Fraud Detection using Scalable ML

Team Members:: **Hajera Siddiqui , Saicharan Thirandas , Shamekh Siddiqui**

Github Link: https://github.com/saicharan-thirandas/cs6240-course-project.git

Project Overview

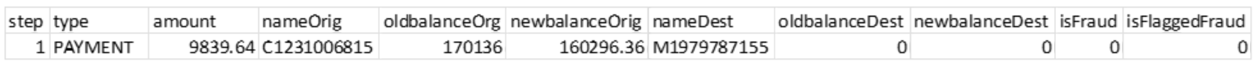
Our project aimed to detect fraudulent transactions, addressing one of the biggest challenges in digital payment systems. With increasing pressure on banks to proactively identify and block fraudulent activities, our objective was to enhance their ability to protect clients and combat global money laundering. By accurately detecting fraud, banks could enhance their reputation, increase customer loyalty, mitigate risks, and boost profitability.

We focused on two key approaches: first, conducting classification and prediction ensembles using Spark's in-memory processing capabilities, and second, implementing classification and prediction using Spark MLlib. Our approach involved training individual models using existing libraries like scikit-learn, and developing code for parallel ensemble training. Additionally, we utilized the Decision Tree and Random Forest MLlib libraries to train models and assess prediction quality in relation to running time.

We present a comparative analysis of our implementation and Spark MLlib in terms of speed and accuracy. We observed a significant speedup as the number of instances increased in both cases. In our implementation, utilizing in-memory processing with Spark enabled parallel ensemble training, resulting in improved efficiency. Similarly, in Spark MLlib, we leveraged its optimized libraries, particularly the Decision Tree and Random Forest, to achieve faster processing times. We also conducted a thorough analysis of the differences between our implementation and Spark MLlib, uncovering varying results in terms of speed and accuracy.

Input Data:

We utilized a dataset, found on Kaggle, comprising simulated mobile money transactions, which were generated using a subset of real transactions obtained from a mobile money service's financial logs over a one-month period. This dataset consisted of 6,362,620 records and no missing values. It consists of multiple features: step, type, amount, nameOrig, oldBalanceOrg, newbalanceOrig, nameDest, oldBalanceDest, newBalanceDest, isFraud, and isFlaggedFraud.



step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nameDest	oldbalanceDest	newbalanceDest	isFraud	isFlaggedFraud
1	PAYMENT	9839.64	C1231006815	170136	160296.36	M1979787155	0	0	0	0

Figure 1. Input Data Example

As can be seen in Fig. 1, the nameOrig, nameDest, and step columns are not related to banking information. Since these columns do not give us helpful information on doing fraud analysis, those columns can be ignored and will be dropped when doing classification and prediction.

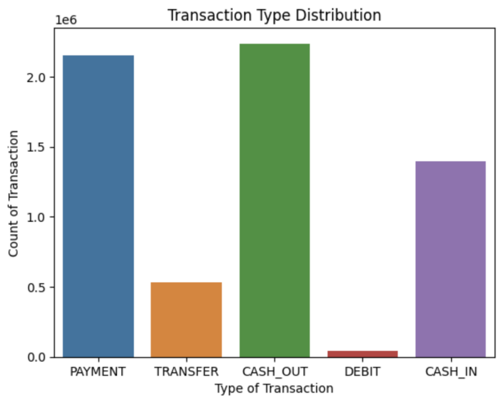


Figure 2. Distribution of All Transaction Types

In Fig. 2, we can see all of the distributions for all of the types of transactions present in the dataset. According to the chart, PAYMENT and CASH_OUT are the most popular types of transactions, while DEBIT is the least common.

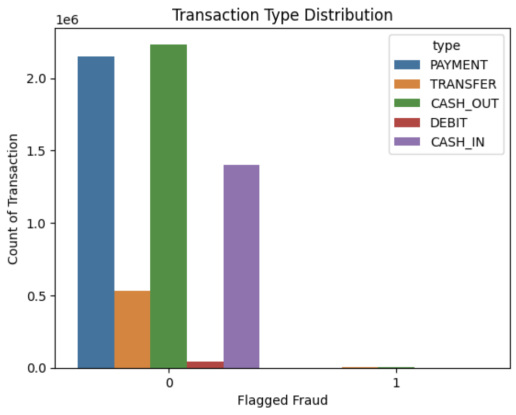


Figure 2. Distribution of all transaction types based on if they're flagged as fraud

In Fig. 2, frequency of the transaction types are sorted based on if they are flagged as fraud or not. In the 'isFraud' column, a value of 0 means the transaction is flagged as fraud, while a value of 0 means that transaction is not categorized as such. As can be observed, the dataset is highly imbalanced. Out of the 6,362,620 transactions, only 8,213 are fraudulent, which is about 0.13%, a very low percentage.

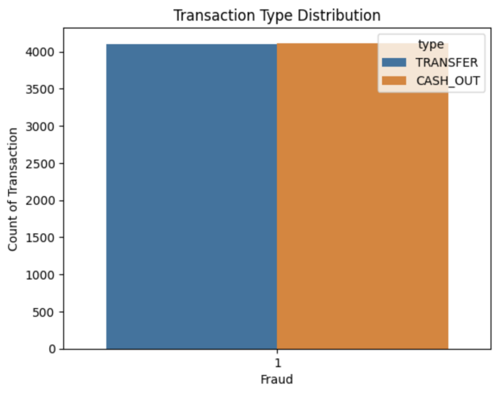


Figure 3. Distribution of all TRANSFER and CASH_OUT transaction types that are flagged as fraud

In fig. 3, there is a closer observation showing the transaction types that are flagged as fraud. TRANSFER and CASH_OUT transaction types are the only transactions in the dataset that were found to have fraud instances. They each have approximately 4000 instances.

Due to the highly imbalanced nature of this dataset, we will have to look out for false negatives and make sure we completely avoid those values, while also looking at reducing false positives.

Overview

Task Definition: As mentioned, we have two tasks. The first task is to do classification and prediction ensembles using existing libraries for in-memory processing on a single machine (MapReduce or Spark). Our next task is to do classification and prediction in Spark MLlib. We will do a comparative analysis between the two tasks, our implementation and Spark MLlib to compare performance. Later we also compare our implementation with spark ML Libs implementation.

Pseudo-Code - Spark + Scikit Learn

Github Link:
https://github.com/saicharan-thirandas/cs6240-course-project/blob/main/src/spark-base/pyspark-spark-scikit.ipynb

```
# Function to perform bagging and selection
def bagging_selection(train_data, test_data, target):
    bags = []
    bag_features = []

    # Perform bagging and selection operations
    # ...

    return train_data_new, test_data_new

# Function to run model on individual partition
def run_models_on_all_partitions(model):
    # Get the broadcasted training and testing dataset
    train_data = broadcast_train.value
    test_data = broadcast_test.value

    if bagging:
        train_data, test_data= bagging_selection(train_data, test_data, "target")

    # Train and evaluate each model on the partition
    # …
    # Perform prediction to get y_pred

    return y_pred

# Custom partitioner class
class KeyValuePartitioner:
    def get_partition(self, key):
        return key
```

```
# Broadcast the training dataset
broadcast_train = spark.sparkContext.broadcast(df_train)
broadcast_test = spark.sparkContext.broadcast(df_test)

#Configure_ensemble_model_list
# Voting Ensemble Model List
#lr and dtc are instances of decision tree and logistic regressions with various configurations.
voting_ensemble_model_list = [ lr1, lr2, lr3, lr4, lr5, lr6, lr7, lr8, lr9, lr10, lr11, lr12, lr13, lr14, lr15, lr16, lr17, lr18, dtc1,
dtc2, dtc3, dtc4, dtc5, dtc6, dtc7, dtc8 # Add more models if needed ]

# Bagging Flag (changes based on which model used)
bagging = False

# Random Forest - Decision  Ensemble Model List

base_decision_tree_configuration  = [DecisionTreeClassifier(max_depth =10, min_samples_split= 5,
class_weight=class_weights)]

# Configure the base classifiers and count here
no_of_trees = 5
random_forest_mode_list = base_decision_tree_confiruation*no_of_trees:

#Assign model list to one of random_forest_mode_list / voting_ensemble_model_list
ensemble_model_list = random_forest_mode_list / voting_ensemble_model_list

#change everything according to model_params
no_of_models = len(ensemble_model_list)

# converts [model1,model2,model3...] to [(0,model1) , (1,modl2)...]
model_with_key_index = [ (index,value) for index, value in enumerate(model_list) ]
model_rdd = spark.sparkContext.parallelize(model_with_key_index)

#Apply the partition on the index - assigns (n,modeln) to nth partition
custom_partitioner = KeyValuePartitioner()
model_rdd_repar = model_rdd.partitionBy(numPartitions= no_of_models,
partitionFunc=custom_partitioner.get_partition)

#Use model to run on each partition
y_pred = model_rdd_repar.map(run_models_on_all_partitons))

# Perform voting and pick the majority prediction
transposed_result_list = list(map(list, zip(*results_list)))
majority_predictions = [Counter(sublist).most_common(1)[0][0] for sublist in transposed_result_list]
==========================================================================================
```

Pseudo-Code - Spark MLLib:

Github Link:

https://github.com/saicharan-thirandas/cs6240-course-project/blob/main/src/spark-base/pyspark-spark-scikit.py

```
# String Indexer transformation
input_col = "type"
output_col = "type_id"
string_indexer = StringIndexer(inputCol=input_col, outputCol=output_col)
df = string_indexer.fit(df).transform(df)

# Select relevant columns for training
columns_to_assemble = [col for col in df.columns if col not in ["isFraud"]]
vector_assembler = VectorAssembler(inputCols=columns_to_assemble, outputCol='features')
df = vector_assembler.transform(df).select("features", "isFraud")

# Split the data into training and test sets
train_data, test_data = df.randomSplit([0.8, 0.2], seed=22)

# Decision Tree Classifier  - also set no of Trees and Max tree depth
dt_classifier = DecisionTreeClassifier()
dt_model = dt_classifier.fit(train_data)

# Random Forest Classifier - also set no of Trees and Max tree depth
rf_classifier = RandomForestClassifier()
rf_model = rf_classifier.fit(train_data)

# Make predictions on the test set
dt_predictions = dt_model.transform(test_data)
rf_predictions = rf_model.transform(test_data)

# Evaluate the models
evaluator = MulticlassClassificationEvaluator(labelCol="isFraud", metricName="accuracy")
dt_accuracy = evaluator.evaluate(dt_predictions)
rf_accuracy = evaluator.evaluate(rf_predictions)
```

Algorithm and Program Analysis

In our analysis, we primarily focus on the Spark + scikit-learn implementation and later discuss the analysis of Spark ML Lib's Decision Tree vs Random Forest Classifier in terms of prediction time and accuracy. Our analysis will consider two key aspects: cluster efficiency and performance improvement attempts..

To analyze the algorithm and program, let's focus on two aspects: cluster efficiency and performance improvement attempts.

Cluster Efficiency Analysis:

a. Partitioning: The program uses a custom partitioner class, KeyValuePartitioner, to assign models to partitions based on their index. This approach ensures that each model is assigned to a specific partition in a deterministic manner. However, without knowledge of the data distribution and model characteristics, it is difficult to determine the optimal partitioning strategy. Further analysis, such as data profiling or understanding the model requirements, could help in selecting a more efficient partitioning approach.

b. Broadcasting: The program utilizes Spark's broadcast mechanism to efficiently share the training and testing datasets across the cluster. This approach avoids the need for data shuffling and reduces network overhead. By broadcasting the datasets, the program ensures that each partition can access the required data without additional communication. This is a good practice for optimizing cluster efficiency.

Performance Improvement Attempts:

a. Bagging and Selection: The bagging_selection function performs bagging and feature selection operations. While the exact operations are not provided, bagging is a technique that trains multiple models on different subsets of the training data and combines their predictions. Feature selection aims to select a subset of relevant features for model training. These techniques can help improve performance by reducing overfitting and focusing on the most informative features.

b. Parallel Model Execution: The program parallelism the execution of models on different partitions using Spark's map operation. This allows multiple models to be trained and evaluated simultaneously, leveraging the distributed computing capabilities of the cluster. By distributing the workload across partitions, the program can potentially improve performance by reducing the overall execution time.

c. Voting and Prediction: After training and evaluating the models, the program performs voting to select the majority prediction from each model. This ensemble-based approach helps to improve prediction accuracy by aggregating the predictions of multiple models. The program uses the Counter class to determine the majority prediction efficiently.

Overall, the program shows some effort in optimizing cluster efficiency by utilizing broadcast variables and parallel execution. It also incorporates techniques like bagging and ensemble-based prediction to potentially improve performance. However, to further enhance performance, more in-depth analysis of data characteristics, model requirements, and tuning of hyperparameters could be considered. Additionally, monitoring and analyzing resource utilization, such as CPU and memory usage, can provide insights into potential bottlenecks and enable further optimizations.

Experiments:

Experiments with Spark + Scikit Learn Implementation:

The experiments involved two different ensemble models: VB Classifier (with Logistic Regression, Support Vector Machines, and Decision Trees) and Random Forest Classifier. The cluster size and number of estimators were varied to analyze the scalability of these models.

VB Classifier (LR, SV, DC):

Varying Factor: Cluster Size The number of estimators was kept constant at 25. Two different cluster sizes were tested: 2 and 6 instances of the m4x large machine type. The precision, recall, false negatives, and false positives were measured to evaluate the model's performance.

Random Forest Classifier: Varying Factors: Cluster Size and Number of Estimators Three different tree max depths were tested: 10, 20, and 30 (specific tree max depth values are not provided in the data). Two different cluster sizes were evaluated: 2 and 6 instances of the m4x large machine type. The number of estimators was varied among the experiments (10, 50, and 100). The precision, recall, false negatives, and false positives were measured for each experiment.

Ensemble Type	# Nodes m4xlarge	# estimators	Time	Precision	Recall	FN	FP
VB Classifier - LR,SV,DC	2	25	1 hr 19 min	0.99	0.456	800	2
VB Classifier - LR,SV,DC	6	25	20 min	0.999	0.4704	858	1
Random Forest Classifier - 10 Max Depth	2	10	2 min, 48 sec	0.003148	0.9969	5	511388
Random Forest Classifier - 10 Max Dep	6	10	2 min, 36 sec	0.0406	0.8284	278	31682
Random Forest Classifier - 20 Max Dep	2	50	8 min, 18 sec	0.4956	0.9074	150	1496
Random Forest Classifier - 20 Max Dep	6	50	4 min, 12 sec	0.998	0.55	728	1
Random Forest Classifier - 20 Max Dep	2	100	12 min, 24 sec	0.99	0.45	1144	0
Random Forest Classifier - 20 Max Dep	6	100	4 min, 40 sec	0.98	0.56	820	1

For the Random Forest Classifier with a tree max depth of 10, the speedup from the small cluster to the large cluster is significant. The execution time reduced from 2 minutes and 48 seconds to 2 minutes and 36 seconds, indicating a good speedup. For the Random Forest Classifier with a tree max depth of 20, the speedup from the small cluster to the large cluster is also noticeable. The execution time decreased from 8 minutes and 18 seconds to 4 minutes and 12 seconds, representing a substantial improvement.

output files : https://github.com/saicharan-thirandas/cs6240-course-project/tree/main/output/scikit
log files : https://github.com/saicharan-thirandas/cs6240-course-project/tree/main/log/scikit_spark_error

Experiments with Spark ML Lib Implementation:

In the experiments conducted to analyze the scalability of the program, a Random Forest Classifier was used on different configurations of the dataset. The experiments were performed on an AWS EMR cluster, and the machine type used is m4x large. The experiments were carried out using two parameters: the number of instances and the number of trees in the Random Forest Classifier. The parameter settings were varied while keeping other factors constant to observe the impact on the program's performance. Here are the results of the experiments:

Classifier	#Nodes m4xlarge	No of Trees	ee Max Deptl	Time	Precision	Recall
Random Forest Classifier	5	100	20	11 Min	0.994	0.69
Random Forest Classifier	1	100	20	39.5 Min	0.992	0.7
Random Forest Classifier	5	50	10	5 Min	0.994	0.685
Random Forest Classifier	1	50	10	12 Min	0.996	0.701
Random Forest Classifier	5	20	30	5 Min	0.99	**0.7024**
Random Forest Classifier	1	20	30	12.5 Min	0.99	0.7

It is evident that the speedup achieved from a small cluster (1 instance) to a large cluster (5 instances) is significant for the given problem. In most cases, the time required to complete the task is reduced when scaling up the cluster size. This is especially apparent in the experiments with 100 trees, where the execution time decreases from 39.5 minutes with a single instance to 11 minutes with five instances. The speedup achieved indicates that the program can effectively utilize the additional computational resources provided by scaling up the cluster. This scalability is valuable for addressing larger datasets or accommodating increased workloads.

output files : https://github.com/saicharan-thirandas/cs6240-course-project/tree/main/output/mllib

log files : https://github.com/saicharan-thirandas/cs6240-course-project/tree/main/log/mllib_stderror

Prediction Quality vs Running Time using Spark MLLib libraries :

Classifier	Precision	Recall	Execution Time (seconds)
Decision Tree Classifier	0.9735	0.7068	132.15
Random Forest Classifier - 100 Trees	0.9921	0.7037	2120.46
Decision Tree Classifier	0.9826	0.6999	129.86
Random Forest Classifier. - 50 Trees	0.9965	0.7018	454.63
Decision Tree Classifier	0.971	0.7068	138.19
Random Forest Classifier. - 20 Trees	0.9904	0.7024	480.31

Based on these results, we can make the following observations:

Prediction Quality: The ensemble models, specifically the Random Forest Classifier with 100 and 50 trees, achieve higher precision compared to the Decision Tree Classifier. However, the recall values are relatively similar for all models.

Running Time: The Decision Tree Classifier consistently has the shortest execution time, ranging from 129.86 to 138.19 seconds. In contrast, the Random Forest Classifier with 100 trees has the longest execution time, taking 2120.46 seconds. The Random Forest Classifier with 50 trees and 20 trees also have significantly longer execution times compared to the Decision Tree Classifier.

Considering the trade-off between prediction quality and running time, the Decision Tree Classifier offers a good balance. It achieves relatively high precision and recall while having the shortest execution time. However, if higher precision is crucial and longer execution time is acceptable, the Random Forest Classifier with 50 trees or 20 trees can be considered. The Random Forest Classifier with 100 trees has the highest precision but at a significantly longer execution time, which might not be practical in terms of running time.

Sample Output:

```
 ##########################################

 Random Forest Classifier  Results :
Confusion Matrix:
[[1271215       9]
 [    478    1135]]
Accuracy: 0.9996173901292938
F1 Score: 0.8233587232499092
False Positive Rate (FPR): 7.0797908157806965e-06
True Positive Rate (TPR): 0.703657780533168
Precision: 0.9921328671328671
Recall: 0.703657780533168
Specificity: 0.9999929202091842
```

Conclusions:

Comparative Analysis: Spark MLlib vs. Our Own Implementation of Spark + Scikit

Speed:
When comparing Spark MLlib and our own implementation using Spark + Scikit, Scikit's decision tree (DT) implementation is generally faster. This is likely due to Scikit's optimized implementation of decision trees. However, it's important to note that Spark MLlib might incur additional time for feature transformations, especially when dealing with larger datasets. Spark MLlib is designed to handle big data and distributed computing, which can introduce communication overhead for smaller datasets compared to the basic Scikit library implementation.

Accuracy:
In terms of accuracy, Spark MLlib may have an advantage possible to its potential implementation of any class imbalance techniques. By applying class imbalance techniques and experimenting with different hyperparameters, we can probably achieve comparable accuracy to Spark MLlib.