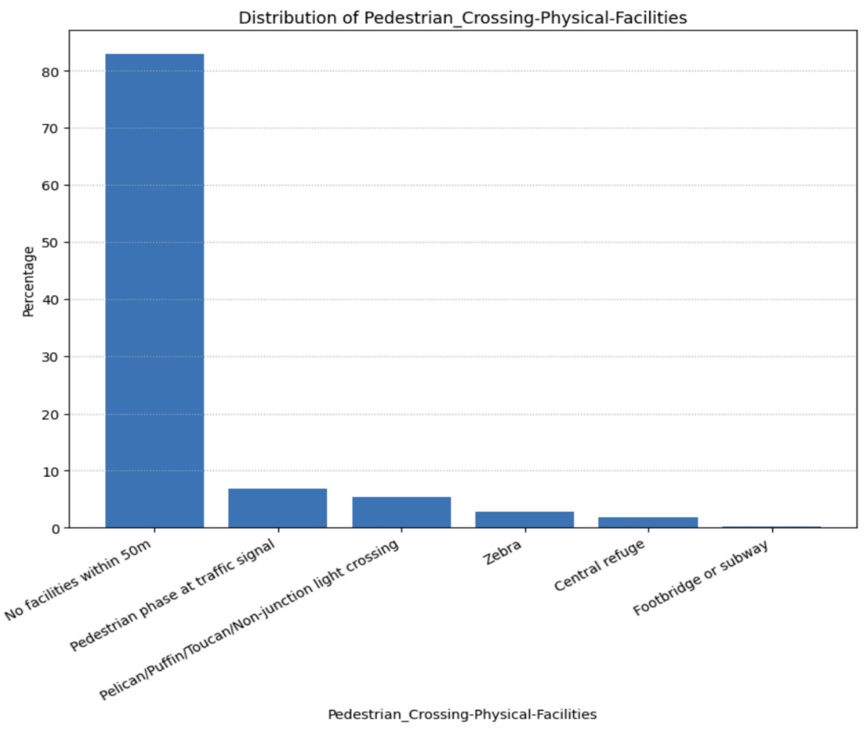


Figure 3: Distribution of Pedestrian_Crossing-Physical_Facilities. Over 80% of the instances in this feature are labeled "No facilities within 50m

6.3 Encoding and Imbalanced Data :

To prepare the dataset for analysis, we handled categorical features and addressed imbalances:

- Urban_or_Rural_Area: Rows with “Unallocated” values were dropped, and the feature was encoded with 0 for “Urban” and 1 for “Rural” to train the model.
- Road_Surface_Conditions: Categories like “Wet or Damp”,” Frost or Ice”, “Snow”, and “Flood over 3cm deep” were grouped into one category called “Slippery”, which was then encoded as 0 for “Dry” and 1 for “Slippery”.
- Weather_Conditions: Various weather conditions, such as “Raining no high winds”, “Snowing, Fog, or mist”, were grouped into “Intense”, while “Fine no high winds” were labeled as “Normal”. The feature was encoded as 0 for “Normal” and 1 for “Intense”.
- Light_Conditions: All Darkness conditions, including “Darkness - lights lit”, “Darkness - no lighting”, “Darkness - lighting unknown”, and “Darkness - lights unlit”, were combined into one category called “Darkness”. The feature was encoded as 0 for “Daylight” and 1 for “Darkness”.
- Junction_Control: Categories such as “Stop sign”,” Authorised person”, and “Not at junction or within 20 meters” were grouped as “Non-signal control”. The feature was then encoded into three categories: 0 for “Give way or uncontrolled”, 1 for “Auto traffic signal” and 2 for “Non-signal control”.

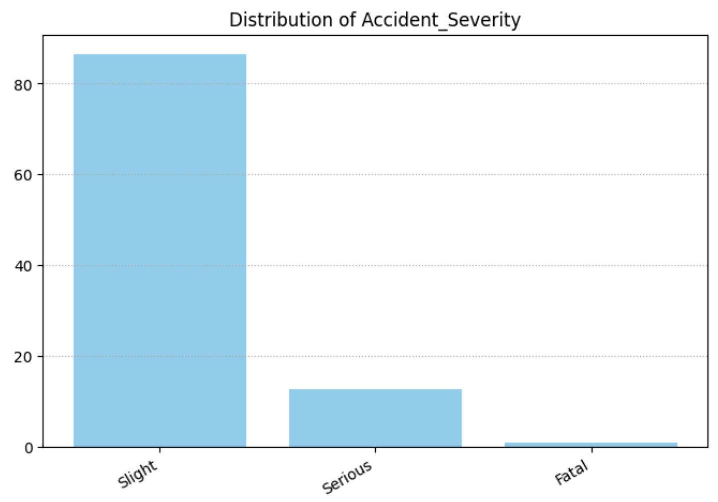


Figure 4: This is an example of how the "Accident_Severity" feature was used to balance the unbalance between accident types. Instead of keeping Serious and Fatal accidents as separate categories, they were grouped in one category called "Serious_to_Fatal". To make the data easier to model, the feature was encoded as 0 for “Slight” and 1 for "Serious_to_Fatal".

6.4 Feature Creation

Feature creation is creating new features from existing data to improve model performance. This can include creating new features by applying a transformation to the existing features, problem domain features from understanding the problem domain, and interaction features by combining a number of features in order to represent interactions between features [6]. The techniques are helpful for revealing patterns and relationships that cannot be seen in the original features.

- Time: We split the Time feature into two columns: Accident_Hour and Accident_Minute. This helps us look at patterns related to the time of day when accidents happen.
- Date: The Date feature was broken down into Day and Month to better understand trends based on specific days or months of the year.
- Season: We created a new attribute based on the month of the accident, grouping the months into four seasons: Winter, Spring, Summer, and Autumn. We then encoded these seasons with numerical values, where Winter was assigned 0, Spring 1, Summer 2, and Autumn 3, allowing the model to process the seasons numerically.

7. Data Exploration

Data exploration is an important part of finding the patterns and structure of a dataset. It brings out important features and relationships required for model training. In this project, we used Apache Spark to explore data for the Accident_Information dataset. This benefited us in processing and handling the large dataset as we performed intensive exploratory data analysis. The most important findings that we were able to discover through our exploration are shown below:

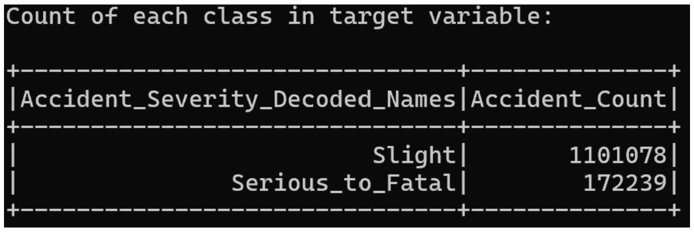
SQL queries were run, which retrieved the required data from the Temporary View created on the Dataframe. The following command created the TempView required to directly query data with SQL.

df_transform.createOrReplaceTempView("accidents")

7.1 Target Variable Analysis:

Running the query below, we observed an imbalance in the distribution of the target variable, suggesting that we need to be cautious when evaluating the model.

spark.sql("""SELECT Accident_Severity_Decoded_Names, COUNT(*) AS Accident_Count FROM accidents GROUP BY Accident_Severity_Decoded_Names ORDER BY Accident_Count DESC""").show()



```
Count of each class in target variable:

+------------------------------+--------------+
|Accident_Severity_Decoded_Names|Accident_Count|
+------------------------------+--------------+
|                        Slight|       1101078|
|              Serious_to_Fatal|        172239|
+------------------------------+--------------+
```

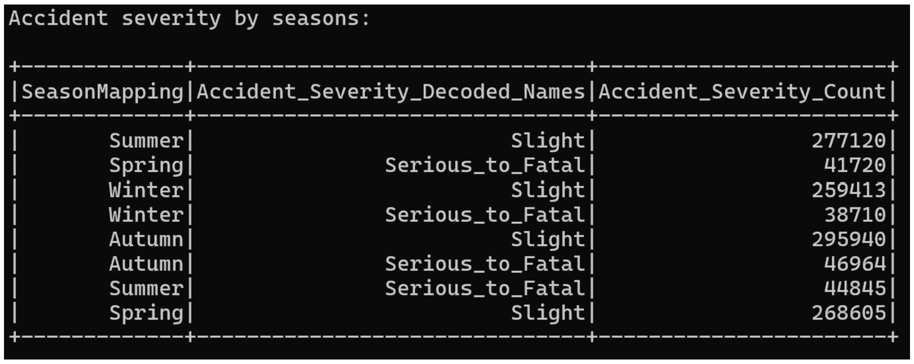
Figure 5: This table illustrates the distribution of the Target variable (Accident_Severity)

7.2 Feature Analysis

7.2.1 Accident Severity by Seasons

Running the following query shows that the most number of accidents were recorded in the "Autumn" season, with 295,940 "Slight" accidents and 46,964 "Serious_to_Fatal" accidents.

spark.sql("""SELECT CASE WHEN Season=0 THEN 'Winter' WHEN Season=1 THEN 'Spring' WHEN Season=2 THEN 'Summer' WHEN Season=3 THEN 'Autumn' ELSE 'Unknown' END AS SeasonMapping, Accident_Severity_Decoded_Names, COUNT(*) AS Accident_Severity_Count FROM accidents GROUP BY SeasonMapping, Accident_Severity_Decoded_Names""").show()



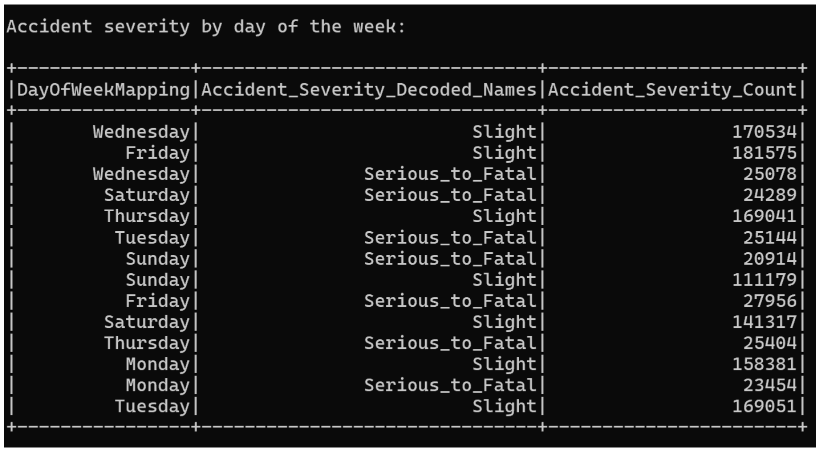
```
Accident severity by seasons:

+-------------+------------------------------+-----------------------+
|SeasonMapping|Accident_Severity_Decoded_Names|Accident_Severity_Count|
+-------------+------------------------------+-----------------------+
|       Summer|                        Slight|                 277120|
|       Spring|              Serious_to_Fatal|                  41720|
|       Winter|                        Slight|                 259413|
|       Winter|              Serious_to_Fatal|                  38710|
|       Autumn|                        Slight|                 295940|
|       Autumn|              Serious_to_Fatal|                  46964|
|       Summer|              Serious_to_Fatal|                  44845|
|       Spring|                        Slight|                 268605|
+-------------+------------------------------+-----------------------+
```

Figure 6: This table illustrates the distribution of accident severity across different seasons.

7.2.2 Accident Severity by Day of Week

Running the query below shows that the highest number of accidents was reported on "Friday", with 181,575 "Slight" accidents and 27,956 "Serious_to_Fatal" accidents. "Wednesday" follows closely with 170,534 "Slight" accidents and 25,078 "Serious_to_Fatal" accidents. The data suggests that "Friday" and "Wednesday" are the most critical days, showing a higher frequency for each type of accidents.

spark.sql("""SELECT CASE WHEN Day_of_Week=0 THEN 'Monday' WHEN Day_of_Week=1 THEN 'Tuesday' WHEN Day_of_Week=2 THEN 'Wednesday' WHEN Day_of_Week=3 THEN 'Thursday' WHEN Day_of_Week=4 THEN 'Friday' WHEN Day_of_Week=5 THEN 'Saturday' WHEN Day_of_Week=6 THEN 'Sunday' ELSE 'Unknown' END AS DayOfWeekMapping, Accident_Severity_Decoded_Names, COUNT(*) AS Accident_Severity_Count FROM accidents GROUP BY DayOfWeekMapping, Accident_Severity_Decoded_Names""").show()



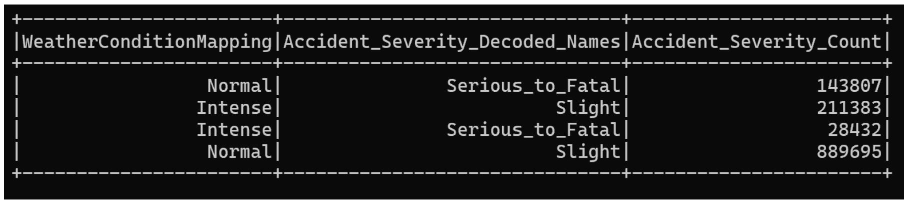
```
Accident severity by day of the week:

+----------------+------------------------------+----------------------+
|DayOfWeekMapping|Accident_Severity_Decoded_Names|Accident_Severity_Count|
+----------------+------------------------------+----------------------+
|       Wednesday|                        Slight|                170534|
|          Friday|                        Slight|                181575|
|       Wednesday|             Serious_to_Fatal|                 25078|
|        Saturday|             Serious_to_Fatal|                 24289|
|        Thursday|                        Slight|                169041|
|         Tuesday|             Serious_to_Fatal|                 25144|
|          Sunday|             Serious_to_Fatal|                 20914|
|          Sunday|                        Slight|                111179|
|          Friday|             Serious_to_Fatal|                 27956|
|        Saturday|                        Slight|                141317|
|        Thursday|             Serious_to_Fatal|                 25404|
|          Monday|                        Slight|                158381|
|          Monday|             Serious_to_Fatal|                 23454|
|         Tuesday|                        Slight|                169051|
+----------------+------------------------------+----------------------+
```

Figure 7: This table illustrates the distribution of accident severity by day of the week.

7.2.3 Accident Severity by Weather Conditions

Running the query below shows that accidents that occurred in "Normal" weather conditions have a higher count of "Slight" accidents (889,695), and a significant number of "Serious_to_Fatal" accidents (143,807). The data shows that weather conditions significantly affect the severity of accidents and that "Normal" weather conditions lead to more severe accidents.

spark.sql("""SELECT CASE WHEN Weather_Conditions=0 THEN 'Normal' WHEN Weather_Conditions=1 THEN 'Intense' ELSE 'Unknown' END AS WeatherConditionMapping, Accident_Severity_Decoded_Names, COUNT(*) AS Accident_Severity_Count FROM accidents GROUP BY WeatherConditionMapping, Accident_Severity_Decoded_Names""").show()



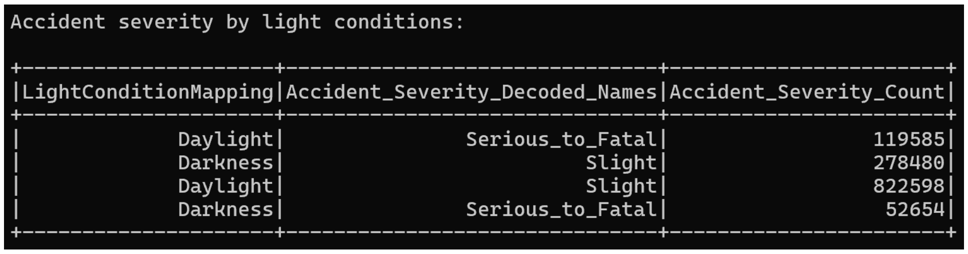
```
+-----------------------+------------------------------+----------------------+
|WeatherConditionMapping|Accident_Severity_Decoded_Names|Accident_Severity_Count|
+-----------------------+------------------------------+----------------------+
|                 Normal|             Serious_to_Fatal|                143807|
|                Intense|                        Slight|                211383|
|                Intense|             Serious_to_Fatal|                 28432|
|                 Normal|                        Slight|                889695|
+-----------------------+------------------------------+----------------------+
```

Figure 8: This table illustrates the distribution of accident severity by weather condition

7.2.4 Accident Severity by Light Conditions

Running the following query shows that accidents occurring in "Daylight" have a higher count of "Slight" accidents 822,598, and "Serious_to_Fatal" accidents are also significant under "Daylight" conditions 119,585. The data suggests that "Daylight" conditions cause more "Slight" accidents, while "Darkness" causes both "Slight" and "Serious_to_Fatal" accidents.

spark.sql("""SELECT CASE WHEN Light_Conditions=0 THEN 'Daylight' WHEN Light_Conditions=1 THEN 'Darkness' ELSE 'Unknown' END AS LightConditionMapping, Accident_Severity_Decoded_Names, COUNT(*) AS Accident_Severity_Count FROM accidents GROUP BY LightConditionMapping, Accident_Severity_Decoded_Names""").show()



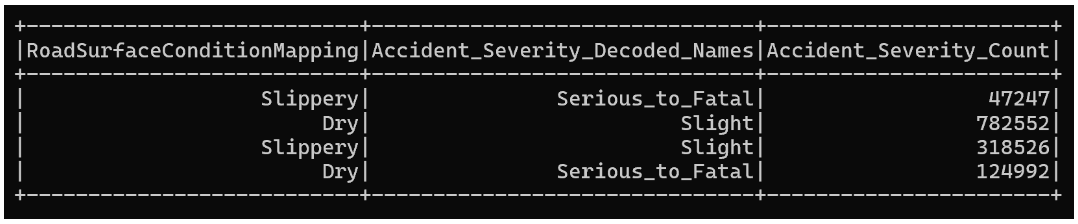
```
Accident severity by light conditions:

+---------------------+-------------------------------+-----------------------+
|LightConditionMapping|Accident_Severity_Decoded_Names|Accident_Severity_Count|
+---------------------+-------------------------------+-----------------------+
|             Daylight|                Serious_to_Fatal|                 119585|
|             Darkness|                          Slight|                 278480|
|             Daylight|                          Slight|                 822598|
|             Darkness|                Serious_to_Fatal|                  52654|
+---------------------+-------------------------------+-----------------------+
```

Figure 9: This table illustrates the distribution of accident severity by light conditions.

7.2.5 Accident Severity by Road Surface Conditions

Upon running the below code on the DataFrame API the below results were obtained. We see that " Dry" conditions contribute to a high count of both "Serious_to_Fatal" accidents (124,992) and "Slight" accidents (782,552). The data indicates that "Dry" conditions are associated with a significant number of both "Serious_to_Fatal" and "Slight" accidents, with slight accidents occurring much more frequently.

df_transform.withColumn("RoadSurfaceConditionMapping", when(col("Road_Surface_Conditions")==0,"Dry").when(col("Road_Surface_Conditions")==1,"Slippery").otherwise("Unknown")).groupBy("RoadSurfaceConditionMapping","Accident_Severity_Decoded_Names").agg(count("").alias("Accident_Severity_Count")).show()*



```
+---------------------------+-------------------------------+-----------------------+
|RoadSurfaceConditionMapping|Accident_Severity_Decoded_Names|Accident_Severity_Count|
+---------------------------+-------------------------------+-----------------------+
|                   Slippery|                Serious_to_Fatal|                  47247|
|                        Dry|                          Slight|                 782552|
|                   Slippery|                          Slight|                 318526|
|                        Dry|                Serious_to_Fatal|                 124992|
+---------------------------+-------------------------------+-----------------------+
```

Figure 10: This table illustrates the distribution of accident severity by road surface conditions

8. Methods and Experiments

8.1 Classification Problem Framing

The prediction of Accident_Severity is a supervised learning problem, as the model is trained on the labeled dataset that contains accident details and their associated accident severity. The target variable, i.e., Accident_Severity, is a categorical label. This makes it a classification task. Specifically, this is a binary classification task where we predict one of two classes:

- **Slight:** Minor accidents with low injury risk.
- **Serious_to_Fatal:** A class that merges the original "Serious" and "Fatal" categories to address data imbalance.

The Logistic Regression, Random Forest, and Gradient Boosting models were selected as a mix of baseline, ensemble, and boosting strategies, specifically under class imbalance. Model training was conducted using *pyspark.ml* on the preprocessed data. Models were trained directly on Spark DataFrames containing cleaned and transformed features. VectorAssembler was used to assemble those features into a single feature vector. The *fit* method executed the training. All the models were trained using the train and test split method with 70% train data and 30% test data.

8.2 Logistic Regression Experiment

Logistic Regression served as the baseline model. It was selected for its simplicity, interpretability, and suitability for binary classification. No explicit hyperparameters were tuned in this model. We used the basic settings provided by PySpark. This was done so as the goal was to establish a baseline performance for comparison with more complex models.

lr = LogisticRegression(featuresCol="features", labelCol=target)

8.3 Random Forest Experiment

Random Forest is an ensemble learning algorithm that builds multiple decision trees. It is helpful in case of imbalance. It is particularly useful for problems with many features and can handle complex relationships in the data [8]. The following parameters were chosen after several experiments on tuning the parameters manually:

rfc=RandomForestClassifier(labelCol=target, featuresCol="attributes", numTrees=150, maxDepth=12, minInstancesPerNode=5, featureSubsetStrategy="log2", impurity="entropy", seed=42)

The model uses 150 trees, each with a maximum depth of 12. This will prevent overfitting and also capture enough complexity. The model ensures each split in the tree has at least five examples. The "log2" featureSubsetStrategy will select a random subset of features for every split to add diversity. Furthermore, the model uses "entropy" as an impurity measure. This splits the nodes based on information gain[7].

8.4 Gradient Boosting Experiment

Gradient Boosting builds a series of simple models, usually decision trees. Every new tree tries to correct the mistakes made by previous ones. This process continues until the model's performance stops improving. This model is effective for handling complex and imbalanced datasets. This is because it focuses on classes that are hard to classify [9]. GBT was chosen as an advanced model for this very reason. The following parameters were chosen to maximize the classification performance.

gbt = GBTClassifier(labelCol=target, featuresCol="attributes", maxIter=150, maxDepth=8, stepSize=0.05, minInstancesPerNode=10, subsamplingRate=0.6, seed=42)

The model is configured with 150 boosting iterations to improve performance. The tree depth is set to 8 to avoid overfitting. A step size of 0.05 was used to help the model learn more gradually, which also helps prevent overfitting by using a smaller learning rate. Each tree is trained with at least

10(*minInstancesPerNode=10*) data points in each split to avoid trees that may not generalize well. Furthermore, 60% of the data is used to train each tree, adding randomness [10].

9. Challenges

9.1 Imbalanced Dataset and Misleading Accuracy

Initial training runs revealed unexpectedly high accuracy, even though the models performed poorly in minority classes. Upon examining the confusion matrix, it became clear that:

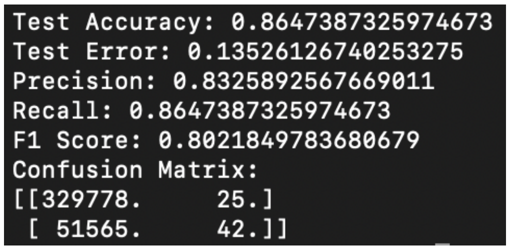
- Random Forest predicted only the majority class (Slight). This resulted in True Positives = 0 and False Positives = 0
- Logistic Regression predicted TP = 42 and FP =25 for the minority class.

Despite grouping "Serious" and "Fatal" classes to "Serious_to_Fatal" in the preprocessing stage to handle imbalance, it was attributed that the imbalance still played a role in these results.

Distribution after grouping the classes:

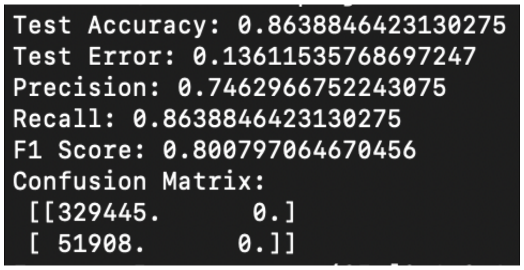
- Slight: 1,101,078
- Serious_to_Fatal: 172,239

Below are the results obtained for each model trained. The results included test accuracy, which measures test data accuracy, and test error. Weighted precision, weighted F1 scores, and weighted recall were also computed for each model, along with a confusion matrix. The results below are obtained on the preprocessed dataset before downsampling. These results highlight that accuracy masks off the True Positives and False Positives.



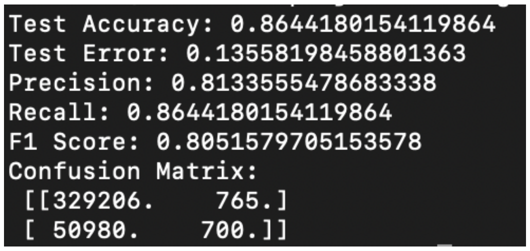
```
Test Accuracy: 0.8647387325974673
Test Error: 0.13526126740253275
Precision: 0.8325892567669011
Recall: 0.8647387325974673
F1 Score: 0.8021849783680679
Confusion Matrix:
[[329778.     25.]
 [ 51565.     42.]]
```

Figure 11: Evaluation results of the Logistic Regression model.



```
Test Accuracy: 0.8638846423130275
Test Error: 0.13611535768697247
Precision: 0.7462966752243075
Recall: 0.8638846423130275
F1 Score: 0.800797064670456
Confusion Matrix:
 [[329445.      0.]
 [ 51908.      0.]]
```

Figure 12: Evaluation results of the Random Forest model.



```
Test Accuracy: 0.8644180154119864
Test Error: 0.13558198458801363
Precision: 0.8133555478683338
Recall: 0.8644180154119864
F1 Score: 0.8051579705153578
Confusion Matrix:
 [[329206.    765.]
 [ 50980.    700.]]
```

Figure 13: Evaluation results of the Gradient Boosting model.

9.2 Downsampling Strategy

To address the imbalance, we applied random downsampling to reduce the majority of class instances in a 1:2 ratio. This was done to achieve the rebalance of the data.

After downsampling, the distribution is as follows:

- Slight: 344,478
- Seriou_to_Fatal: 172,239

The models were retrained using the downsampled dataset.

10. Results

10.1 Evaluation Metrics

Model performance evaluation was performed using Spark's *pyspark.mllib* - MultiClassMetrics as well as *pyspark.ml* - MultiClassClassificationEvaluator. Spark DataFrame was converted to an RDD before using the MultiClassMetrics. The following metrics were computed:

- Accuracy: It measures the total correct predictions made by the model. It may not be the best metric for imbalanced datasets.
- Weighted Precision: It is the average of the precision for each class. Each class's precision is weighted by the number of instances in that particular class.
- Weighted Recall: It is the average of the recall for each class. Each class is weighted by the number of instances in that particular class.
- Weighted F1 Score: It combines precision and recall using the harmonic mean. It is weighted by the number of instances in each class.
- Confusion Matrix: It shows how many predictions were true positives, true negatives, false positives, and false negatives to evaluate the model's performance in detail.

For computing accuracy, the *MultiClassClassificationEvaluator* from *pyspark.ml* was used [14]. Evaluation metrics such as *weightedPrecision, weightRecall, weightedFMeasure(),* and *confusionMatrix()* were obtained from PySpark's *MultiClassMetrics* class within the *pyspark.mllib*. The expressions *metrics.weightedPrecision and metrics.weightedRecall* are implemented as Python properties and are part of the PySpark API. These provide access to the calculated values. This means they behave like attributes but internally execute methods that compute the weighted precision and recall scores [11], [12], [13].

The *weightedFMeasure()* method computes the weighted F1 score across all classes. This method allows an optional beta parameter [11]. The *confusionMatrix()* is a function call in PySpark's evaluation API. It returns a confusion matrix as a Spark matrix object. The *toArray()* method was used to convert that matrix to a regular NumPy-style array for easy interpretation and display.

10.2 Post-Downsampling Metrics

Models	**Accuracy**	**Weighted Precision**	**Weighted Recall**	**Weighted F1 score**
Logistic Regression	0.672603582	0.643550072	0.672603582	0.575827091
Random Forest	0.681026272	0.658116094	0.681026272	0.606706213
Gradient Boosting	0.682801516	0.655424907	0.682801516	0.642059113

Table 2: Comparison evaluation results of Logistic Regression, Random Forest, and Gradient Boosting models post downsampling.

Logistic Regression		
	0	1
0	99954 (TN)	2829 (FP)
1	47703 (FN)	3859 (TP)

T**able 3:** Confusion matrix of the Logistic Regression model post downsampling.

Random Forest		
	0	1
0	98060 (TN)	4723 (FP)
1	44509 (FN)	7053 (TP)

Table 4: Confusion matrix of the Random Forest model post downsampling.

Gradient Boosting		
	0	1
0	92443 (TN)	10340 (FP)
1	38618 (FN)	12944 (TP)

Table 5: Confusion matrix of the Gradient Boosting model post downsampling.

There was a significant drop in accuracy and weighted metrics. Despite the classification performance of all the models being a bit better than random guessing, it shows that the model is learning to classify the minority class better post-downsampling. All the models perform slightly better than random guessing. Almost all three models have obtained similar results. Of the three models, Gradient Boosting performs better with a better F1 score, and the confusion matrices show that the model has fewer false negatives than the others. This is especially important while dealing with imbalanced data cases.

11. Performance

11.1 Spark Configurations

The entire project is implemented using PySpark on the Taki cluster, part of UMBC's High Performance Computing Facility (HPCF). We explored the effect of resource allocation on runtime performance. We experimented with different combinations of Spark configurations, including executor memory, executor cores, and total executor cores. By adjusting these parameters across multiple runs, we could identify configurations that yielded lower execution time.

Models	**Configurations**	**Nodes**	**Executor Memory**	**Total Executors**	**Executor Cores**	**Execution Time**
Logistic Regression	Config1	3	14g	18	3	0m52.520s
Random Forest						1m36.211s
Gradient Boosting						8m58.887s
Logistic Regression	Config2	3	10g	9	4	0m55.471s
Random Forest						1m47.783s
Gradient Boosting						7m41.703s
Logistic Regression	Config3	2	8g	6	3	1m9.544s
Random Forest						2m1.848s

Gradient Boosting						7m43.199s

Table 6: Execution times for Logistic Regression, Random Forest, and Gradient Boosting models with a change in the number of nodes, executor-memory, total-executors, and executor-cores.

The *Config1* has more memory and executor count than the other settings. It was observed that both the Logistic Regression and Random Forest models executed faster in the *Config1* scenario. However, these models took a maximum execution time under the *Config3* setting, which had fewer resources. The GBT achieved a minimum execution time in the *Config2* setting. This could be because the GBT trains trees one after another, and each tree depends on the previous one. In settings with too many resources, Spark might have spent more time managing them instead of speeding things up. *Config 2* is likely performing better because it has an optimal balance of nodes, memory, and executor cores.

11.2 Partitioning

The table below shows the performance of the Gradient Boosting model.

Partitions	23	20	12	8	4
Gradient Boosting Model	8m17.079s	7m42.952s	7m44.434s	8m43.256s	8m30.527s

Table 7: Execution times of the Gradient Boosting with changing number of partitions

In Spark, the data is divided into partitions so it can be processed in parallel. We tested how the number of partitions affects the execution time for the Gradient Boosting model. We manually set the number of partitions using repartition(). For example, to set the number of partitions to 4, repartition(4) was used. We conducted experiments on the Gradient Boosting model and plotted a line graph shown below. We took the resource configuration (Nodes - 2, Executor Memory - 8g, Total Executors - 6, Executor Cores - 3).

It was observed that using the partitioning did not optimize the execution time much. The table shows how changing the number of partitions affected the execution time of the Gradient Boosting model. The model ran the fastest when we experimented with 20 partitions, taking about 7 minutes and 43 seconds. Slightly higher or lower partitions (12, 23) gave similar but slightly slower results. However, when the number of partitions was reduced to 8 and 4, the model took more time to run (over 8 minutes). The below graph is plotted for Number of Partitions vs Executime Time (in minutes). It shows that using too many or too few partitions can slow performance. Based on the results, 12 and 20 partitions performed relatively well.

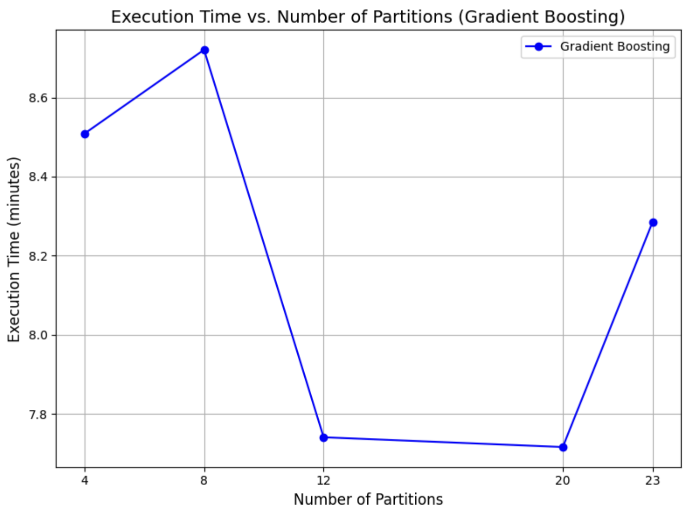


Figure 14: Execution Time vs. Number of Partitions for the Gradient Boosting model.

12. Conclusion

We used PySpark on the Taki cluster to build and test three models: Logistic Regression, Random Forest, and Gradient Boosting to predict accident severity. In the beginning, we noticed that the accuracy was not a reliable metric, as the dataset was very imbalanced. After applying downsampling, the results became balanced, and the weighted metrics made more sense. All three models showed similar performance patterns, but Gradient Boosting gave slightly better results. It gave a good balance with around 64% weighted F1 score and was consistent across other weighted metrics. In terms of execution, Logistic Regression and Random Forest performed better with more memory and executors (Config1). However, Gradient Boosting ran faster in Config2, with optimal balance. We also found that the partitioning technique reduced the execution time. Changing the number of partitions did not significantly improve the execution time. The model ran fastest with 20 partitions.

13. Future Work

In the future, this project can be extended by using the Synthetic Minority Oversampling Technique instead of downsampling. Also, neural networks can be explored to improve classification performance. Using Spark Structured Streaming to predict accident severity in real time is helpful for live traffic systems. Additionally, a dashboard to display predictions and insights in a user-friendly way would make the results interpretable and useful for decision makers.

Work Division

1. **Praveesha Gongura -** I worked on a part of the data preprocessing, which included handling missing values and identifying and removing unhelpful attributes. I also wrote the code and trained the Random Forest and Gradient Boosting models. I also worked on experimenting with the models before and after the downsampling. I worked on evaluation metrics and also tuned parameters through multiple experiments. And I also addressed class imbalance using downsampling. Additionally, I worked on experimenting with different configurations(Random

Forest and Gradient Boosting models) and final report documentation (Methods and Experiments, Challenges, Results, Performance, Conclusion, and Future Work).

2. **Tejaswini Jaladurgam -** I worked on data pre-processing by dropping irrelevant features, addressing class imbalance by grouping similar categories, and encoding the variables accordingly. I also performed feature extraction and used both Spark SQL queries and the DataFrame API for data exploration. I wrote code for the model of logistic regression and its training, performed both logistic regression and gradient boosting configuration, partition handling, and final report documentation(Dataset, Pre-processing, Data exploration).
3. **Prathamesh Bhosale -** I worked on the writing of the Abstract, Introduction, and Objectives parts of the report, which provided a clear explanation of the goals and purpose. In order to expand the Related Work section, I additionally located and added relevant academic sources. I worked with several Random Forest and Gradient Boosting model setups, such as changing executors and Spark nodes, to see how they impacted the model's resource consumption and performance.

References

[1] J. Abellán, G. López, and J. De Oña, "Analysis of traffic accident severity using decision rules via decision trees," *Expert Systems with Applications*, vol. 40, no. 15, pp. 6047–6054, 2013. https://doi.org/10.1016/j.eswa.2013.05.027

[2] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002. https://doi.org/10.1613/jair.953

[3] *PLOS ONE*, "Evaluating the effectiveness of machine learning techniques in predicting traffic accident severity," 2023. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC10407198/ [Accessed: May 7, 2025]

[4] J. Yang, S. Han, and Y. Chen, "Prediction of traffic accident severity based on Random Forest," *Journal of Advanced Transportation*, Feb. 2023. [Online]. Available: https://www.researchgate.net/publication/367987656_Prediction_of_Traffic_Accident_Severity_Based_on_Random_Forest [Accessed: May 10, 2025]

[5] Y. Zhang, A. Haghani, and M. Hamedi, "Using ensemble machine learning models to predict the severity of traffic accidents," *Accident Analysis & Prevention*, vol. 157, p. 106160, 2021. https://doi.org/10.1016/j.aap.2021.106160

[6] A. Omoseebi., G. Ola, and J. Tyler, "*Data Preparation and Feature Engineering*", *ResearchGate, Feb.2025.* [Online]. Available: (https://www.researchgate.net/publication/389860294_Data_Preparation_and_Feature_Engineering) [Accessed: March 27, 2025]

[7] Apache Spark, "pyspark.ml.classification.*Randomforestclassifier*", [Online]. Available: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.classification.RandomForestClassifier.html [Accessed: March 30, 2025]

[8] Wikipedia contributors, "*Random Forest*", Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Random_forest [Accessed: May 10, 2025]

[9] Wikipedia contributors. (n.d). "*Gradient boosting*", Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Gradient_boosting [Accessed: May 11, 2025]

[10] Apache Spark. "*pyspark.ml.classification.GBTClassifier*". [Online]. Available: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.classification.GBTClassifier.html [Accessed: April 13, 2025]

[11] Apache Spark, "pyspark.mllib.evaluation.MulticlassMetrics", [Online]. Available: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.mllib.evaluation.MulticlassMetrics.html [Accessed: April 10, 2025]

[12] Apache Spark, "*evaluation.py - PySpark MLlib Evaluation Module*", *GitHub*, [Online]. Available: https://github.com/apache/spark/blob/master/python/pyspark/mllib/evaluation.py [Accessed: May 11, 2025]

[13] Python Software Foundation, "*Built-in Functions - property*", Python 3 Documentation, [Online]. Available: https://docs.python.org/3/library/functions.html#property [Accessed: May 11, 2025]

[14] Apache Spark, "pyspark.ml.evaluation.MulticlassClassification Evaluation", [Online]. Available: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.evaluation.MulticlassClassificationEvaluator.html. [Accessed: March 30, 2025].