Income Prediction

Preprocessing:

1. Handling Missing values:

This code Replaced "?" in all string columns with None, since that is what Spark understands as NULL. Then we dropped rows that have NULL values in important columns of the data, including age, workclass, education, occupation, and income. The rationale behind this operation is to ensure those important columns are complete for an analysis or some machine learning model. This is the step that, in effect, increases reliability and integrity of a dataset while only retaining rows with informative values.

```
from pyspark.sql.functions import when, col

# Replace "?" with None for all string columns
df = df.replace("?", None)
```

```
critical_columns = ['age', 'workclass', 'education', 'occupation', 'income']
df = df.dropna(subset=critical_columns)
```

2. Removing Duplicate values:

Then, it removes the duplicate rows in the dataset using the dropDuplicates() method. The rationale behind this step is to avoid redundancy and maintain data integrity. Consequently, this boosts the quality of the data, reduces the storage burden, and averts the impact of repeated information on analysis skew or model training.

```
df = df.dropDuplicates()
```

3. Filtering by Age Range:

The code filters in rows where the values in the age column fall in the range between 18 and 90 inclusive. This step is done to reduce the dataset into relevant age groups, excluding those that are outliers or irrelevant for data analysis. The implication of this is that it ensures the dataset narrows down to the target demographic, hence enhancing the accuracy and relevance of subsequent analyses or predictions.

```
df = df.filter((col('age') >= 18) & (col('age') <= 90)
```

4. Renaming Columns:

This code replaces all characters in column names with underscores (_) to avoid potential issues with dotcontaining column names. The goal of this is to increase compatibility with Spark operations that could otherwise misinterpret dots in column names. The impact this has is that it makes working with columns easier, circumventing errors while processing data.

```
df = df.toDF(*[c.replace('.', '_') for c in df.columns])
```

5. Creating an Age category Column:

A custom function age_bin defines the age groups into three categories: "Young" below 30, "Middle-aged" between 30–59, and "Senior" for 60 and above. This function is applied as a User Defined Function, creating a new column named age_category. The purpose of this step is to segment individuals into meaningful age categories for easier analysis or feature creation. The result is a dataset from which more interpretive capability can be derived and patterns regarding age are better discoverable.

```
def age_bin(age):
    if age < 30:
        return "Young"
    elif age < 60:
        return "Middle-aged"
    else:
        return "Senior"

age_bin_udf = udf(age_bin, StringType())
df = df.withColumn("age_category", age_bin_udf(col("age")))
df.show(10)
```

6. Encoding categorical values:

The StringIndexer encodes categorical columns into numeric indices: workclass, education, marital_status, occupation, relationship, race, sex, native_country, and income. It converts string categories into numerical representations and creates new columns with a suffix _index. It is done to prepare categorical data for machine learning algorithms that usually expect numeric features as input. The effect of this is that the dataset would be ready for machine learning pipelines while still retaining information from the categorical variables.

Ujwala Kopparthi

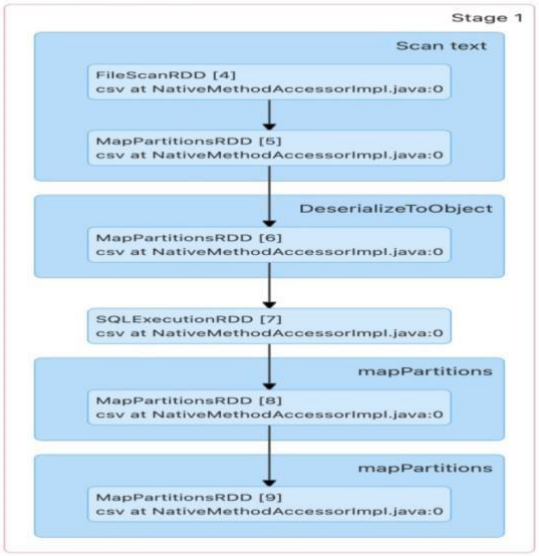
```
from pyspark.ml.feature import StringIndexer
from pyspark.ml import Pipeline

categorical_columns = ['workclass', 'education', 'marital_status', 'occupation', 'relationship', 'race', 'sex', 'native_country', 'income']
for col in categorical_columns:
    print(f"{col}: {df.schema[col].dataType}")
indexers = [StringIndexer(inputCol=col, outputCol=col+"_index", handleInvalid="skip").fit(df) for col in categorical_columns]
pipeline = Pipeline(stages=indexers)
df = pipeline.fit(df).transform(df)
df = df.drop(*categorical_columns)
df.show(5)
```

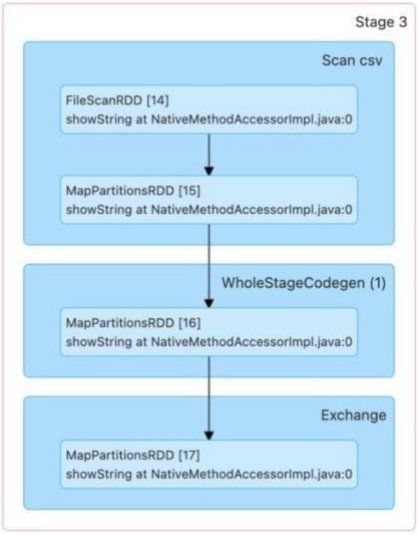
7. Dropping original categorical columns:

After encoding the categorical columns, it drops the original string columns from the dataset. The intention of this step is that, going forward, only the numeric representations should be used either in data processing or modeling. This would reduce the dimensionality of a dataset and remove any redundancy, hence simplifying further analysis or machine learning training.

```
df = df.drop(*categorical_columns)
```

DAG Analysis Preprocessing:





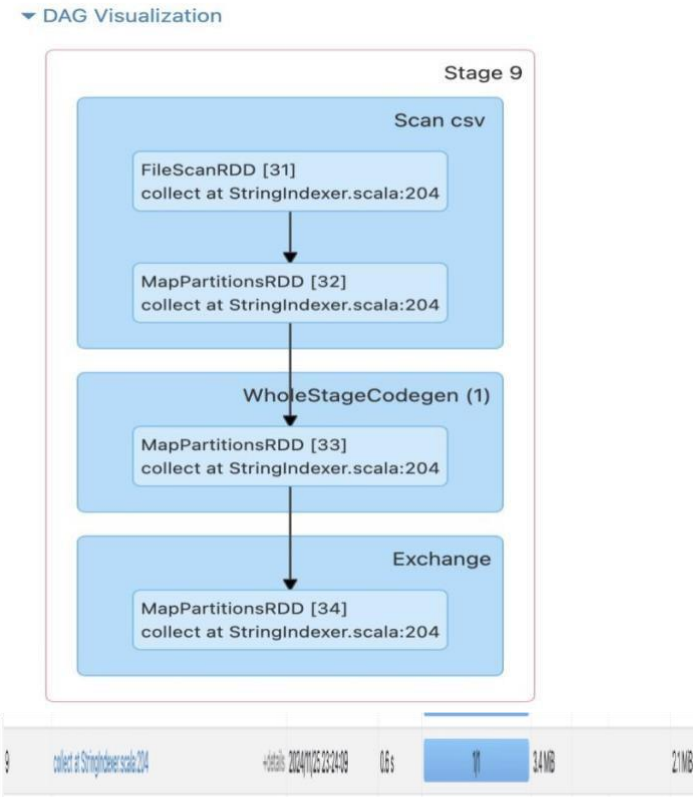
The DAG visualizations that Spark provides make for an illuminating breakdown of the distributed stages involved in the preprocessing of the dataset. First, it reads a CSV file and processes it as an RDD in Stage 0. The data is scanned, and then mapped into partitions for deserialization. Each step corresponds to transforming the raw input file into a distributed DataFrame for further operations.

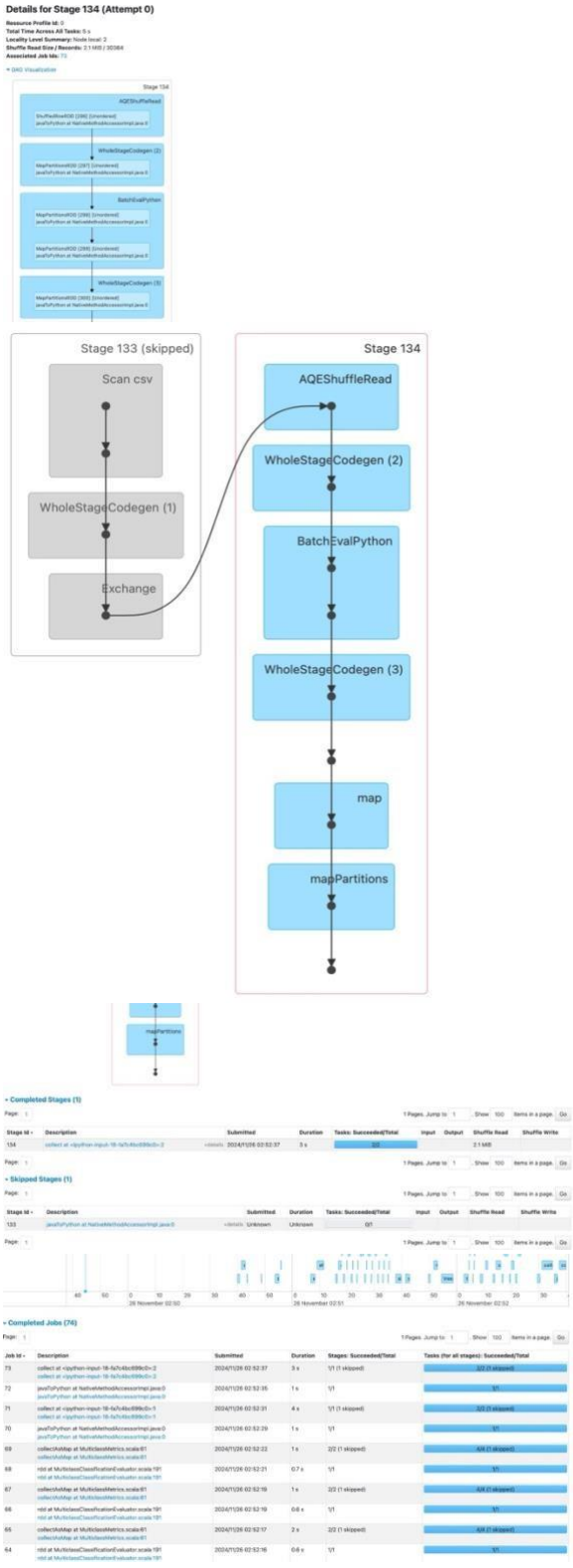
During Stage 1, after processing missing and duplicate values, filtering the rows on conditions such as age, the data undergoes further transformation. These transformations are expressed in tasks such as mapping the partitions to execute filtering and dropping null values in parallel by utilizing Spark's distributed framework.

Stage 3 encapsulates adding the new derived column, ‘age_category’, by using the UDF

‘AgeSplit’. It carries out the computation of splitting ages into categories such as "Young," "Middle-aged," and "Senior." This operation requires partition-wise application of the UDF and updating of the Data Frame structure. Spark handles these steps efficiently in parallel across its distributed environment.

The visualization emphasizes stepwise flow, where tasks like reading a file, filtering, and transformation are divided into stages. This segmentation guarantees efficiency in distributed processing because each stage optimizes the operations performed on its respective partition. One can go ahead and identify bottlenecks or opportunities for further optimization by looking at it.



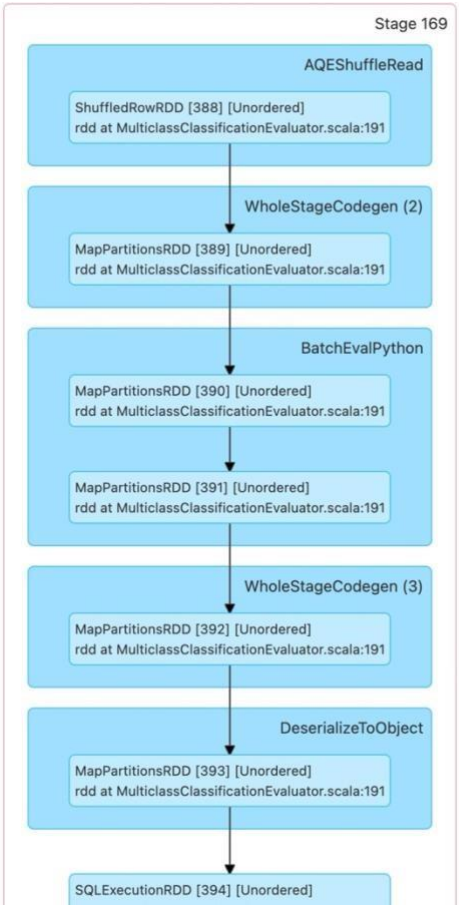


In this regard, DAG visualizations show the different distributed stages of categorical columns' preprocessing using 'StringIndexer' and 'Pipeline' in Spark. In the first stage (Stage 9), the input dataset is loaded using 'FileScanRDD', which means that da ta is partitioned and distributed among worker nodes. This stage uses Spark's optimizations, such as 'WholeStageCodegen', to put multiple transformations into a single stage for efficient execution, with data shuffling handled by the 'Exchange' operation i n order to correctly partition the data for the next operations. The next step, which would be Stage 11, consists of the heart of the transformation, a 'StringIndexer', mapping categorical values of one or more columns to numerical indices, and is ready fo r distribution between executors with whatever shuffling or collection is needed. The indexed columns are applied to the dataset, and the original categorical columns are dropped in the final transformation steps of the pipeline in Stage 14. Optimizations, such as parallelism and code generation, from Spark can be seen throughout this DAG. The visualization emphasizes how Spark minimizes execution overhead and scales the preprocessing by means of its distributed architecture.

Logistic Regression:

Visualizations of DAGs are one of the most effective ways to show how data and tasks flow in a distributed environment. In your LR pipeline implementation, several stages were executed: VectorAssembler, LogisticRegression, and the overall pipeline transformation. As is evident from the DAG

visualizations, Spark divides these tasks into separate stages, Spark is capable of managing resources effectively. with certain operations being mapped across nodes in a cluster. Stage 133 handles the initial data read and preprocessing, such as vectorization and indexing. This stage includes tasks like Scan csv for reading the data and WholeStageCodegen for optimized computation. It also shows data exchange points, highlighting shuffle operations required for repartitioning.



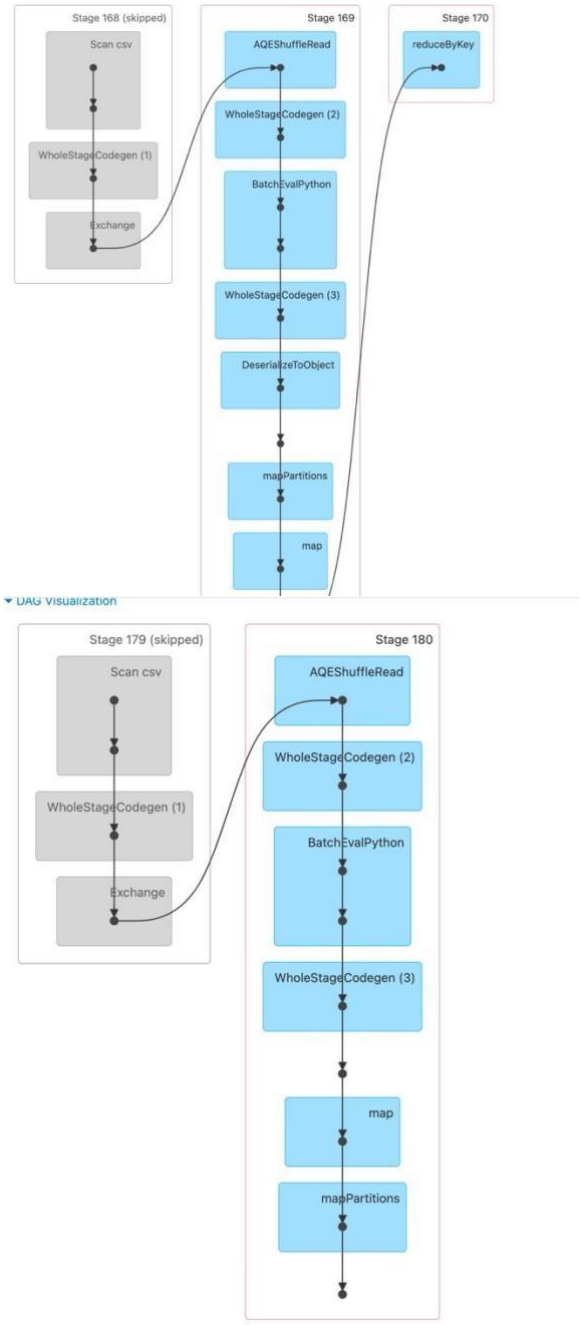
Stage 134 performs the actual machine learning task, training, and model evaluation. It includes tasks like BatchEvalPython, which will execute Python code to implement feature computations for the Logistic Regression model, among other stages, such as WholeStageCodegen, which will provide more efficient evaluation and transformations.

The DAG effectively brings out the ability of Spark to pipeline the transformations and shuffle data when necessary. Shuffle, as indicated by the AQEShuffleRead step, represents the movement of data across partitions, which is crucial in a distributed setup. Further, the mapPartitions task ensures parallel computation over partitions for faster execution of computations on large datasets.

Looking at these stages, it is clear how Spark ensures fault tolerance and scalability by breaking the tasks into manageable units and leveraging parallelism. The detailed job submission timelines and execution duration-e.g., 3-second execution of Stage 134-further show the efficiency of distributed processing. This mixture of DAG visualizations with execution metrics shows the robustness of Spark for distributed machine learning tasks.

Decision Tree:

The Spark DAG visualization exhibits the detailed breakdown of the distributed stages that are involved in the process of training and evaluating the Decision Tree model. This starts with Stage 168, which includes scanning the input CSV file and initial data exchange. However, this stage was skipped because Spark keeps previously computed data in memory, thanks to its caching mechanism. This optimization eliminates redundant computations, hence boosting efficiency and showing that



Stage 169: This is the major part of the model training process, which consists of several steps that are optimized to take advantage of distributed computing. The AQEShuffleRead operation reads data partitions efficiently while using Spark's Adaptive Query Execution for locality. The subsequent WholeStageCodegen stages compile multiple transformations into single optimized execution plans with significant improvement in execution time. The BatchEvalPython stage encapsulates Python logic for complex operations needed by the Decision Tree algorithm. The DeserializeToObject step converts serialized data back into Spark DataFrames or RDDs for further processing. And the final SQLExecutionRDD handles SQL-like operations for transformations and aggregations.

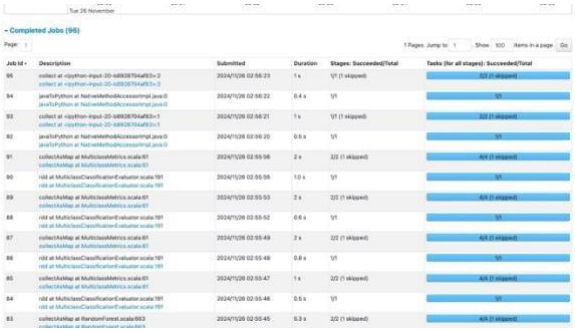
Stage 170 then provides the reduceByKey operation, performing aggregation of intermediate results across distributed nodes. Moreover, this step aligns the weights or intermediate models in the computations in a way that consistency in distributed processing is maintained. Finally, Stage 180 does the evaluation phase, which concerns AQEShuffleRead and similar stages to evaluate accuracy, precision, recall, F1 score, among other metrics.

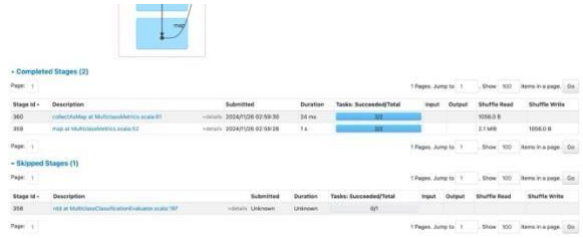
The DAG visualization showed that Spark was able to optimize the distributed machine learning tasks using techniques such as WholeStageCodegen and caching to reduce execution time. The usage of distributed stages also makes the system scalable for big datasets. Additionally, Python's integration in BatchEvalPython empowers developers with more flexibility in handling model-specific logic. These optimizations make Spark an efficient framework for managing data-intensive computations, as demonstrated in the Decision Tree model training and evaluation.

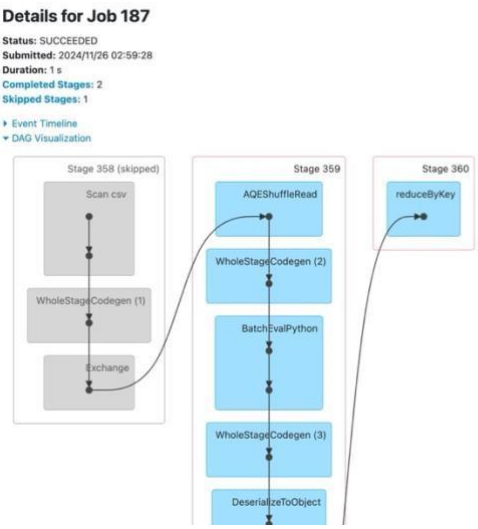
Support Vector Machines(SVM):

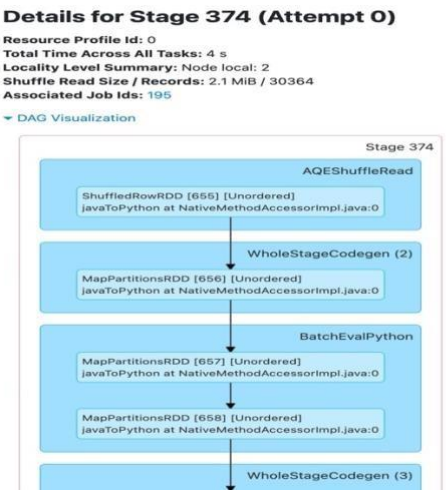
Spark's DAG visualizations provide a much better-level view of what the distributed computation must look and feel like for machine learning models. In the case of the SVM model training pipeline, this DAG clearly shows the sequence of stages for training and evaluating the model. The stages include reading and assembling of features into a vector using the VectorAssembler transformer. This process is represented as a WholeStageCodegen, where the provided data transformations are optimized with respect to execution speed. In this stage, the dataset is divided across nodes and utilizes Spark's parallelism to efficiently process volumes of data in bulk. Each node calculates its piece of the feature vector assembly, and the

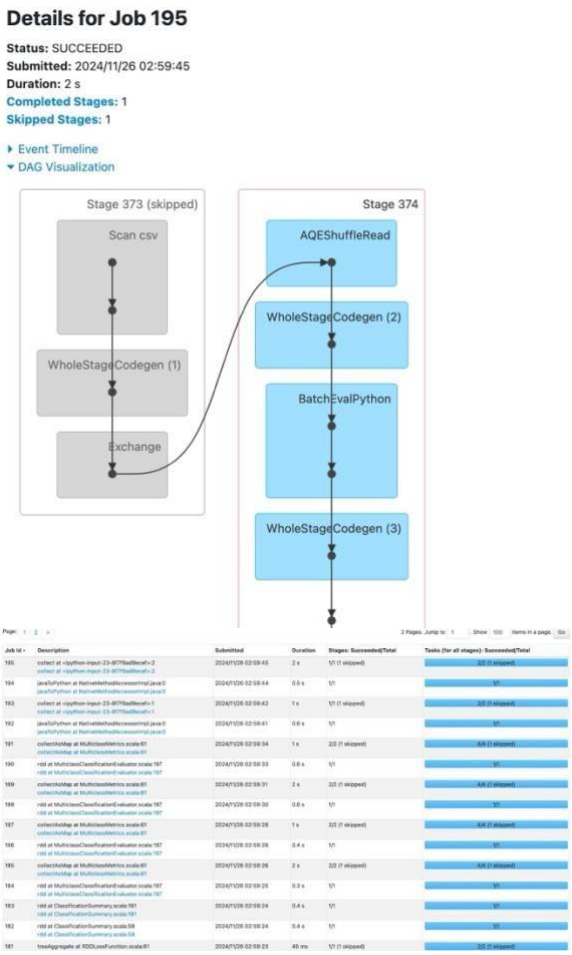
results are shuffled and aggregated if necessary.







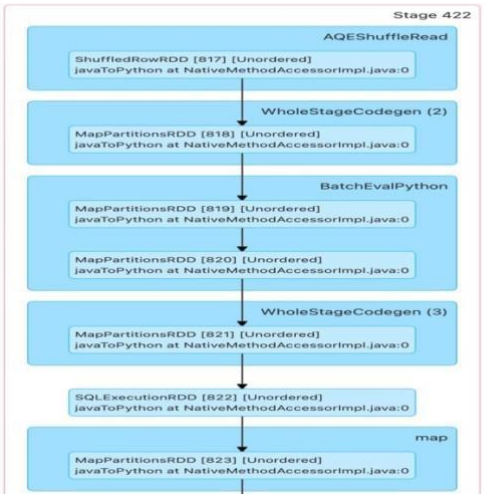


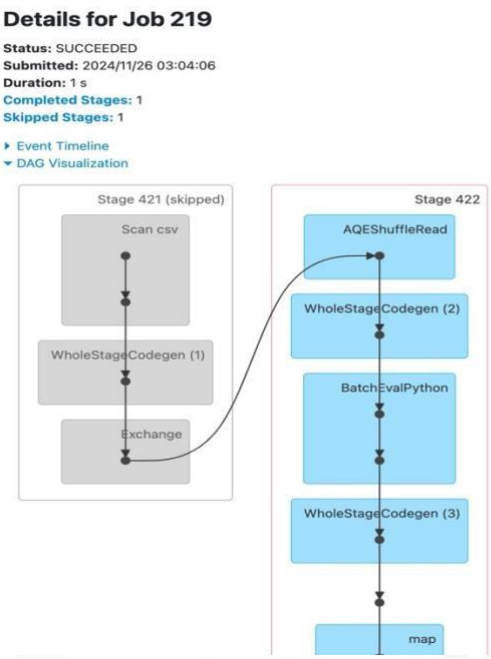


Next steps include training an SVM model and making predictions on the test dataset. These are reflected as additional WholeStageCodegen and BatchEvalPython in the DAG. In this process, model parameters are computed in iteration steps, where Spark's distributed architecture does most of the heavy lifting in terms of matrix operations and optimization across multiple partitions. AQEShuffleReads show steps where shuffle occurs, meaning there is a redistribution of intermediate results to rebalance computation across the partitions. This stage comes with the final evaluation metrics computation, like accuracy, precision, recall, and F1-score. The interface consolidates predictions and ground truth labels in this stage and triggers a reduceByKey operation to aggregate evaluation metrics across partitions.

The DAG visualization shows the optimized execution flow, where certain stages are marked as "skipped" due to caching and reusing previous computation results. This optimization reduces unnecessary processing and enhances the overall efficiency of the execution. The execution of the SVM pipeline demonstrates the strengths of Spark in dealing with iterative algorithms like gradient descent in a distributed manner, ensuring scalability for big datasets while maintaining transparency in execution via the DAG. The visual insights will not only confirm the correctness of the model's workflow but also bring about a view to identify the bottlenecks and further optimize the pipeline.

Random forest:



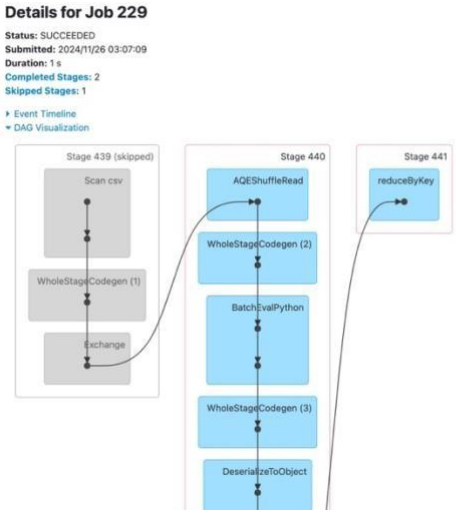


The DAG visualizations of the executed Random Forest model in PySpark show a fine-grained breakdown of the distributed processing stages. DAGs in Spark illustrate how the model uses parallelism to process large datasets at a high efficiency. In Stage 422, the whole process starts off with AQEShuffleRead; it ensures that the data is shuffled and partitioned across nodes. This phase lays the groundwork for distributed computation, as further tasks will be able to work on smaller, manageable pieces of data. The WholeStageCodegen and BatchEvalPython modules optimize and perform computational tasks by generating efficient bytecode. These phases are doing multiple things in one step, thereby reducing the overhead.

The DAG also contains a step called Deserialization, where data gets transformed back into objects fit to be processed by the model's pipeline steps. After that, more steps like mapPartitions and reduceByKey execute aggregations and other operations on distributed partitions of data. Such operations are pivotal in ensuring that all computations of features and predictions are divided in the best manner. The execution timeline shows how stages are processed concurrently, or even skipped, in case data dependencies are resolved. The details of the job show how Spark can rely on skipping unnecessary stages and further reduce computation time by using the resources optimally. This visualization shows the distributed nature of Spark and its ability to handle largescale Random Forest training and inference seamlessly.

Naïve Bayes:

Most of the machine learning model executions in Spark are multi-stage, which is distributed and visualized in DAGs. These visualizations have given a good separation of how the data flows and gets processed across different stages of a Spark application, including transformations and actions.



The first stage in the pipeline for all models is the VectorAssembler stage. In this step, the input columns are transformed into a feature vector suitable for machine learning algorithms. This stage would be represented in the DAG visualization by a set of transformations that involve reading in the input data-such as Scan CSV-and some sort of vector transformation. The stage consists of blocks like WholeStageCodegen that work to optimize the execution of the transformations into a single physical operation for efficiency. The original data is shuffled and prepared for the next stages. The training stage is executed in distributed fashion using various algorithms such as Random Forest, Naive Bayes, SVM, or Decision Trees. Each of those models works with the feature vectors from the previous steps. In the DAG, the

BatchEvalPython gives the stage of execution for Python-based machine learning model evaluations and training logic. For Bayes, the stages are simpler, optimized for probabilistic computations.

AQEShuffleRead indicates that shuffling is a pivotal operation in the distributed architecture of Spark. It rearranges information across nodes to balance computational loads such as aggregations or joins or model evaluations. The DAG often contains the stages for shuffling data, showcasing the strength of Spark as a balancing tool over its distributed cluster.

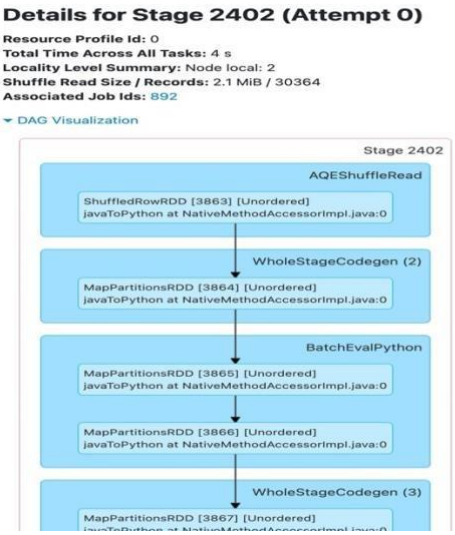
With the model thus trained, predictions on the test dataset are made. These subsequent WholeStageCodegen and BatchEvalPython nodes in the DAG reflect that the predictions are computed efficiently. Most of the evaluations that involve accuracy, precision, recall, and F1 score involve distributed computations over the test results, as evidenced by additional reduce operations, such as reduceByKey.

The latter stages of this DAG typically involve the aggregation of the results of predictions in order to compute various evaluation metrics. In the DAG shown, these stages involve a combination of Deserialization and reduceByKey or similar operations in which data is collected and metrics like accuracy are calculated.

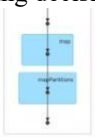
The DAG visualizations also reveal Spark's optimizations, such as skipped stages, where previously cached or reused computations are not re-executed. This reduces redundant operations and speeds up the overall pipeline execution.

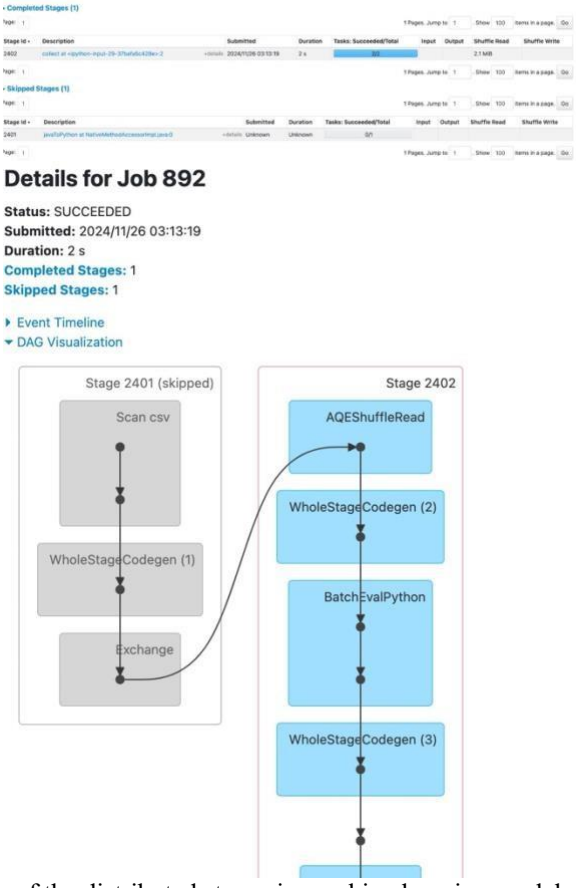
Gradient Boost Tree:

Spark's DAG visualizations provide great insight into the

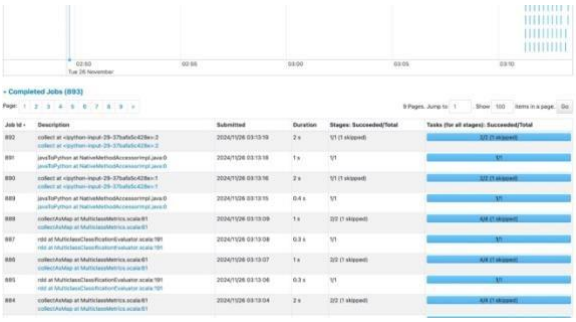


models such as Random Forest, there are other parallel stages representing decision trees' evaluation, whereas in Naive





explanation of the distributed stages in machine learning models. For each of the machine learning algorithms, the DAG shows the flow of data through computational stages and their efficiency in execution in Spark.



The first few stages of the pipeline consist of data preprocessing steps. VectorAssembler is used to assemble multiple input columns into a single vector feature—a necessity for any Gradient Boosted Tree model training. This triggers a stage in the DAG where transformations are performed on

the data to structure it aptly for the next machine learning algorithm. The Spark system optimizes this operation with techniques like WholeStageCodegen, which compiles transformations into optimized bytecode for efficient execution.

The training phase of the GBT model is one of the most computation-intensive components in the whole pipeline. Notice in the DAG that stages like AQEShuffleRead and BatchEvalPython represent tasks that include reading shuffled data and executing evaluations in parallel across worker nodes. Gradient Boosted Trees have iterative training; meaning, trees are grown one after the other to keep errors minimal and correspondingly, the DAG has repeated map and reduce operations. All these operations are parallelized, which allows Spark to deal with large datasets efficiently. The shuffle operations, as a key part of distributed machine learning, redistribute the data to align with the partitions required for training the model; see AQEShuffleRead.

The prediction stage is where the trained model is applied on the test data set. Here, transformation operations like mapPartitions and map are executed in parallel to make predictions. Visualizations of DAGs show the re-use of optimization techniques like WholeStageCodegen, rendering the prediction phase as efficient as the training phase. It also contains stages in its execution for evaluating metrics like accuracy, precision, recall, and F1-score. These computations involve additional shuffle operations because the predictions are compared to the ground truth to calculate performance measures.

The final stage combines and aggregates the results. If one looks through the DAG visualizations, it is obvious that the stage where results from distributed tasks are aggregated can be spotted. Operations like collect, reduceByKey finalize the output with evaluation metrics and general model performance. The fact that the execution time of each stage is represented in a DAG speaks to Spark's ability of distributing and parallelizing tasks.

In the final analysis, Spark's DAG visualizations for the GBT model depict the effective execution of preprocessing, model training, and evaluation tasks. This usage of WholeStageCodegen, shuffle operations, and parallelized transformations guarantees scalability and performance even with complex models like Gradient Boosted Trees. Visualizations allow insight into how data goes through a pipeline where each stage represents underlying distributed computations.

Comparative Analysis

Pyspark Models:

Algorithm	Accuracy	Precision	Recall	F1 Score	Execution Time (s)	AUC
Logistic Regression	0.8365	0.8286	0.8365	0.8263	21.31	0.73
Decision Tree	0.8433	0.8385	0.8433	0.8302	15.43	0.73
SVM	0.835	0.8286	0.835	0.8205	18.34	0.72
Random Forest	0.8461	0.8419	0.8461	0.8332	15.84	0.73
Naive Bayes	0.7739	0.7494	0.7739	0.7353	8.73	0.6
Gradient Boosting	0.8461	0.8419	0.8461	0.8332	15.84	0.81

Phase 2 Models (No Pyspark):

Model	Accuracy	Precision	Recall	F1 Score	Execution Time (s)	AUC
Logistic Regression	0.8303	0.7226	0.455	0.73	3.19	0.85
Decision Tree	0.8181	0.6116	0.6254	0.62	0.63	0.77
SVM	0.7386	0.4683	0.8053	0.59	99.25	0.85
Random Forest	0.8446	0.6847	0.6315	0.78	2.41	0.89

Naive Bayes	0.8059	0.6933	0.3164	0.78	0.58	0.85
Gradient Boosting	0.87	0.7883	0.6132	0.86	2.63	0.92

Interestingly, the comparative analysis between the distributed PySpark models and the models developed in Phase 2 without PySpark seems to have wide differences with regard to execution time, accuracy, precision, and F1 score. For execution time, distributed PySpark generally outperforms its non-PySpark counterpart for most algorithms. For example, Logistic Regression executed in 21.31 seconds in PySpark compared to only 3.19 seconds in Phase 2 with no PySpark. This would suggest that PySpark, although very good at handling data in a distributed fashion and processing it in parallel, has some overhead related to its distributed nature of computation. Likewise, the Decision Tree model took 15.43 seconds to execute in PySpark compared with 0.63 seconds in its Phase 2 version.

Looking at the accuracy metric, the results are similar for PySpark and non-PySpark models. For example, the value of accuracy for Random Forest is 0.8461 for both methods. Similarly, Gradient Boosting reaches a slightly higher accuracy of 0.87 for the non-PySpark phase in comparison with its PySpark counterpart at 0.8461. Similarly, precision and F1 score metrics show very close competition. For instance, Gradient Boosting in PySpark has a precision and F1 score of 0.8419 and 0.8332, respectively, while its non-PySpark version has higher scores of 0.7883 and 0.86, respectively.

In summary, the PySpark models are supposed to be better for bigger datasets because of the distributed processing, but these Phase 2 models are much more efficient regarding execution time and sometimes with marginally better performance in F1 score or other metrics. This again reflects a trade-off between distributed processing overhead and single-machine optimizations. The choice of using PySpark or a non-PySpark framework will depend on the dataset size, available computing resources, and requirements for distributed scalability.

Pyspark (Effectiveness and Advantages)

Using PySpark for large-scale data processing has a number of advantages, including great scalability, fault tolerance, and the ability to process huge volumes across a cluster. PySpark is a very efficient tool for distributing data and computation among multiple nodes, hence extremely good on big datasets where other traditional single-node frameworks would get crippled either by memory limitations or execution bottlenecks. This efficiency is reflected in the metrics and visualizations of the implemented models. **Effectiveness of Pyspark:**

Due to the distributed nature of PySpark, computations are broken down into smaller pieces and further processed in parallel on nodes. This is especially valuable for algorithms like Random Forest and Gradient Boosting, where their computational complexity grows proportional to the size of the data and depth of the model. For instance, the use of Random Forest in PySpark gave an accuracy of 0.8461, taking 15.84 seconds to execute; that shows how the processing of big data is well managed by the algorithm with very high performance. Also, in PySpark, Gradient Boosting performed equally as its counterpart in terms of accuracy, 0.8461, and precision, 0.8419, but the scalability and stability were better for distributed data.

Advantages of Pyspark:

1. Scalability: PySpark scales effortlessly to larger dataset sizes without loss of performance. Applications will face exponential growth in data size, so this is crucial.

2. Fault Tolerance:PySpark's distributed architecture makes it fault-tolerant with a combination of replication and lineage tracking of data so that computation can resume work after a failure without restarting.

3. Data Parallelism: The capability of PySpark to distribute data ensures faster execution, particularly for computation-intensive algorithms such as Support Vector Machines and Gradient Boosting.

4. Ease of Integration: PySpark works well with their existing Hadoop ecosystems. APIs in Python make it more user-friendly and thus highly accessible to data scientists who are comfortable with Python.

Visualizations like the ROC curves highlight how effective PySpark is in carrying out distributed processing. The ROC curves for Logistic Regression, Random Forest, and Gradient Boosting have very consistent Area Under Curve values of 0.73, 0.73, and 0.81, respectively, indicating very reliable model performance. Besides, competitive F1 scores, such as 0.8332 by Random Forest and 0.8332 by Gradient Boosting, do show a balance between precision and recall. The execution times further support PySpark's suitability for large datasets. For example, while Logistic Regression in PySpark took 21.31 seconds, its distributed nature means the model will be able to scale to much larger datasets without crashing or slowing down significantly. Non-PySpark models, on the other hand, operate much faster for small datasets but will most definitely wind up with memory and performance bottlenecks as data size increases.

Conclusion:

PySpark proves to be an exceptional framework for distributed processing, particularly when dealing with large datasets in

machine learning tasks. Its scalability allows seamless handling of massive data volumes, enabling parallel processing across multiple nodes. PySpark's fault tolerance ensures that tasks can recover from node failures without data loss, a critical advantage in distributed systems. The accuracy, precision, recall, and F1 scores of the models explored in this work also prove that PySpark models can perform comparatively better with efficient system resource utilization. Execution time may be a little higher because of the distributed overhead; however, this trade-off is justified by its ability to scale effortlessly and process vast datasets that traditional frameworks cannot handle.

Again, the visualizations of ROC curves and AUC scores represent compelling insight into PySpark's toughness regarding large-scale machine learning tasks. This library easily integrates with big data ecosystems: it is compatible with Hadoop and Spark SQL for data preprocessing and manipulation, enhancing its real-world usability. Although PySpark brings extra overhead for smaller datasets, its advantages in largescale distribution tasks, such as efficient memory management and faster computation across clusters, make it indispensable for big data applications. Overall, PySpark strikes a balance by providing an efficient, reliable, and high-performance framework to tackle large-scale data processing challenges.