

Master’s Degree Program in

**Data Science and Advanced Analytics**

Big Data Analytics

Fraud Detection

Rodrigo Luís, 20240742

João Marques, Number

Gustavo Gomes, Number

Diogo Ruivo, Number: 20240584

Rafael Lopes, Number: 20240588

Group 17

**NOVA Information Management School Instituto Superior de Estatística e Gestão de Informação**

Universidade Nova de Lisboa

2025

**INDEX**

[LIST OF TABLES AND FIGURES 2](#_Toc199848179)

[1. INTRODUCTION 3](#_Toc199848180)

[2. DATA COLLECTION AND PREPROCESSING 3](#_Toc199848181)

[3. METHODOLOGY 5](#_Toc199848182)

[4. RESULTS AND EVALUATION 5](#_Toc199848183)

[5. CONCLUSION 6](#_Toc199848184)

# LIST OF TABLES AND FIGURES

Table 1:

Table 2:

Table 3:

Figure 1:

Figure 2:

# INTRODUCTION

In a context where the volume and complexity of transactional data are growing exponentially, detecting financial fraud is proving to be a critical and current problem. Motivated by the practical relevance and the associated technical challenge, our group chose to dedicate this project to the development of a fraud detection system based on machine learning techniques and Big Data analysis.

We chose the dataset because we wanted to look into a real-life case that had a big effect on the financial sector. Therefore, we opted for a structured and representative dataset obtained from the Kaggle platform, with the aim of simulating a scenario of detecting atypical behavior in large-scale financial transactions. In this context, we used the Databricks platform to develop and train the models, and Apache Kafka as a streaming system to simulate the continuous flow of financial transactions, allowing fraud detection in real time.

The biggest problem in this field is that the data is not balanced; only a small percentage of transactions are actually fraudulent. In this sense, it is crucial to apply methodologies that ensure not only the effectiveness in detecting anomalies, but also the ability to scale solutions to production contexts, while maintaining computational efficiency. Using the Databricks platform and the Spark ecosystem, this work covers all essential steps, from data collection and pre-processing to feature engineering, variable selection, training and evaluation of multiple classification models.

# DATA COLLECTION AND PREPROCESSING

To conduct the project, the team required a suitable dataset and selected a structured dataset titled *“financial\_fraud\_detection\_dataset”*, sourced from Kaggle—a widely recognized platform for public datasets used in research and practical applications.

The dataset, provided in .csv format, was uploaded to Databricks, where development and execution of notebooks took place. Data preprocessing was performed in an initial notebook named *“PreProc*”. The first step involved exploratory data analysis (EDA), during which the team reviewed the dataset and identified several data quality issues.

The target variable, *“is\_fraud”*, is binary, with values (0, 1), representing non-fraudulent and fraudulent transactions, respectively. This defines the problem as a binary classification task. The dataset comprises exactly 5 million transaction records, of which only 3.26% were labeled as fraudulent. This class imbalance presents a challenge that was addressed in subsequent steps.

During preprocessing, a key issue was found in the “*time\_since\_last\_transaction*” column. This feature contained missing values, which were imputed using the median of the non-missing values. Additionally, some entries in this column had negative values, which are not logically valid in the context of elapsed time. Rather than converting these to nulls, the team opted to replace them with their absolute values, ensuring data consistency while avoiding further data loss.

To further enrich the dataset, time-derived features were engineered by decomposing the original “*timestamp*” variable into components such as “*hour\_of\_day”*, “*day\_of\_week*”, and other relevant temporal attributes. These new features enabled the team to capture temporal patterns in transaction behavior.

In addition, rolling behavioral features were created to provide contextual insights over specific time windows. For example, using the time-derived columns and identifiers such as “*sender\_account”*, the team computed metrics such as the number of transactions initiated by an account in the past 24 hours. These rolling features aimed to highlight unusual behavior patterns that may indicate fraudulent activity.

To ensure consistency and reproducibility across the training, validation, and test phases, all preprocessing steps were integrated into a single Spark ML Pipeline. This design ensures that the same data transformations are applied not only during model development but also when the model is deployed in a production environment.

The initial step focused on addressing skewness in numerical variables. Specifically, the features “*amount*” and “*time\_since\_last\_transaction\_imp*” exhibited long-tailed distributions, which can hinder the performance of machine learning algorithms. To mitigate this, a logarithmic transformation was applied using the *log1p(x)* function (computing *log(1 + x)*), which is effective in handling values near zero while compressing outliers. This transformation resulted in the creation of two new features: “*log\_amount*” and “*log\_time\_since\_last\_tx*”.

In addition to these transformations, categorical feature encoding was performed to convert string-based variables into a machine-readable format. This was achieved through a two-step process involving *StringIndexer* and *OneHotEncoder*. *StringIndexer* assigned a numeric index to each category, while *OneHotEncoder* converted these indices into sparse binary vectors. The *OneHotEncoder* was configured with *dropLast*=False to retain all categories, which is especially useful for tree-based models, which will be tested later, and *handleInvalid*="keep" to gracefully handle any previously unseen categories in validation or test datasets.

Some categorical features, such as “*sender\_account*” and “*receiver\_account*”, presented challenges due to their high cardinality. Applying one-hot encoding to these fields would result in extremely large and sparse feature vectors, leading to computational inefficiency and a risk of overfitting. To address this, *FeatureHasher* was employed to project these high-cardinality categorical variables into a fixed-length hashed feature space. Each was hashed into a 1,024-dimensional vector, maintaining a compact representation while preserving useful signal for fraud detection.

After all individual transformations were completed, a *VectorAssembler* was used to consolidate the various feature types — numerical, log-transformed, one-hot encoded, and hashed features—into a single vector column named “*features\_raw*”. This step is essential for downstream model training in Spark ML, which expects a unified feature vector as input.

To normalize feature magnitudes and ensure compatibility across different model types, a *StandardScaler* was applied to the assembled feature vector. While tree-based algorithms typically do not require feature scaling, this step is crucial for algorithms that rely on distance metrics or assume normally distributed features. Standardizing the input ensures that the pipeline remains modular and flexible, capable of supporting a wide range of machine learning models.

The target variable “*is\_fraud*” was cast to double type to meet Spark ML’s requirements. As mentioned before and given that fraudulent transactions account for only 3.26% of the total dataset, the problem is characterized by class imbalance. To address this, a weight column was introduced to rebalance the class distribution during training. A class weight ratio of 26:1 was applied, assigning a weight of 26.0 to fraudulent transactions and 1.0 to non-fraudulent ones. This approach ensures that the learning algorithm pays appropriate attention to the minority class, thereby improving the model's ability to detect fraud.

**### not sure if here or methodology**

To enhance model efficiency and reduce potential overfitting, a feature selection step was carried out using a Random Forest-based approach. This step aims to identify and retain the most informative features for downstream modeling.

The process begins by loading the preprocessed datasets (“*train\_ready*”, “*val\_ready*”, and “*test\_ready*”) from Delta format storage in Databricks. These datasets include all previously engineered features consolidated into a vector column named features, alongside the label and weight columns used for supervised learning.

Random Forests inherently compute feature importances as part of the training process. These scores reflect the relative contribution of each input feature to the model's predictive accuracy.

Following model training, the feature importances were extracted from the trained Random Forest model and sorted in descending order. The top 10 most important feature indices were then selected using NumPy’s *argsort* function.

To improve interpretability, the indices were mapped back to their corresponding feature names. Once the top 10 features are identified, a new transformation pipeline was created using Spark’s *VectorSlicer*. This tool extracts only the selected feature indices from the full feature vector, creating a reduced feature vector named “*features\_topK”.*

A Spark ML Pipeline was then constructed with the VectorSlicer as its sole stage. This pipeline was trained (i.e., fit to the data) to finalize the slicing configuration, and the resulting model was saved to persistent storage at /FileStore/models/slicer\_top10.

This pipeline will later be used to transform validation and test datasets, or any new incoming data, to include only the top-ranked features identified during training. This streamlined feature set is expected to simplify models, reduce training time, and potentially improve generalization performance.

# METHODOLOGY

To address the classification task, the team applied the developed preprocessing pipeline across different machine learning models, including Decision Trees, Gradient-Boosted Trees, Logistic Regression, and Random Forests. This allowed for an evaluation of different approaches in identifying fraudulent transactions.

Decision Trees: a simple supervised learning algorithm that splits data into branches based on decision rules inferred from input features. For binary classification, they recursively divide the data to separate classes (e.g., fraud vs. non-fraud) by maximizing a metric like information gain or Gini impurity. They are interpretable and fast to train. Decision Trees are scalable across large datasets and benefit from Spark's distributed processing capabilities, making them suitable for real-time decision-making in massive, high-volume transactional data.

Gradient-Boosted Trees (GBT): an ensemble method that builds a sequence of weak learners (typically shallow trees), where each subsequent tree corrects the errors of the previous one by minimizing a loss function. GBTs are highly effective for binary classification problems, especially on imbalanced datasets, due to their ability to capture complex, nonlinear relationships and interactions between features. GBT can be implemented using distributed training that enables model training over millions of rows while leveraging memory and processing efficiencies in a Spark cluster. This makes them a strong candidate for large-scale fraud detection systems.

Logistic Regression: a linear model used for binary classification due to its simplicity, interpretability, and probabilistic output. It models the log-odds of the target class as a linear function of the input features. Despite its linearity, it performs well when the features are linearly separable and is effective when combined with proper feature engineering and regularization. Logistic Regression is optimized for distributed datasets, supports weighted examples (critical for imbalanced data), and integrates seamlessly into Spark ML Pipelines, making it ideal for high-throughput scoring tasks where speed and scalability are essential.

Random Forest: an ensemble learning method that builds multiple Decision Trees and combines their predictions by averaging (for regression) or voting (for classification). For binary classification, Random Forests are robust to noise and overfitting and can naturally handle missing values and feature importance estimation. Their strength lies in their ability to generalize well across diverse datasets. Random Forests are implemented to scale efficiently with distributed data, making them ideal for processing massive transactional datasets and deriving stable, high-performing models in fraud detection and other anomaly detection tasks.

# RESULTS AND EVALUATION

To make a good fraud detection system, we had to thoroughly test machine learning models. We focused on improving the Gradient Boosted Trees (GBT) Classifier in the Databricks environment using Apache Spark's MLlib framework. This section details the meticulous hyperparameter tuning process undertaken to enhance the model’s performance, the strategic adjustments made to address the dataset’s inherent challenges, and the outcomes achieved.

We chose the GBTClassifier because it has been shown to work well with datasets that are not evenly distributed, which is a common problem in fraud detection assignments where there are a lot more genuine transactions than fraudulent ones (around 3.5% of the dataset). Gradient Boosted Trees are a kind of ensemble method that is like XGBoost. They set weak learners together in a chain to lower a loss function. They are very good at finding complex, non-linear relationships and interactions between features. Their ability to give feature importance scores fits perfectly with our preprocessing pipeline, which used Random Forest-based feature selection to find the top 10 features (features\_topK). GBT is a great way to find fraud because they can be used in distributed environments like Spark, are resistant to noisy data, and support class weighting, which makes the effects of class imbalance less severe. The GBTClassifier is great for finding fraud in high-throughput transactional systems in real time because it trained on the huge data sets (about 5 million records) using Spark's distributed processing.

The hyperparameter tuning process was designed to make the GBTClassifier work as well as possible on a balanced training set (train\_balanced) by giving the minority class (label=1.0) a class weight of 20.0 and undersampling the majority class (label=0.0) at fractions of 0.1 and 0.2. This was done to make fraud detection easier. The preprocessing pipeline changed the data by adding a VectorSlicer model that was stored at /FileStore/models/slicer\_top10. This was done to focus on the features that were most likely to be useful. At first, a full grid search with 16 combinations of the following parameters was planned:

* maxIter: [10, 20]
* maxDepth: [3, 5]
* stepSize: [0.01, 0.1]
* subsamplingRate: [0.8, 1.0]

The goal of this grid was to investigate a wide variety of model complexities and learning rates. Early tests, however, showed a problem: all combinations had the same performance metrics. The model over-predicted the positive class, most likely as a result of the dataset's extreme imbalance and the limited hyperparameter variation.

The full 16-combination grid was not feasible due to the Databricks cluster's 1-hour auto-termination limit, which was a major limitation because each training iteration took about 8 minutes. We adjusted by condensing the grid to a single, well-chosen combination in order to avoid possible cluster crashes:

* maxIter: 10
* maxDepth: 10
* stepSize: 0.5
* subsamplingRate: 0.5

This configuration, tested with majority class sampling at 0.2, yielded promising results:

* **F1-score**: 0.3393
* **Weighted Precision**: 0.9597
* **Weighted Recall**: 0.2387
* **Confusion Matrix**:
* [[151017.0, 562904.0], [ 920.0, 25778.0]]

The tuned GBTClassifier demonstrated a modest improvement over initial runs, with the F1-score increasing from 0.3051 to 0.3393 and false positives decreasing from 581,276 to 562,904 but in the other hand every single try with more robust parameters or class sampling made the cluster come to a crash after the one hour line.

# CONCLUSION

Here we conclude