**Methodology**

**OVERVIEW**

We are a Data Scientist within the financial institution "Prêt à dépenser" which give loans to people with insufficient or non-existent credit history.



The company wishes to implement a credit scoring model in order to make good decisions.

Two types of risks are associated with the bank’s decision:

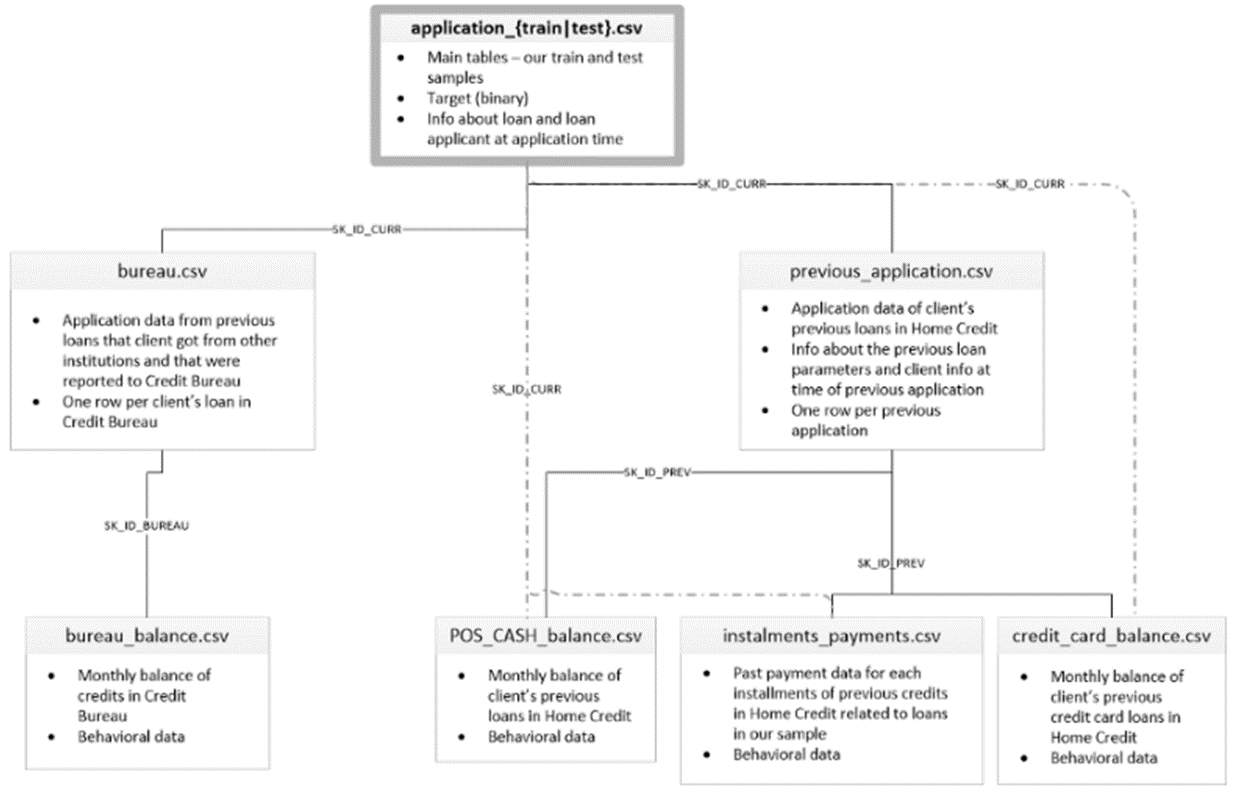
* If the applicant is likely to repay the loan, then not approving the loan results in a loss of business to the company
* If the applicant is not likely to repay the loan, i.e. he/she is likely to default, then approving the loan may lead to a financial loss for the company.

In addition, customers service is pushed more and more by clients to explain their decision to grant a loan and it's in this mindset that "Prêt à dépenser" wants to move forward.

"Prêt à dépenser" has then decided to develop an interactive dashboard so that not only their customers service team can explain - in the most transparent way - their decisions to give or not the loans to their clients but also to give full access to their own information.

**DATA OVERVIEW**

Data provided for the project consists of 8 csv files containing personal and banking information about customers who previously applied for a loan. 307511 applications were recorded and files are all linked through the relationship diagram below.

****

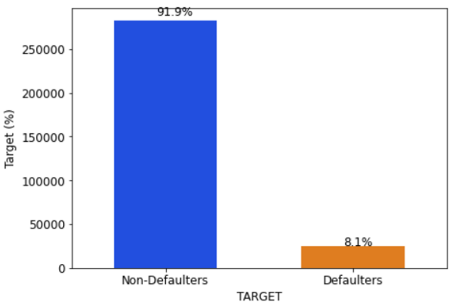
**DATA PREPROCESSING**

After an exploratory data analysis based on an existing Kaggle kernel\*, data preprocessing consisted in different phases:

* **Dataset cleaning**: detecting and correcting (or removing) corrupt or inaccurate records, imputation of missing values…
* **Feature engineering**: creation of new features from the existing data that could help to gain more insight into the data, adding min, mean, max….
* **Encoding** categorical values: Label encoder, One hot encoder…
* **Aggregation** of all files to get a single file ready for modeling. After this step, we obtained a dataframe with 633 features which could affect the performance & running time of our model.
* **Feature selection** which is a technique where we choose features from our data that contribute most to the target variable. We used BorutaPy for this selection to get a final dataframe with 87 features.

\*(<https://github.com/rishabhrao1997/Home-Credit-Default-Risk>)

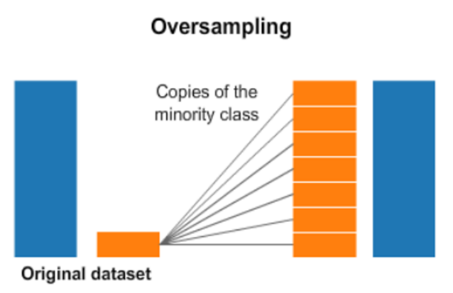
**IMBALANCED CLASSIFICATION**

****

We are here facing an imbalanced classification problem as there is an unequal distribution of classes in the training dataset.

* 92% of the customers managed to refund their loan and are called Non-Defaulters (class or target 0)
* 8% couldn’t refund their loan, they are the Defaulters (class or target 1).

Most machine learning algorithms assume data equally distributed. So when we have a class imbalance, the machine learning classifier tends to be more biased towards the majority class, causing bad classification of the minority class. In order to mitigate this problem and see its impact on the results, we tried few methods:

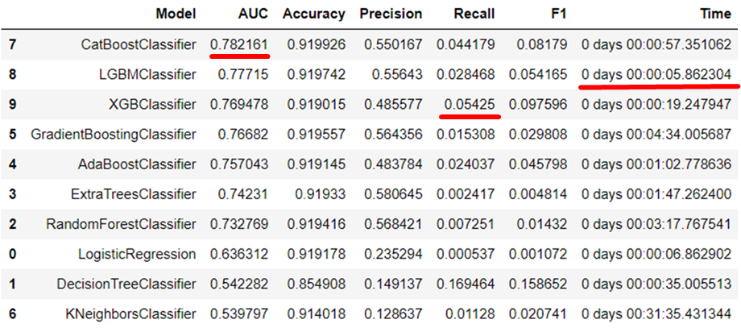
1. **SMOTE** in order to oversample the examples in the minority class. This can be achieved by simply duplicating examples from the minority class in the training dataset prior to fitting a model. This can balance the class distribution. Undersampling or a combination of under/over-sampling are other techniques that can balance the class distribution but these ones have not been investigated in this project.
2. **Class\_weight:** Most of the libraries have an in-built parameter “class\_weight” which helps us optimize the scoring for the minority class. By default, the value of class\_weight=None, i.e. both the classes have been given equal weights. Other than that, we can either give it as ‘balanced’ or we can pass a dictionary that contains manual weights for both the classes. When the class\_weights = ‘balanced’, the model automatically assigns the class weights inversely proportional to their respective frequencies.

**MODELING**

As we don’t want our model to over-learn from training data and perform poorly after being deployed in production, we separated our input data into training & validation sets to prevent our model from overfitting and to evaluate our model effectively:

* Train set: 70% of the data for model training
* Validation set: 30% of the data for model performance validation

**MODEL SELECTION**

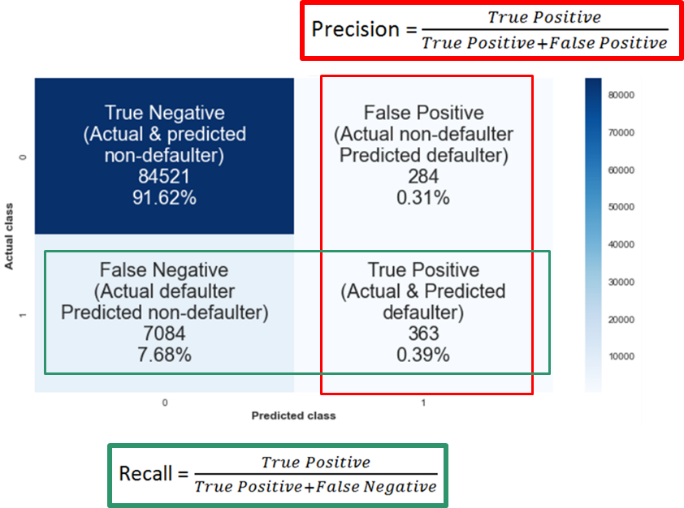
We decided to select different models at first in order to evaluate their performance & running time without any optimization. Based on the results, we kept 3 models:

* CatBoost
* XGBoost
* LightGBM

After additional runs of comparison between those three models, we decided to retain the LGBM Classifier based on the best combination between high metrics and low running time.

**ADAPTED METRICS**

In our problem as to whether customers should receive a loan or not, an adapted metric is highly important. Giving a loan to a bad customer marked as a good customer (False Negative or FN in our case) results in a greater cost to the bank than denying a loan to a good customer marked as a bad customer (False Positive or FP). This requires careful selection of a performance metric that both promotes minimizing misclassification errors in general, and favors minimizing one type of misclassification error over another.

* **Precision** talks about how precise/accurate our model is out of those predicted positive, how many of them are actual positive. Precision is a good measure to determine, when the costs of False Positive are high.
* **Recall** actually calculates how many of the Actual Positives our model capture through labeling it as Positive (True Positive). Recall shall be the model metric we use to select our best model when there is a high cost associated with False Negative which is the case in our problem.
* F1 Score might also be a good measure to use if we need to seek a balance between Precision and Recall AND there is an uneven class distribution (large number of True Negatives).

**COST FUNCTION**

In addition to these metrics, we developed a special cost function J to maximize, with a normalized custom metric penalizing False Negatives to prevent the bank to lose too much money.

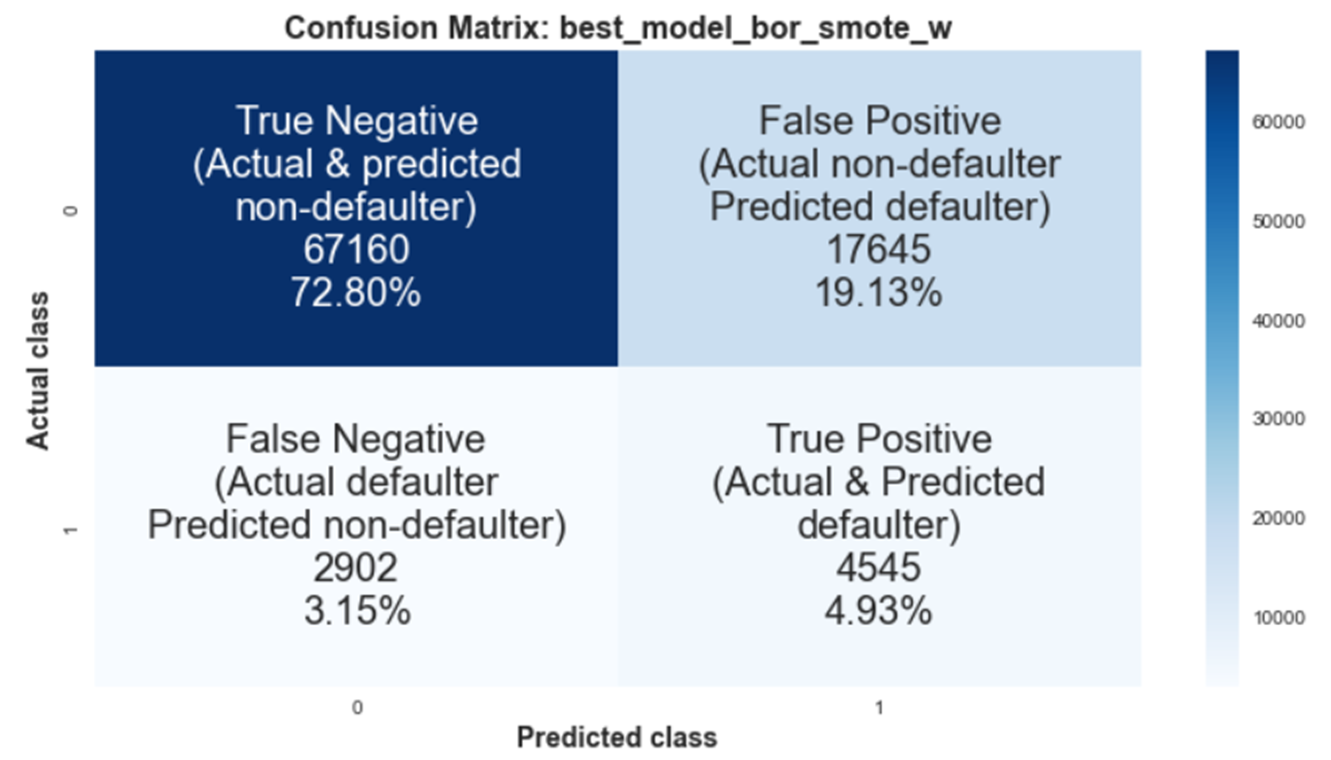
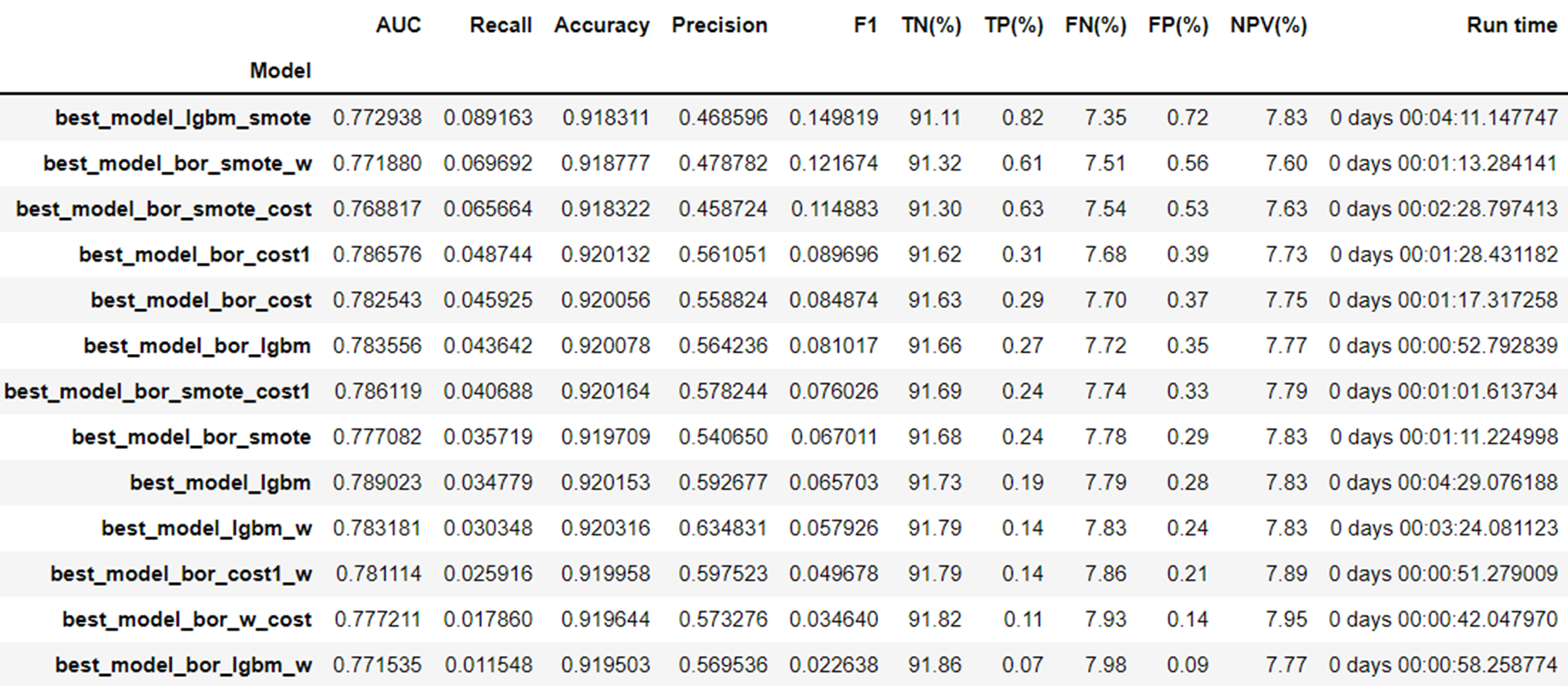
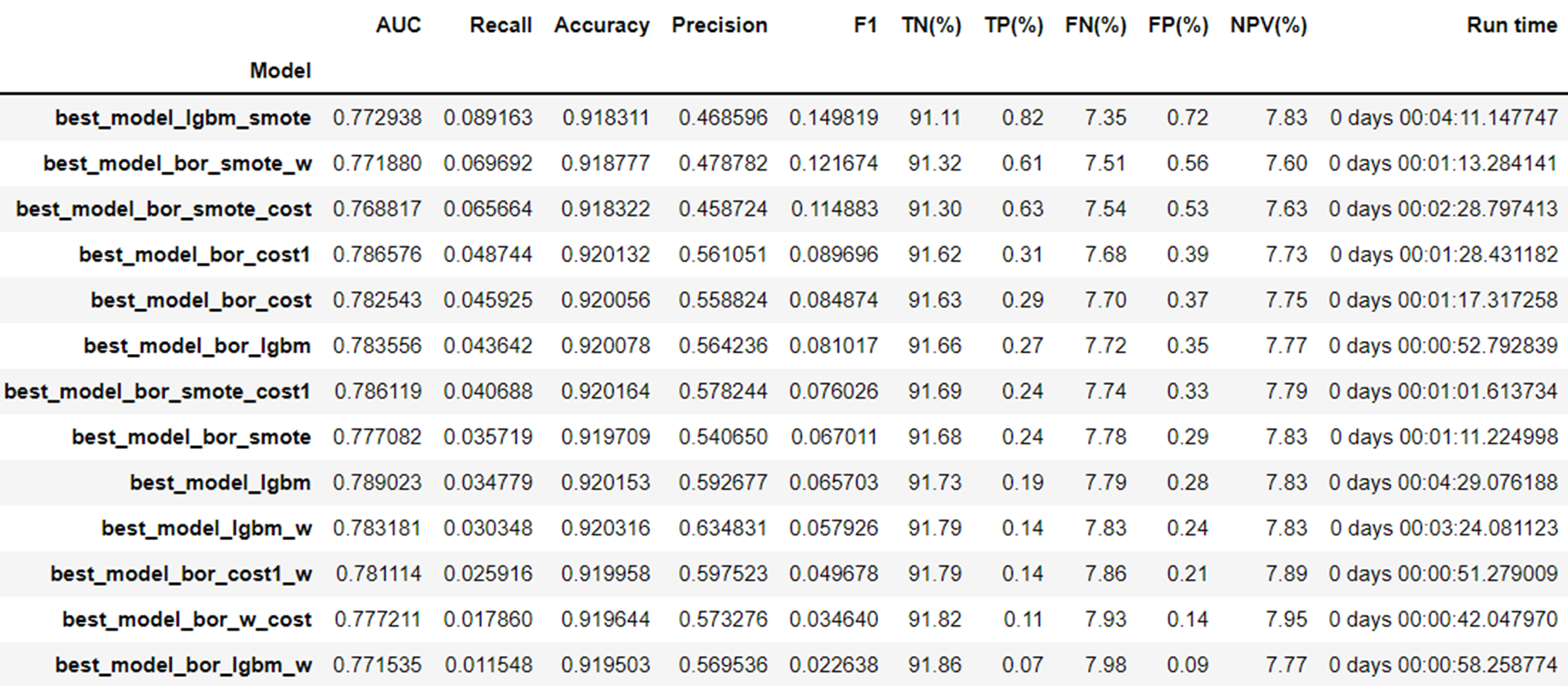
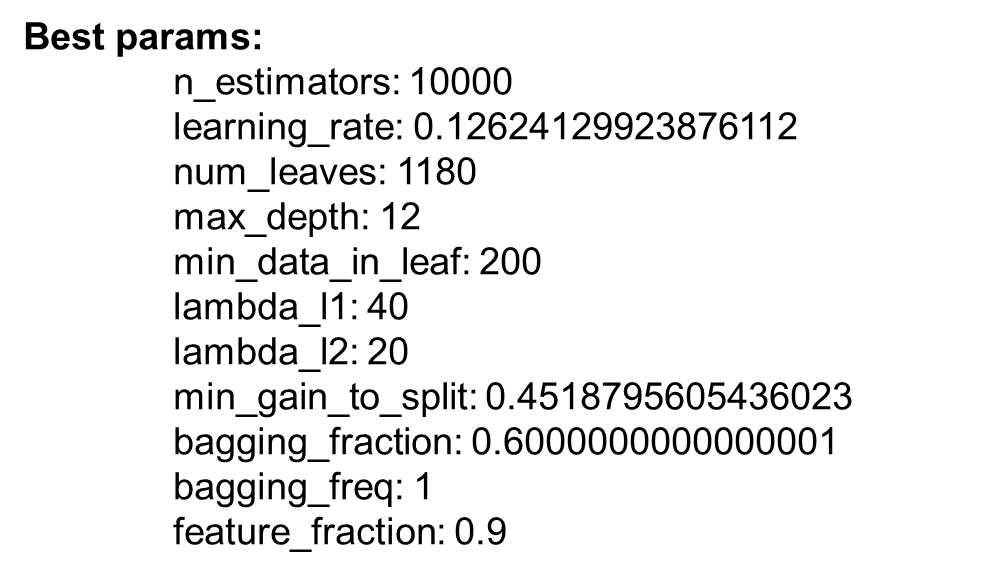
In order to maximize the gains saved by a proper loan approval, some weights are going to be assigned to the actual number of TP, TN, FP and FN.

* TN\_weight: True Negative, loan is paid back, good for the bank (default value=1)
* FP\_weight: False positive, loan is refused while the customer is a non-defaulter, the bank will lose the interests (Type I error) (default value=0), to penalize
* FN\_weight: False negative, loan is granted to a defaulter, the bank will lose lots of money (Type II error), to penalize with a coefficient -10
* TP\_weight: True positive, loan is refused as the customer is a defaulter (default value=1), neutral for the bank, money not lost but not gained either

**Solvency threshold**

Our model returns a score between 0 and 1 and assigns – by default - class 1 Defaulters if the score is above 0.5. A solvency threshold of 0.1 was found i.e. when the customer’s score is above this threshold, he is considered as Defaulter. This metric reduces the number of False Negative and prevent the bank to grant too many loans to actual defaulters.

Considering all of the above and after hyper-parameters optimization through Optuna, our best model was picked as being the LightGBM Classifier on our features selected dataset with class\_weight=’balanced’ and after oversampling with SMOTE. It returns the highest F1 score, the highest Recall i.e. the lowest False Negative and a running time considerably smaller than with our full features model.

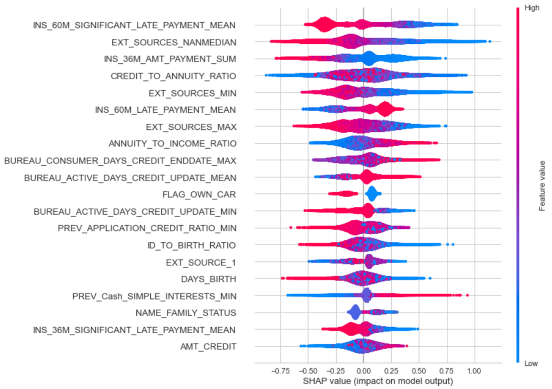


**MODEL INTERPRETABILITY**

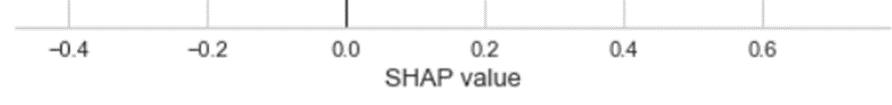
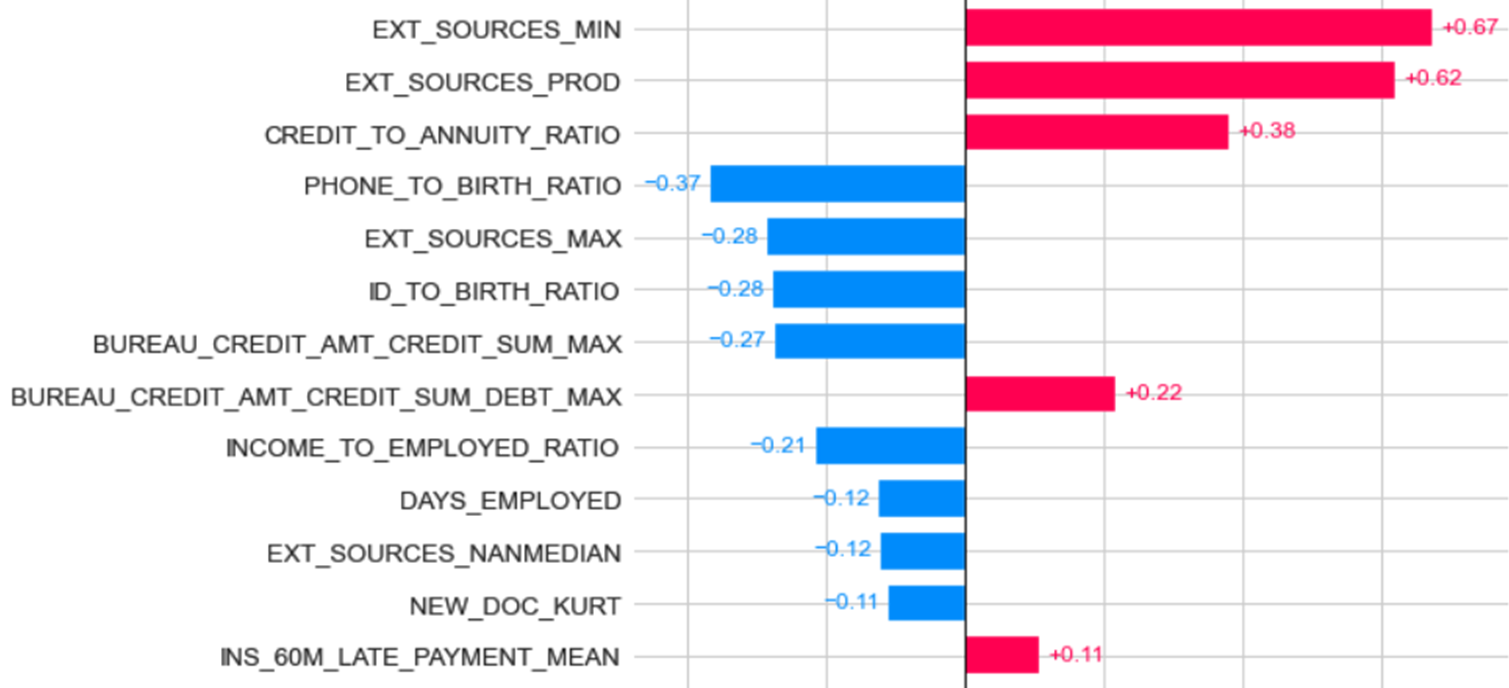
**Feature importance** refers to techniques that calculate a score for all the input features for a given model — the scores simply represent the “importance” of each feature. A higher score means that the specific feature will have a larger effect on the model that is being used to predict a certain variable.

As seen on the graph above, the features that have the most impact on our model are:

* CREDIT\_TO\_ANNUITY\_RATIO, ANNUITY\_TO\_INCOME\_RATIO… banking information
* ID\_TO\_BIRTH\_RATIO, DAYS\_BIRTH… personal information
* EXT\_SOURCES\_MIN, EXT\_SOURCES\_MAX… external data

**Global interpretability** — the collective SHAP values can show how much each predictor contributes, either positively or negatively, to the target variable. This is like the variable importance plot but it is able to show the positive or negative relationship for each variable with the target.

**Local interpretability** — each observation gets its own set of SHAP values. This greatly increases its transparency. We can explain why a case receives its prediction and the contributions of the predictors. Traditional variable importance algorithms only show the results across the entire population but not on each individual case. The local interpretability enables us to pinpoint and contrast the impacts of the factors.



SHAP values estimate the impact of a feature on *predictions* whereas feature importance estimates the impact of a feature on *model fit*.

**LIMITS & IMPROVEMENTS**

* Due to memory problems, we didn’t investigate fully all the available models. Performance enhancement could be looked further with CatBoost for example which returned the lowest False Negative with no optimization. Its running time was considerably higher so we focused only on LGBM.
* By using a metric more adapted to the business, we managed to reduce the False Negative to prevent the bank to lose big money. This metric can be parametrized and a sensitivity analysis could be performed after an agreement about the acceptable ratio of FN & FP.
* Class distribution/over-sampling with SMOTE has been carried out. Methods like BorderlineSMOTE or ADASYN might bring different results. Under-sampling and a combination of under/over-sampling could have also been studied.
* Feature selection was performed with BorutaPy. We tried Recursive feature elimination with cross-validation (RFECV) – with no success - to compare both methods. We ran into constant memory errors.
* We understand that our dashboard is quite basic and could have been enhanced through the addition of multiple comparison features. However, we spent quite a bit of time trying to link FastAPI to Streamlit. As per 2020, Streamlit is a standalone app which does not require FastAPI or Flask anymore. It can be deployed automatically through share\_Streamlit. For the sake of the project, we got into FastAPI.

The key here is that not only humans, but also other computer applications may want to access our ML predictions. Exposing an API endpoint can also be handy to be able to reach and test our model after deployment to make sure that it's behaving as expected. As of today, Streamlit doesn't yet support HTTP requests.

API URL: <https://h7o-fastapi-heroku.herokuapp.com/docs>

Dashboard URL: <https://h7o-streamlit-heroku.herokuapp.com/>