Universidade de São Paulo
Instituto de Física de São Carlos

SCC00277-201-2021 Project 2

Éverton Luís Mendes da Silva (10728171)

Contents

1 Introduction

Undoubtedly, with the advent of technology, the use of cards (credit or debit) has increased over the years. Thus, both banks and customers are concerned about the security that this type of operation can provide. With this in mind, this project aims to build models for the prediction of fraud in transactions provided by the kaggle database, IEEE- CIS Fraud Detection. Therefore, in order to predict frauds, 7 types of ML (machine learning) models were trained using data modeling techniques (Pipelines for Preprocessing) and hyperparameter optimization (Bayesian optimization).

2 Question One

2.1 Item a

For the first part of this project, the AUC (area under curve) metric was analyzed for three types of models, each model are listed below:

- Model that randomly ranks (50% chance of saying it is fraud and 50% of saying it's not) (a)
- Model that classifies all cases as fraud (b)
- Model that classifies all cases as non-fraud. (c)

Furthermore, we can find below the AUC measurements for each of the models listed.

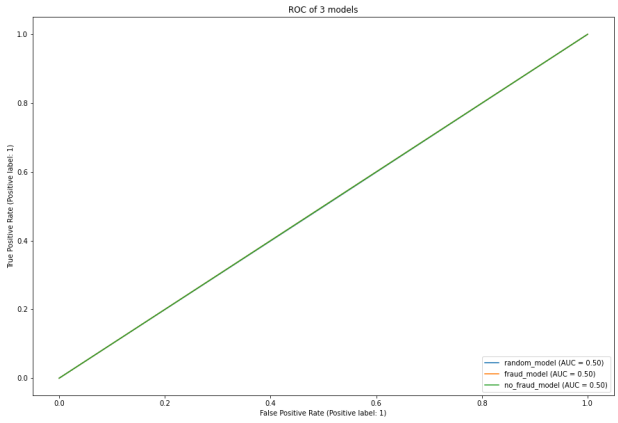


Figure 1: AUC of 3 models

2.2 Item b

To evaluate the graph above, it is necessary to understand a little more about this method of measuring performance in binary classifications, that is, to understand how Area Under the Receiver Operating Characteristics has the ability to distinguish classes. For example, the higher the AUC value, the better the model is at predicting classes 0 as 0 and 1 as 1 (sick patients as sick and healthy as healthy).

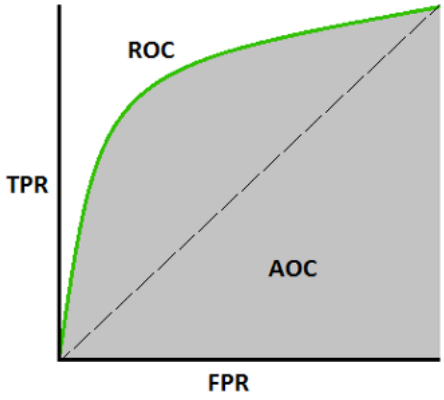


Figure 2: AUC-ROC(Roc Curve)

With this in mind, below we have possible scenarios for the distribution of probabilities for each class.

2.3 Ideal scenario

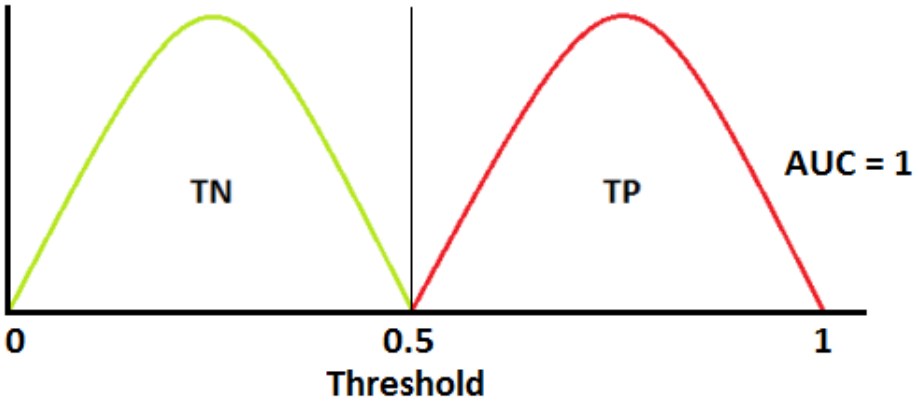


Figure 3: Probabilities

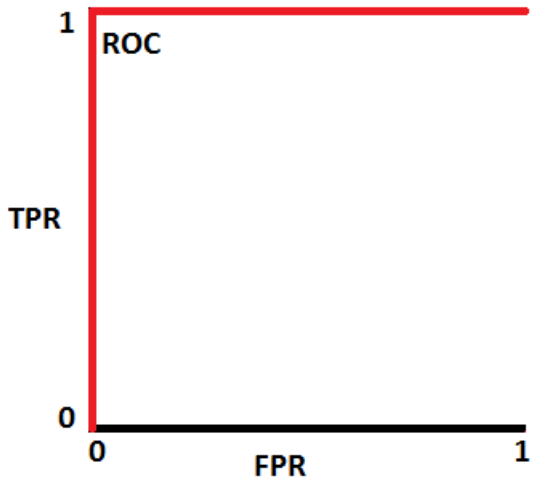


Figure 4: Roc Curve

2.4 Scenario with Errors

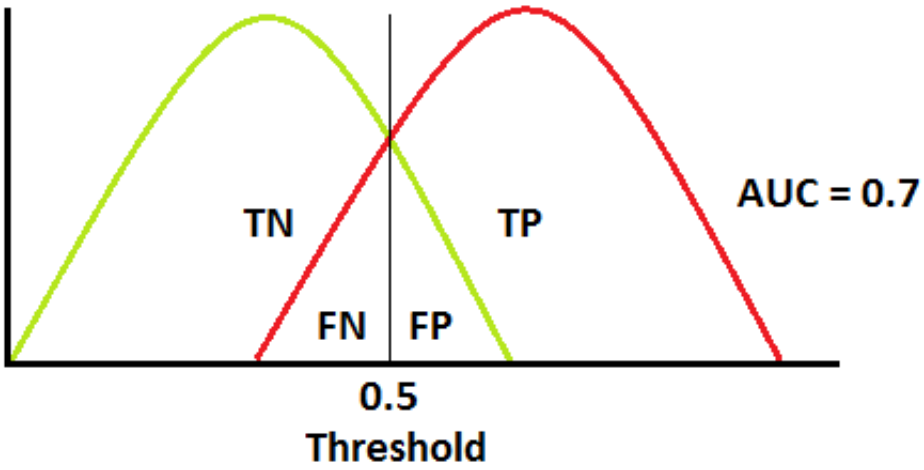


Figure 5: Probabilities

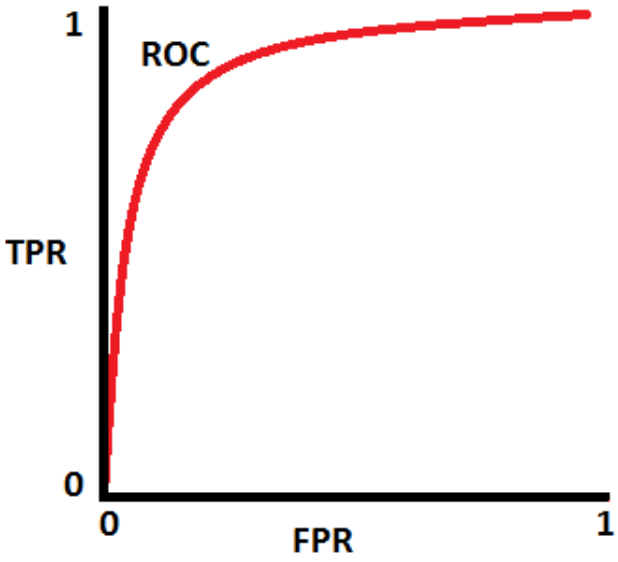


Figure 6: Roc Curve

2.5 Random scenario

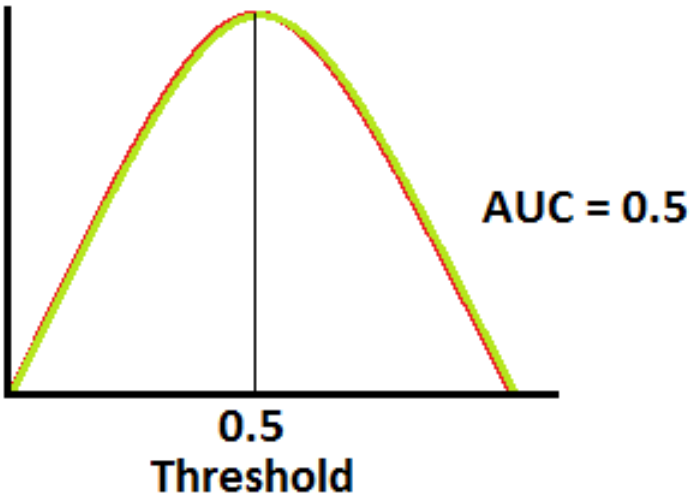


Figure 7: Probabilities

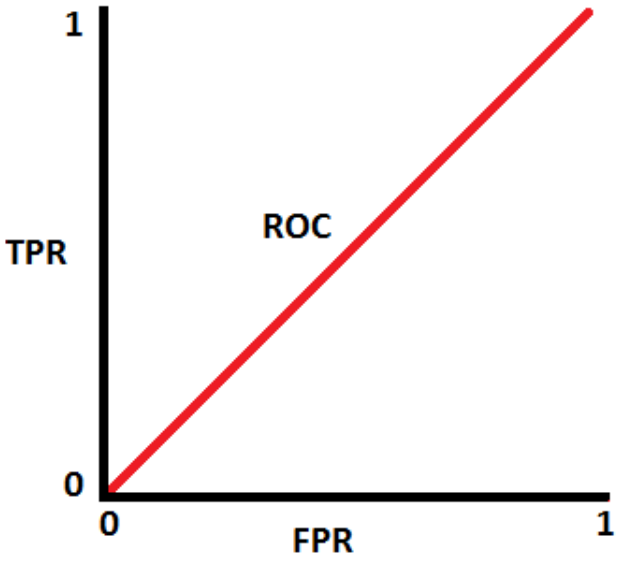


Figure 8: Roc Curve

2.6 Inverse scenario

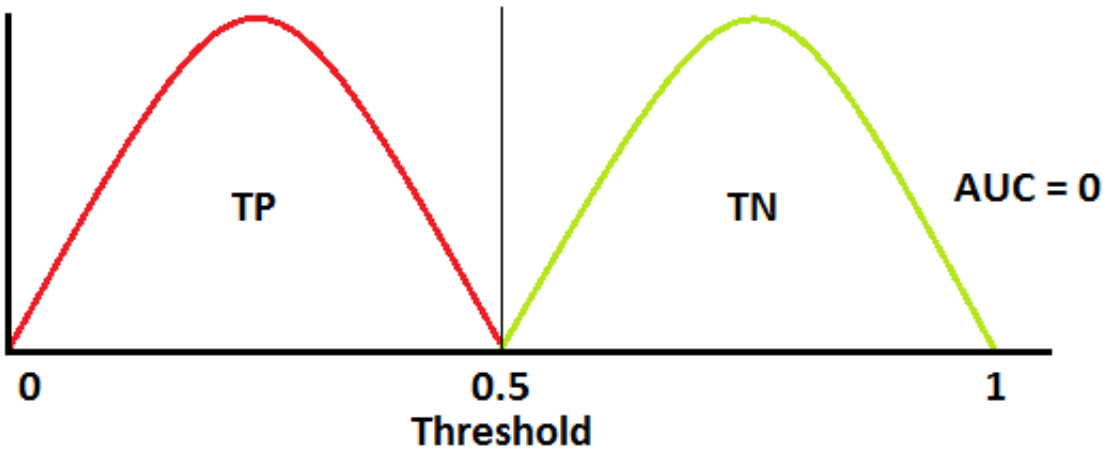


Figure 9: Probabilities

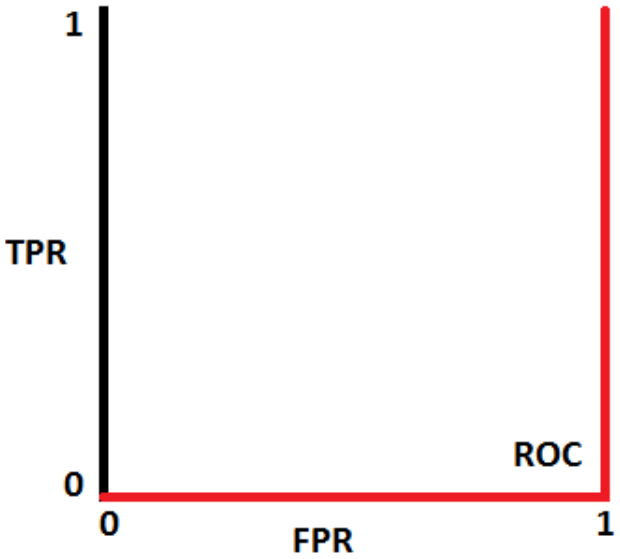


Figure 10: Roc Curve

In short, after presenting the possible scenarios, it is evident that the three types of models are random. That is, they predict the classes with the greatest possible chance of error.

3 Questão 2

3.1 Item a and b

In this second part of the project, new explanatory variables were created for the training of models. The basic idea for these new variables is the reliability of the domains, type of device or browser used. That is, eight new variants were created that assume the value between 0 and 1 according to the unique values found. For example, the column "P_emaildomain" has several types of domains, and each one of them has a certain number of people who use it. Thus, the new reliability column takes into account the number of people using that domain with the total number of transactions performed, that is, if 800 people use the "yahoo" domain in a total of 1000 transactions, this domain acquires a reliability of 0.8 .

$$\begin{cases} \dfrac{N\_Domains\_x_1}{Total\ of\ Transactions} & if N\_Domains\_x_1 > Reliability\ Limit * Total\ of\ Transactions \\ 0 & if\ N\_Domains\_x_1 < Reliability\ Limit * Total\ of\ Transactions \end{cases}$$

In our case, we consider that there is a reliability limit (between 0 and 1), where domains with a very low number of people are obligatorily zero.

With this in mind, below we have a list of the variables created by this method along with the corresponding code.

- P_emaildomain, id_30, id_31, id_33, id_34, DeviceInfo, card6, R_emaildomain

```
import numpy as np
import pandas as pd


def domain_reliability(df, columns, p_trustfull_above=0.01):
    '''creates a column with the domain reliability of each sample
    Args:
      df, DataFrame
      columns, list of columns to analyse realiability
      p_trustfull_above, percentage of trusted domains
    Return:
      domain_values, dict with the numbers of samples in each domain
    '''
    columns = [column for column in columns if column in df.columns]

    for column in columns:
        domain_values = df[column].value_counts()
        n_all_domains = domain_values.sum()
        trustfull_domains = domain_values[domain_values > p_trustfull_above*n_all_domains].to_dict()

        df[str(column)+'_reliability'] = [trustfull_domains[domain]/n_all_domains if domain in trustfull_domains else 0 for domain in df[column]]

    df.drop(columns=columns, inplace=True)
```

```
26
27      return domain_values
```

Listing 1: Part of data_treatment.py

4 Question Three

For this study, 7 types of ML models were used, below we can find which they are and their respective hyperparameter search spaces, which took into account the AUC(ROC) metric. This search was performed by the Bayesian optimization found in the optimize(BayesSearchCV) scikit.

- QuadraticDiscriminantAnalysis
- BernoulliNB
- PassiveAggressiveClassifier
- RidgeClassifier
- Perceptron
- SGDClassifier
- MLPClassifier

```
1  from sklearn.naive_bayes import BernoulliNB
2  from sklearn.linear_model import RidgeClassifier
3  from sklearn.linear_model import Perceptron
4  from sklearn.linear_model import SGDClassifier
5  from sklearn.discriminant_analysis import
       QuadraticDiscriminantAnalysis
6  from sklearn.linear_model import PassiveAggressiveClassifier
7  from sklearn.neural_network import MLPClassifier
8
9  from skopt.space import Real, Categorical, Integer
10
11
12 mlp_clf = {
13      'model': Categorical([MLPClassifier()]),
14      'model__activation': Categorical(['identity', 'logistic',
        'tanh', 'relu']),
15      'model__solver': Categorical(['sgd', 'adam']),
16      'model__alpha': Real(0.0000001, 0.001,  'uniform'),
17      'model__learning_rate': Categorical(['constant', '
       invscaling', 'adaptive']),
```

```
    'model__learning_rate_init': Real(0.0000001, 0.001,  'uniform'),
    'model__power_t': Real(0.0005, 0.5, 'uniform'),
    'model__max_iter': Integer(100,5000, 'uniform'),
    'model__momentum': Real(0.1, 0.99, 'uniform'),
    'model__beta_1': Real(0.1, 0.99, 'uniform'),
    'model__beta_2': Real(0.1, 0.99, 'uniform'),
    'model__epsilon': Real(0.0000001, 0.001,  'uniform'),
}


bernoulli_clf={
    'model': Categorical([BernoulliNB()]),
    'model__alpha':Real(0.01, 0.99, 'uniform')
}

QDA_clf={
    'model':Categorical([QuadraticDiscriminantAnalysis()]),
    'model__tol':Real(0.0000001, 0.01,  'uniform')
}

PassiveAggressive_clf={
    'model': Categorical([PassiveAggressiveClassifier()]),
    'model__max_iter':Integer(1000, 10000, 'uniform'),
    'model__tol':Real(0.0000001, 0.01,  'uniform'),
    'model__C': Real(0.01, 0.99, 'uniform'),
    'model__loss': Categorical(['hinge', 'squared_hinge'])
}


ridge_clf_positive = {
    'model': Categorical([RidgeClassifier(positive=True)]),
    'model__tol': Real(0.0000001, 0.001,  'uniform'),
}

ridge_clf_false = {
    'model': Categorical([RidgeClassifier(positive=False)]),
    'model__solver': Categorical(['svd', 'cholesky', 'sparse_cg', 'sag', 'saga']),
    'model__tol': Real(0.0000001, 0.001,  'uniform')
}

perceptron_clf = {
    'model': Categorical([Perceptron(fit_intercept=False)]),
    'model__penalty': Categorical(['l2', 'l1', 'elasticnet']),
    'model__alpha': Real(0.00000001, 0.001, 'uniform'),
    'model__l1_ratio': Real(0.01, 0.99, 'uniform'),
    'model__max_iter': Integer(1000, 10000, 'uniform'),
```

```
    'model__tol': Real(0.0000001, 0.001,  'uniform'),
}


sgd_clf = {
    'model': Categorical([SGDClassifier(fit_intercept=False)]),
    'model__loss': Categorical(['hinge', 'log', 'modified_huber', 'squared_hinge', 'perceptron', 'squared_error', 'huber', 'epsilon_insensitive', 'squared_epsilon_insensitive']),
    'model__alpha': Real(0.00000001, 0.001, 'uniform'),
    'model__max_iter': Integer(1000, 10000, 'uniform'),
    'model__epsilon': Real(0.0000001, 0.001,  'uniform'),
    'model__power_t': Real(0.01, 0.99, 'uniform'),
    'model__eta0': Real(0.01, 0.99, 'uniform'),
    'model__warm_start': Categorical([True, False]),
    'model__tol': Real(0.0000001, 0.001, 'uniform'),
    'model__penalty': Categorical(['l2', 'l1', 'elasticnet']),
    'model__l1_ratio': Real(0.01, 0.99, 'uniform'),
    'model__learning_rate': Categorical(['constant', 'optimal', 'invscaling', 'adaptive']),
}
```

Listing 2: hyperparameters for each model

4.1 Feature Engineering

Before training the models, it was necessary to perform data engineering to create new variables and perform normalizations. First, we removed columns with more than 90% the Nan values and columns that had a number of unique values close to the number of samples. So below, I have the code that performs these parts.

```
def drop_Nan_columns(df, drop_limit=0.7):
    ''' drop columns with my Nan values
    Args:
      df, DataFrame
      drop_limit, percentage limit of Nan values in a column
    Return:
      Nan_dropped, list of features dropped
    '''
    Nan_dropped = [feature for feature in df.columns if df[feature]
                   [df[feature].isna() == True].shape[0] > drop_limit*df.shape[0]]
    df.drop(columns=Nan_dropped, inplace=True)
```

```
12
13      return Nan_dropped
14
15
16  def nunique_upperBound_columns(df, p_upper_bound=0.85,
    drop_nunique=False):
17      '''get columns with unique values above upper bound
18      Args:
19        df, DataFrame
20        upper_bound, percentage of samples above upper bound
21        drop_nunique, drop the nunique_features above upper
    bound if True
22      Return:
23        nunique_features, dict of samples of above upper bound
24      '''
25      nunique_features = {feature: df[feature].nunique() for
    feature in df.columns if df[feature].nunique() >
    p_upper_bound*df.shape[0]}
26      if drop_nunique:
27          df.drop(columns=nunique_features.keys(), inplace=True
    )
28
29      return nunique_features
```

Listing 3: feature engineering

4.2 Undersample and Bootstrap

Also, looking at the dataset, you can see that the classes are unbalanced (about 500,000 non-cheats to approximately 20,000 cheats). Undeniably, this imbalance creates an absurd bias that was taken away with an undersampling of the data, with a bootstrap afterwards to have a greater generality of the reduced class.

```
1  import numpy as np
2  import pandas as pd
3
4
5  def undersample_boostrap(inputs: pd.DataFrame, targets: pd.
    DataFrame, bootstrap_size=1000):
6      '''undersample a Daframe with with features(inputs) and
    targets
7      Args:
8        inputs, DataFrame with the features
9        targets, Datframe with the targets
10     Return:
11       undersampled_data,
12       undersampled_targets,
```

```
    '''
    min_sample = min(targets.value_counts().tolist())

    undersampled_data = pd.DataFrame(columns=inputs.columns)
    undersampled_targets = pd.DataFrame()

    # undersample
    for class_type in targets.value_counts().index:

        indices_class = np.where(targets == class_type)[0]

        indices_atrr_sample = np.random.choice(
            a=indices_class, size=min_sample, replace=False)
        undersampled_data = undersampled_data.append(
            inputs.iloc[indices_atrr_sample])
        undersampled_targets = undersampled_targets.append(
            targets.iloc[indices_atrr_sample].tolist())

    # bootstrap
    for class_type in targets.value_counts().index:

        indices_class = np.where(targets == class_type)[0]

        indices_atrr_sample = np.random.choice(
            a=indices_class, size=bootstrap_size, replace=
    True)
        undersampled_data = undersampled_data.append(
            inputs.iloc[indices_atrr_sample])
        undersampled_targets = undersampled_targets.append(
            targets.iloc[indices_atrr_sample].tolist())

    return undersampled_data, undersampled_targets
```

Listing 4: undersample and boostrap

4.3 Preprocessing

Finally, these data went to the processing part in order to replace the Nan values and modify categorical variables with OneHotEncoder. These procedures were done using numeric and categorical pipelines, as shown in the code below.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder,
    OrdinalEncoder
```

```
6
7
8  def preproc_normalize(X_train=None, X_test=None, y_train=None
       , y_test=None, scaler=None, scaler_trigger=False,
       X_test_trigger=True):
9      '''normilize the data with MinMaxScaler
10     Args:
11       X_train, X_test, y_train, y_test
12     Return:
13       X_train, X_test, y_train, y_test
14     '''
15
16     if scaler_trigger == False:
17         scaler = StandardScaler()
18         scaler.fit(X_train)
19
20     X_train = scaler.transform(X_train)
21     if X_test_trigger == True:
22         X_test = scaler.transform(X_test)
23
24     return X_train, X_test, y_train, y_test, scaler
25
26
27 def preprocess(X_train=None, X_test=None, y_train=None,
       y_test=None, categorical_features=None, numerical_features
       =None, normalize_fn=None, scaler_fn=None, scaler_trigger=
       False, X_test_trigger=True):
28     '''replace Nan values of categorical and numerical
       features. Moreover, transform categorical features in
       numerical data
29     Args:
30       X_train, X_test, y_train, y_test
31       categorical_features, list with the name of the
       categorical columns
32       numerical_features, list with the name of the numerical
        columns
33     Return:
34       X_train, X_test, y_train, y_test
35     '''
36
37     numerical_pipeline = Pipeline(steps=[
38         ('imputer', SimpleImputer(strategy='mean'))])
39
40     categorical_pipeline = Pipeline(steps=[
41         ('imputer', SimpleImputer(strategy='most_frequent')),
42         ('onehot', OneHotEncoder())])
43
44     transformation = ColumnTransformer(
45         transformers=[
```

```
            ('numerical transformation', numerical_pipeline,
numerical_features),
            ('categorical transformation',
             categorical_pipeline, categorical_features),
        ])

    X_train = transformation.fit_transform(X_train)
    if X_test_trigger == True:
        X_test = transformation.transform(X_test)

    if scaler_trigger == False and X_test_trigger == False:
        X_train, X_test, y_train, y_test, scaler =
normalize_fn(
            X_train=X_train, y_train=y_train, X_test_trigger=
False)

    elif scaler_trigger == True and X_test_trigger == False:
        X_train, X_test, y_train, y_test, scaler =
normalize_fn(
            X_train=X_train, y_train=y_train, X_test_trigger=
False, scaler=scaler_fn, scaler_trigger=True)

    elif scaler_trigger == False and X_test_trigger == True:
        X_train, X_test, y_train, y_test, scaler =
normalize_fn(
            X_train, X_test, y_train, y_test)
    else:
        X_train, X_test, y_train, y_test, scaler =
normalize_fn(
            X_train=X_train, X_test=X_test, y_train=y_train,
y_test=y_test, scaler=scaler_fn, X_test_trigger=False,
scaler_trigger=True)

    return X_train, X_test, y_train, y_test, scaler
```

Listing 5: replace Nan

5 Question Four

The purpose of this section is to discuss the results obtained by the Bayesian optimization of several models and hyperparameters, in order to choose the one with the best Score for the AUC metric. With this in mind, in the image below we can see a convergence plot for the models trained in each period of study.

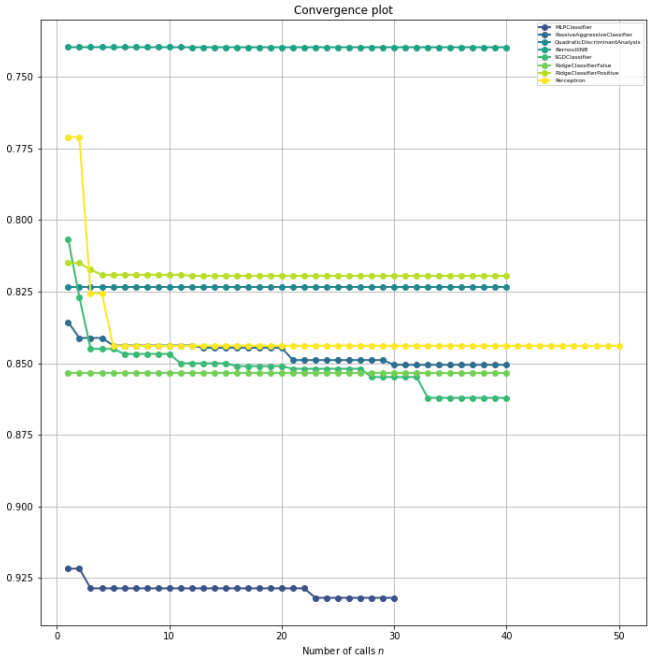


Figure 11: Convergence of models

5.1 Hyperparameter dependency

At first, we can see the model MLPClassifeier has the best score for the metric. However, to have a deeper analysis we need to see how each of the hyperparameters of each model depend on each other. As a result, below we have the dependency graphs for each of the optimized models.

5.1.1 Bernoulli Naive Bayes

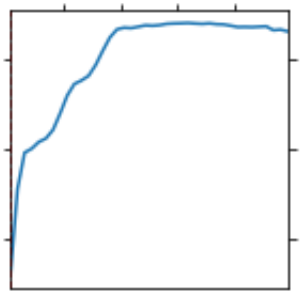


Figure 12: Dependence Plot

5.1.2 Passive Agressive Classifier

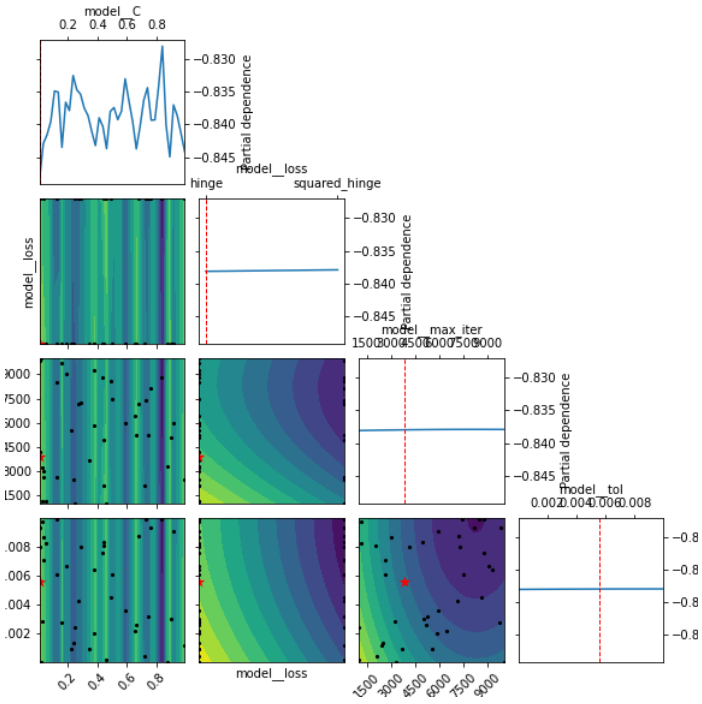


Figure 13: Dependence Plot

5.1.3 Ridge Classifier Negative

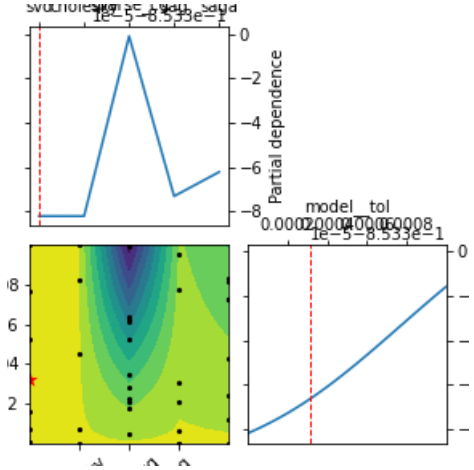


Figure 14: Dependence Plot

5.1.4 Ridge Classifier Positive

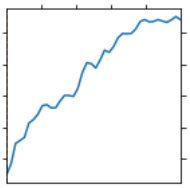


Figure 15: Dependence Plot

5.1.5 Perceptron

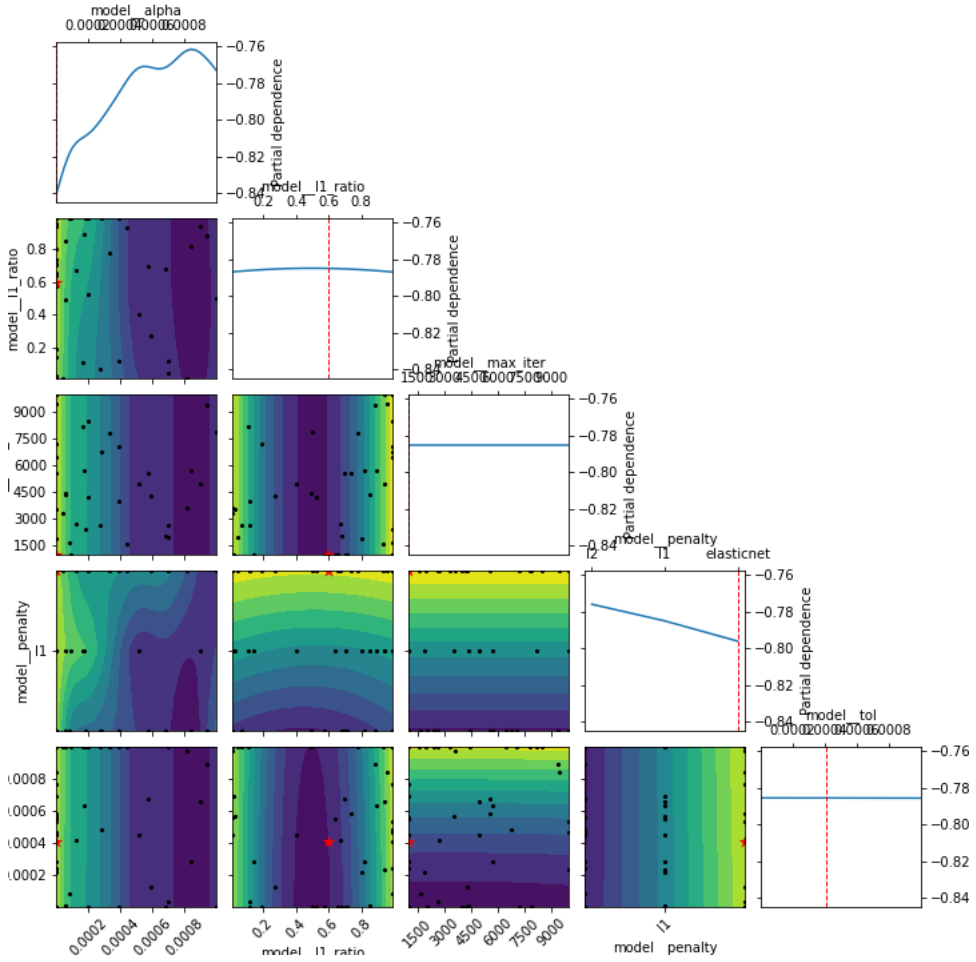


Figure 16: Dependence Plot

5.1.6 Stochastic Gradient Descent Classifier

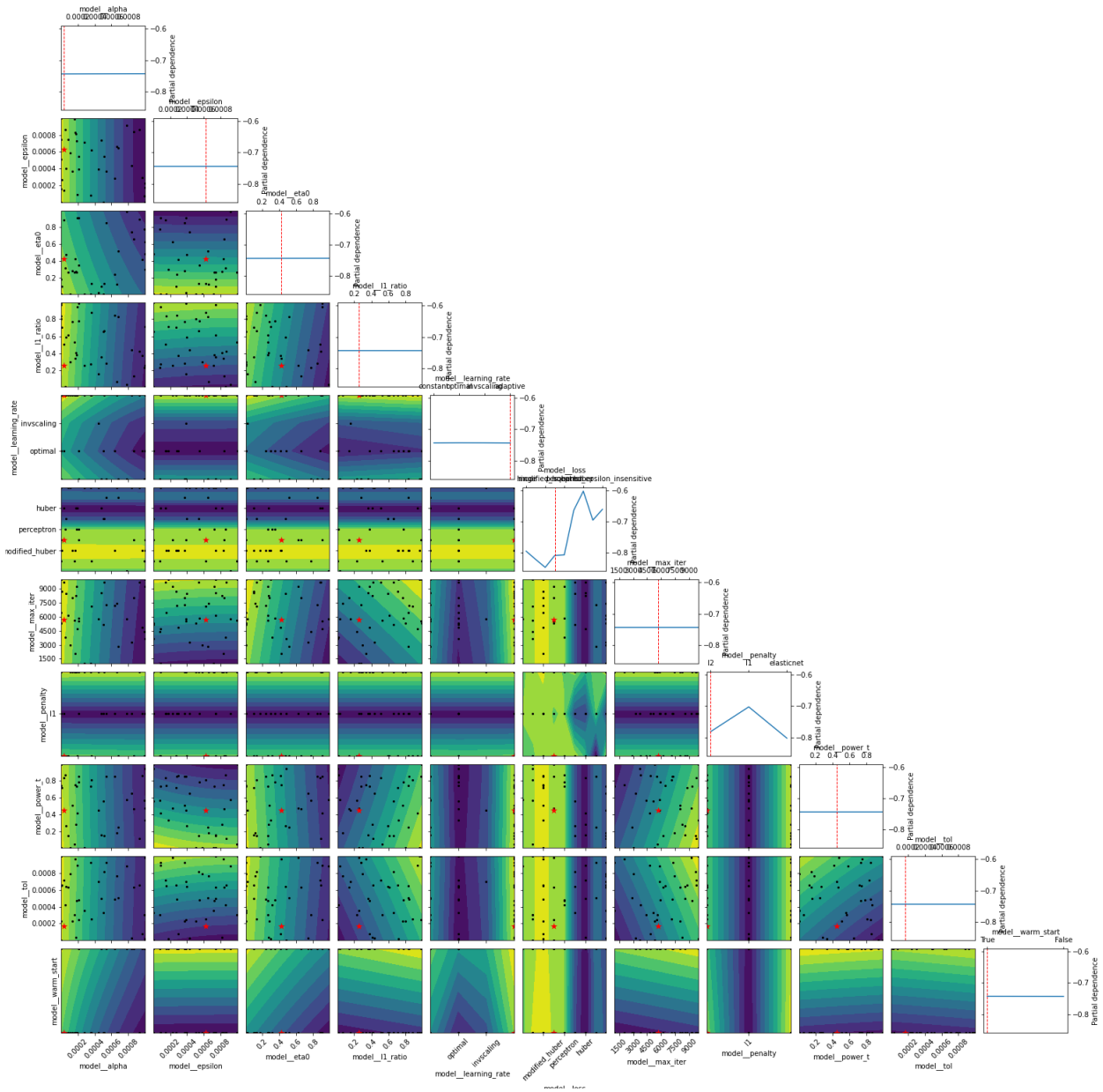


Figure 17: Dependence Plot

5.1.7 Multi-Layer Perceptron Classifier

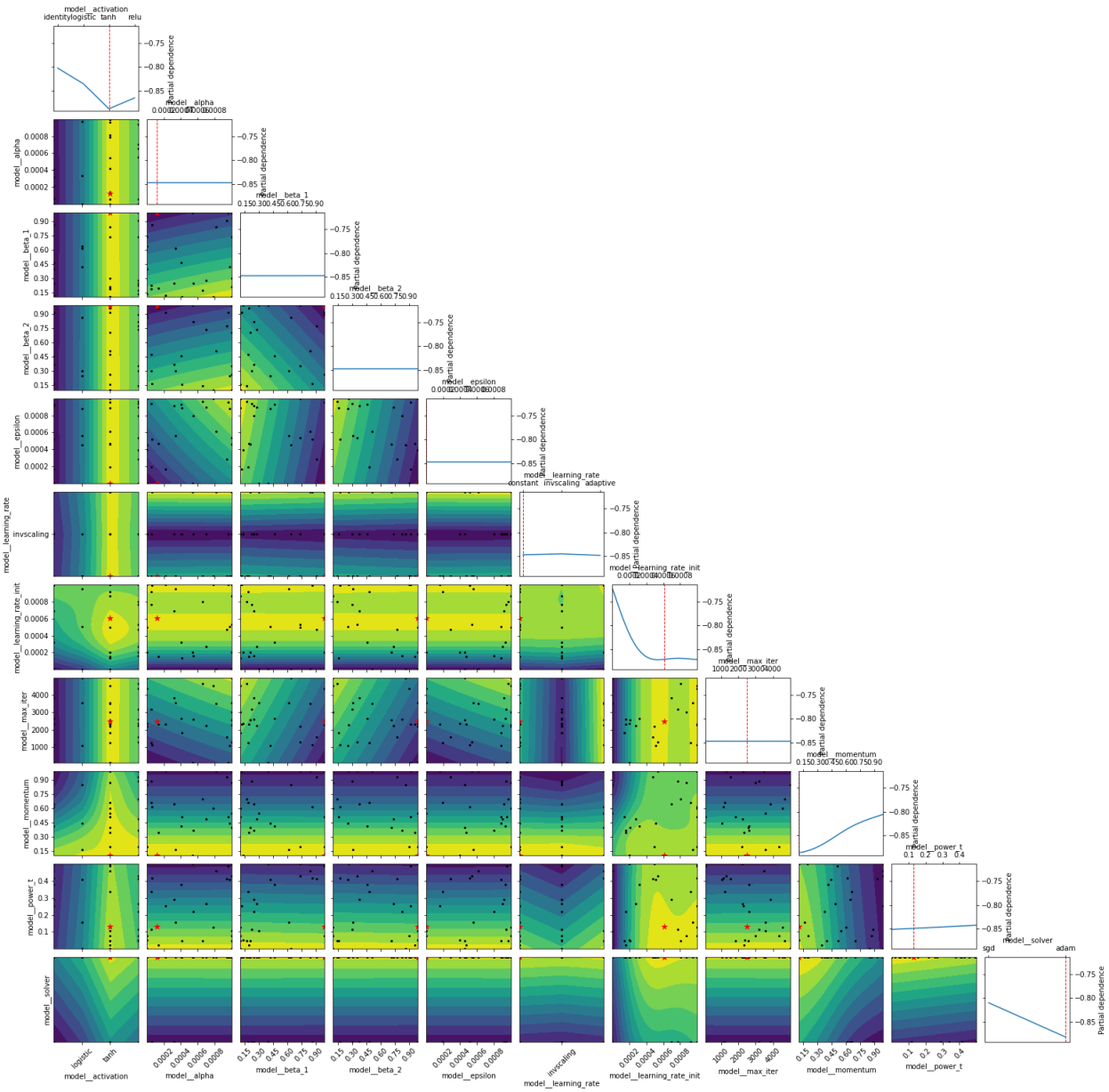


Figure 18: Dependence Plot

5.1.8 Quadratic Discriminant Analysis

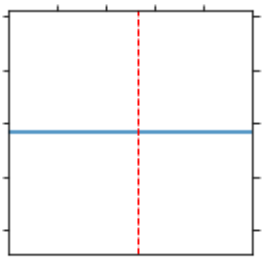


Figure 19: Dependence Plot

5.2 Best Model

In short, the best model of this project achieved a good ROC curve for the parameters of TPR, FPR, TNR, FNR. To reiterate this inference, we have below the images generated for the ROC curve and confusion matrix of the best model, respectively.

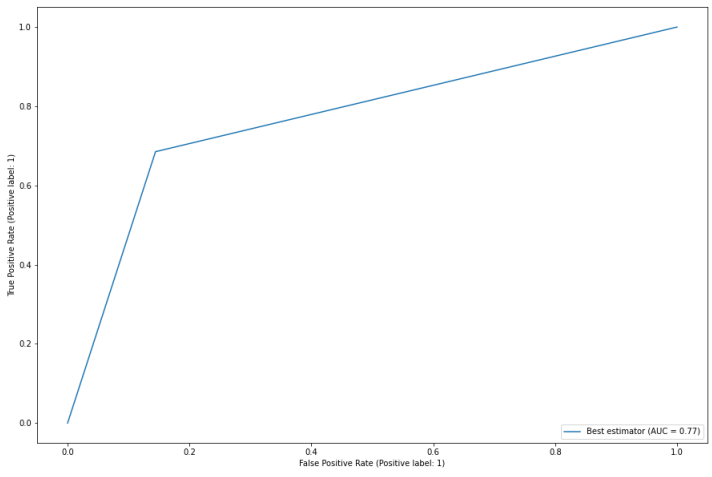


Figure 20: Roc curve

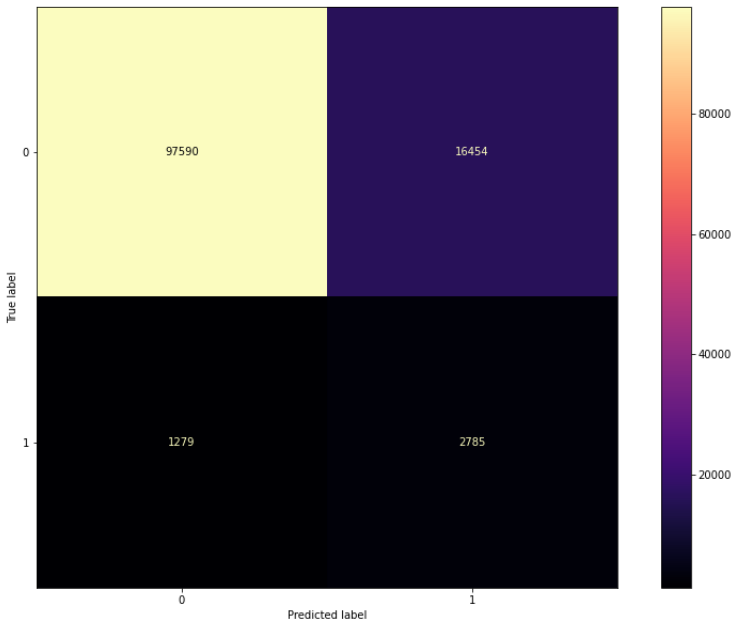


Figure 21: Confusion Matrix

5.3 My Kaggle Submission

Figure 22: kaggle submission

6 Question Five

In summary, having the best model, we can obtain the probabilities of each transaction being a fraud. In sklearn, we have the predict_proba function that gives us the necessary information to use with decision trees and binomial distributions to find a cutoff point from which transactions will be

considered as fraud and barred/blocked. Taking into account the false positives that interfere with the customer's convenience, making him reflect on going to a competitor in the market.

In this way, we can build a decision tree with the confusion matrix of the best model, obtained in the previous section, and the probability of each transaction being a fraud (through the histogram of the training file, "train_transaction.csv").

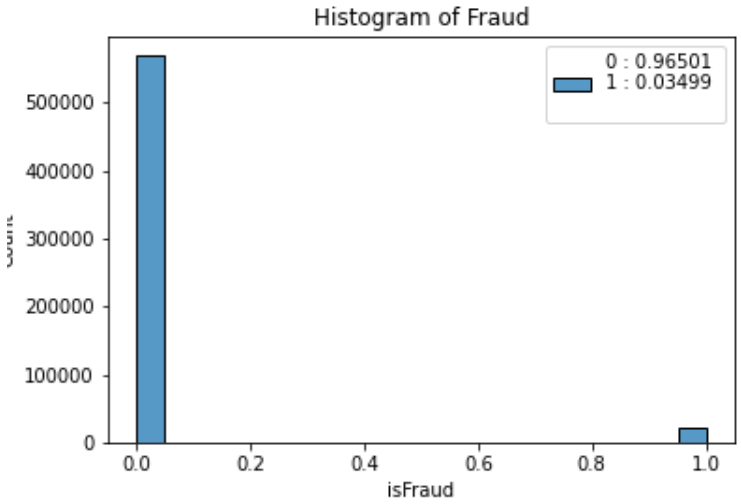
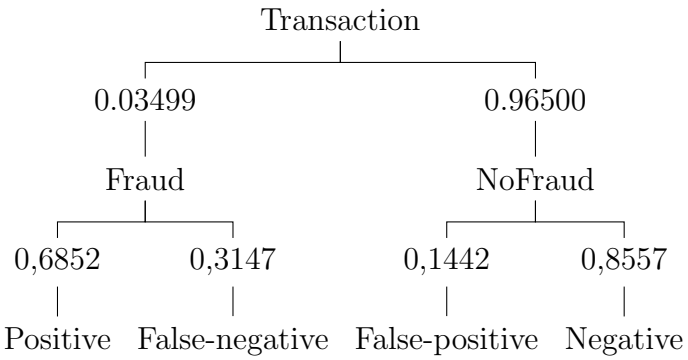


Figure 22: Histogram of "train_transaction.csv"



$$\text{Fraud} \rightarrow \begin{cases} Positive = 0.02397 \\ False - negative = 0.01101 \end{cases}$$

$$\text{NoFraud} \rightarrow \begin{cases} False - Positive = 0.13915 \\ Negative = 0.82575 \end{cases}$$

Taking the above probability paths into consideration, it is possible to find the probability of correctly hitting the cheat when I predict a cheat, and the probability of missing a cheat when I predict NoFraud.

$$Right\_Frauds = \frac{0.02397}{0.13915 + 0.02397} = 14,6947\% \tag{1}$$

$$Missing\_Frauds = \frac{0.01101}{0.01101 + 0.82575} = 1,3157\% \tag{2}$$

With this in mind, let's consider binomial distributions and the concept of value at risk to obtain a cutoff point for the customer's card. The distributions assume a 30-day card history and the respective probability found above, Right_Frauds and Missing_Frauds.

$$P(n, k, p) = \frac{n!}{k!(n-k)!}\ p^k(1-p)^{n-k} \tag{3}$$

To show how the cutting bridge will be calculated, we will present some variables of the problem.

- number of transactions = n_t
- probability of Right_Frauds = p_r = 14, 6947%
- probability of Missing_Frauds = p_m = 1,3157%
- number of Frauds = $k_r[\ ]_i$
- number of Missing Frauds = $k_m[\ ]_i$
- value of each transaction = $t_v[\ ]_i$
- predict_proba Fraud = $pb_r[\ ]_i$
- prdict_proba No Fraud = $pb_m[\ ]_i$

First, we need to initialize the Fraud and Non-Fraud vectors, $k_r[\ ]_i$ and $k_m[\ ]_i$ respectively. The initialization of these vectors depends solely on the output of the predict_proba function. As an example, let's consider only the $k_r[\ ]_i$ algorithm, however it is also valid for the $k_m[\ ]_i$ considering the necessary changes. The initialization follows the next step, we have a K_{aux} that starts with zero and at each interaction of predict_proba the helper's value is incremented if the Fraud probability is greater than not Fraud, inserting the helper's value in position i of the vector $k_r[\ ]_i$.

Finally, we can build the Risk equations, as shown below:

$$R_m = \sum_{i=0}^{n_t} t_{vi} P(n_t, k_{mi}, p_m) pb_{mi} \tag{4}$$

$$R_r = \sum_{i=0}^{n_t} t_{vi} P(n_t, k_{ri}, p_r) pb_{ri} \tag{5}$$

$$if \ \frac{R_m + R_f}{\sum_i t_{vi}} > 5\% \quad denied \ card$$

7 Reference

[1] Links with the images of Graph and Adjancecy List

[2] da Silva, Éverton Luís Mendes. Codes from this project

[3] Images to explain AUC-ROC metric

[4] da Silva, Éverton Luis Mendes. Codes from this project in Drive