UNIVERSITY OF PISA

DEPARTMENT OF INFORMATION ENGINEERING

Data Mining and Machine Learning

Credit Card Fraud Detection

Work group:

Valentina Bertei

Francesco Tarchi

ACADEMIC YEAR 2024/2025

Contents

List of Figures

1. Introduction

Credit Card Fraud Detection is a simple security application which allows a user to manually input transaction details, which are then classified as Fraudulent or Legitimate by a set of classifiers. The key idea behind this project is to provide both performance comparison and explainability of the classification process. This way, the user can not only see which transactions are flagged as fraudulent, but also understand the reasoning behind each classification.

In particular, the application implements six classifiers: Decision Tree, Random Forest, Naive Bayes, K-Nearest Neighbors (KNN), AdaBoost, and XGBoost. For each classifier, the system provides visualizations of performance metrics as well as explanations for the classification decisions, allowing the user to compare the models not only in terms of accuracy, precision, recall, and F1-score, but also in terms of interpretability.

The application is developed using Streamlit, providing an interactive and user-friendly interface for entering transactions and observing the results.

1.1 GitHub Repository

All the codes and the raw materials (apart from the dataset) can be found at the following repository.

https://github.com/francetarchi/CreditCardFraudDetection

2. Dataset and Preprocessing

2.1 Dataset

The dataset used for this application comes from the *IEEE-CIS Fraud Detection* competition on Kaggle [2]: it contains historical online transaction data enriched by behavioral signals and device information. The dataset includes 590,540 transactions (1.08 GB), of which 20,663 are labeled as fraudulent (approximately 3.5%). The binary target variable is `isFraud`, where 1 represents a *fraudulent* transaction and 0 a *legitimate* one.

The dataset appears strongly imbalanced due to the small proportion of fraudulent transactions. As a consequence, proper evaluation metrics were used to assess the performance of the models, instead of relying solely on *accuracy*.

The dataset contains 432 features, grouped according to their type and origin [1].

- **Transaction features:** These include transaction amount (`TransactionAmt`) and a timedelta feature (`TransactionDT`) representing the time difference from a reference date. Product codes are included via `ProductCD`.
- **Address features:** Billing and mailing information, such as `addr1`, `addr2`, and distances between addresses (`dist` features).
- **Email domain features:** Recipient and purchaser email domains (`Remaildomain` and `Pemaildomain`).
- **Card features:** Payment card information including type, category, issuing bank, and country (`card1-card6`).
- **Counting features:** Features `C1-C14` representing counts, with masked meaning for privacy reasons.
- **Timedelta features:** Features `D1-D15` representing time differences between events.
- **Match features:** Features `M1-M9` describing matches, e.g., between cardholder name and address.
- **Identity features:** Features `id01-id38` and `Device`, `DeviceInfo`, capturing device information, behavioral signals, and security ratings.
- **Vesta engineered features:** Features `V1-V339`, numeric features engineered by Vesta, including rankings, counts, and entity relations.

The input to the models consists of multivariate features including transaction details, card and address information, identity data, and engineered Vesta features. The output is the binary class label `isFraud` $\in \{0, 1\}$.

A graphical representation of the dataset distribution (fraudulent vs legitimate[1] transactions) is reported in Figure 2.1.

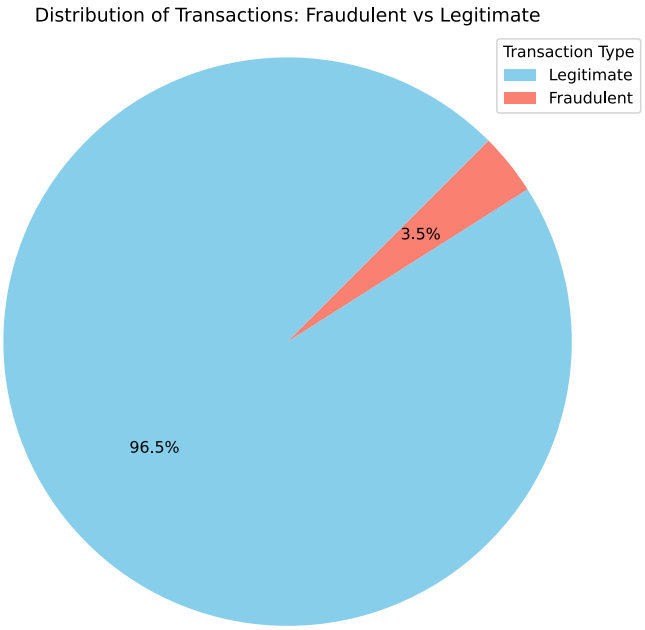


Figure 2.1: Distribution of transactions: *fraudulent* vs *legitimate*.

2.2 Data preprocessing

This section describes the complete preprocessing pipeline applied to the raw dataset prior to model training. The implementation follows the script included in the project and is organized into sequential steps: feature definition, temporal feature engineering, initial filtering, imputation, scaling, encoding, feature selection, class balancing, and persistence of preprocessing artifacts and datasets.

2.2.1 Feature lists and initial setup

We start by defining two master lists of features:

- `all_categorical_features`: categorical and identifier-like columns (e.g. `ProductCD`, `card1..card6`, `addr1`, `DeviceType`, `id_12..id_38`, ...).

[1] In next paragraphs, legitimate transactions are often assessed as *nonfraudulent* transactions.

- `all_numerical_features`: continuous and numeric columns (e.g. `C1..C14`, `D1..D15`, `TransactionAmt`, `V1..V339`, `dist1`, `dist2`, `id_01..id_11`, and engineered temporal features).

An auxiliary function `update_features_arrays(df,categorical_features,numerical_features)` is used repeatedly to restrict these master lists to the columns actually present in the DataFrame, returning the available categorical features, available numerical features, and their concatenation.

2.2.2 Loading and train/test splitting

The raw dataset is loaded from `paths.RAW_ALL_PATH` into a DataFrame `df`. A first stratified split (with respect to the `isFraud` label) produces `raw_train` and `raw_test` using `train_test_split(...,stratify=df["isFraud"])`. This preserves the class distribution between training and testing partitions.

2.2.3 Temporal feature engineering

The original field `TransactionDT` contains cumulative seconds from an unknown origin, therefore the raw value is not meaningful as an absolute timestamp. Therefore, we derive cyclic and not cyclic temporal features to expose periodic patterns.

- $\texttt{TransactionDT\_days} = \frac{\texttt{TransactionDT}}{24\times60\times60}$
- $\texttt{hour} = \left\lfloor \frac{\texttt{TransactionDT}}{3600} \right\rfloor \bmod 24$
- $\texttt{dayofweek} = \left\lfloor \frac{\texttt{TransactionDT}}{24\cdot3600} \right\rfloor \bmod 7$
- Cyclical encodings (these transformations preserve **circular continuity**, e.g. 23:00 → 0:00, and are commonly used for modeling periodic patterns):

$$\texttt{hour\_sin} = \sin\left(\frac{2\pi \cdot \texttt{hour}}{24}\right)$$

$$\texttt{hour\_cos} = \cos\left(\frac{2\pi \cdot \texttt{hour}}{24}\right)$$

$$\texttt{dayofweek\_sin} = \sin\left(\frac{2\pi \cdot \texttt{dayofweek}}{7}\right)$$

$$\texttt{dayofweek\_cos} = \cos\left(\frac{2\pi \cdot \texttt{dayofweek}}{7}\right)$$

2.2.4 Feature / target separation

The pipeline starts by separating features and target:

- $X = \{\text{all columns } \textit{except } \texttt{isFraud}\}$;
- $y = \{\texttt{isFraud}\}$.

By now on, all the preprocessing operations are implicitly carried out on X, if not differently specified.

2.2.5 Initial filtering

First of all, columns deemed *useless for prediction* (`TransactionID` and `TransactionDT` raw field) are dropped.

Then, by analyzing the dataset structure, it is easy to notice that many columns are almost full of *missing values*: such sparse features are unlikely to be informative and just creates noise. A filter is introduced to remove columns whose proportion of missing values exceeds a configured threshold (`const.MISSING_VALUES_THRESHOLD`, in our case 90%).

2.2.6 Train / test split

After preliminary filtering and features creation, a second *stratified* `train_test_split` with the same random seed (`const.RANDOM_STATE`) and test size (`const.DIM_TEST`) produces X_{train}, X_{test}, y_{train} and y_{test}.

2.2.7 Missing-value imputation

Although the initial filtering of missing values, columns with a proportion of missing values under 90% are still present in the sets: the *remaining missing values* are handled using a `ColumnTransformer` with two transformers.

- For categorical features: `SimpleImputer(strategy="most_frequent")` replacing missing values with the most frequent category.
- For numerical features: `SimpleImputer(strategy="median")` replacing missing values with the median of the respective column.

The *imputer* is fitted (`.fit()`) on train set and then applied (`.transform()`) to both train and test sets. The fitted imputer object is persisted for reproducibility and later use.

2.2.8 Scaling

In transactions datasets *extreme outliers are statistically common* and the classifier may turn out to be sensitive to them: in order to reduce such sensitivity, we use a `ColumnTransformer` with two transformers.

- For categorical features, a `passthrough` is applied to keep them unchanged (they are categorical tokens, not yet encoded).
- For numerical features, a `RobustScaler` is applied to center the median and scale according to interquartile range.

As before (during imputation), the *scaler* is fitted (`.fit()`) on train set and then applied (`.transform()`) to both train and test sets. The fitted scaler object is persisted for reproducibility and later use.

2.2.9 Encoding categorical features

As pointed out by the previous (double) use of the transformers, in the dataset there are some *categorical features* that need to be encoded to be correctly processed by the classifiers: we use a `ColumnTransformer` with two transformers.

- For categorical features, `OneHotEncoder` is applied to one-hot encode them.
- For numerical features, a `passthrough` is applied to keep them unchanged (they are categorical tokens, not yet encoded).

As before (during both imputation and scaling), the *encoder* is fitted (`.fit()`) on the train set and then applied (`.transform()`) to both train and test sets. The fitted encoder object is persisted for reproducibility and later use.

After encoding, the feature representation is *sparse* to save memory. Moreover, from now on, we do not need anymore transformers to act differently on numerical and categorical features, thanks to the encoding.

2.2.10 Variance-based filtering

After encoding, a `VarianceThreshold` filter removes encoded features whose variance falls below a configured threshold (`const.VARIANCE_THRESHOLD`). Low-variance features contribute little to predictive power and can be removed to reduce dimensionality and computational cost.

2.2.11 Discrete/continuous mask for feature selection

A boolean mask named `discrete_mask` is dynamically created to mark which remaining features are discrete (integer dtype) or continuous. This mask is used by the mutual-information based selector to correctly treat discrete features during score computation.

2.2.12 Mutual-information feature selection

To reduce dimensionality further and keep only the most informative features with respect to the target, the pipeline applies `SelectKBest` with `score_func=mutual_info_classif` (implemented as the helper function `mi_score`). The selector keeps the top `const.BEST_K_FEATURES` features by mutual information with the target. The pipeline also plots and saves a bar chart of the ordered mutual information scores (`graphs/mi_scores.png` and `graphs_svg/mi_scores.svg`) for inspection.

After selection, dense DataFrames X_{train} and X_{test} containing only the selected features are reconstructed and the selected feature names are saved to `paths.SELECTED_FEATURES_PATH`.

2.2.13 SMOTE class balancing

Because the dataset is strongly imbalanced (fraudulent transactions are rare), the training set is balanced using Synthetic Minority Over-sampling Technique (SMOTE). SMOTE synthesizes new minority-class samples in feature space to reach a target minority ratio defined by `const.TARGET_MINORITY_RATIO_1_5` (the sampling strategy). SMOTE is applied only to the training set; the test set remains untouched to preserve a realistic evaluation distribution. The balanced training set is stored as `SMOTED_TRAIN_PATH`.

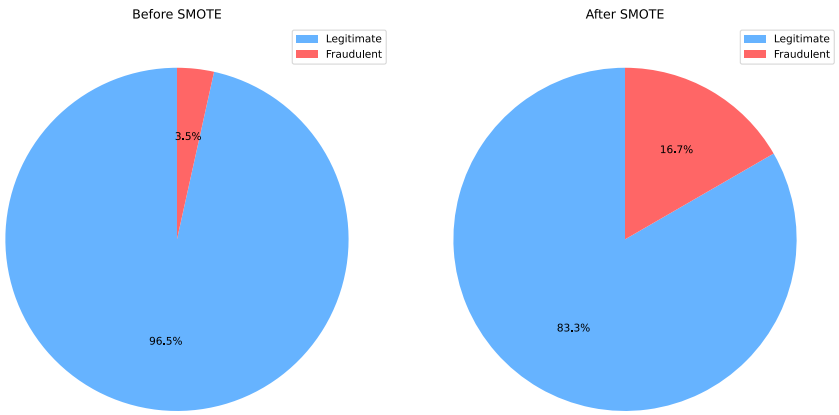


Figure 2.2: Comparison of the class distribution in the training set before (left) and after (right) the application of SMOTE.

2.2.14 Persistence of datasets and preprocessing objects

For reproducibility and for downstream training and evaluation steps, the following artifacts are saved to disk:

- Raw train/test splits: `paths.RAW_TRAIN_PATH`, `paths.RAW_TEST_PATH`.
- Preprocessed (but not SMOTE-balanced) train/test sets: `paths.PREP_TRAIN_PATH`, `paths.PREP_TEST_PATH`.
- SMOTE-balanced preprocessed training set: `paths.SMOTE20_PREP_TRAIN_PATH`.
- Selected feature names: `paths.SELECTED_FEATURES_PATH`.
- Fitted preprocessing objects: imputer (`paths.IMPUTER_PATH`), scaler (`paths.SCALER_PATH`), encoder (`paths.ENCODER_PATH`), variance-threshold (`paths.VAR_THRESH_PATH`), and mutual-information selector (`paths.SELECTOR_PATH`) — all saved using `joblib.dump`.

2.2.15 Design choices and rationale

- **Temporal cyclic encoding:** transforming `TransactionDT` into sine/-cosine features preserves cyclicity (hours, weekdays) without relying on an absolute reference date.
- **Imputation strategies:** choosing *most frequent* for categorical fields and *median* for numeric fields is robust to outliers and preserves typical categorical modalities.
- **RobustScaler:** mitigates the influence of outliers which are common in transaction amounts and derived features.
- **One-hot encoding:** appropriate for tree-based or linear models that operate on explicit feature axes; `handle_unknown="ignore"` ensures safe transformation of unseen categories.
- **VarianceThreshold & mutual-information selection:** two-stage dimensionality reduction first removes near-constant columns (cheap), then retains only the top-k features most informative about the target (scoring with mutual information accounts for non-linear dependencies).
- **SMOTE only on training:** avoids contaminating test data with synthetic samples and ensures a realistic evaluation on naturally imbalanced data.

- **Persistence of fitted objects:** saving imputer/encoder/scaler/selector makes pipeline reproducible and guarantees consistent preprocessing during model training and production inference.

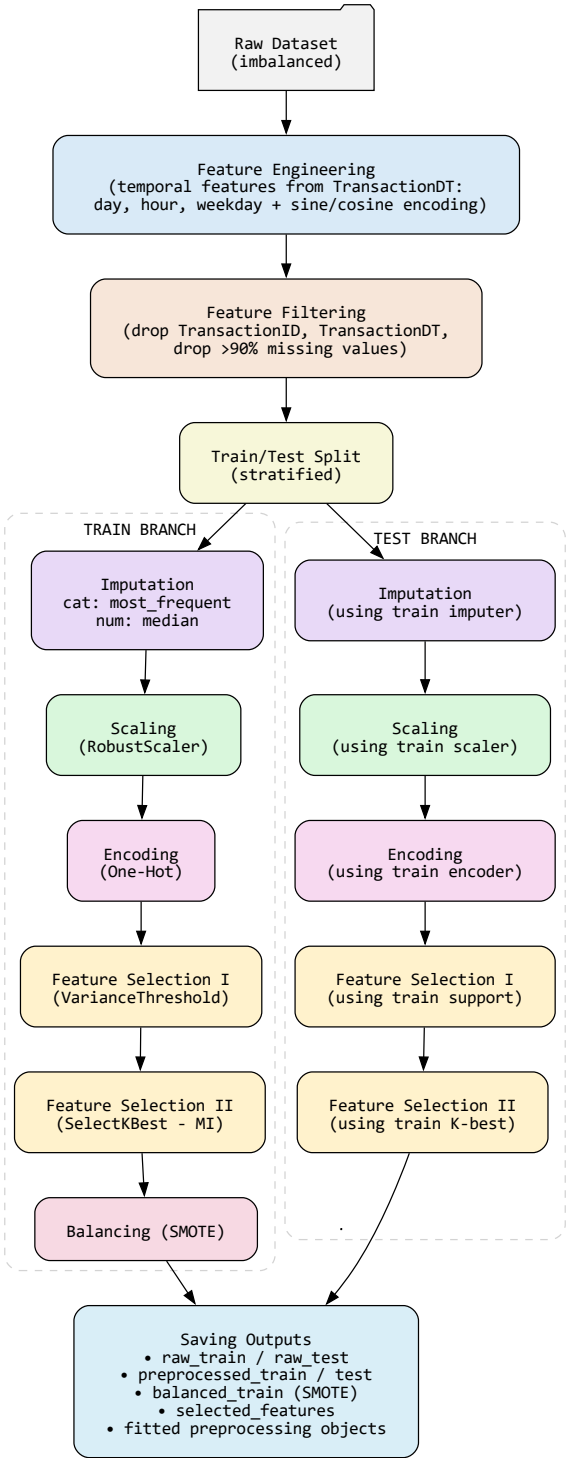


Figure 2.3: Preprocessing pipeline.

3. Classifiers

One of the main goals of this project was to compare several classifiers in order to verify which one had the best performances on the dataset. We chose to compare the following models:

- *K-Nearest Neighbours*, sometimes referred from here on as *KNN*;
- *Gaussian NaiveBayes*, sometimes referred from here on as *NB*;
- *DecisionTree*, sometimes referred from here on as *DT*;
- *RandomForest*, sometimes referred from here on as *RF*;
- *AdaBoost*, sometimes referred from here on as *ADA*;
- *XGBoost*[3], sometimes referred from here on as *XGB*.

3.1 Grid Search

In order to take each classifier to his own best performance on our dataset, we applied a *Grid Search* to find the best hyper-parameters for each of them. We used the `GridSearchCV` class provided by the `scikit-learn` library.

We have set the `cv` parameter of the `GridSearchCV` class to `5`, meaning that a *Cross-Fold Validation* with `k=5` has been done during this process to obtain a more robust choice of the best hyper-parameters.

We have chosen *f1* as scoring metric of the `GridSearchCV` class because we needed a metric that suffered less the strongly imbalanced dataset: the *accuracy* was not the right choice for this exact reason and the *f1* was the perfect balance between *precision* and *recall* (the two metrics on which we focused during the examination of the results).

3.2 Training & Testing

After the identification of the best hyper-parameters, each classifier has been trained on the *rebalanced train set* (*SMOTE* at 20% ratio).

During the training phase, another *Cross-Fold Validation* has been applied in order to obtain more robust validation metrics, this time with the number of folds `k=10` (for the *Grid Search* we used `k=5` to speed up a bit the process).

After the training phase, each classifier has been tested on the *imbalanced test set* and the results of the predictions have been used to obtain the evaluation metrics.

3.3 Evaluation metrics

Before listing all the metrics we used, we need to clarify that we wanted our classifier to correctly classify the fraudulent transactions as main goal (increase *TP* and decrease `FN`); after that, we wanted them to do so with the smallest possible number of non-fraudulent transactions misclassified (decrease *FP*).

For these reasons, the evaluation metrics on which we mainly focused were *precision* and *recall*, even though we have calculated the following evaluation metrics for each classifier.

- *Confusion matrix*: they have been used to obtain a quick look to the performances of each classifier and as "main metric" to obtain more specific ones.
$$\begin{bmatrix} TN & FP \\ FN & TP \end{bmatrix}$$
- *Accuracy*: even if it is not so suitable for strongly imbalanced datasets (as ours is), it is however a good metric to quickly evaluate the general performance of a classifier.
- *Balanced accuracy*: defined as *(specificity + recall)/2*, it is more robust to imbalanced datasets then the "normal" *accuracy*.
- *Precision*: one of the two metrics on which we focused to improve classifiers performances.
- *Weighted precision*: the weighted version of *precision*.
- *Recall*: the metric on which we firstly focused to improve classifiers performances.
- *Weighted recall*: the weighted version of *recall*.
- *F1*: mainly used as scoring metric of the *Grid Search* to order models considering both *recall* and *precision*.
- *Weighted f1*: the weighted version of *f1*.
- *ROC_AUC*: the area under the *ROC Curve* of each classifier, mainly used at the end of the project to compare all the classifiers' performances.
- *PR_AUC*: the area under the *PR Curve* of each classifier, mainly used at the end of the project to compare all the classifiers' performances.

3.3.1 Acceptable Level of Performance

The *ROC Curves* graph and the *PR Curves* graph (as we will see in section 4.1) gave us some interesting results about the adaptation of the classifiers to different values of the *decision threshold*[1]: if we modify such threshold, all the classifiers improve their own *TPR*, but always worsening *FPR*. Consequently, we decided that we consider "a good classifier" a model that has at least a *TPR* of `0.8` (that is a classifier which can identify correctly the 80% of fraudulent transactions), because the main goal of a fraud detector should be to identify as much frauds as possible, even accepting a considerable amount of misclassified non-fraud transactions[2].

We called **Acceptable Level of Performance** (**ALP**) the condition in which a classifier reaches `tpr = 0.8`.

We called **ALP_threshold** the threshold at which a classifier reaches the *ALP*.

We called **ALP_FPR** the *FPR* of a classifier when it reaches *ALP*.

We made a further analysis of the *ROC Curves* graph to obtain the *ALP* for each classifier[3] and find the classifier with the smallest *ALP_FPR*, plotting a *Thresholds VS TPR* graph and a *Thresholds VS FPR* graph. Such graphs helped us to identify the model with the best trade-off between *TPR* and *FPR* at *ALP*.

[1] With *decision threshold* we mean the value `t` used by each classifier to consider as 1 or 0 the probability of a tuple of being positive. By default, `scikit-learn` uses `t=0.5`, meaning that, if a tuple has a probability `p >= 0.5` of being positive, it will be classified as positive (label 1). From now on, we will refer to the *decision threshold* simply as *threshold*.

[2] Obviously, the *ALP* has been chosen reasonably to obtain good performances, but is not mandatory to choose our value: it is just an example to show that the classifiers can reach even better performances than those with the default threshold. The *ALP* could be increased more to obtain even better *recall* (but with more *FP*), or could be decreased to obtain less *FP* (but with less *recall*), depending on what the final user of the application wants.

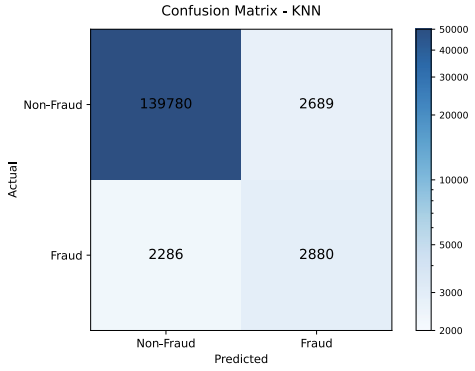
[3] Due to a lack of time, the classifier have not been re-trained using their own *ALP_threshold*: we have just found the *ALP* for each classifier and reported them in the present paper. The models that the user finds in our real world application are trained with the `scikit-learn` default threshold `t = 0.5`.

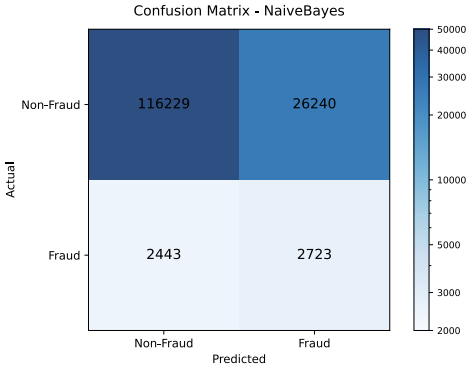
4. Experimental Results

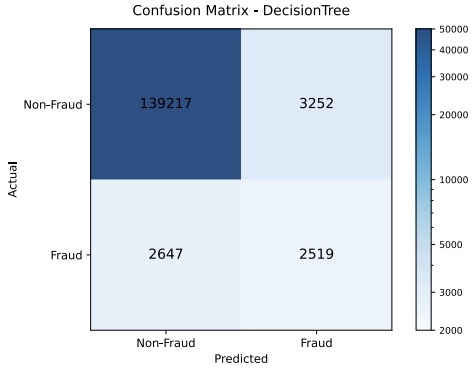
In this chapter we include both the results of each classifier (with the comparison among all of them), the results of the *t-tests* and the results of the *XAI methods* we used to explain the classifiers.

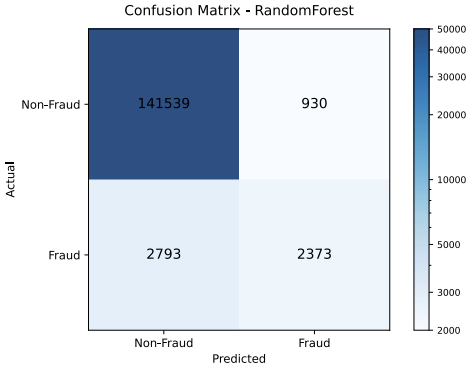
4.1 Results of the Models

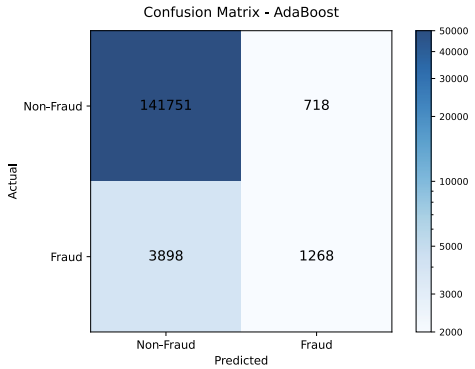
The results of each model are here presented individually as *confusion matrices* of their predictions.

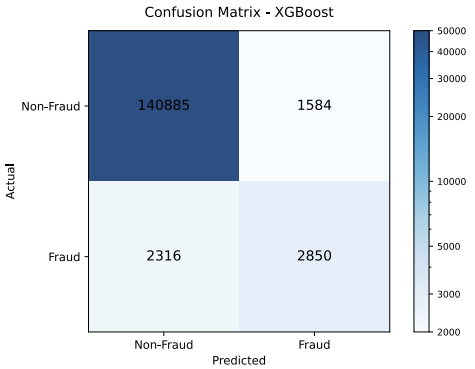












4.2 Comparison of the Models

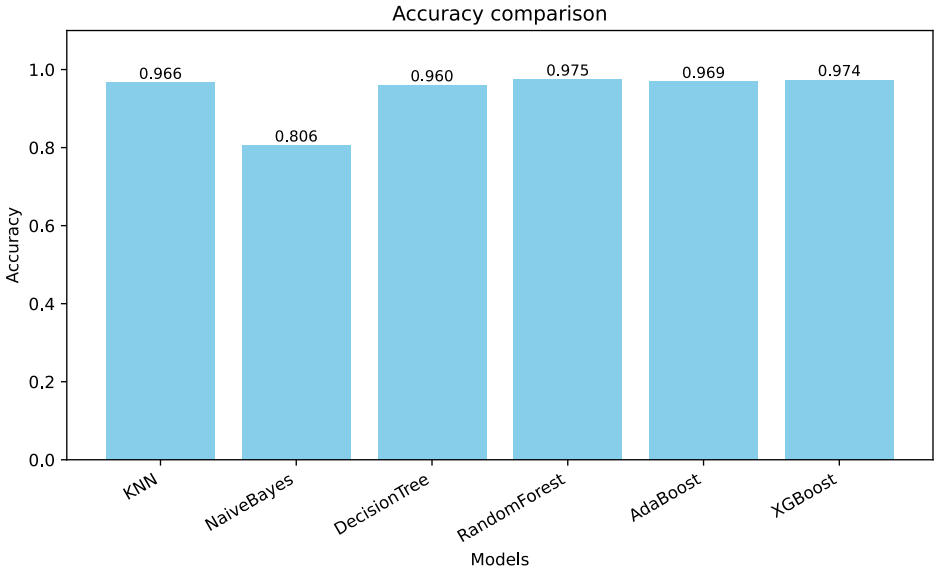


Figure 4.1: Comparison of the *accuracy* among all the classifiers:

The *accuracy* metric is **really high for all classifiers**, with only NaiveBayes a bit lower than the others.

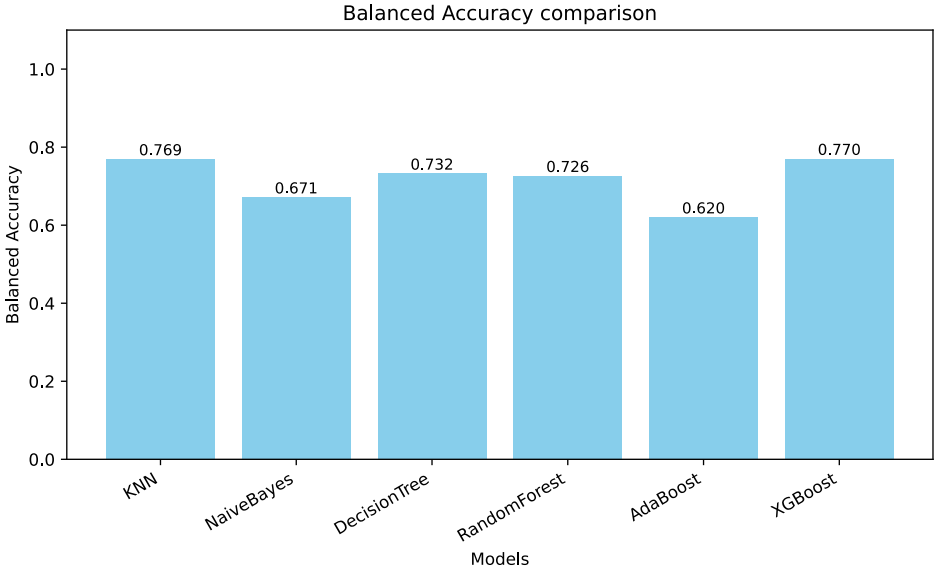


Figure 4.2: Comparison of the *balanced accuracy* among all the classifiers.

The *balanced accuracy* metric is **lower in all the classifiers** respect to "normal" *accuracy*: AdaBoost becomes the worst model.

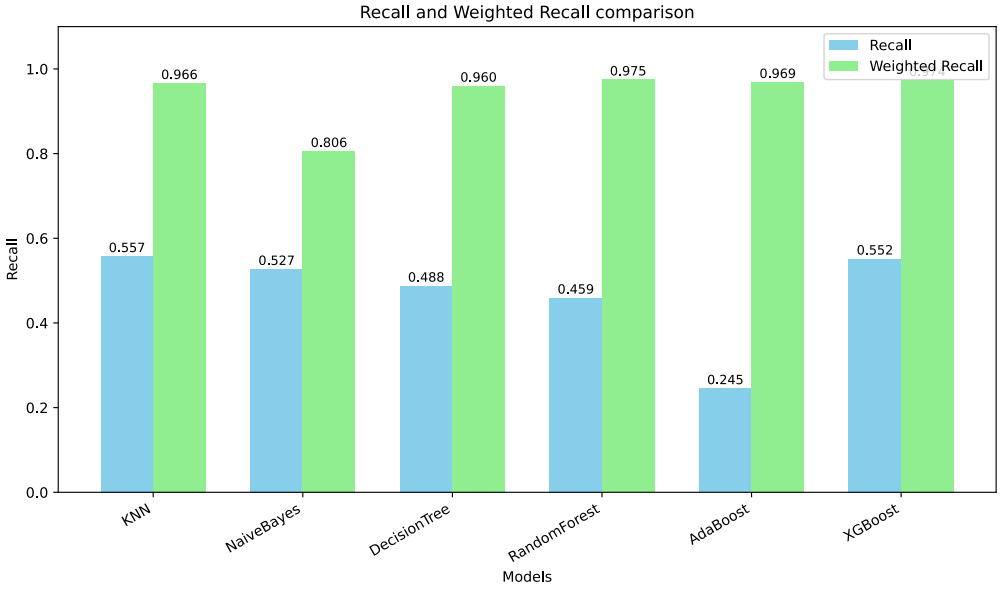


Figure 4.3: Comparison of the *recall* among all the classifiers.

The *recall* metric gives us larger differences between model: **Knn** and **XGBoost** are the best ones, followed by NaiveBayes not so far. AdaBoost is instead the worst one by far. Instead, all classifiers are comparable in *weighted recall*, with only NaiveBayes a bit worse than the others.

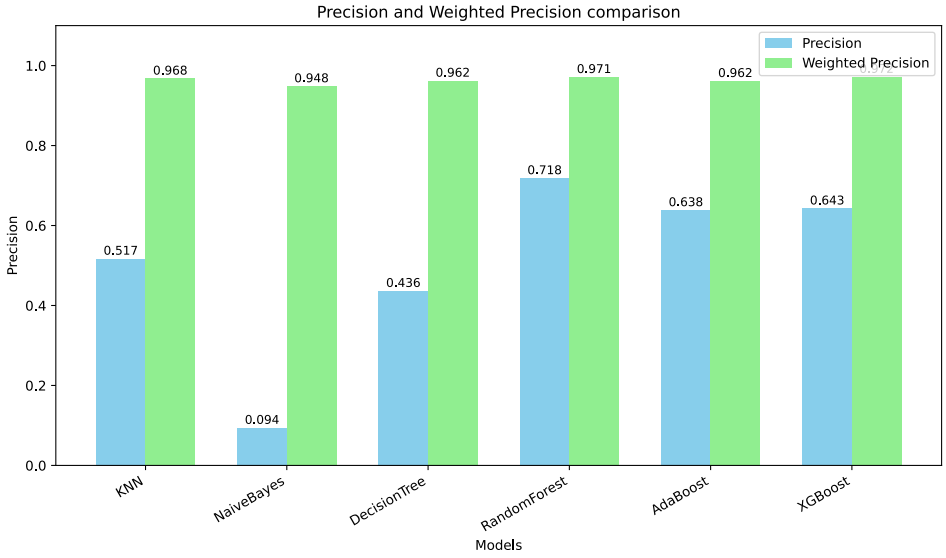


Figure 4.4: Comparison of the *precision* among all the classifiers.

The *precision* metric enlarges differences respect to *recall* and changes hierarchy among some classifiers: **RandomForest** is the best one, followed not so far by **XGBoost** and Adaboost. NaiveBayes is instead the worst one by far. Instead, all classifiers are comparable in *weighted precision*.

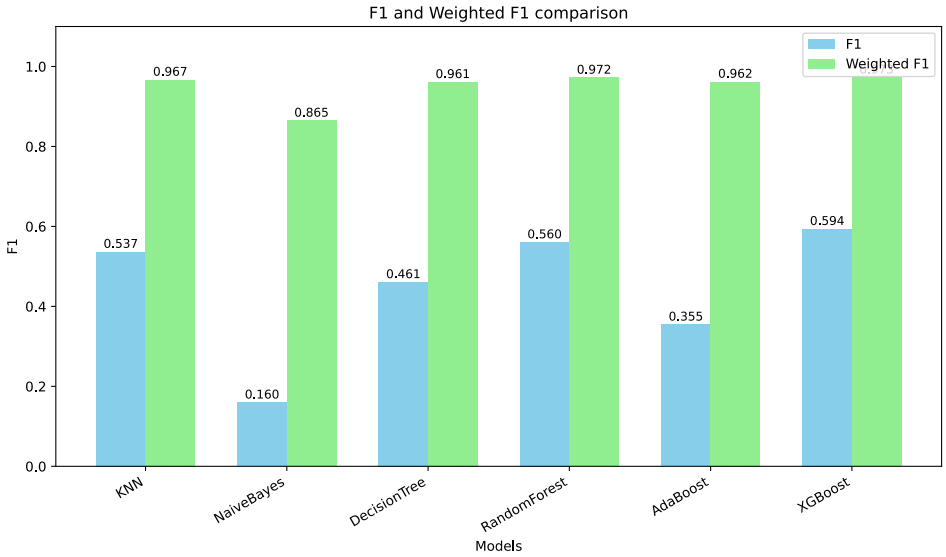


Figure 4.5: Comparison of the *f1* among all the classifiers.

The *f1* metric balances the different hierarchies given by *recall* and *precision*: **XGBoost** is the best, closely followed by RandomForest and KNN a bit further. DecisionTree and AdaBoost have mid-performances. NaiveBayes is by far the worst. Instead, all classifiers are comparable in *weighted f1*, with only NaiveBayes a bit worse than the others.ith only NaiveBayes a bit worse than the others.

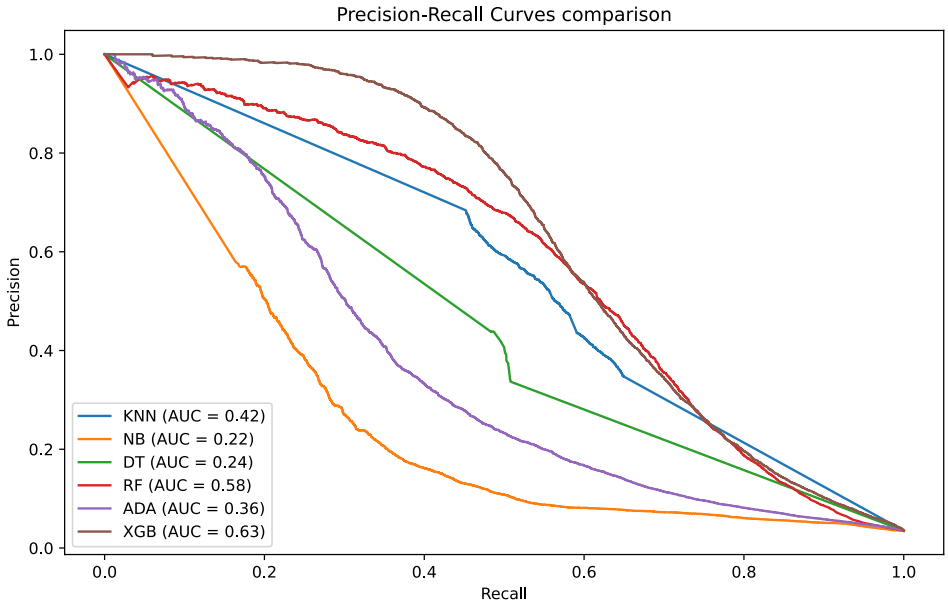


Figure 4.6: *PR Curves* of all the classifiers.

The *PR Curves* say us that the best[1] model, yet since the beginning of the graph, is **XGBoost**, followed by RandomForest (which is even the best in the last part of the graph) and KNN.

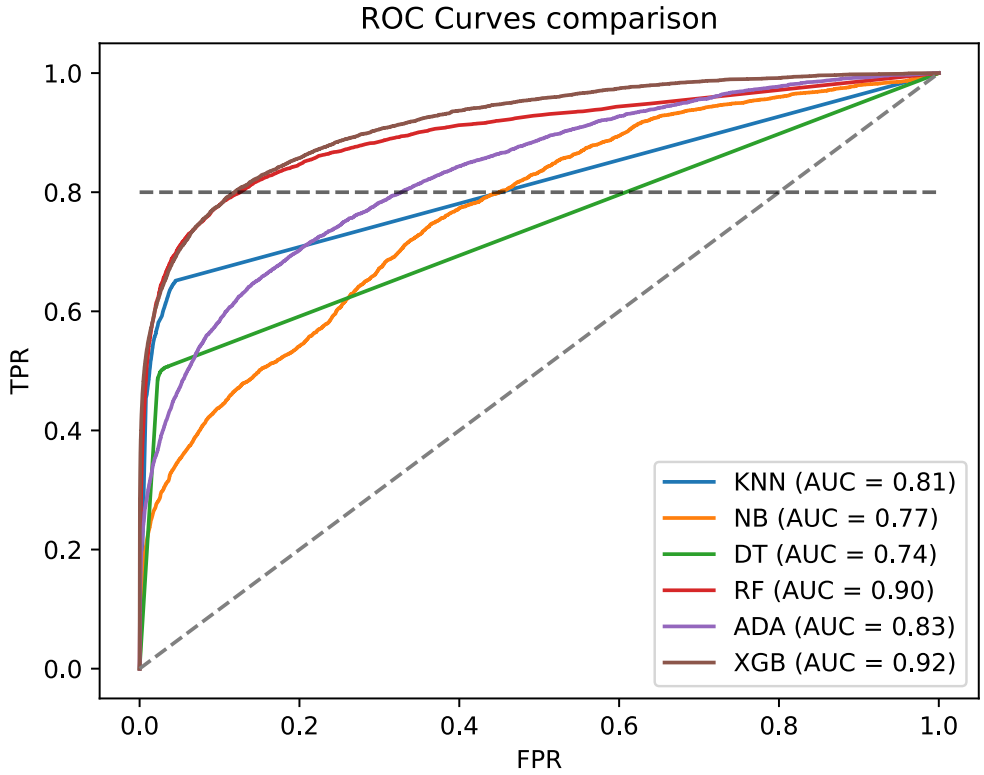


Figure 4.7: *ROC Curves* of all the classifiers.

The *ROC Curves* say us that the best[2] model is **XGBoost**, followed immediately by RandomForest.

[1] We recall that in the *PR graph* "the best" means the curve that is the nearest to the *upper right* angle of the graph.

[2] We recall that in the *ROC graph* "the best" means the curve that is the nearest to the *upper left* angle of the graph.

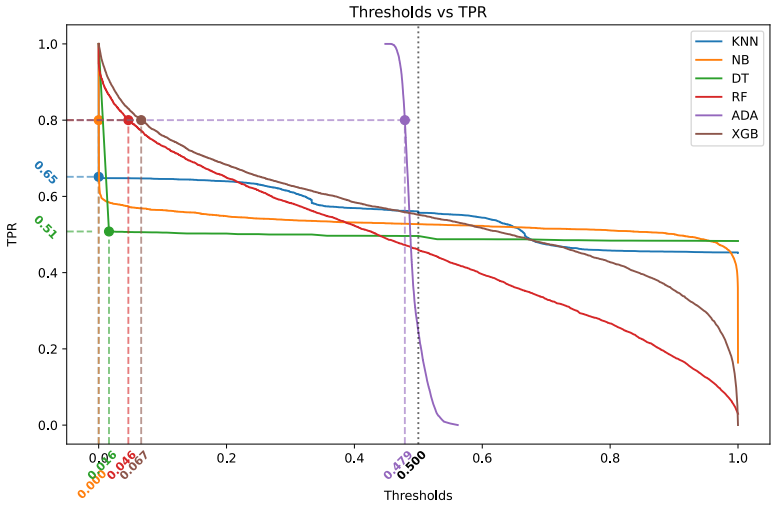


Figure 4.8: Thresholds used to compute the *ROC* values plotted against the *TPR* values (the dots on the curves represents the *ALP* for each classifier).

We can see how KNN and DecisionTree cannot reach the *ALP*[3]. We then see that AdaBoost reaches *ALP* really quickly (*ALP_threshold* near 0.5), while XGBoost and RandomForest need a very low *ALP_threshold*.

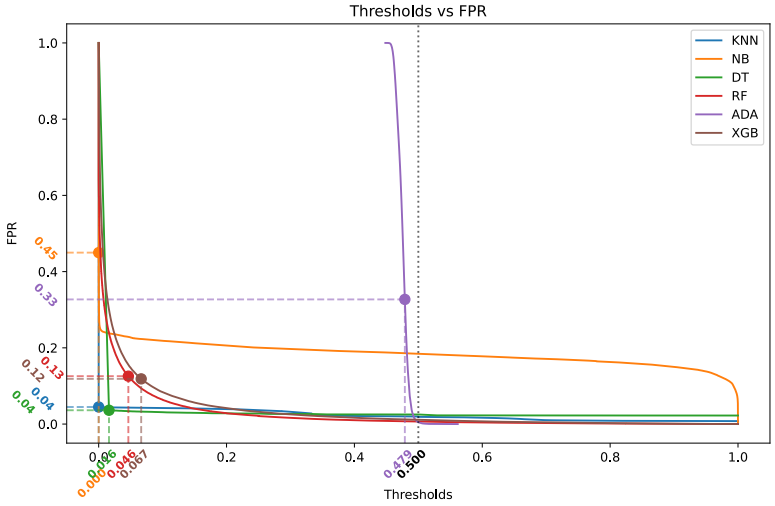


Figure 4.9: Thresholds used to compute the *ROC* values plotted against the *FPR* values (the dots on the curves represents the *ALP* for each classifier).

We can see that at the *ALP_thresholds* the classifiers with the lowest *ALP_FPR* are **XGBoost**, **RandomForest**, KNN and DecisionTree, but the last two classifiers could not reach *ALP*.

[3]The straight lines from the blue and green dots the point `(0,1)` depend on the implementation of `matplotlib`, which draws on the graphs all the input points and then connects them with straight lines.

4.3 Statistical Model Comparison

To rigorously evaluate and compare the performance of the six candidate models, we employed a **10-fold stratified** *cross-validation (CV)* procedure. The class imbalance inherent in the training data has been addressed applying *Synthetic Minority Over-sampling TEchnique (SMOTE)* to the training set within each fold.

The performance of each model on each fold was recorded for two key metrics: *f1-score* (which balances precision and recall for the minority class) and *ROC AUC* (which measures the model ability to discriminate between classes).

To determine if the observed performance differences between models were statistically significant or merely due to chance, we conducted a **paired t-test** on the 10-fold scores for every possible model pair.

The results of these tests are comprehensively visualized in two sets of heatmaps.

4.3.1 *p-value* Significance Heatmaps

This first set of heatmaps visualizes the *p-value* for each *paired t-test*. The *p-value* indicates the **probability that the observed difference in performance is due to random variation**. We used a significance level $(\alpha) = 0.05$:

- a *p-value* < 0.05 suggests a **statistically significant** difference between the two models;
- a *p-value* ≥ 0.05 suggests that we **cannot conclude** a statistically significant difference exists.

To make the charts intuitive, we used a *Red-Yellow-Green Reversed* colormap (`RdYlGn_r`):

- **green (low *p-value*)** represents a highly significant difference (a "good" statistical result);
- **red (high *p-value*)** represents an insignificant difference.

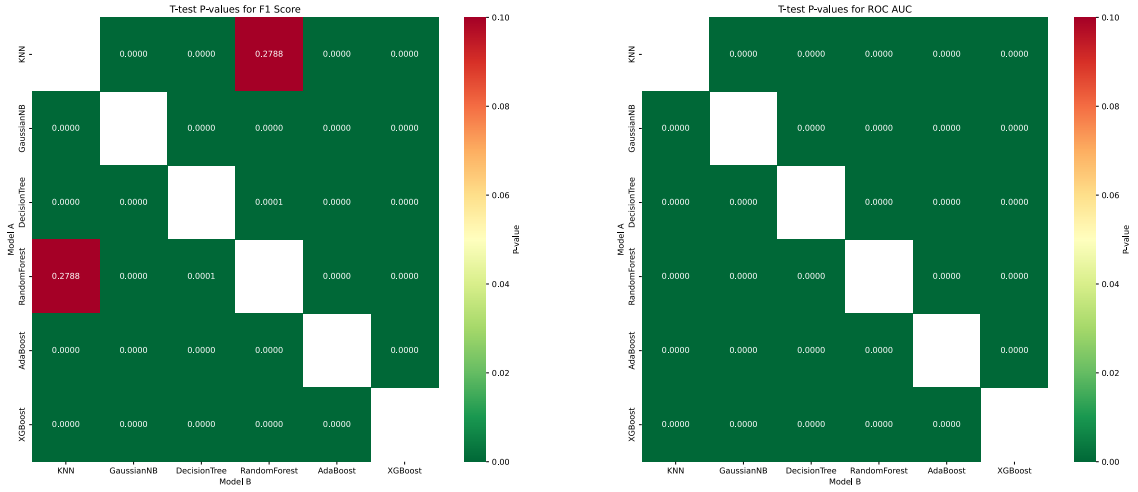


Figure 4.10: *p-value* significance heatmaps for *f1-score* (left) and *ROC AUC* (right).

As clearly shown in the heatmaps, nearly all model comparisons yielded *p-values* well below 0.05 (formatted as `0.0000`), confirming that the performance differences are statistically robust.

The only exception is the *f1-score* comparison between **KNN and RandomForest**, which, with a *p-value* of `0.2788`, shows no significant performance difference for that specific metric.

4.3.2 *t-statistic* Performance Heatmaps

This second set of heatmaps visualizes the *t-statistic* for each comparison, but only for pairs that were found to be statistically significant (i.e., where $p - value < 0.05$). Cells with insignificant *p-values* are not present in the following graphs.

The *t-statistic* shows the direction and magnitude of the performance difference.

We used a *Red-Yellow-Green* colormap (`RdYlGn`):

- **green (Positive *t-statistic*)** says that Model A (row) is **statistically superior** to Model B (column);
- **red (Negative *t-statistic*)** says that Model A (row) is **statistically inferior** to Model B (column).

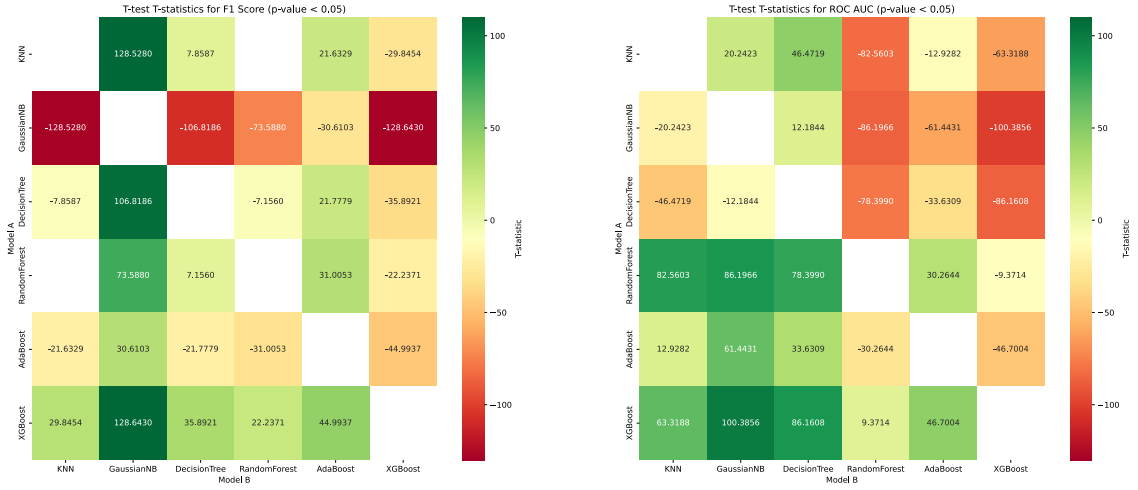


Figure 4.11: *t-statistic* performance heatmaps for *f1-score* (left) and *ROC AUC* (right), masked for $p \geq 0.05$.

4.3.3 Key Findings and Model Ranking

The *t-statistic* heatmaps provide a clear and definitive ranking of the models.

1. **Best Model: XGBoost**

   The XGBoost row is entirely green, and its column is entirely red in both *f1-score* and *ROC AUC* heatmaps: this indicates that XGBoost **statistically outperformed all other five classifiers** across both metrics.

2. **Worst Models: NaiveBayes** and **DecisionTree**

   Conversely, ad regards *f1-score*, the NaiveBayes row is entirely red, while as regards *ROC AUC*, the whole DecisionTree row is red: this shows that NaiveBayes has been **statistically outperformed by all other models** on *f1-score* metric and the same goes for DecisionTree regarding *ROC AUC*.

3. **Relative Rankings**

   The remaining models can be ranked observing their rows and columns. For example, AdaBoost (a green row) and RandomForest (a mostly green row) clearly outperform DecisionTree and KNN.

Conclusion: The paired statistical analysis provides strong evidence to conclude that **XGBoost** is the optimal model for this fraud detection task, demonstrating a statistically significant and superior performance in discriminating between fraudulent and non-fraudulent transactions.

4.4 Models Explainability

In order to understand better the predictions of the models and provide insights into the factors influencing the classification, several explainability techniques were applied to all the trained classifiers.

Overall, the combination of all techniques enables both global and local interpretability of the classifiers, making it possible to analyze not only which features are important, but also how they contribute to specific predictions.

The figures generated for each method are saved in the `./explanations` folder.

4.4.1 Feature Importances

For models supporting intrinsic feature importance metrics, such as DecisionTree, RandomForest and XGBoost, the top features contributing to the classification were computed and visualized. Horizontal bar plots highlight the most relevant features, providing an immediate understanding of which variables drive the predictions.

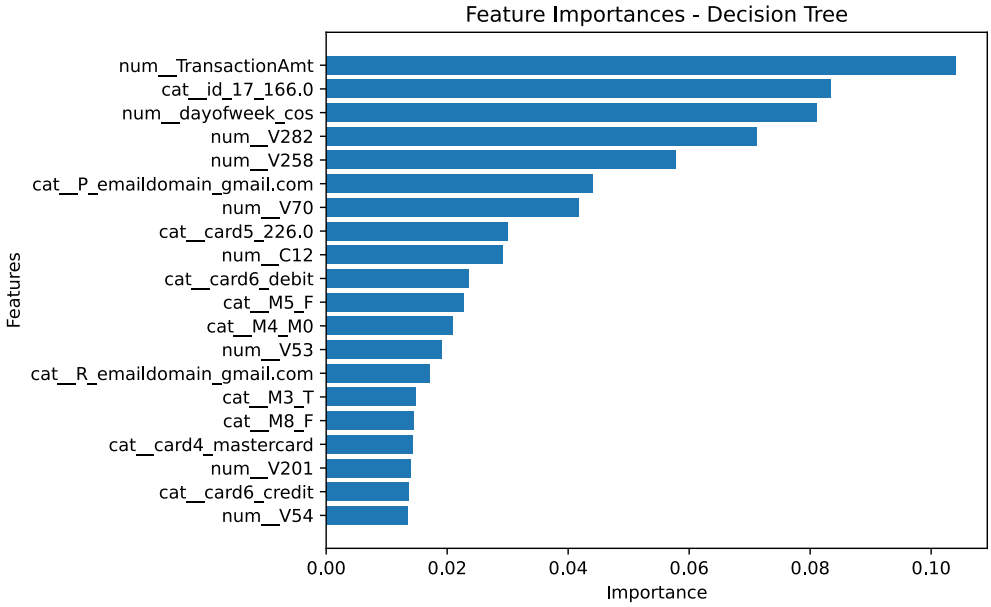


Figure 4.12: Top 20 features ranked by intrinsic feature importance as computed by the DecisionTree classifier on the preprocessed training set.

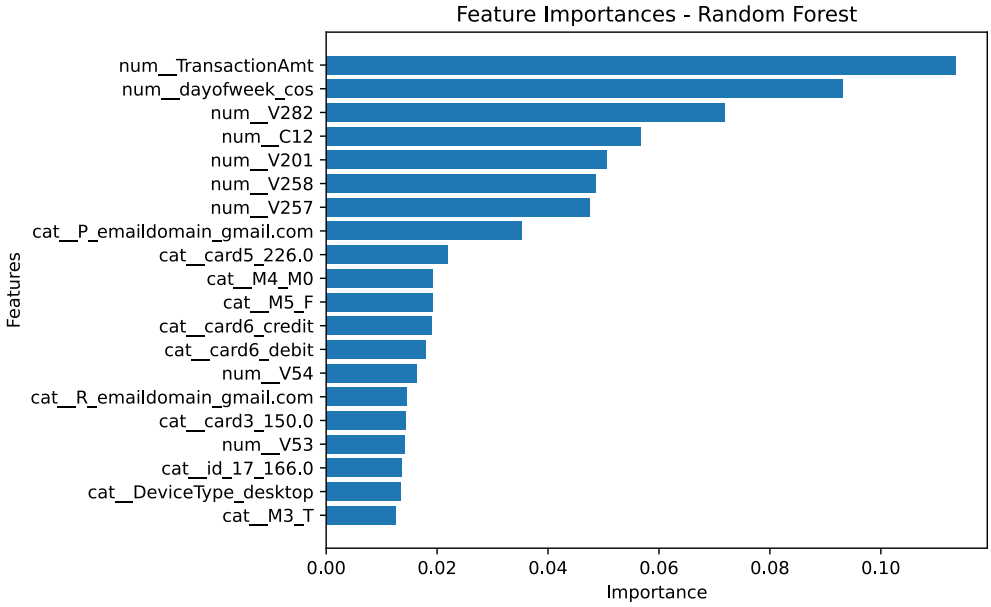


Figure 4.13: Top 20 features ranked by intrinsic feature importance as computed by the RandomForest classifier on the preprocessed training set.

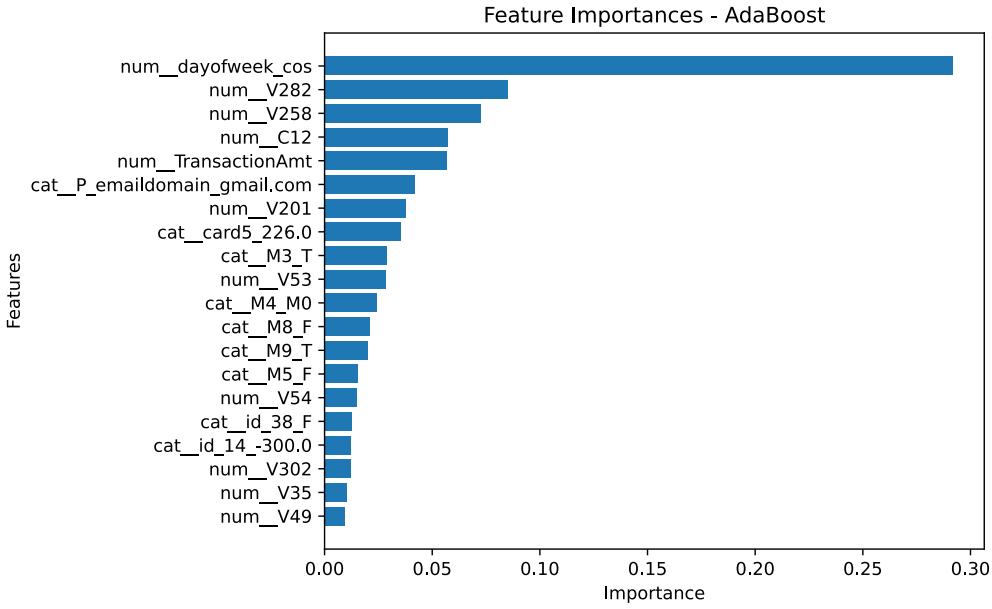


Figure 4.14: Top 20 features ranked by intrinsic feature importance as computed by the AdaBoost classifier on the preprocessed training set.

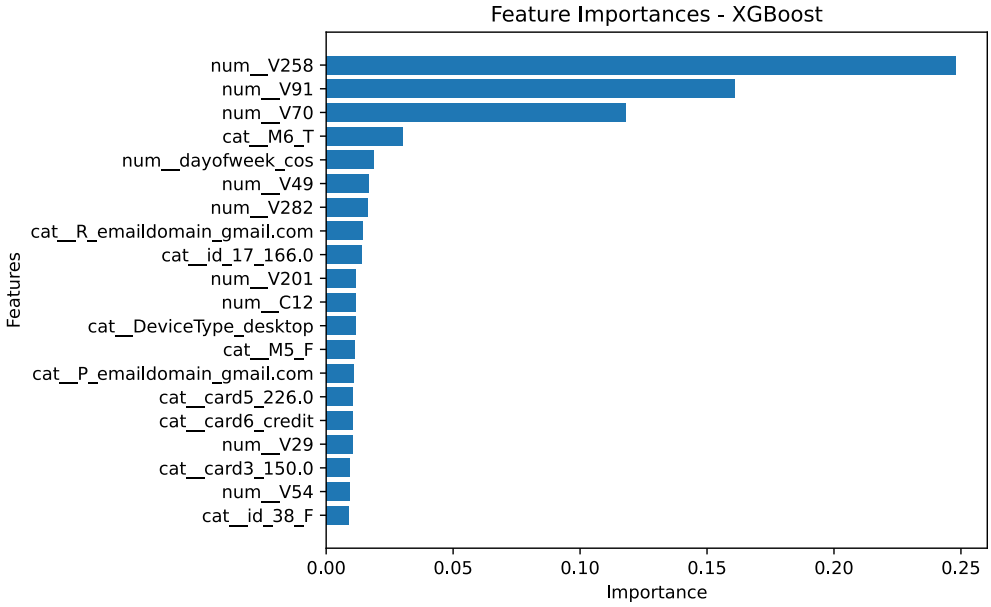


Figure 4.15: Top 20 features ranked by intrinsic feature importance as computed by the XGBoost classifier on the preprocessed training set.

4.4.2 SHAP Values

SHapley Additive exPlanations (SHAP) values were computed, whenever feasible, using a stratified sample of the testing set. SHAP values allow to quantify the contribution of each feature to individual predictions as well as to the overall behavior of the model. Summary plots were generated to illustrate the global impact of features across the dataset.

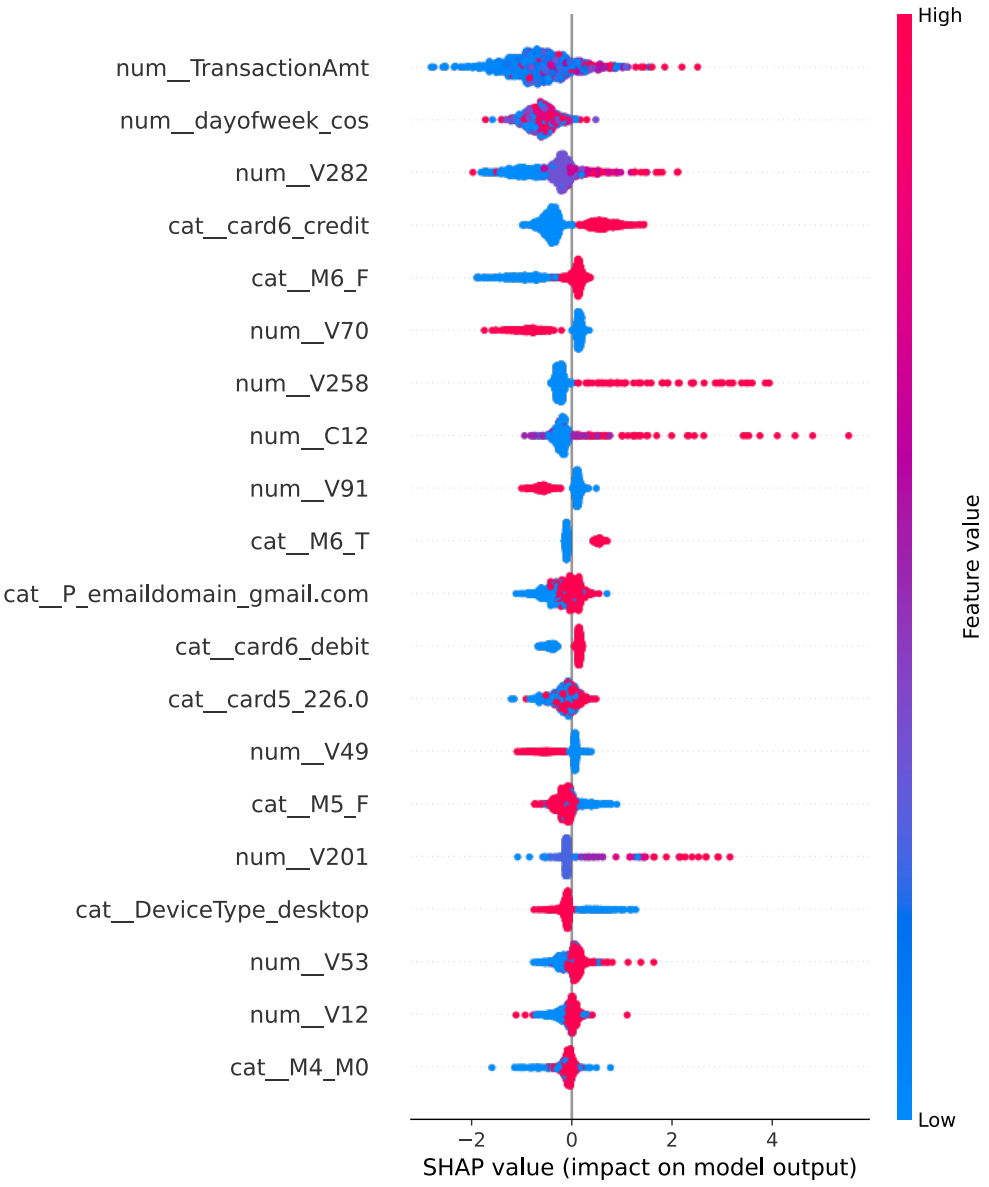


Figure 4.16: SHAP summary plot for XGBoost classifier, showing the global impact of the top features on the model predictions. Positive values indicate a contribution towards predicting fraud, while negative values indicate a contribution towards predicting legitimate transactions.

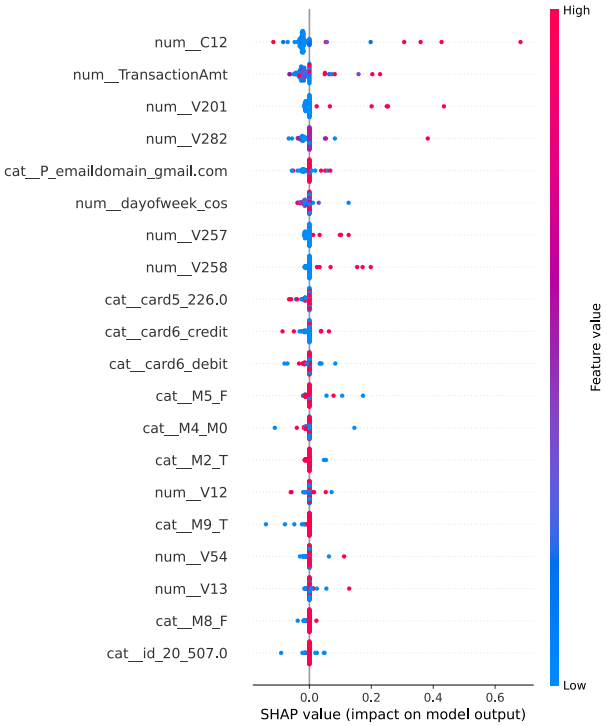


Figure 4.17: SHAP summary plot for K-NN classifier.

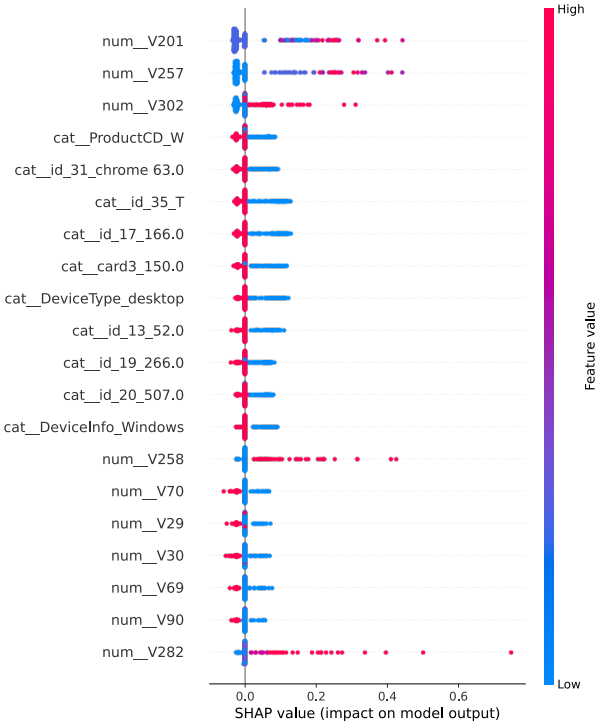


Figure 4.18: SHAP summary plot for NaiveBayes classifier.

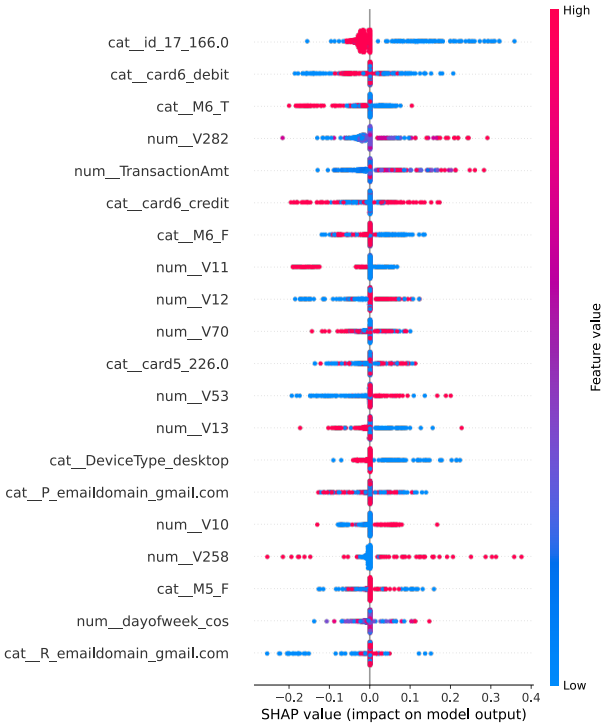


Figure 4.19: SHAP summary plot for DecisionTree classifier.

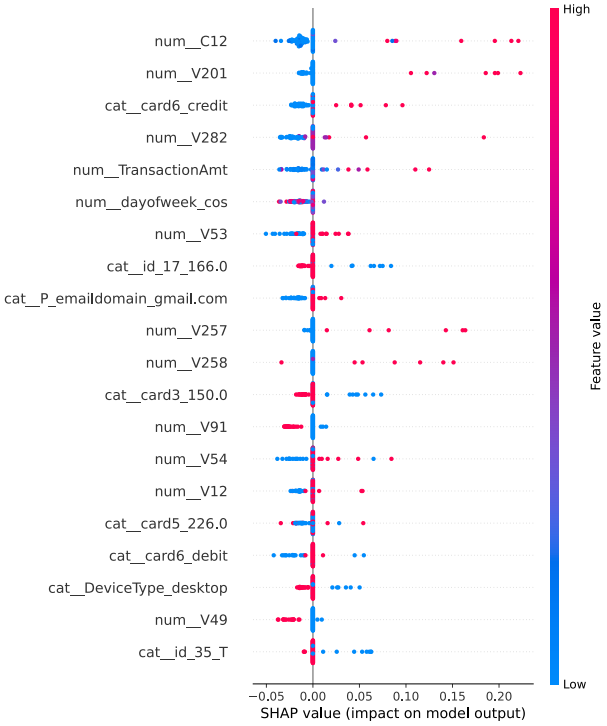


Figure 4.20: SHAP summary plot for RandomForest classifier.

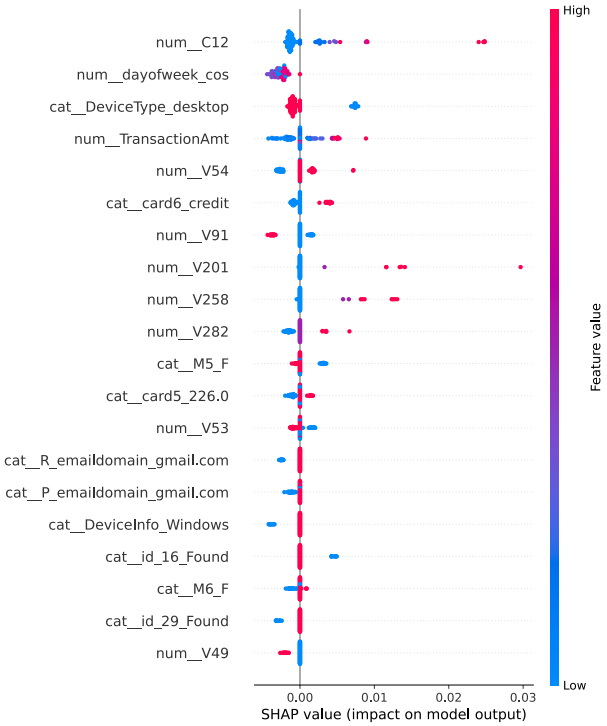


Figure 4.21: SHAP summary plot for AdaBoost classifier.

4.4.3 Permutation Feature Importance

Permutation importance was also employed to estimate the effect of features on the model performance. By randomly shuffling the values of one feature at a time and measuring the decrease in prediction accuracy, the importance of features is inferred. This approach is model-agnostic and complements both intrinsic feature importance and SHAP analyses.

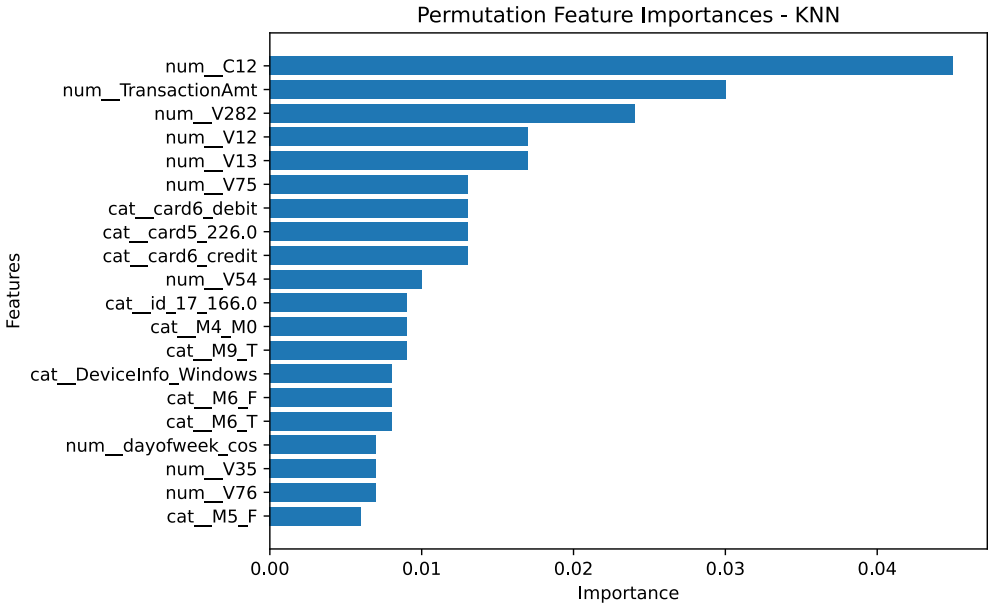


Figure 4.22: Permutation feature importances for K-NN classifier.

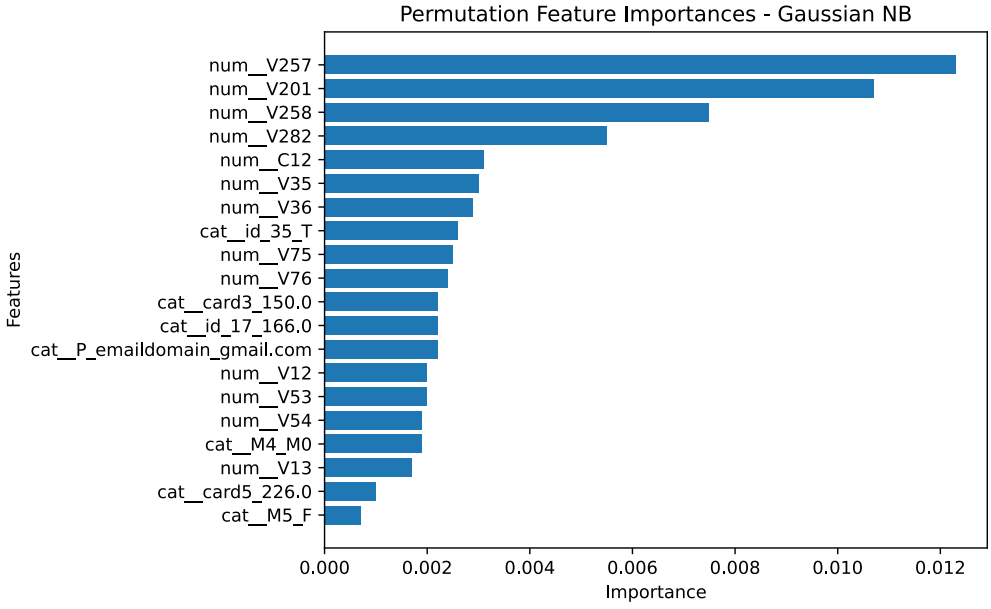


Figure 4.23: Permutation feature importances for NaiveBayes classifier.

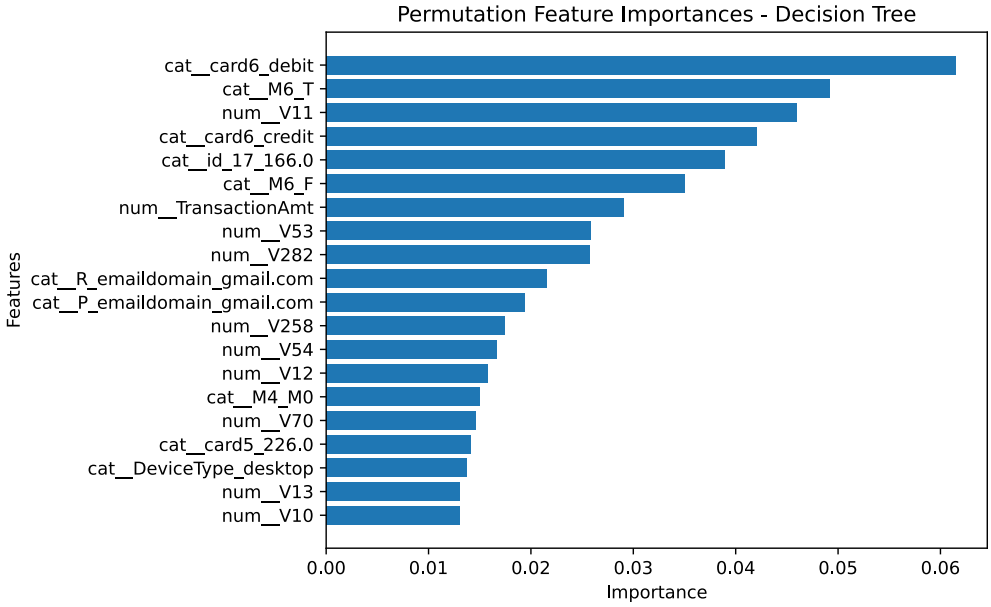


Figure 4.24: Permutation feature importances for DecisionTree classifier.

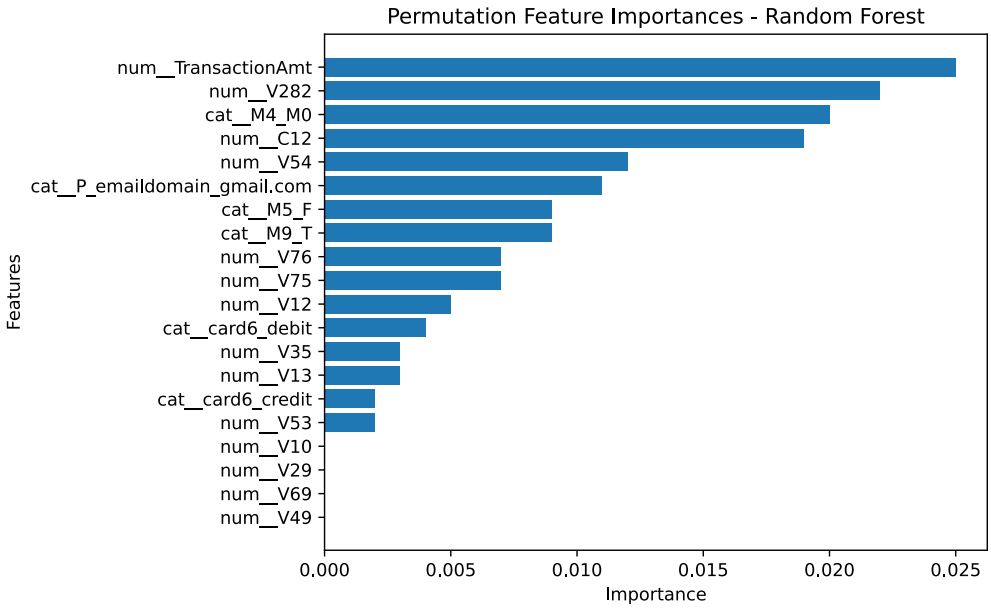


Figure 4.25: Permutation feature importances for RandomForest classifier.

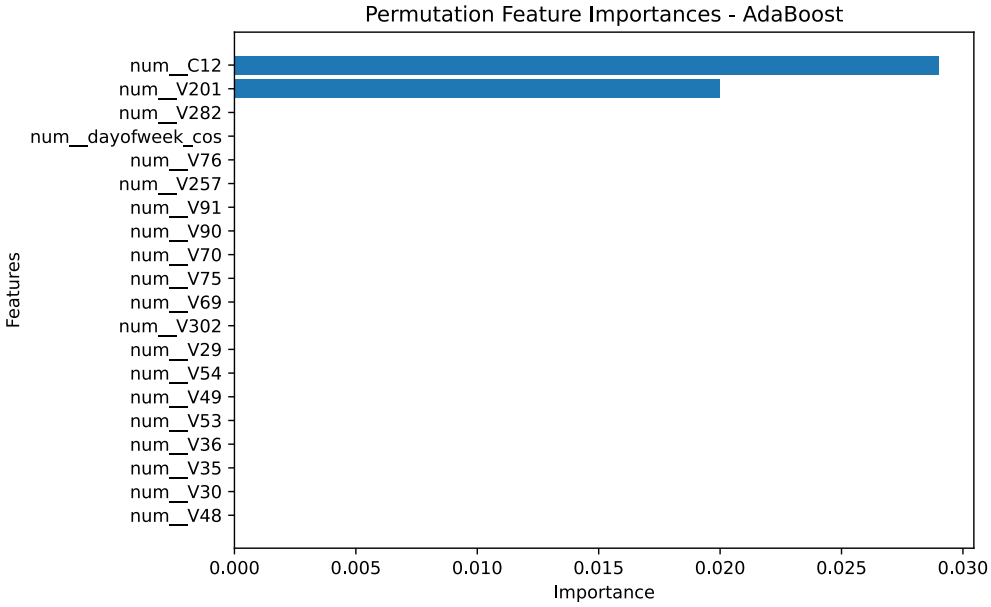


Figure 4.26: Permutation feature importances for AdaBoost classifier.

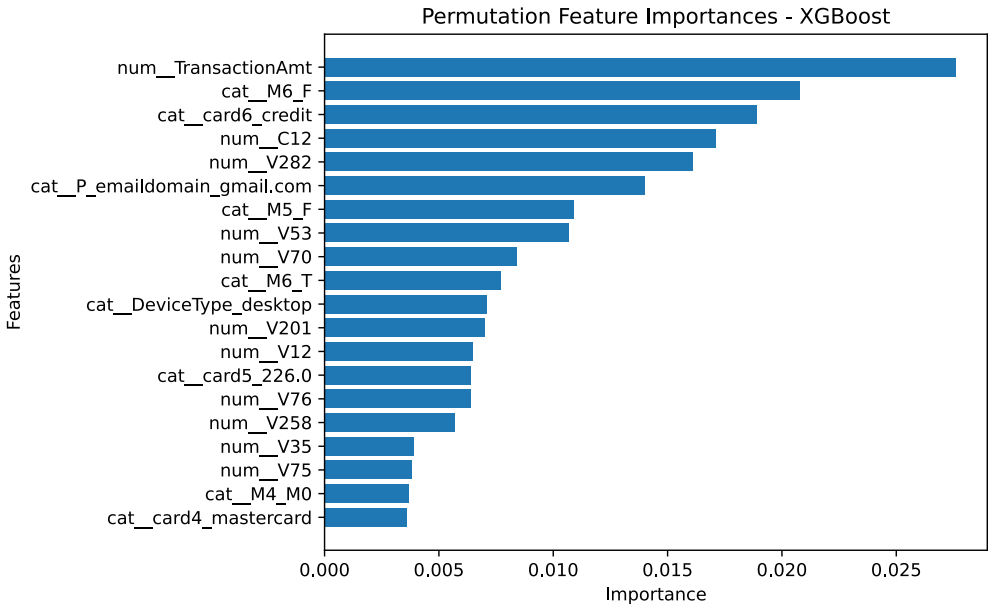


Figure 4.27: Permutation feature importances for XGBoost classifier.

4.4.4 Rule Extraction via Surrogate Models

To further enhance interpretability, surrogate decision trees of depth 3 were trained to approximate the classifiers predictions. These surrogate models allow for extracting simple decision rules that can explain the original complex model behavior in an understandable form. The rules were saved as text files.

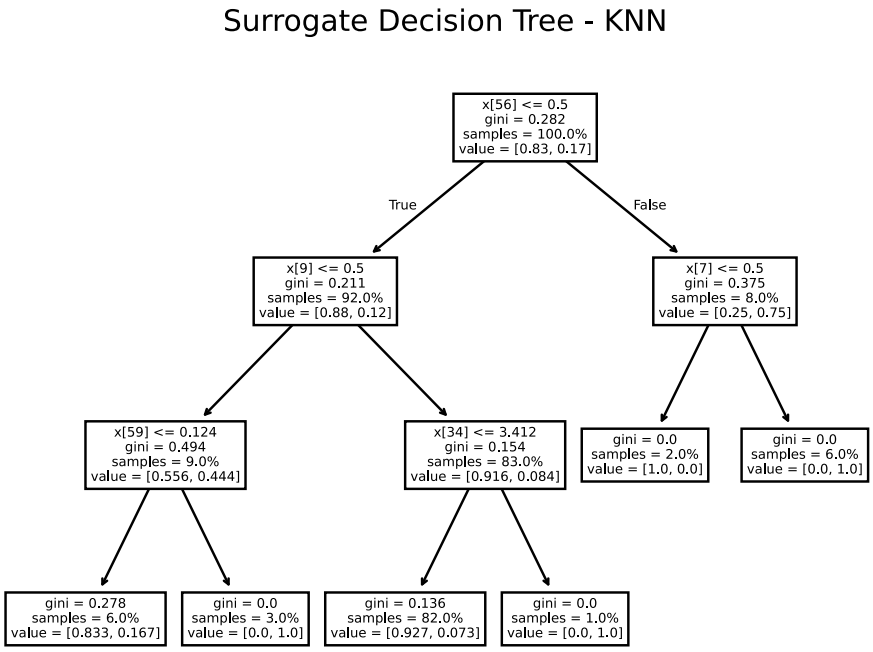


Figure 4.28: Example of extracted decision rules from K-NN classifier.

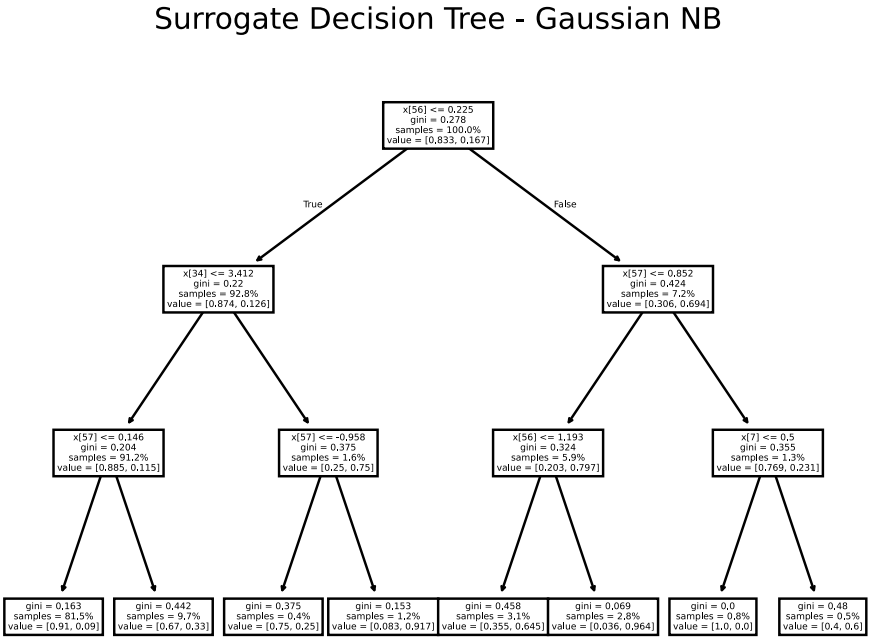


Figure 4.29: Example of extracted decision rules from NaiveBayes classifier.

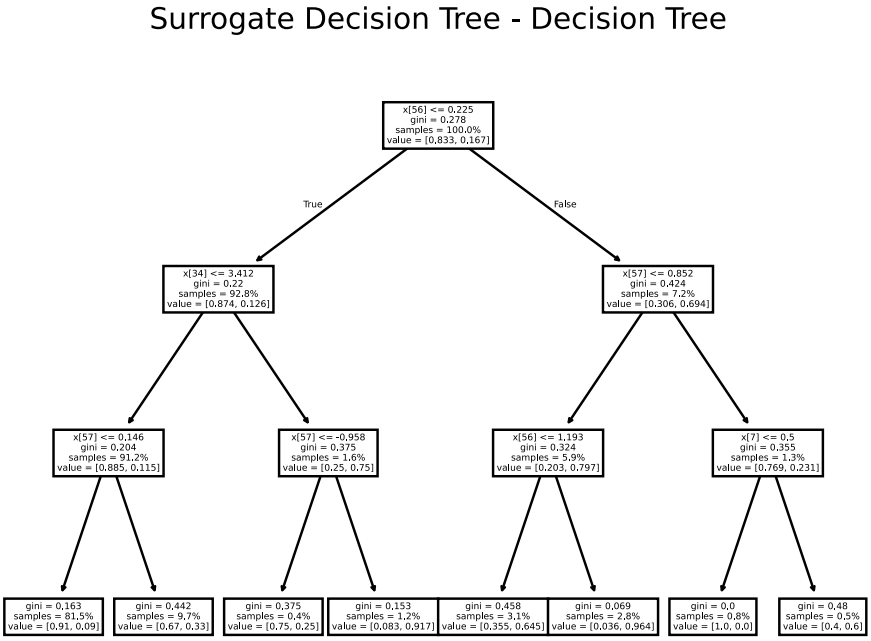


Figure 4.30: Example of extracted decision rules from DecisionTree classifier.

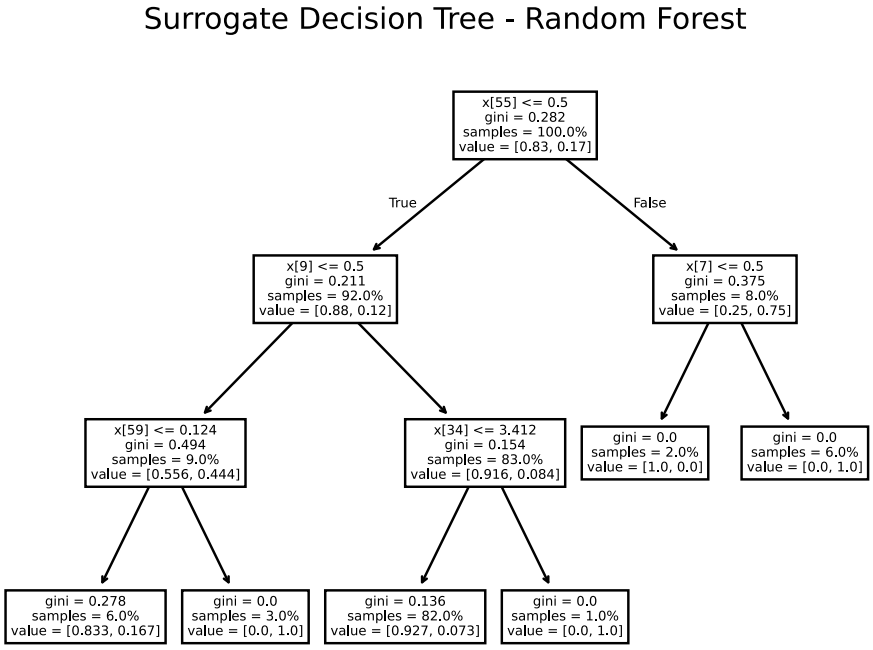


Figure 4.31: Example of extracted decision rules from RandomForest classifier.

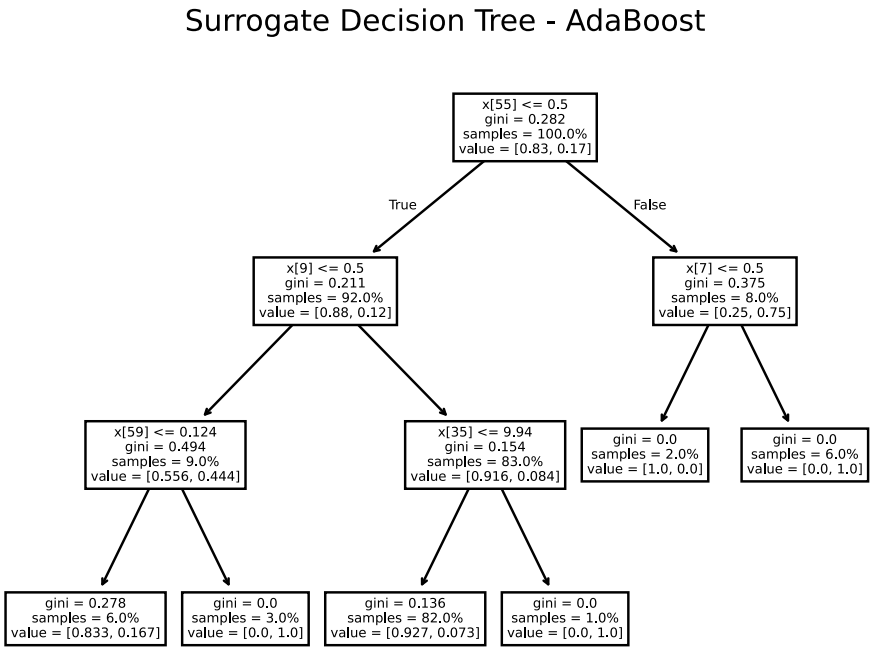


Figure 4.32: Example of extracted decision rules from AdaBoost classifier.

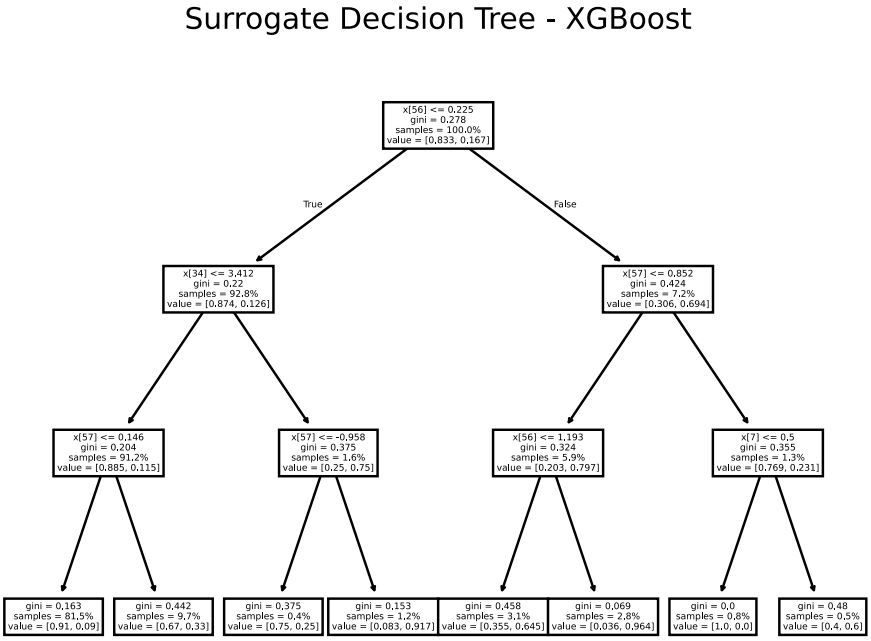


Figure 4.33: Example of extracted decision rules from XGBoost classifier.

5. Real-world Application

A practical scenario for the application of the models developed in this project is an online payment system, where transactions are continuously performed and the risk of fraudulent activity is high. The aim of the application is to assist financial institutions, e-commerce platforms, or payment processors in detecting potentially fraudulent transactions in real time.

The implemented prototype is a simple web interface realized with `Streamlit`, allowing a user to manually input transaction details.

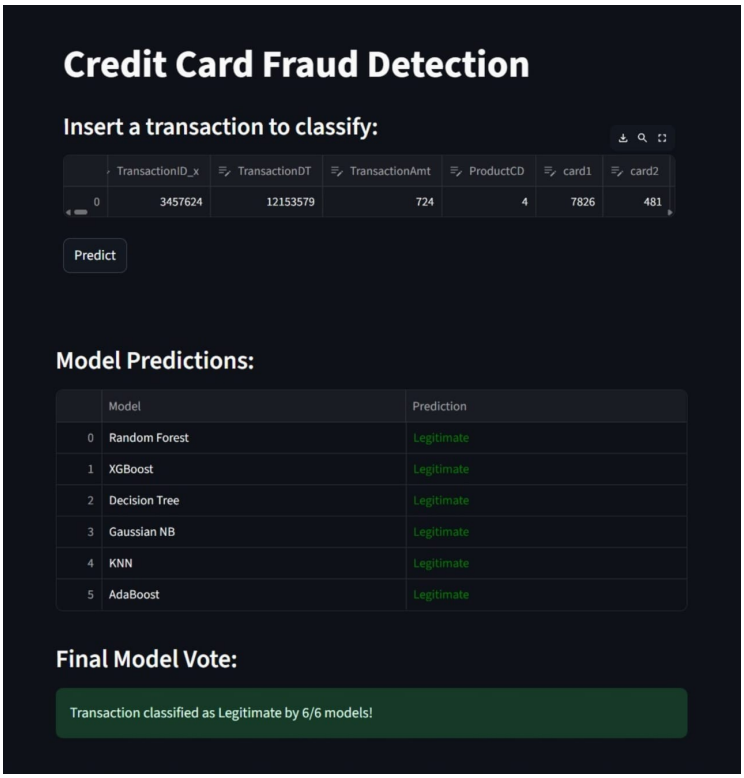


Figure 5.1: Screenshot of the *Streamlit* interface used for real-time fraud detection (users can input a transaction and see predictions from all models along with interpretability explanations).

Once the transaction data is submitted, the application performs the following steps.

- **Pre-processing:** The input is transformed using the same preprocessing pipeline applied to the training data. This includes imputation of missing values, scaling, temporal feature engineering (e.g., hour and day of the week), and feature selection based on variance threshold and mutual information.
- **Classification:** Six different classifiers are applied simultaneously: *Deci-*

sion Tree, *Random Forest*, *Naive Bayes*, *K-Nearest Neighbors*, *AdaBoost*, and *XGBoost*. Each model outputs a prediction indicating whether the transaction is *Legitimate* or *Fraudulent*.

- **Ensemble decision:** A majority voting mechanism aggregates the predictions from all models to produce a final decision.
- **Explainability:** For each model, the application provides explanations of the prediction:
  - Tree-based models (Decision Tree, Random Forest, XGBoost) utilize SHAP values to show the contribution of each feature to the prediction.
  - AdaBoost uses a kernel SHAP explainer with a small background sample from the training set.
  - Naive Bayes displays posterior probabilities for each class.
  - K-Nearest Neighbors shows the most similar examples from the training set and their labels, helping the user understand the decision.

Figure 5.2: Screenshot of the application displaying explanations for each classifier. For tree-based models (Decision Tree, Random Forest, XGBoost) SHAP values highlight feature contributions, AdaBoost uses a Kernel SHAP explainer, Naive Bayes shows posterior probabilities, and K-Nearest Neighbors presents the most similar examples from the training set.

This application demonstrates how the developed models can be employed in a real-world context to detect and explain potentially fraudulent transactions, providing both automated classification and interpretable insights to support decision-making.

6. Conclusions

This project aimed to develop and compare a set of classifiers for credit card fraud detection, with a special emphasis not only on performance but also on the interpretability of the models. After an initial preprocessing phase (which included temporal feature engineering, handling of missing values, feature selection, and balancing the training set using *SMOTE*) it was possible to train six different classification models: *K-Nearest Neighbors*, *Gaussian NaiveBayes*, *DecisionTree*, *RandomForest*, *AdaBoost* and *XGBoost*.

- The analysis of the experimental results highlighted the **superiority of ensemble based models**, specifically **XGBoost** and **Random Forest**, with an advantage of the first one. These latter models achieved the best performance across most of the evaluation metrics considered, such as *precision*, *recall*, *f1-score*, and *ROC AUC*, proving to be more effective in identifying fraudulent transactions while minimizing false positives.

- The *t-test* results pointed out that the only couple of models on which we cannot see a statistically significant difference is KNN-RandomForest, only about *f1-score*.

- The comparison between classifiers highlighted, in all metrics, the **uselessness** of the *weighted* versions of the metrics and of the *accuracy* for credit card fraud detection: the vast majority of datasets containing fraudulent and non-fraudulent transactions are strongly imbalanced, with much more non-fraudulent transactions respect to fraudulent ones (in order to represent real world scenarios, in which frauds are a tiny part of the total number of transactions). The aforementioned metrics tend to flatten towards the values of the majority class.

- The introduction of the *Acceptable Level of Performance* (*ALP*) concept, defined as the classifier's ability to correctly identify at least 80% of frauds, allowed for deeper analysis of the trade-off between *TPR* and *FPR*. This analysis revealed that models like **XGBoost** and **Random Forest** can reach this performance level with a **limited number of false alarms** by setting the correct threshold.

- The thresholds analysis showed that **KNN** and **DecisionTree could not reach the peak performances** as all the other classifiers on our dataset. KNN seemed a pretty good model in some metrics, but just could not keep the pace of the best ones when it came to *TPR-FPR* trade-off. Moreover, it is also the slowest model to test by far, therefore we can conclude that it is not a good choice for a real-world application that should give real-time predictions.

- AdaBoost was the most "conservative" and "safe" model in terms of *ALP_threshold*: it reached *ALP* with the smallest deviation of the threshold from 0.5, but this came along with the biggest *ALP_FPR*. These reason, along with the fact that it turned out to be pretty bad in the majority of metrics, makes it a poor choice, despite being an ensemble model.

- Overall, **XGBoost** turned out to be the **best choice** for a credit card fraud detection application, as already documented in related works [1] and proved by *t-test* results.

- Overall, **Gaussian NaiveBayes** turned out to be the **worst choice** for a credit card fraud detection application, as proved by *t-test* results.

- A key aspect of the project was the study of model interpretability (*XAI*). The use of techniques such as intrinsic *feature importance analysis*, *SHAP values*, and *permutation feature importance* helped to find the factors that most influence predictions, increasing the **transparency** and **trustworthiness** of the classifiers. Furthermore, *rule extraction* via surrogate models offered a simplified and understandable view of the behavior of the more complex models.

- Finally, the project culminated in the development of an **interactive Streamlit application** that simulates a real-world use case. This prototype not only classifies transactions in real time but also provides detailed explanations for each model's prediction, offering a practical tool for analysis and decision support. The application integrates the six classifiers and a majority voting mechanism for the final decision, demonstrating how different machine learning techniques can be combined to create a robust, high-performing, and interpretable fraud detection system.

Bibliography

[1] Cho Do Xuan, Dang Ngoc Phong, and Nguyen Duy Phuong. A new approach for detecting credit card fraud transaction. *International Journal of Nonlinear Analysis and Applications*, 14:133–146, 2023. URL: https://ijnaa.semnan.ac.ir/article_7623_b95b41b8707a1ba645b2ad938f3cd76f.pdf.

[2] IEEE Computational Intelligence Society. Ieee-cis fraud detection. Kaggle, 2025. Accessed: 2025-07-20. URL: https://www.kaggle.com/c/ieee-fraud-detection.

[3] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*, pages 785–794. ACM, 2016. doi:10.1145/2939672.2939785.