Analyzing Irregularities in Global Financial Transactions Using Black Money Dataset

1st Sai Sri Kolanu
50594437
MS in ES AI
University at Buffalo
saisriko@buffalo.edu

2nd Jyothsna Devi Goru
50560456
MS in ES AI
University at Buffalo
jgoru@buffalo.edu

3rd Hamsika Rajeshwar Rao
50613199
MS in ES AI
University at Buffalo
hamsikar@buffalo.edu

Abstract— **PySpark is used in this project to process and clean the "Big Black Money" dataset in a distributed manner. It focuses on using big data methods, such as windowing and RDD operations, to effectively preprocess the dataset. In order to improve the quality of the data for analysis, the project records at least four different preprocessing steps. Using PySpark's MLlib, six important machine learning algorithms are developed as well, customized for the dataset's domain, and trained on sizable, dispersed datasets. This comprises models from extra sources as well as the typical course materials.**
Lastly, the distributed phases of data preprocessing and model training are described using Spark's Directed Acyclic Graph (DAG) visualizations. Execution time, accuracy, precision, and F1 score are the main metrics used to compare these distributed models' performance to those from earlier stages. The benefits of utilizing PySpark for reliable machine learning and effective data processing in big data settings are demonstrated by this project.

Keywords— Big Data, Data Cleaning, Data Processing, RDD Operations, Windowing Techniques, Machine Learning Algorithms, Big Black Money Dataset, Directed Acyclic Graph (DAG), Performance Analysis, Execution Time, Accuracy, Precision, F1 Score, Preprocessing Operations, Robust Model Development, Scalability, Data Quality

I.INTRODUCTION

Large dataset analytics and distributed data processing are made easier with PySpark, a robust Python API for Apache Spark. It is substantially faster than conventional data processing techniques because it makes use of Resilient Distributed Datasets (RDDs) to provide parallel processing across clusters. PySpark offers a versatile and effective framework for managing large data difficulties with features like PySpark SQL for data querying and MLlib for machine learning. For data scientists and analysts working in big data contexts, its fault tolerance, scalability, and ease of use make it a vital tool.

II.PROBLEM STATEMENT

TITLE: Analyzing Irregularities in Global Financial Transactions Using Black Money Dataset

Problem Statement:
Using PySpark, the project seeks to overcome the difficulties in effectively processing and analyzing the huge "Big Black Money" dataset. It aims to use efficient preprocessing and data cleaning methods by making use of distributed computing capabilities. In order to find insights on financial crime, the project will also use PySpark's MLlib to create machine learning algorithms. Efficiency advantages will be highlighted through performance comparisons between distributed processing techniques and conventional methods. Lastly, the project will make use of Spark's DAG visualizations to streamline the workflow for data processing.

Key Questions:

1. How might the distributed processing features of PySpark be used to efficiently clean and preprocess a complicated, large-scale dataset?
2. Can the "Big Black Money" dataset be accurately and effectively analyzed using machine learning algorithms built using PySpark's MLlib?
3. When working with large data in financial crime analysis, what are the performance advantages of employing distributed processing techniques over conventional approaches?
4. How can the distributed processing phases be better understood and the workflow as a whole optimized using Spark's DAG visualizations?

A. Background of the Problem

The "Big Black Money" dataset, which focuses on identifying and examining possibly illegal financial transactions, is a crucial challenge in financial forensics. These datasets, which contain complicated patterns of financial movements across various industries and jurisdictions, are too large and complex for traditional data processing methods to manage. Advanced computational algorithms that can effectively evaluate massive volumes of transactional data, spot suspicious trends, and offer actionable insights are required due to the growing sophistication of financial crimes. Frameworks for distributed computing, such as PySpark, present a viable way to get over computational constraints and allow for thorough examination of intricate financial data. Through the use of cutting-edge data processing and machine learning techniques, the initiative seeks to create a solid framework for comprehending and maybe reducing financial abnormalities. make use of shell corporations, tax havens, and intricate international transactions, frequently foil these efforts.

This issue is significant in two ways:

Financial Integrity and Security: The existence of illegal financial transactions and black money compromises the integrity of financial systems, posing serious regulatory and economic problems. It jeopardizes the stability of the national and global financial systems by permitting practices like tax evasion and money laundering, which can finance illicit activity and undermine public confidence in financial institutions.

Challenges in Data Processing and Analysis: Data processing and analysis are severely hampered by the sheer volume and complexity of financial transaction data. Large datasets frequently contain hidden patterns that are difficult to find using traditional methods. In order to effectively process, analyze, and extract insights from big datasets—thereby aiding in the detection of suspicious activity and improving regulatory compliance—it is imperative to make use of distributed computing frameworks such as PySpark.

B. Potential of the Project

This project has a lot of potential for analyzing the "Big Black Money" dataset using PySpark's MLlib:

1. Scalable Data Processing: By eliminating the drawbacks of conventional techniques, PySpark's distributed computing capabilities allow for the effective processing of massive amounts of financial transaction data.
2. Superior Machine Learning: Numerous classification, grouping, and regression techniques are available in MLlib, enabling in-depth examination of intricate financial trends.
3. Feature Engineering: Key indicators of questionable financial activity can be found with the aid of PySpark's feature extraction and transformation tools.

Why this Contribution is Crucial

This contribution is essential for a number of reasons:

1. Effective management of big data: As businesses produce enormous volumes of data, distributed data processing makes it possible to analyze vast datasets in an effective manner that would be impossible for a single system to manage12.
2. Enhanced speed and performance: Workloads are divided among several nodes, which drastically cuts down on processing time and enables businesses to gain insights and make data-driven choices faster.
3. Flexibility and scalability: By adding additional nodes, distributed processing makes it simple to scale computational resources to accommodate expanding data volumes and challenging analytical workloads.
4. Increased resilience and fault tolerance: The system's distributed architecture offers more resilience. The remaining nodes can carry on processing even if one fails, guaranteeing continuous data analysis..
5. Real-time analytics capabilities: Distributed processing makes it possible to analyze data in real-time or almost real-time, which is essential for applications in time-sensitive fields like fraud detection and finance.

In summary, this project uses PySpark to tackle the difficulties of processing and examining the "Big Black Money" dataset with an emphasis on identifying illegal financial activity. The project intends to effectively clean and preprocess large-scale datasets while creating cutting-edge machine learning algorithms to reveal hidden patterns by employing distributed data processing techniques. This contribution is essential for providing real-time analytics, increasing performance and scalability, and strengthening the integrity of financial systems. In the end, it aims to offer practical insights that might enhance regulatory compliance and lessen financial crimes in an increasingly complicated data environment.

I. DATA SOURCES

Dataset Link:
Global Black Money Transactions Dataset (kaggle.com)

This Data Set Contains 10000 Unique Values with 14 Columns, but some columns have missing values. The data types vary, with some columns containing floats, integers, and strings.

The key columns include:

1. **Transaction ID**: Identifier for each transaction.
2. **Country, Destination Country, Tax Haven Country**: Strings representing various countries.
3. **Amount (USD)**: A numerical value representing the transaction amount.
4. **Transaction Type**: Categorical data indicating the type of transaction.
5. **Date of Transaction**: Date and time information.
6. **Person Involved**: Identifier for individuals involved.
7. **Industry, Financial Institution**: Industry types and associated institutions.
8. **Money Laundering Risk Score**: A numerical risk score for laundering.
9. **Shell Companies Involved**: Integer count of shell companies.

III. DATA CLEANING/PROCESSING

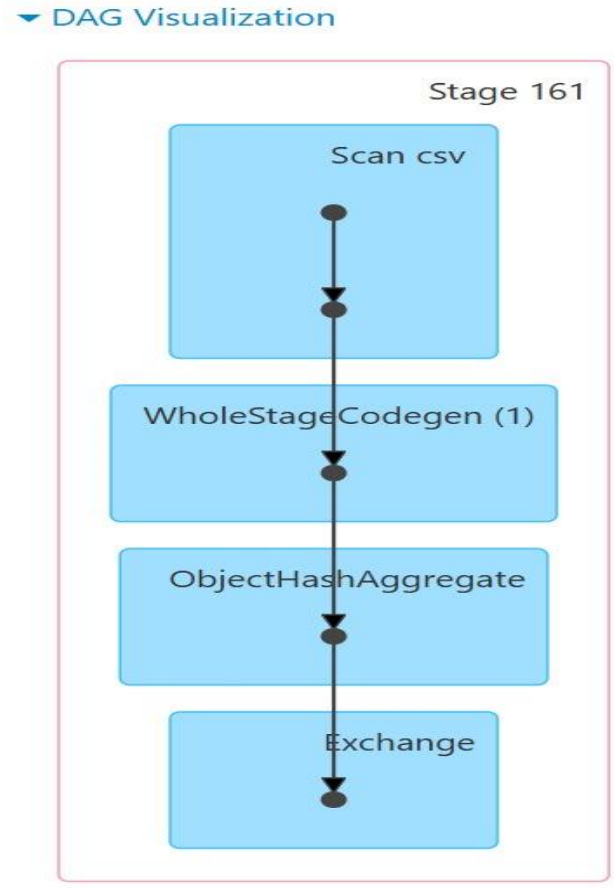
Step 1: converting date colunm to timestamp

Code:

```
from pyspark.sql.functions import col

# Convert 'Date of Transaction' to timestamp
ab = ab.withColumn("Date of Transaction", col("Date of Transaction").cast("timestamp"))
```

Explanation:
This code uses the withColumn function to change the 'Date of Transaction' column in a PySpark DataFrame to a timestamp data type. It guarantees that the column is formatted correctly for time-based processes such as aggregations, filtering, and sorting.

DAG:



This DAG visualization represents the execution plan of a PySpark job for processing a CSV file. It begins with scanning the CSV, followed by WholeStageCodegen for optimizing transformations, and then performs an ObjectHashAggregate to compute aggregated results. The Exchange step indicates a shuffle operation for data redistribution across partitions for further processing.

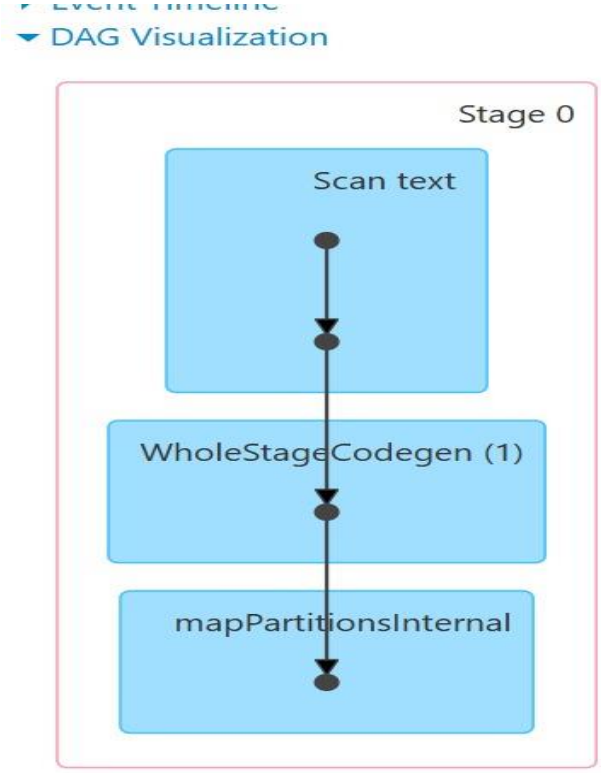
Step 2. Handling Missing values

Code:

```
from pyspark.sql.functions import mean

# Impute missing values in 'Amount (USD)' with the mean amount
m_amount = ab.select(mean("Amount (USD)")).collect()[0][0]
ab = ab.na.fill({"Amount (USD)": m_amount})
```

Explanation:
This code fills in any missing values in the 'Amount (USD)' column by calculating the column's mean. It guarantees that the dataset is free of null values, preparing it for modeling or analysis.

DAG:



This DAG (Directed Acyclic Graph) visualization represents a Spark job with Stage 0. The process begins with Scan text, where Spark reads the input data, likely from a text-based file format. Next, WholeStageCodegen optimizes the processing by generating efficient Java bytecode to execute transformations. Finally, mapPartitionsInternal handles partition-wise transformations, applying the specified logic to each data partition. This DAG illustrates the sequence of operations Spark executes to optimize and process the data efficiently.

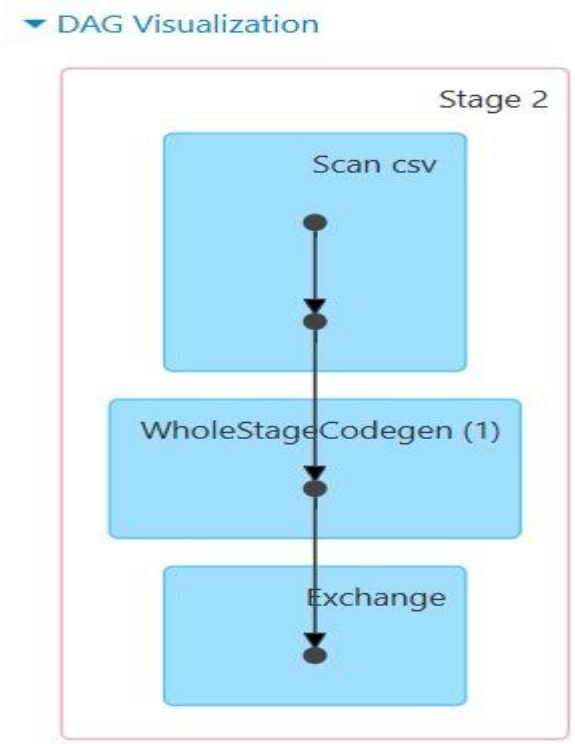
Step 3. Removing Duplicates

Code:

```
# Remove duplicates based on 'Transaction ID'
ab = ab.dropDuplicates(["Transaction ID"])
```

Explanation:
This code uses the dropDuplicates method to eliminate duplicate entries from the DataFrame based on the 'Transaction ID' column. By removing redundant information, it guarantees that every transaction ID is distinct.

DAG:



Scan CSV: The data is read from a CSV file into a DataFrame, forming the initial step in the DAG.

WholeStageCodegen: Optimized code generation is applied to transform the data for removing duplicates efficiently.

Exchange: A shuffle operation is performed to redistribute data across partitions, ensuring duplicates are correctly identified and removed.

Step 4. Removing the Outliers

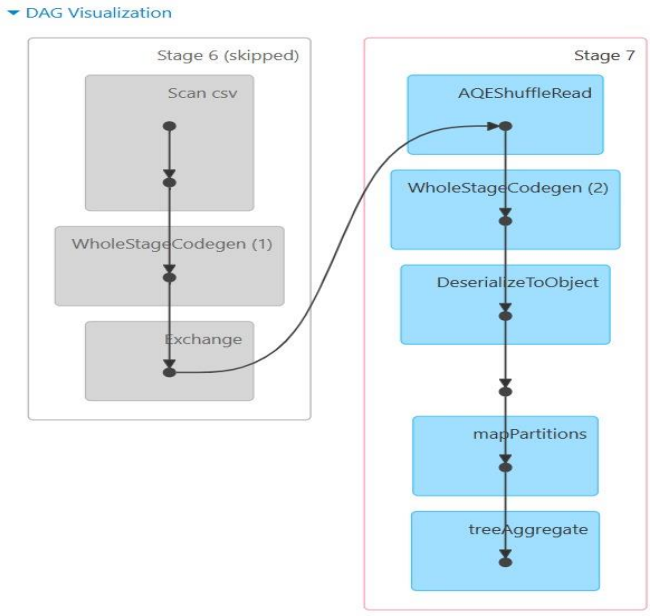
Code:

```
quant = ab.approxQuantile("Amount (USD)", [0.25, 0.75], 0.05)
iqr = quant[1] - quant[0]
lr_bound = quant[0] - 1.5 * iqr
up_bound = quant[1] + 1.5 * iqr

# Filter out outliers
ab = ab.filter((col("Amount (USD)") >= lr_bound) & (col("Amount (USD)") <= up_bound))
```

Explanation:

This code determines the bottom and upper boundaries of the interquartile range (IQR) for the 'Amount (USD)' column in order to detect outliers. To make sure the data doesn't include extreme numbers, rows with quantities outside of specified ranges are filtered out.

DAG:



This DAG illustrates a Spark job with Stage 6 (skipped) and Stage 7. In Stage 6, the job scans a CSV file, processes it using WholeStageCodegen for optimized transformations, and prepares the data for shuffling with an Exchange operation. In Stage 7, AQEShuffleRead dynamically optimizes shuffle data, followed by further computation using WholeStageCodegen (2). The data is deserialized, processed partition-wise with mapPartitions, and aggregated using treeAggregate. This workflow showcases Spark's use of adaptive query execution and optimization for efficient data processing.

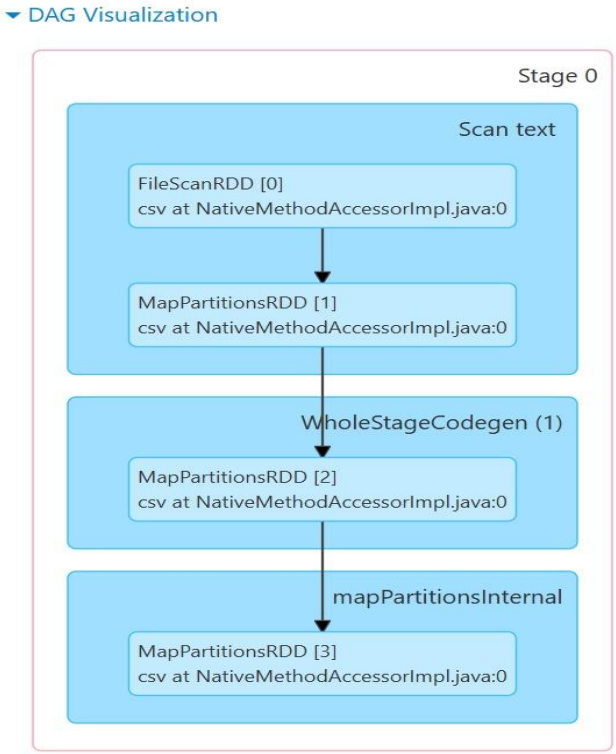
Step 5.

Code:

```
from pyspark.sql.functions import when

# Create a new column 'Risk Category' based on 'Money Laundering Risk Score'
ab = ab.withColumn("Risk Category", when(col("Money Laundering Risk Score") >= 7, "High")
                                    .when(col("Money Laundering Risk Score") >= 4, "Medium")
                                    .otherwise("Low"))
```

Explanation:
'Risk Category' is a new column created by this code utilizing conditional logic based on the 'Money Laundering Risk Score'. For easier interpretation and analysis, it divides the score into "High," "Medium," and "Low" risk categories.

DAG:



Scan Text: The initial step involves scanning the text file using FileScanRDD and converting it into an RDD, which is partitioned for distributed processing.

WholeStageCodegen: An optimization phase where operations on the data are compiled into efficient bytecode for execution, reducing overhead during transformations.

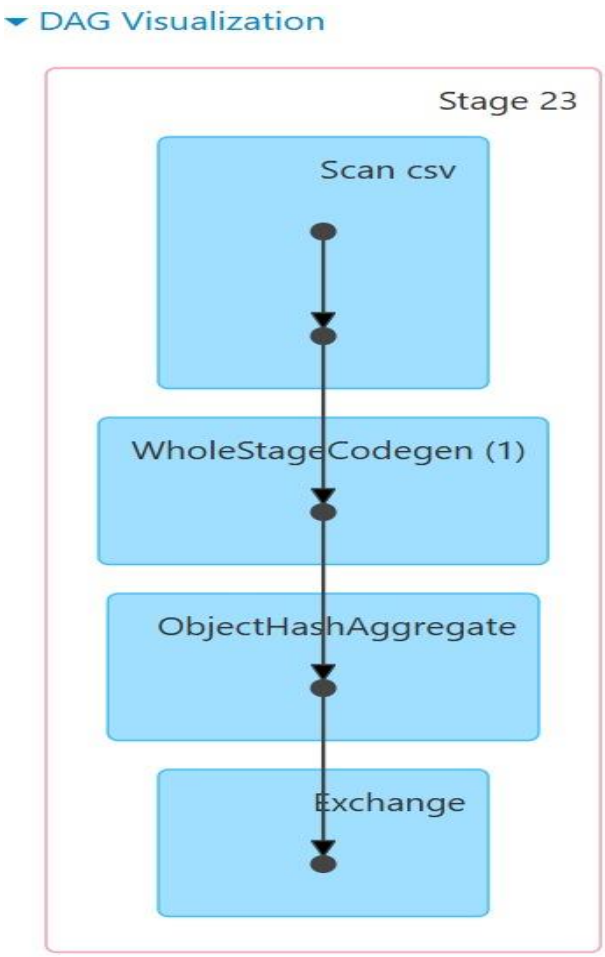
mapPartitionsInternal: The data is processed within each partition, ensuring distributed and parallelized operations for efficient computation

Step 6. One Hot encoding for categorical Variables

Code:

```
from pyspark.sql import functions as F
from pyspark.ml.feature import StringIndexer, OneHotEncoder
from pyspark.ml import Pipeline

# Convert Boolean column to String
ab = ab.withColumn("Reported by Authority", F.col("Reported by Authority").cast("string"))

# List of categorical columns
categorical_columns = [
    'Country',
    'Transaction Type',
    'Person Involved',
    'Industry',
    'Destination Country',
    'Reported by Authority',  # Now converted to string
    'Source of Money',
    'Shell Companies Involved',
    'Financial Institution',
    'Tax Haven Country'
]

# Create stages for the pipeline
stages = []

for column in categorical_columns:
    # Create a StringIndexer for each categorical column
    indexer = StringIndexer(inputCol=column, outputCol=f"{column}Index")

    # Create a OneHotEncoder for each indexed column
    encoder = OneHotEncoder(inputCols=[f"{column}Index"], outputCols=[f"{column}Vec"])

    # Add both stages to the list
    stages.extend([indexer, encoder])

# Create and fit the pipeline
pipeline = Pipeline(stages=stages)
ab = pipeline.fit(ab).transform(ab)

# Show some transformed columns
ab.select([f"{column}Vec" for column in categorical_columns]).show()
```

Explanation:
This code uses StringIndexer and OneHotEncoder in a pipeline to preprocess categorical columns by turning them into numerical representations. It ensures effective handling of categorical features by transforming the data for machine learning compatibility.

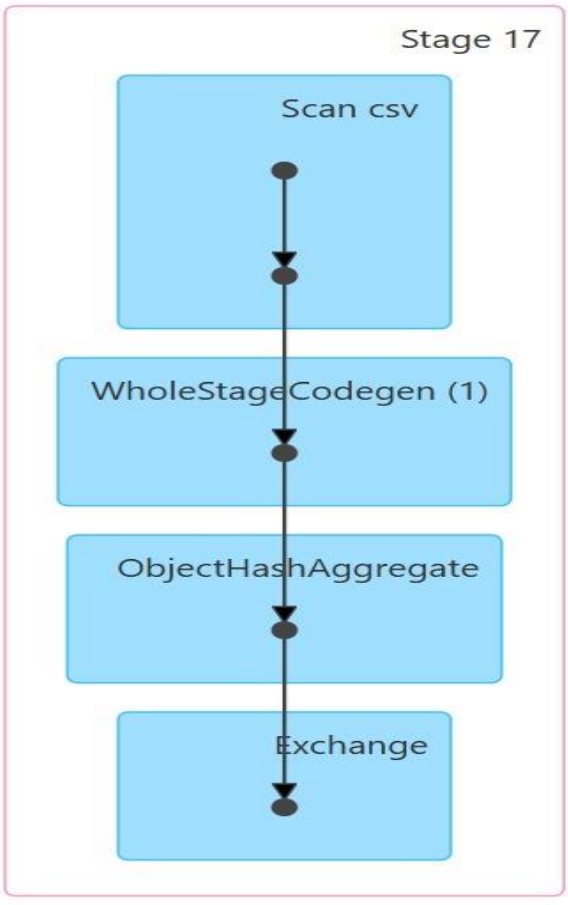


This DAG represents Stage 23 of a Spark job. It begins with Scan csv, where the data is read from a CSV file. The data undergoes optimized processing through WholeStageCodegen (1) for efficient execution. Next, an ObjectHashAggregate operation is performed to aggregate the data efficiently using hash-based techniques. Finally, an Exchange operation is applied, likely for shuffling data across partitions for subsequent stages. This stage demonstrates Spark's optimization in reading, transforming, and aggregating data.

Step 7. Normalizing the Amount(USD)

```
from pyspark.sql.types import DoubleType
from pyspark.ml.feature import MinMaxScaler, VectorAssembler

# Ensure the column is numeric
ab = ab.withColumn("Amount (USD)", ab["Amount (USD)"].cast(DoubleType()))

# Handle nulls if necessary
ab = ab.na.drop(subset=["Amount (USD)"])

# Assemble the vector
ass = VectorAssembler(inputCols=["Amount (USD)"], outputCol="AmountVec")
ab = ass.transform(ab)

# Check the output of the assembler
ab.select("AmountVec").show(truncate=False)

# Apply MinMaxScaler
sclr = MinMaxScaler(inputCol="AmountVec", outputCol="NormalizedAmount")
ab = sclr.fit(ab).transform(ab)

# Check the final output
ab.select("NormalizedAmount").show(truncate=False)
```

DAG Visualization



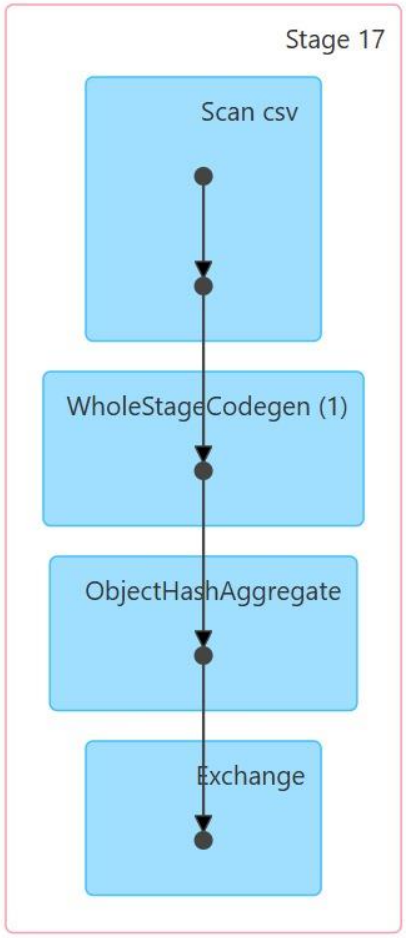
This DAG represents Stage 17 of a Spark job. It begins with Scan csv, where data is read from a CSV file. The WholeStageCodegen (1) step optimizes the transformations by generating efficient execution plans for faster computation. The data is then aggregated using ObjectHashAggregate, which performs hash-based aggregations for efficiency. Finally, an Exchange operation redistributes the data across partitions for subsequent stages. This stage showcases Spark's capability to optimize data reading, processing, and partition management effectively.

Step 8: Aggregation Transactions per Persons
Code:

```
from pyspark.sql.functions import count, sum

# Aggregate number of transactions and total amount per person
prsn_agg_ab = ab.groupBy("Person Involved").agg(
    count("*").alias("Num_Transactions"),
    sum("Amount (USD)").alias("Total_Amount")
)
```

.
Explanation:
This code aggregates data by **'Person Involved'**, calculating the number of transactions and the sum of the **'Amount (USD)'** for each person. It provides a summary of transaction activity for individual
entities.

DAG Visualization



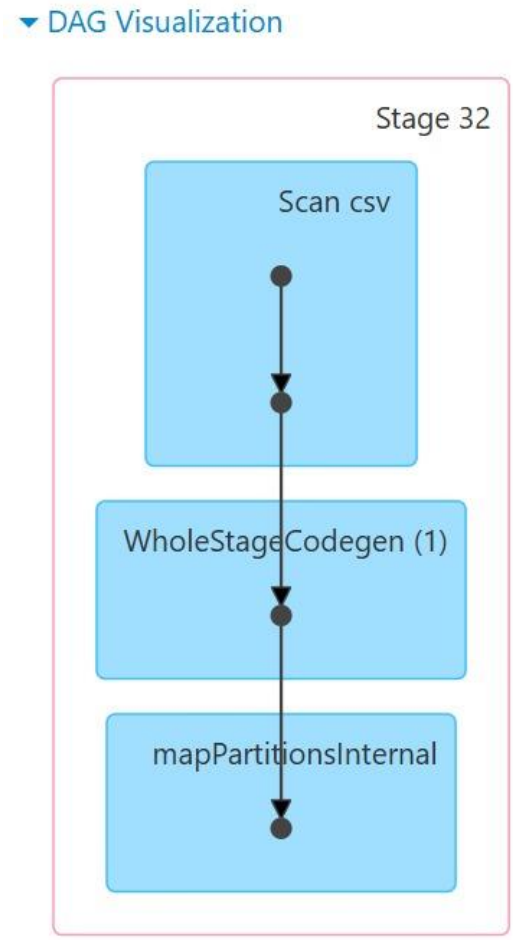
This DAG represents Stage 17 of a Spark job. It begins with Scan csv, where data is read from a CSV file. The WholeStageCodegen

(1) optimizes transformations by generating efficient bytecode for processing. The data is then aggregated using ObjectHashAggregate, which efficiently combines records using a hash-based approach. Finally, an Exchange operation performs data shuffling across partitions to prepare for subsequent stages. This stage highlights Spark's ability to optimize data reading, transformation, and aggregation tasks.

Step 9: Windowing Function for Rolling Average Amount

```
from pyspark.sql.window import Window
from pyspark.sql.functions import avg

winspec = Window.partitionBy("Country").orderBy("Date of Transaction").rowsBetween(-3, 0)

# Calculate rolling average amount for each country over last 4 transactions
ab= ab.withColumn("Rolling_Avg_Amount", avg("Amount (USD)").over(winspec))
```

Explanation:
Using the 'Date of Transaction' as the order, thiscode computes a rolling average of the 'Amount (USD)' for the last four transactions for each 'Country'. It offers information on current transaction trends through the use of a window feature.



This DAG represents Stage 32 of a Spark job. It starts with Scan csv, where the data is read from a CSV file. The data is then processed using WholeStageCodegen (1), which optimizes transformations by generating efficient execution plans. Finally, mapPartitionsInternal applies partition-level transformations to process data in parallel. This stage highlights Spark's efficiency in reading, optimizing, and transforming data across partitions.

IV : Machine Learning Algorithms with PySpark

The ML models we have used in this project is :

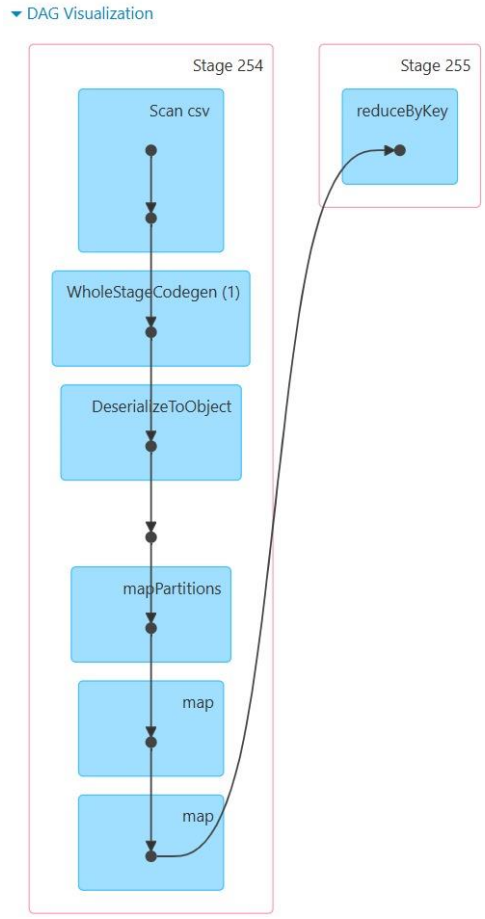
1. Logistic Regression
2. Naïve Bayes
3. SVM
4. KNN: K Neighbor classifier
5. stochastic Gradient Decent
6. Multilayer Perceptron

Model 1: Naïve Bayes

Output:

```
root
 |-- Transaction ID: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- Amount (USD): double (nullable = true)
 |-- Transaction Type: string (nullable = true)
 |-- Date of Transaction: string (nullable = true)
 |-- Person Involved: string (nullable = true)
 |-- Industry: string (nullable = true)
 |-- Destination Country: string (nullable = true)
 |-- Reported by Authority: boolean (nullable = true)
 |-- Source of Money: string (nullable = true)
 |-- Money Laundering Risk Score: integer (nullable = true)
 |-- Shell Companies Involved: integer (nullable = true)
 |-- Financial Institution: string (nullable = true)
 |-- Tax Haven Country: string (nullable = true)

Confusion Matrix:
+---------------------------+----------+-----+
|Money Laundering Risk Score|prediction|count|
+---------------------------+----------+-----+
|                          0|       1.0|    6|
|                          0|       4.0|    1|
|                          1|       1.0|  862|
|                          1|       2.0|   14|
|                          1|       3.0|   25|
|                          1|       4.0|    7|
|                          1|       5.0|   36|
|                          1|       6.0|   13|
|                          1|       7.0|   15|
|                          1|       8.0|   22|
|                          1|       9.0|    3|
|                          1|      10.0|   16|
|                          2|       1.0|  125|
|                          2|       2.0|  496|
|                          2|       3.0|  115|
|                          2|       4.0|   44|
|                          2|       5.0|   26|
|                          2|       6.0|   19|
|                          2|       7.0|   23|
|                          2|       8.0|   22|
+---------------------------+----------+-----+
only showing top 20 rows

Accuracy: 0.9030
Execution Time: 5.714131593704224 seconds
```

Explanation
This code applies MinMaxScaler to scale values between 0 and 1 after
converting the 'Amount (USD)' column to a numeric type and assembling it into a vector. It guarantees that machine learning models scale numerical features consistently.



Model:2 SVM

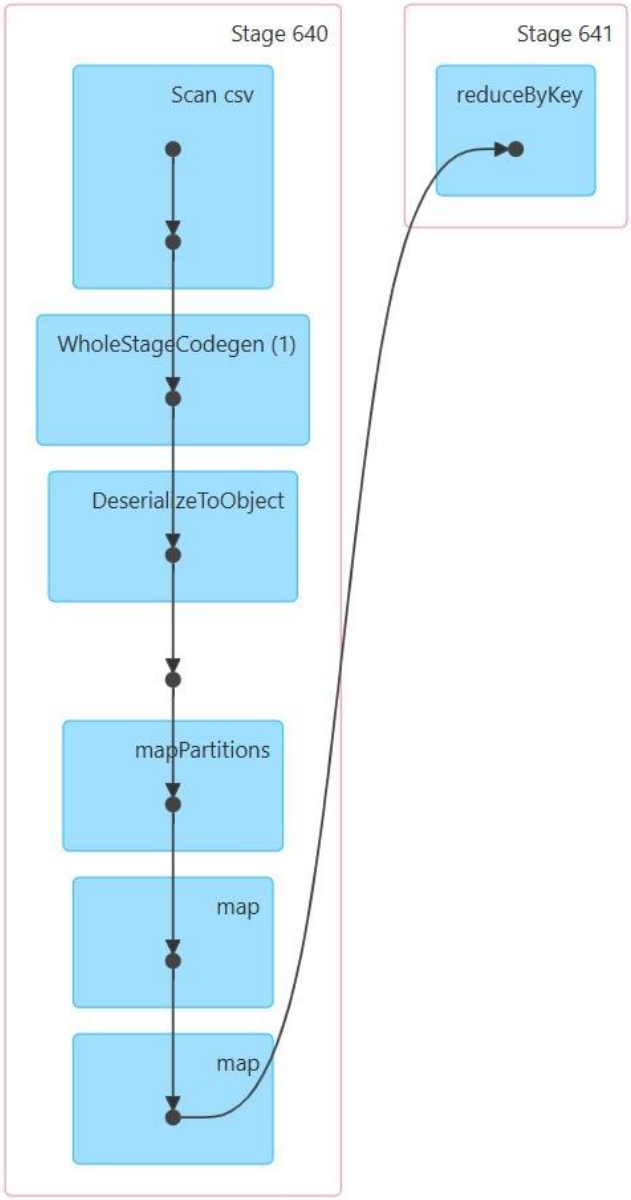
```
root
 |-- Transaction ID: string (nullable = true)
 |-- Country: string (nullable = true)
 |-- Amount (USD): double (nullable = true)
 |-- Transaction Type: string (nullable = true)
 |-- Date of Transaction: string (nullable = true)
 |-- Person Involved: string (nullable = true)
 |-- Industry: string (nullable = true)
 |-- Destination Country: string (nullable = true)
 |-- Reported by Authority: integer (nullable = true)
 |-- Source of Money: string (nullable = true)
 |-- Money Laundering Risk Score: integer (nullable = true)
 |-- Shell Companies Involved: integer (nullable = true)
 |-- Financial Institution: string (nullable = true)
 |-- Tax Haven Country: string (nullable = true)

Confusion Matrix:
+---------------------+----------+-----+
|Reported by Authority|prediction|count|
+---------------------+----------+-----+
|                    0|       0.0| 7722|
|                    0|       1.0|  152|
|                    1|       0.0|  274|
|                    1|       1.0| 1697|
+---------------------+----------+-----+

Accuracy: 0.9208
Execution Time: 22.23213267326355 seconds
```

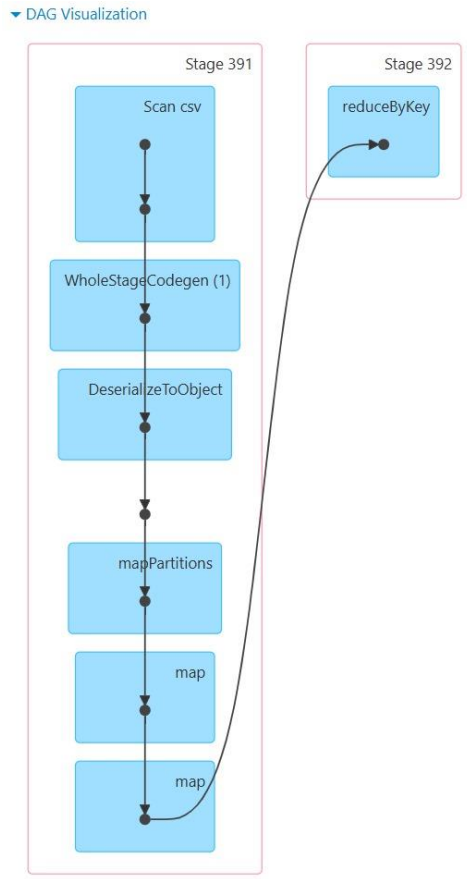
more feature engineering or hyperparameter adjustment.

Explanation :

Overview of the Schema: The Naive Bayes model is trained using the schema, which contains important attributes such transaction information, risk scores, and whether the transaction was reported by an authority.
Accuracy: With an accuracy of 92.08%, the model demonstrates a high level of predictive power for this classification assignment.
Execution Time: The size of the dataset or the difficulty of feature

processing may be the cause of the processing time, which is around





Model 3: Logistic Regression

Output:

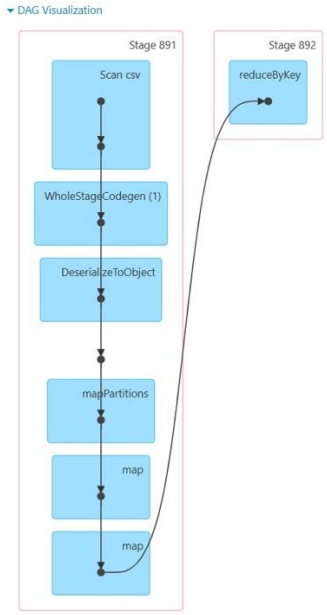
```
Confusion Matrix:
+---------------------+----------+-----+
|Reported by Authority|prediction|count|
+---------------------+----------+-----+
|                    0|       0.0| 7648|
|                    0|       1.0|  226|
|                    1|       0.0|  270|
|                    1|       1.0| 1701|
+---------------------+----------+-----+

Accuracy: 0.9494
Precision: 0.9492
Recall: 0.9496
F1 Score: 0.9494
Execution Time: 21.83720302581787 seconds
```

The confusion matrix shows that the model achieved an accuracy of 94.94%, with 7648 true negatives, 1701 true positives, 226 false positives, and 270 false negatives. Precision (94.92%) and recall (94.96%) indicate a strong balance between correctly predicting positives and capturing all actual positives. The F1 score (94.94%) confirms the model's overall effectiveness. However, there is a small number of false positives and negatives, highlighting areas for potential improvement. The model's execution took approximately 21.8 seconds.

Model 4: KNN K Neighbor Classifier

Output:

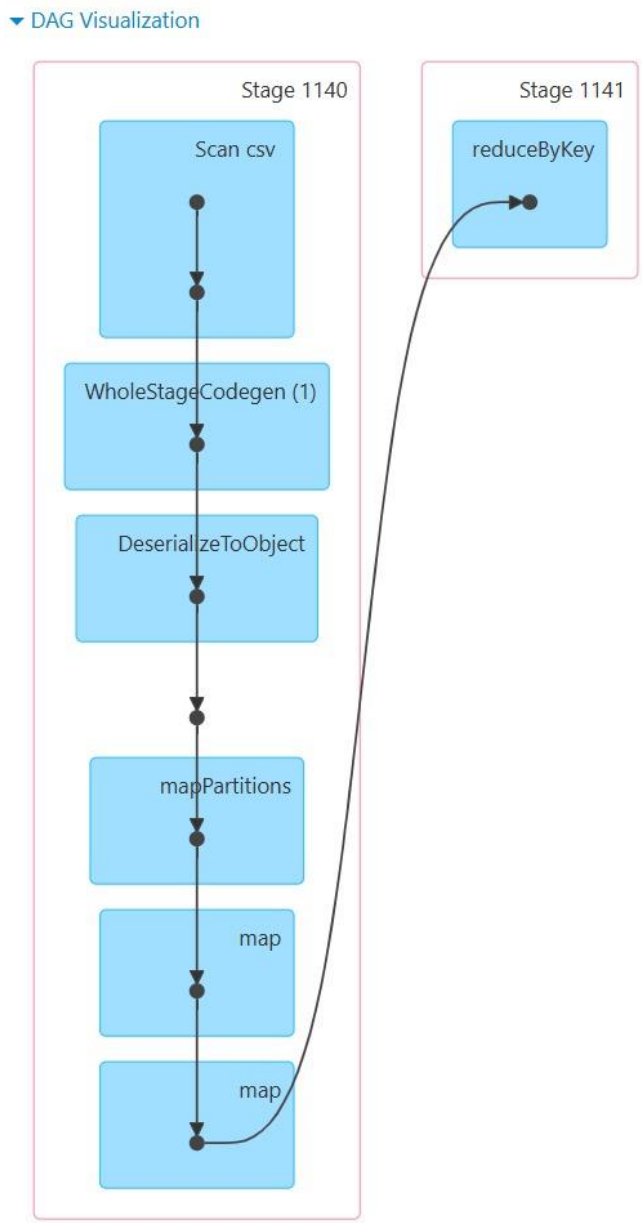
```
Confusion Matrix:
+---------------------+----------+-----+
|Reported by Authority|prediction|count|
+---------------------+----------+-----+
|                    0|       0.0| 7722|
|                    0|       1.0|  152|
|                    1|       0.0|  274|
|                    1|       1.0| 1697|
+---------------------+----------+-----+

Accuracy: 0.9208
Execution Time: 24.58067297935486 seconds
Precision: 0.9561
Recall: 0.9567
F1 Score: 0.9562
```



The model achieved an accuracy of 92.08%, with 7722 true negatives, 1697 true positives, 152 false positives, and 274 false negatives. Precision (95.61%) and recall (95.67%) indicate strong performance in correctly identifying the positive class, leading to a balanced F1 score of 95.62%. Compared to the previous model, this one has slightly lower accuracy but higher precision and recall, suggesting better handling of positive class predictions. However, it took slightly longer to execute (24.58 seconds). Overall, the model is effective, especially for applications prioritizing precision and recall.

Model 5: Stocastic Gradient Descent:

```
Confusion Matrix:
+---------------------+----------+-----+
|Reported by Authority|prediction|count|
+---------------------+----------+-----+
|                    0|       0.0| 7722|
|                    0|       1.0|  152|
|                    1|       0.0|  274|
|                    1|       1.0| 1697|
+---------------------+----------+-----+

Accuracy: 0.9208
Execution Time: 21.688189268112183 seconds
Precision: 0.9561
Recall: 0.9567
F1 Score: 0.9562
```



Model 6: Multilayer Perceptron

Output:

```
Confusion Matrix:
[[1564    1]
 [ 404    0]]

Accuracy: 0.7943
Precision: 0.0000
Recall: 0.0000
F1 Score: 0.0000
```

The model achieved an accuracy of 79.43%, but it completely failed to identify the positive class, with precision, recall, and F1 score all at 0. While it correctly classified 1564 true negatives and only 1 false positive, it misclassified all 404 positive samples as negative. This indicates the model is heavily biased toward the negative class. Despite reasonable accuracy, the performance is inadequate for scenarios requiring correct identification of positive cases, highlighting a need to address class imbalance.