Project Report

INTRODUCTION

The objective of this project is to predict whether a client of a Portuguese banking institution will subscribe to a term deposit. The target variable is the column "*y*" in the "*bank-full.csv*" dataset.

The data were collected from the UCI Machine Learning Repository, which is a collection of databases used by the machine learning community for empirical analysis of machine learning algorithms:
https://archive.ics.uci.edu/dataset/222/bank+marketing

The "*bank-full.csv*" dataset is multivariate, meaning there are two or more variables measured on each unit. Each column in the "*bank-full.csv*" dataset represents a variable. Here are the variables:

- **age**: An integer representing the age of the client.
- **job**: A string representing the type of occupation of the client.
- **marital**: A string representing the marital status of the client.
- **education**: A string representing the educational level of the client.
- **default**: A string representing if the client has a credit in default.
- **balance**: An integer representing the average yearly balance of the client's account in euros.
- **housing**: A string representing if the client has a housing loan.
- **loan**: A string representing if the client has a personal loan.
- **contact**: A string representing the type of contact the bank has with the client.
- **day_of_week**: An integer representing the last contact day of the week the bank reached the client.
- **month**: A string representing the last contact month the bank reached the client.
- **duration**: An integer representing the last contact duration from bank to client in seconds.
- **campaign**: An integer representing the number of contacts performed during this campaign for this client.
- **pdays**: An integer representing the number of days passed after the client was last contacted from a previous campaign.
- **previous**: An integer representing the number of contacts performed before this campaign for this client.
- **poutcome**: A string representing the outcome of the previous marketing campaign.
- **y**: A string representing if the client subscribed to a term deposit.

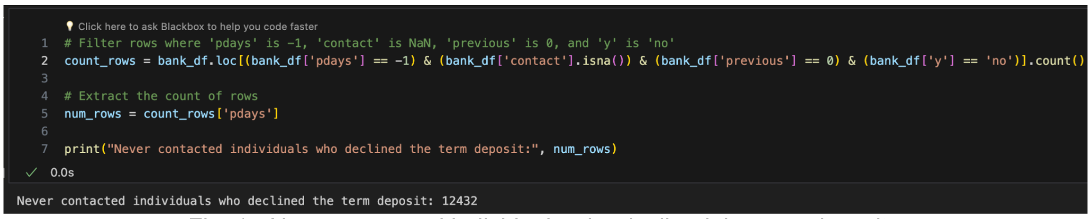
DATA PREPARATION

[PART 1, SECTION B]

As the first steps of data cleaning, I discovered that the categorical columns “*poutcome*”, “*contact*”, “*education*”, and “*job*” contain NaN values. Specifically, there are 36959, 13020, 1857, and 288 missing values in these columns respectively.

Furthermore, upon further investigation into these NaN values, it became evident that 12950 clients have never been contacted (30% of the dataset) but are marked as 'no' in the target variable. This could lead to a bias in the machine learning models, assuming that the clients who were never contacted would almost always respond 'no' to the offer of a term deposit.

Given this situation, I was forced to remove all rows from the DataFrame that had a NaN value in the *'contact'* column. To avoid losing other information, I subsequently replaced the NaN values with 'unknown', so that I could keep track of them during subsequent stages.



```
💡 Click here to ask Blackbox to help you code faster
# Filter rows where 'pdays' is -1, 'contact' is NaN, 'previous' is 0, and 'y' is 'no'
count_rows = bank_df.loc[(bank_df['pdays'] == -1) & (bank_df['contact'].isna()) & (bank_df['previous'] == 0) & (bank_df['y'] == 'no')].count()

# Extract the count of rows
num_rows = count_rows['pdays']

print("Never contacted individuals who declined the term deposit:", num_rows)
✓ 0.0s

Never contacted individuals who declined the term deposit: 12432
```

Fig. 1 - Never contacted individuals who declined the term deposit

Moving on to the quantitative variables, I analyzed their distribution using the Kolmogorov-Smirnov test statistics and noticed that none of them had a Gaussian distribution. Moreover, I could see that almost each of them had thousands of outliers, so I needed a normalization method that takes outliers into account.
Finally, I applied the Robust Scaler, which calculates the median and the interquartile range (IQR) for each feature and then standardizes the features by subtracting the median and dividing by the IQR.

As the final part of this section, I analyzed the skewness of variables. Applying a threshold of 0.95, it was found that only the categorical variable *'default'* had an excessive level and was therefore removed from the DataFrame. I did this to avoid it causing distortions in model predictions or bias in estimated coefficients, leading to inaccurate predictions.

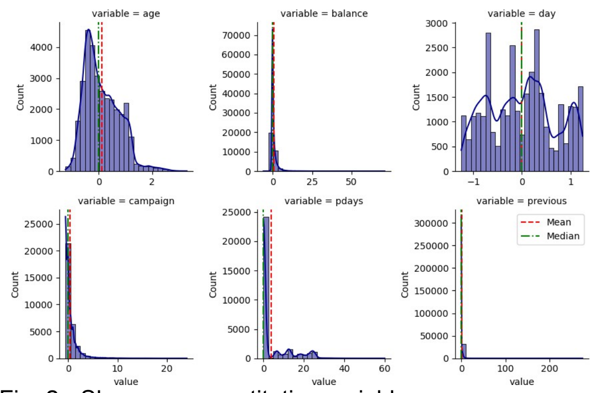


Fig. 2 - Skewness quantitative variables

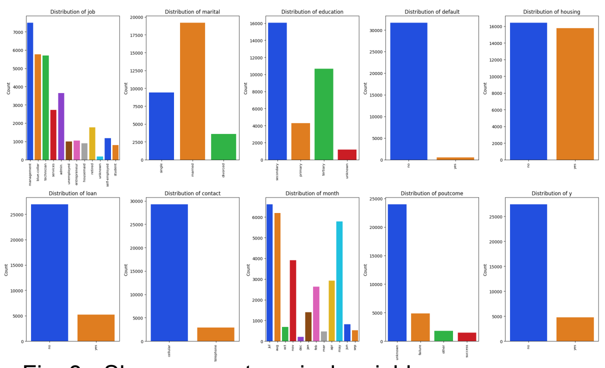


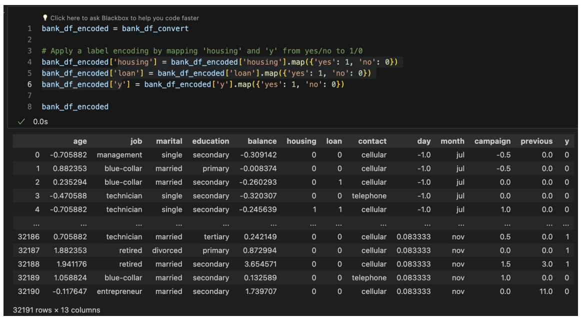
Fig. 3 - Skewness categorical variables

[PART 3, SECTION A]

After determining the unique values of each DataFrame column, it was necessary to modify the categorical variables in such a way that they were easily usable by machine learning models.

This led me to apply both label encoding and one-hot encoding. The former was used with the variables '*housing*', '*loan*', and '*y*', while the latter was with all other categorical variables. For the first categorical variables, I chose label encoding because they had only two features within the variables, namely 'yes' and 'no'. Therefore, I encoded them with 1 and 0. By doing so, I reduced the complexity of the DataFrame, avoiding the creation of new columns. However, I still transformed the Dtype of the variable into numerical, making it easily interpretable by a machine learning model.
For the other categorical variables, I used one-hot encoding because they had more than two features within the variables. This was effective because the categories within the variables do not have an intrinsic order, and the interpretation of the categories was simplified because each category has its separate binary variable encoded with 0 and 1.

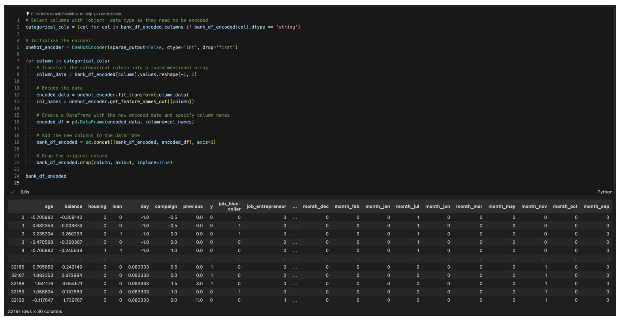
More generally, the label encoding I applied does not suffer from the attribution of artificial order because the categorical variables that enjoy it have the characteristic of being binary. They represent whether a customer has a certain characteristic or has acted in a certain way. Instead, one-hot encoding automatically handles the dummy variable trap. This means that independent variables' multicollinearity is avoided by creating a separate binary variable for each category of the original variable, thus making each category a distinct and independent variable from the others.



```
bank_df_encoded = bank_df_convert

# Apply a label encoding by mapping 'housing' and 'y' from yes/no to 1/0
bank_df_encoded['housing'] = bank_df_encoded['housing'].map({'yes': 1, 'no': 0})
bank_df_encoded['loan'] = bank_df_encoded['loan'].map({'yes': 1, 'no': 0})
bank_df_encoded['y'] = bank_df_encoded['y'].map({'yes': 1, 'no': 0})

bank_df_encoded
```

```
✓ 0.0s

            age          job   marital  education    balance  housing  loan    contact       day  month  campaign  previous  y
    0 -0.705882   management    single  secondary  -0.309142        0     0   cellular      -1.0    jul      -0.5       0.0  0
    1  0.882353   blue-collar  married    primary  -0.008374        0     0   cellular      -1.0    jul      -0.5       0.0  0
    2  0.235294   blue-collar  married  secondary  -0.260293        0     1   cellular      -1.0    jul       0.0       0.0  0
    3 -0.470588   technician    single  secondary  -0.320307        0     0  telephone      -1.0    jul       0.0       0.0  0
    4 -0.705882   technician    single  secondary  -0.245639        1     1   cellular      -1.0    jul       1.0       0.0  0
  ...       ...          ...       ...        ...        ...      ...   ...        ...       ...    ...       ...       ... ...
32186  0.705882   technician   married   tertiary   0.242149        0     0   cellular  0.083333    nov       0.5       0.0  1
32187  1.882353      retired  divorced    primary   0.872994        0     0   cellular  0.083333    nov       0.0       0.0  1
32188  1.941176      retired   married  secondary   3.654571        0     0   cellular  0.083333    nov       1.5       3.0  1
32189  1.058824  blue-collar   married  secondary   0.132589        0     0  telephone  0.083333    nov       1.0       0.0  0
32190 -0.117647 entrepreneur   married  secondary   1.739707        0     0   cellular  0.083333    nov       0.0      11.0  0

32191 rows × 13 columns
```

Fig. 4 - Label encoding



```
# Select columns with 'object' data type as they need to be encoded
categorical_cols = [col for col in bank_df_encoded.columns if bank_df_encoded[col].dtype == 'string']

# Initialize the encoder
onehot_encoder = OneHotEncoder(sparse_output=False, dtype='int', drop='first')

for column in categorical_cols:
    # Transform the categorical column into a two-dimensional array
    column_data = bank_df_encoded[column].values.reshape(-1, 1)
    
    # Encode the data
    encoded_data = onehot_encoder.fit_transform(column_data)
    col_names = onehot_encoder.get_feature_names_out([column])
    
    # Create a DataFrame with the new encoded data and specify column names
    encoded_df = pd.DataFrame(encoded_data, columns=col_names)
    
    # Add the new columns to the DataFrame
    bank_df_encoded = pd.concat([bank_df_encoded, encoded_df], axis=1)
    
    # Drop the original column
    bank_df_encoded.drop(column, axis=1, inplace=True)

bank_df_encoded

```

```
✓ 0.0s

            age    balance  housing  loan       day  campaign  previous  y  job_blue-  job_entrepreneur  ...  month_dec  month_feb  month_jan  month_jul  month_jun  month_mar  month_may  month_nov  month_oct  month_sep
                                                                               collar
    0 -0.705882  -0.309142        0     0      -1.0      -0.5       0.0  0          0                 0  ...          0          0          0          1          0          0          0          0          0          0
    1  0.882353  -0.008374        0     0      -1.0      -0.5       0.0  0          1                 0  ...          0          0          0          1          0          0          0          0          0          0
    2  0.235294  -0.260293        0     1      -1.0       0.0       0.0  0          1                 0  ...          0          0          0          1          0          0          0          0          0          0
    3 -0.470588  -0.320307        0     0      -1.0       0.0       0.0  0          0                 0  ...          0          0          0          1          0          0          0          0          0          0
    4 -0.705882  -0.245639        1     1      -1.0       1.0       0.0  0          0                 0  ...          0          0          0          1          0          0          0          0          0          0
  ...       ...        ...      ...   ...       ...       ...       ... ...       ...               ...  ...        ...        ...        ...        ...        ...        ...        ...        ...        ...        ...
32186  0.705882   0.242149        0     0  0.083333       0.5       0.0  1          0                 0  ...          0          0          0          0          0          0          0          1          0          0
32187  1.882353   0.872994        0     0  0.083333       0.0       0.0  1          0                 0  ...          0          0          0          0          0          0          0          1          0          0
32188  1.941176   3.654571        0     0  0.083333       1.5       3.0  1          0                 0  ...          0          0          0          0          0          0          0          1          0          0
32189  1.058824   0.132589        0     0  0.083333       1.0       0.0  0          1                 0  ...          0          0          0          0          0          0          0          1          0          0
32190 -0.117647   1.739707        0     0  0.083333       0.0      11.0  0          0                 1  ...          0          0          0          0          0          0          0          1          0          0

32191 rows × 36 columns
```

Fig. 5 - One-hot encoding

Since now all my variables are quantitative, I can create a correlation matrix to see the level of correlation among them (Fig. 6). Correlation refers to the shared variance between multiple variables. If this is too high, it increases redundancy and noise in the data and distorts the interpretation of machine learning models. In my case, I chose 0.4 as the correlation threshold because I have a dataset with a skewed target variable, so I need to reduce coefficient instability and, consequently, overfitting in class 0 as much as possible.

The variables with high correlation are:

- *'job_management'* and *'education_tertiary'*
- *'age'* and *'job_retired'*
- *'marital_single'* and *'marital_married'*

- *'marital_single'* and *'age'*
- *'job_management'* and *'education_secondary'*

Given this situation, I removed the variables *'education_tertiary'*, *'job_retired'*, *'marital_single'*, and *'job_management'*.

The result of this process can be seen in the second correlation matrix (Fig. 7).

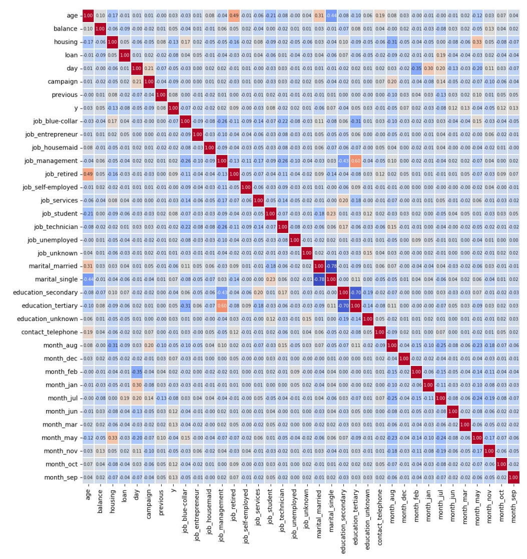


Fig. 6 - First correlation matrix

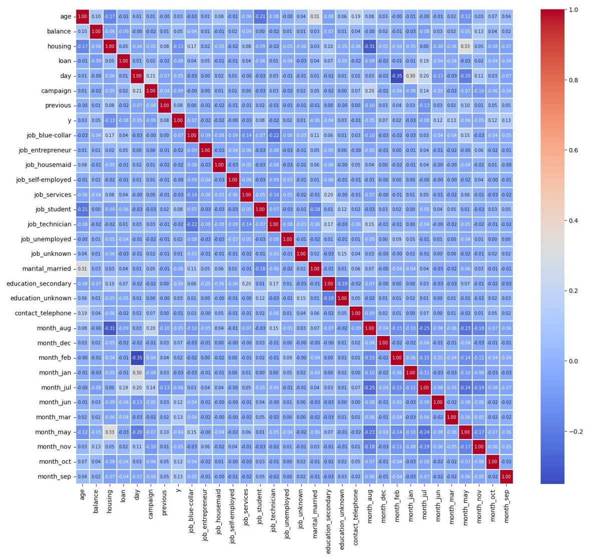
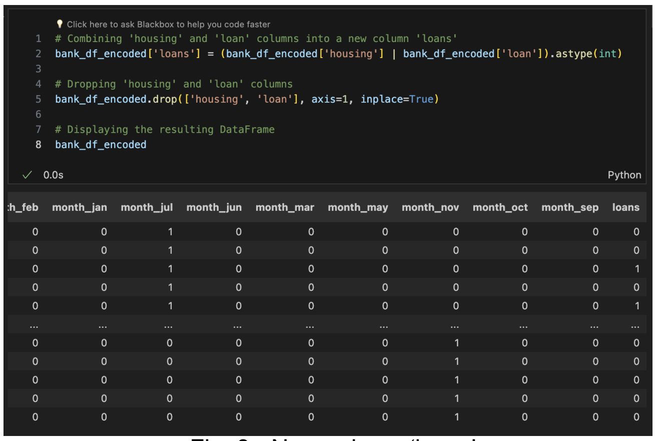


Fig. 7 - Second correlation matrix

As the final step of data preparation, I merged the variables *'housing'* and *'loan'* into the variable *'loans'*. I did this because both generally represent whether customers have any type of debt. In the new *'loans'* column, I applied label encoding according to this logic:
- If there were no loans (0 and 0), the encoding would be 0.
- If there was at least one loan (1 and 0 / 0 and 1 / 1 and 1), the encoding would be 1.

This way, I was able to reduce the dimensionality of the DataFrame and have a consistent representation of customers' debts.



```python
# Combining 'housing' and 'loan' columns into a new column 'loans'
bank_df_encoded['loans'] = (bank_df_encoded['housing'] | bank_df_encoded['loan']).astype(int)

# Dropping 'housing' and 'loan' columns
bank_df_encoded.drop(['housing', 'loan'], axis=1, inplace=True)

# Displaying the resulting DataFrame
bank_df_encoded
```

:h_feb	month_jan	month_jul	month_jun	month_mar	month_may	month_nov	month_oct	month_sep	loans
0	0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	1
...	...	...	...	...	...	...	...	...	...
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0

Fig. 8 - New column ‘loans’

CLASSIFICATION MODELS

[PART 4, SPLIT DATA]

After the final dataset was created, it needed to be split into the target variable (y) and independent variables (X). Then, I further split y and X into train, test, and validation sets. I chose to keep an 80/10/10 split because I wanted to give as many observations as possible to my models in the training part, as I have y skewed towards class 0. Finally, I checked that the proportions of y and X in the different sets were equal, obtaining a positive result.

[PART 4, LOGISTIC REGRESSION]

The first model I applied was a linear regression without any hyperparameters. I did this to see the performance of the model without any external influence, and the result was poor. The accuracy on both the validation and test sets was around 85%, but both the confusion matrix (Fig. 9) and the classification report (Fig. 10) showed significant precision problems in class 1, which then affected the f1-score making it very bad.

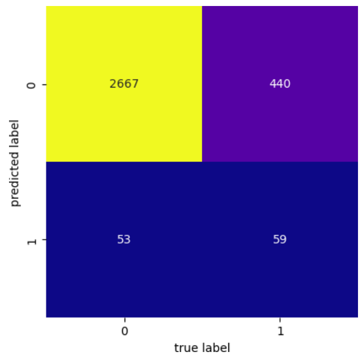
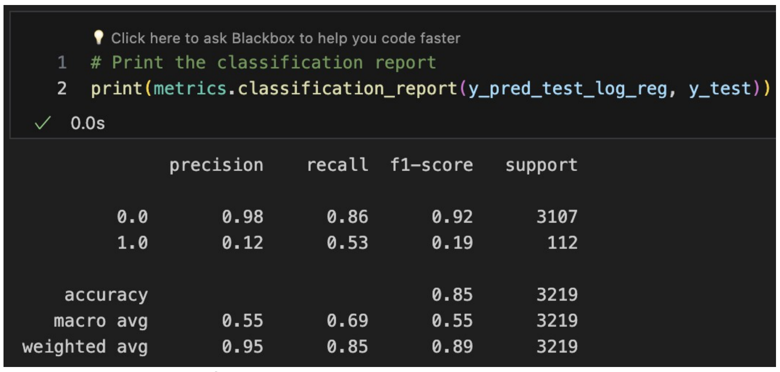


Fig. 9 - First confusion matrix

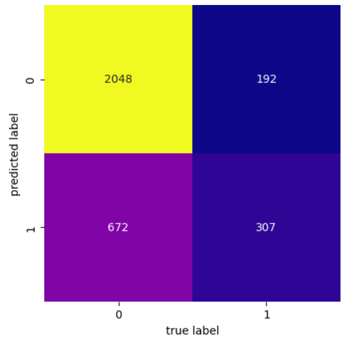


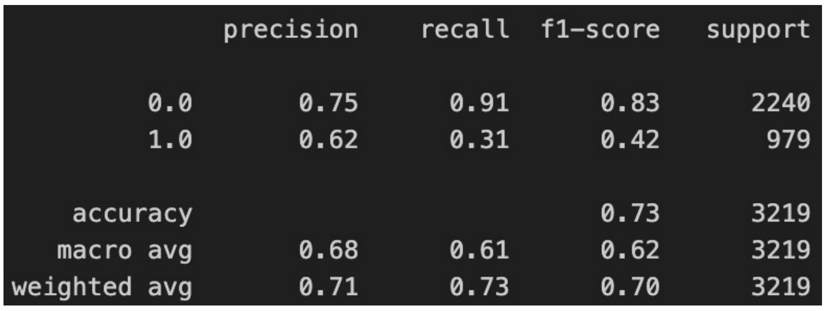
```
# Print the classification report
print(metrics.classification_report(y_pred_test_log_reg, y_test))
✓ 0.0s
              precision    recall  f1-score   support

         0.0       0.98      0.86      0.92      3107
         1.0       0.12      0.53      0.19       112

    accuracy                           0.85      3219
   macro avg       0.55      0.69      0.55      3219
weighted avg       0.95      0.85      0.89      3219
```

Fig. 10 - First classification report

After studying the hyperparameters of logistic regression, I decided to apply several of them. In particular, my goal was to increase the importance of class 1 in the eyes of the machine learning model. The most influential hyperparameter was ”*class_weight='balanced'*” because it automatically balances the weight of the classes during model training. The accuracy of this model decreased to 73%, but the performance improved significantly, reaching 62% precision and 42% f1-score (Fig. 12).

Fig. 11 - Second confusion matrix Fig. 12 - Second classification report





```
              precision    recall  f1-score   support

         0.0       0.75      0.91      0.83      2240
         1.0       0.62      0.31      0.42       979

    accuracy                           0.73      3219
   macro avg       0.68      0.61      0.62      3219
weighted avg       0.71      0.73      0.70      3219
```

[PART 4, RANDOM FOREST CLASSIFIER]

The other model I used is a random forest classifier. I chose it because can reduce overfitting compared to individual decision trees, as the final prediction is an average prediction from all the decision trees in the model.

Again, I started with a model without specified hyperparameters and obtained poor performance. The accuracy remains at 85%, but the initial precision and f1-score are only 19% and 29%, respectively.

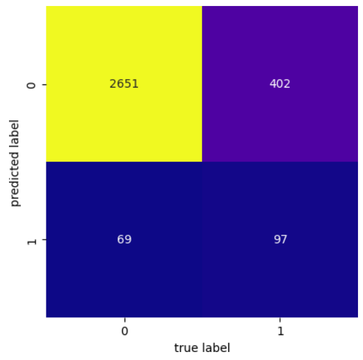
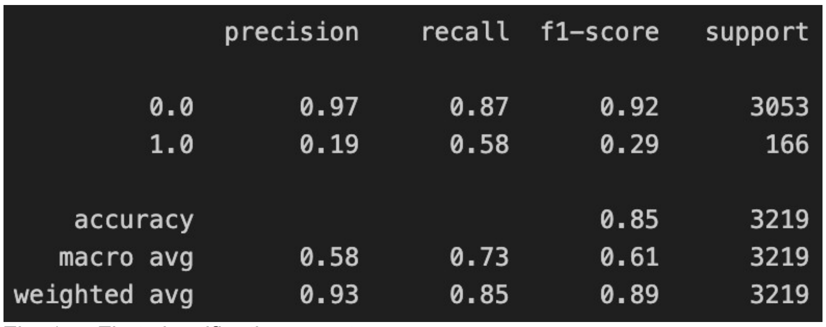


Fig. 13 - First confusion matrix



```
              precision    recall  f1-score   support

         0.0       0.97      0.87      0.92      3053
         1.0       0.19      0.58      0.29       166

    accuracy                           0.85      3219
   macro avg       0.58      0.73      0.61      3219
weighted avg       0.93      0.85      0.89      3219
```

Fig. 14 - First classification report

After studying the hyperparameters of the random forest classifier, I decided to apply several of them. I used "*max_leaf_nodes*" and "*max_depth*" to avoid overfitting on the training set, and I included "*class_weight='balanced'*" to ensure that each class had the same weight during model training. This led me to achieve a 78% accuracy with a precision of 58% and an F1-score of 44%. Since this performance is better than that of logistic regression, I feel satisfied with this model.

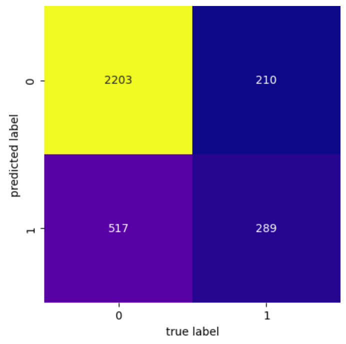
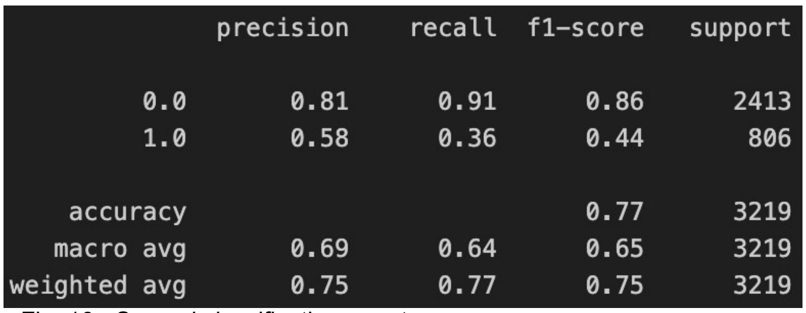


Fig. 15 - Second confusion matrix



```
              precision    recall  f1-score   support

         0.0       0.81      0.91      0.86      2413
         1.0       0.58      0.36      0.44       806

    accuracy                           0.77      3219
   macro avg       0.69      0.64      0.65      3219
weighted avg       0.75      0.77      0.75      3219
```

Fig. 16 - Second classification report

CONCLUSION

The provided dataset had several issues, including a high number of outliers and a target variable skewed towards class 0. I believe the latter was the most critical problem to address, and by using the hyperparameter "class_weight='balanced'", I consider that I tackled this discreetly.

In the end, I am satisfied with my project as I achieved an AUC value of 0.75 for the last model with a ROC curve that was positive in terms of points classification.