Algorithms for massive datasets & Statistical methods for ML: Joint Project 2022-23

Fabio Brambilla[978983]

Università Degli Studi di Milano

Abstract. This project focuses on developing a predictive method to identify fraudulent transactions using data from the "IBM Transactions for Anti Money Laundering" dataset available on Kaggle. The main objective of the project is to create a decision tree classifier from scratch capable of determining whether a transaction is legitimate or fraudulent. Additionally, it is required to implement a Random Forest algorithm, parallelizing the creation of individual trees across multiple workers using Apache Spark. The entire system must be designed for scalability and efficient space utilization.

1 Introduction

Illicit transactions are financial operations that bypass prevailing regulations, often disguised as legitimate transactions. In this context, we focus on analyzing an Anti Money Laundering dataset available on Kaggle, which contains automatically generated data. To address this challenge, we have implemented Random Forests and decision tree classifiers.

Random Forests represent a unique class of machine learning algorithms designed to analyze and classify vast datasets. They are marked by the creation of multiple decision trees during the training process, with the output reflecting the mode of classifications from each individual tree.

During the course of this project, we developed algorithms exclusively using Python and Spark. We constructed the sequential implementation of the decision tree learning algorithm from scratch. For the distributed assembly of the random forest, we adopted an approach rooted in Spark, capitalizing on its parallel processing strengths.

2 Dataset

The utilized data is sourced from "IBM Transactions for Anti Money Laundering (AML)" available on Kaggle[1]. This data is organized as a collection of separate datasets of varying sizes, generated automatically by an algorithm simulating a fictitious virtual world where individuals, companies, and banks exchange money.

These datasets are in CSV format and are categorized by size: `Small`, `Medium` , and `Large`. Within each size category, there are two designations, `HI` and `LI`, referring to the frequency of fraudulent transactions, with `HI` indicating a high frequency and `LI` indicating a low frequency. Additionally, an accompanying text file is provided for each dataset, listing some of the fraudulent patterns found within. These patterns are further elaborated in a paper[2] available directly on the Kaggle website, which concisely details the types of patterns.

Regarding the dataset's class balance, fraudulent transactions are notably fewer compared to regular ones.

2.1 Data organization

As mentioned above, the data are divided into various datasets of different sizes. Each dataset comes in two versions: one with a higher concentration of fraudulent transactions (HI) and the other with a lower concentration (LI).

For the analysis and implementation of our learning models, we focused on the `HI-Small` dataset. This is the most compact version of the dataset with an higher prevalence of fraudulent transactions. This choice was driven by the need to ensure speed in both the analysis and execution of the model.

For greater organisation, we grouped all the datasets in a folder called 'datasets', making it easier to select the desired dataset. We have also created two additional folders: one dedicated to storing the intermediate results obtained from Spark processing, and the other to store the post-processed training, validation and test sets.

3 Data analysis and Pre-processing

Dealing with and managing large datasets while maintaining scalability was one of the main challenges of this project. To solve this problem, we opted for the use of Spark. This choice allowed us to avoid the complete loading of the dataset into memory, facilitating distributed operations.

With the help of Spark, we manipulated the data, calculated features and applied various transformations. For the analysis, we chose to work on the complete HI-Small dataset. This ensured a detailed understanding of the peculiarities of the data, while preserving the original proportions between the classes.

Initially, we analysed and pre-processed the entire dataset, calculating additional features and examining the correlation between the data through a correlation matrix and some intermediate visualisations. Subsequently, we split the dataset and recalculated the features for each subset, making sure to avoid any 'data leakage' between the different sets in order to obtain as accurate an evaluation as possible.

The division of the dataset was based on the recommendations on the Kaggle site: 60% of the data for training, 20% for validation and 20% for testing.

3.1 Data Analisys

Initially, we read the dataset in CSV format using Spark. We opted for a well-defined schema to represent the data in order to simplify future analyses and feature calculations. This schema is detailed in Table 1.

Field	**Type**
timestamp	StringType
from_bank	IntegerType
from_account	StringType
to_bank	IntegerType
to_account	StringType
amount_received	FloatType
receiving_currency	StringType
amount_paid	FloatType
payment_currency	StringType
payment_format	StringType
is_laundering	IntegerType

Table 1. Dataset Structure

The data analysis was divided into two main phases. Initially, a study was conducted that disregarded the temporal sequence, focusing on the structural analysis of the dataset, the relationship between laundering transactions and legitimate ones, and the identification of possible distinctive differences between the characteristics of both classes. Subsequently, after calculating some additional features related to the temporality of transactions, a correlation matrix was shown to obtain an overall overview of the interrelationships between different features.

The analyses carried out on the dataset are as follows:

Handling of Null Values: We searched for cells with null values within the dataset, but none were found.

Class Balancing: We checked the balance of classes in the dataset and found a strong imbalance with a ratio of approximately 1:1000, meaning one fraudulent transaction for every thousand.

Payment Methods in Relation to Fraudulent Transactions: We analyzed whether certain payment methods were more correlated with fraudulent actions. It turned out that out of seven payment methods (ACH, Bitcoin, Cash, Cheque, Credit Card, Reinvestment, Wire), the ACH type stood out in terms of ratio. Furthermore, these analyses demonstrated that the Reinvestment and Wire types were not associated with any fraudulent transactions.

Payment Currency in Relation to Fraudulent Transactions: The analyzed dataset includes fifteen different currency types used for payments. The most frequently used is the US Dollar. The most fraudulent, based on the ratio of transactions to their fraudulent nature, is the Saudi Riyal with a ratio of 4:1000, followed by Euro and US Dollar, both with a ratio of about 1:1000.

Analysis of Correlation between Amount and Fraudulent Transaction: This analysis aimed to determine if there was any correlation between the transaction amount and the likelihood of it being fraudulent. The result showed that the transaction amounts were not significantly meaningful. In fact, the amounts associated with fraudulent transactions fell within the maximum and minimum amounts of non-fraudulent transactions. Furthermore, with a ratio of approximately 1:1000, the data is even less significant. This analysis was useful in deciding not to consider the amount and related features as important.

Attribute Concordance Analysis: We examined the correspondence of various attributes in transactions, including source and destination accounts, issuing and receiving banks, currency, and transaction amounts. For each pair of matching attributes, a specific binary feature was created. The analysis revealed that the correspondence of currency and amount, both in sending and receiving, had little relevance in differentiating the two classes.

Subsequently, additional attributes were generated based on the timestamps of transactions. Among these, we find:

Number of received transactions: Three distinct features have been calculated, one for each time period (day, hour, minute).

Number of sent transactions: Similar to received transactions but considering those sent.

Minutes elapsed since the last transaction: The number of minutes elapsed since the last transaction is calculated. If there are no prior transactions, the default value is set to -1. Minutes were chosen as they represent the smallest unit of time in the timestamp.

Transactions sent to a single account in the last day: The number of different accounts to which the account executing the transaction has sent money in the last day is calculated.

Transactions sent to a single bank in the last day: Similar to the above, but focused on banks.

Payment methods used in the last day: The count of how many different payment methods were used for transactions in the last day.

Types of currency used in the last day: Similar to payment methods, but this time focused on the different currencies used in transactions.

Transactions that have been executed to the same account: All transactions that have occurred to the same account throughout the entire time period are considered.

A visual representation of the correlation matrix of the entire dataset was created, which is presented in Figure 1 as shown in the image below.

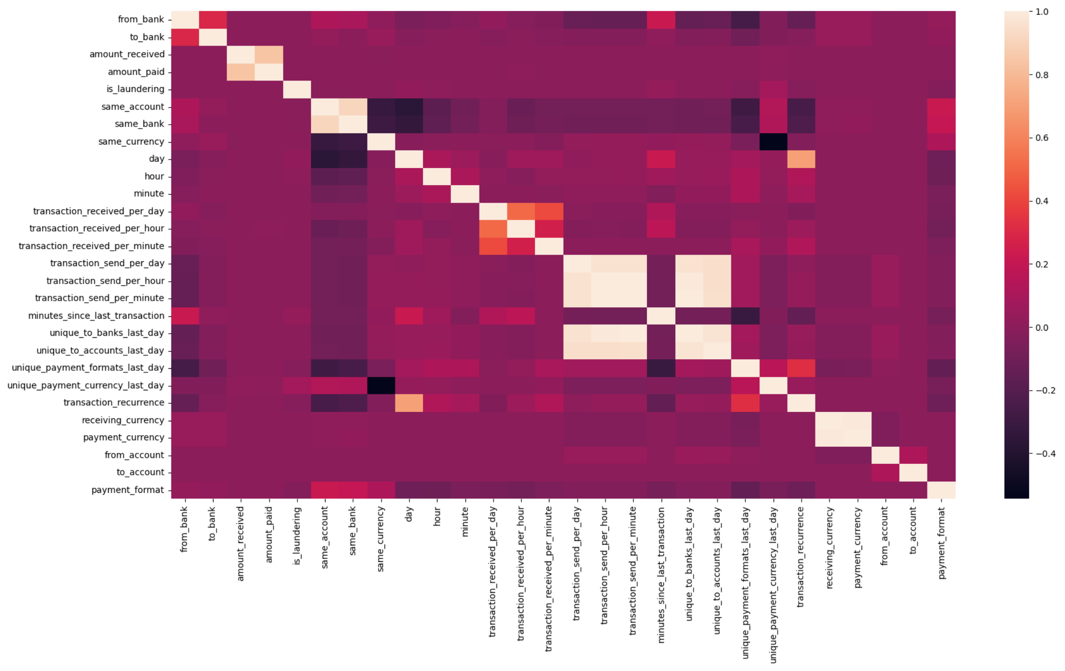

Fig. 1. Correlation matrix HI-Small dataset.

3.2 Pre-processing

The preprocessing phase can be broken down into three main steps:

- Dividing the dataset into training, validation, and test sets.
- Extracting features from each data set.
- Reducing features using a feature selection algorithm.

Before dividing the dataset into its three components, we sort it based on the timestamp. Indeed, it was chosen to follow a chronological partitioning approach to the data to ensure that the temporal features retained their intrinsic meaning. This is especially important since many of our features were computed with the evolution of the data over time in mind. By adopting a random split, we would have mixed data from different periods, compromising the temporal consistency of the values. Instead, by splitting the data in chronological order, we ensure that each set contains consecutive data over time, thus preserving the validity of the computed temporal features.

In addition to sorting the data chronologically, we performed label encoding on the categorical features.

3.2.1 Dataset split The dataset was divided in the following ways: 60% for training, 20% for validation and 20% for testing. This proportion was adopted following the recommendations on the

Kaggle page for the database. The time-based division resulted in a balance in the three datasets that differed from the actual distribution. Specifically, we recorded ratios of approximately 0.7:1,000 in the training set, 1:1,000 in the validation set, and 1.7:1,000 in the test set.

3.2.2 Feature extraction After splitting the dataset, we extracted additional features for each subset, resulting in a total of 22 features for each set. Of these, one represents the label related to the class. The importance of having split the dataset before feature extraction lies in the prevention of any "data leakages" between datasets. This ensures a clear and effective separation between the different subsets of data.

3.2.3 Feature selection We introduced this step to reduce the number of features, eliminating those that might not add significant value to a potential learning model. We adopted Boruta[3] as the feature selection tool. Boruta is an algorithm based on the Random Forest model. It identifies and preserves essential features while eliminating irrelevant or redundant ones by comparing them with randomly generated "shadow features." The choice of Boruta is motivated mainly by two factors: first, it is based on a Random Forest, the same model we intend to use for predictions. Second, the Random Forest has the ability to balance the weight of classes, a crucial aspect given the pronounced imbalance in our dataset and the difficulty of rebalancing classes due to the artificial nature of the data.

Since Boruta is not compatible with Spark, we opted to train it on only 10% of the training set. Although this may somewhat compromise the algorithm's ability to identify features, this decision allowed us to speed up the process and ensure smooth loading of data into memory. For Boruta, we set an early stopping criterion after 50 iterations and let the number of estimators be determined automatically. However, because of the marked imbalance in the data and the limited sample size, the Random Forest was configured to use only 10 estimators.

At the end of its process, Boruta identifies features deemed important. Subsequently, these features are selected for each of the three datasets and saved in parquet format. It is essential to note that the initial segmentation of the data was done based solely on timestamps. Given the artificial nature of the data, many transactions are recorded at the exact same time. This feature, combined with the timestamp-based sorting, implies that the datasets may not always contain the same information. As a result, Boruta may not always identify the same features as relevant unless the index is also considered in the partitioning. However, it turns out that there are at least four features consistently judged irrelevant by Boruta: `receiving_currency`, `payment_currency`, `same_currency`, and `transaction_received_per_minute`. This suggests that these features certainly do not add significant value to the model.

4 ML Models

In this project, we developed two different learning algorithms from scratch. Initially, we implemented a classifier based on decision trees, also optimizing its hyperparameters. Next, we implemented a random forest, leveraging Spark to parallelize the construction of decision trees on various workers. For all models, the seed was uniformly set to 0.

4.1 Decision Tree Classifier

The decision tree classifier was developed by taking as reference some methodologies from online sources[4][5]. This approach allowed us to ensure proper compatibility with scikit-learn libraries and, subsequently, to customize the model according to the specific needs of our project.

The implemented attributes are:

- **random_state**: Seed for the random number generator and to allow reproducibility of experiments.
- **max_depth**: Maximum depth of the tree. If None, the tree grows until all leaves are pure or contain fewer samples than *min_sample_split*.
- **min_sample_split**: Minimum number of samples needed to divide an inner node.
- **criterion**: Function to measure the quality of a split. Supports "gini" and "entropy".
- **min_info_gain**: Minimum information gain required to split an internal node.
- **max_features**: Number of maximum features to be considered at each split.
- **max_thresholds**: Maximum number of thresholds to be considered for each feature.
- **class_weights**: Weights associated with classes in the form of {class_label: weight}.

The main methods available in our implementation of the decision tree and that can be invoked by the user are:

- **fit(X, y)**: This method fits the decision tree to the training data. It takes as input features, represented by `X`, and labels, represented by `y`. At runtime, it initializes and starts the tree creation procedure.
- **predict(X)**: Once the tree is trained, this method is used to predict the labels of new data. It traverses the tree recursively until it reaches a leaf node, returning the predicted label.
- **calculate_metrics(y_true, y_pred)**: After making predictions, this method calculates essential metrics such as accuracy, recall, f1-score based on comparing true (y_true) and predicted (y_pred) labels.
- **score(X, y)**: Provides a quick evaluation of the model, returning a specific metric (by default, the F1-score) based on test data and its labels.
- **print_tree()**: For visual understanding, this method allows you to visualize the structure of the tree, showing the chosen features and their respective thresholds at each node.

4.2 Information Gain

The essence of a decision tree classifier lies in its ability to extract relevant information whenever an internal node divides into subnodes. This subdivision process is critical in determining how decisions are made within the tree. In the version we implemented, we adopted two basic criteria to guide the division of nodes: Gini and Entropy. The decision on which criterion to use is influenced by the specific nature of the data and its distribution. Each criterion has its own peculiarities and advantages, and the optimal choice may vary depending on the needs of the analysis in question.

Given a target data set y, we define C as the set of unique classes in y. The probability $p(y_i)$ of a specific class y_i in C is given by:

$$p(y_i) = \frac{\text{Number of occurrences of } y_i}{\text{Total samples of } y}$$

Where i is an index representing each unique class in C.

In the function `__calculate_split_entropy`, we go to calculate the unique values and their occurrences, then the probabilities of each class are determined. If there are class weights, provided through the dictionary `class_weights`, these are applied to the class probabilities. If a unique value has no associated weight in the dictionary, it is assigned a default weight of 1.0. Defining $w(y_i)$ as the weight associated with class y_i, the weighted probability $p_w(y_i)$ is given by:

$$p_w(y_i) = p(y_i) \times w(y_i)$$

With these probabilities (weighted or unweighted), we calculate the division criteria as follows:

- **Gini**:
$$G(y) = 1 - \sum_{i \in C} p_w(y_i)^2$$
- **Entropy**:
$$E(y) = -\sum_{i \in C} p_w(y_i) \log_2(p_w(y_i) + \epsilon)$$
  where ϵ is a small constant to prevent the logarithm of zero, set at $\epsilon = 1 \times 10^{-10}$.

4.3 Node structure

The nodes in the tree can be classified into two types:*leaf nodes* and *internal nodes*. In our implementation, these nodes are represented by dictionaries. The keys vary according to the node type.

An **internal node** has the following structure:

```
{
    'splitting_threshold': Threshold for splitting,
    'splitting_feature': Feature of splitting,
    'info_gain': Information gain,
    'left_child': Left child node,
    'right_child': Right child node.
}
```

While a **leaf node** is represented as:

```
{
    'label': Majority class,
    'samples': Number of samples
}
```

It is important to note that the leaf node takes as its label the majority class present within it.

4.3.1 Compare with scikit-learn To evaluate the effectiveness of our decision tree, we conducted a test using the 'breast cancer' dataset available in the scikit-learn library. We compared the performance of our tree with that of the decision tree provided directly by scikit-learn, finding a correspondence in behavior between the two.

4.4 Hyperparameter tuning

A decision tree, like many other machine learning models, needs hyperparameter tuning to ensure optimal performance. Without this tuning, the results may not be the best possible. To handle this optimization step, we have chosen to use Hyperopt[6], a library dedicated to tuning hyperparameters.

Hyperopt adopts a Bayesian search approach to navigate the space of hyperparameters. Instead of testing every possible combination of hyperparameters as in GridSearch, or selecting combinations at random as in Randomized GridSearch, Hyperopt tries to find the combination that minimizes a given objective function, usually the error of a model on a validation set, using a process guided by prior information.

The main advantage of Hyperopt over methods such as GridSearch or Randomized GridSearch is efficiency. While GridSearch can become very computationally expensive when the space of hyperparameters is large, and Randomized GridSearch may not explore the space optimally, Hyperopt uses information from previous iterations to guide the search toward combinations of hyperparameters that are more promising.

Hyperparameter tuning was conducted using 50% of the training and validation data in order to speed up the process. In configuring the Hyperopt parameters, we set the number of iterations to 50. This choice was guided by two considerations: first, to reduce the algorithm's execution time; second, to prevent overfitting that might occur with an excessive number of iterations.

To ensure a balanced treatment of classes during the training phase, we used the parameter `class_weights`. Specifically, we set `class_weights` to reflect the inverse distribution of classes in the training dataset.

Finally, after a series of preliminary tests, we identified the following ranges of hyperparameters in which the model tends to produce the best score:

```
{
    'max_depth': [10, 11, 12, 13, 14],
    'min_sample_split': [10, 11, 12, 13, 14],
    'criterion': ['gini', 'entropy'],
    'max_features': ['sqrt', 'log2'],
    'max_thresholds': [10, 11, 12, 13, 14]
}
```

In the process of tuning the hyperparameters, using the F1-score as an evaluation metric, we identified the following values as optimal:

criterion: entropy
max_depth: 11
max_features: sqrt
max_thresholds: 13
min_sample_split: 11

4.5 Train and decision tree test

After the hyperparameters were selected, the model was trained and, later, tested on the test set also at 50%. The results showed an `accuracy` of 0.9987, a `recall` of 0.4575, a `precision` of 0.7537 and a `F1-score` of 0.5694. It can be seen that the accuracy is particularly high. This is attributable to the marked imbalance in the dataset. The confusion matrix, which provides further details, is presented in Figure 2.

It is important to note that these results may vary depending on several factors. As mentioned in the 3.2.3 section, the initial division of the data into the various sets may affect the results. Also, the number of iterations set in hyperopt could, if modified or re-run, lead to the discovery of better or worse combinations of hyperparameters, this is because it does not perform all possible combinations.

4.6 Random Forest

Regarding the implementation of the Random Forest from scratch, we leveraged the parallelization capabilities provided by Spark. This parallelization was applied both during the training phase and the testing phase.

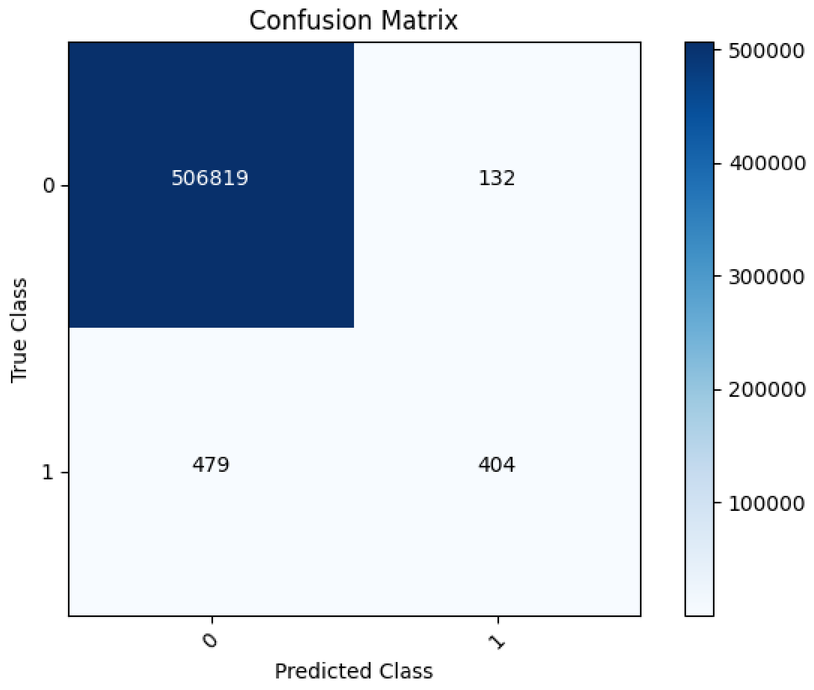

Fig. 2. Confusion matrix.

Concerning the class structure, it retains the same methods as the Decision Tree Classifier that can be invoked by the user. The main distinction lies in the `fit` and `predict` methods: both utilize a single parameter, `X`, which corresponds to the Spark dataframe. This dataframe includes a `features` column, which gathers the vectorized features using the `VectorAssembler` class from PySpark, and a `is_laundering` column, which denotes the labels associated with the corresponding feature vectors.

In comparison to the Decision Tree Classifier class, the Random Forest class introduces additional attributes, as follows:

- **n_estimators**: Represents the number of trees in the forest. The dataset is segmented into n parts, with n equivalent to the value of `n_estimators`. Each worker manages one or more segments of the dataset, on which it trains a specific decision tree.
- **bootstrap**: This boolean variable determines whether to bootstrap the dataset. When enabled, it allows the creation of different samples with replacement from the original dataset, providing greater variability and robustness to the model during training.
- **features_col**: Specifies the column intended to represent the feature vector.
- **label_col**: Specifies the column designated for labels.

In addition, the `class_weights` attribute has been removed since it is handled automatically within each worker during execution. This change was made because each worker receives a portion of the dataset and, therefore, is responsible for balancing the class weights of the single portion.

4.7 Training phase

Unlike the decision tree-based classifier, tuning of hyperparameters was not performed for the Random Forest. The selected hyperparameters were set to a value of 10 for the `max_thresholds` parameter and to 'sqrt' for the `max_features` parameter. Setting `max_thresholds` to 10 was chosen to reduce the complexity of the model, speed up the training time and reduce the risk of overfitting.

As for the `n_estimators` parameter, it was deliberately set to 20. This choice was motivated by the need to prevent trees trained on subsets of the dataframe from containing insufficient amounts of minority class data. This precaution was taken because of the high imbalance between the classes; by greatly increasing the number of estimators we would have risked generating trees based on a very limited amount of data from the minority class, thus compromising the prediction capability for that class. Finally, dataframe bootstrapping was employed.

For training the Random Forest model, unlike the Decision Tree Classifier, the entire training dataset was employed. This choice was driven by the ability to parallelize the training process, which not only makes the model more scalable but also significantly speeds up the training phase.

The trees generated by each worker were stored in a model array within the Random Forest class in order to later employ them for predictions.

4.8 Testing phase

For the testing phase, the entire test dataset was used. Since at this stage it is not necessary to generate new trees but only to make predictions, it was possible to increase the number of estimators used, thus setting it at 100. The results of the predictions on the test dataset produced the following values: `Accuracy`: 0.9987, `Recall`: 0.4407, `Precision`: 0.7593, `F1-score`: 0.5577. The confusion matrix is presented in Figure 3.

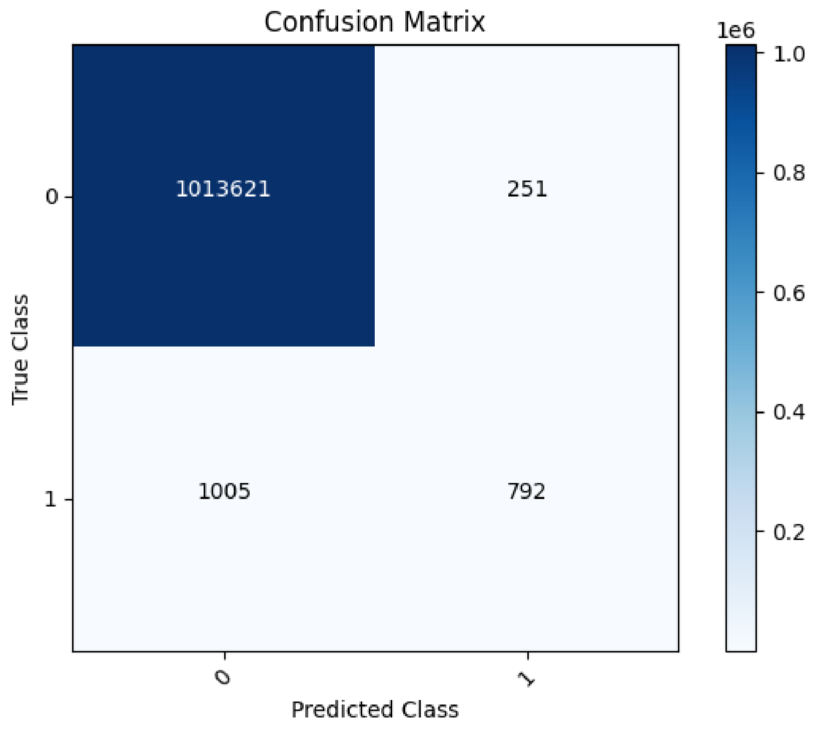

Fig. 3. Confusion Matrix.

It is important to note that the `F1-score` obtained was lower than that obtained using the Decision Tree Classifier. This discrepancy can be attributed to the fact that, unlike the Decision Tree Classifier, no tuning of the hyperparameters was performed for the Random Forest model. Similarly, it is evident that the Random Forest, without hyperparameter tuning, shows an ability to achieve performance comparable to that of the Decision Tree Classifier when the latter was subjected to hyperparameter tuning.

5 Scaling

Regarding the scalability of the proposed solution, the use of Spark allows processing large amounts of data in a parallel manner and, most importantly, avoids the need to load all data directly into the main memory of a single node. In the case where a large dataset is to be processed, such as the HI-Large dataset, it would be necessary to enhance the memory capacity in the various workers and, ideally, increase the number of available workers.

6 Conclusions and final considerations

In conclusion, the approach taken reported acceptable results. This is especially notable considering the highly unbalanced nature of the initial dataset and the artificial nature of the data itself. During the course of the project, we chose not to use the provided file containing the fraudulent patterns among the data. However, a possible future development could be to integrate this file into our analytical methods. We could also consider transactions as nodes in a graph, thus allowing the computation of additional features and a more complete view of the problem.

7 Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

References

1. IBM Transactions for Anti Money Laundering (AML). (link)
2. Altman, E., Egressy, B., Blanuša, J., & Atasu, K. (2023). Realistic Synthetic Financial Transactions for Anti-Money Laundering Models. arXiv preprint arXiv:2306.16424. (link)
3. Kursa, M. B., & Rudnicki, W. R. (2010). Boruta - A System for Feature Selection. Fundamenta Informaticae, 101(4), 271-285. DOI: 10.3233/FI-2010-288. (link)
4. How to program a decision tree in Python from 0 (link)
5. Master Machine Learning: Decision Trees From Scratch With Python (link)
6. Bergstra, J., Yamins, D., & Cox, D. D. (2013). Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. Proceedings of the 30th International Conference on Machine Learning (ICML 2013), I-115 to I-23. (link)