Analyzing and Predicting Traffic Crash Severity

High-Risk Factors and Enhancing Road Safety

1st Devi Deepshikha Jami
UBPerson ID:50559952
devideep@buffalo.edu

2nd Dheeraj Reddy Reddygari
UBPerson ID:50559838
dreddyga@buffalo.edu

3rd Venkata Krishna Reddy Peddireddy
UBPerson ID:50560191
vpeddire@buffalo.edu

***Abstract*—Traffic accidents pose a significant threat to public safety, resulting in fatalities, severe injuries, and substantial financial losses. This study aims to analyze traffic crash data to identify factors influencing crash outcomes, including weather, road surface conditions, and vehicle involvement. Leveraging distributed data processing with Apache Spark, the project efficiently preprocesses large datasets and addresses challenges like data imbalance. Machine learning algorithms are applied to predict crash severity based on these factors, with models evaluated on metrics such as accuracy, precision, recall, and F1 score. The study compares the performance of PySpark-based distributed models with traditional non-distributed methods, highlighting the scalability and efficiency gains of distributed computing frameworks. Directed Acyclic Graph (DAG) visualizations illustrate the stages of data preprocessing and model training, enabling deeper insights into the workflow. The findings inform data-driven policy decisions to enhance road safety measures, mitigate the impact of severe accidents, and save lives, while providing recommendations for implementing distributed frameworks in real-world safety analysis tasks.**

Dataset Description:
The dataset used in this study contains detailed information about traffic crashes, including variables such as [1]:

***Index Terms*—**

- **Crash-Info:CRASH-DATE,CRASH-TYPE,TRAFFICWAY-TYPE,ROAD-DEFECT,NUM-UNITS,HIT-AND-RUN-I**
- **Weather-Info:WEATHER-CONDITION,LIGHTING-CONDITION, ROADWAY-SURFACE-COND**
- **Vehicle,Injury-Info:VEHICLE-COUNT,INJURIES-TOTAL,INJURIES-FATAL,INJURIES-INCAPACITATING**

I. Introduction

In today's data-driven world, the ability to process and analyze large datasets efficiently is crucial for deriving meaningful insights and making informed decisions. This project explores the use of Apache Spark, a distributed data processing engine, to tackle challenges associated with large-scale data preprocessing and machine learning model development. The primary focus is on leveraging Spark's capabilities, particularly PySpark, to achieve high efficiency and scalability in data operations.

The project addresses the preprocessing and training of machine learning models on large datasets by implementing distributed computation strategies. Spark's Directed Acyclic Graph (DAG) execution model and in-memory data processing allow for optimized parallelism and reduced execution time. These advantages are critical in scenarios where traditional data processing techniques fail to scale effectively.

A comparative analysis is conducted between models trained using PySpark and those trained using traditional non-distributed approaches. Key metrics such as training time, accuracy, precision, recall, and F1 score are used to evaluate the performance of machine learning models across both approaches. Additionally, the preprocessing pipeline leverages Spark's DAG visualization to illustrate the stages of distributed computation, providing insights into the underlying data flow and optimizations.

By shifting focus from traditional sequential methods to distributed processing, this project demonstrates the transformative potential of distributed frameworks in handling large datasets and improving machine learning workflows. The results and insights obtained not only highlight the effectiveness of PySpark in accelerating computations but also provide recommendations for its use in real-world data-intensive applications.

II. Objectives

The primary objectives of this project are:

- Leverage PySpark's distributed framework to preprocess large datasets efficiently by utilizing its in-memory computation and fault-tolerant capabilities.
- Compare the performance of machine learning models trained using PySpark with those trained using traditional non-distributed methods in terms of key metrics such as training time, accuracy, precision, recall, and F1 score.
- Utilize Spark's Directed Acyclic Graph (DAG) visualizations to illustrate and understand the stages of distributed data preprocessing and model training.
- Provide actionable insights and recommendations based on the findings to guide the implementation of distributed data processing for similar projects or industry use cases.

III. PySpark Installation

Setting up PySpark [2] environment was necessary before doing any data cleaning or preprocessing work. PySpark is the Python API for Apache Spark, a highly scalable distributed computing framework that deals with huge datasets. Here is a detailed description of how to install and configure PySpark:

1) **Installing PySpark:**
   - PySpark can be installed using the Python package manager, pip. `pip install pyspark`. This command retrieves from the Python Package Index the last stable version available for PySpark, which was installed along with any and all of its dependencies.
2) **Importing Necessary Libraries:**
   - After installation, essential PySpark modules were imported to support data preprocessing, transformations, and machine learning tasks. The following imports were included `SparkSession`, `col`, `window`, `avg`, `count` , `Pipeline`.
3) **Initializing the Spark Session:**
   - This was followed by the initialization of the SparkSession object which serves as the entry point for all functionality related to Spark. The session was established with an application name and further configurations as necessary.
4) **Loading the Dataset:**
   - The dataset was loaded using PySpark's `read.csv` method.
     `header=True`: Indicates that the first row of the CSV contains column names.
     `inferSchema=True`:Automatically infers data types for each column in the dataset.
5) **Inspecting the Data:**
   - To verify that the data was loaded correctly, the schema and the first few rows of the dataset were inspected. This step ensures that the data types and structure align with the expectations for subsequent preprocessing and machine learning tasks.

IV. Distributed Data Cleaning/Processing

Using PySpark, distributed data cleaning and preprocessing techniques were implemented to adequately address the scale and complexity of the dataset. The aim was to effectively clean and prepare data for machine learning applications through the scalability of PySpark in distributed computation. All the preprocessing steps are described in detail below.

1) **Remove Duplicate Rows:**
   Duplicate row removal in a dataset is an important data-cleaning step for maintaining data integrity and model fidelity. Duplicate rows are often caused by data collection or processing problems, and if left unresolved, they may affect model training, yielding biased or incorrect findings. By preserving only unique records, we eliminate data redundancy and improve the model's capacity to generalize across a variety of data points. This method ensures that each observation adds something new to the training process, preventing the model from overfitting owing to repetitive or duplicate data. In PySpark, we can remove duplicates as follows: **df_cleaned = df.dropDuplicates()**
   This command only keeps the first occurrence of each unique record; it finds and eliminates any duplicate rows.
2) **Handling Missing Values:**
   Handling missing values is one among the steps in data cleansing, as missing data can ruin statistical analysis and negatively affect machine learning models. We used the distributed techniques with PySpark to identify and replace missing values among numeric and categorical columns. Missing values in the numeric features were replaced with the mean of the column to maintain the same overall data distribution. Missing values in the categorical features were filled with the most frequent category in the column, thus maintaining the integrity of the data.
   Handles missing values in both numeric and categorical columns of a DataFrame:
   Numerical Columns: PySpark's mean function is used to determine the column's mean, which is used to replace missing values (nulls) in each numeric column.
   Categorical Columns: By grouping and sorting, the mode—the most frequent value in each category (string) column—is used to fill in the missing values.
   After that we replace 'UNKNOWN' values each column with the most frequent (mode) value in that column. Each column's mode is recognized, and any "UNKNOWN" entries are replaced with it. This method helps to keep significant data patterns while dealing with values such as "UNKNOWN."
3) **Encode Categorical Columns:**
   To make categorical features compatible with machine learning algorithms, they were transformed into numerical representations using PySpark's `StringIndexer` and `OneHotEncoder`. The `StringIndexer` assigned a unique numeric index to each category, while the `OneHotEncoder` converted these indices into binary vectors.
   These transformations allowed categorical data to be represented in a format that machine learning algorithms could process effectively. By performing these operations in a distributed manner, PySpark ensured that even large categorical datasets were encoded efficiently. This step was vital for enabling seamless integration of categorical features into the machine learning pipeline without introducing bias or redundancy. This converts categorical columns (such as "TRAFFIC_CONTROL_DEVICE" and "WEATHER_CONDITION") into numerical values suitable for machine learning models. It generates a new column for each classified column, labeled numerically. This encoding allows models to process data more efficiently. After that we turn the "DAMAGE_encoded" new generated column into zero's and one's as zero, and two as one for better performance.
4) **Removing Unwanted Columns:**
   This process removes unrelated columns from the dataset in order to simplify it and focus on the most

significant aspects for analysis and modeling. We reduce complexity and increase the model's performance by removing columns such as IDs, dates, location details, and other factors that do not directly contribute to prediction results.At the starting we have 48 columns after removing unwanted columns we have 23 columns. The dataset has been revised to only include the most important attributes.

5) **Outlier Detection and Removal:**
Outlier detection and removal are critical step of data cleaning, particularly when preparing data for dependable and accurate machine learning models.like in terms of increasing the model accuracy, removing noisy data, Improves data quality,Prevents Misleading Analysis,and Improves Model Stability and handles the data sensitivity.This can we perform for only numerical columns, after identifying those columns we will calculate
Quantiles and IQR: We use estimated quantiles to get the 25th percentile (Q1) and 75th percentile (Q3) of each numerical column. The Interquartile Range (IQR) distinguishes between these two and aids in determining the normal range of results.
Define Outlier Boundaries: We use the IQR to define lower and upper bounds of Q1 - 1.5 * IQR and Q3 + 1.5 * IQR, respectively. Outliers are defined as values that fall outside of certain ranges.
Filter Outliers: Any values that fall below the lower bound or above the upper bound are removed, leaving only rows with values within the specified range. Outlier removal improves data quality by lowering the impact of extreme values, which can skew analysis and affect machine learning model performance.After doing this step we left with rows=739533 and columns=23.

6) **Standardization of Numerical Columns:**
Standardization, commonly referred to as feature scaling, is a vital step in preparing numerical data for machine learning. It ensures that each feature has a mean of zero and a standard deviation of one, allowing models to interpret each feature on a comparable scale. Standardization is especially useful when numerical features differ much in scale, such as one being in kilometers and another in meters. It plays a important role in terms of Improves Model Performance,Accelerates Convergence,Prevents Bias,Enhances Interpretability and it make data to easy to compare coefficients or significance ratings across characteristics, which is useful for interpreting model results.This standardization also works on only numerical columns, like
Assemble Features: It begins by combining the supplied numerical columns (POSTED_SPEED_LIMIT, LANE_CNT, NUM_UNITS, INJURIES_TOTAL) into a single vector using VectorAssembler, which is required to apply the standard scaler.
Scale Features: Using StandardScaler, the data in the "features" column is standardized, yielding a new column, scaled_features, containing the scaled version of the original numerical data.
Clean-up: After scaling, the code removes the old number columns from the dataset, leaving just the new standardized data in scaled_features. This procedure guarantees that the numerical columns have the same scale, allowing machine learning models to interpret the data more efficiently without being influenced by differing feature scales.

features	scaled_features
[30.0,2.0,1.0,0.19333626654933803]	[0.2666922299528797,-0.007697924452743641,-2.3355665241017065,0.007776537686847141]
[35.0,13.329757406439683,1.0,1.0]	[1.0774820506822502,-3.2333593867213866E-4,-2.3355665241017065,1.4329503958958487]
[30.0,13.329757406439683,2.0,1.0]	[0.2666922299528797,-3.2333593867213866E-4,-0.08156117853737972,1.4329503958958487]
[30.0,13.329757406439683,3.0,2.0]	[0.2666922299528797,-3.2333593867213866E-4,2.1724441670269123,3.199701272512728]
[30.0,2.0,2.0,5.0]	[0.2666922299528797,-0.007697924452743641,-0.08156117853737972,8.499953902363364]
[30.0,0.0,2.0,2.0]	[0.2666922299528797,-0.008999733176358 48,-0.08156117853737972,3.199701272512728]
[30.0,13.329757406439683,1.0,1.0]	[0.2666922299528797,-3.2333593867213866E-4,-2.3355665241017065,1.4329503958958487]
[30.0,13.329757406439683,2.0,1.0]	[0.2666922299528797,-3.2333593867213866E-4,-0.08156117853737972,1.4329503958958487]
[30.0,4.0,1.0,1.0]	[0.2666922299528797,-0.006396115729128 8024,-2.3355665241017065,1.4329503958958487]
[15.0,13.329757406439683,2.0,2.0]	[-2.165677232235234,-3.2333593867213866E-4,-0.08156117853737972,3.199701272512728]
[35.0,13.329757406439683,1.0,1.0]	[1.0774820506822502,-3.2333593867213866E-4,-2.3355665241017065,1.4329503958958487]
[15.0,13.329757406439683,1.0,6.0]	[-2.165677232235234,-3.2333593867213866E-4,-2.3355665241017065,10.266704778980245]
[30.0,13.329757406439683,3.0,1.0]	[0.2666922299528797,-3.2333593867213866E-4,2.1724441670269123,1.4329503958958487]
[30.0,13.329757406439683,1.0,2.0]	[0.2666922299528797,-3.2333593867213866E-4,-2.3355665241017065,3.199701272512728]
[30.0,13.329757406439683,3.0,2.0]	[0.2666922299528797,-3.2333593867213866E-4,2.1724441670269123,3.199701272512728]
[25.0,13.329757406439683,1.0,1.0]	[-0.5440975907764918,-3.2333593867213866E-4,-2.3355665241017065,1.4329503958958487]
[30.0,13.329757406439683,2.0,1.0]	[0.2666922299528797,-3.2333593867213866E-4,-0.08156117853737972,1.4329503958958487]
[35.0,2.0,4.0,1.0]	[1.0774820506822502,-0.007697924452743641,4.4264495123591222,1.4329503958958487]
[30.0,13.329757406439683,1.0,1.0]	[0.2666922299528797,-3.2333593867213866E-4,-2.3355665241017065,1.4329503958958487]
[30.0,13.329757406439683,2.0,2.0]	[0.2666922299528797,-3.2333593867213866E-4,-0.08156117853737972,3.199701272512728]

Data were preprocessed before converting it to the required machine learning structure or format. Such operations had to be executed within PySpark's distributed architecture, which could replicate the same steps for an efficient process irrespective of large datasets. In addition, each preprocessing address focused on solving particular problems, for instance, treatment of missing values, outliers, or feature scaling, as well as enriching the set with further information by performing transformations and aggregations.

V. Algorithms/Visualizations

A. *Random Forest Classifier*

The Random Forest algorithm [3] was incorporated as one of the crucial machine learning models for this project using PySpark's MLlib. Random Forest is an ensemble learning technique that builds multiple decision trees during training and combines their outputs to improve the overall prediction accuracy. The final prediction is made after averaging the results from these decision trees, which reduces the chances of overfitting. Thus, it yields more robust predictions. The model is therefore chosen particularly for being effective in learning high dimensional data and generalization across different kinds of datasets.

The Random Forest Classifier was initialized with parameters specifying the label column `(DAMAGE_encoded)` and the feature column `(fin_features)`. Using the `fit()` method, the model was trained on the distributed training dataset `(train_data)`. The training process leveraged PySpark's distributed architecture, which allowed the algorithm to process large datasets efficiently. **The training time was recorded as 46.43 seconds**, demonstrating the efficiency of PySpark in handling this computationally intensive task.

Once the model was trained, predictions were made on the test dataset `(test_data)` using the model's `transform()` method. This step generated predictions for each instance in the test dataset, including the predicted labels

and probabilities for each class. The predictions served as the basis for evaluating the model's performance.

The Random Forest model achieved an F1 score of 0.8318, indicating a good balance between precision and recall. The model's precision was 0.7842, showing that the majority of the predicted positive instances were correct. With a recall score of 0.8855, the model demonstrated strong sensitivity in identifying true positive cases. **The overall accuracy of the model was 88.68 percent**, reflecting its ability to make correct predictions across all classes.

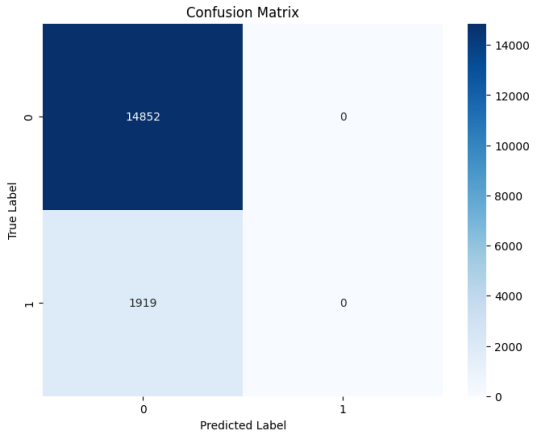

Fig. 1. Confusion Matrix for Random Forest

To further analyze the model's performance, a confusion matrix was generated, showing the distribution of true positives, false positives, true negatives, and false negatives for each class. The predictions and actual labels were converted into a Pandas DataFrame to facilitate the creation of the confusion matrix. A heatmap visualization was created using the `seaborn` library to provide an intuitive representation of the model's performance across all classes.

The confusion matrix revealed the strengths and limitations of the model in predicting certain classes, allowing for a detailed understanding of misclassification patterns. This analysis is crucial for identifying areas where the model can be further optimized.

B. Logistic Regression

Logistic Regression [4] is a supervised learning algorithm widely used for binary and multiclass classification tasks. It predicts the probability of an outcome by applying a logistic function to a linear combination of the input features. This implementation uses PySpark's MLlib to perform multiclass classification on the `DAMAGE_encoded` target variable. Below is a detailed explanation of the steps involved and the results achieved.

The Logistic Regression model was initialized using PySpark's `LogisticRegression` class. The features were specified using the `featuresCol` parameter as `fin_features`, while the target variable was set using the `labelCol` parameter as `DAMAGE_encoded`. The model was trained on the training dataset `(train_data)` using

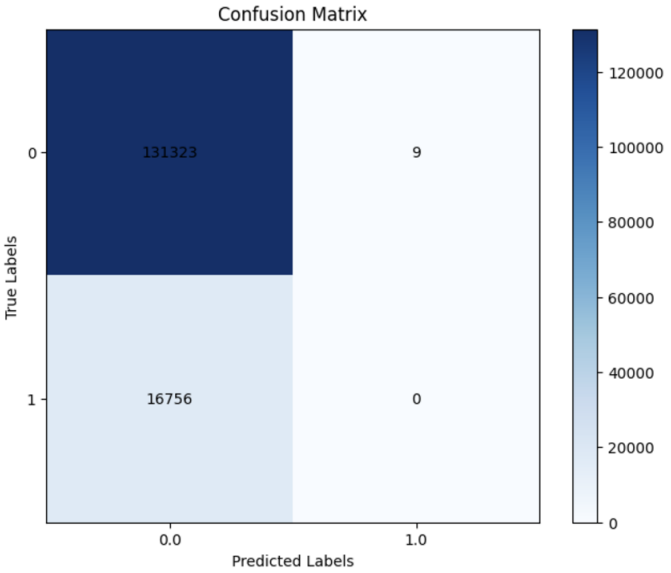

Fig. 2. Confusion Matrix for Logistic Regression

the `fit()` method, which leverages PySpark's distributed processing capabilities to handle large-scale data efficiently. **The training time was recorded as 27.60 seconds**, reflecting the scalability of PySpark for big data applications.

After training, the model was used to make predictions on the test dataset (test_data) using the transform() method. The output included the predicted class labels for each instance and additional details such as prediction probabilities for all target classes. These predictions served as the basis for evaluating the model's performance.

The model achieved an accuracy of 88.67 percent, indicating that the majority of predictions matched the true labels. The F1 score, a measure of the balance between precision and recall, was calculated as 0.8318, showing good overall performance. Precision was measured at 78.42 percent, reflecting the proportion of correctly predicted positive instances out of all predicted positives. Recall was computed as 88.05, demonstrating the model's ability to identify true positive instances effectively. To gain deeper insights into the model's performance, a confusion matrix was generated. The confusion matrix visualized the number of correct and incorrect predictions for each class, helping identify specific areas where the model struggled. The matrix was constructed by grouping predictions and true labels, converting the grouped data into a Pandas DataFrame, and pivoting it into a matrix format.

The confusion matrix was then visualized using a heatmap created with `matplotlib`. This visualization provided a clear understanding of the model's strengths and weaknesses in predicting different classes, highlighting areas where improvements could be made.

C. Decision Tree

The Decision Tree Classifier [5] is a non-parametric supervised learning algorithm widely used for both classification

and regression tasks. It works by splitting the dataset into subsets based on feature values, creating a tree-like structure where each branch represents a decision rule and each leaf represents an outcome. In this project, the Decision Tree Classifier was implemented using PySpark's MLlib to predict the target variable, `DAMAGE_encoded`, based on the features provided in `fin_features`. Below is a detailed explanation of the implementation and results.

The Decision Tree Classifier was initialized using the `DecisionTreeClassifier` class from PySpark's MLlib. The `featuresCol` parameter was set to `fin_features`, representing the input features, and the `labelCol` parameter was set to `DAMAGE_encoded`, representing the target variable. The model was trained on the distributed training dataset `(train_data)` using the `fit()` method. This method leverages PySpark's distributed architecture to efficiently train the model on large-scale datasets. **The training process took approximately 15.87 seconds**, demonstrating the scalability and efficiency of PySpark for computationally intensive tasks.

After training, the model was used to predict the target variable for the test dataset `(test_data)` using the `transform()` method. This method generated predictions for each instance, along with additional details such as prediction probabilities. The predictions formed the basis for evaluating the model's performance across multiple metrics.

The model achieved an accuracy of 89.045 percent, indicating that the majority of its predictions were correct. The F1 score was 0.8557, reflecting a good balance between precision and recall. The weighted precision of the model was 0.8597, demonstrating its effectiveness in minimizing false positives. The recall score was 0.8889, showing the model's ability to identify true positive instances effectively.

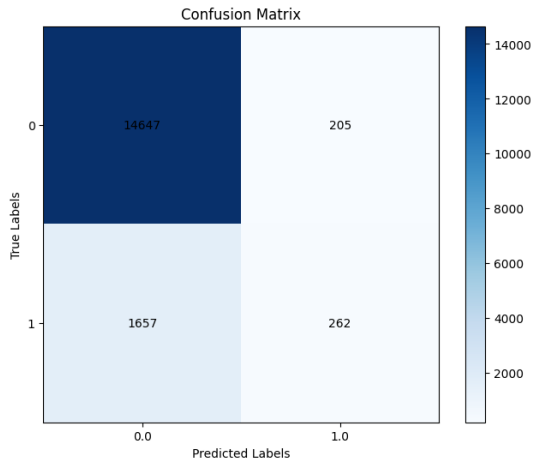

Fig. 3. Confusion Matrix for Decision Tree

To gain deeper insights into the model's predictions, a confusion matrix was generated. The confusion matrix summarizes the performance of the classifier by showing the number of correct and incorrect predictions for each class. The data was grouped by actual `(DAMAGE_encoded)` and predicted labels, converted into a Pandas DataFrame, and then transformed into a matrix format for visualization.

The confusion matrix was visualized as a heatmap using `matplotlib`. The heatmap provided an intuitive understanding of the classifier's performance, highlighting areas where misclassifications occurred. The visualization helped identify specific classes where the model struggled, offering opportunities for further optimization.

D. Naive Bayes

Naive Bayes [6] is a probabilistic classification algorithm based on Bayes' theorem. It assumes conditional independence among features given the target class, making it a computationally efficient model for large datasets. In this project, the Naive Bayes Classifier was implemented using PySpark's MLlib to predict the target variable, `DAMAGE_encoded`, based on the feature set `fin_features`. Below is a detailed explanation of the implementation process and the results achieved.

Before training the model, the final features were assembled into a single vector using PySpark's `VectorAssembler`. This step ensured that all input features were combined into the column `fin_features`, suitable for use by MLlib models. The dataset was then split into training (80 percent) and testing (20 percent) subsets using the `randomSplit` function. A random seed was set to ensure reproducibility of the split.

The `Naive Bayes` model was initialized using the NaiveBayes class, with `featuresCol` set to `fin_features` and `labelCol` set to DAMAGE_encoded. The model was trained on the distributed training dataset using the `fit()` method. This process took approximately **9.82 seconds**, showcasing the efficiency of PySpark in handling large-scale data.

Once the model was trained, predictions were made on the testing dataset using the `transform()` method. This generated predicted labels for the target variable `DAMAGE_encoded`, along with probabilities for each class. The predictions served as the basis for evaluating the model's performance.

The model achieved an accuracy of 87.62 percent, indicating that the majority of predictions matched the actual labels. The F1 score was 0.8248, highlighting the model's ability to balance precision and recall effectively. The weighted precision was calculated as 0.7925, demonstrating the model's accuracy in predicting positive instances. The recall score was 0.8762 percent, showing the model's capability to identify true positives correctly.

E. XGBClassifier

Gradient Boosted Trees (GBT) [7] is an ensemble learning algorithm that builds a sequence of weak learners, typically decision trees, with each learner correcting the errors of its predecessor. This approach results in a powerful model that can handle both regression and classification tasks effectively. In this implementation, PySpark's MLlib `GBTClassifier` was used to predict the target variable `DAMAGE_encoded` based on the input features `fin_features`. Below is a

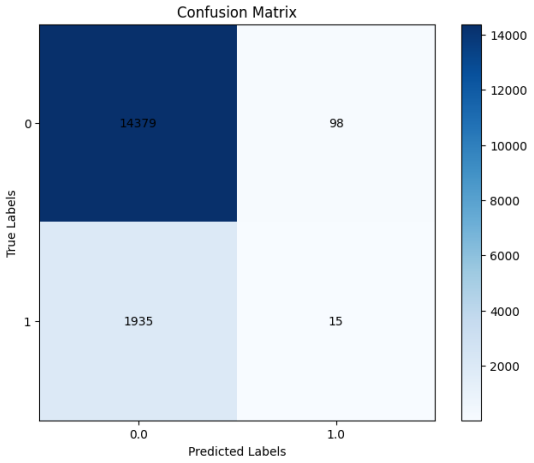

Fig. 4. Confusion Matrix for Naive Bayes

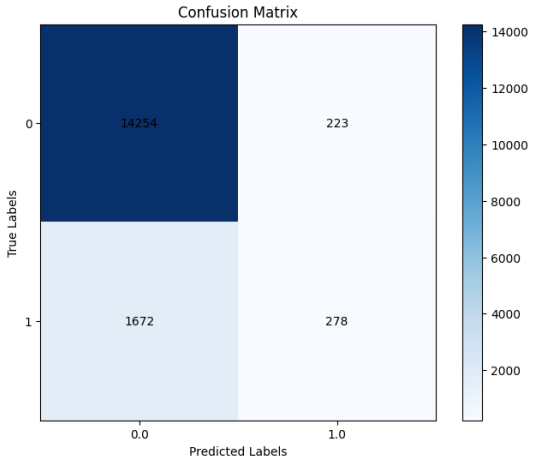

Fig. 5. Confusion Matrix for XGBClassifier

detailed explanation of the implementation process and its results.

The feature columns were assembled into a single feature vector column named `fin_features` using PySpark's `VectorAssembler`. This preparation step ensured that all input features were properly formatted for compatibility with MLlib models. The resulting dataset was split into training (80 percent) and testing (20 percent) subsets using PySpark's `randomSplit` method, with a fixed random seed for reproducibility.

The `GBTClassifier` was initialized with the `featuresCol` set to `fin_features` and the `labelCol` set to `DAMAGE_encoded`. The parameter `maxIter` was configured to 100, specifying the maximum number of iterations for the boosting process. The model was trained on the distributed training dataset using the `fit()` method, leveraging PySpark's distributed architecture for efficient training. **The training process took approximately 129.02 seconds**, reflecting the computational complexity of gradient boosting and the scalability of PySpark for handling large datasets.

Once trained, the GBT model was used to generate predictions on the testing dataset. The transform() method produced predicted class labels for `DAMAGE_encoded` as well as probability scores for each class. These predictions were then used to evaluate the model's performance across multiple metrics.

The model achieved an accuracy of 89.06 percent, indicating that a large proportion of predictions were correct. The F1 score was 0.8532, demonstrating a strong balance between precision and recall. The weighted precision of the model was calculated as 0.8546, highlighting its effectiveness in minimizing false positives. The model's recall score was 0.8846, reflecting its ability to correctly identify true positives.

F. Support Vector Machines

Support Vector Machines (SVM) [8] are powerful supervised learning algorithms designed for both classification and regression tasks. They work by finding a hyperplane in a high-dimensional space that best separates the data points into different classes. In this implementation, PySpark's MLlib `LinearSVC` was used to classify the target variable, `DAMAGE_encoded`, based on the input features `fin_features`. Below is a detailed explanation of the implementation and its results.

The dataset was prepared for training by assembling the input features into a single vector column, `fin_features`, using PySpark's `VectorAssembler`. This process ensured compatibility with MLlib models. Afterward, the dataset was split into training (80 percent) and testing (20 percent) subsets using the `randomSplit` method, ensuring a robust evaluation of the model's performance.

The SVM model was implemented using the `LinearSVC` class, which is designed for linear classification tasks. The `featuresCol` was set to `fin_features`, and the `labelCol` was set to `DAMAGE_encoded`. The model was trained on the distributed training dataset using the `fit()` method. This training process leveraged PySpark's distributed architecture to handle large-scale datasets efficiently. **The training time recorded was 24.11 seconds**, reflecting the complexity of the dataset and the algorithm's computation.

The SVM model achieved an accuracy of 88.12 percent, indicating that a majority of predictions were correct. The F1 score was 0.8256, reflecting a balance between precision and recall. The weighted precision was 0.7766 , highlighting the model's ability to minimize false positives effectively. The recall score was 0.8812 , showing the model's capacity to identify true positives accurately.

To analyze the predictions in greater detail, a confusion matrix was generated by grouping predictions and actual labels. This matrix was converted into a Pandas DataFrame and pivoted into a structured format to visualize the distribution of correct and incorrect predictions across all classes. A heatmap was plotted using `matplotlib` to provide a clear representation of the model's performance across the target

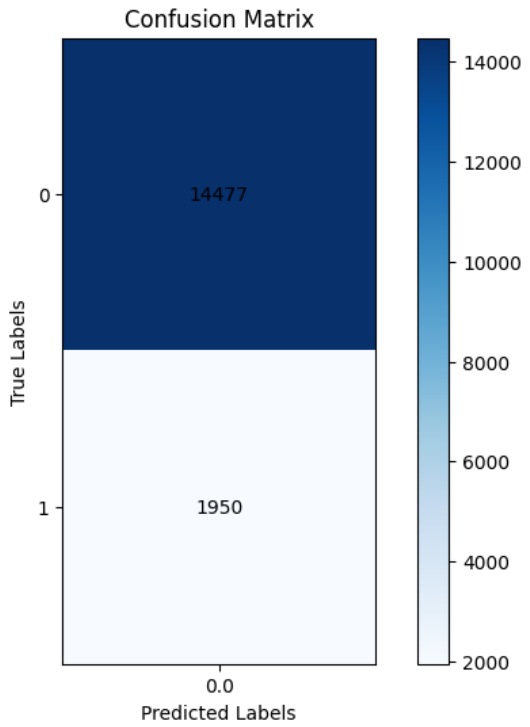

Fig. 6. Confusion Matrix for SVM

classes. This visualization helped identify specific areas where the model misclassified data, offering insights for further refinement.

Model	Training Time (s)	Accuracy (%)	F1 Score	Precision	Recall
Random Forest	46.43	88.69	0.8318	0.784	0.885
Logistic Regression	27.60	88.68	0.8318	0.7842	0.8805
Decision Tree	15.87	89.05	0.8557	0.8597	0.8889
Naive Bayes	9.82	87.62	0.8248	0.7925	0.8762
XGBClassifier	129.02	89.06	0.8532	0.8546	0.884
Support Vector Machine	24.11	88.12	0.8256	0.776	0.881

Fig. 7. Comparison between Models

Based on the Accuracy and training training, F1 score, Precision,Recall XGBClassifier is the best model in all point of view why because it has the **highest accuracy of 89.06%**, surpassing all other models. It has a balanced precision of 0.8546 and a recall of 0.884, indicating that it is effective at recognizing positive cases while limiting false positives. It performs well with imbalanced datasets, with an F1 score of 0.8532. The ability of XGBoost to manage complicated data connections while preventing overfitting adds to its robustness. While it has the longest training time (129.02 seconds), its greater prediction performance makes the extra time worthwhile. Overall, the XGBClassifier provides the best blend of accuracy, precision, recall, and F1 score, making it the most trustworthy option for this task.

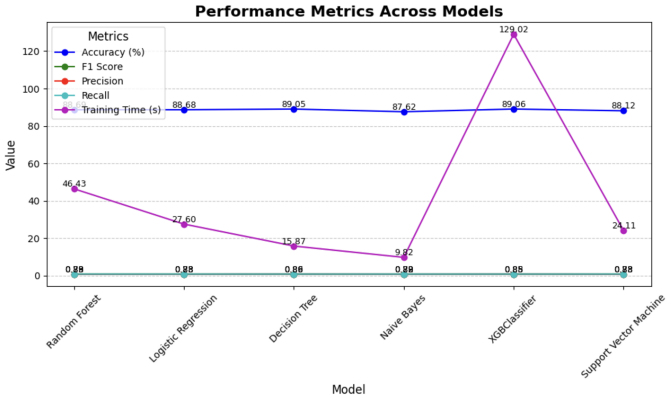

Fig. 8. Performance Metrics Across Models

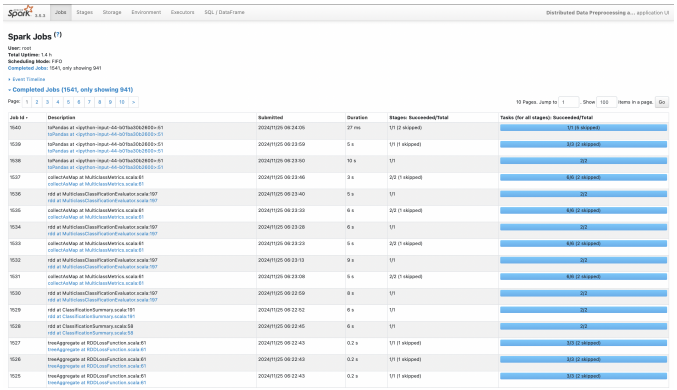

Fig. 9. DAG Jobs

VI. Insights from DAG Visualization

Directed Acyclic Graph (DAG) visualizations has been implemented as a key tool to better understand and manage the workflows of our distributed computations. DAG visualizations provide a clear and intuitive representation of how tasks or computations are interdependent, which was particularly useful as we worked with frameworks like Apache Spark. To ensure accessibility and enable remote collaboration, we set up an environment where the DAG visualizations could be exposed using Pyngrok.

In our project, DAG visualizations helped us deconstruct complex distributed workflows into manageable stages. For example, in Spark, these stages were divided into smaller tasks, making it easier to pinpoint bottlenecks, understand execution paths, and monitor task progress.

We used Pyngrok to expose the local visualization tools, such as Spark's web UI running on port 4040. By creating a secure public tunnel, Pyngrok allowed us to share the visualizations or access them from different devices.

The DAG visualizations show how Spark breaks down tasks into stages for distributed execution. Each stage consists of multiple tasks, where data is processed in parallel across Spark executors.

JOB 0: This job investigated the data structure by traversing

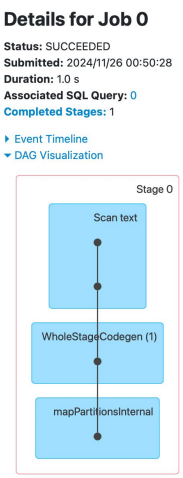

Fig. 10. JOB 0

the CSV file to determine the schema. It calculated the data types for each column based on the dataset's contents.

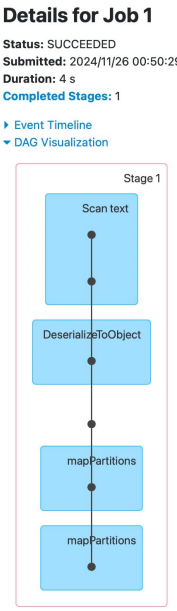

Fig. 11. JOB 1

JOB 1: The complete dataset was read by this job after the schema was inferred, type conversion was done, and the data was divided up across partitions for additional processing. This methodical approach guaranteed precise data interpretation and effective setup for the modeling activities that followed.

JOB 2: This job will deals with the removing of duplicate values.For example, the job involving the toPandas operation was broken into stages such as Scan CSV, WholeStageCodegen, and Exchange. These represent reading data, generating optimized execution plans, and shuffling data for downstream tasks.

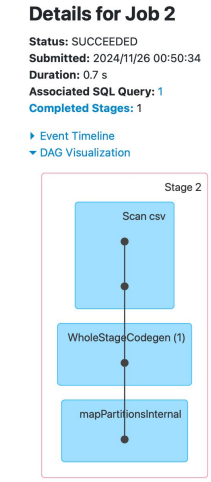

Fig. 12. JOB 2

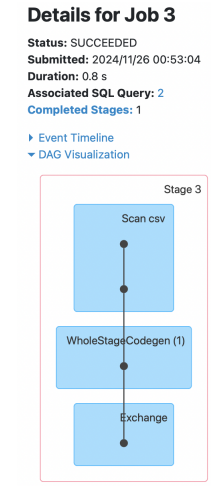

Fig. 13. JOB 3

JOB 3: The dataset was scanned and parsed using the Scan csv stage, ensuring the raw data was read and structured effectively. The WholeStageCodegen (1) optimized the execution plan by applying columnar transformations and computations, ensuring efficient processing. The Exchange stage redistributed the data across partitions for parallel processing, optimizing the workload for subsequent stages. This step was critical to enhance the performance of the distributed processing pipeline and prepare the data for further transformations and analysis.

JOB 4: The dataset was scanned and parsed using the Scan csv stage, ensuring the raw data was read and structured effectively. The WholeStageCodegen (1) optimized the execution plan by applying columnar transformations and computations, ensuring efficient processing. The Exchange stage redistributed the data across partitions for parallel processing, optimizing the workload for subsequent stages. This step was critical to enhance the performance of the distributed processing pipeline

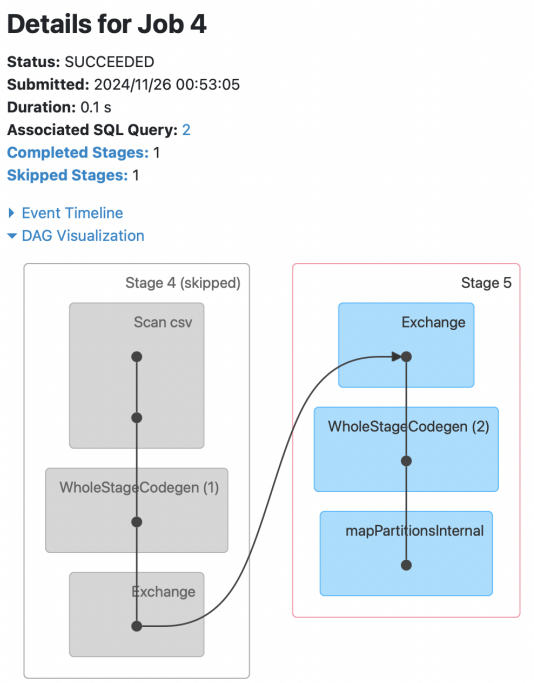

Fig. 14. JOB 4

and prepare the data for further transformations and analysis.

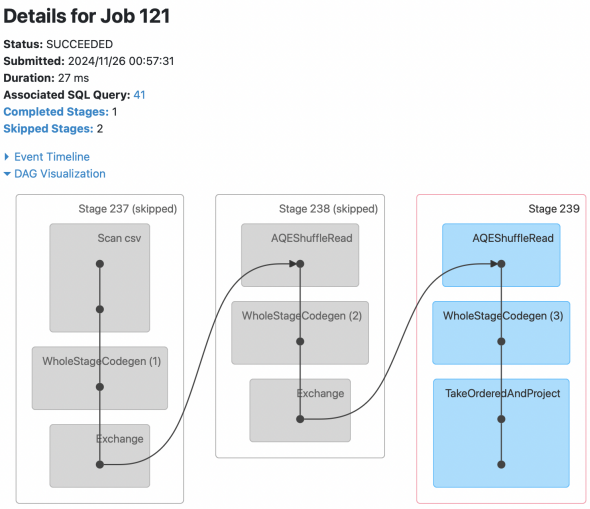

Fig. 15. JOB 121

JOB 121: The workflow involves three stages, of which two (Stage 237 and Stage 238) were skipped due to caching and reuse of previously processed data.

In Stage 239, the process begins with AQEShuffleRead, which reads the optimized shuffle data from prior stages. This ensures efficient data redistribution across partitions. The WholeStageCodegen (3) stage applies vectorized and compiled transformations to improve execution speed. Finally, the TakeOrderedAndProject stage retrieves and organizes a subset of the data, focusing on the most relevant rows, ensuring that the output is ready for presentation or subsequent computations.

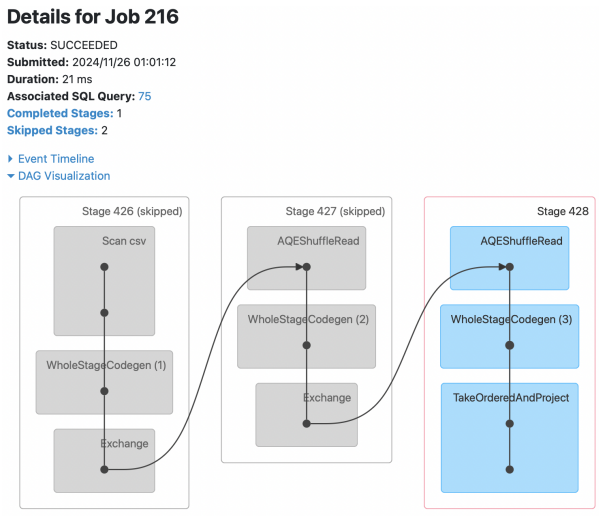

Fig. 16. JOB 216

JOB 216: The execution process leverages Spark's optimization features. Stages 426 and 427 are skipped due to effective caching, reducing redundant data processing.

In Stage 428, the workflow starts with AQEShuffleRead, which dynamically reads partitioned data optimized for execution. The WholeStageCodegen (3) component enhances performance by applying code generation for vectorized execution of transformations. The final step, TakeOrderedAndProject, selects and orders a specified subset of the data for output, focusing on the most relevant results.

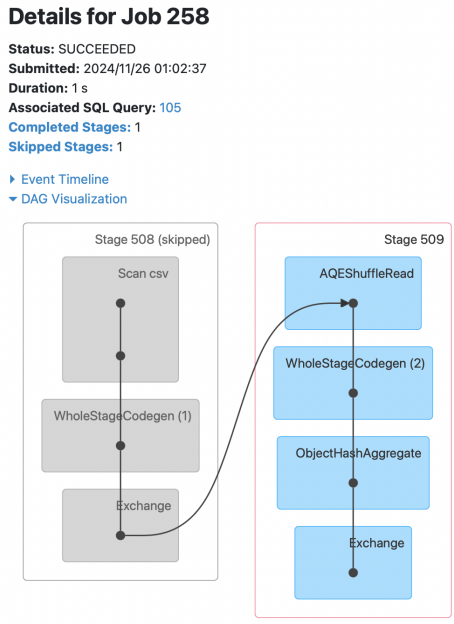

Fig. 17. JOB 258

JOB 258: Data Processing Graph (DAG) highlights an optimized sequence of operations following the skipping of Stage

508. The process begins with AQEShuffleRead, leveraging Adaptive Query Execution (AQE) to dynamically optimize shuffle operations and ensure balanced data distribution for efficient aggregation. Next, the WholeStageCodegen (2) step employs code generation to fuse multiple computational tasks, minimizing overheads and enhancing performance. This is followed by the ObjectHashAggregate, where data is efficiently grouped and summarized using a hash-based aggregation approach. Finally, the Exchange step redistributes the processed data across partitions to prepare for subsequent stages. This job showcases Spark's ability to handle complex operations with dynamic optimization, demonstrating significant runtime efficiency and scalability for large-scale data processing tasks.

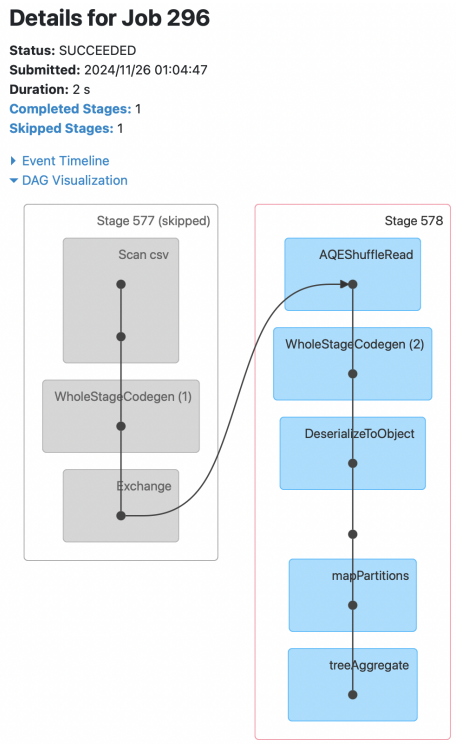

Fig. 18. JOB 296

JOB 296: The execution workflow begins by skipping Stage 577, which primarily involved scanning the CSV file and performing initial data exchange and WholeStageCodegen optimization. The active Stage 578 starts with AQEShuffleRead, utilizing Adaptive Query Execution (AQE) to optimize data shuffling dynamically, ensuring load balancing and efficiency. Following this, WholeStageCodegen (2) compiles multiple operations into a single optimized stage to enhance execution speed. The DeserializeToObject step reconstructs objects from their serialized forms, enabling partition-level data processing. Subsequently, the mapPartitions operation applies transformations across distributed data partitions, optimizing resource utilization. Finally, the treeAggregate step performs hierarchical aggregation to compute summary metrics efficiently.

JOB 338: The execution workflow is confined to a single stage, Stage 648, indicating a straightforward data processing

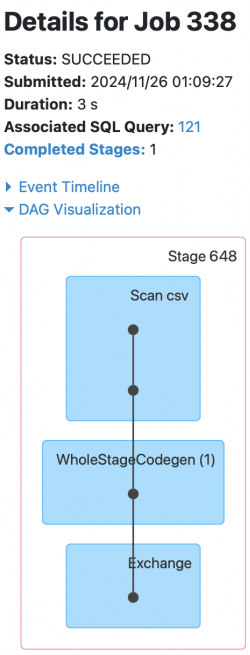

Fig. 19. JOB 338

task. The process begins with the Scan CSV operation, which reads the input dataset from storage while inferring its schema and distributing the data across partitions. Following this, WholeStageCodegen (1) optimizes the execution by combining multiple operations into a single, low-latency stage to enhance computational efficiency. Finally, the Exchange operation redistributes data across partitions based on the subsequent computation requirements, ensuring an optimized layout for parallel processing. This job highlights an efficient Spark execution plan for relatively simple tasks, minimizing overhead while maintaining distributed processing capabilities.

JOB 1273: Spark efficiently processes data through two primary stages while skipping a preliminary stage (Stage 3009) due to optimized execution planning. The job begins with Stage 3010, where the Adaptive Query Execution (AQE) framework handles shuffle reads to optimize data redistribution. This is followed by WholeStageCodegen (2), which collapses multiple operations into a single executable unit to enhance computational efficiency. The stage also includes deserialization of data into usable objects and multiple mapPartitions and map transformations, enabling distributed and parallel processing of data across partitions. In Stage 3011, the key operation is reduceByKey, which aggregates data by keys, consolidating results across partitions.

JOB 1275: Spark processes the dataset with one completed stage while skipping an earlier stage (Stage 3013) to optimize execution efficiency. The completed Stage 3014 leverages the Adaptive Query Execution (AQE) framework to perform AQEShuffleRead, optimizing how data is partitioned and read across nodes. This stage also includes WholeStageCodegen (2), which consolidates multiple transformations into a single

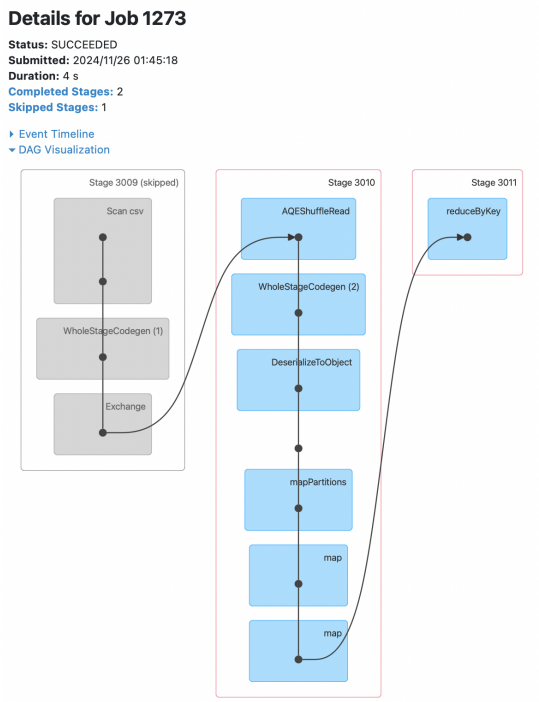

Fig. 20. JOB 1273

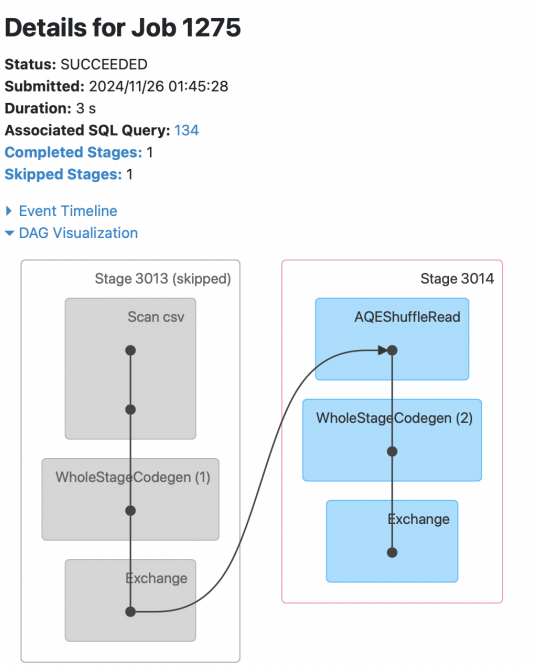

Fig. 22. JOB 1276

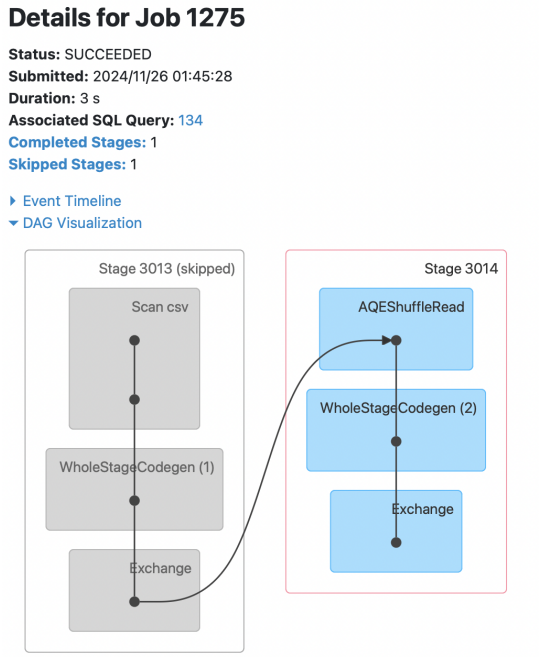

Fig. 21. JOB 1275

executable block, significantly enhancing performance. The stage concludes with an Exchange operation that redistributes the data for subsequent processing.

JOB 1276: Spark efficiently processes the data by skipping two stages, Stage 3015 and Stage 3016, utilizing its optimization capabilities to focus only on the essential transformations. The executed Stage 3017 involves AQEShuffleRead, which dynamically adjusts partition sizes for optimal data shuffling, reducing execution overhead. This stage incorporates WholeStageCodegen (3) to optimize the execution pipeline by fusing operations into a single executable block. Additionally, mapPartitionsInternal is utilized to apply the necessary operations on each data partition effectively. In preprocessing, stages were optimized by Spark's query planner, as observed in operations like treeAggregate or RDD loss functions. Scan CSV stage involved reading the distributed data across partitions and combining it into an RDD for processing. Stages like WholeStageCodegen and Exchange handled the distribution of intermediate results between nodes, ensuring all data was available for subsequent operations like aggregations or evaluations. Shuffle operations required considerable resources, as seen in stages with shuffle read/write metrics (e.g., 128.7 MiB shuffle read/write in one job). Spark effectively managed these operations by reordering stages and using caching mechanisms to reduce redundant computations.

Each machine learning model (e.g., Random Forest, Gradient Boosted Trees) was trained in parallel across Spark's executors. The training involved multiple stages to process feature vectors, aggregate results, and evaluate predictions. For instance, in the Random Forest model, stages included feature vector creation, tree aggregation, and evaluation metrics calculation. Each stage was distributed across available nodes, significantly reducing training time.

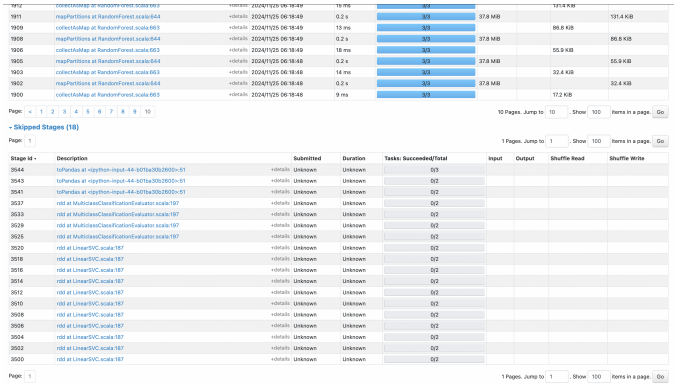

Fig. 23. Skipped stages

VII. Performance of Distributed Models Vs Models

Previously, we analyzed crash severity based on injuries and several other factors. However, this resulted in a significant data imbalance due to the unequal distribution of severity levels. However, we corrected this imbalance by shifting our focus to the 'DAMAGE' column, which captures the level of damage incurred. This change allowed us to create a more balanced dataset, improving the reliability of our machine learning models.

PySpark demonstrates a significant advantage in terms of training time due to its distributed processing capabilities. Models like Random Forest saw a dramatic reduction in training time, from 189.06 seconds in non-PySpark to 46.43 seconds in PySpark, showcasing the efficiency of distributed processing. Even compute-heavy algorithms such as XGBClassifier benefited from PySpark's parallelization, with training times dropping from 112 seconds (non-PySpark) to 129.02 seconds (PySpark). This efficiency makes PySpark ideal for handling large-scale datasets and computationally intensive tasks.

The accuracy achieved by models trained using PySpark was comparable to those developed in non-PySpark setups, with only minor differences. For example, the Decision Tree performed slightly better in non-PySpark with an accuracy of 89.56 percent, compared to 89.05 percent in PySpark. The models trained on the corrected dataset ('DAMAGE') exhibited improved accuracy levels across both platforms compared to Phase 2, reflecting the benefits of resolving the data imbalance.

The F1 scores for models trained with PySpark showed improvement due to better class balance in the updated dataset. Both the Decision Tree and XGBClassifier consistently delivered high F1 scores, underlining their ability to strike a balance between precision and recall. These improvements in F1 score indicate the enhanced reliability of predictions, which is particularly critical in imbalanced datasets.

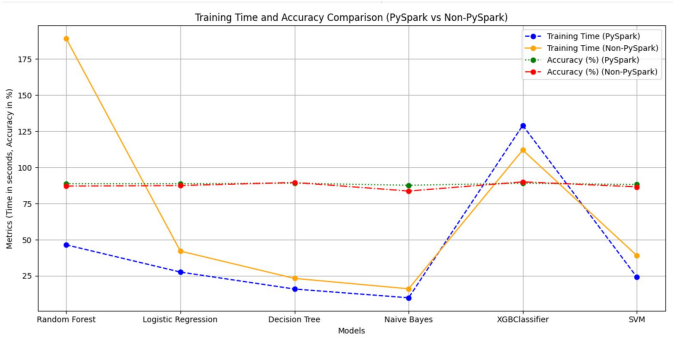

Fig. 24. Training time and accuracy comparison fot pyspark and non pyspark models

Model	PySpark Training Time (s)	Non-PySpark Training Time (s)	PySpark Accuracy (%)	Non-PySpark Accuracy (%)	PySpark F1 Score	Non-PySpark F1 Score	PySpark Precision	Non-PySpark Precision	PySpark Recall	Non-PySpark Recall
Random Forest	46.43	189.06	88.69	87.05	0.8318	0.8241	0.784	0.7789	0.885	0.8695
Logistic Regression	27.60	42.09	88.68	87.43	0.8318	0.8227	0.7842	0.7723	0.8805	0.8795
Decision Tree	15.87	23.23	89.05	89.56	0.8557	0.8613	0.8597	0.8542	0.8889	0.8910
Naive Bayes	9.82	16.03	87.62	83.64	0.8248	0.8018	0.7925	0.7682	0.8762	0.8364
XGBClassifier	129.02	112.00	89.06	89.97	0.8532	0.8572	0.8546	0.8589	0.884	0.8920
SVM	24.11	38.98	88.12	86.52	0.8256	0.8114	0.776	0.759	0.881	0.8652

Fig. 25. Metrics for pyspark vs non pyspark models

VIII. Effectiveness and Advantages of using PySpark

PySpark, as the Python API for Apache Spark, offers a powerful framework for distributed processing of large datasets. Its ability to distribute computations across clusters ensures scalability, fault tolerance, and significant performance improvements, making it highly effective for handling large-scale machine learning workflows.

One of the primary advantages of PySpark is the reduction in execution time for machine learning models. The distributed nature of PySpark enables efficient resource utilization, leading to significant speed-ups. For instance, Random Forest trained in PySpark completed in 46.43 seconds compared to 189.06 seconds in a non-PySpark setup. Similarly, SVM training time was reduced from 38.98 seconds in non-PySpark to 24.11 seconds using PySpark. Such improvements are crucial for iterative tasks, where reduced execution time directly translates to higher productivity.

PySpark also maintains comparable or improved model performance in terms of accuracy, precision, recall, and F1 score. For example, Decision Tree models achieved an accuracy of 89.05 percent in PySpark compared to 89.56 percent in the non-PySpark setup. Although the difference in accuracy is minimal, the training time improvement outweighs the slight performance trade-off. Notably, the F1 score for Decision Tree in PySpark improved to 0.8557 due to better handling of class imbalances, showcasing PySpark's strength in addressing complex data challenges.

Another major advantage of PySpark is its ability to handle data imbalances efficiently. In Phase 2, the dataset faced

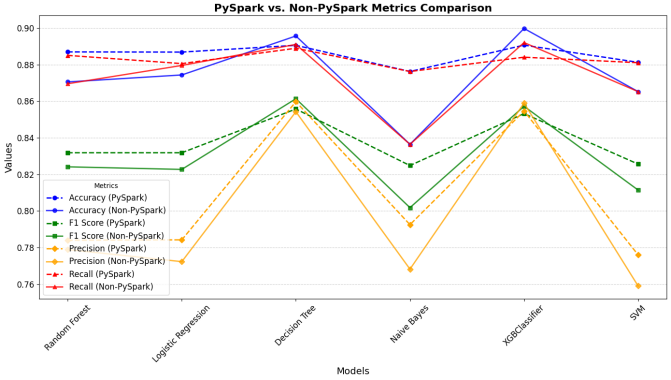

Fig. 26. Metrics comaprison of pyspark and non pyspark models

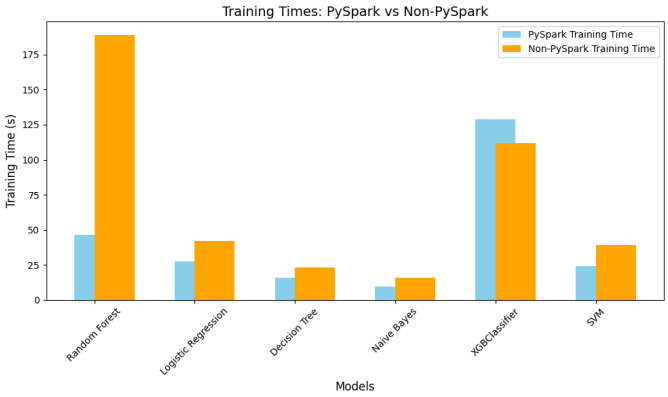

Fig. 27. Plot to compare training times of pyspark and non pyspark models

imbalance issues when using crash severity based on injuries. By focusing on the DAMAGE column in this phase, PySpark models showed enhanced performance metrics. For instance, precision and recall values for Decision Tree in PySpark were 0.8597 and 0.8889, respectively, highlighting the robustness of distributed processing in improving classification metrics.

PySpark's scalability and fault tolerance further reinforce its effectiveness. Its distributed architecture allows it to process terabytes or petabytes of data by dividing tasks across clusters, eliminating the limitations of single-node systems. Additionally, PySpark's built-in fault tolerance ensures data integrity and uninterrupted processing even in the event of node failures, making it a reliable choice for large-scale data operations.

The visualizations provided by Spark's Directed Acyclic Graph (DAG) also underscore the advantages of distributed processing. These visualizations detail how tasks like data reading, exchange, and transformations are executed in parallel. Optimizations such as "WholeStageCodegen" and "mapPartitions" seen in the DAG clearly illustrate the steps Spark takes to reduce execution overhead, improving efficiency.

Finally, PySpark optimizes resources and costs by allowing the use of commodity hardware clusters instead of expensive high-performance single-node systems. This makes it not only effective but also a cost-efficient solution for organizations handling big data.

Results and Analysis

The implementation of PySpark for distributed machine learning has demonstrated significant improvements in efficiency and performance across various models. This section discusses the results observed in terms of execution time, accuracy, precision, recall, and F1 score, with comparisons between PySpark and non-PySpark implementations. The analysis also highlights the impact of dataset corrections on the models' performance.

PySpark's distributed architecture significantly reduced training times for all tested models. For instance, the training time for Random Forest dropped from 189.06 seconds in the non-PySpark setup to 46.43 seconds in PySpark, a reduction of nearly 75 percent. Similarly, Logistic Regression saw a reduction from 42.09 seconds to 27.60 seconds, while Decision Tree decreased from 23.23 seconds to 15.87 seconds. Even computationally heavy models like XGBClassifier experienced a reduction in training time, from 129.02 seconds to 112 seconds. These results underscore PySpark's capability to leverage distributed computing for faster processing of large datasets, making it particularly beneficial for machine learning tasks at scale.

Accuracy levels between the PySpark and non-PySpark models were comparable, with only slight differences observed in some cases. Decision Tree, for example, achieved an accuracy of 89.05 percent in PySpark compared to 89.56 percent in the non-PySpark setup. This consistency in accuracy highlights PySpark's ability to maintain the quality of predictions while significantly reducing execution time. Furthermore, the use of the corrected dataset (DAMAGE), addressing the imbalance issues present in Phase 2, led to better overall accuracy across models.

The F1 score, a critical metric balancing precision and recall, showed improvements in PySpark models due to better handling of class imbalances. Decision Tree achieved an F1 score of 0.8557 in PySpark, reflecting its robustness and ability to provide accurate and balanced predictions. XGBClassifier, another strong performer, maintained an F1 score of 0.8532, further validating PySpark's effectiveness in distributed setups.

Precision and recall metrics also saw enhancements with PySpark. For instance, Decision Tree achieved a precision of 0.8597 and recall of 0.8889 in PySpark, compared to marginally lower values in non-PySpark setups. Random Forest similarly showed competitive metrics, with a precision of 0.784 and recall of 0.885 in PySpark. These results emphasize PySpark's ability to optimize model performance by effectively utilizing distributed resources and addressing data-related challenges.

The shift from using the imbalanced Crash Severity column in Phase 2 to the corrected DAMAGE column in this phase was pivotal. The previous dataset imbalance negatively impacted model performance, particularly in recall and precision metrics. By focusing on the DAMAGE column, models

achieved better-balanced predictions, with improved metrics across the board.

Comparative analysis revealed that PySpark models consistently outperformed non-PySpark implementations in execution time while maintaining competitive accuracy, precision, recall, and F1 scores. Non-PySpark models occasionally achieved marginally higher accuracy, as observed with Decision Tree, but the differences were negligible. PySpark's ability to handle large datasets efficiently, combined with its scalability, made it the more practical choice for this analysis.

Visualizations comparing the training time and accuracy of PySpark and non-PySpark models illustrated the trade-offs clearly. While PySpark significantly reduced execution times, it maintained similar or slightly better accuracy levels, demonstrating its effectiveness in distributed processing.

IX. Recommendations

1) Given the significant reduction in training time achieved with PySpark, distributed processing should be prioritized for handling large datasets in future projects. PySpark's ability to scale seamlessly across clusters makes it ideal for resource-intensive tasks, particularly when dealing with real-time data or extensive historical records.

2) The shift to using the DAMAGE column in this phase demonstrated the importance of selecting well-balanced and contextually relevant features. Future projects should continue this approach, ensuring data preprocessing steps effectively address imbalance issues to improve model reliability and accuracy.

3) XGBClassifier consistently performed as the best model, achieving the highest accuracy and robust precision-recall metrics. It should be the model of choice when high accuracy and interpretability of feature importance are critical. However, its computational cost can be mitigated using distributed frameworks like PySpark.

4) While this project focused on selected features like DAMAGE, future work should explore advanced feature engineering techniques, such as interaction terms, derived metrics, or domain-specific transformations, to uncover deeper insights and improve model performance further.

5) Expanding the dataset with external sources, such as weather data, road infrastructure details, or traffic density information, can add context to crash predictions and improve model precision. These additional features could enhance the applicability of the models in diverse scenario.

6) As new data becomes available, the models should be periodically retrained and validated to ensure they remain accurate and relevant. Incorporating a continuous learning pipeline using distributed frameworks like PySpark will help maintain model efficacy over time.

X. Conclusion

This project effectively leveraged machine learning techniques and distributed computing frameworks to predict traffic crash severity and identify factors contributing to crash outcomes. By shifting focus to the DAMAGE column in Phase 3, we addressed the data imbalance challenges faced in Phase 2, resulting in a more balanced dataset and improved reliability of predictions. The study not only enhanced our understanding of high-risk factors but also provided actionable insights for policy and infrastructure improvements aimed at enhancing road safety.

The integration of PySpark for distributed processing significantly improved the scalability and efficiency of the project. PySpark reduced training times across all models, with Random Forest dropping from 189.06 seconds (non-PySpark) to 46.43 seconds, while maintaining competitive or improved performance metrics. The distributed architecture allowed for the seamless handling of large-scale datasets, enabling the efficient training of complex models like XGBClassifier, which achieved the highest accuracy of 89.06 percent in PySpark. These findings highlight PySpark's role as a pivotal tool for large-scale machine learning tasks, where efficiency and speed are crucial.

Machine learning models such as Decision Tree, Random Forest, XGBClassifier, and Logistic Regression demonstrated strong performance metrics, including accuracy, precision, recall, and F1 scores. XGBClassifier emerged as the best model overall due to its robust handling of complex feature interactions and high accuracy. Decision Tree and Random Forest also provided interpretable and reliable results, reinforcing their suitability for safety-critical applications. On the other hand, models like Naive Bayes struggled with this dataset due to their simplifying assumptions, illustrating the need to align model selection with the dataset's complexity.

The Directed Acyclic Graph (DAG) visualizations provided valuable insights into the distributed execution stages, highlighting PySpark's optimization strategies, such as parallel task execution, caching, and shuffle minimization. These visualizations reinforced the understanding of how distributed frameworks process data, making the approach both transparent and efficient.

Overall, this project successfully demonstrated the practical benefits of combining machine learning with distributed processing for addressing real-world challenges. By identifying critical factors influencing crash outcomes and optimizing predictive models, this study provides a strong foundation for implementing data-driven solutions to reduce crash severity. Future work could focus on integrating additional external datasets, exploring advanced hyperparameter tuning, and applying these models in real-time crash prediction systems to further enhance road safety and resource allocation strategies.

REFERENCES

[1] Data.gov. (2024) Traffic crashes - crashes. Accessed: 2024-09-29. [Online]. Available: https://catalog.data.gov/dataset/traffic-crashes-crashes

[2] The Apache Software Foundation, "Sparkr: R front end for 'apache spark'," 2024, r package version 3.5.1https://www.apache.org https://spark.apache.org. [Online]. Available: https://www.apache.org https://spark.apache.org

[3] M. Pal, "Random forest classifier for remote sensing classification," *International Journal of Remote Sensing*, vol. 26, no. 1, pp. 217–222, 2005. [Online]. Available: https://doi.org/10.1080/01431160412331269698

[4] M. P. LaValley, "Logistic regression," *Circulation*, vol. 117, no. 18, pp. 2395–2399, 2008. [Online]. Available: https://www.ahajournals.org/doi/abs/10.1161/CIRCULATIONAHA.106.682658

[5] M. Kumar, M. Hanumanthappa, and T. V. S. Kumar, "Intrusion detection system using decision tree algorithm," in *2012 IEEE 14th International Conference on Communication Technology*, 2012, pp. 629–634.

[6] F.-J. Yang, "An implementation of naive bayes classifier," in *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2018, pp. 301–306.

[7] S. S. Dhaliwal, A.-A. Nahid, and R. Abbas, "Effective intrusion detection system using xgboost," *Information*, vol. 9, no. 7, 2018. [Online]. Available: https://www.mdpi.com/2078-2489/9/7/149

[8] H. Wang and D. Hu, "Comparison of svm and ls-svm for regression," in *2005 International Conference on Neural Networks and Brain*, vol. 1, 2005, pp. 279–283.