ALDER: Anti Money Laundering Using Decision Trees Methods

1nd Renato Avellar Nobre
Computer Science Department
Università degli Studi di Milano
Courses "AMD & SMML" - Exam Project, A.Y. 22/23

***Index Terms*—big data, decision trees, random forest, binary classification, money laundering**

I. Introduction

Money laundering is a multi-billion dollar issue involving money generated from crime deeply inserted into the financial system in a way untraceable by the authorities [1]. Detecting those laundering transactions is of tremendous public interest and is consequently given wide publicity [2]. A wide range of national and international agencies have attempted to quantify and detect organized crime and components of money laundering [2]. Recently, a significant effort has been employed to use machine learning techniques to identify such laundering transactions [3], [4].

However, detecting laundering is a challenging task [1]. The first challenge is that the data is highly unbalanced, and most automated algorithms have many legitimate transactions incorrectly flagged as laundering and undetected laundering transactions [1]. Another major challenge is the tremendous amount of transactions being made in the world daily. In Brazil in 2021, approximately 327.4 million transactions were generated every day [5]. This quantity of data enters the realm of big data and needs appropriate techniques and computational power to be handled properly [6].

With the aforementioned in mind, this project proposes and explores ALDER, an implementation of Decision Trees (DT) and Random Forest (RF) algorithms to detect money laundering. It is designed to classify laundering transactions and can process large-scale data in a simulated distributed environment. ALDER utilizes the IBM Transactions for Anti Money Laundering (IBM TAML) dataset [1] and leverages Apache Spark for scalable computing. It can either use a single DT or distribute the computation of multiple DTs to create a scalable random forest predictor. Therefore, ALDER aims to address research questions related to its effectiveness in classifying money laundering, its performance with different dataset sizes, and its scalability in real distributed environments.

Over many contributions, ALDER applies a rigorous data engineering pipeline with a strong exploratory data analysis. Such combinations allowed ALDER to create robust features able to generalize the content of the data without creating data leakages to potential overfitting. Additionally, ALDER does not rely on external libraries for its DT nor its random forest, designing a DT from scratch with mechanisms to work with data imbalance, big data thresholds and multiple information gain functions. Finally, ALDER applies a state-of-the-art hyperparameter tuning with Bayesian optimization and the Hyperopt library [7].

Results indicated that the ALDER DT can learn well even in the simulated distributed environment with large amounts of data. Its models strongly prefer precision, without significant false alarms on non-laundering transactions. However, such strength is absent in the recall metric, allowing multiple fraudulent transactions to go undetected. Overall the best model achieved an F1-Score of 0.5990 on test data, while the best distributed version stood at 0.5748.

The remainder of this work is organized as follows. Section II describes the methodology pipeline, starting from describing the used dataset, fetching the data, considering preprocessing and exploratory analysis, and the decision tree and random forest model development. Section III explains the methodology used to validate the proposed solution, considering the hyperparameter tuning methodology, employed metrics to validate the binary classification and presents the results obtained from the experimental procedure both in a "small" and "big" data scenario. Finally, Section IV presents the main conclusions and opportunities for future research.

II. Methodology

This section describes ALDER[1] (**A**nti Money **L**aundering using **DE**cision T**R**ees), a "from scratch" implementation of the decision trees algorithms to classify money laundering, design to work also on large-scale data processing in a distributed environment. For that, ALDER takes advantage of the IBM Transactions for Anti Money Laundering (IBM TAML) [1], a publicly available dataset from the Kaggle[2] website and the power of Apache Spark, the most widely-used engine for scalable computing [8].

Considering this approach, ALDER can use a single constructed decision tree to identify money laundering transactions with traditional processing techniques or leverage the power of Apache Spark to distribute the computation of multiple decision trees and create a scalable random forest predictor for large datasets. Therefore, with all the proposed capabilities and technologies, ALDER was designed to evaluate the following Research Questions:

[1]Available at https://github.com/Skalwalker/AntiMoneyLaundering
[2]Available at https://www.kaggle.com/datasets/ealtman2019/ibm-transactions-for-anti-money-laundering-aml

- **RQ1** - Can ALDER learn a Decision Tree constructed "from scratch" to classify money laundering without significant false alarms or misses?
- **RQ2** - Can ALDER learn a Random Forest of the proposed Decision Trees on a simulated distributed environment with small (1GB) and medium (4GB) large-scale datasets?
- **RQ3** - Does the ALDER's distributed Random Forest outperform the single decision tree approach for the small dataset?
- **RQ4** - Does increasing data size from the small to the medium dataset increase ALDER's distributed Random Forest performance?
- **RQ5** - Considering a real distributed environment and a larger data scale, how would ALDER scale in such scenarios?

A. Overview

Figure 1 shows an overview of the proposed ALDER methodology for detecting money laundering using decision tree methods. ALDER starts by downloading the IBM TAML dataset from the Kaggle website. Among multiple files available in the dataset, ALDER selects a subset (further described in Subsection II-C) for its methodology. The original dataset files are in CSV format and large enough to have memory issues while loading them in primary memory. Therefore, to be able to process all the selected data and further parallelize the distribution of decision trees in the random forest, ALDER relies on PySpark.

PySpark is the Python API for Apache Spark, enabling real-time, large-scale data processing in a distributed environment using Python. From opening the CSV files to creating the Pandas input version of the data for the model training, ALDER utilizes the PySpark framework, as shown in Figure 1. Starting from the data engineering pipeline, ALDER utilizes common data engineering thinking based on cookie-cutter framework [9] to process the data from its raw format to model inputs. The data engineering pipeline performs a series of data visualization, cleaning and manipulation tasks, including exploratory data analysis, feature engineering and categorical encoding. After the pipeline is executed, ALDER generates four parquet datasets ready for training a model: train/test small and train/test medium, split with an 80%-20% ratio. We can use the small one from these datasets to convert it whole (or a fraction) to pandas and train a single decision tree model.

On the other hand, to train the distributed random forest, ALDER performs further processing with the input data. Especially, ALDER breaks down the dataset by performing rows and columns sub-sampling the available data, thus creating multiple subsets. These subsets are distributed to a random forest that consists of multiple workers, and each worker is responsible for loading the data into the main memory and training a single decision tree. After every worker finishes training, the decision tree model is ready for use.

B. Dataset

The idea behind the ALDER methodology is to use decision trees and random forests to detect money laundering transactions. Especially for the proposed methodology, we consider TAML, a dataset of fraudulent transactions proposed by IBM [1] and publicly available on Kaggle. The IBM dataset generates synthetic money laundering data from a virtual world where individuals, companies, and banks interact with each other through various financial transactions. The transactions include purchases, payments, and loans, which are conducted through different types of accounts. A small fraction of the entities in this virtual world engage in criminal activities, such as smuggling and illegal gambling. These criminals attempt to hide the illicit funds through a series of financial transactions known as money laundering. The simulation covers the entire money laundering cycle, including the placement of illicit funds, the layering of funds into the financial system, and the integrating of those funds through spending.

Thanks to the usage of synthetic data, the resulting data can be labelled and therefore used for training and testing Anti Money Laundering (AML) models. Additionally, unlike real-world data, limited to transactions involving a specific institution, the synthetic data in this virtual world represents a complete financial ecosystem. This allows for the creation of laundering detection models that can understand transactions across different institutions but apply their inferences specifically to transactions at a particular bank.

The IBM TAML dataset consists of 6 different CSV files, divided into two main groups: Higher illicit (HI) and Lower illicit (LI) ratio. HI and LI have three datasets: small, medium, and large, where each dataset is separate and independent from the others. The purpose of having different datasets is to support a broad degree of modelling and computational resources.

C. Considered Datasets

Considering the proposed research questions and the technical limitations, ALDER is validated only on a subset of all available data (see Fig. 1 again). Specially, we consider only the higher illicit small and medium datasets.

This restricted selection is largely motivated by the restrictions of the Colab environment processing, main and secondary memory. Additionally, since Colab is executed into a temporary virtual machine [10], all the persisted files are lost after some time. These combinations of restrictions make the environment inhospitable for common data engineering convention [11], where a pipeline is broken up and stored into different layers according to how data is processed. Therefore Colab is not suitable to work in a big data environment; data engineering tasks are usually made in a cookie-cutter structure [12], where the layers of data processing and storage would allow to reduce the cost-intensive data processing tasks by storing intermediate representations for quickly recovering in latter use [11].

A second set of motivations for selecting only HI small/medium datasets are devised from the data analysis. The small dataset consists of approximately 1GB of data, and the medium dataset of 4 GB. Considering the Colab environment of 12GB RAM, the 4GB dataset already has issues while loading, requiring more primary memory and therefore can be

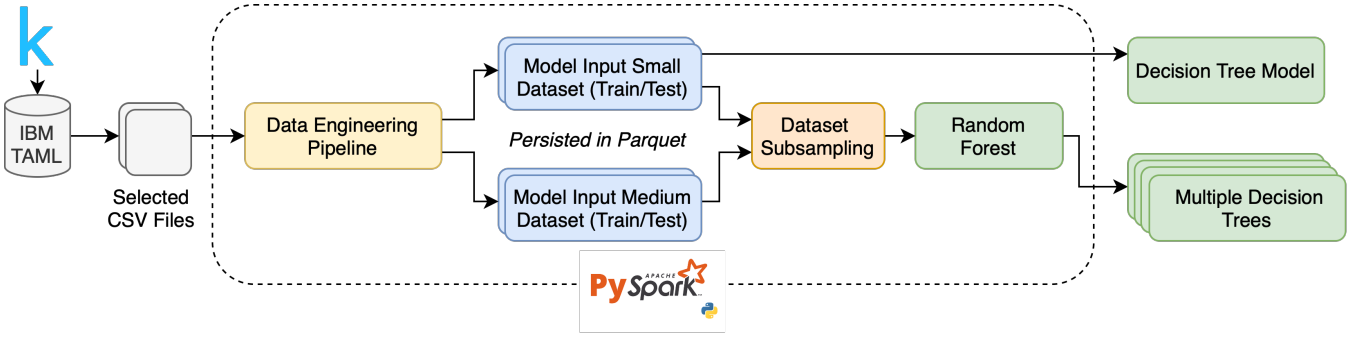

Fig. 1: Overview of the proposed methodology

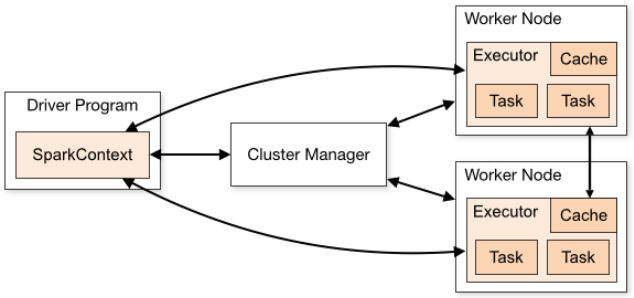

Fig. 2: Apache spark overall architecture (Source: Apache Spark Website [8])

considered a "big" data problem for the proposed environment. Additionally, considering the class imbalance for both data sets are approximately one laundering per 950 transactions, which is already significantly imbalanced, considering the lower illicit datasets would approximate the problem to an anomaly detection problem where other solutions, such as an auto-encoder, would be better suited [13].

With all that in mind, ALDER has experimented with only the HI small dataset to train the sequential decision tree model and the HI small and medium to execute the random forest experiments on distributed workers. Finally, since we only consider HI data, we refer to these datasets only by small and medium.

D. Apache Spark and PySpark

ALDER's first encounter with large-scale techniques starts with reading the CSV files and executing the data engineering pipeline. At this point, ALDER utilizes PySpark technology to perform the desired computations. However, before entering in detail the data pipeline, it is worth discussing some considerations of the Spark usage in ALDER.

Apache Spark is designed to efficiently process and analyze large-scale distributed data across a cluster of workers. Overall, its underlying mechanism combines data parallelism, in-memory processing, fault tolerance, and optimization techniques [8]. Its underlying mechanism involves several key components:

- **Resilient Distributed Datasets (RDDs):** RDDs are the fundamental data structure in Spark. They can be processed in parallel across a cluster, allowing in-memory computations that can be cached to optimize performance [14].
- **Cluster Manager:** The cluster manager allocates resources and manages the execution of Spark applications across a cluster. Spark leverages distributed computing techniques, such as data partitioning, parallel processing, and distributed task allocation, to achieve faster and parallel execution of computations across the cluster. By distributing both data and processing tasks, Spark enables efficient parallel data processing [14].
- **DAG Scheduler:** Spark applications consist of a series of transformations and actions on RDDs. The DAG scheduler breaks down these operations into a logical execution plan where each stage represents a set of transformations that can be executed in parallel [14].
- **Task Scheduler:** Takes the stages generated by the DAG scheduler and assigns tasks to individual workers nodes in the cluster. It considers data locality to schedule tasks closer to where the data resides, reducing network overhead [14].

As demonstrated in Figure 2 the SparkContext is responsible for coordinating the execution of tasks across the cluster, managing the distributed data sets, and handling the communication with the cluster managers. Therefore the SparkContext is considered an entry point for the low-level functionality in PySpark, primarily focused on RDD operations [8]. However, with the introduction of Spark 2.0, a new layer of functionality was introduced: the SparkSession, a higher-level API that provides mechanisms for working with structured data and DataFrames.

ALDER's solutions utilize the SparkSession API and the dataframe structured data. The SparkSession is preferred over the RDDs when working with structured data [8]. Dataframes generally perform better than RDDs, generating optimized execution plans based on the operations, leading to faster and more efficient data processing. Besides, data frames provide a higher-level API with multiple functions for data manipulation, thus being more user-friendly and expressive compared to RDDs transformations (e.g. map, reduce) [8]. Finally, even though RDDs offer more control over data distribution, optimizing the data partitioning and distribution depends on the user implementation, while data frames provide automatic data distribution optimizations [8].

With the basic underlying mechanisms of PySpark briefly

explained, we can dive deep into the concepts used for setting up PySpark in ALDER. We initialize and configure a spark session before opening and preprocessing the data files. The following list is a selected subset of the spark configurations used in ALDER's code to initialize the spark session considering the available environment capabilities. Especially the values are optimized to load a 4GB dataset on the considered colab environment with a 2-core CPU and 12GB primary memory.

- **Driver Memory (3GB):** This property specifies the amount of memory allocated for the Spark driver, which is responsible for coordinating the Spark application [14].
- **Executor Instances (2):** With a 2-core CPU, we can have a maximum of two executor instances. Setting this property to 2 will utilize both cores for parallel processing [14].
- **Executor Memory (4GB):** This property sets the amount of memory allocated for each executor [14]. Since we have roughly 12GB available in the Colab and 3 GB are set for the driver, each executor will have 4GB available, leaving 1GB for free as a margin.
- **Executor Cores (1):** This property determines the number of CPU cores assigned to each worker [8]. It is generally recommended to set this value to the number of available CPU cores on the machine [14]. We assign one core peer worker since we only have two cores and two workers.
- **Shuffle Partitions (2):** This property sets the number of partitions to be used for shuffling data during Spark SQL operations, which is a costly operation in terms of memory and CPU usage [8]. Since we have limited resources, we keep the number of shuffle partitions low.

E. Data-preprocessing & Feature Engineering

Figure 3 shows an overview of the data engineering pipeline, from reading the considered files to splitting and saving the final formats. For this specific pipeline, we consider abstract representations of the cookie-cutter [12] data layers for data engineering [11], which means that we are following only the cookie-cutter logical reasoning of data organization rather than persisting the data for each layer.

First, we start with the *Raw Data Layer*. The raw layer is the starting point of the data pipeline and includes the sourced data that should never be altered [11]. It serves as the single source of truth for all subsequent work [11]. Typically, these data models are untyped, such as CSV files. Recalling Subsection II-C we are considering two CSV data files, small and large. Those CSVs will not be altered and will remain in the disk as our source of truth. We start the pipeline in the raw layer by verifying the columns, checking if they are the same for both datasets and checking for null values. Since the data does not contain null values and the columns are the same, we start the processing in the intermediate layer by fixing its schema.

In practice, the *Intermediate Data Layer* only needs to be a "typed mirror" of the raw layer within the source data model [11]. Once the intermediate layer exists, we never have to touch the raw layer, and we eliminate the risks associated

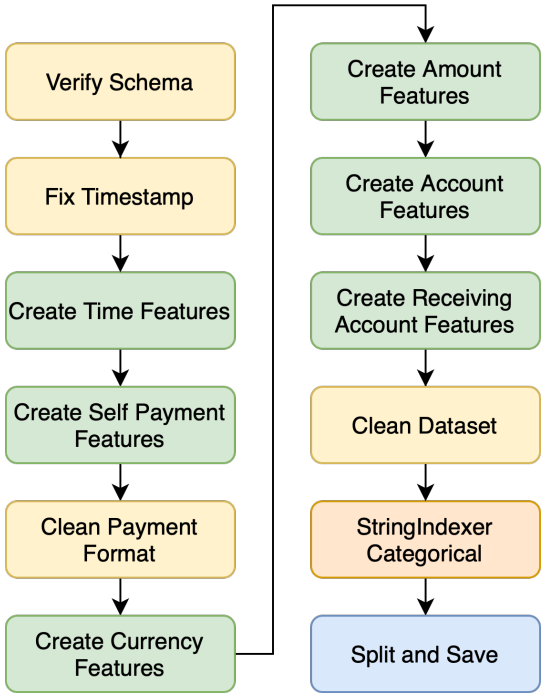

Fig. 3: Data Pipeline

with mutating the original data [12]. Cleaning column names, parsing dates and dropping completely null columns are other transformations commonly performed at this stage. Also, profiling, an Exploratory Data Analysis (EDA) and any data quality assessments should be performed at this point [11]. In Figure 3, the first and second steps reside in the intermediate layer. The first transformation we do with our data is to apply a proper schema to the dataframe, thus creating the "typed mirror" of our data. Following we fix the timestamp column to process it as a proper timestamp data format.

With the data properly typed, we can now perform an EDA to generate insights on the data format and thus make informed decisions on our feature engineering process. We analyzed only the small dataset for simplicity and time-saving and extrapolated its insights to the medium. The first step of the EDA process was to split the dataset into legit and laundering transactions to analyze the features in each case. Visualizing the "Amount Received" and "Amount Paid" features, we noticed that the statistics do not change significantly between one another, realizing that the only change between those values is when the payment and receive are made from different currencies. We also analyzed that most of the laundering occurs with the "ACH" payment format and that there is no laundering in "Reinvestment". Further investigation also showed that laundering is rarely made in a transaction where the origin and destination accounts are the same and never in transactions with different currencies. This comprehensive analysis also showed that numerous accounts are linked with fraudulent transactions and can happen over multiple currencies.

Following the EDA, we deeply studied how accounts were related to money laundering. Unusually high transaction activ-

ity or many accounts linked to a particular entity may indicate suspicious behaviour. From the analysis insights, we devised that it could be useful to create features that capture the frequency and volume of transactions from both the "From Account" and "To Account". Most importantly, this analysis clarified that including the account identifiers as features would introduce data leakage because the model could learn to classify specific accounts instead of generalizing the account behaviour. Finally, the destination bank and the behaviour of destination accounts appear to have strong predictive power. Finally, the last step of the EDA was to extract insights from the time correlation of the laundering transactions. At this point is worth noticing that ALDER is not a time series model. We processed the features so the predictor could work with daily batches of transactions and did not consider which transaction was made first. Insights from the time-related analysis showed a higher ratio of illicit transactions on Saturday and Sunday.

The insights from the EDA allow the processing and creation of the *Feature Data Layer*. The feature layer comprises a data model with a collection of features defined based on the primary data [11]. In practical terms, this layer represents the independent variables, engineered features and the target variable, which will serve as the foundation for applying machine learning models [12]. Additionally, any data cleaning and new features are removed at this step [11]. Returning to Figure 3, we start creating the time-related features: Hour of the Day, Day of the Month, and Weekday (we also create intermediary features Minute and Date to assist further feature creation). Next, we create a self-payment Boolean feature indicating if the transaction's origin and final account are the same, and we clean the payment format to remove all the "Reinvestment" transactions. Further, we create a Boolean currency feature indicating if the transaction's origin and final currencies are the same, thus being able to drop the Payment Currency and keep only the Receiving for the model. We do a similar process for the amount features by creating a feature of received ratio and removing the amount paid feature.

At this point, we are starting the second node of the second column of Figure 3 and considering that we will be removing the account features from the final data, we needed to find ways of extracting the relationship of the transactions with the accounts. Therefore we create the following features related to each account: total transaction amount in hour/day, the number of the transaction in the hour/day, and the time since the last transaction. Similarly, we also create features to establish the relationship between the origin and destination account: total transaction to the same account in hour/day, number of the transaction to this account in the hour/day, time in minutes since the last transaction to this account, amount of transactions where the receiving account has been involved in the last hour/day.

The final step at the data feature layer is cleaning the unwanted features; therefore, we drop the Timestamp, the Origin and the Destination Account. Now we have a final feature layer dataset and can start the *Model Input Data Layer*. In the model input layer, we store data formats that should be ready to train a model [11]. Therefore the first step is to encode the categorical features. ALDER encode the features using PySpark String Indexer encoder, a label indexer that maps a string label column to a column of label indices. After encoding, all that is left to do is split between training and testing, which is done with an 80% 20% split, and saving in the disk in parquet. We decided to save the dataset at this point to process all of the transformations overhead in advance instead of mixing the cost of applying the transformations with the model training.

Figure 4 shows a correlation matrix of the created features and the label. A correlation matrix is a table that displays the correlation coefficients between features. It provides a way to examine the relationships and dependencies between variables in a dataset. Each cell in the matrix represents the correlation coefficient between two variables, indicating the strength and direction of their relationship [15]. The correlation coefficients can range from -1 to $+1$, where a coefficient of -1 indicates a perfect negative correlation, a coefficient of $+1$ indicates a perfect positive correlation and a coefficient of 0 indicates no correlation [15]. By analyzing the correlation matrix, we can identify variables strongly related to the label of a particular classification problem [15].

Notice that in Figure 4, some of the created features have a strong correlation, which may indicate that those features provide similar information and can affect the model's performance, potentially overfitting. However, since ALDER uses a random forest with feature sub-sampling, these information redundancies might prove useful for the problem. Looking at the correlation coefficients between each feature and the label, we notice that no feature highlights itself by correlating with the label. Most of the features have a weak positive correlation with the label. Remember that correlation coefficients measure linear relationships [15]. If the relationships between features and the label are non-linear, the correlation matrix may not capture them accurately [15].

F. Decision Trees

ALDER utilizes decision tree methods to predict money laundering in the created datasets. A decision tree is a flowchart-like tree structure where each internal node (non-leaf node) denotes a test on an attribute, each branch an outcome of the test, and each leaf a class label [15]. Predictions of a class in a decision tree occur by traversing the tree from its root (top-most node) to a leaf, where for each internal node, the data point being predicted will perform a test on its attribute and decide the path to take until a classification is made [15]. Among multiple devised algorithms for the creation of decision trees, the most common are the ID3 [16], C4.5 [17] and CART [18]. The ID3 algorithm, developed by Quinlan [16], introduced the concept of decision trees and information gain as a criterion for attribute selection. It focused on recursively partitioning the data based on the most informative attributes to create a decision tree [15]. C4.5 is an extension of the ID3 algorithm also proposed by Quinlan [16] that introduced several improvements and new features, including handling missing values, handling continuous attributes, pruning decision trees to prevent overfitting, and incorporating cost-based

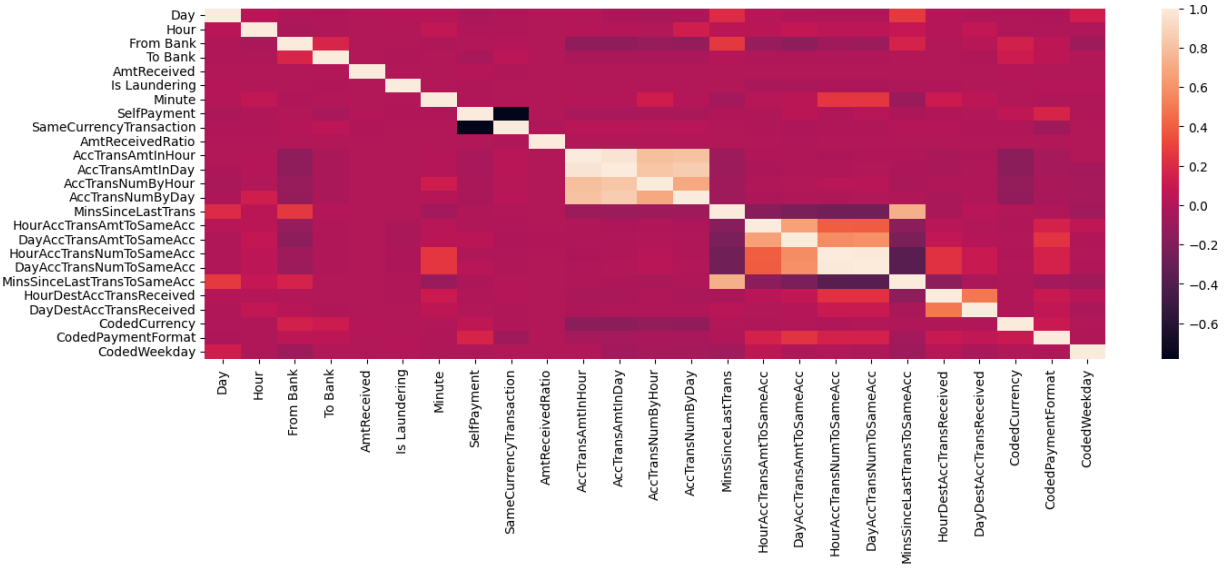

Fig. 4: Correlation Matrix

learning [15]. Finally, the CART algorithm, developed by Leo Breiman, Jerome Friedman, Richard Olshen, and Charles Stone [18], presents a framework for building decision trees that can be used for both classification and regression tasks. The paper describes the methodology of constructing binary recursive partitioning trees using Gini impurity as the criterion for attribute selection.

ID3, C4.5 and CART adopt a greedy approach in which decision trees are constructed in a top-down recursive divide-and-conquer manner [15]. ALDER's decision tree implementation is more closely related to the C4.5 algorithm with multiple simplifications and additions. Specifically, we devised a complete binary tree to work with discrete and continuous features that can use different impurity functions to handle information gain, class weights to handle class imbalance, approximation threshold methods for big data efficiency and multiple stopping criteria. The Algorithm 1 demonstrate a simplified pseudocode version of the core function to build ALDER's decision tree.

Algorithm 1 Recursive Decision Tree Construction

```
procedure BUILD DECISION TREE(data, labels)
    node ← Node()
    if stop criteria = True then
        node.label ← most common label in labels
        return node

    feature,threshold ← select_attribute(data, labels)
    node.feature ← feature
    node.threshold ← threshold
    node.isLeaf ← False

    left_data ← data[feature] ≤ threshold
    left_labels ← labels[feature] ≤ threshold
    right_data ← data[feature] > threshold
    right_labels ← labels[feature] > threshold

    node.left_child ← build_dt(left_data, left_labels)
    node.right_child ← build_dt(right_data, right_labels)
```

The algorithm starts by initializing a node. A tree predictor has the structure of an ordered and rooted tree where each node is either a leaf (if it has zero children) or an internal node (if it has at least two children) [19]. The node is initialized as a leaf and can be substituted for an internal node later in execution. The idea is to grow the tree classifier starting from a single-node tree (which must be a leaf) that corresponds to the classifier assigning to any data point the label that occurs most frequently in the training set [19]. The tree is grown recursively by picking a leaf and replacing it with an internal node and two new leaves. To account for replacing nodes from leaves to internal, ALDER implements the node as an abstract class cable of being both types of nodes. Therefore the node class has the properties: "Is Leaf", "Label", "Feature" and "Threshold". If "Is Leaf" is set to True, then the "Label" is assigned to a certain value, and the remaining properties are *None*. Otherwise, we set "Feature" and "Threshold" with appropriate values and set "Label" to *None*.

After initializing the node, we check the algorithm for stopping criteria. Since this algorithm is recursive, we might be at a point at the execution heap where we need to stop constructing the tree in depth and return a leaf node. ALDER implement three types of stopping criteria to handle overfitting and controlling the training process. The first and most trivial is stopping if our data partition contains only a single label and therefore does not need further splitting. The second is considering the maximum depth of the tree. In order to avoid growing the tree infinitely until finding individual partitions, we set a maximum size for the tree to grow. This stop criteria also allows to control the overfitting of the tree. Finally, we consider a third stopping criteria, which also allows for controlling overfitting, considering the minimum amount of

samples in a split. This approach will stop growing a tree even if the label is not unique. Thus finding the right value for this parameter will help create a proper balance between acceptable mistakes and generalization.

If any stop criteria are met, Algorithm 1 assigns the most common label of the data partition and returns the node. On the other hand, if we have not reached a stopping point, the algorithm looks for features and thresholds with more information gained to split the data. After the feature and threshold are computed, we assign the selected values to the node and set that the node is not a leaf anymore (Algorithm 1 Lines 7-9). We split the remaining data and labels with the node split conditions now assigned. The data/labels with the select feature below or equal to the select threshold are used to create the left child, and the data/labels with the select feature above the select attribute are used to create the right child [19]. Thus, we recursively call the Algorithm 1 for each of the nodes children (Lines 17 and 18) and since we are implementing a binary tree, this function will only split into two subsets. In practice, while calling the build tree function for the children, we also increase the depth counter to stop when maximum depth is reached eventually.

Another important aspect to describe in the process of constructing a decision three is how the feature and threshold, with more information gain to split the data, are calculated. How we split the data will directly impact the model's training error. When a leaf does not contribute to the training error, we say that that leaf is pure [19]. On the other hand, leaves that do contribute are impure; thus, we focus on measuring the impurity of the tree (the entropy). Additionally, the decision tree may also suffer from overfitting, which usually happens when the tree overgrows the cardinally of the training set [19]. Therefore, choosing the leaf expansions guaranteeing the largest decrease in the training error is important.

The Algorithm 2 shows a pseudocode of ALDER's feature and threshold selection implementation. The process iterate for every possible feature in the data and get all the values of those features to calculate the candidate thresholds (Lines 3-4). Selecting a candidate threshold is an important step that can interfere significantly with the execution time. ALDER implements two methodologies of selecting threshold. The first is a naive approach that selects all unique values for the feature ordered and calculates the threshold as the middle distance between one data point and the other. This solution is not scalable for big data scenarios, where calculating the unique values every time is a cost-intensive operation, as well as looping through all possible unique values. With this in mind, ALDER also implements a technique suitable for big data, where the idea is to calculate approximated thresholds based on the percentiles of the feature values. The granularity of the approximation is based on the amount of "buckets" we want to separate the data. In this way, the user sets a hyperparameter beforehand to select how many possible thresholds the percentiles will create.

After the candidate thresholds are calculated, the algorithm will loop through all the possible threshold values for that feature. It separates the left and right labels based on the current combination of features/thresholds, allowing the calculation of the information gain of a tree split with the current combination. The split's information gain is calculated as the difference between the parent's entropy and the children's weighted entropy. ALDER's implementation calculates the entropy with three possible approaches: Shannon [20] (Eq. 1), and the ones proposed by [19]: Gini (Eq. 2) and Scaled Entropy (Eq. 3).

Algorithm 2 Select the Feature and Threshold with Best Information Gain

```
1:  procedure SELECT ATTRIBUTE(data, labels)
2:      for feature in data.columns do
3:          values ← data[feature]
4:          thresholds ← candidate_thresholds(values)
5:          for threshold in thresholds do
6:              left_labels ← labels[values ≤ threshold]
7:              right_labels ← labels[values > threshold]
8:              gain ← calculate_gain(data, labels,
                                      left_labels,
                                      right_labels)
9:
10:             if gain ≥ best_gain then
11:                 best_gain ← gain
12:                 best_feature ← feature
13:                 best_threshold ← threshold
14:
15:     return best_feature, best_threshold
```

$$\psi(p) = -p\log_2(p) - (1-p)\log_2(1-p) \tag{1}$$

$$\psi(p) = 2p(1-p) \tag{2}$$

$$\psi(p) = -\frac{p}{2}\log_2(p) - \frac{1-p}{2}\log_2(1-p) \tag{3}$$

In the above formulas, p is a value between 0 and 1, representing the ratio of one of the labels in the complete label set, while $(1-p)$ is the ratio of the other label. ALDER goes a step further while calculating the value of p. In order to handle class imbalance, the model should be able to implement a higher value of information for the sparse class. The solution that ALDER proposes is to implement class weights in the calculation. Therefore it multiplies the value of p and $1-p$ for the appropriate weight for the class. The weight is usually set to 1 for the dominant class, and for the rare class, the ratio of the length of the dominant class to the length of itself. Finally, Algorithm 2 Lines 10-13 handle the saving of the best feature and threshold combination found so far. The algorithm ends when all feature/threshold combinations have been considered.

In order to perform a sanity test on the created algorithm, ALDER relies on a traditional dataset used in literature, the Iris [21]. Even though Iris is not a binary classification problem, we can use it as a simple binary classification by removing one of the classes and considering only the non-linear separable remaining class. Figure 5 show a result of a single split values selection on the IRIS dataset. Notice that it finds the feature and the point where it best splits the data.

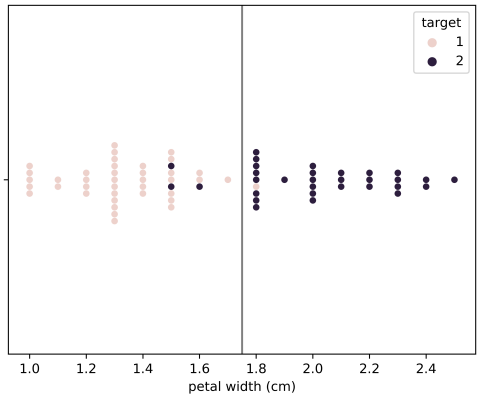

Fig. 5: Split Feature and Threshold in Iris dataset

Finally, to perform a prediction over the data, ALDER transverses the tree for every data point in the prediction set, performing the necessary evaluation of the feature for each internal node. As a result, demonstrating empirically that the constructed algorithm is sound, the training error on the non-linear separable Iris subset was 0.0, and the test error of 0.03.

G. Random Forest

With the decision tree model created, ALDER is ready to handle Random Forests. Random Forest is one kind of ensemble method, a technique that combines a series of learning models intending to create an improved composite model that outperforms a single classifier [15]. In the case of the Random forest, each model is a single decision tree so that the collection of classifiers is a "forest" [15]. ALDER individual decision trees are generated using a random selection of the data features for each tree. During the classification, each tree in the forest has its own "vote", and the mode of the "votes" of the trees is selected as the final classification [15].

ALDER's Random Forest techniques are built using *Bagging*. Given a complete training set, we generate multiple bootstrap sampled training sets to be used one for each tree [15]. A bootstrap sampled training set is a subset of the complete training set sampled with replacement, where some of the samples in the complete set will not occur in the subset, and some may occur more than once [15]. The idea of the random forests algorithm works side-by-side with the concepts of distributed processing and big data [15]. Due to its multiple model characteristics, it is intuitive to parallelize the construction of the model by distributing the creation of individual decision trees to multiple available workers. Additionally, the bagging methodology allows us to create multiple sub-datasets which could load in the primary memory of each worker in a scenario where the complete big dataset would not.

Therefore, before training the Random Forest, ALDER performs the dataset subsampling for the workers (see Fig. 1) again. The subsampling is performed with bootstrap for the rows and complete random for the available features. The amount of sub-sampled datasets is the same as the number of decision trees defined by the user in the random forest. The user may also define the fraction of rows and columns to sample from the complete data before the experiment. ALDER performs two experiments, one with the small dataset and one with the medium data, which are further detailed in the next section. Finally, it is worth noticing that ALDER prediction for the random forest is performed in batches. Since all trees must predict all test data, loading all data in memory at once may be unfeasible in big data scenarios. Handling this issue, the random forest predicts data in batches. After every tree predicted the data of the first batch, we start processing the next batch until there are no more batches to predict.

III. Performance Evaluation

This section presents the methodology adopted and the results obtained to draw insights from the proposed solution. For this, the performance evaluation was made by designing a few experiments to establish links between the methodologies developed in ALDER for creating decision trees and random forests and handling them in a big data scenario and proposed Research Questions (RQs) in Section II.

To answer these questions, a set of experiments has been designed. Especially three experiments were performed. Before the experiment, a hyperparameter tuning phase (Subsection III-A) was performed, and the found results were used in all the proposed experiments. The first experiment (Subsection III-B) tested a single sequential decision tree with a subset of the small dataset. The second experiment (Subsection III-C) consisted in training a distributed random forest with a small dataset. Finally, the third experiment (Subsection III-D) executes the distributed random forest with the medium dataset.

However, before diving into the experiments, we first need to define the proposed metrics and definitions for the experiments to standardize the methodology. The experiments use a simple training, test dataset division. The training consists of 80% of the data, while the other 20% is used for the test phase. To analyze the experiments, the following metrics were used:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4a}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4b}$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{4c}$$

$$F_1 = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4d}$$

Given a classification task experiment, each of the results could be in one of the following groups of prediction

- true positives TP (correct positive prediction),
- true negatives TN (correct negative prediction),
- false positives FP (incorrect positive prediction), and
- false negatives FN (incorrect negative prediction).

Considering the nature of the problem, the most important metrics to analyze that highly imbalanced prediction task are the Precision, Recall and F1-Score. A high precision algorithm

TABLE I: Space for the hyperparameters' exploration.

Maximum Depth	[5-20]
Minimum Samples per Split	[2-15]
Amount of Thresholds	[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
Class Weights	["No Weights", "With Weights"]
Impurity Functions	["Shannon", "Gini", "Scaled"]

TABLE II: Summary results for all executed experiments.

	Accuracy	**Precision**	**Recall**	**F1**
DT Train	0.9994	0.9498	0.5237	0.6751
DT Test	0.9992	0.8477	0.4631	0.5990
RF S Test	0.9993	0.9931	0.4044	0.5748
RF M Test	0.9985	0.2203	0.0978	0.1355

is important from the business point of view because we are not accusing transactions of being fraudulent mistakenly. On the other hand, a high recall is also important because we are keeping many fraudulent transactions from being undetected. Finding a good balance between precision and recall is important for many binary classification problems, and the preference between one metric or the other is usually ad-hoc for the situation. This is where F1-Score comes to hand. The score measures a test accuracy that considers both precision and recall. The score is the harmonic mean of those metrics, where an F1 score reaches its best value at 1, which indicates the best precision and recall, and worst at 0.

The experiments were conducted using a virtual machine in the Google Colab. The co-laboratory environment was a free tier with 2 Core CPUs, 12 GB of RAM and 100GB of storage.

A. Hyperparameter Tunning

Finding an optimum set of hyperparameters is an essential task to achieve model performance since setting the hyper-parameters in the wrong way can lead to underfitting or overfitting [19]. ALDER selects the best set of hyperparameters in a given search space using the Hyperopt: Distributed Asynchronous Hyper-parameter Optimization [7].

Hyperopt[3] is a python library that implements Sequential Model-Based Optimization (SMBO) algorithms, also known as Bayesian optimization [7]. SMBO is applicable where minimizing the value for some function $f(x)$ leads to a high cost due to the complex evaluation process [7]. Such a method uses a search algorithm to determine hyper-parameters through search interactions denominated trials.

The usage of the Hyperopt library is based on defining three main characteristics:

1) An objective function value to be minimized
2) A search algorithm used to find the optimization set
3) The search space for the selected algorithm

The selection of these characteristics is then used to conduct the desired experiments to find the best hyperparameters.

The Hyperopt experiment was conducted offline in the Colab environment[4] in a Macbook Air M2 8-Core CPU with 32GB RAM. The reasoning for not using the Colab to perform the hyperparameter tuning is two-fold: it would take a considerable amount of time to complete, and the Macbook has faster processing and more available memory. The hyperparameter optimization was performed with the complete small dataset in memory, and the optimization results in the ALDER were extrapolated to the small distributed and medium distributed experiments as well, assuming that the characteristics of the data would not change, only the amount.

[3]Available at http://hyperopt.github.io/hyperopt/

[4]However, the code used is present for inspection on the Google Colab.

Therefore the hyperparameter tuning was executed with the following characteristics: (i) the objective function was the F1-Score value returned from the decision tree model; (ii) the search algorithm was the Tree Parzen Estimator [22]; and (iii) the search space is presented in Table I. The Tree Parzen estimator is a widely used Bayesian optimization method that has achieved various outstanding performances and has been the key factor in winning multiple Kaggle competitions [23].

The experiments were executed with 100 trials, and the training set was further split into training and validation. Therefore, for the hyperparameter tuning, we considered a 60%-20%-20% training, validation, and test split, as proposed for this dataset [1]. The best set of hyperparameters achieved an F1-Score of 0.6446 in the validation set and consisted of the following configuration: 18 depth, minimum of 14 samples per split; 45 thresholds; class weights and Shannon function. However, considering the execution time cost, we decided to select the second-best set of parameters which achieved an F1-Score of 0.6400, which consisted of the following configuration:

- **Maximum Depth:** 16
- **Minimum Samples per Split:** 14
- **Amount of Thresholds:** 20
- **Class Weights:** With Weights
- **Impurity Function:** Shannon

B. Performance Evaluation for the Decision Tree

Our first experiment tested a single decision tree in the IBM TAML data, especially the small dataset. This experiment was designed to answer the first research question (**RQ1**) on whether the devised decision tree can learn to detect money laundering. For this experiment, we considered only a fraction of the small dataset. Even though the dataset can be completely loaded into memory, its execution would take time. Therefore the experiment is run with 50% of the training set and 50% of the testing set. Notice that it does not change the proposed 80%-20% data division.

Table II shows an overview of the experiments' results. The first and second content lines are the metrics for the training and testing process, respectively. Overall the model achieved an F1-Score of 0.6751 on training that decreased to 0.5990 on the test set, a Recall of 0.5237 on training and 0.4631 in test, and a Precision of 0.9498 on train and 0.8477 on test. These values are a good indication of the training process. We notice that the decision tree overall was able to learn but with a higher focus on precision rather than the recall, and the lack of focus on the recall decreases the F1-Score. However, we also notice that the difference between the training and testing metrics is not significant, showing that the model was able to generalize. Therefore, this model has a higher bias for the F1-Score but a low variance.

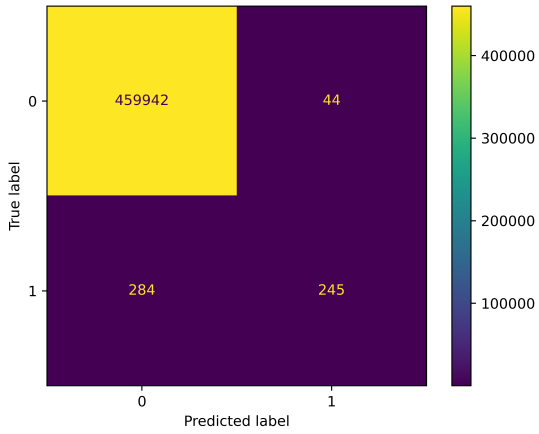

Fig. 6: Decision Tree Confusion Matrix

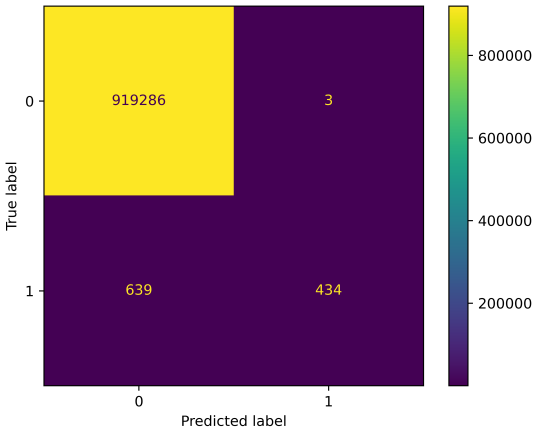

Fig. 7: RF Confusion Matrix for Small Dataset

We can analyze the results further by looking at the confusion matrix in Figure 6. As we can see, the matrix reflects the metrics. Since the recall is slightly below 0.5, we can observe that false negatives are slightly higher than true positives. Additionally, we see only 44 transactions mistakenly being predicted as fraudulent, which is a result of our higher precision. Therefore, answering **RQ1**, ALDER can learn without significant false alarms, however, with some considerable misses.

C. Performance Evaluation for the Small Random Forest

Given the satisfactory results in the single decision tree process, we implemented the first experiment with a random forest. This experiment was designed to answer the first part of **RQ2** and the **RQ3**. Therefore, we are trying to understand if the random forest could learn in a distributed environment for the small dataset and if the distributed random forest outperforms the single decision tree experiment.

For the random forest experiments, we had to configure an additional set of parameters, those being: the number of workers (which is the same as the number of decision trees in the forest), the size of the column subsampling and the size of the row bootstrap subsampling. Considering the small dataset, each decision tree is trained with 40% of the rows and 80% of the columns. A total of four trees were used to populate the forest.

The results are shown in Table II (lines 3 and 4), and the confusion matrix in Figure 7. From the table, we can notice that the overall performance of the predictor is worse than the single decision tree, except for the precision metric. The random forest predictor has a lower F1 Score than the decision tree, which is a reflection of an even lower recall. Therefore, this result answers the **RQ3**; the distributed random forest for the given row/column split does not outperform a single decision tree.

From the confusion matrix, we notice the strength of the precision for this predictor, only miss-classifying 3 transactions. However, we also notice the weakness of the recall. This result could be related to the portion of the dataset columns we exclude. By sampling fewer columns on the dataset, we may exclude important features that detect fraudulent transactions for some predictors, thus decreasing the recall. Even so, since we are achieving good precision, answering the **RQ2**, the proposed random forest can also learn in a distributed environment, however, with a strong preference for precision. It is possible that given the available processing power and memory availability for a strong distributed environment, we could increase the fraction of rows/columns in the data subset of each worker, potentially increasing the performance and metrics of the predictor.

D. Performance Evaluation for the Medium Random Forest

Finally, we execute the last experiment proposed by ALDER, concluding the discussion over **RQ2** and **RQ4**. Therefore, we will finish answering whether ALDER can learn in a simulated distributed environment with a medium data set and if the data increase of the medium dataset improved the random forest performance. For this second random forest experiment, we had to configure the additional set of parameters considering the increase in the dataset and the amount of available resources. Hence, the medium dataset, which has approximately 4 GB, was trained with six decision trees. Each tree was trained with 20% of the rows and 80% of the columns.

The results are shown in Table II (lines 5 and 6), and the confusion matrix in Figure 8. From the table, we can notice that the overall performance of the predictor is even worse than the small random forest, especially for the test metrics. The random forest predictor has lower values for every possible metric on the test set. Therefore, answering the **RQ4**, the medium distributed random forest for the given row/column split does not outperform the small random forest or a single decision tree, being ALDER's worst predictor.

Figure 8 confusion matrix only confirms the metrics showing the number of mistakes made. Therefore, finishing answer **RQ2**, the ALDER model fails to learn with the proposed decision tree and random forest parameters in the medium dataset. It is possible that the lower subsampling of rows

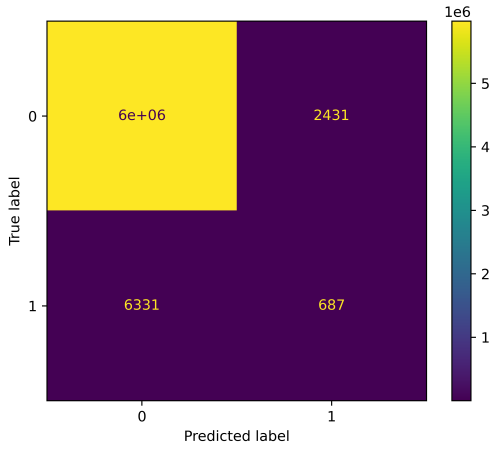

Fig. 8: RF Confusion Matrix for Medium Dataset

combined with an insufficient number of decision trees in the random forest is failing to obtain the information to predict the positive class successfully. Additionally, it can be true that the hyperopts optimal parameters in the small dataset underperformed when extrapolated to the medium dataset. A possible improvement to the predictor would be to consider training with more decision trees; however, it comes with a cost of execution time.

E. Considerations on Scalability

Scalability is a critical aspect of the ALDER methodology. Practically detecting money laundering requires being able to process even larger data files. Processing large-scale data efficiently and effectively requires careful consideration of various scalability factors.

Overall the ALDER methodology scales up properly with little consideration. At the actual moment, some assumptions are made that could break if the amount of data scales. We consider that each worker can load its data into its primary memory and create its tree. Therefore, if the amount of data exceeds the worker's memory size, we need to devise lazy evaluating techniques to construct the decision tree. We also consider that the predictions and label vectors can be loaded into primary memory. If the data increases exponentially, some adaptations will be needed to calculate the metrics and plot the confusion matrix when data is not loadable in the memory.

Finally, expanding ALDER to a real distributed environment should be sufficiently straightforward by assigning the construction tree function for each available worker. However, some considerations on the distributed environment are also essential to guarantee good portability. Such as:

- optimizing data partitioning and distribution among computing nodes to minimize network overhead and maximize parallelism
- ensuring even distribution of workload across nodes to prevent bottlenecks and utilize computing resources efficiently
- implementing mechanisms to handle node and network failures
- considering both adding more computing nodes (horizontal scaling) and increasing individual node resources (vertical scaling) to achieve scalability

IV. Conclusion and Future Work

In conclusion, this project presents ALDER, an implementation of decision trees and random forest algorithms for detecting money laundering. By leveraging the IBM TAML dataset and utilizing Apache Spark for scalable computing, ALDER aims to address research questions regarding its effectiveness, performance with different dataset sizes, and scalability in real distributed environments. Through a rigorous data engineering pipeline and exploratory data analysis, ALDER creates robust features that generalize the content of the data without causing overfitting. Furthermore, ALDER's decision tree and random forest are designed from scratch, incorporating mechanisms to handle data imbalance and big data thresholds. The implementation also includes state-of-the-art hyperparameter tuning using Bayesian optimization. The results demonstrate that ALDER's decision tree performs well in the simulated distributed environment, achieving a high precision rate while leaving room for improvement in recall. The best model achieved an F1-Score of 0.5990 on test data, while the best distributed version attained 0.5748.

In future work, ALDER can be further improved by deploying it in a real distributed environment with higher computational capability to assess its scalability. Additionally, executing ALDER in a distributed environment with more robust decision trees and increasing the number of decision trees in the random forest can enhance its detection capabilities for money laundering transactions. These enhancements will contribute to ALDER's effectiveness in combating financial crimes.

Declarations

Declaration of originality

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.

Data availability

All used data are publicly available on the Kaggle website: https://www.kaggle.com/datasets/ealtman2019/ibm-transactions-for-anti-money-laundering-aml

Code availability

Every code developed is publicly available on Github: https://github.com/Skalwalker/AntiMoneyLaundering

REFERENCES

[1] E. Altman, B. Egressy, J. Blanuša, and K. Atasu, "Realistic synthetic financial transactions for anti-money laundering models," 2023.

[2] J. Walker, "How big is global money laundering?" *Journal of Money Laundering Control*, vol. 3, no. 1, pp. 25–37, 1999.

[3] M. Jullum, A. Løland, R. B. Huseby, G. Ånonsen, and J. Lorentzen, "Detecting money laundering transactions with machine learning," *Journal of Money Laundering Control*, vol. 23, no. 1, pp. 173–186, 2020.

[4] Z. Chen, L. D. Van Khoa, E. N. Teoh, A. Nazir, E. K. Karuppiah, and K. S. Lam, "Machine learning techniques for anti-money laundering (aml) solutions in suspicious transaction detection: a review," *Knowledge and Information Systems*, vol. 57, pp. 245–285, 2018.

[5] CNN Brazil, "7 em cada 10 transações bancárias no país são feitas por canais digitais, mostra febraban," 2021, visited on: 04/07/2023.

[6] A. Rajaraman and J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2011.

[7] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *International conference on machine learning*. PMLR, 2013, pp. 115–123.

[8] Apache Software Foundation, "Apache spark," 2018, visited on: 04/07/2023. [Online]. Available: https://spark.apache.org

[9] J. Rybicki, "Best practices in structuring data science projects," in *Information Systems Architecture and Technology: Proceedings of 39th International Conference on Information Systems Architecture and Technology–ISAT 2018: Part III*. Springer, 2019, pp. 348–357.

[10] Google, "Google colab," 2021, visited on: 04/07/2023. [Online]. Available: https://colab.google

[11] J. Schwarzmann, "The importance of layered thinking in data engineering," 2021, visited on: 04/07/2023. [Online]. Available: https://towardsdatascience.com/the-importance-of-layered-thinking-in-data-engineering-a09f685edc71

[12] DrivenData, "Cookiecutter data science," 2021, visited on: 04/07/2023. [Online]. Available: https://drivendata.github.io/cookiecutter-data-science/

[13] R. Chalapathy and S. Chawla, "Deep learning for anomaly detection: A survey," 2019.

[14] B. Chambers and M. Zaharia, *Spark: The definitive guide: Big data processing made simple*. " O'Reilly Media, Inc.", 2018.

[15] J. Han, J. Pei, and H. Tong, *Data mining: concepts and techniques*. Morgan kaufmann, 2022.

[16] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, pp. 81–106, 1986.

[17] ——, *C4.5: programs for machine learning*. Elsevier, 2014.

[18] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[19] N. Cesa-Bianchi, "Statistical methods for machine learning class notes," 2023. [Online]. Available: https://cesa-bianchi.di.unimi.it/MSA/index_22-23.html

[20] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[21] R. A. Fisher, "Iris," UCI Machine Learning Repository, 1988, DOI: https://doi.org/10.24432/C56C76.

[22] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," *Advances in neural information processing systems*, vol. 24, 2011.

[23] S. Watanabe, "Tree-structured parzen estimator: Understanding its algorithm components and their roles for better empirical performance," 2023.