Applications of Big Data

Project Report

Adin HRELJA
Grégoire NOWACKI

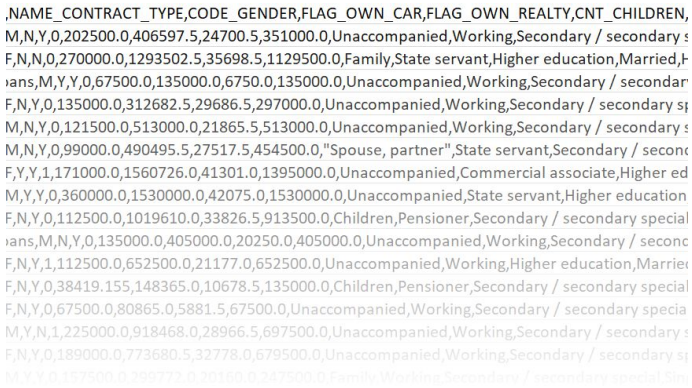
Summary

I. Data Processing & ML Models

1. Data Exploration and Feature Engineering

Throughout this project, we will be working on a very specific dataset that is available on the Kaggle website. It is called **Home Credit Default Risk**, and is composed of 10 different csv files (however, we will only be working on 2 files, which are *application_train.csv* and *application_test.csv*). This dataset is focused on bank datas with a big amount of features (there are 122 in the train csv). Each line has info on a specific client case stored as either numerical or categorical values, and has a specific column registering if the client will be able to repay his loan in time (which is either 0 or 1).

This dataset is a great exercise to perform some Machine Learning models : a large number of features of many different types, and a target value, our labels (which makes it a supervised Machine Learning problem).



In order to perform some data exploration, we will be using Jupyter Notebook first, and we will be working within a conda environment to make it easier to work in a group, and to avoid conflicts with our current base environment.

The first thing we did was to check out every feature types we currently had to work with, and how many of each.

```
1  train.dtypes.value_counts()
```

```
float64    65
int64      41
object     16
dtype: int64
```

Next, we will be checking our categorical variables, to see how much categories each feature have within the same column.

```
train.select_dtypes('object').apply(pd.Series.nunique, axis = 0)
```

```
NAME_CONTRACT_TYPE              2
CODE_GENDER                     3
FLAG_OWN_CAR                    2
FLAG_OWN_REALTY                 2
NAME_TYPE_SUITE                 7
NAME_INCOME_TYPE                8
NAME_EDUCATION_TYPE             5
NAME_FAMILY_STATUS              6
NAME_HOUSING_TYPE               6
OCCUPATION_TYPE                18
WEEKDAY_APPR_PROCESS_START      7
ORGANIZATION_TYPE              58
FONDKAPREMONT_MODE              4
HOUSETYPE_MODE                  3
WALLSMATERIAL_MODE              7
EMERGENCYSTATE_MODE             2
dtype: int64
```

As we can see, we have some very different categorical features, where some only have only 2 categories, and some have way more (ORANIZATION_TYPE has 58 different types of categories for example). In order to use this data through Machine Learning models, we will have to perform some Label Encoding and One Hot Encoding on it.

Another important thing to check is the amount of NaN values we have in each feature, which has a critical impact on how our model will perform.

```
train_missing = (train.isnull().sum() / len(train)).sort_values(ascending = False)
train_missing.head(20)

test_missing = (test.isnull().sum() / len(test)).sort_values(ascending = False)
test_missing.head(20)

train_missing = train_missing.index[train_missing > 0.3]
test_missing = test_missing.index[test_missing > 0.3]

all_missing = list(set(set(train_missing) | set(test_missing)))
print('There are %d columns with more than 30%% missing values' % len(all_missing))
```

```
There are 50 columns with more than 30% missing values
```

As you can see, the dataset has a lot of NaN values. We will need to fill every NaN values in order to pass the data through models (because RandomForest doesn't support these, opposed to XGBoost). There are 50 features that have over 30% missing values.

Now, we will proceed in cleaning the data to make it usable in our Machine Learning models.

First, to simplify the cleaning for now, we chose to use 30% as a threshold for the feature we decided to get rid of.

```
###READING EVERY FEATURE WITH OVER 30% MISSING VALUES

train_missing = (train.isnull().sum() / len(train)).sort_values(ascending = False)
test_missing = (test.isnull().sum() / len(test)).sort_values(ascending = False)
train_missing = train_missing.index[train_missing > 0.3]
test_missing = test_missing.index[test_missing > 0.3]

###DROPPING EVERY FEATURE WITH OVER 30% MISSING VALUES

for i in train_missing:
    train = train.drop([i], axis = 'columns')

for i in test_missing:
    test = test.drop([i], axis = 'columns')
```

After these few lines, we got rid of the 50 features previously mentioned in both the training and testing set.

Then, we will fill every NaN values in numerical variables with the mean of each column.

```
cat_vars = train.select_dtypes('object').columns.tolist()

###FILLING NaN IN NUMERICAL VALUES WITH THE MEAN OF EACH COLUMN

for col in train:
    if col not in cat_vars:
        train[col] = train[col].fillna(train[col].mean())

for col in test:
    if col not in cat_vars:
        test[col] = test[col].fillna(test[col].mean())
```

Finally, we will drop the remaining lines with NaN values (the only remaining Nan values are contained in categorical variables).

```
###DROPPING REMAINING LINES WITH NaN VALUES (FROM CATEGORICAL VARIABLES)

train = train.dropna(axis=0, how='any', subset=cat_vars)
test = test.dropna(axis=0, how='any', subset=cat_vars)
```

We now have both our training and testing dataset free of any NaN values. We can now perform Label Encoding on every categorical variables with up to 2 categories, and One Hot Encoding on the other ones.

```
###LABEL ENCODING

le = LabelEncoder()

for col in train:
    if train[col].dtype == 'object':
# If 2 or fewer unique categories
        if len(list(train[col].unique())) <= 2:
            le.fit(train[col])
            train[col] = le.transform(train[col])
            test[col] = le.transform(test[col])

###ONE HOT ENCODING

train = pd.get_dummies(train)
test = pd.get_dummies(test)
```

Finally, as a final step, we can align the training and testing set to get feature consistency. After that, we can define X and y.

```
###ALIGN TRAIN AND TEST

train, test = train.align(test, join='inner', axis=1)
train['TARGET'] = labels

###DEFINE X AND Y

X = train.drop(columns=['TARGET'])
y = train['TARGET']
```

Now, our data is ready to be used in some Machine Learning models.

2. Machine Learning Models

The goal of this project, as said earlier, is to predict the TARGET value. In order to do that, we will be using three different models which are XGBoost Classifier, Random Forest Classifier and Gradient Boosting Classifier.

- XGBoostClassifier :

```
seed = 7
test_size = 0.33
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=seed)
model = XGBClassifier()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

Using the XGBClassifier from the xgboost package, we are able to create a model and train it with our training dataset.

```
Accuracy: 91.81%
Classification report:
              precision    recall  f1-score   support

           0       0.92      1.00      0.96     92837
           1       0.44      0.03      0.05      8216

    accuracy                           0.92    101053
   macro avg       0.68      0.51      0.50    101053
weighted avg       0.88      0.92      0.88    101053

Confusion matrix:
[[92563   274]
 [ 8003   213]]
```

- RandomForestClassifier :

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

classifier = RandomForestClassifier(n_estimators=20, random_state=0)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
```

Using the RandomForestClassifier from the sklearn package, we are able to create a model and train it with our training dataset.

```
Accuracy: 91.83%
Classification report:
              precision    recall  f1-score   support

           0       0.92      1.00      0.96     92807
           1       0.45      0.00      0.01      8246

    accuracy                           0.92    101053
   macro avg       0.69      0.50      0.48    101053
weighted avg       0.88      0.92      0.88    101053

Confusion matrix:
[[92760    47]
 [ 8207    39]]
```

- GradientBoostingClassifier :

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
clf = GradientBoostingClassifier(random_state=0)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

Using the GradientBoostingClassifier from the sklearn package, we are able to create a model and train it with our training dataset.

```
Accuracy: 91.87%
Classification report:
              precision    recall  f1-score   support

           0       0.92      1.00      0.96     92807
           1       0.58      0.01      0.03      8246

    accuracy                           0.92    101053
   macro avg       0.75      0.51      0.49    101053
weighted avg       0.89      0.92      0.88    101053

Confusion matrix:
[[92727    80]
 [ 8137   109]]
```

We get a very good accuracy on all models (they are very similar), which is highly biased by the very high amount of 0 in our dataset that are correctly predicted. The problem is that the amount of 1 is very low and the ratio of predictions is not the same for both classes. The data in itself is very **unbalanced**. In order to get better predictions, we would have to balance the dataset by generating more data for the 1 class using some other Machine Learning models.

II. MLFlow Integration

In this next part, we will be implementing our models on MLFlow, which is a platform made especially for the Machine Learning lifecycle. With this platform, we will be able to track our different models with many different parameters and to compare the different outputs we get. In order to work like this, we will be stepping away from the usual notebook format we're used to using : we reworked our code in a script format.

First, we created a python file called *preprocessing_data.py* to process our csv files as described in the first part. In this file, two functions are defined : *preprocessing_train()* and *preprocessing_test()*, which returns respectively the training set and the testing set processed and ready to be used in our models.

Next, we will be creating a *train.py* for each models we want to track. We created *trainxgb.py, pytrainrf.py* and *traingbc.py*, which are respectively scripts for training XGBoostClassifier, RandomForestClassifier and GradientBootingClassifier.

As arguments, we chose **learning_rate** (for XGBoost and GradientBoostingClassifier), **n_estimators** and **n_jobs**, which are some interesting arguments to tweak to get different outputs from our models. For the metrics to track, we decided to send to MLFlow every metrics returned by a classification report : **precision**, **recall**, **f1-score** and **support** (for each 0 and 1 classes we want to predict). We track the **accuracy** metric as well.

```
mlflow.log_param("learning_rate", lr)
mlflow.log_param("n_estimators", ne)
mlflow.log_param("n_jobs", nj)
mlflow.log_metric("precision0", precision0)
mlflow.log_metric("precision1", precision1)
mlflow.log_metric("recall0", recall0)
mlflow.log_metric("recall1", recall1)
mlflow.log_metric("fscore0", fscore0)
mlflow.log_metric("fscore1", fscore1)
mlflow.log_metric("support0", support0)
mlflow.log_metric("support1", support1)
mlflow.log_metric("accuracy", accuracy)
```

Now that everything is set up correctly, we can start running our models with many different parameters, track and then compare the results.

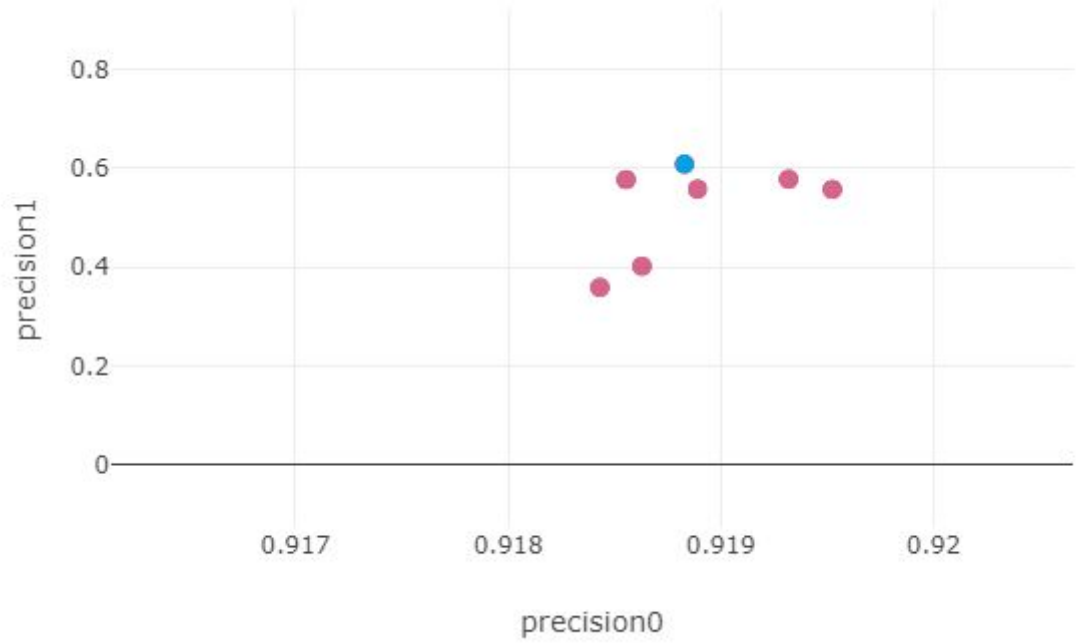
First, let's run *mlflow ui* in our project directory to track our current *mlruns* folder.

For an example, we will be running XGBoost Classifier 3 times, RandomForest Classifier 2 times and GradientBoostingClassifier 2 times with different parameters. This is what we get inside our MLFlow UI when we compare the 7 runs :

Parameters							
learning_rate	0.1	0.05	0.05	0.1	0.1		
n_estimators	100	100	100	20	100	100	20
n_jobs			-1	-1	-1	-1	-1
Metrics							
accuracy	0.919	0.919	0.919	0.918	0.919	0.918	0.918
fscore0	0.958	0.958	0.958	0.957	0.958	0.957	0.957
fscore1	0.026	0.012	0.014	0.005	0.032	0.001	0.007
precision0	0.919	0.919	0.919	0.919	0.92	0.918	0.919
precision1	0.577	0.607	0.557	0.576	0.556	0.357	0.4
recall0	0.999	1	0.999	1	0.999	1	1
recall1	0.013	0.006	0.007	0.002	0.016	6.064e-4	0.004
support0	92807	92807	92807	92807	92807	92807	92807
support1	8246	8246	8246	8246	8246	8246	8246

As we can see, the **accuracy** is very similar on each run, but it's not a very pertinent metric to track in our case, as explained before. However, the **precision** on the 1 class is different on every model (while the 0 class **precision** is high in all cases).

We can do a scatter plot to see both precisions on both axis :



On this plot, we highlighted a point, which is the model we will be using for our deployment on the HTTP server. The **precision** on the 0 class is similar in all cases, but the **precision** on the 1 class is the highest on this one : this is the most coherent choice we can make. It is a GradientBoostingClassifier with the following parameters : **learning_rate** = 0.05, **n_estimators** = 100.

After that, we want to deploy our model on a HTTP server in order to do some predictions on our model with some requests.

First, in our MLProject file, we replace our default parameters with the parameter we want to use :

```
name: aobdproject

conda_env: conda.yaml

entry_points:
  main:
    parameters:
      learning_rate: {type: float, default: 0.05}
      n_estimators: {type: int, default: 100}
    command: "python traingbc.py {learning_rate} {n_estimators}"
```

Then, we set our dependencies in our conda.yaml file :

```
name: aobdproject
channels:
  - defaults
  - anaconda
  - conda-forge
dependencies:
  - python=3.8
  - scikit-learn =0.23.1
  - pip
  - mlflow>=1.0
  - pip:
    - xgboost
```

After all this is done, we can run it with the *mlflow run .* command.

This id is the id of the model we will be using to deploy the server. Finally, we can deploy the model on the server using the following command :

mlflow models serve -m mlruns\0\a844b3c7d62b48aa8916971686bb7ea2\artifacts\GradientBoostingClassifier -p 1234

After executing this command, the server is up and running, and ready to receive some requests for predictions !

```
Serving on http://kubernetes.docker.internal:1234
```

III. XAI with SHAP Method

XAI refences to Explainable AI. It is used to make understandable AI models. During the project we execute a XGBoost classifier model and use Shap (python package) to explain it. In this part, we are going to see some explanation about the xgboost model that we did previously.

AMT_CREDIT	599657.288601
AMT_GOODS_PRICE	538916.385634
AMT_INCOME_TOTAL	168407.710963
DAYS_EMPLOYED	64718.515027
AMT_ANNUITY	27110.114626
HOUR_APPR_PROCESS_START	12.073288
CNT_FAM_MEMBERS	2.155354
REGION_RATING_CLIENT	2.053131
REGION_RATING_CLIENT_W_CITY	2.031756
AMT_REQ_CREDIT_BUREAU_YEAR	1.903869

Individual SHAP Values

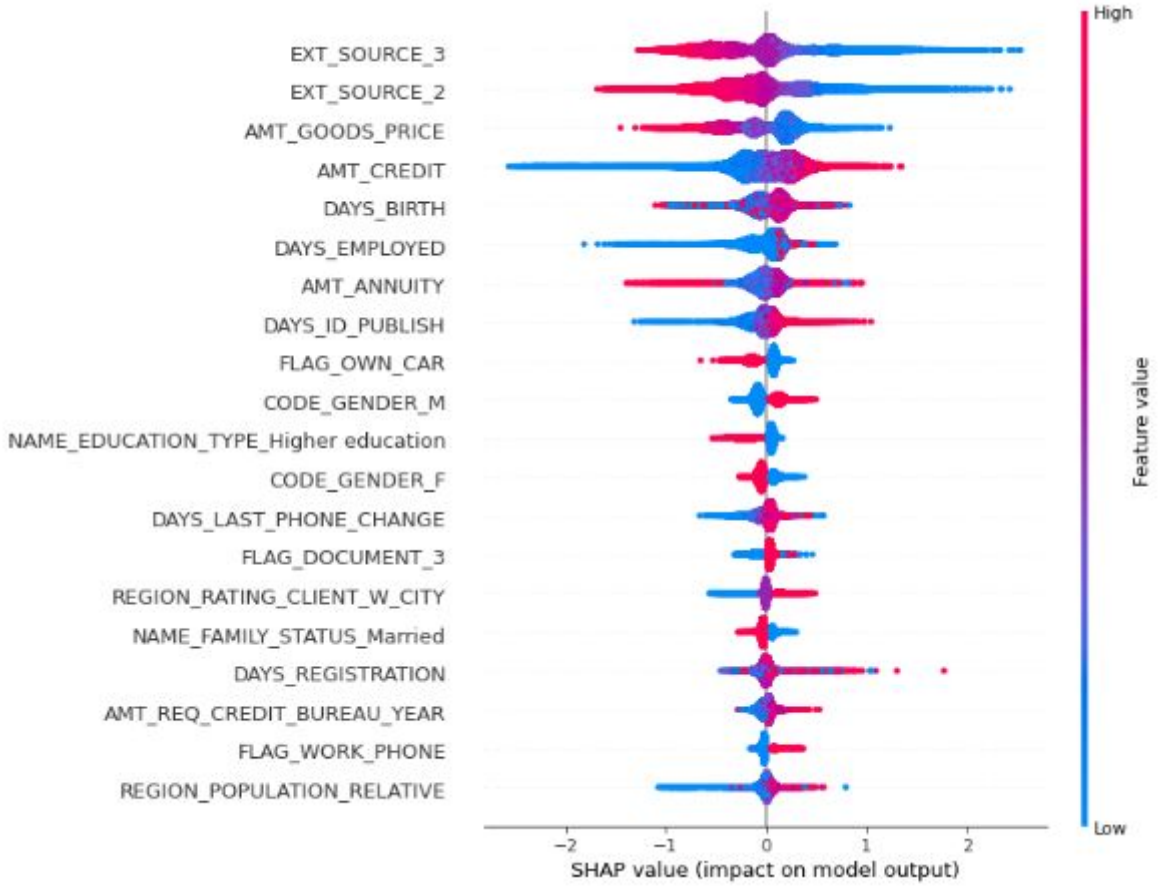
Individual SHAP Value Plot for Observation [NUMERO_OBS] of the random raw

Describe :

- The *output value* is the prediction for that observation (*output value* = -2.90);
- The *base value* : according to the original paper, the base value E(ŷ) is the value that would be predicted if we did not know any features for the current output. It is the mean prediction, or mean(ŷ);

- Red/blue (color) : Features that push the prediction higher (to the right) are shown in red, and those pushing the prediction lower are in blue;
- AMT_CREDIT : has a negative impact on the quality rating. A lower than the average AMT_CREDIT (= 4.975e+5 < 599657.29) drives the prediction to the left.
- To know how we know the average values of the predictors, it is the X_train.mean(). The SHAP model is built on the training data set.

Summary Plot



- *Feature importance* : Variables are ranked in descending order
- *Impact* : The horizontal location shows whether the effect of that value is associated with a higher or lower prediction
- *Original value* : Color shows whether that variable is high (in red) or low (in blue) for that observation
- *Correlation* : A high lever of the “EXT_SOURCE_3” content has a high and negative impact on the quality rating. The low comes from the blue color, and the negative impact is shown on the X-axis. Similarly, we will say the volatile acidity is positively correlated with the target variable.

Variable Importance plot

There is a simplified version for easier interpretation. It represents the importance feature of the model. A variable importance plot lists the most significant variables in descending order. The top variables contribute more to the model than the bottom ones and thus have high predictive power.