Data Intensive Computing Project Phase 3 Report

Project: Eco-Friendly but is it safe? Analysis and Predicting Trends in E-vehicle Accidents

Group members:

Abhijeet Sanjiv Bonde (50624352)

Raghav Potdar (50631527)

Hui Yu (50311663)

Date: 12 April 2025

Data Sources:

The data used for this project is the NYC Motor Vehicle Collisions (Crashes) dataset, which is available for free of charge on the NYC Open Data Portal. The New York City Police Department (NYPD) keeps this data record which has extensive records of motor vehicle crashes that have been reported in New York City. The timeframe for the dataset is from June 2012 to Feb 2025 (Updated on a regular basis).

Dataset Overview

- Source: NYC Open Data (NYPD)
- URL: NYC Motor Vehicle Collisions Dataset
- Number of Records: Over 2 million crash reports (updated regularly)
- Number of Columns: 29+ features that cover various aspects of each collision

Problem Statement

With the increasing adoption of E-scooters/E-bikes as a convenient and eco-friendly mode of transportation, concerns regarding their safety have emerged. This project aims to analyze and predict accident trends using historical motor vehicle collision data. Specifically, we will investigate the severity of E-scooter/E-bikes accidents compared to other vehicle types, identify key contributing factors, and determine high-risk locations and periods for accidents.

Distributed Data Cleaning/Processing:

Initially, we start by importing all the PySpark modules that are required for us to create a SparkSession.builder. Then we load the dataset that we have into the PySpark DataFrame using the spark.read.csv() function. This will help us to read the data from the CSV file into the data frame, here we have carried the headers with “header=True” and data types using “inferSchema=True”. [1] Once all the data has been loaded, we now check its structure using the printSchema() to understand what data is present in each column, as this will help us to format the columns that need formatting or standardization. The size of the data frame is df.count(), which is a RDD operation, which will print the number of rows that are present and use the len(df.columns) for printing the number of columns.

1. **Filling missing data to null or 0 [2]**
   a. Contributing Factors - Since there is a lot of garbage data in our dataset like ‘nan’, ‘None’, ‘’, ‘ ’, ‘N/A’, we converted all such values to ‘unspecified’.
   b. Death and Injuries - We filled NA values with 0 as there were no deaths or injuries in the accident.
   c. Latitude and Longitude - We fill NA values with 0 as we require this step for future processes where we will fill the latitude and longitude data from external sources.

      Spark: We used the withColumn to update the current column in the RDD’ containing the garbage which we recognize by using the when(), isin() functions. The col() function is used to refer to the column that we are working on. Once isin() function detects that an unwanted value is present, which is done by when() function, it replaces that garbage value with ‘None’. Then using the .otherwise() operator we define the else condition if there is None, then replace it with ‘unspecified’.

2. **Shifting Attributes**
   a. Car Type - If VEHICLE TYPE CODE 1 is missing but there is data present in TYPE CODE columns 2 to 5, then we shift the first available value to TYPE CODE 1 to make sure that primary vehicle is prioritised.
   b. E-Vehicle - We shift E-Vehicles to VEHICLE TYPE CODE 1 when they are in other columns than 1 to prioritise E-Vehicles. [3]

      Spark: We use PySpark to backfill the missing values in the ‘VEHICLE TYPE CODE’ columns. We start by finding this word in the column name then by looping, we apply WithColumn() to update it. We use When().otherwise() to check if the current column is null using the isNull(). If Null is there, then we fill the values from the next column using col(vehicle_columns[i+1]), or else we keep the original value. This will make sure that things run in a parallel manner.

3. **Changing Case**
   a. Combining the "ON STREET NAME", "CROSS STREET NAME" and "OFF STREET NAME" to a new field called "Addresses - We use external api to fill the

details that are missing in the original dataset. To make the addresses full, we merge the 3 street name columns into one and append new york at the end of the field.

b. Vehicle Types - To maintain a consistency we convert all the Vehicle Type code columns to lowercase.

c. Contributing Factors - To standardize the data we convert it to lowercase.

Spark: We have used withColumn to create and update columns, we use concat_ws() which is a concatenate using a separator to combine street names into a single string for the Addresses column. Then we use lower() to convert it to lowercase and then regexp_replace() to clean the extra spaces that were present.

4. **Fixing date and time** - We do this to bring the time format to a single and standardized format.

Spark: We use the withColumn to create a new column called 'CRASH DATE & TIME', then we use concat_ws(), which joins function in spark to join 2 columns with a space in between and then we use to_timestamp() to convert the timestamp into a single time format.

5. **Removing Duplicate Values** - There are few duplicate values in our dataset that we need to drop to avoid inconsistency in the data in the modelling part.

Spark: We can directly use the dropDuplicates() function to remove duplicate rows based on 'CRASH DATE & TIME', 'LATITUDE' and 'LONGITUDE'.

6. **Creating new columns**

a. Create a new column called 'is_e_vehicle' - Based on if E-Vehicle is involved in the crash we define a new column where we set it to True or False.

Spark: We use withColumn, which is a Spark Dataframe operation to create a new column called 'Time of Day' based on our extracted information from 'CRASH DATE & TIME' using the hour() function. We use functions like when and otherwise to classify into different timings of the day. So all these operations are processed in parallel in spark partitions.

7. **Changing DataTypes**

a. Zip Code - Convert the various data types into a single integer datatype.
b. Number of Persons Injured - Converting the decimal values into integer type.
c. Number of Persons killed - Converting the decimal to integer because if the person is alive then it will be counted as injured and not a person killed in decimal.

Spark: We use withColumn with a combination of cast() function to convert the data types to integers. The col function is also used to detect a column and cast(IntegerType()) is used to change the datatype to integer.

8. **Dropping Columns**

a. ON STREET NAME, CROSS STREET NAME, OFF STREET NAME – As we discussed in step 3.a we have already created a column that has combined information of these 3 columns.
b. Reordering the CRASH DATE & TIME - Here we will put the CRASH DATE & TIME column first while we keep other columns in their original order.

   Spark: We remove the unwanted columns using the drop() function. First we create a list of columns that need to be dropped then using the unpacking operator '*' we pass these to the drop() function which drops these columns from the dataframe. After this is done, we again create a list where 'CRASH DATE & TIME' is placed at the beginning and then we place all the other columns, then we use the select() function in spark to rearrange these columns according to this new order. Here select() function allows us to pinpoint the exact sequence of columns that we want to work with.

9. **Filling missing values of Latitude and Longitude using external API**

a. NLat, NLong, Location - Create a new column using NOMINATIM API to fetch latitude, longitude and cleaned address. This API handles 1 request/second per IP.
b. BOROUGH - Then we parse the data from 9.a to fill out the NYC boroughs.
c. LATITUDE, LONGITUDE, LOCATION - We parse the data from 9.a to fill missing values of these fields which are fetched using API.
d. ZIP CODE -For all the missing ZIP CODEs entries we extract the zip code from the location field of the detached addresses.
e. ZIP CODE and BOROUGH (LOCAL INSTANCE) - Here we use the local Nominatim to get more ZIP and BOROUGH, and missing values to approximately 800 entries.

   Spark: We first load the DataFrame using the operation called read.option().csv(), then we use withColumn to add and modify the columns and then we also use the 'lit' to insert constant values. We also use when() to update the values. Then we use a filter to separate the rows that have missing data for the above-mentioned columns which is a transformation operation. We also define our own function using and apply it using UDF(), which fetches the data from the API. At the end, we write the DataFrame using the write.option().csv() method. This will generate a lot of files as they are running in parallelly.

10. **Standardizing vehicle types using domain knowledge** - Here we group all the similar and misspelled values into a smaller range of vehicles like ''suv', 'sedan', 'ambulance', 'passenger vehicle', 'truck', etc.'

Spark: Here we define a user-defined function which will standardize the predefined mappings. Then we apply this UDF to multiple columns using the withColumn(). The col() function is used to identify columns within withcolumn(). We also use the StringType() to define the return type for the UDF to make sure that it fits well with the Spark DataFrame schema.

Algorithms/Visualizations

We have implemented a total of 8 models using PySpark. Also, for classification, we have used SMOTE for class rebalancing and performed comparative analysis between unbalanced and balanced data.

SMOTE - SMOTE (Synthetic Minority Over-sampling Technique) is a popular technique used to address class imbalance in classification tasks. Instead of simply duplicating instances of the minority class, SMOTE generates new synthetic instances by interpolating between existing minority instances. This helps to balance the training dataset, allowing models to learn decision boundaries between classes more efficiently.

Classification Models

Things that have changed from phase 2:

1. Using the PySpark technique to run the modelling [4]
2. Reduce the number of classes to two (Causality and Non-Causality).
3. Tried rebalancing the classes using SMOTE (Synthetic Minority Over-sampling Technique).
4. Comparison of performance metrics between modelling from balanced data and unbalanced data.

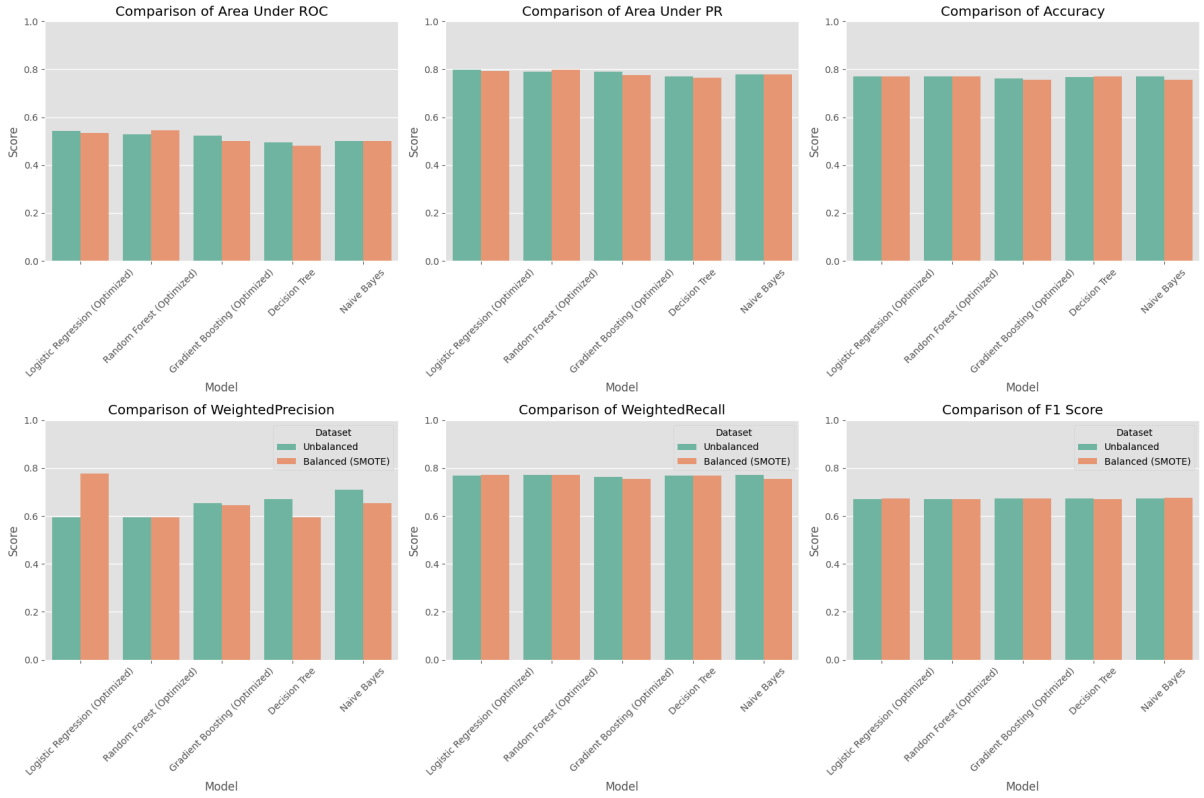
Logistic Regression - Logistic Regression is a linear binary classification model. It estimates the probability of a sample belonging to a certain class by passing a linear combination of input features through the logistic function.

Random Forest - Random Forest is an ensemble machine learning algorithm that trains numerous decision trees during training. Final class (or regression value) prediction is achieved by taking the majority vote (or mean, in regression) of the ensemble of trees.

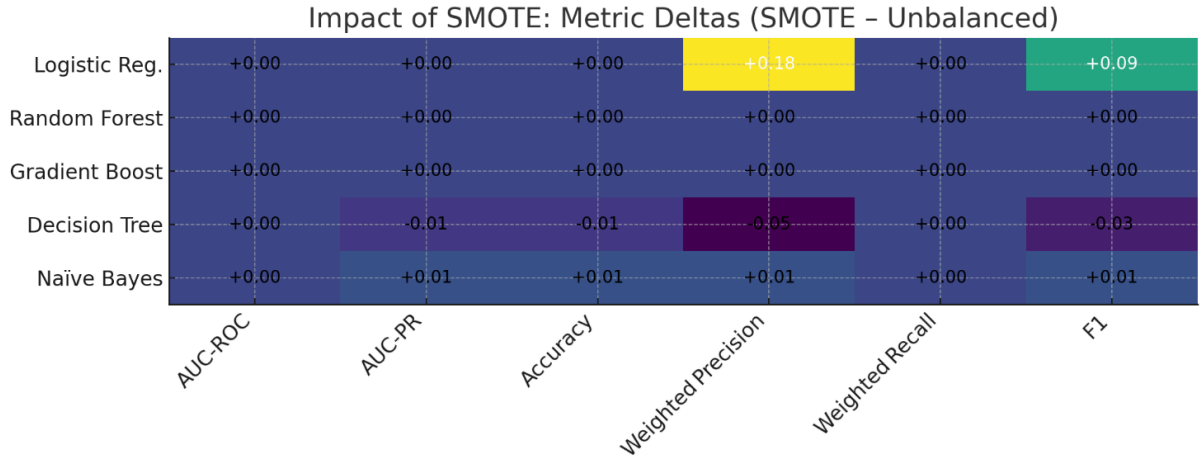
Gradient Boosting - Gradient Boosting builds a sequence of weak learners (typically decision trees) as an ensemble, with the new tree trying to predict the errors committed by the old trees. It minimizes a specified loss function.

Decision Tree - A Decision Tree is a non-parametric model that predicts the value of a target variable by learning simple decision rules inferred from the features. It splits the data into branches to form a tree-like structure.

Naive Bayes - Naive Bayes is a probabilistic classifier based on Bayes' theorem under the assumption that features are independent of each other in the context of the class label.



Results - Classification models have better results than regression models. The performance comparison of a number of classification models on balanced and unbalanced datasets gave largely comparable performances on key metrics. Although marginal improvements were noted in a few cases - most prominently in precision in certain models - overall, SMOTE utilization didn't enhance model effectiveness. The results indicate that class balancing had little impact.



To understand the graph above:

Colour change	What does it mean?	value
Indigo → Blue	Metric worsened after SMOTE	Negative delta's (-0.05)
Green → tealish	No major change	Approx zero delta's (=0)
Yellow → light yellow	Improvement after SMOTE	Positive delta's (+0.18)

Regression Models

Things that have changed from phase 2:

1. Using the PySpark technique to run the modelling
2. Created INJURY_SEVERITY column as a target variable. We have used the formula Number of persons injured + Number of persons killed x 5, keeping higher weightage if a person is killed.

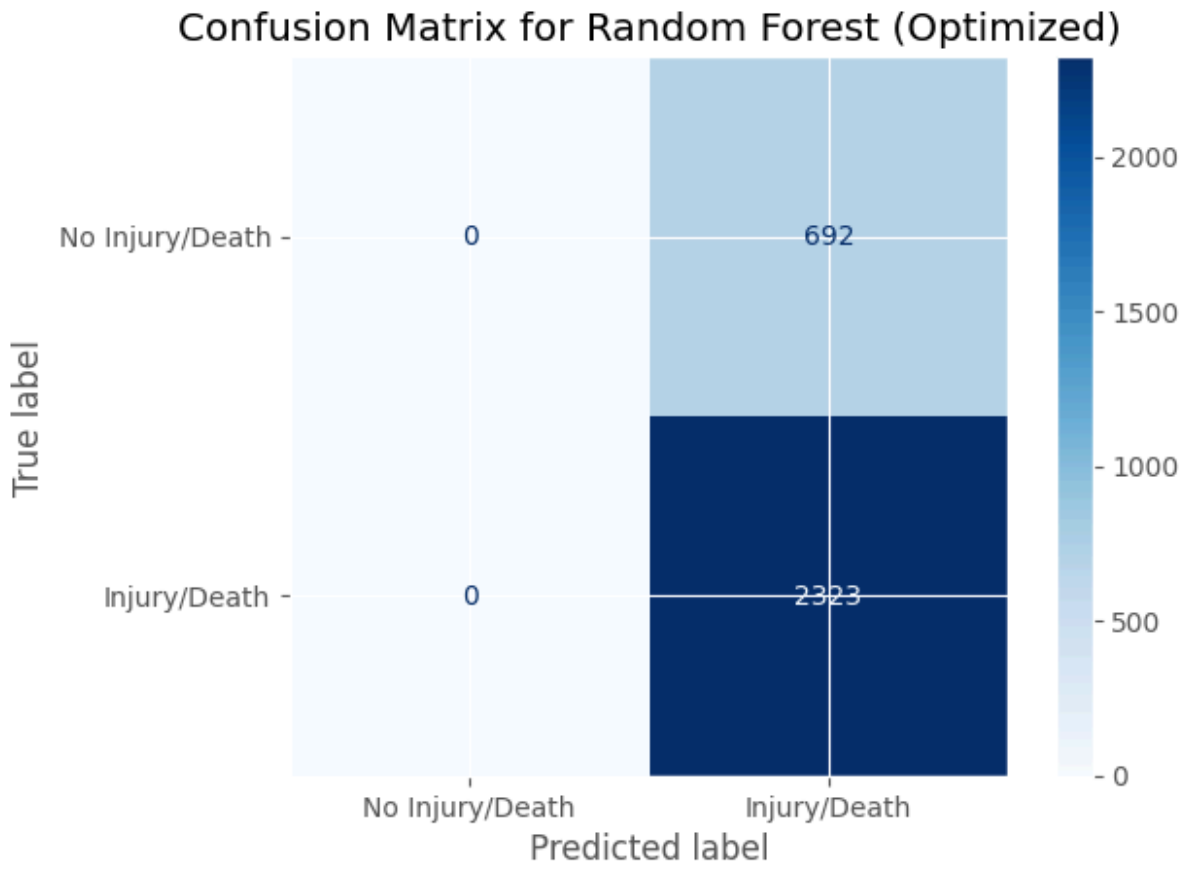
Linear Regression - Linear Regression is a basic statistical method used to model the relationship between an independent variable and one or more independent variables by using a linear equation.

Random Forest Regressor - Random Forest Regressor is an ensemble method that builds a set of multiple decision trees and provides a prediction as the average of the predictions of the individual trees.

Gradient Boosting Regressor - Gradient Boosting Regressor is another ensemble technique that builds models sequentially. Each subsequent model is fit to reduce the residual errors of previous models, and predictions are made by averaging the output of all models.

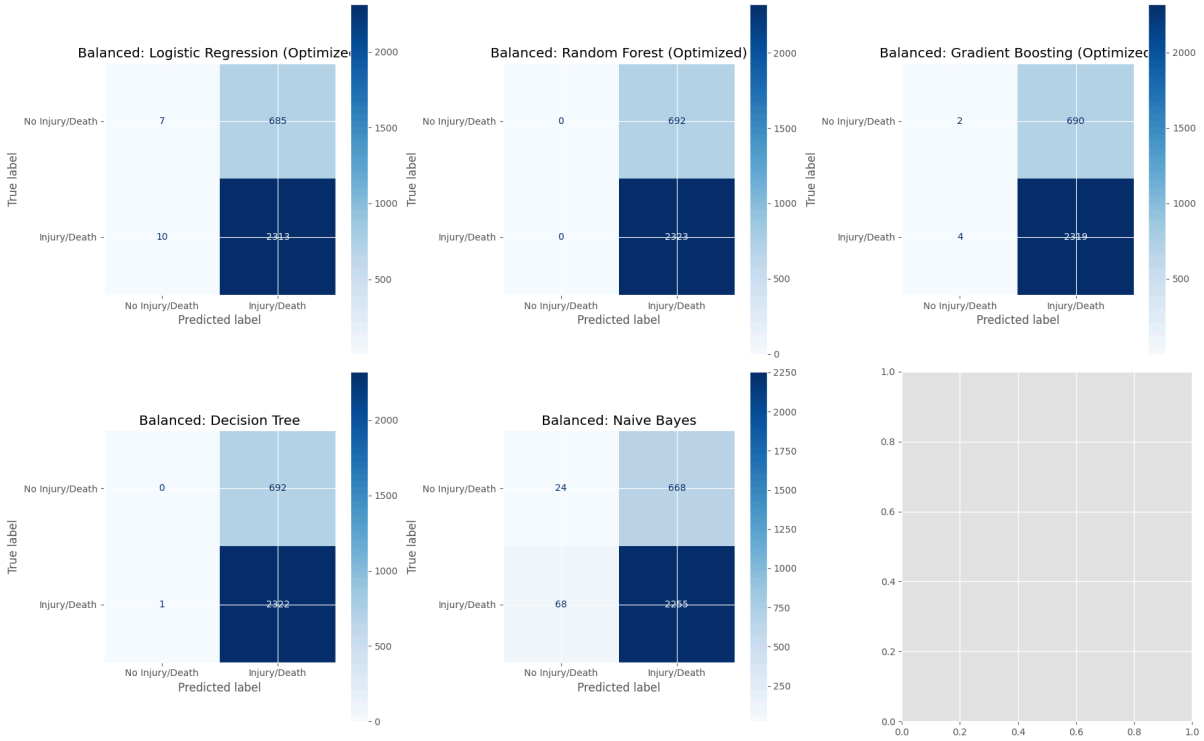
Results - This data has a huge class imbalance, and according to the output, it's better to use classification models to classify labels than to predict using regression.

.

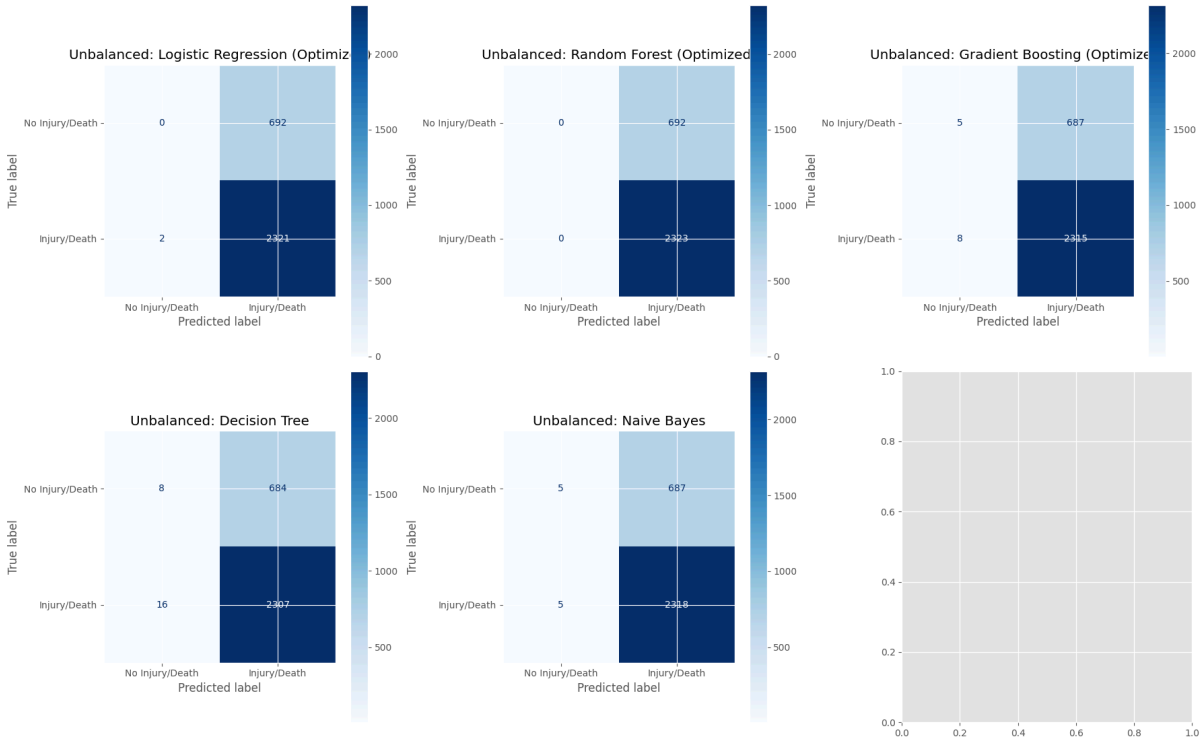


This is the confusion matrix for our best-performing model based on Area Under ROC(Receiver Operating Characteristic curve). For this confusion matrix, RandomForest flags every other sample as Injury or Death, which is why there are 0 true negatives and 692 false positives. Overall accuracy is around 77 percent, but the model is heavily biased towards the positive class, so it is unusable. This is because of the severe class imbalance that is present in the dataset, since a forest can maximize the accuracy and recall by always taking a label.

Confusion Matrices for Models Trained on Balanced Data



Confusion Matrices for Models Trained on Unbalanced Data



Explanation and Analysis:

DAG from Spark Application UI:

1. **Naive Bayes**

   For the Naive bayes, job number 1823 it runs a naive bayes step which finishes in 0.2 seconds. This job is completed in a single stage which is 3690 and then runs 96 tasks with 0 errors. It processed 2.1 MB out of input data and then produced 171.1 KB output. This was all done without needing to write to any disk (no spill as well). This workflow also involved a simple set of operations which was parallelized → mapPartitions → map, which was performed in the memory. THis is the reason it was fast as well as the efficient performance of naive bayes.

2. **Random Forest**

   For the RandomFOrest, job number 2042, shows that the RandomForest step finished in only 23 ms. The DAG has 2 stages though, Stage 4098 which processes 4.9 MB CSV data and performs mapPartitions transformations, whereas the stage 4099 executes a reduceByKey then collectAsMap, producing only 211 KB of data. All the tasks in each stage completed in 0 errors, and there were no errors or spill or skew, which show that the balanced partitioning and enough of executor memory was present. SInce the runtime is so small, the output is small as well. Also it can be said that this aggregation logic is tightly optimized.

3. **Decision Tree**

   This job number 2036 runs a scala aggregation, specifically DecisionTreeMetadata.scala. It also shows that the job finished in 2 seconds, processing 370 MB of data from CSV that too in a single stage (4087) which has 8 parallel tasks. The flow is CSV scan → deserialization → a series of mapPartition/map operations. This task finished with no shuffles or writes, which shows that it was computed in-memory that produced its results without spills.

4. **SQL DataFrame**

   The Query number 67, spark reads 1748161 rows from the 8 CSV files of 368 MB totally and then finishes the job in 2 seconds. A single stage handled about 12.1 seconds of the executor time around all its tasks. After the first filter the data was about 15226 rows, then it was cut into 3015 rows which also eliminated the need to shuffle or sort and then memory size was at 257 MB. There are no spills and or any sortings, which shows that the query is CPU based and not I/O bound.

5. **SparkJobs**

   This spark job was of 27 minutes which was interactive as well in which 2049 jobs were done in FIFO. Most of them were around 50ms. collectAsMap called RandomForest and DecisionTree pipelines, which involved 1 to 2 toPandas conversions and some of them were NaiveBayes collect operations. All the tasks were completed, which can show that the cluster was stable. The main time consuming factor here was that toPandas steps in this took a lot of time which made the job longer.

6. **CSV**

   This job completed successfully in 3 s, reading 55.4 MiB of CSV data and writing 62.5 MB of results; Adaptive Query Execution ruined the DAG to a single executed stage (Stage 5) with 9 evenly balanced tasks and whole-stage code generation, removing the shuffle stage that was planned. Also there were no skew or spills and indicating a small CPU-bound workload with no skew or spills.

Comparison of PySpark and Pandas on Performance:

- As we have changed the target for both classification and regression models, we don't have a one-to-one comparison for it. But, we can compare the time required for a certain process with the equivalent process using PySpark.
- It took 4 mins and 42 seconds to run KNN, Random Forest, and Gradient Boosting in a combined run for optuna. We are ignoring the time required for training all classification models, as it took 0 seconds to run.
- PySpark doesn't natively support KNN. For comparison, we are just taking Random Forest (3 mins 27 sec) and Gradient Boosting(5 mins 46 seconds).
- PySpark's performance advantage can help some models reduce their running time due to its ability to distribute the workload to multiple nodes. But overhead from distributed systems of pyspark could negate the advantage when running more complex or iterative processes, such as gradient boosting.

Comparison for Performance Metrics:

Accuracy:

PySpark Models (Logistic Regression, Random Forest, Gradient Boosting) were slightly more accurate (~0.769 – 0.770) compared to Pandas-based Random Forest and Gradient Boosting models (~0.736 – 0.740).

Precision:

PySpark models were more precisely weighted precision (~0.59 – 0.68) compared to Pandas RF/GB (~0.73 – 0.74), but this is due to differences in averaging types (macro vs. weighted).

F1 Score:

PySpark models performed slightly higher F1 scores (~0.67 – 0.68) than Pandas RF/GB (~0.66 – 0.67), which indicates a more balanced precision and recall in PySpark.

The difference suggests that PySpark's speed-optimized implementations may handle the dataset more efficiently, though both libraries are very close in performance. Also, this is not a fair comparison, as in phase 2, we have used 3 classes in classification models. But, in pyspark we have used just 2 classes.

References:

[1] Pirge, Gursev. “Text Cleaning: Standard Text Normalization with Spark NLP.” John Snow Labs, 7 June 2023, www.johnsnowlabs.com/text-cleaning-standard-text-normalization-with-spark-nlp/.

[2] Nelamali, N. (2024, May 17). PySpark fillna() & fill() – Replace NULL/None Values. Spark. https://sparkbyexamples.com/pyspark/pyspark-fillna-fill-replace-null-values/

[3] Malli. (2024, July 29). Pandas DataFrame shift() Function. Spark. https://sparkbyexamples.com/pandas/pandas-dataframe-shift-function/

“API Reference — PySpark 3.5.5 documentation.” https://spark.apache.org/docs/latest/api/python/reference/index.html

[4] “MLLiB: Main Guide - Spark 3.5.5 Documentation.” http://spark.apache.org/docs/latest/ml-guide.html

[5] Chaudhary, V. (2023, September 15). Understanding spark DAGs - Plumbers of Data Science - medium. *Medium*. https://medium.com/plumbersofdatascience/understanding-spark-dags-b82020503444

[5] Chaudhary, V. (2023, September 15). Understanding spark DAGs - Plumbers of Data Science - medium. *Medium*.