Home Credit Default Risk

Group members: Genovese Francesco, Jagarlapoodi Srinivas,
Malavita Daniela, Mioli Riccardo, Santini Guglielmo.

Master in Data Science and Business Analytics - 19/04/23

Contributions

- **Guglielmo**: Initial data exploration, data visualization;
- **Riccardo**: Data preparation, feature engineering;
- **Daniela**: Dimensionality reduction and class rebalancing, feature engineering;
- **Srinivas**: Training phase and hyperparameter tuning;
- **Francesco**: Testing phase and evaluation.

Problem description and resources available

Home Credit is a non-bank financial institution that provides consumer loans to people who have limited access to traditional banking services.

This dataset contains information about loan applicants and their previous credit history.

We want to predict if a customer is capable of repaying a loan.

application_{train|test}.csv [307511, 121]
This is the main table, broken into two files for Train (with TARGET) and Test (without TARGET).
Contains data asked to people that are requesting a loan.

bureau.csv [1716428, 16]
Contains the data regarding previous loans asked by the applicants.

previous_application.csv [1670214, 36]
Contains data of previously requested credits at Home Credit.

bureau_balance.csv [27299925, 3]
Monthly balances of previous credits in Credit Bureau.

POS_CASH_balance.csv [10001358, 7]
Contains monthly balance snapshots about the loans that customers have taken during the past with Home Credit.
Behavioral data

credit_card_balance.csv [3840312, 22]
Contains monthly balance of client's previous credit card loans in Home Credit

installments_payments.csv [13605401, 7]
Contains data regarding the monthly repayments of the loans previously insured by Home Credit.

HomeCredit_columns_description.csv
This file contains the descriptions for the columns in the various csv data.

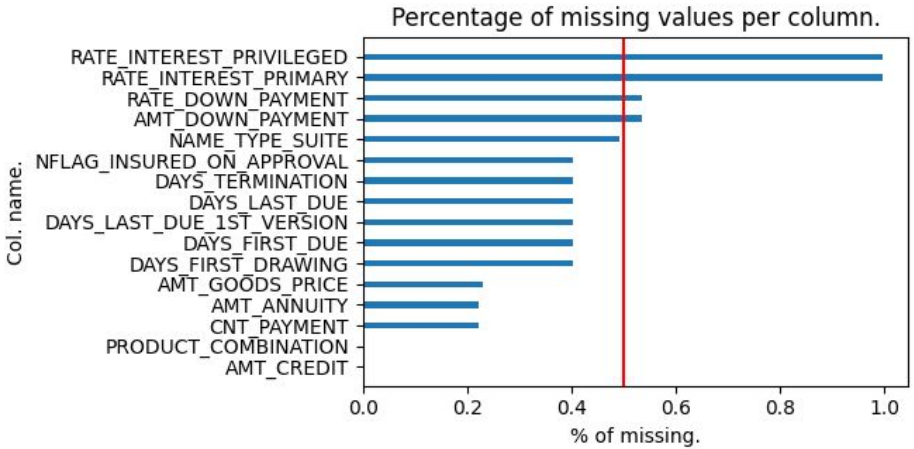
Data visualization

The function is used to plot the percentage of missing values, if present, for each column of the data-frame that will be passed as input.

```
def na_status(dataset: pd.DataFrame):
    shape = dataset.shape
    na_data = dataset.isna().sum().to_frame(name="na_count")
    if na_data.sum()[0] > 0:
        display("Dataset has %i rows and %i columns." % shape)
        display(dataset.head())
        display(na_data.transpose())
        na_data = na_data[na_data["na_count"] > 0]
        na_data.sort_values(ascending=True, by="na_count", inplace=True)
        plt.figure(figsize=(5, na_data.shape[0]*0.2))
        x = na_data["na_count"]/shape[0] #Number of nas in percentage
        plt.barh(y=na_data.index,
                 width=x, height=0.3)
        # If more or equal that 50% of values are missing, draw a red vline
        if x.max() >= 0.5:
            plt.axvline(0.5, color="r")
        plt.title("Percentage of missing values per column.")
        plt.xlabel("% of missing.")
        plt.ylabel("Col. name.")
        plt.show()
    else:
        print("No na values")
```

Data visualization and initial inspection

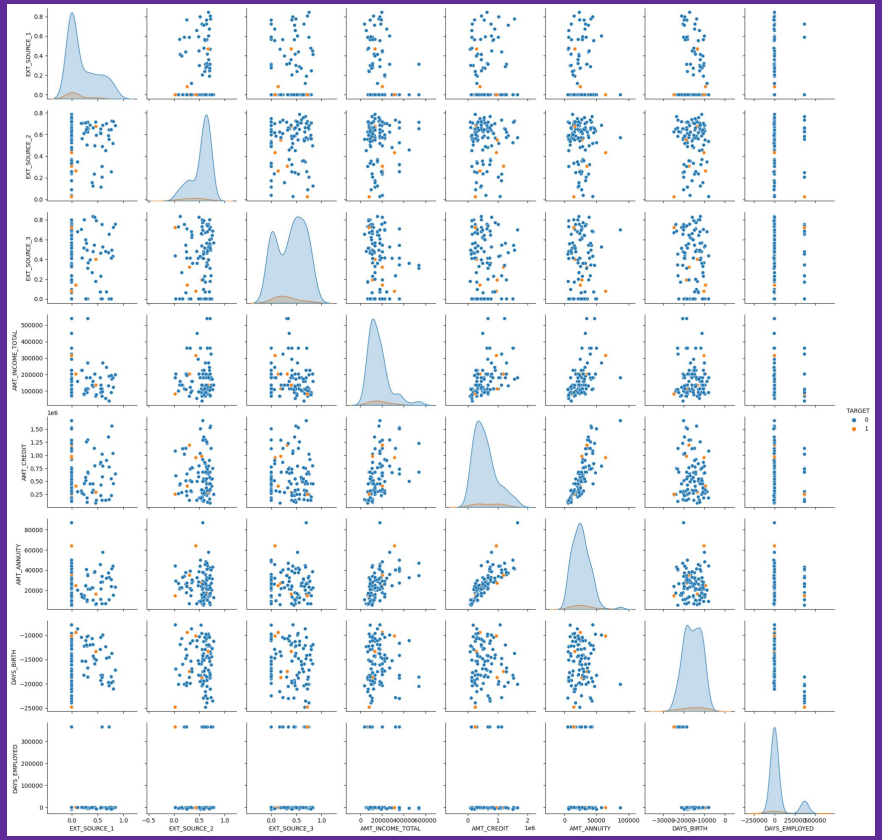
```
na_status(previous_appl)
```



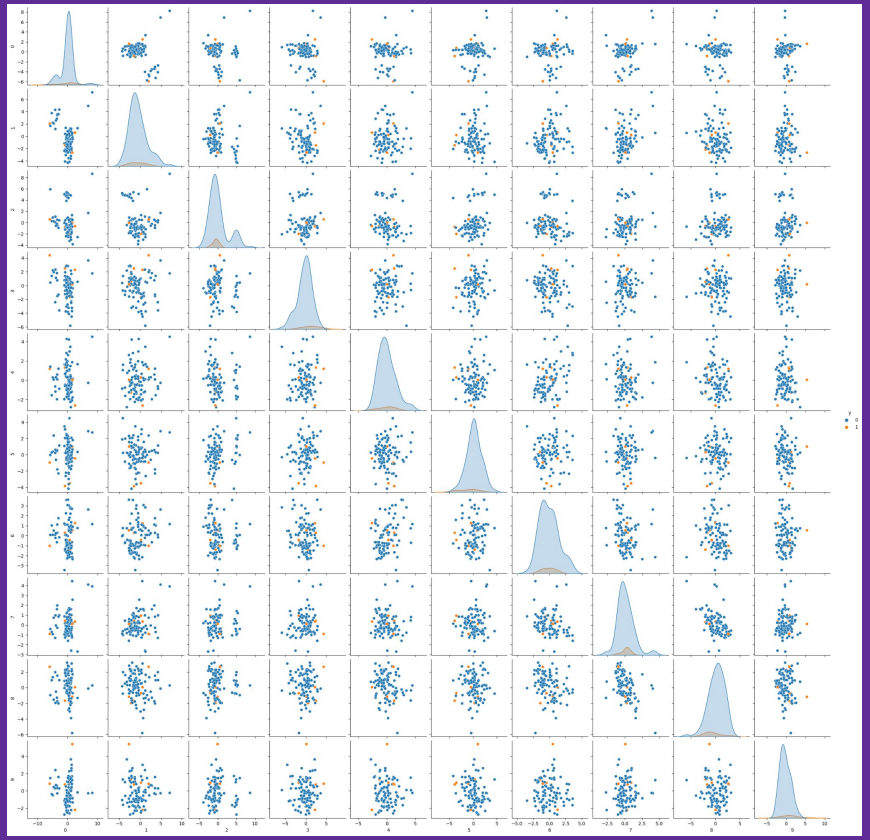

```
display(application.describe())
```

	TARGET	CNT_CHILDREN	AMT_INCOME_TOTAL	AMT_CREDIT	AMT_ANNUITY	AMT_GOODS_PRICE
count	307511.000000	307511.000000	3.075110e+05	3.075110e+05	307499.000000	3.072330e+05
mean	0.080729	0.417052	1.687979e+05	5.990260e+05	27108.573909	5.383962e+05
std	0.272419	0.722121	2.371231e+05	4.024908e+05	14493.737315	3.694465e+05
min	0.000000	0.000000	2.565000e+04	4.500000e+04	1615.500000	4.050000e+04
25%	0.000000	0.000000	1.125000e+05	2.700000e+05	16524.000000	2.385000e+05
50%	0.000000	0.000000	1.471500e+05	5.135310e+05	24903.000000	4.500000e+05
75%	0.000000	1.000000	2.025000e+05	8.086500e+05	34596.000000	6.795000e+05
max	1.000000	19.000000	1.170000e+08	4.050000e+06	258025.500000	4.050000e+06

Pairplot before and after PCA



Before pca



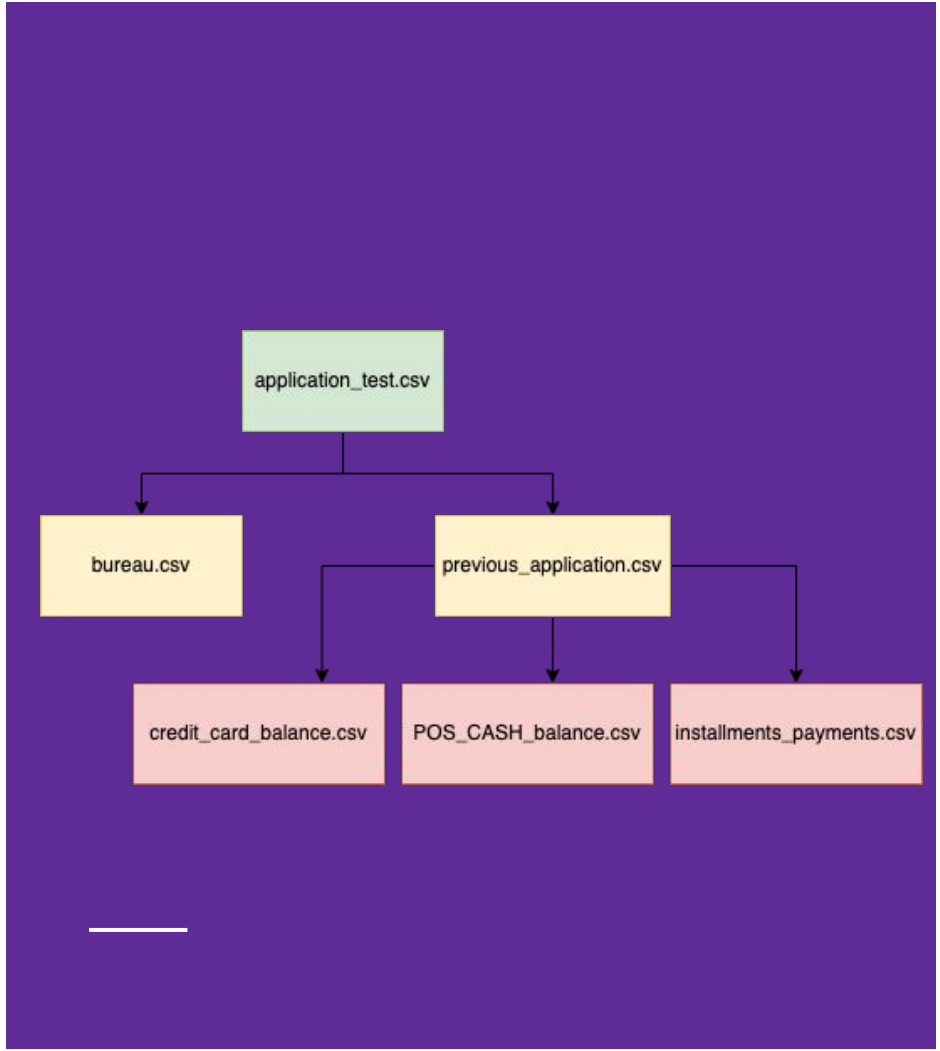
After pca

There is no obvious separation between the data classes for different values, so we can expect the task will be difficult.

Data preparation

The main tasks:

1. data aggregation and transformation of few columns;
2. transformation of temporal data;
3. merge of different .csv file;
4. Na value management;



Data transformation

```
application = pd.read_csv(os.path.join(
    BASE_PATH, "application_train.csv"), index_col=id_curr)
na_status(application)
# More than 50% of the values of those columns are null, we create new columns and drop the old ones.
application["car_age_avail"] = application["OWN_CAR_AGE"].notna().map({
    True: 1, False: 0})
application["house_data_avail"] = application.loc[:,
                                                  "APARTMENTS_AVG":"EMERGENCYSTATE_MODE"].notna().sum(axis=1)
application.drop(
    application.loc[:, "APARTMENTS_AVG":"EMERGENCYSTATE_MODE"], inplace=True, axis=1)
application.drop("OWN_CAR_AGE", inplace=True, axis=1)

# Interesting data, we will keep the values and put 0 in to fill the gaps
application.fillna({"EXT_SOURCE_1": 0, "EXT_SOURCE_2": 0, "EXT_SOURCE_3": 0, "AMT_REQ_CREDIT_BUREAU_YEAR": 0,
                    "AMT_REQ_CREDIT_BUREAU_QRT": 0, "AMT_REQ_CREDIT_BUREAU_WEEK": 0,
                    "AMT_REQ_CREDIT_BUREAU_MON": 0, "AMT_REQ_CREDIT_BUREAU_DAY": 0,
                    "AMT_REQ_CREDIT_BUREAU_HOUR": 0}, inplace=True)
```

Transformation of temporal data

```
previous_appl = pd.read_csv(os.path.join(
    BASE_PATH, 'previous_application.csv'), index_col=id_prev)
na_status(previous_appl)
previous_appl_new = {}
# Historical data of loans requested here.
previous_appl_new["n_rejections"] = previous_appl[previous_appl["NAME_CONTRACT_STATUS"]
                                                  == "Canceled"].groupby(id_curr)["NAME_CONTRACT_STATUS"].count()
previous_appl_new["n_approvals"] = previous_appl[previous_appl["NAME_CONTRACT_STATUS"]
                                                 == "Approved"].groupby(id_curr)["NAME_CONTRACT_STATUS"].count()
previous_appl_new["rejections_approval_ratio"] = previous_appl_new["n_rejections"] / \
    previous_appl_new["n_approvals"]
# Avg previously accepted credits, rejected, goods price
previous_appl_new["avg_sum_accepted"] = previous_appl[previous_appl["NAME_CONTRACT_STATUS"]
                                                      == "Approved"].groupby(id_curr)["AMT_CREDIT"].mean()
previous_appl_new["avg_sum_rejected"] = previous_appl[previous_appl["NAME_CONTRACT_STATUS"]
                                                      == "Canceled"].groupby(id_curr)["AMT_CREDIT"].mean()
previous_appl_new["avg_goods_price_accepted"] = previous_appl[previous_appl["NAME_CONTRACT_STATUS"]
                                                              == "Approved"].groupby(id_curr)["AMT_GOODS_PRICE"].mean()
previous_appl_new["avg_goods_price_rejected"] = previous_appl[previous_appl["NAME_CONTRACT_STATUS"]
                                                              == "Canceled"].groupby(id_curr)["AMT_GOODS_PRICE"].mean()
# Average sums granted and rejected during the last 3, 6, 12, 24 months
for i, j in [(0, 3), (3, 6), (6, 12), (12, 24)]:
    bool_mask_approved = (previous_appl["DAYS_DECISION"] >= -j*30) & (
        previous_appl["DAYS_DECISION"] <= -i*30) & (previous_appl["NAME_CONTRACT_STATUS"] == "Approved")
    bool_mask_rejected = (previous_appl["DAYS_DECISION"] >= -j*30) & (
        previous_appl["DAYS_DECISION"] <= -i*30) & (previous_appl["NAME_CONTRACT_STATUS"] == "Canceled")
    previous_appl_new["avg_sum_accepted_%im" % j] = previous_appl[bool_mask_approved].groupby(id_curr)[
        "AMT_CREDIT"].mean()
    previous_appl_new["avg_sum_rejected_%im" % j] = previous_appl[bool_mask_rejected].groupby(
        id_curr)["AMT_APPLICATION"].mean()
```

Data preparation

- **application**: data regarding this application (gender, amount of goods, amount required, house data, car data, child data, income, etc.);
- **bureau**: data regarding previous loans in different institutes (number of open credits, open credits, count of credit prolongs, days past due, amount still to be paid, etc.)

- **previous_application**: data regarding previous loans at HomeCredit (number of rejections, approvals, average sum accepted and rejected, avg sum price, data about latest loan rejected and approved)

Merge of the final dataframes

- **POS_CASH_balance:** data about trend of remaining installments and days past due;
- **installment_payments:** trend of payments during the last months (average payment and deviation in payments);
- **credit_card_balance:** trend of credit card payments, credit limit, credit repayments, etc;

We perform a left join, otherwise we would lose all the samples which are not present in the other dataframes.

```
application = application.merge(bureau_new, how="left", on=id_curr)
application = application.merge(previous_appl_new, how="left", on=id_curr)
application = application.merge(pos_new, how="left", on=id_curr)
application = application.merge(inst_pay_new, how="left", on=id_curr)
application = application.merge(cc_bal_new, how="left", on=id_curr)
```

At this point we have 307k lines and 295 columns.

Feature Engineering

General Finance notions on **LTV** and **DTI:**

- They are used by banks for specifically calculating **credit scores for mortgages**
- **Lower LTV – lower mortgage rate**
- **Higher LTV – higher mortgage rate**
- **Average LTV** in Italy is around **65%**
- Italian banks prefer for the **DTI** to be **no more than 30%**

Feature engineering consists in generating **new columns** by **performing operations** on the **existing ones.**

New columns generated are:

- **Loan To Value**
- **Debt To Income**
- **Good_empl**
- **Downpayment**
- **Age**
- **NEW_PRICE_RATIO** (for accepted and rejected requests)
- **NEW_AMT_CREDIT_RATIO** (for accepted and rejected requests)

```
# Features engineereed
application["ltv"] = application["AMT_CREDIT"]/application["AMT_GOODS_PRICE"]
application["dti"] = application["AMT_ANNUITY"]/application["AMT_INCOME_TOTAL"]
application["good_ind"] = application["AMT_INCOME_TOTAL"] / \
    application["CNT_FAM_MEMBERS"]
```

Dimensionality Reduction and Scaling

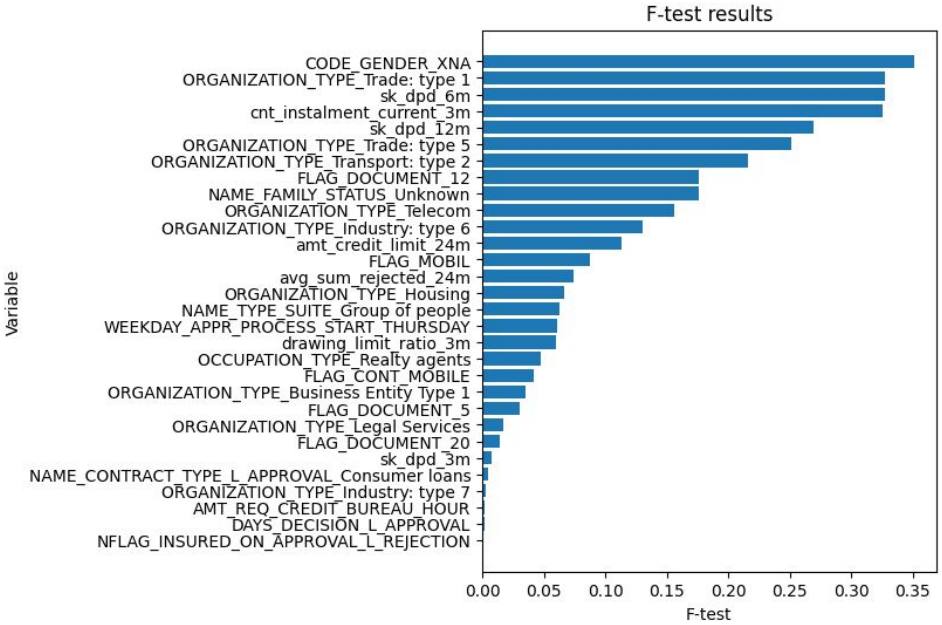
In this part of the project, several techniques to **reduce** the **number of columns without losing** too much **information** were implemented.
The techniques are in the following order: **One-hot Encoding, FillNa, Feature Selection with F test and PCA.**

Since the **vast majority of the models** only support **numerical data**, **dummy encoding** was performs and several columns were condensed into one (e.g. gender)

The NA values were filled with the **most frequent value** for the selected column inside the dataset

```
# Data scaling
# Normalizer, MinMaxScaler
transformer = StandardScaler()
X = transformer.fit_transform(X)
```

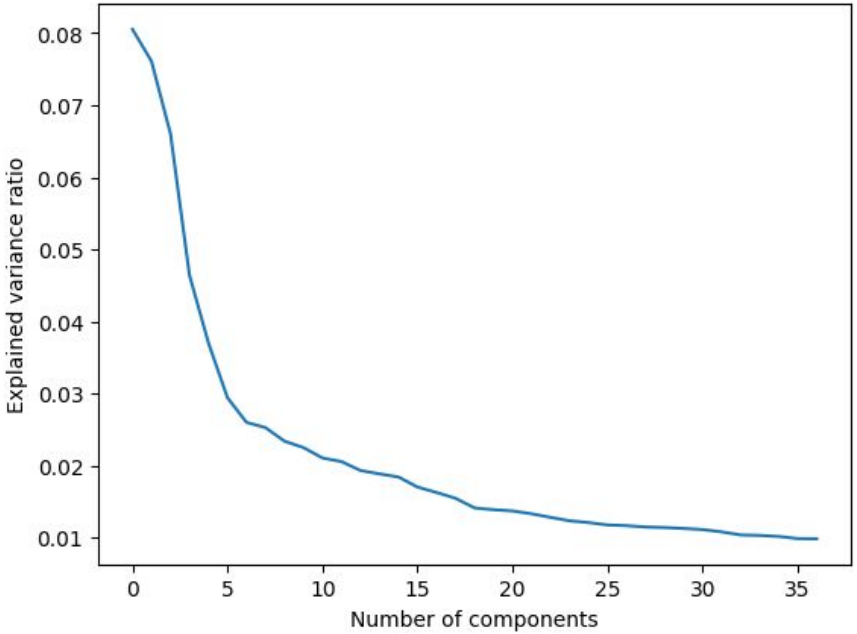
Feature Selection



Feature Selection was implemented by using **f_regression** and **fit_transform.** The results of the F-tests explain that there is **not one variable that alone will be able to predict if a person will not give back the loan or not.**

```
# Feature selection
X = SelectPercentile(f_regression, percentile=30).fit_transform(X, y)
```

Scaling and PCA

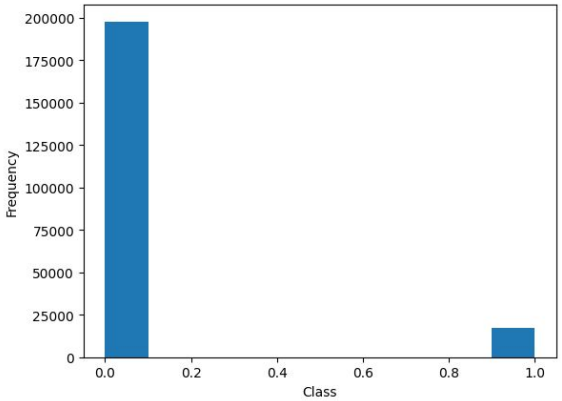


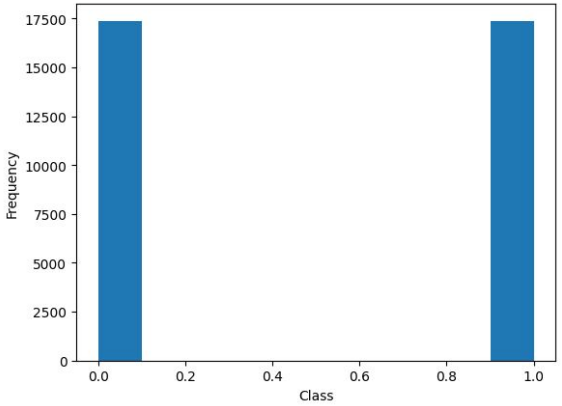
In order to perform a **Principal Component Analysis**, the data was scaled using **StandardScaler.**

The PCA was set in order for the Principal Components to keep an **80% explained variance.**

```
# Dimensionality reduction
n_components = .8
pca = PCA(n_components)
X = pca.fit_transform(X, y)
```

Splitting and Rebalancing





The dataset was split into the four train and test sets using **Stratify** in order to **keep the proportion of the classes** in both input and output for training and testing sets.

In order for the model **not to be biased towards only class**, the dataset was **rebalanced** using **RandomUndersampler.**

```
# Dataset balancing
plt.hist(Y_train)
plt.xlabel("Class")
plt.ylabel("Frequency")
plt.show()
# RandomUndersampler() RandomOversampler() SMOTE() BorderlineSMOTE()
balancer = RandomUnderSampler()
X_train, Y_train = balancer.fit_resample(X_train, Y_train)
plt.hist(Y_train)
plt.xlabel("Class")
plt.ylabel("Frequency")
plt.show()
```

Training Model Selection and Tuning

```
"rf": {"name": "Random Forest",
       "enabled": 1,
       "estimator": RandomForestClassifier(),
       "params": {"n_estimators": list(range(100, 500)),
                  "criterion": ["gini", "entropy", "log_loss"],
                  "max_depth": list(range(1, 30)),
                  "min_samples_split": list(range(2, 20)),
                  "min_samples_leaf": list(range(1, 30))
                  }
       },
```

- 5 machine learning models: Decision Tree, Random Forest, Naive Bayes, Support Vector Machine, and Multi-Layer Perceptron
- Hyperparameters tuned to improve performance with RandomizedSearchCV function
- Best estimator for each model added to "optimized" dictionary based on F1 score

Ensemble Learning with VotingClassifier

```
for key, model in models.items():
    if model["enabled"]:
        t0 = time.time()
        log_level = 0  # Set 3 for logging purposes
        jobs = os.cpu_count() - 1
        optimizer = RandomizedSearchCV(model["estimator"], model["params"],
                                       scoring="f1_macro", n_jobs=jobs, n_iter=10, cv=5, refit=True
        trained_model = optimizer.fit(X_train, Y_train)
        optimized[model["name"]] = trained_model.best_estimator_
        print("Training %s, " % model["name"] + "elapsed time: %f, " %
              (time.time() - t0) + "f1 score: %f" % trained_model.best_score_)
        print(trained_model.best_params_)
    else:
        print("Skipping", model["name"], "because is not enabled.")
optimized["Voting Classifier"] = VotingClassifier(
    list(optimized.items())).fit(X_train, Y_train)
```

- VotingClassifier combines optimized models to predict highest-voted class
- Trained on X_train and Y_train and added to "optimized" dictionary
- "Optimized" dictionary stores best estimator and VotingClassifier, F1 score and best hyperparameters printed for each enabled model

Evaluation Phase

```python
# Test over test set
print("[Macro Metrics]")
for name, model in optimized.items():
    y_pred = model.predict(x_test)
    #Macro scores
    macro_scores = precision_recall_fscore_support(y_test, y_pred, average="macro")
    print(name, "Precision: %f, Recall: %f, f1: %f" % macro_scores[:3])
    matrix = confusion_matrix(y_test, y_pred)
    matrix = ConfusionMatrixDisplay(confusion_matrix=matrix,
                                    display_labels=model.classes_)
    matrix.plot()
    matrix.ax_.set_title(name)
```

✓ 5m 43.1s

```
[Macro Metrics]
Decision Tree Precision: 0.540358, Recall: 0.622759, f1: 0.502570
Random Forest Precision: 0.552746, Recall: 0.663600, f1: 0.514247
Naive Bayes Precision: 0.531512, Recall: 0.565014, f1: 0.528880
Support Vector Machine Precision: 0.557578, Recall: 0.674771, f1: 0.526297
Multi Layer Perceptron Precision: 0.556625, Recall: 0.672603, f1: 0.524005
Voting Classifier Precision: 0.557308, Recall: 0.669826, f1: 0.531122
```

In the Evaluation Phase we want to evaluate our classification models, doing the following operations:

- making prediction on the test set
- calculating the confusion matrix
- calculate precision, recall and F1
- Print the results of the evaluation metrics for the model
- visualize the confusion matrix

Evaluation Phase without macro averaging

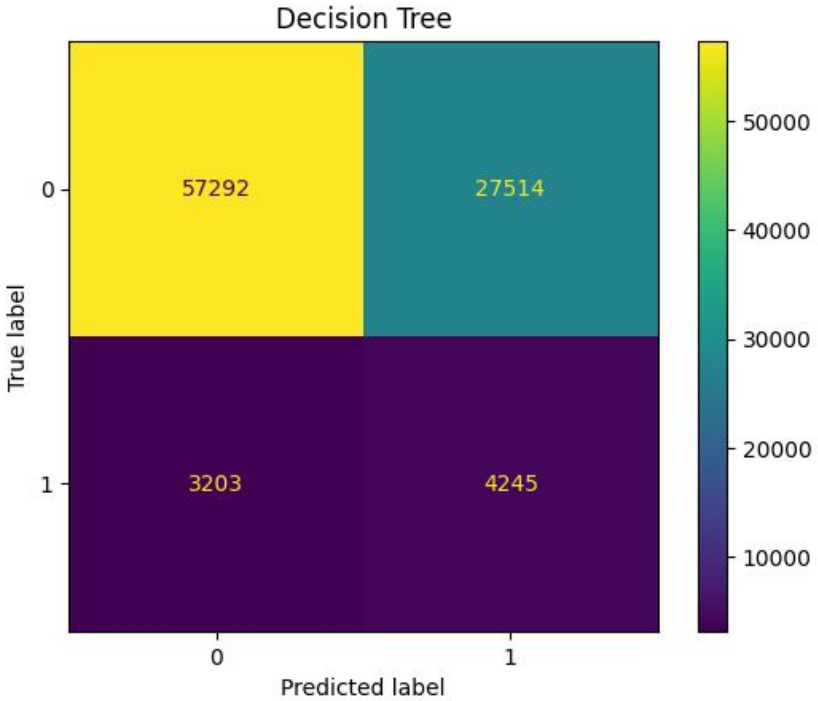
```
print("[Metrics for each class]")
for name, model in optimized.items():
    y_pred = model.predict(x_test)
    #Non-macro scores
    scores = precision_recall_fscore_support(y_test, y_pred)
    print(name, "Precision: %s, Recall: %s, f1: %s" % scores[:3])
```

✓ 5m 36.6s

```
[Metrics for each class]
Decision Tree Precision: [0.94705348 0.1336629 ], Recall: [0.67556541 0.56995166], f1: [0.78859746 0.21654296]
Random Forest Precision: [0.95716533 0.14832695], Recall: [0.66715798 0.66004296], f1: [0.78627264 0.24222118]
Naive Bayes Precision: [0.93116165 0.13186308], Recall: [0.82175789 0.30827068], f1: [0.8730457 0.1847144]
Support Vector Machine Precision: [0.95873602 0.15641999], Recall: [0.68547037 0.66407089], f1: [0.79939494 0.25319955]
Multi Layer Perceptron Precision: [0.95842792 0.15482273], Recall: [0.68207438 0.66313104], f1: [0.79697435 0.2510356 ]
Voting Classifier Precision: [0.9566995 0.1579157], Recall: [0.70082305 0.63882922], f1: [0.80901109 0.25323327]
```

We calculate again the precision, recall and F1 score without the macro averaging to show the negative impact on the results of the class that do not repay their debts.

Evaluation Phase confusion matrix



Here we show the confusion matrix for one of the models, as you can see there are 57292 customers who give back the money and 27514 who don't which are incorrectly classified.

Then there are 3203 customers who were classified as good customers but did not paid back the money and 4245 customer that we made the right choice to not give them the money.

Conclusions

The performances of the classifiers are not very good in recognizing people that will not give back the loans:

1. The dataset is highly unbalanced. We are not able to extract sufficient knowledge to separate well the minority class (who doesn’t give back the loan);
2. The features created do not separate very well the two classes;
3. The large amount of missing data has a negative impact on this task.

Lessons learned

- A lot of time was expended in data integration from various .csv files.
- Real world data has a lot of missing values and finding the right way to replace those values is very difficult.

- Training with this relatively small dataset required time. When we tried oversampling an entire night passed and the process was still not finished. Parallelize the training is important.