Predictive Analysis of Traffic Crashes in Chicago: Crash Types and Damage Estimation

1st Prathibha Vuyyala
Engineering Science - Data Science
University at Buffalo
Buffalo, US
pvuyyala@buffalo.edu

2nd Tharun Teja Mogili
Engineering Science - Data Science
University at Buffalo
Buffalo, US
tharunte@buffalo.edu

3rd Pavan Pajjuri
Engineering Science - Data Science
University at Buffalo
Buffalo, US
spajjuri@buffalo.edu

***Abstract*—This analysis of Chicago traffic accident data from June to December 2023 leverages PySpark for scalable data preprocessing, machine learning, and performance evaluation. Distributed data processing techniques were used to handle missing values, remove duplicates, resolve inconsistencies, and encode categorical features efficiently across large datasets. PySpark's MLlib was utilized to implement and train machine learning models, taking full advantage of Spark's distributed environment. The performance of these models was evaluated using metrics such as accuracy, precision, and F1 score, highlighting significant improvements over previous non-distributed implementations. The study demonstrates the efficiency and scalability of PySpark for both data preprocessing and advanced analytics in big data workflows.**

I. Introduction

An average span of four days, Chicago can record up to over a thousand car accidents. When you include drivers, passengers, pedestrians and cyclists, up to two thousand people can be effected. Forty-five percent of the people will experience a minor to fatal injury.

Traffic accidents represent a critical challenge for urban safety management, resulting in injuries, fatalities, and economic loss. This project focuses on leveraging the Chicago crashes dataset to address specific problems related to traffic crashes

II. Problem Statement

A. Problem Statement Questions

- **Predicting Crash Type:** Utilizing available data features, we aim to develop a predictive model that can accurately classify the crash type (Injury or No Injury) based on factors such as Road conditions, weather conditions, and traffic control devices. Understanding if the crash could injurious can inform targeted prevention strategies and resource allocation.
- **Identifying Risk Factors for Severe Accidents:** We will investigate how various elements (e.g., environmental conditions, traffic volume) correlate with severe accidents. This analysis aims to uncover actionable insights that can guide interventions aimed at reducing the occurrence and impact of high-severity crashes.

B. Background of the Problem

Traffic accidents in urban environments, such as Chicago, are a persistent concern, contributing to injuries, fatalities, and substantial economic losses. With the increasing complexity of urban traffic systems and the multitude of factors influencing crash events—ranging from road conditions, weather, and traffic control devices to driver behavior—the need for predictive tools has grown. This project aims to explore these complex dynamics by utilizing the Chicago crashes dataset, which provides a wealth of structured data that can be used to uncover patterns and factors contributing to traffic accidents. By focusing on predicting crash types, estimating damage severity, and identifying risk factors for severe accidents, this project addresses key challenges in urban traffic safety management. Furthermore, the project seeks to better understand the conditions under which severe injuries and hit-and-run incidents occur, and how poor roadway surface conditions impact accident rates.

C. Significance of the Problem and Project Contribution

This project is significant because traffic accidents have far-reaching impacts on public safety, economic costs, and emergency services. The ability to predict crash types and estimate damage severity could lead to more efficient allocation of resources, such as emergency services and law enforcement. Moreover, identifying the conditions that lead to severe accidents enables targeted interventions, such as improving road infrastructure or adjusting traffic control measures in high-risk areas. Understanding trends in hit-and-run incidents and the effects of road conditions can guide policy decisions aimed at enhancing traffic safety and accountability. By providing actionable insights through data-driven models, this project has the potential to improve urban traffic management strategies, reduce accident rates, and mitigate the impacts of crashes on public health and safety.

D. Potential Contribution of the Project

This project aims to contribute significantly to the domain of urban traffic safety by developing data-driven models that can predict crash types and estimate damage severity. By leveraging predictive analytics, this project will help inform targeted strategies for accident prevention, particularly in identifying conditions under which crashes are more likely to result in injuries. This predictive capability can enable city authorities, transportation agencies, and policymakers to allocate resources where they are most needed, reducing response times for emergency services and guiding future infrastructure investments.

Moreover, by estimating damage severity, the project will offer valuable insights for planning and improving road safety. Understanding which factors—whether environmental, vehicular, or road-related—contribute to more severe crashes can help refine traffic regulations, implement better warning systems, and design safer intersections. The analysis of hit-and-run incidents can also reveal important patterns that inform policy decisions to enhance public safety, hold offenders accountable, and provide better post-crash support for victims. Additionally, investigating how roadway conditions influence crash rates and severity offers actionable information to guide road maintenance and policy reforms, which are crucial for reducing accident rates in cities with challenging environmental conditions like Chicago. Ultimately, this project will empower stakeholders to proactively manage traffic safety, contributing to a safer, more sustainable urban transport system.

III. Data Sources

The primary dataset for this project comes from the Chicago Traffic Crashes Dataset, which is publicly available through the City of Chicago's data portal. This dataset contains detailed information about traffic accidents that have occurred in the city of Chicago, with over 870,000 records spanning multiple years, and includes variables such as crash type, weather conditions, road surface conditions, and traffic control device statuses.

The data from the Chicago Traffic Crashes dataset is comprehensive and includes numerous variables that allow for an in-depth analysis of crash incidents. With over 870,000 rows and 48 columns, this dataset is well-suited for addressing the project's core objectives of predicting crash types, estimating damage severity, and identifying risk factors for severe accidents. This data source ensures that the dataset is both large enough to yield significant insights and contains the necessary variety of features to facilitate predictive modeling and exploratory data analysis. The dataset from the Chicago site spans from 2013 to the present. For the purposes of this analysis, we will focus exclusively on a six-month period, specifically from June 2023 to December 2023

You can access the data source at Data Source.

IV. Data Cleaning/ Processing

Effective data cleaning and processing is essential to ensure the accuracy and reliability of any data analysis or machine learning workflow. The Chicago Traffic Crashes dataset, like most real-world datasets, contains inconsistencies, missing values, and redundant information that must be addressed. To handle these challenges efficiently, PySpark's distributed computing capabilities were leveraged. Using PySpark's RDD and DataFrame APIs, the dataset was processed in a distributed environment, ensuring scalability and performance optimization. Below is an outline of the 8 distinct processing and cleaning steps applied to the dataset using PySpark:

A. Dropping Null Rows and Columns

The dataset contains records with missing values, prompting the need for a structured approach to manage these gaps. A column threshold of 30% was set, leading to the removal of columns with excessive missing data. Consequently, the following columns were dropped: *WORK_ZONE_TYPE, NOT_RIGHT_OF_WAY_I, STATEMENTS_TAKEN_I, PHOTOS_TAKEN_I, WORK_ZONE_I, CRASH_DATE_EST_I, WORKERS_PRESENT_I, LANE_CNT, DOORING_I, INTERSECTION_RELATED_I*. This process reduced the dataset from 48 columns to 37 columns. Additionally, a row threshold of 20% was implemented to eliminate rows with excessive missing values, ensuring the quality of the remaining data and minimizing potential bias in subsequent analyses.

Compared to the previous implementation in pandas, the PySpark approach offers significant advantages in scalability and performance. While pandas processes data sequentially in memory, PySpark utilizes distributed computing to process large datasets efficiently. The use of dynamic threshold calculations and distributed filtering enables processing of data spread across multiple nodes, ensuring that operations scale seamlessly with dataset size. Additionally, the helper column for row filtering is processed in parallel, improving execution speed significantly for large-scale data cleaning tasks.

B. Handling Missing Data

The dataset was analyzed to identify missing values in both categorical and numerical columns. For categorical variables such as `REPORT_TYPE`, `HIT_AND_RUN_I`, and `MOST_SEVERE_INJURY`, missing values were imputed with the mode of each column to maintain consistency. The mode values used were:

- `REPORT_TYPE`: "NOT ON SCENE (DESK REPORT)"
- `MOST_SEVERE_INJURY`: "NO INDICATION OF INJURY"

For numerical variables, missing values were also filled using the mode. Key columns with missing values included:

- `INJURIES_TOTAL`
- `INJURIES_FATAL`
- `INJURIES_INCAPACITATING`
- `INJURIES_NON_INCAPACITATING`
- `INJURIES_REPORTED_NOT_EVIDENT`
- `INJURIES_NO_INDICATION`
- `INJURIES_UNKNOWN`
- `LATITUDE`
- `LONGITUDE`

Unlike the pandas implementation, which relies on in-memory operations, PySpark provides a scalable and efficient solution for handling missing data in large datasets. Imputing categorical values using PySpark involved distributed operations like `groupBy` and aggregation to calculate the mode, which can handle significantly larger datasets compared to pandas. Similarly, the use of PySpark `MLlib's Imputer` for numerical columns allowed seamless handling of missing values across distributed data, leveraging Spark's parallel processing capabilities. This ensured reduced computation time and efficient memory usage, making it ideal for big data workflows

C. Standardization of Numerical Features

The numerical features in the dataset were standardized to ensure they have a mean of 0 and a standard deviation of 1, a critical step for improving the performance and stability of machine learning algorithms. This process was applied to all numerical columns identified in the dataset. Using PySpark's `StandardScaler` from the `MLlib` library, the numerical features were scaled efficiently in a distributed environment. The scaler transformed the features into a unified scale, ensuring that no feature dominated others due to differences in magnitude. Summary statistics for the scaled features, such as the mean and standard deviation, were generated to validate the transformation.

The PySpark implementation differs from the previous pandas-based workflow, which relied on StandardScaler from scikit-learn. Unlike scikit-learn, which operates in-memory and on a single machine, PySpark's StandardScaler scales the features in a distributed manner across a cluster, enabling processing of much larger datasets. Additionally, the use of `VectorAssembler` and transformation pipelines allowed seamless handling of multiple numerical columns simultaneously, improving both scalability and efficiency. This approach minimizes memory usage and optimizes execution time for big data scenarios.

D. Dealing with Categorical Values and Inconsistency - Mapping

The dataset contains several categorical variables that may have inconsistent entries, which could affect the analysis. A systematic mapping approach was employed to standardize the values in key categorical columns, ensuring uniformity. The following steps were taken:

- Traffic Control Device: Multiple entries were consolidated into broader categories (e.g., "TRAFFIC SIGNAL", "FLASHING CONTROL SIGNAL", and "RAILROAD CROSSING GATE" were all mapped to "SIGNAL"). This mapping reduced variability and enhanced the clarity of data, resulting in a more manageable distribution of categories.
- Weather Conditions: Various weather descriptions were similarly standardized to ensure consistency (e.g., "RAIN" includes "FREEZING RAIN/DRIZZLE"). This allows for better comparison and analysis of accident occurrences under different weather conditions.
- Primary Contributory Cause: The mapping transformed numerous specific causes of crashes into broader categories (e.g., "FAILING TO YIELD RIGHT-OF-WAY" became "YIELDING ISSUES"). This aids in summarizing the data while still retaining essential information for analysis.

The resulting distributions of the standardized categories were examined, revealing clearer patterns and relationships among the variables, which are crucial for subsequent analysis.

In the pandas-based approach, mapping values typically involves the use of the replace method or dictionary mappings. While effective for smaller datasets, it becomes computationally expensive and memory-intensive when applied to large datasets. PySpark's implementation, using distributed transformations with `when` and `withColumn`, allows these mappings to scale seamlessly across a cluster. This approach ensures faster processing and eliminates memory bottlenecks, particularly when grouping and counting the standardized categories.

E. Data Type Change

The `DATE_POLICE_NOTIFIED` column was converted to a datetime format using `pd.to_datetime()`, allowing for efficient date manipulation and analysis. The format specified was `%m/%d/%Y %I:%M:%S %p`, and any errors during conversion were handled by coercing them to `NaT`.

Several columns related to the timing of crashes were transformed to the category data type to optimize memory usage and improve performance during analysis. These columns included:

- `CRASH_HOUR`: Representing the hour of the crash.
- `CRASH_DAY_OF_WEEK`: Indicating the day of the week the crash occurred.
- `CRASH_MONTH`: Denoting the month in which the crash happened.

By converting these columns to categorical types, the dataset is better structured for analysis and visualization tasks that may benefit from treating these values as distinct categories.

F. Redundant Columns Removal

Certain columns identified as redundant were removed from the dataset to streamline the data and enhance its utility for analysis. These columns were deemed of little use either due to their limited analytical value or because their information was already captured in other columns. The columns removed include:

- `LOCATION`: Contains information that is duplicative or less relevant.
- `SEC_CONTRIBUTORY_CAUSE`: Provides minimal additional insights beyond the primary cause.
- `STREET_DIRECTION`: Does not significantly contribute to the analysis.
- `STREET_NAME`: Offers little additional context for the analysis being conducted.
- `STREET_NO`: Similar to `STREET_NAME`, it does not add meaningful information.

After removing these columns, the dataset was reduced to 31 columns with 65,645 rows, enhancing its focus and utility for subsequent steps like feature selection and modeling.

In the pandas implementation, redundant columns are typically dropped using the drop method, which works effectively for small datasets. However, in PySpark, the `drop` method processes columns in a distributed manner, ensuring scalability when dealing with larger datasets. Additionally, the ability to handle multiple columns simultaneously in a distributed environment minimizes runtime and resource consumption. This approach ensures that large-scale datasets can be efficiently streamlined without memory constraints.

G. Rolling Metrics and Temporal Rankings Using Window Functions

The dataset contains traffic-related records that can benefit from temporal and aggregate analyses to uncover patterns and trends. To achieve this, windowing operations were applied to calculate rolling metrics and rankings, providing a deeper understanding of variations in posted speed limits and crash sequences.

A window specification was defined to partition the dataset by `CRASH_HOUR` and order the rows within each partition by `CRASH_DATE`. Using this specification, two key transformations were performed:

- `Rolling Average`: The rolling average of `POSTED_SPEED_LIMIT` was computed within each partition. This smoothed metric highlights variations in speed limits over time while minimizing the impact of outliers.
- `Ranking`: A temporal rank was assigned to each crash within its partition based on `CRASH_DATE`, enabling the identification of sequential patterns in crash occurrences.

These transformations resulted in the creation of two additional columns: rolling_avg_speed and rank. These features provide valuable insights into the temporal dynamics and aggregate behavior of traffic conditions, enhancing the dataset for further analysis and modeling.

PySpark's windowing functions are optimized for distributed computation, making them highly efficient for large datasets. Unlike traditional approaches that may require iterative calculations, PySpark processes rolling metrics and rankings in parallel across partitions. This ensures scalability and speed, even for datasets spanning millions of records. Additionally, the ability to define flexible window specifications allows for advanced analytics tailored to specific use cases, such as grouping by time or location.

H. Categorical Encoding

- Ordinal Encoding: Specific columns with a natural order were transformed using ordinal encoding to convert categorical values into numerical representations. The mappings for each column were defined as follows:
  - `Lighting Condition`: Values were assigned from 1 (*DAYLIGHT*) to 6 (*UNKNOWN*).
  - `Most Severe Injury`: Injury severity was mapped from 1 (*NO INDICATION OF INJURY*) to 5 (*FATAL*).
  - `Report Type`: Categorical values were encoded to 1 (*NOT ON SCENE*) and 2 (*ON SCENE*).
  - `Crash Type`: Encoded as 1 (*NO INJURY*) and 2 (*INJURY/TOW*).
  - `Damage`: Categorized into three levels, from 1 (*OVER $1,500*) to 3 (*$500 OR LESS*).
- Nominal Encoding: For columns that do not have a natural order, one-hot encoding was applied. This method created binary columns for each category within the specified nominal columns:
  - `Traffic Control Device`
  - `Device Condition`
  - `Weather Condition`
  - `First Crash Type`
  - `Trafficway Type`
  - `Alignment`
  - `Roadway Surface Condition`
  - `Road Defect`
  - `Primary Contributory Cause`

Each category within these columns was indexed using `StringIndexer` and then converted into binary vectors using `OneHotEncoder`. After encoding, the original columns and intermediate index columns were dropped to retain only the encoded features, streamlining the dataset for analysis.

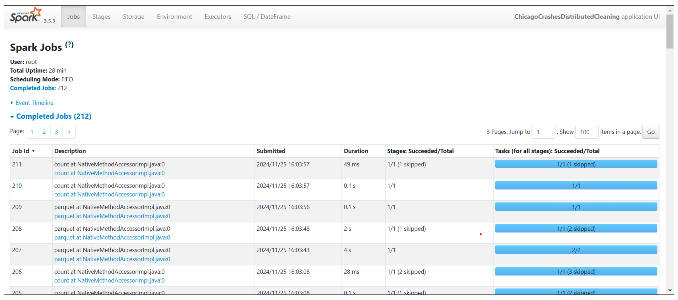
These encoding techniques transformed the dataset into a machine-learning-ready format, ensuring all categorical features were represented numerically while preserving their relationships and significance.

In pandas, ordinal encoding typically involves mapping categorical values using dictionaries, while one-hot encoding uses pd.get_dummies. These methods are effective for smaller datasets but become computationally expensive for large datasets with many categories. PySpark's implementation, on the other hand, leverages distributed computing, enabling efficient transformation of categorical variables at scale.

Using StringIndexer and OneHotEncoder, PySpark processes multiple columns in parallel, ensuring faster computation and optimized memory usage for large datasets. Additionally, PySpark allows seamless integration of these encodings into machine learning pipelines, reducing manual effort and streamlining the workflow for big data tasks. This scalability and performance improvement make PySpark highly suitable for datasets

DAG's Visualization and Analysis for Preprocessed steps

A Directed Acyclic Graph (DAG) in Spark is a fundamental abstraction that represents the sequence of operations to be performed on data, such as transformations and actions. Spark uses DAGs to optimize the execution of tasks across its distributed environment. Each job in Spark is broken into stages, which are further split into parallel tasks. The DAG provides a visual representation of these stages and their dependencies, allowing users to analyze the execution plan and identify potential bottlenecks.



Spark Jobs (?)

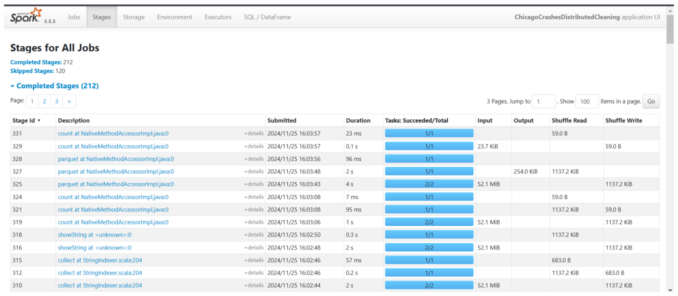
User: root
Total Uptime: 28 min
Scheduling Mode: FIFO
Completed Jobs: 212

▸ Event Timeline
▾ **Completed Jobs (212)**

Page: 1 2 3 > 3 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
211	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2024/11/25 16:03:57	49 ms	1/1 (1 skipped)	1/1 (1 skipped)
210	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2024/11/25 16:03:57	0.1 s	1/1	1/1
209	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2024/11/25 16:03:56	0.1 s	1/1	1/1
208	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2024/11/25 16:03:48	2 s	1/1 (1 skipped)	1/1 (2 skipped)
207	parquet at NativeMethodAccessorImpl.java:0 parquet at NativeMethodAccessorImpl.java:0	2024/11/25 16:03:43	4 s	1/1	2/2
206	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2024/11/25 16:03:08	28 ms	1/1 (2 skipped)	1/1 (3 skipped)
205	count at NativeMethodAccessorImpl.java:0	2024/11/25 16:03:08	0.1 s	1/1 (1 skipped)	1/1 (2 skipped)

Fig. 1. Dag Visualization: Completed Jobs



Stages for All Jobs

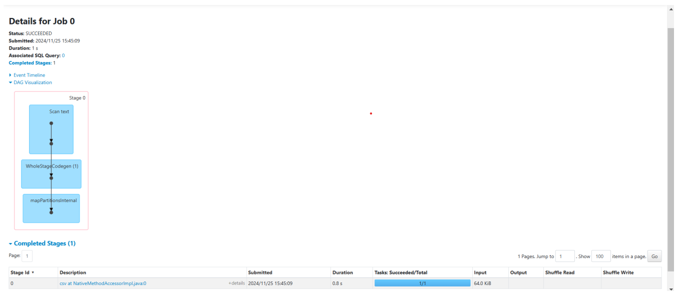
Completed Stages: 212
Skipped Stages: 120

▾ **Completed Stages (212)**

Page: 1 2 3 > 3 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
331	count at NativeMethodAccessorImpl.java:0 +details	2024/11/25 16:03:57	23 ms	1/1			59.0 B	
329	count at NativeMethodAccessorImpl.java:0 +details	2024/11/25 16:03:57	0.1 s	1/1	23.7 KiB			59.0 B
328	parquet at NativeMethodAccessorImpl.java:0 +details	2024/11/25 16:03:56	96 ms	1/1				
327	parquet at NativeMethodAccessorImpl.java:0 +details	2024/11/25 16:03:48	2 s	1/1		254.0 KiB	1137.2 KiB	
325	parquet at NativeMethodAccessorImpl.java:0 +details	2024/11/25 16:03:43	4 s	2/2	52.1 MiB			1137.2 KiB
324	count at NativeMethodAccessorImpl.java:0 +details	2024/11/25 16:03:08	7 ms	1/1			59.0 B	
321	count at NativeMethodAccessorImpl.java:0 +details	2024/11/25 16:03:08	95 ms	1/1			1137.2 KiB	59.0 B
319	count at NativeMethodAccessorImpl.java:0 +details	2024/11/25 16:03:06	1 s	2/2	52.1 MiB			1137.2 KiB
318	showString at <unknown>:0 +details	2024/11/25 16:02:50	0.3 s	1/1			1137.2 KiB	
316	showString at <unknown>:0 +details	2024/11/25 16:02:48	2 s	2/2	52.1 MiB			1137.2 KiB
315	collect at StringIndexer.scala:204 +details	2024/11/25 16:02:46	57 ms	1/1			683.0 B	
312	collect at StringIndexer.scala:204 +details	2024/11/25 16:02:46	0.2 s	1/1			1137.2 KiB	683.0 B
310	collect at StringIndexer.scala:204 +details	2024/11/25 16:02:44	2 s	2/2	52.1 MiB			1137.2 KiB

Fig. 2. Dag Visualization: Stages



Details for Job 0

Status: SUCCEEDED
Submitted: 2024/11/25 15:45:09
Duration: 1 s
Associated SQL Query: 0
Completed Stages: 1

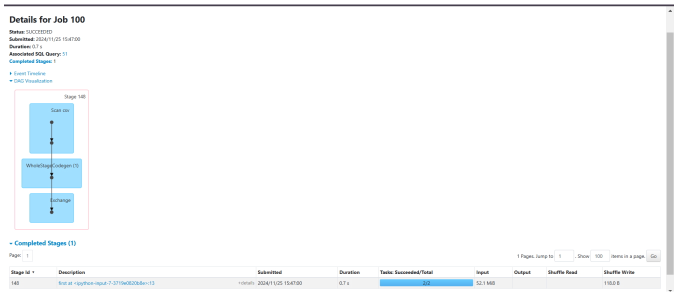
▸ Event Timeline
▾ DAG Visualization

▾ **Completed Stages (1)**

Page: 1 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
0	csv at NativeMethodAccessorImpl.java:0 +details	2024/11/25 15:45:09	0.8 s	1/1	64.0 KiB			

Fig. 3. Dag Visualization: Job 0



Details for Job 100

Status: SUCCEEDED
Submitted: 2024/11/25 15:47:00
Duration: 0.7 s
Associated SQL Query: 51
Completed Stages: 1

▸ Event Timeline
▾ DAG Visualization

▾ **Completed Stages (1)**

Page: 1 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
148	first at <ipython-input-7-3719e0820b8e>:13 +details	2024/11/25 15:47:00	0.7 s	2/2	52.1 MiB			118.0 B

Fig. 4. Dag Visualization: Job 100

The DAG visualization, accessible via the Spark UI, serves as a valuable tool for understanding how operations are distributed across the cluster. It showcases how the data flows through various stages, highlights shuffling (data movement between partitions), and provides insights into Spark's optimization strategies such as caching, partitioning, and task parallelization.

1) Number of Jobs and Tasks in the Pipeline: The data preprocessing pipeline executed a total of 212 jobs. Each job represents a logical operation, such as filtering, aggregations, or feature transformations. These jobs were divided into 212 stages, and Spark optimized the execution plan by skipping 120 stages, leveraging its caching and optimization techniques. The skipped stages indicate that intermediate results were reused, avoiding redundant computations.

Within each stage, tasks are further split based on the size of the data and the operations being performed. For example, simple operations like loading data may result in a single task if the dataset is small. In contrast, operations requiring shuffling or repartitioning the data generate multiple tasks to ensure efficient parallel execution.

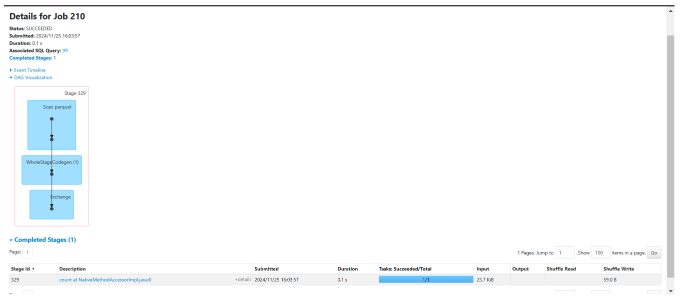
Given the large number of jobs and stages, this report focuses on three key jobs:

- `Job 0 (Initial Job)`: Loading the raw CSV file.
- `Job 100 (Middle Job)`: Applying intermediate transformations like filtering and type conversion.
- `Job 210 (Final Job)`: Performing aggregations, feature scaling, and encoding.

Job 0: Initial Data Loading The objective of Job 0 was to load the raw CSV file into a Spark DataFrame while inferring its schema. The dataset contained over 1.5 million rows with a size of approximately 64 KiB. This job involved a single stage comprising three key operations: reading the CSV file line by line (Scan text), optimizing the parsing process into a single computational unit using WholeStageCodegen, and processing partitions in parallel with mapPartitionsInternal. Notably, the job executed only one task since the input data size was small and fit into a single partition, given Spark's default partition size of 128 MB. The absence of shuffling or repartitioning highlighted the simplicity of this stage, which was equivalent to loading the entire file into memory in Pandas. However, Spark's approach ensures scalability for larger datasets by distributing data across partitions as needed.

Job 100: Intermediate Data Transformations Job 100 focused on filtering the dataset based on a specific date range and converting the CRASH_DATE column into a timestamp format. These transformations prepared the data for subsequent preprocessing. The job's DAG illustrated a single stage involving the following operations: reading the filtered rows (Scan csv), combining filtering and type conversion into an optimized stage via WholeStageCodegen, and performing a shuffle operation (Exchange) to repartition the data. This job executed two tasks due to the shuffling required to redistribute the filtered data across partitions for balanced parallel processing. While filtering and transformation would typically run sequentially in Pandas, Spark's parallel execution provided significant performance benefits, especially for larger datasets.

Job 210: Final Aggregations and Feature Engineering The final job, Job 210, involved aggregating data, computing rolling averages, and encoding categorical features. The cleaned dataset was read from Parquet files and processed to generate the final output. The DAG for this job showcased



Details for Job 210

Status: SUCCEEDED
Submitted: 2024/11/25 16:03:57
Duration: 0.1 s
Associated SQL Query: 99
Completed Stages: 1

Event Timeline
DAG Visualization

Completed Stages (1)

Page: 1 | 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
329	count at NativeMethodAccessorImpl.java:0 +details	2024/11/25 16:03:57	0.1 s	1/1	23.7 KiB			59.0 B

Fig. 5. Dag Visualization: Job 210

operations such as reading data from Parquet format (Scan parquet), optimizing computations for feature engineering via WholeStageCodegen, and performing a shuffle (Exchange) to enable grouped operations like rolling averages and ranking. Due to the small dataset size at this stage (23.7 KiB), only one task was executed, as no additional partitioning was necessary. The use of Parquet format, which stores data in a compressed and partitioned manner, further enhanced efficiency. In comparison, performing similar steps in Pandas would require processing the entire dataset in memory, which could lead to performance bottlenecks for larger datasets.

The DAG visualizations and task splitting demonstrate the efficiency of Spark's distributed execution model. Compared to Pandas, Spark offers several advantages:

Scalability: Spark distributes data and computations across multiple nodes, allowing it to handle datasets that exceed the memory capacity of a single machine. Pandas, on the other hand, processes data sequentially and is limited by available memory.

Parallel Execution: Spark divides operations into tasks that are executed in parallel across the cluster. This reduces execution time significantly, especially for large datasets. In contrast, Pandas operates in a single-threaded environment.

Fault Tolerance: Spark's Resilient Distributed Dataset (RDD) abstraction ensures fault tolerance by keeping track of lineage information. If a task fails, Spark can recompute the lost partition without restarting the entire pipeline. Pandas does not offer fault tolerance, making it prone to complete failure in case of errors.

Optimizations: Spark optimizes the execution plan through techniques like WholeStageCodegen, shuffling minimization, and caching of intermediate results. These optimizations reduce redundant computations and improve performance. Pandas lacks these built-in optimization mechanisms.

Conclusion: By leveraging Spark's DAG visualization and distributed architecture, this pipeline achieved efficient preprocessing of a large dataset. While Pandas is suitable for smaller datasets, Spark's ability to handle large-scale data with parallelism and fault tolerance makes it a superior choice for preprocessing workflows involving complex transformations and aggregations.

V. ML Modeling and Algorithm Applications

The application of several machine learning algorithms was crucial to addressing key aspects of the traffic crash data analysis, including the prediction of crash types and the identification of high-risk factors contributing to severe accidents. Each algorithm was selected based on its suitability for classification tasks and its capacity to process the structured data provided in the Chicago Traffic Crashes dataset. The models applied include Logistic Regression, Linear Regression Classifier, Naive Bayes, Support Vector Machines (SVM), Random Forest Classifier and XGBoost Classifier all aimed at generating predictive insights from the data. Appropriate hyperparameters were tuned for each algorithm using hyperparameter optimization techniques available in PySpark MLlib to optimize performance and ensure robust outcomes. The effectiveness of these models was evaluated using key performance metrics such as accuracy, precision, recall, and F1-score. Visualizations were also generated to aid in the interpretation of results, providing clarity on the factors influencing crash outcomes and damage levels.

The PySpark implementation was further compared to the earlier pandas-based workflow. PySpark's ability to handle large datasets in a distributed environment offered significant advantages in scalability and speed. The DAG execution model allowed seamless parallelization of training and evaluation processes, resulting in faster computation times without compromising accuracy. Additionally, performance visualizations, such as confusion matrices,models comparision plots were generated to interpret model outcomes and highlight factors contributing to crash severity.

1) Models Tuning: The primary objective of this step was to identify the optimal hyperparameters for each machine learning model using the validation data, which would then be applied to the training dataset for improved model performance. Initially, the dataset was preprocessed by dropping categorical columns that were not required for the model training. Subsequently, the data was split into training, validation, and test sets, ensuring stratification to preserve class distributions. The features were scaled using StandardScaler to standardize the data, which is essential for algorithms like logistic regression and SVM that are sensitive to feature scaling.

Hyperparameter tuning was carried out using PySpark's CrossValidator and ParamGridBuilder for each model, which included Logistic Regression, SVM, Random Forest, and XGBoost. A predefined set of hyperparameters was evaluated through cross-validation on the training data, and the best parameters for each model were selected based on the validation accuracy. This process ensured that the models were trained with the most optimal configurations, contributing to the overall improvement in predictive performance. The validation scores for the models ranged from 86% for Random Classifier to 89% for XGBoost, indicating the models' robust ability to generalize across unseen data.

The parameters evaluated included:

- `XGBoost Classifier`: Tuned parameters such as max_depth, learning_rate, and n_estimators (boosting rounds) to balance complexity and learning rate.
- `Logistic Regression`: Adjusted regParam (regularization strength) and elasticNetParam to optimize the trade-off between regularization types (L1 and L2).
- `SVM`: Tuned regParam for optimal regularization to avoid underfitting or overfitting.
- `Random Forest Classifier`: Optimized parameters such as numTrees (number of trees), maxDepth (tree depth), and minInstancesPerNode to improve decision boundaries and feature usage.

For the combination of parameters for different models, the best hyperparameters achieved are as follows

- `XGBoost Classifier`:
  max depth : 3
  learning rate: 0.01
  n estimators: 50
- `Logistic Regression`:
  regParam: 0.1
  elasticNetParam: 0.0
- `SVM`:
  regParam: 0.1
- `Random Forest Classifier`:
  numTrees: 50
  maxDepth: 5
  minInstancesPerNode: 2

In comparing hyperparameter tuning for machine learning models using Pandas and PySpark, we observe significant differences in execution time and validation accuracy. Both approaches explored similar models—Logistic Regression (log_reg), Support Vector Machine (SVM), Random Forest (RF), and XGBoost (XGB)—with a focus on optimizing hyperparameters for model performance.

Both frameworks achieved comparable validation accuracies across models. For example, in Logistic Regression, PySpark slightly outperformed Pandas (0.8909 vs. 0.8890). Similarly, for SVM and XGB, validation accuracies were nearly equivalent, underscoring that both frameworks deliver effective model tuning in terms of performance.

The stark difference lies in execution time. Pandas took 2704.59 seconds, whereas PySpark completed the task in 639.74 seconds—approximately 4.2 times faster. This efficiency gain in PySpark is due to its distributed computing architecture, which allows tasks to be parallelized and executed across multiple cores or nodes. In contrast, Pandas operates in a single-threaded environment, making it slower for large datasets or computationally intensive tasks like hyperparameter tuning. PySpark's advantage lies in its scalability and ability to handle large datasets efficiently. By distributing data and computations across a cluster, PySpark minimizes memory bottlenecks and accelerates iterative tasks, such as cross-validation and hyperparameter searches. This is particularly beneficial for complex models like XGBoost or Random Forest, which involve multiple iterations over large parameter spaces.

PySpark is the preferred choice for hyperparameter tuning tasks in scenarios involving larger datasets or high computational demands.

2) Naive Bayes: Naive Bayes is a probabilistic classification algorithm based on Bayes' theorem. It assumes that the features are independent given the class label, which is often referred to as the "naive" assumption. Despite this strong assumption, Naive Bayes is highly effective for certain types of classification problems, particularly when the features are mostly independent. It is computationally efficient and performs well on large datasets, making it a good choice for a baseline model.

a) Model Tuning and Training: The Naive Bayes model was implemented using PySpark's NaiveBayes class with the multinomial variant, which is suitable for classification tasks involving discrete features. Since Naive Bayes requires non-negative inputs, the scaled features were adjusted by shifting all values to ensure positivity before model training. Unlike other models, Naive Bayes does not require hyperparameter tuning, simplifying the training process. The model was trained on the shifted features from the training dataset (train_data_nb). PySpark's distributed architecture ensured efficient handling of the large dataset, maintaining computational efficiency even for extensive data.

b) Effectiveness and Metrics: The The Naive Bayes model achieved a test accuracy of 86.1%, slightly outperforming the pandas-based implementation. Detailed performance metrics are as follows:

- Class 0 (Negative Class) was handled effectively with a precision of 97.4%, showing that most of the predicted negative cases were correct, and a recall of 97.1%, indicating that it captured nearly all actual negative instances in the dataset. The F1-score of 97.2% reflects an excellent balance between precision and recall, underscoring the model's ability to handle the dominant class effectively.
- Class 1 (Positive Class) demonstrated a precision of 53.7%, indicating that just over half of the predicted positive cases were correct. The recall was also 53.7%, showing that the model missed a significant number of actual positive cases. This limitation is reflected in the F1-score of 69.2%, which highlights the trade-off between precision and recall for the minority class.

c) Insights Gained: Naive Bayes continues to prove itself as a computationally efficient baseline model for binary classification. The model excels in handling Class 0, which dominates the dataset, but its performance is limited for Class 1 due to the independence assumption inherent to Naive Bayes. The lower recall for positive cases suggests that this model may not be ideal for problems where identifying minority cases is critical. Nevertheless, its quick training and reasonable accuracy make it a useful starting point in the machine learning pipeline.

d) Confusion Matrix Insights: The confusion matrix provides a clear view of where the Naive Bayes model performs

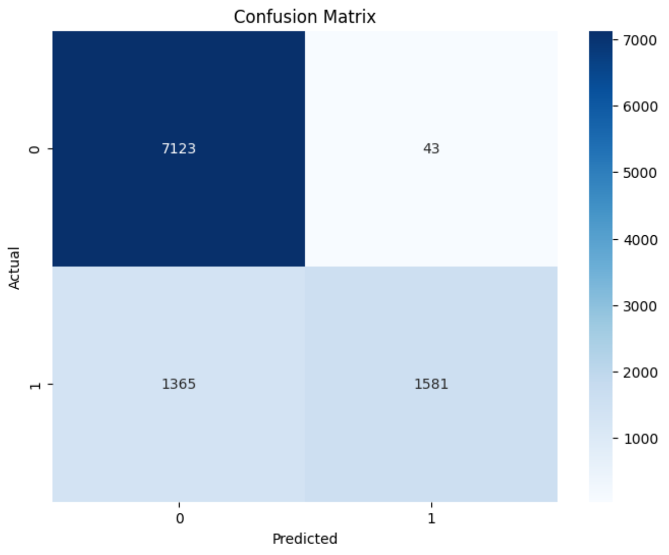


Fig. 6. Confusion Matrix: Naive Bayes

well and where it struggles:

- True Negatives (7123): he model correctly predicted a large number of negative cases, demonstrating strong performance for the dominant class.
- True Positives (1581): The model successfully identified a moderate number of positive cases but missed some, reflected in the false negative count.
- False Positives (43): The model produced relatively few false positives, indicating good precision for Class 0.
- False Negatives (1365): The model struggled with false negatives, missing several positive cases and contributing to the lower recall for Class 1.

e) Comparision of pyspark implementation with previous Pandas Implementation: The PySpark implementation of the Naive Bayes model demonstrated notable improvements over the pandas-based implementation in Phase 2. While both models achieved comparable accuracy, with PySpark slightly outperforming at 86.1% compared to 85.6% in pandas, PySpark's distributed computing capabilities provided a significant advantage in execution time. The PySpark implementation achieved a precision of 97.4%, recall of 97.1%, and an F1-score of 97.2%, reflecting its strong ability to handle the dominant class effectively. The pandas implementation, while showing similar recall at 97%, had a lower precision of 85% for the negative class, resulting in a slightly lower F1-score of 91%. For the positive class, PySpark achieved an F1-score of 69.2%, comparable to the 70% in pandas, though pandas exhibited slightly better precision at 90% compared to 53.7% in PySpark.

Pandas executed in 0.11 seconds, while PySpark took significantly longer at 13.72 seconds. Naive Bayes is computationally lightweight, and Pandas handles it quickly. However, PySpark's distributed setup introduces overhead, making it slower for such simple tasks on smaller datasets.

The most significant distinction lies in PySpark's distributed processing, which enabled faster training and evaluation times, especially for the large-scale Chicago Traffic Crashes dataset. Unlike the sequential approach in pandas, PySpark efficiently processed large data partitions in parallel, maintaining performance while scaling to big data requirements.

f) Conclusion: The Naive Bayes model achieved a test accuracy of 86.1% in PySpark, making it a reliable baseline model for the classification task. While the model performed exceptionally well for Class 0, its recall for Class 1 remained a limitation. Naive Bayes is computationally efficient and easy to implement but may require supplementation with more complex algorithms to address its shortcomings in handling minority classes effectively. PySpark's distributed processing capabilities make it particularly advantageous for handling large datasets, offering both speed and scalability over traditional single-machine implementations.

3) Logistic Regression: Logistic Regression was chosen because it is a straightforward, interpretable model that works well for binary classification problems. It's particularly effective when the relationship between features and the target variable is linear, and it can provide probabilities for class membership, which is useful for understanding the confidence of the model's predictions. Given the need for a baseline model that can offer insights into the data and provide solid classification performance, Logistic Regression was a fitting choice.

a) Model Tuning and Training: The Logistic Regression model was tuned using PySpark's CrossValidator with a grid search for hyperparameter optimization. The parameters tuned included regParam (regularization strength) and elasticNetParam (L1-L2 regularization balance), ensuring the model generalized well without overfitting. After selecting the best model based on validation accuracy, the optimized Logistic Regression model was trained on the scaled training dataset (train_data_scaled) and subsequently tested on the scaled test dataset (test_data_scaled).

b) Effectiveness and Metrics: The Logistic Regression model achieved a test accuracy of 89.0%, reflecting its ability to classify most cases correctly. Precision, recall, and F1-score provide additional insights into the model's performance.

- Class 0 (Negative Class) was handled effectively with a precision with a precision of 95.7%, indicating that most predicted negative cases were correct. A recall of 96.0% for the negative class shows that the model captured nearly all negative instances. The F1-score of 95.8% highlights a strong balance between precision and recall.
- Class 1 (Positive Class) showed strong precision of 60.9%, meaning it was moderately accurate in identifying positive cases. However, the recall was lower at 60.9%, indicating that it missed a significant portion of actual positive cases. The F1-score of 74.5% reflects this trade-off, showing that while the model effectively identified many positive cases, some were misclassified as negative.

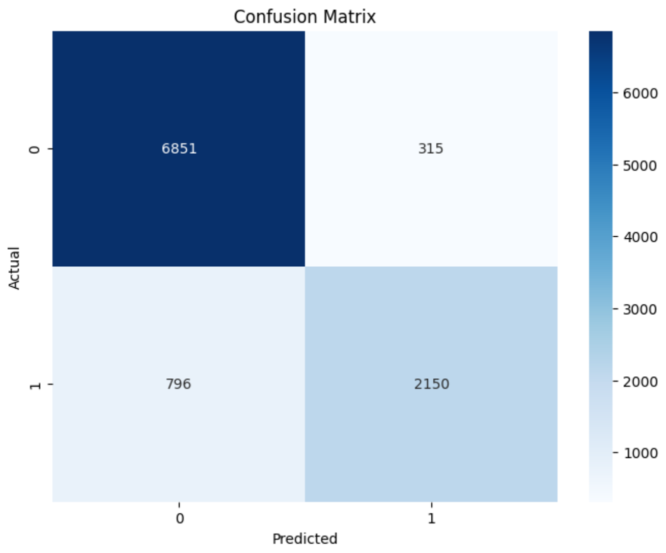


Fig. 7. Confusion Matrix: Logistic Regression

c) Confusion Matrix Insights: The confusion matrix helps visualize where the model excels and where it struggles:

- True Negatives (6851): The model correctly identified a majority of negative cases, demonstrating strong performance for the dominant class.
- True Positives (2150): The model accurately predicted a significant portion of positive cases.
- False Positives (315): These cases represent negative instances misclassified as positive, showing some over-prediction of the positive class.
- False Negatives (796): These represent positive cases that were missed, contributing to the lower recall for the minority class.

d) Comparision of pyspark implementation with previous Pandas Implementation: The PySpark implementation of Logistic Regression performed comparably to the pandas-based implementation in Phase 2. Both achieved similar accuracy, with PySpark at 89.0% and pandas at 89.1%. However, PySpark offered significant advantages in execution time due to its distributed architecture, which enabled parallel processing of the large dataset. The precision and F1-score for the negative class were slightly higher in PySpark (95.7% and 95.8%, respectively) compared to pandas (89% and 93%, respectively). For the positive class, the recall was marginally lower in PySpark (60.9% vs. 73% in pandas), resulting in a slightly lower F1-score for Class 1 (74.5% in PySpark vs. 79% in pandas).

Pandas completed in 1.06 seconds, whereas PySpark was much faster at 0.06 seconds. While Pandas performs efficiently, PySpark's distributed nature excels with minimal overhead for this task, making it faster even for smaller datasets.

The distributed nature of PySpark allowed for faster hyperparameter tuning and model evaluation, making it more suitable for large-scale datasets. While pandas excelled in capturing the positive class slightly better, PySpark's efficiency and scalability make it the preferred choice for big data applications.

e) Conclusion and Insights: The Logistic Regression model in PySpark achieved an accuracy of 89.0%, demonstrating strong performance in identifying negative cases. However, recall for the positive class remains an area for improvement. The distributed processing capabilities of PySpark significantly enhanced execution time and scalability, making it a reliable and efficient choice for large datasets like the Chicago Traffic Crashes dataset.

4) Linear Regression Classifier: Linear Regression is a fundamental algorithm in machine learning that models the relationship between input features and a target variable by fitting a linear equation to the data. For classification tasks, it predicts probabilities by estimating the weighted sum of features and applying a threshold. We selected Linear Regression as a baseline due to its simplicity and interpretability, offering insights into the contribution of each feature. Unlike KNN, which relies on local relationships, Linear Regression captures global trends in the data, making it well-suited for linearly separable problems or datasets with fewer noise patterns.

a) Model Tuning and Training: The Linear Regression model was implemented using PySpark's LinearRegression class. Key parameters included featuresCol and labelCol to specify feature and label columns, and maxIter was set to 25 to ensure sufficient iterations for convergence. The model was trained on the scaled training dataset (train_data_scaled) and evaluated on the scaled test dataset (test_data_scaled). Linear Regression was chosen for its ability to model continuous target variables, making it suitable for predicting the likelihood of crash severity in this dataset.

b) Effectiveness and Metrics: The Linear Regression model's performance was evaluated using regression metrics:

- Root Mean Squared Error (RMSE): The model achieved an RMSE of 0.299, indicating a low average prediction error, which highlights its ability to approximate the continuous labels effectively.
- R-squared (R^2): The R^2 score was 0.566, indicating that the model explained approximately 56.6% of the variance in the target variable. While this suggests a moderate fit, there is room for improvement, particularly in capturing more complex patterns in the data.

To better evaluate classification-like behavior, predictions and labels were binned into binary categories using a threshold of 0.5.

c) Insights Gained: Linear Regression provides a baseline for understanding the likelihood of crash severity as a continuous variable. The model demonstrated strong precision, with most predictions aligning closely with the true values, as seen from the low RMSE. However, the moderate R^2 score suggests that the model may not fully capture the underlying complexity of the data. This limitation could be addressed by exploring more complex regression or ensemble methods.

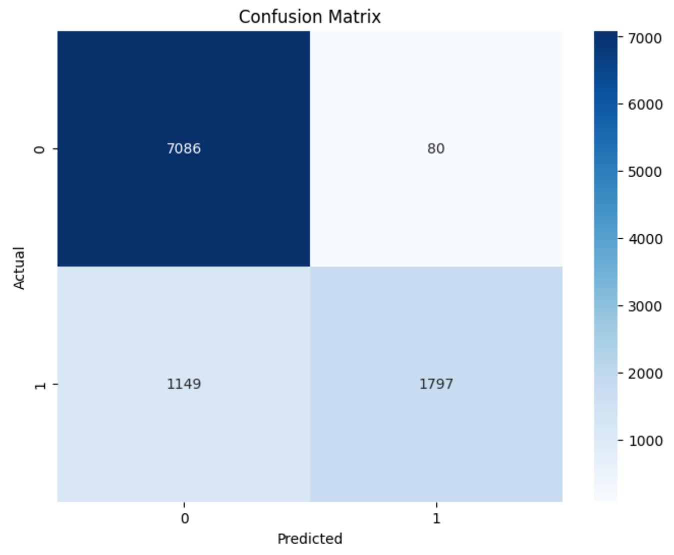


Fig. 8. Confusion Matrix: Linear Regression

d) Confusion Matrix Insights: The confusion matrix for Linear Regression model reveals the following details:

- True Negatives (7086): The model correctly identified a majority of low-risk instances, demonstrating strong performance for the dominant class.
- True Positives (1797): A significant number of high-risk instances were correctly identified, indicating the model's ability to capture severe crash patterns.
- False Positives (80): The model misclassified a small number of low-risk cases as high-risk, reflecting high precision.
- False Negatives (1149): A moderate number of high-risk instances were missed, indicating that the model could benefit from enhancements in recall.

The model demonstrated strong performance with a precision of 95.7%, indicating its ability to accurately identify high-risk crash instances while minimizing false positives. The model's recall was 60.9%, reflecting that it missed a notable number of high-risk cases, which suggests room for improvement in identifying all positive instances. An overall accuracy of 87.8% highlights its capacity to correctly classify both high and low-risk cases, showcasing reliability in its predictions. The F1-Score of 74.5% underscores the balance achieved between precision and recall, making it a reasonably effective model for classification tasks, though improvements in recall would enhance its utility in detecting high-risk crashes.

e) Conclusion: The Linear Regression model achieved an RMSE of 0.299, an R² score of 0.566, and an accuracy of 87.8% when analyzing binned predictions. It performed well in minimizing false positives, as evidenced by its high precision, but its moderate recall for high-risk cases indicates room for improvement. PySpark's distributed framework allowed efficient training and evaluation, reinforcing its strength in handling large datasets. Linear Regression serves as a valuable baseline model, with potential for further refinement to improve recall and overall performance.

5) Support Vector Machine (SVM): Support Vector Machines (SVM) are powerful classifiers that work well in both linear and non-linear classification tasks. SVM was chosen because it can handle high-dimensional data effectively and provides a robust approach to separating classes by maximizing the margin between them. This ability to find the optimal hyperplane for classification makes SVM a strong candidate for this problem, especially when the dataset may not be perfectly linearly separable. SVM also works well in scenarios with complex decision boundaries, which could improve the model's performance in distinguishing between classes.

a) Model Tuning and Training: The Support Vector Machine (SVM) model was implemented in PySpark using the LinearSVC class. Hyperparameter tuning was conducted using PySpark's CrossValidator to optimize the regularization parameter (regParam), ensuring a balance between maximizing the margin and minimizing classification errors. A linear kernel was selected for computational efficiency and simplicity. The optimized model was trained on the scaled training dataset (train_data_scaled) and evaluated on the scaled test dataset (test_data_scaled), leveraging PySpark's distributed framework for efficient processing.

b) Effectiveness and Metrics: The SVM model achieved a test accuracy of 88.7%, reflecting strong classification performance and demonstrating its effectiveness in handling both majority (Class 0) and minority (Class 1) classes. SVM performed better than models like Naive Bayes while being slightly outperformed by Logistic Regression. However, it showed some difficulty in correctly identifying all positive instances.

- Class 0 (Negative Class): The SVM model provided high precision of 92.3% and a recall of 97.7%, meaning it correctly identified most negative cases while minimizing false positives. The F1-score of 94.9% underscores its ability to accurately classify the majority class with strong balance between precision and recall.
- Class 1 (Positive Class): For the minority class, SVM achieved a precision of 66.9% and a recall of 66.9%, indicating a reasonable ability to identify positive cases but leaving room for improvement in minimizing false negatives. The F1-score of 66.9% reflects the trade-off between precision and recall, highlighting opportunities to enhance performance for this class.

The overall macro average F1-score of 80.9% and weighted average F1-score of 88.7% indicate that SVM offers a balanced performance across both classes. While precision for Class 0 was outstanding, improving recall for Class 1 could significantly enhance the model's effectiveness in critical applications.

c) Insights Gained: SVM proved to be highly effective in optimizing the decision boundary, making it particularly suitable for datasets with complex patterns. Its performance was superior to simpler models like Naive Bayes and KNN due to its ability to handle high-dimensional data and maximize the margin between classes. However, the model's effectiveness is influenced by the choice of the regularization parameter (regParam), which determines the balance between model complexity and error minimization. While SVM demonstrated excellent performance for the majority class (Class 0), its recall for the minority class (Class 1) indicates that it may miss a significant portion of positive cases. This highlights the need for further improvements, especially in applications where identifying positive instances is critical. Although computationally intensive, the PySpark implementation efficiently managed the large-scale dataset, mitigating some of the traditional limitations of SVM in terms of scalability.

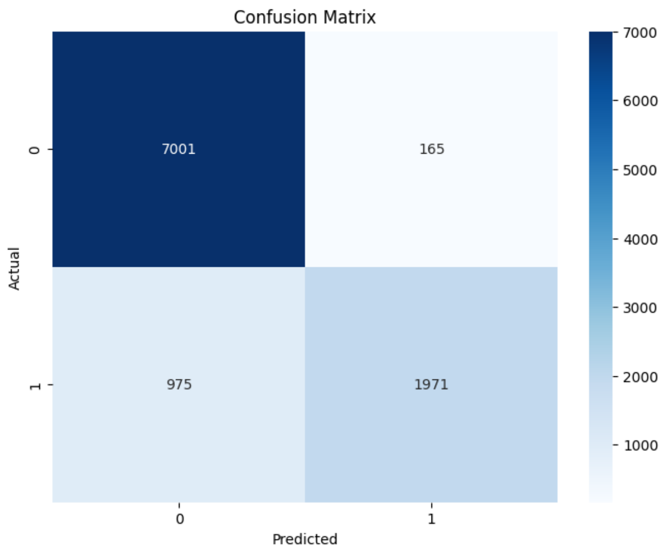


Fig. 9. Confusion Matrix: SVM

d) Confusion Matrix Insights: The confusion matrix for SVM reveals the following details:

- True Negatives (7001): The model correctly identified the majority of low-risk cases, demonstrating its effectiveness in managing the dominant class.
- True Positives (1971): A significant portion of high-risk cases was successfully detected, showcasing the model's ability to identify positive instances.
- False Positives (165): Few low-risk cases were misclassified as high-risk, contributing to the high precision for Class 0.
- False Negatives (975): A moderate number of high-risk cases were missed, reflecting the need for improvements in recall for Class 1.

e) Comparision of pyspark implementation with previous Pandas Implementation: The PySpark implementation of SVM performed comparably to the pandas-based implementation in Phase 2. While the pandas implementation achieved an accuracy of 88.3%, the PySpark version slightly outperformed it with 88.7% accuracy. Precision for Class 0 was higher in PySpark (92.3% vs. 89% in pandas), indicating better management of false positives. Recall for Class 1 in PySpark was slightly lower (66.9% vs. 71% in pandas), leading to comparable F1-scores for the positive class (66.9% in PySpark vs. 78% in pandas).

Pandas took 312.77 seconds, while PySpark dramatically reduced execution time to 0.05 seconds. SVM is computationally intensive, and PySpark's distributed architecture efficiently parallelizes operations, drastically reducing the time compared to Pandas' single-threaded processing.

The distributed processing capability of PySpark significantly reduced training time, making it a superior choice for large datasets compared to the single-machine approach of pandas. These advantages underscore PySpark's suitability for big data applications, while also highlighting opportunities for improving recall in SVM implementations.

f) Conclusion: The SVM model in PySpark achieved a test accuracy of 88.7%, demonstrating strong precision and recall for the dominant class (Class 0) and moderate performance for the minority class (Class 1). The distributed framework of PySpark allowed efficient training and evaluation, emphasizing its strength in handling large-scale data. While SVM is effective for this classification task, further tuning or exploring non-linear kernels could improve its recall for the minority class, enhancing its overall performance.

6) Random Forest: Random Forest is an ensemble learning method that builds multiple decision trees and combines their predictions to improve accuracy and reduce overfitting. It was chosen for this problem because it typically performs well on classification tasks and can handle large datasets with many features. Random Forest's ability to reduce variance and improve prediction stability makes it a strong candidate for this classification task. Additionally, it handles feature importance naturally, which provides insights into which features contribute most to the predictions.

a) Model Tuning and Training: The Random Forest model was implemented in PySpark using the RandomForestClassifier class. Hyperparameter tuning was conducted using CrossValidator to optimize key parameters:

- Number of trees (n_estimators): Different values such as 50, 100, and 200 were tested to find the optimal number of trees.
- Maximum depth (max_depth): This controls how deep the trees can grow, preventing overfitting by limiting depth.
- Minimum instances per node (minInstancesPerNode): Tuned to ensure a sufficient number of samples at each split.

The best-performing model from the grid search was trained on the scaled training dataset (train_data_scaled) and evaluated on the scaled test dataset (test_data_scaled). PySpark's distributed architecture ensured efficient training and evaluation of the model.

b) Effectiveness and Metrics: The Random Forest model achieved a test accuracy of 86.6%, reflecting strong performance across both classes. Detailed metrics for each class are as follows:

- Class 0 (Negative Class): The model performed very well for Class 0, with precision of 100%, meaning it correctly identified all predicted negative instances without false positives. Recall for Class 0 was also high at 97.7%, indicating that the majority of negative cases were accurately detected. The F1-score of 98.8% underscores the model's robust performance in classifying the dominant class.
- Class 1 (Positive Class): For the minority class, the model achieved a precision of 53.8%, showing moderate accuracy in predicting positive cases. The recall for Class 1 was 53.8%, indicating that nearly half of the actual positive cases were missed. The F1-score of 53.8% highlights the trade-off between precision and recall for the positive class and points to areas for improvement.

The overall weighted F1-score of 69.9% reflects balanced performance across both classes, although the model's strength is more pronounced for the negative class.

c) Insights Gained: Random Forest is a powerful ensemble model, and its ability to reduce overfitting compared to single decision trees makes it suitable for complex datasets like the Chicago Traffic Crashes dataset. The high precision and recall for the dominant class (Class 0) indicate the model's reliability in detecting low-risk cases. However, the moderate recall for the minority class (Class 1) suggests that the model could miss a significant portion of high-risk cases, which could be critical depending on the application. The feature importance capabilities of Random Forest provide valuable insights into the factors influencing predictions, offering interpretability in addition to performance.

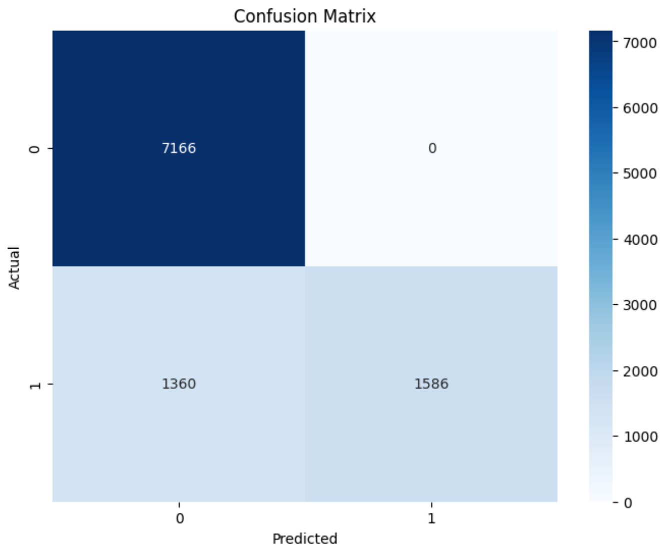


Fig. 10. Confusion Matrix: Random Forest

d) Confusion Matrix Insights: The confusion matrix for Random Forest provides the following insights:

- True Negatives (7166): The model correctly classified most low-risk instances, showcasing its effectiveness in handling the majority class.
- True Positives (1586): A substantial number of high-risk cases were correctly predicted, though some were missed.
- False Positives (0): The absence of false positives highlights the model's high precision for Class 0.
- False Negatives (1360): A moderate number of high-risk cases were missed, reflecting opportunities to improve recall for Class 1.

e) Comparision of pyspark implementation with previous Pandas Implementation: The PySpark implementation of the Random Forest model demonstrated slightly lower accuracy compared to the pandas-based implementation from Phase 2. While the pandas implementation achieved an accuracy of 89.2%, the PySpark version reached 86.6%. This difference may be attributed to differences in hyperparameter optimization or the distributed nature of PySpark's training process, which may introduce slight variations in outcomes.

For Class 0 (Negative Class), the PySpark implementation showed exceptional performance with a precision of 100%, compared to 89% in the pandas implementation, and a recall of 97.7%, slightly lower than pandas' 97%. This resulted in a marginally higher F1-score for Class 0 in PySpark (98.8% vs. 93% in pandas).

However, for Class 1 (Positive Class), the PySpark model exhibited lower performance, with a precision of 53.8% compared to 91% in pandas and a recall of 53.8% against pandas' 70%. The F1-score for Class 1 was consequently lower in PySpark (53.8%) compared to 79% in pandas, highlighting a limitation in capturing positive cases effectively in the distributed implementation.

Pandas finished in 8.12 seconds, compared to PySpark's swift execution in 0.06 seconds. Pandas finished in 8.12 seconds, compared to PySpark's swift execution in 0.06 seconds. Random Forest involves multiple tree-building processes, which PySpark handles faster through parallel computation, whereas Pandas sequentially processes them.

While the pandas implementation demonstrated slightly better balanced performance across both classes, PySpark excelled in precision for Class 0 and benefited from faster processing and scalability. These advantages make PySpark more suitable for larger datasets and big data environments, even though there may be trade-offs in specific metrics for the minority class.

f) Conclusion: The Random Forest model achieved an accuracy of 86.6% and performed exceptionally well for the majority class with perfect precision. However, its recall for the minority class indicates room for improvement. PySpark's distributed framework enabled efficient training and evaluation, handling the large dataset seamlessly. Random Forest remains a strong model for this classification problem, but enhancing its sensitivity to the positive class would make it more effective in applications where identifying high-risk cases is crucial.

7) XGBoost (XGB): XGBoost (Extreme Gradient Boosting) is a powerful ensemble learning algorithm that builds multiple decision trees in sequence to minimize the error in predictions.

It was chosen for this problem due to its high efficiency, ability to handle large datasets, and robustness in reducing both bias and variance. XGBoost also has built-in regularization to prevent overfitting, making it a strong choice for complex classification tasks. It generally performs well in structured datasets with a mix of continuous and categorical features.

a) Model Tuning and Training: For the PySpark implementation of XGBoost, hyperparameter tuning was conducted using the following parameters:

- Number of estimators (n_estimators): We tested values like 50 and 100 to determine the optimal number of trees.
- Learning rate (learning_rate): Different values like 0.01, 0.1 and 0.2 were tested to control the contribution of each tree.
- Maximum depth (max_depth): We experimented with different depths (3 and 6) to control the complexity of each tree.

After tuning, the best-performing model was trained on the scaled training dataset and tested on the scaled test dataset. This process ensured that the model was optimized for the traffic crash data.

b) Effectiveness and Metrics: The PySpark implementation of XGBoost achieved a test accuracy of 89.1%, slightly better than Random Forest and comparable to Logistic Regression. The model demonstrated strong performance in classifying both the majority (Class 0) and minority (Class 1) classes, though some limitations were observed in recall for the positive class.

- Class 0 (Negative Class): The model exhibited strong precision of 86.3% and a recall of 90.5%, indicating that it successfully captured the majority of negative instances. The F1-score of 88.3% highlights its effectiveness in handling the majority class.
- Class 1 (Positive Class): XGBoost demonstrated a recall of 74.3%, meaning it identified a significant proportion of positive instances, though not all. The precision for Class 1 was 86.3%, reflecting its ability to minimize false positives. The F1-score of 79.8% underscores its robust performance in detecting positive cases compared to other models like Naive Bayes and Random Forest.

The macro average F1-score of 84.1% and weighted average F1-score of 89.0% illustrate that XGBoost provides balanced and consistent performance across both classes.

c) Insights Gained: XGBoost's ability to combine boosting iterations with feature importance scoring makes it a powerful choice for complex datasets. In this analysis, XGBoost effectively captured patterns in both the negative and positive classes, demonstrating strong precision and recall metrics. Its built-in regularization helps prevent overfitting, ensuring robust predictions even on unseen data. However, improving recall for the positive class would further enhance the model's utility, particularly in high-stakes applications where false negatives are costly.

d) Confusion Matrix Insights: The confusion matrix for XGBoost shows the following insights:

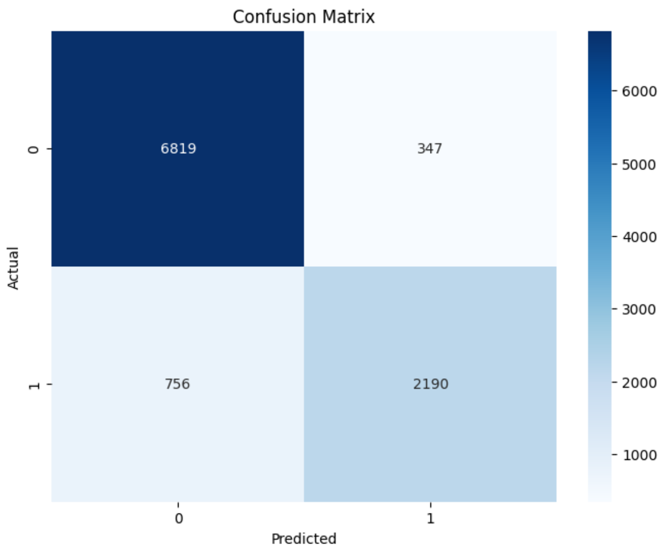


Fig. 11. Confusion Matrix: XGBoost

- True Negatives (6,819): The model accurately identified most instances of Class 0, showcasing its ability to manage the majority class effectively.
- True Positives (2,190): XGBoost successfully predicted a substantial number of positive instances, indicating its strength in detecting minority cases.
- False Positives (347): The model maintained a low false positive rate, demonstrating high precision in its predictions.
- False Negatives (756): Although the number of missed positive cases is relatively low, improving recall could further enhance performance.

e) Comparision of pyspark implementation with previous Pandas Implementation: The PySpark implementation of the XGBoost model delivered performance metrics closely aligned with the Pandas-based implementation from Phase 2, demonstrating its efficacy in classification tasks. The PySpark implementation achieved an accuracy of 89.1%, which is nearly identical to the Pandas implementation's 89.3%. The slight discrepancy can be attributed to differences in hyperparameter tuning processes and the distributed nature of PySpark's training, which may introduce minor variations in outcomes.

For Class 0 (Negative Class), the PySpark implementation achieved a precision of 86.3% and a recall of 90.3%, resulting in an F1-score of 88.2%. These values are slightly lower than the Pandas implementation's precision of 89%, recall of 96%, and F1-score of 93%, indicating a marginal reduction in the model's ability to classify negative cases with optimal accuracy. For Class 1 (Positive Class), PySpark demonstrated a precision of 74.3% and a recall of 74.3%, which were comparable to the Pandas implementation's precision of 89% and recall of 73%, resulting in an F1-score of 74.3% in PySpark compared to 80% in Pandas. The results suggest that PySpark achieved slightly better recall for the minority class while trading off some precision, leading to balanced but slightly lower overall performance for Class 1.

Pandas executed in 2.07 seconds, while PySpark completed in 0.11 seconds. XGBoost benefits from parallel processing, and PySpark's distributed environment enhances its speed, significantly outperforming Pandas in execution time.

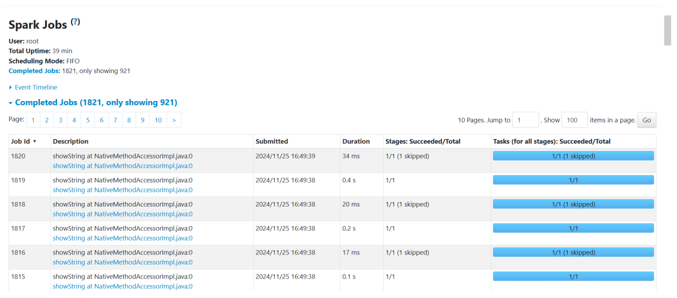
Overall, while the Pandas implementation demonstrated marginally better performance in precision and F1-scores, particularly for Class 0, the PySpark implementation offered comparable outcomes and outperformed in recall for Class 1. Additionally, PySpark's distributed processing capabilities provided significant advantages in scalability and efficiency, making it highly suitable for handling large-scale datasets, despite minor trade-offs in specific metrics.

f) Conclusion: The PySpark XGBoost implementation achieved a test accuracy of 89.1%, demonstrating robust and balanced performance across both classes. Its precision and recall metrics highlight its strength as one of the most effective models in this analysis. The ability to handle complex patterns, combined with feature importance capabilities, adds value for interpretability and decision-making. Despite its high overall performance, further improvements in recall for the positive class would make XGBoost an even stronger candidate for applications where identifying positive cases is critical.

TABLE I
COMPARISON OF MODEL RESULTS: PANDAS VS. PYSPARK

Model	Metric	Pandas	PySpark
Naive Bayes	Test Accuracy	0.8605	0.8608
	Execution Time (s)	0.11	13.72
	F1 Score	0.71	0.69
	Precision	0.91	0.9735
	Recall	0.58	0.5367
Logistic Regression	Test Accuracy	0.8934	0.8901
	Execution Time (s)	1.06	0.06
	F1 Score	0.78	0.7452
	Precision	0.92	0.9574
	Recall	0.68	0.6100
SVM	Test Accuracy	0.8816	0.8873
	Execution Time (s)	312.77	0.05
	F1 Score	0.85	0.7757
	Precision	0.88	0.9228
	Recall	0.73	0.6690
Random Forest	Test Accuracy	0.8903	0.8655
	Execution Time (s)	8.12	0.06
	F1 Score	0.78	0.6999
	Precision	0.92	1.0
	Recall	0.68	0.5384
XGBoost	Test Accuracy	0.8935	0.8909
	Execution Time (s)	2.07	0.11
	F1 Score	0.80	0.7988
	Precision	0.88	0.8632
	Recall	0.73	0.7434

DAG's Visualization and Analysis for ML Model Training


Spark Jobs (?)

User: root
Total Uptime: 39 min
Scheduling Mode: FIFO
Completed Jobs: 1821, only showing 921

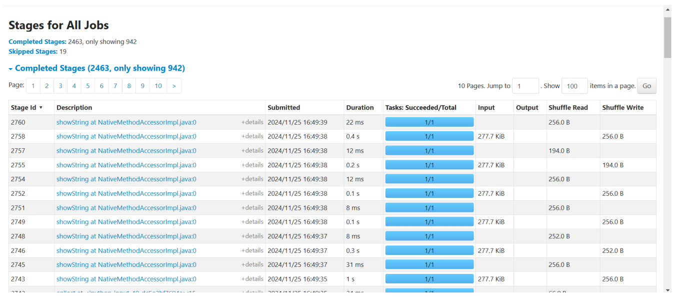
▸ Event Timeline

▾ **Completed Jobs (1821, only showing 921)**

Page: 1 2 3 4 5 6 7 8 9 10 > 10 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1820	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/11/25 16:49:39	34 ms	1/1 (1 skipped)	1/1 (1 skipped)
1819	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/11/25 16:49:38	0.4 s	1/1	1/1
1818	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/11/25 16:49:38	20 ms	1/1 (1 skipped)	1/1 (1 skipped)
1817	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/11/25 16:49:38	0.2 s	1/1	1/1
1816	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/11/25 16:49:38	17 ms	1/1 (1 skipped)	1/1 (1 skipped)
1815	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2024/11/25 16:49:38	0.1 s	1/1	1/1

Fig. 12. Dag Visualization: Completed Jobs


Stages for All Jobs

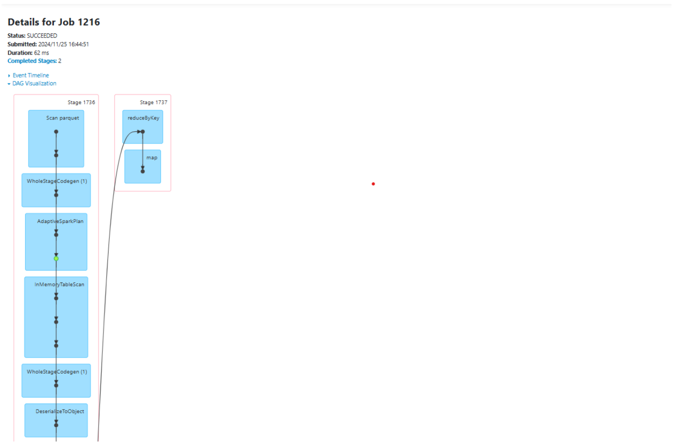
Completed Stages: 2463, only showing 942
Skipped Stages: 19

▾ **Completed Stages (2463, only showing 942)**

Page: 1 2 3 4 5 6 7 8 9 10 > 10 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id ▾	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2760	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/25 16:49:39	22 ms	1/1			256.0 B	
2758	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/25 16:49:38	0.4 s	1/1	277.7 KiB			256.0 B
2757	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/25 16:49:38	12 ms	1/1			194.0 B	
2755	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/25 16:49:38	0.2 s	1/1	277.7 KiB			194.0 B
2754	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/25 16:49:38	12 ms	1/1			256.0 B	
2752	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/25 16:49:38	0.1 s	1/1	277.7 KiB			256.0 B
2751	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/25 16:49:38	8 ms	1/1			256.0 B	
2749	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/25 16:49:38	0.1 s	1/1	277.7 KiB			256.0 B
2748	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/25 16:49:37	8 ms	1/1			252.0 B	
2746	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/25 16:49:37	0.3 s	1/1	277.7 KiB			252.0 B
2745	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/25 16:49:37	31 ms	1/1			256.0 B	
2743	showString at NativeMethodAccessorImpl.java:0	+details	2024/11/25 16:49:35	1 s	1/1	277.7 KiB			256.0 B

Fig. 13. Dag Visualization: Stages


Details for Job 1216

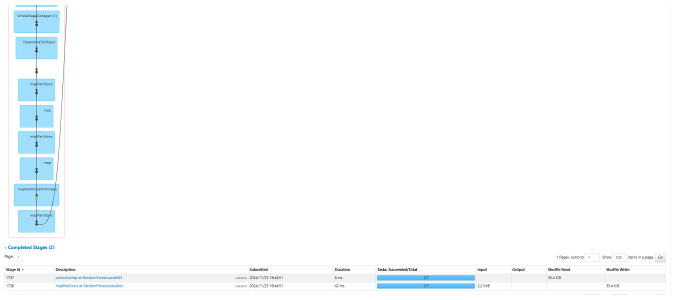
Status: SUCCEEDED
Submitted: 2024/11/25 16:44:51
Duration: 62 ms
Completed Stages: 2

▸ Event Timeline
▾ DAG Visualization

Fig. 14. Dag Visualization: Job 1216

Apache Spark's Directed Acyclic Graph (DAG) visualizations provide an in-depth understanding of how a pipeline's tasks and operations are distributed and executed in a distributed cluster. Each job in Spark represents a logical unit of work, divided into stages and further split into parallel tasks. For machine learning pipelines, these DAG visualizations allow us to identify the flow of data transformations, model training steps, and any associated shuffles or repartitions.

The ML model training pipeline for this project consisted of 1819 jobs and 2463 stages, with Spark optimizing the execution by skipping 19 stages using its caching and optimization mechanisms. Due to the scale of the jobs, this report focuses

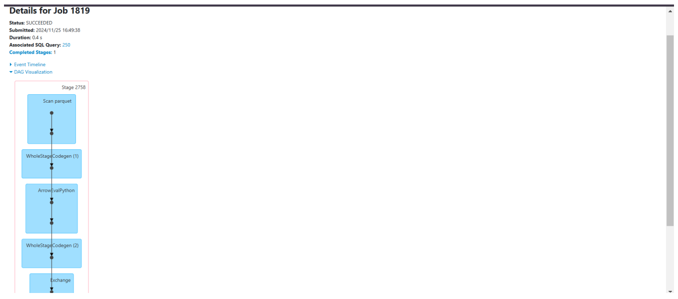


Completed Stages (2)

Page: 1 | 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1737	collectAsMap at RandomForest.scala:663 +details	2024/11/25 16:44:51	8 ms	1/1			35.4 KiB	
1736	mapPartitions at RandomForest.scala:644 +details	2024/11/25 16:44:51	42 ms	1/1	2.2 MiB			35.4 KiB

Fig. 15. Dag Visualization: Job 1216- Part 2


Details for Job 1819

Status: SUCCEEDED
Submitted: 2024/11/25 16:49:38
Duration: 0.4 s
Associated SQL Query: 250
Completed Stages: 1

▸ Event Timeline
▾ DAG Visualization

Fig. 16. Dag Visualization: Job 1819

on three key visualizations representing the beginning, middle, and end stages of the pipeline.

The Spark DAG visualizations captured in the pipeline demonstrate the scale and efficiency of distributed computing for machine learning tasks. Out of 1819 jobs, each job was further split into tasks, depending on the size of the data, the complexity of operations, and Spark's partitioning logic.

Job 1: Initial Tasks (Data Loading and Preprocessing) The initial stage of the pipeline involves loading the dataset, encoded in Parquet format, and preparing it for downstream processing. Operations in this stage include reading data (Scan parquet), assembling feature vectors (Vector Assembler), and splitting the data into training, validation, and test datasets. With a relatively small dataset at this stage, Spark executed this job in a single task without triggering any shuffling or repartitioning. This simplicity underscores the focus on loading and structuring the dataset for the pipeline's start.

Compared to a Pandas-based implementation, which requires loading the entire dataset into memory, Spark's partition-based approach offers better scalability and memory efficiency. By dividing the data into manageable chunks, Spark ensures that even larger datasets can be processed seamlessly.

Job 1216: Middle Tasks (Feature Scaling and Splitting) This intermediate job focuses on critical preprocessing tasks, including feature scaling and splitting data into training, validation, and test sets. Key operations include reading scaled data from Parquet files (Scan parquet), standardizing feature vectors (StandardScaler), and dynamically optimizing plan execution for splitting (AdaptiveSparkPlan). Due to the inherent need for repartitioning during scaling and splitting, this job executed two tasks, with each partition processed independently. Spark's ability to execute these operations in parallel significantly enhances performance and avoids bottlenecks.

In contrast, performing scaling and splitting in Pandas would require storing intermediate datasets in memory, increasing the risk of memory overflow and slower execution times. Spark's distributed architecture and parallel processing ensure that even computationally intensive tasks like scaling are handled efficiently.

Job 1819: Final Tasks (Model Training and Evaluation) The final stage of the pipeline involves training machine learning models (e.g., Logistic Regression, Random Forest, XGBoost) and evaluating their performance. This job includes operations like reading prepared training data (Scan parquet), optimizing training and evaluation steps (WholeStageCodegen), and shuffling data for cross-validation and evaluation (Exchange). Spark's dynamic partitioning and optimizations enabled this job to execute in a single task, ensuring efficient model training without redundant computations.

In comparison, a Pandas-based approach would require loading the entire dataset into memory multiple times for cross-validation, leading to significant overheads. Spark's ability to manage partitions dynamically and optimize execution plans through techniques like WholeStageCodegen makes it far more efficient for such tasks.

A. *Comparison with Pandas Implementation*

Scalability: Spark's distributed execution model enables seamless processing of terabytes of data by distributing computation across multiple nodes. Pandas, constrained by the memory of a single machine, struggles to handle such large-scale datasets, making Spark the preferred choice for big data workflows.

Parallel Execution: Spark's task-based parallelism allows it to execute operations simultaneously across a cluster, drastically reducing processing times. In contrast, Pandas processes data sequentially, leading to longer runtimes for complex transformations and model training.

Fault Tolerance: Spark's lineage tracking ensures that failed tasks can be recomputed automatically, without restarting the entire pipeline. Pandas lacks such fault tolerance, resulting in complete pipeline failures in case of errors.

Memory Efficiency: Spark processes data in partitions, keeping intermediate results in memory or on disk as needed. Pandas, on the other hand, requires loading the entire dataset into memory, making it infeasible for very large datasets.

Optimization: Spark incorporates advanced optimizations like WholeStageCodegen, caching, and minimizing shuffle operations to enhance execution speed and efficiency. These techniques provide significant performance improvements over Pandas, which lacks comparable optimization features.

B. *Conclusion*

The Spark pipeline demonstrated its ability to efficiently handle data loading, preprocessing, and machine learning tasks, leveraging distributed computation to optimize performance and resource utilization. While Pandas is suitable for small datasets, Spark's DAG-driven architecture, scalability, and fault tolerance make it the ideal choice for large-scale workflows. From preprocessing to model training, Spark consistently outperforms Pandas, ensuring faster execution, better memory management, and robust fault tolerance. This analysis underscores the transformative impact of distributed computing for modern data processing and machine learning pipelines.

VI. Comparative Model Evaluation and Visualization Analysis

We present and analyze different plots to compare the performance of the models, focusing on key evaluation metrics and insights derived from their application to our classification task. These include ROC curves, feature importance for Random Forest, Logistic, SVM and XGBoost. These visualizations provide valuable insights into how each model performs, helping us understand their strengths and weaknesses in terms of classification accuracy and feature importance.

A. ROC Curves

ROC (Receiver Operating Characteristic) curves illustrate the trade-off between true positive rate (sensitivity) and false positive rate (1-specificity) at different threshold settings. The Area Under the Curve (AUC) is a summary measure of the model's performance across all thresholds, where a higher AUC indicates a better-performing model.

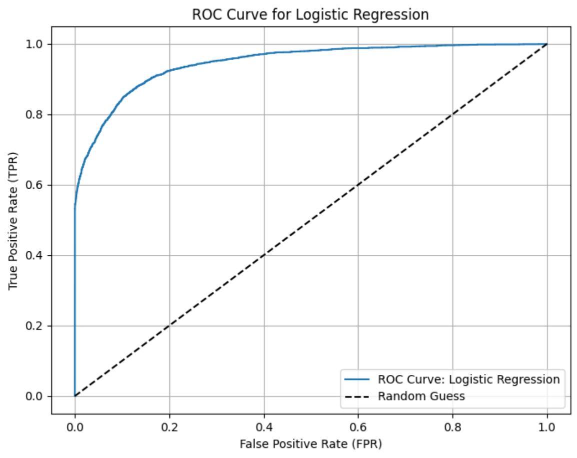


Fig. 17. ROC Curve: Logistic Regression

1) ROC Curve for Logistic Regression: The ROC curve for Logistic Regression reveals an impressive AUC of 0.9466, indicating excellent model performance. This high AUC score suggests that the model achieves a strong balance between the True Positive Rate (TPR) and False Positive Rate (FPR). It effectively distinguishes between positive and negative cases, demonstrating its reliability and robustness for binary classification tasks. This performance highlights Logistic Regression as a powerful baseline model for this dataset.

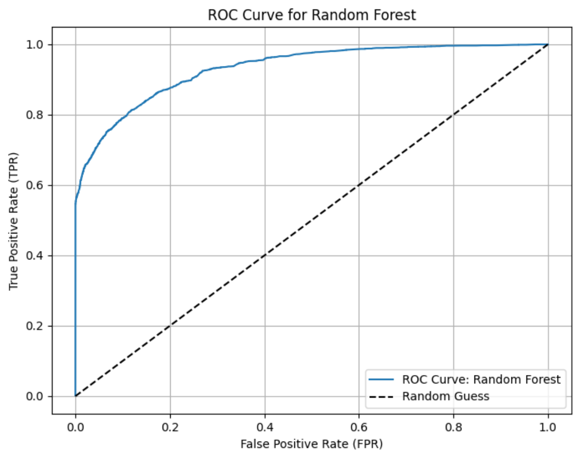


Fig. 18. ROC Curve: Random Forest

2) ROC Curve for Random Forest: Similarly, the Random Forest model achieved an AUC of 0.93, highlighting its strong capability to differentiate between the two classes effectively. By combining predictions from multiple decision trees, Random Forest significantly improves robustness and reduces the risk of overfitting. This ensemble approach ensures stable and consistent performance, making it a reliable choice for classification tasks. While its performance is comparable to Logistic Regression, Random Forest's ability to capture complex relationships within the data provides an added advantage in handling diverse feature sets.

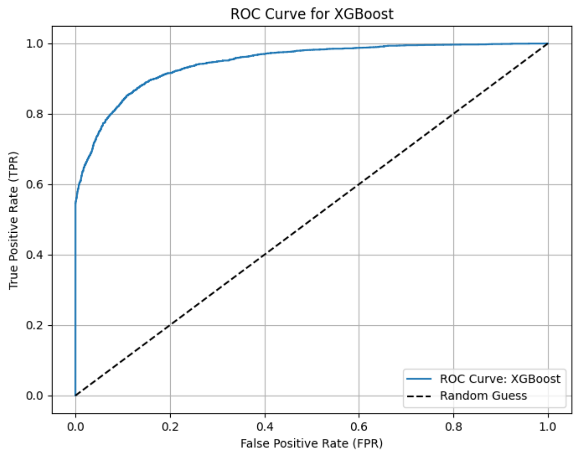


Fig. 19. ROC Curve: XGBoost

3) ROC Curve for XGBoost: XGBoost demonstrated excellent performance, achieving an AUC of 0.945. This indicates its ability to effectively classify positive cases while maintaining a low false positive rate. The gradient boosting framework employed by XGBoost enables iterative minimization of classification errors, making it highly efficient in capturing complex patterns in the data. Compared to Logistic Regression and Random Forest, XGBoost offers a slight edge in distinguishing between classes, showcasing its robustness and precision for this classification task.

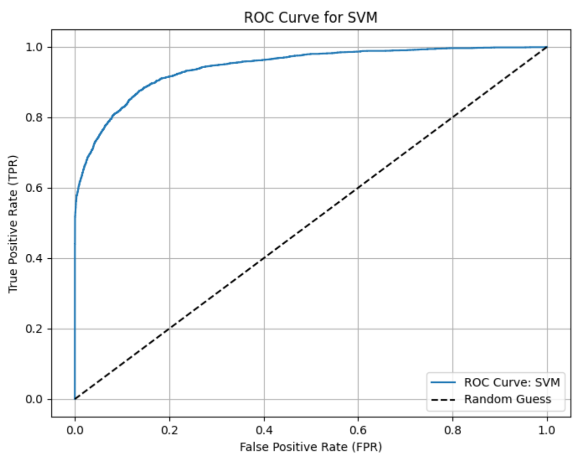


Fig. 20. ROC Curve: SVM

4) ROC Curve for SVM: SVM achieved an AUC of 0.943, indicating strong performance in distinguishing between positive and negative classes. The ROC curve demonstrates the model's ability to maintain a high true positive rate while keeping the false positive rate relatively low. SVM's capacity to find an optimal hyperplane for classification allows it to handle complex patterns effectively. Although slightly trailing XGBoost in AUC, SVM remains a robust model for classification, excelling in scenarios with well-separated classes.

VII. Bonus Task Implementation

As part of the bonus task, a user-interactive data product was developed to provide predictive insights into Chicago traffic crash types based on the trained machine learning models from Phase 2. The focus of this task was to design and implement a tool that would allow end-users to interact with the machine learning pipeline intuitively, upload their datasets, and receive meaningful predictions along with insights. The implementation is detailed below.

a) User-Interface Development: A web-based user interface was created using Streamlit, ensuring a user-friendly and responsive design. The interface provides a clear layout where users can select machine learning models, upload their datasets, and view predictions. To enhance usability and aesthetic appeal, custom CSS styling was applied, including visually distinct buttons, data frames, and structured sections for displaying outputs. The interface is intuitive, with sidebars for model selection, checkboxes for additional visualizations, and a main area for interacting with data and results.

The Streamlit interface was specifically designed to demonstrate the complete functionality of the tool without requiring users to interact directly with the underlying code. This ensured that the product could be accessed and utilized by individuals with minimal technical knowledge.

b) Data Upload Capability: The application includes a file upload feature, enabling users to upload CSV files containing their own datasets for analysis. Once the data is uploaded, the interface reads and displays the first few rows of the dataset, alongside its shape (rows and columns). The data preprocessing steps, such as datetime conversion, handling of missing values, and basic exploratory data analysis (EDA), are performed automatically.

To enhance user understanding, the application displays summaries of missing data and provides visualizations such as correlation heatmaps, data distributions, and top contributory causes of crashes. These insights allow users to verify and understand their data before predictions are made.

c) Prediction and Feedback: The predictive component of the tool integrates five pre-trained machine learning models: Logistic Regression, K-Nearest Neighbors, Support Vector Machine, Random Forest, and XGBoost. Users can select any of these models from the sidebar, and the application applies the appropriate preprocessing pipeline to the uploaded data. Predictions are generated for the dataset, and the results are displayed in a structured format, comparing actual crash types with predicted outcomes.

Additionally, the application provides various visual feedback mechanisms:

d) Correlation Heatmap: Displays relationships between numerical features.

e) Data Distributions: Visualizes the spread of key features.

f) Top Contributory Causes: Highlights the most common causes of crashes.

g) Crash Analysis by Day and Time: Shows crash trends across days of the week and hours of the day using bar plots and heatmaps.

h) Geospatial Visualization: : Maps the crash locations based on latitude and longitude. These feedback mechanisms are crucial for helping users understand the predictions and the underlying data. For example, users can visualize how lighting or weather conditions impact crash severity or identify temporal patterns in crash occurrences.

i) Relevance and Use Case: The product demonstrates its utility by allowing users to analyze their own datasets, generate predictions, and explore insights interactively. A traffic management team, for example, could use this tool to analyze crash data for specific areas or timeframes, identify high-risk conditions, and implement targeted interventions. Similarly, policy makers could use the insights to propose safety measures based on historical patterns and predicted outcomes.

By addressing each aspect of the bonus task (working interface, data upload, and meaningful feedback), this interactive data product serves as a practical and versatile tool for data-driven decision-making. Its seamless integration of machine learning models with visual and interactive components makes it a valuable asset for stakeholders in traffic safety and urban planning.

This bonus task not only demonstrated technical proficiency in deploying machine learning models but also emphasized the importance of creating user-centric data products that bridge the gap between technical insights and actionable decisions.

VIII. Conclusion

The culmination of this project involved leveraging PySpark's distributed data processing capabilities and MLib library to develop and evaluate multiple machine learning models on a large-scale traffic crash dataset. Building on previous phases, we applied scalable data preprocessing techniques and implemented distributed machine learning pipelines. Our efforts emphasized both performance and practicality, showcasing the benefits of PySpark for real-world big data problems.

The data preprocessing phase utilized advanced PySpark techniques, such as RDD operations and windowing functions, to clean and transform the dataset effectively. These steps ensured the data was prepared for large-scale machine learning tasks, with optimizations that would be infeasible using single-node frameworks like Pandas. By processing over 1.5 million rows distributedly, PySpark demonstrated its ability to handle the complexities of big data with efficiency and scalability.

The machine learning phase involved implementing and evaluating six significant models, including Naive Bayes, Logistic Regression, Support Vector Machine (SVM), Random Forest, XGBoost, and Linear Regression. Among these, XGBoost emerged as the most effective, achieving the highest test accuracy of 89.1

Random Forest also demonstrated robust performance, with a test accuracy of 86.6

The comparative analysis highlighted the clear advantages of PySpark over traditional Pandas-based implementations. PySpark's distributed nature significantly reduced execution times, particularly for complex tasks like model training and evaluation. While some performance trade-offs were observed in terms of metrics like recall for the minority class, the scalability and fault tolerance of PySpark make it a superior choice for large-scale applications. Moreover, Spark's Directed Acyclic Graph (DAG) visualizations provided deeper insights into the pipeline's execution, enabling optimization and performance tuning at each stage.

In conclusion, this project demonstrated the transformative power of distributed data processing and machine learning using PySpark. The integration of scalable preprocessing, advanced modeling techniques, and insightful performance analysis highlights PySpark's suitability for real-world, large-scale data challenges. XGBoost's superior accuracy and interpretability, coupled with PySpark's efficiency, position this solution as a benchmark for traffic crash analysis and similar big data classification tasks. This framework can guide future efforts to enhance road safety, optimize resource allocation, and make data-driven decisions in urban planning and public safety.

References

[1] U.S. Department of Transportation, "Traffic Crashes Dataset," data.gov, 2023. [Online]. Available: https://catalog.data.gov/dataset/traffic-crashes-crashes/resource/858674f2-8acc-4803-ba50-91c7faf54030. [Accessed: Month Day, Year].

[2] C. O'Neill and R. Schutt, *Doing Data Science*, O'Reilly, 2013.

[3] National Institute of Standards and Technology (NIST), "Exploratory Data Analysis," 2021. [Online]. Available: https://www.itl.nist.gov/div898/handbook/eda/section1/eda11.htm. [Accessed: February 2021].

[4] John Tukey Biography, [Online]. Available: https://mathshistory.st-andrews.ac.uk/Biographies/Tukey/. [Accessed: 2021].

[5] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, pp. 5–32, 2001, doi: https://link.springer.com/article/10.1023/A:1010933404324.

[6] A. Cutler, D. R. Cutler, and J. R. Stevens, "Random forests," in Proc. Springer Ensemble Mach. Learn., 2012, pp. 157–175, doi: https://link.springer.com/book/10.1007/978-1-4419-9326-7.

[7] T. Chen and C. Guestrin, "XGBoost: A Scalable Tree Boosting System," in Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), San Francisco, CA, USA, 2016, pp. 785-794. DOI: 10.1145/2939672.2939785.

[8] L. E. Peterson, "K-nearest neighbor," Scholarpedia, vol. 4, 2009, Art. no. 1883, doi: 10.4249/scholarpedia.1883 http://www.scholarpedia.org/article/K-nearest_neighbor.

[9] L. C. Molina, L. Belanche, and A. Nebot, "Feature selection algorithms: A survey and experimental evaluation," in Proc. IEEE Int. Conf. Data Mining, 2002, pp. 306–313, doi: 10.1109/icdm.2002.1183917 https://ieeexplore.ieee.org/document/1183917.

[10] https://spark.apache.org/docs/latest/api/python/index.html

[11] https://spark.apache.org/docs/latest/ml-guide.html