University of Colorado Boulder
Introduction to ML: Supervised Learning
DTSA 5509

Credit Scoring Model

Stanislav Liashkov

Introduction

This project focuses on building a ML model that predicts the probability of a customer to not return a loan. This kind of problem is quite common since all of the banks face certain needs in credit scoring to be able to manage risks using data-driven tools. Apart from building a model itself, we need to do EDA, observe data to find interesting relations and come up with a few hypotheses that might help us later, when we build the model. Home Credit strives to broaden financial inclusion for the unbanked population by providing a positive and safe borrowing experience. In order to make sure this underserved population has a positive loan experience, Home Credit makes use of a variety of alternative data--including telco and transactional information--to predict their clients' repayment abilities. While Home Credit is currently using various statistical and machine learning methods to make these predictions, they're challenging Kagglers to help them unlock the full potential of their data.

Data Description

The data comes from a Kaggle competition called *Home Credit Default Risk*.(competition). We have **8 tables** of data, **2.68 GB** in total.

Tables:

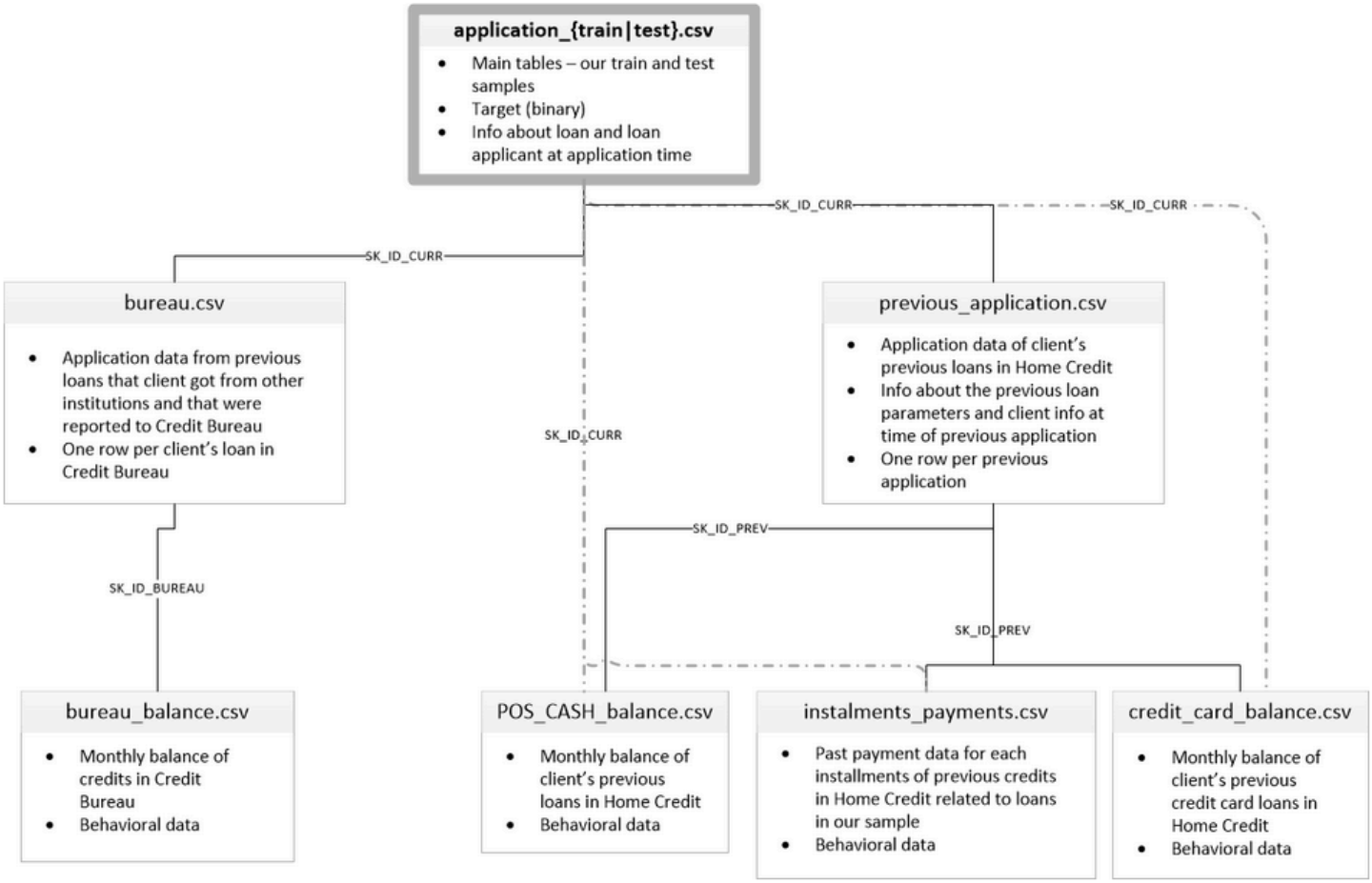
- **application_{train|test}.csv**
  This is the main table, broken into two files for Train (with TARGET) and Test (without TARGET). Static data for all applications. One row represents one loan in our data sample.

- **Bureau.csv**

  All client's previous credits provided by other financial institutions that were reported to Credit Bureau (for clients who have a loan in our sample).For every loan in our sample, there are as many rows as number of credits the client had in Credit Bureau before the application date.

- **bureau_balance.csv**
  - Monthly balances of previous credits in Credit Bureau.
  - This table has one row for each month of history of every previous credit reported to Credit Bureau – i.e the table has (#loans in sample * # of relative previous credits * # of months where we have some history observable for the previous credits) rows.
- **POS_CASH_balance.csv**
  - Monthly balance snapshots of previous POS (point of sales) and cash loans that the applicant had with Home Credit.
  - This table has one row for each month of history of every previous credit in Home Credit (consumer credit and cash loans) related to loans in our sample – i.e. the table has (#loans in sample * # of relative previous credits * # of months in which we have some history observable for the previous credits) rows.
- **credit_card_balance.csv**
  - Monthly balance snapshots of previous credit cards that the applicant has with Home Credit.
  - This table has one row for each month of history of every previous credit in Home Credit (consumer credit and cash loans) related to loans in our sample – i.e. the table has (#loans in sample * # of relative previous credit cards * # of months where we have some history observable for the previous credit card) rows.
- **previous_application.csv**
  - All previous applications for Home Credit loans of clients who have loans in our sample.
  - There is one row for each previous application related to loans in our data sample.
- **installments_payments.csv**
  - Repayment history for the previously disbursed credits in Home Credit related to the loans in our sample.
  - There is a) one row for every payment that was made plus b) one row each for missed payment.
  - One row is equivalent to one payment of one installment OR one installment corresponding to one payment of one previous Home Credit credit related to loans in our sample.

Data Schema


application_{train|test}.csv
Main tables – our train and test samples
Target (binary)
Info about loan and loan applicant at application time
SK_ID_CURR
SK_ID_CURR
SK_ID_CURR
bureau.csv
Application data from previous loans that client got from other institutions and that were reported to Credit Bureau
One row per client's loan in Credit Bureau
previous_application.csv
Application data of client's previous loans in Home Credit
Info about the previous loan parameters and client info at time of previous application
One row per previous application
SK_ID_CURR
SK_ID_PREV
SK_ID_BUREAU
SK_ID_PREV
bureau_balance.csv
Monthly balance of credits in Credit Bureau
Behavioral data
POS_CASH_balance.csv
Monthly balance of client's previous loans in Home Credit
Behavioral data
instalments_payments.csv
Past payment data for each installments of previous credits in Home Credit related to loans in our sample
Behavioral data
credit_card_balance.csv
Monthly balance of client's previous credit card loans in Home Credit
Behavioral data

EDA: brief summary

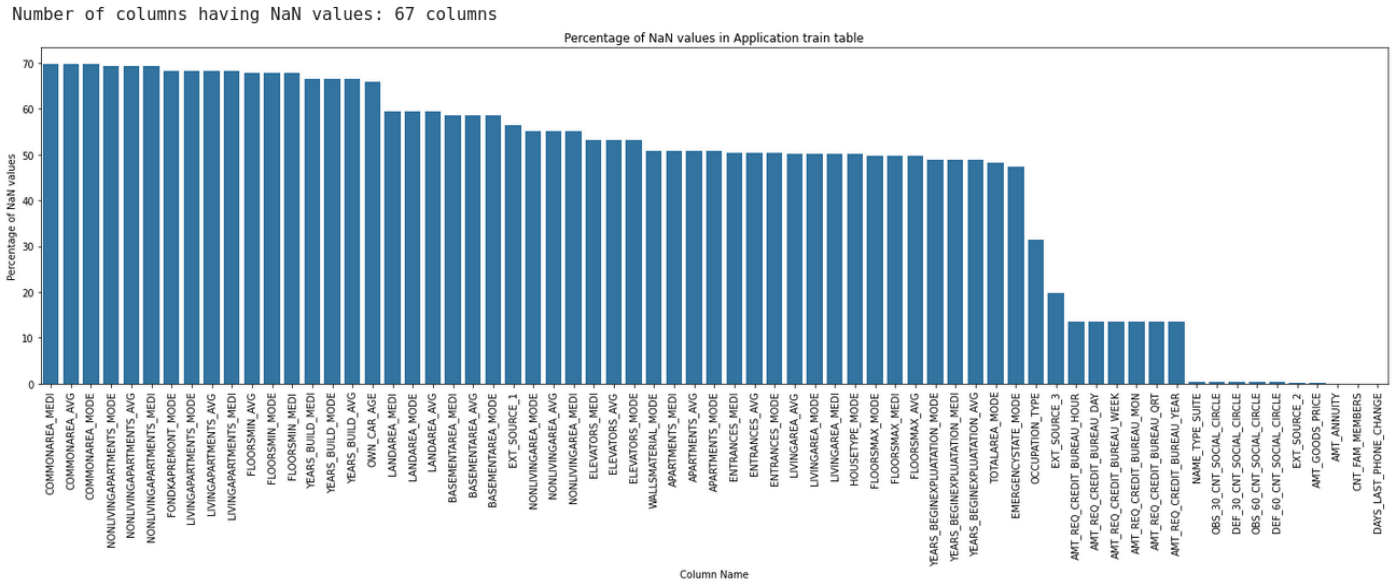
This section is a short summary of EDA provided as jupyter notebook. We will go over main observations and plots. Our exploratory data analysis mostly focuses on the main tables **application_train.csv** as it contains most of the features corresponding to our customers.

```
application_train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 307511 entries, 0 to 307510
Columns: 122 entries, SK_ID_CURR to AMT_REQ_CREDIT_BUREAU_YEAR
dtypes: float64(65), int64(41), object(16)
memory usage: 286.2+ MB
```

Our main table has 122 columns and 307510 rows as records. Total size of the tables is 286 MB.

Missing values



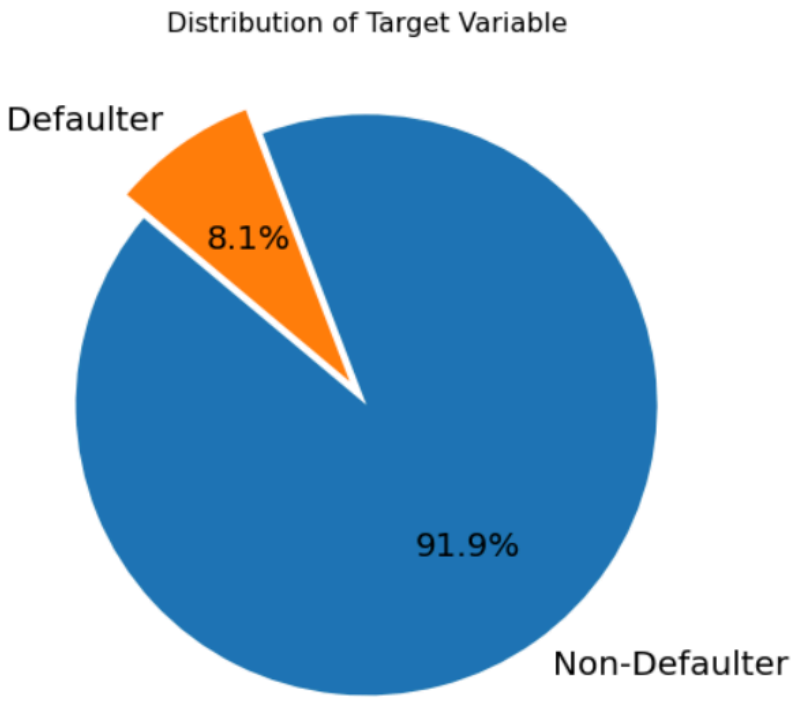
Roughly 40% of features in the table contain NaNs (missing values). Moreover, there are some features that have a very high rate of missing values such as 50-70% of missing values in the data column. A high rate of missing values in some features has to be solved using either imputing these values or dropping the column which is essentially loss of data.

Target variable

In our case, the target variable is called *TARGET* and is included in the main table called **application_train.csv**. This variable is binary where 0 means that a customer paid the loan back and 1 means that a customer turned out to be a defaulter.

Our target variable is quite unbalanced and the fraction of defaulters is much smaller than non-defaulters.

Here, we can see the distribution of a target variable in our table.



The fact that our target variable is highly unbalanced tells us that we will need to use appropriate techniques and evaluation metrics for our model. It is clearly not a good idea to use such metrics as accuracy since we are actually interested in detecting potential defaulters. Because of our target being unbalanced, we can achieve accuracy of 92% by simply marking everyone as non-defaulter which sounds silly and doesn't make sense. We will also need to focus on algorithms that are more robust to unbalanced data.

Features correlated with Target

After computing correlation of features and target variable, we can provide a list of most correlated features. Such a list might be useful in further observation steps since it tells us where to look at and it is also useful in feature engineering because we can use correlation info as a hint which features to combine.

Here, I provide a list of top 20 highly correlated features with *TARGET*

10 features with positive correlation

```
DAYS_BIRTH                    0.078239
REGION_RATING_CLIENT_W_CITY   0.060893
REGION_RATING_CLIENT          0.058899
DAYS_LAST_PHONE_CHANGE        0.055218
DAYS_ID_PUBLISH               0.051457
REG_CITY_NOT_WORK_CITY        0.050994
FLAG_EMP_PHONE                0.045982
REG_CITY_NOT_LIVE_CITY        0.044395
FLAG_DOCUMENT_3               0.044346
DAYS_REGISTRATION             0.041975
Name: TARGET, dtype: float64
```

10 features with negative correlation

```
EXT_SOURCE_3                 -0.178919
EXT_SOURCE_2                 -0.160472
EXT_SOURCE_1                 -0.155317
DAYS_EMPLOYED                -0.044932
FLOORSMAX_AVG                -0.044003
FLOORSMAX_MEDI               -0.043768
FLOORSMAX_MODE               -0.043226
AMT_GOODS_PRICE              -0.039645
REGION_POPULATION_RELATIVE   -0.037227
ELEVATORS_AVG                -0.034199
Name: TARGET, dtype: float64
```

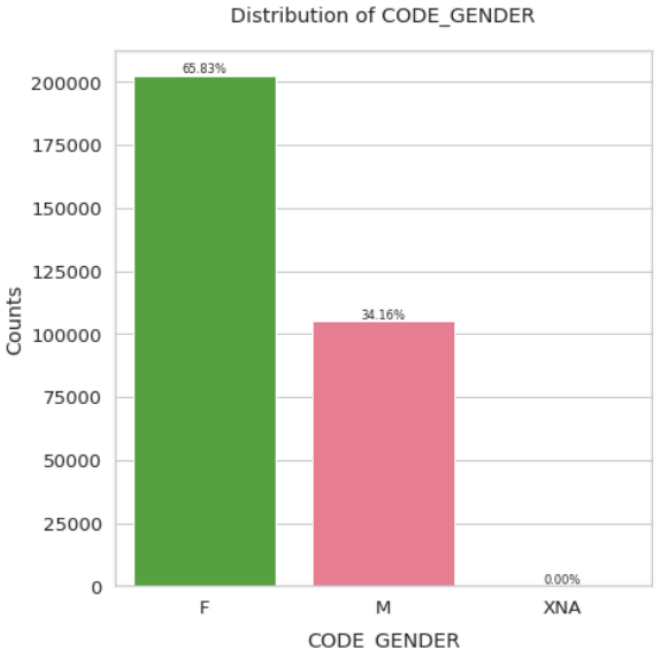
As an important observation, I would like to point out that the most influential and informative features according to correlation are *EXT_SOURCE1, EXT_SOURCE2, EXT_SOURCE3.* These features are actually credit scoring (prediction of default) that come from external scoring sources. It is probably a good sign since we can see at least some evidence that those external sources are generally helpful. Another informative feature might be *DAYS_BIRTH* that corresponds to the age of customer (in days) and such columns as REGION_RATING_CLIENT and REGION_RATING_CLIENT_W_CITY that both correspond to assigned score of the region where client is living. Correlations info is a good hint, although correlation doesn't catch non-linear relations and doesn't imply any causation either, so we should be careful interpreting correlations.

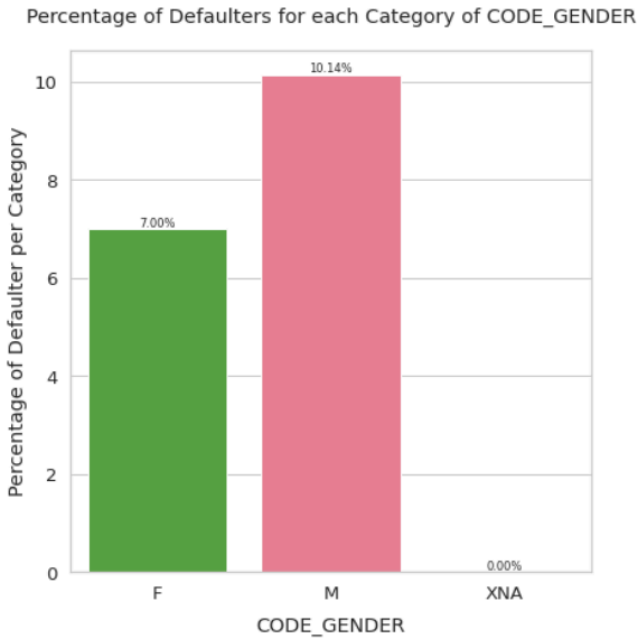
Categorical features

This section contains most interesting plots and observations extracted from categorical features of the main table. Such plots might tell us more information and insights about our customers and how different features may affect the probability of default.

CODE_GENDER:

Gender of the client

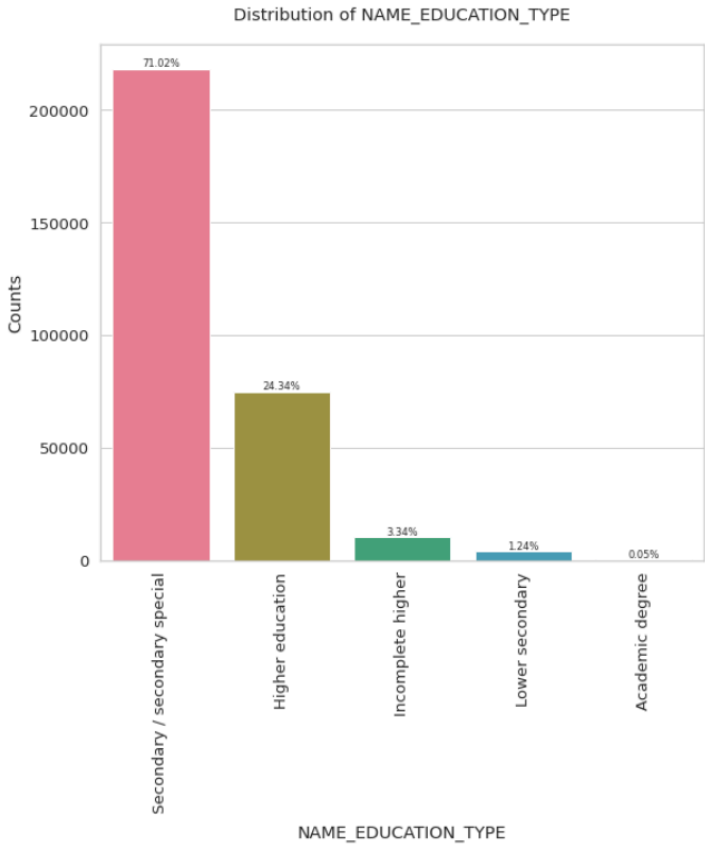


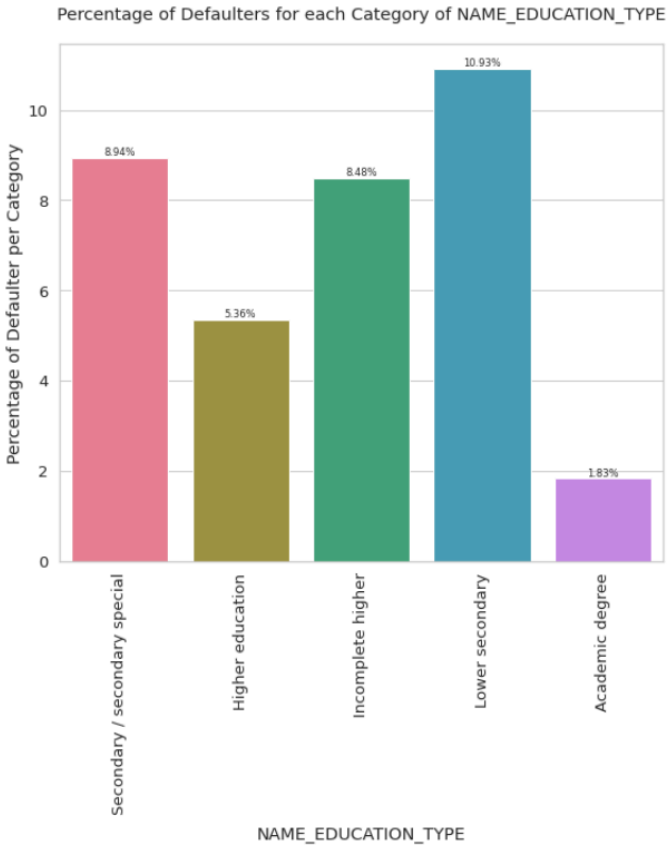


Our dataset is unbalanced in terms of gender since we have more female clients (66% vs 34%). There is also a noticeable difference in fraction of defaulters between men and women. Women tend to return loans more often (7% defaulters of women vs 10% of defaulters among men).

NAME_EDUCATION_TYPE:

Highest level of education achieved by a client

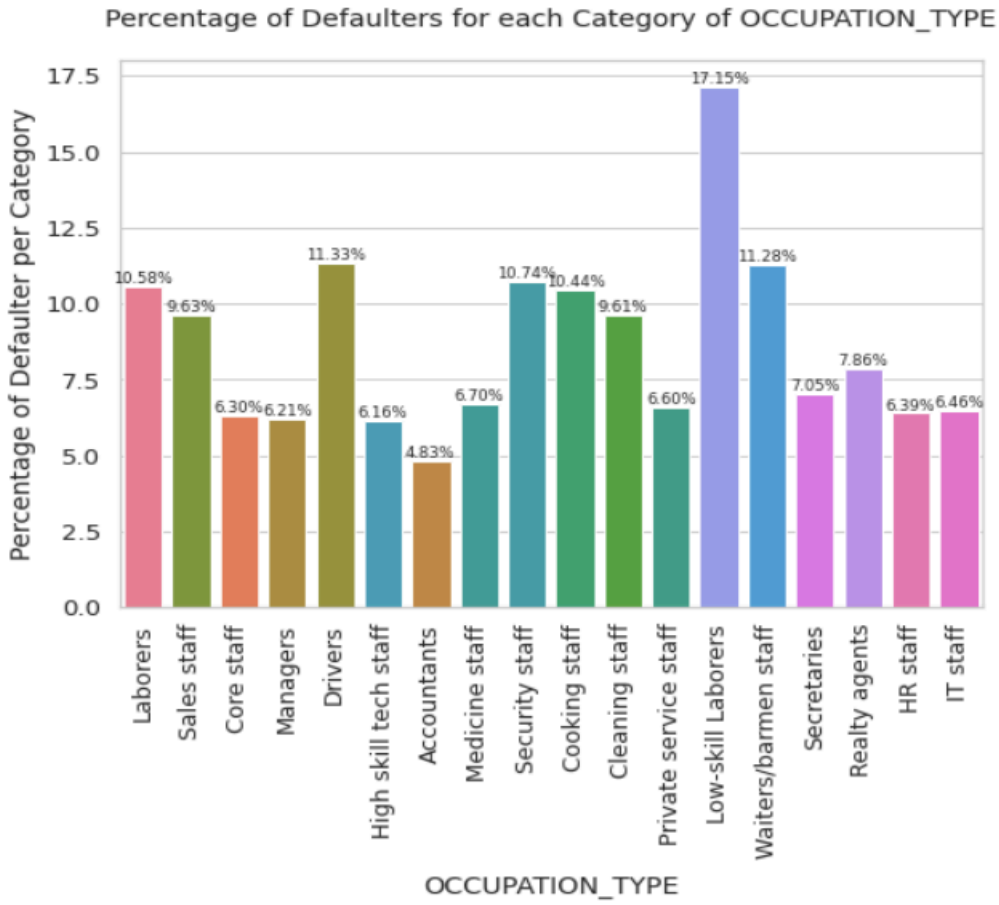




This feature at first glance looks like a very good predictor for default risk since we see that fraction of defaulters increases as we move to groups of people with lower and lower levels of education. At least by visualization, we can claim that the higher level of education our client has the more certainty we assign to his return. This interesting relation may look quite expected since we mostly believe that higher level of education open up new opportunities and therefore, more stable financial situation. To sum up, this feature looks capable to provide good splits into groups with different risk of default.

OCCUPATION_TYPE

What kind of occupation does the client have

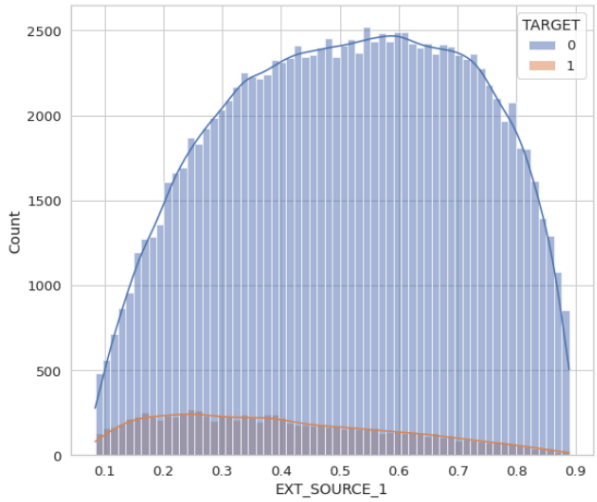


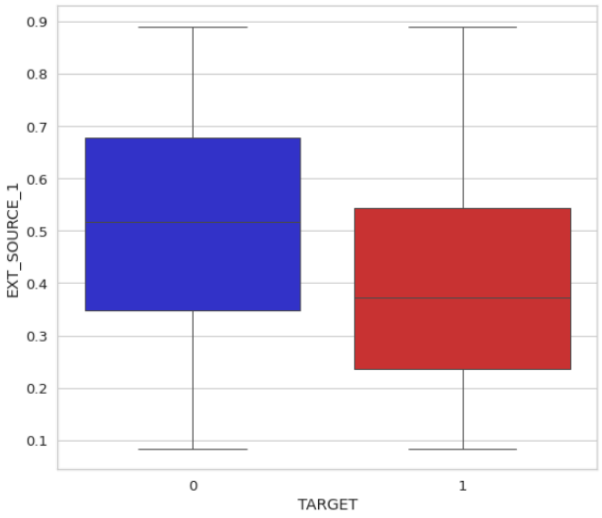
I should say that this feature also looks quite useful and interesting since we can see how default rate varies across different types of occupation. Workers in low-skill positions tend to be less educated and financially successful in general which leads to higher risk of something going wrong in their life and potential default. So, such a distribution makes sense for me. As a note, I can emphasize that *low-skill staff and waiters* categories have the highest fraction of defaulters whereas *high-tech staff, accountants and managers* tend to return money on a regular basis. I also suppose that this feature should be in strong correlation with type of education of a customer. Overall, occupation type seems to be an informative column that tells us how risk varies among different jobs.

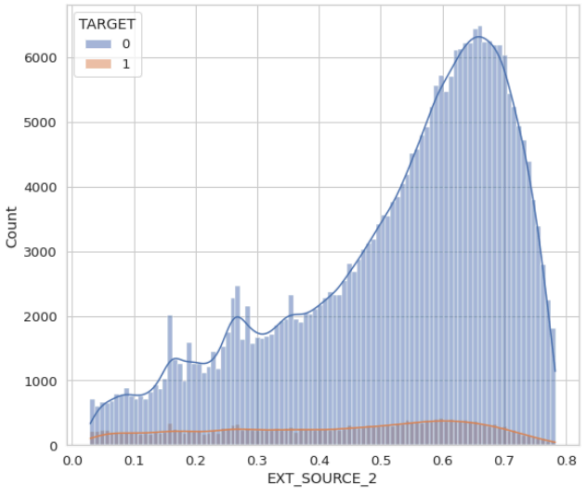
Numerical features

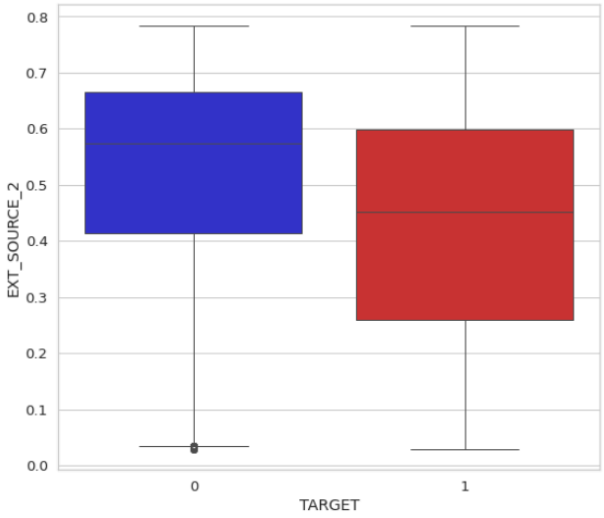
EXT_SOURCE1, EXT_SOURCE2, EXT_SOURCE3

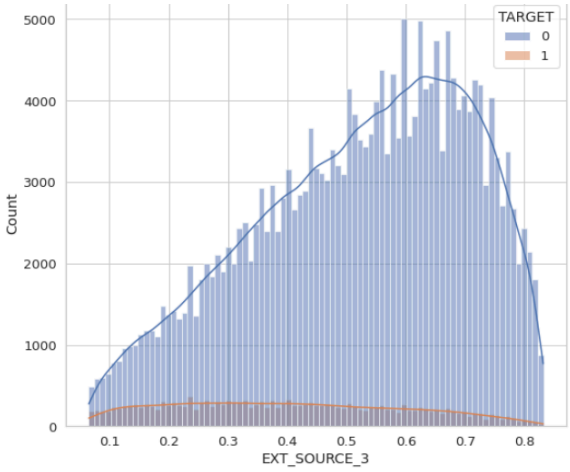
Scores from external sources

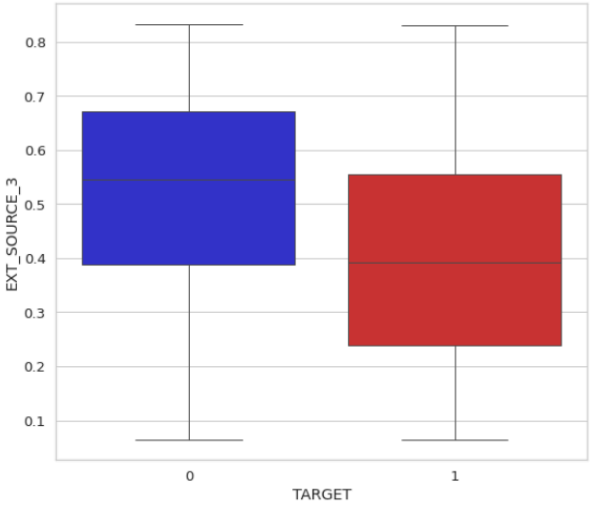




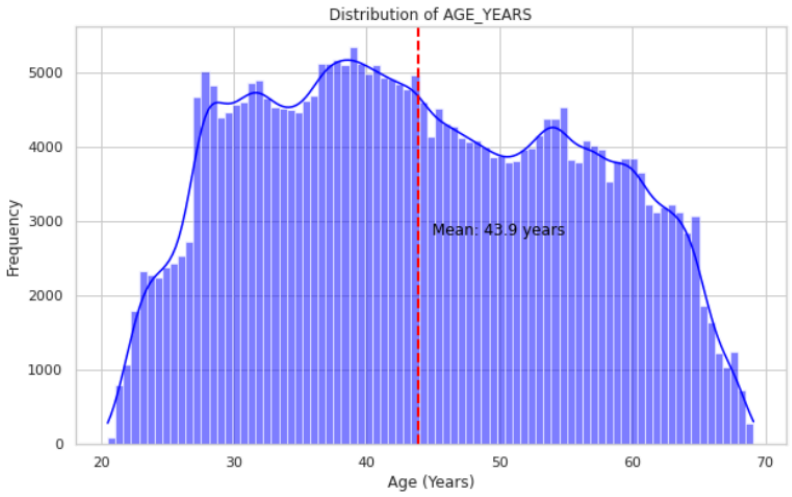






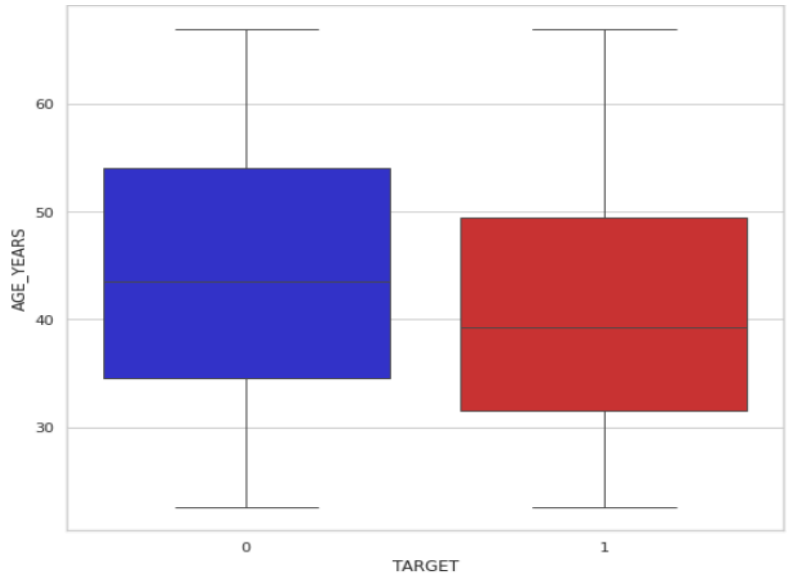


Age of clients



Distribution of age is almost symmetric with heavy tails and slightly skewed to the left. The average age of our customer - 44 years old.

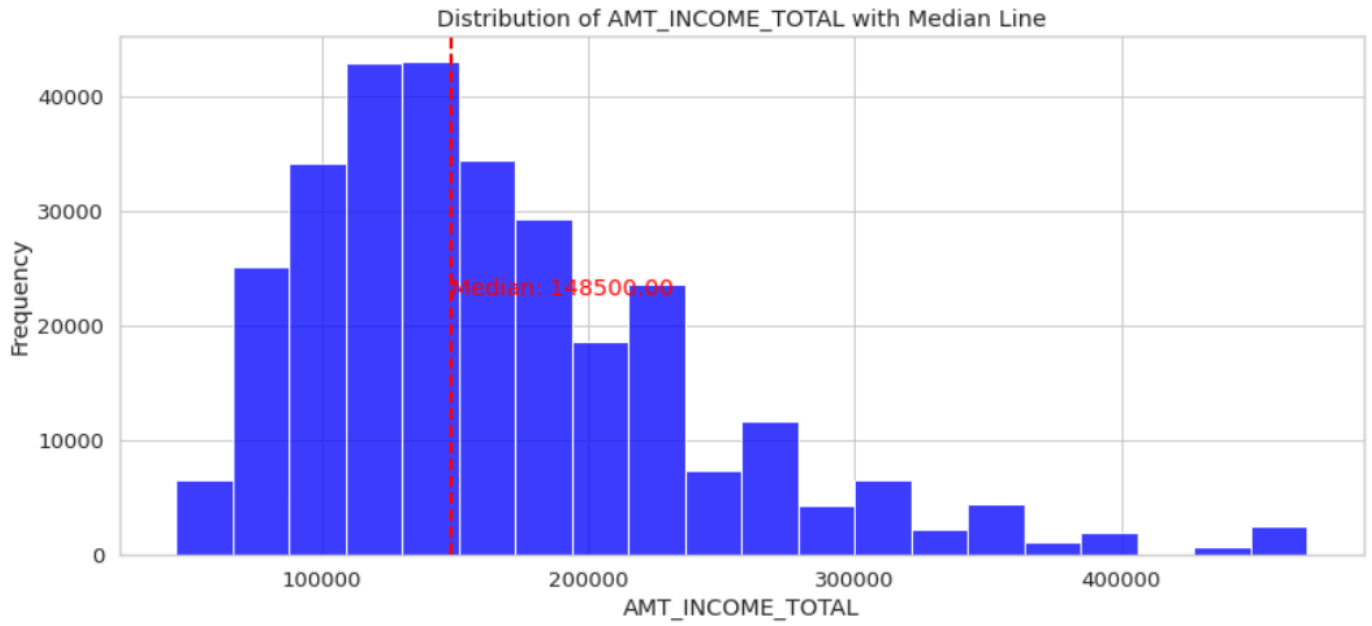
Defaulters age vs Non-defaulters age barplot



Generally, those who didn't return the loan are slightly younger, we can see this from the barplot above.

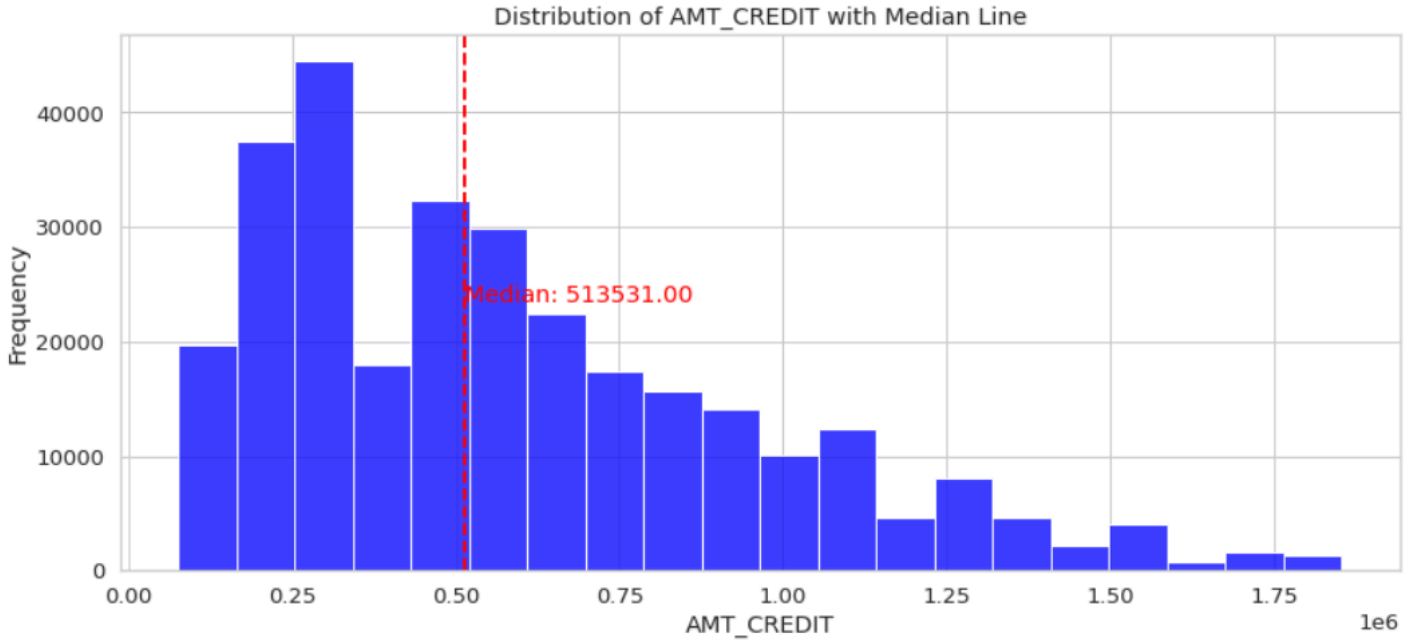
AMT_INCOME_TOTAL

Annual income of the client (currency not specified in dataset)



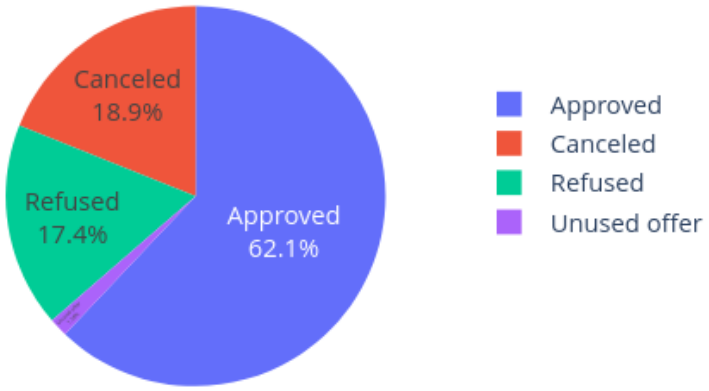
AMT_CREDIT

Credit amount of the loan (currency not specified in dataset)



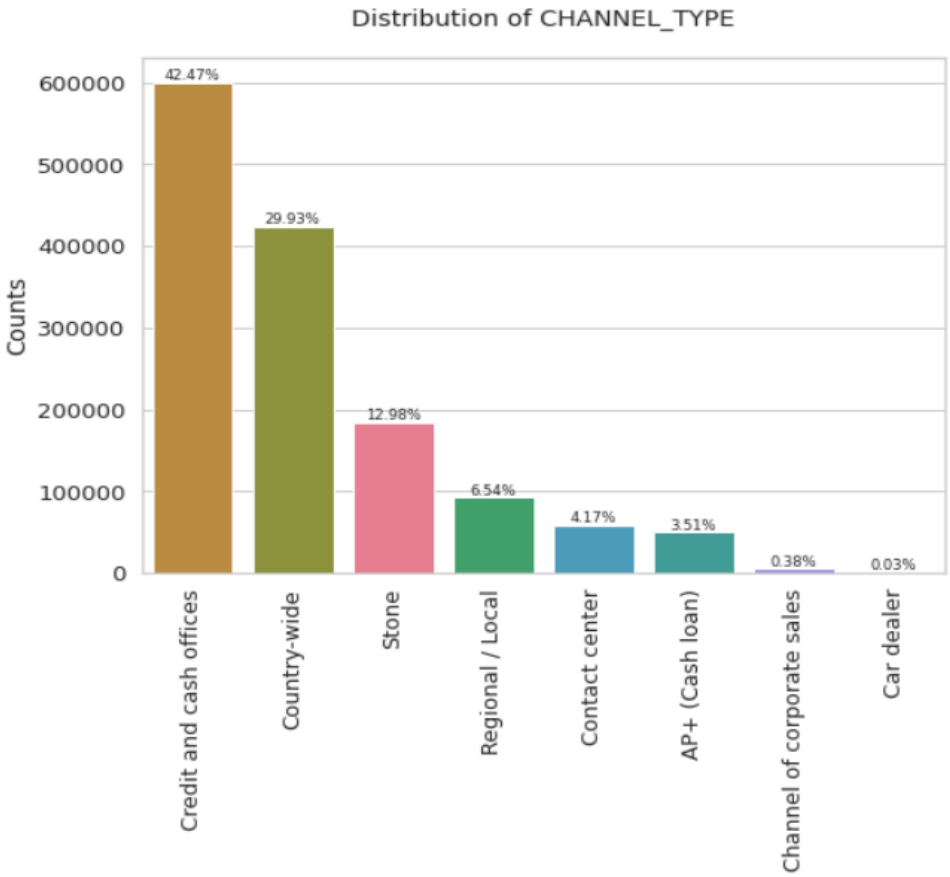
NAME_CONTRACT_STATUS (previous_application.csv)

Result of previous application



CHANNEL_TYPE (previous_application.csv)

Which way the client discovered our bank



Summary on EDA

1) Firstly, it is essential to address the numerous missing values in our dataset. Simply dropping rows with NaN would be overly destructive, resulting in the loss of a substantial fraction of data. A more reasonable approach is to impute values using two distinct strategies for categorical and numerical features. For categorical features, I propose filling missing values with a new category labeled MISSING, as there may be underlying reasons why a specific category is unspecified for a client. Regarding numerical data, my preference is to impute with the median.

2) The **TARGET** variable exhibits a significant imbalance, with **92%** of non-defaulters and **8%** of defaulters. Consequently, employing appropriate techniques for handling unbalanced data becomes imperative.

3) Upon examining the correlation matrix with TARGET, it appears that features with the highest correlation with the target variable might possess substantial predictive and explanatory power. Notable features include *DAYS_BIRTH, REGION_RATING_CLIENT_W_CITY, REGION_RATING_CLIENT, DAYS_LAST_PHONE_CHANGE, DAYS_ID_PUBLISH, EXT_SOURCE_3, EXT_SOURCE_2, EXT_SOURCE_1, DAYS_EMPLOYED, FLOORSMAX_AVG, FLOORSMAX_MEDI, FLOORSMAX_MODE, and AMT_GOODS_PRICE.*

4) We explored the distributions of select variables, stratified by TARGET values. This exploration yielded insightful observations and provided data-driven answers to questions such as the likelihood of loan repayment based on gender, the impact of education levels on default risk, and more (refer to comments below the plots for specific insights).

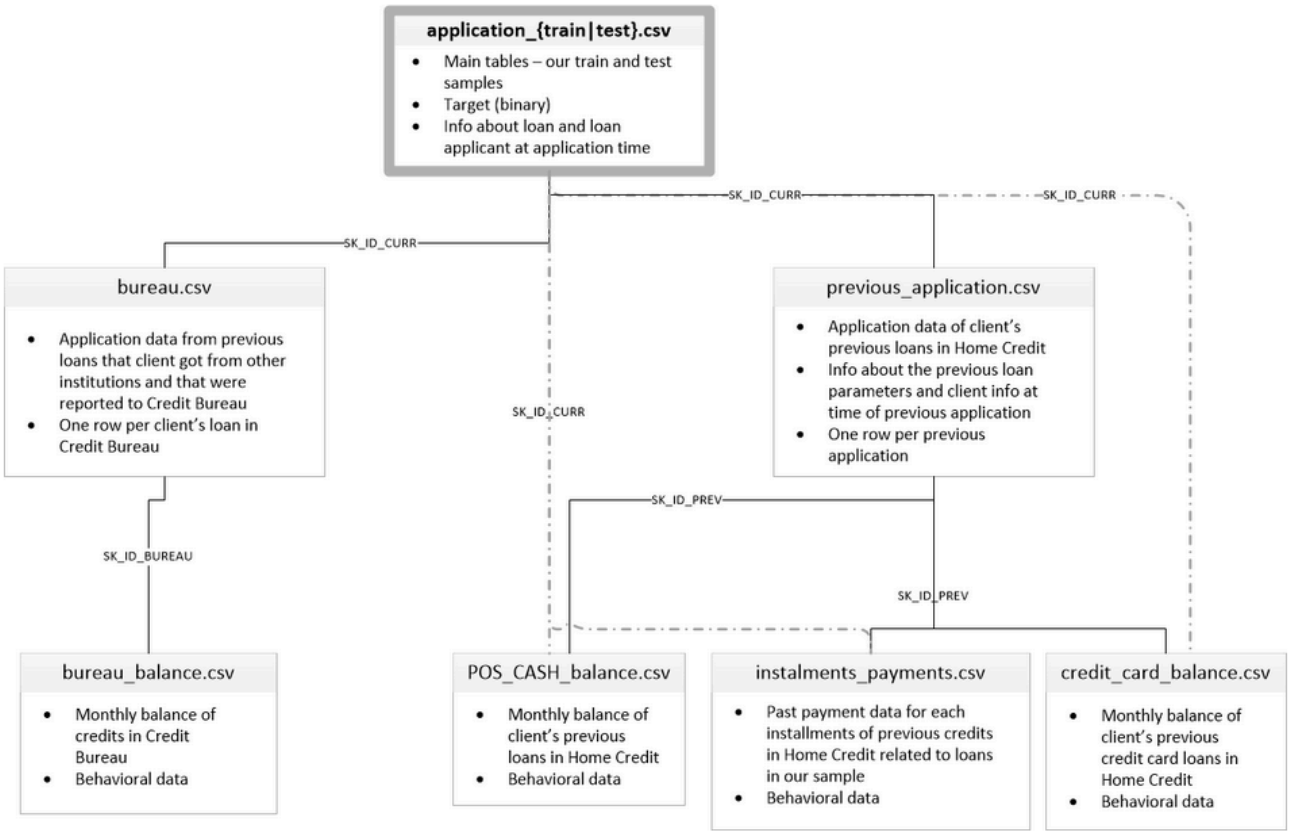
5) Additionally, it's worth noting that our insights were derived from visualizations, the correlation matrix, and descriptive statistics. However, our focus was primarily on the two largest tables out of six, as these tables contain the majority of our features. Merging all tables together is necessary to analyze data from the remaining tables.

In conclusion, we are ready to proceed with merging our data, consolidating it into a single comprehensive table that incorporates all the necessary information.

Data processing

Merging tables

As a first step in data processing, we need to merge our tables together. Before merging, we will need to aggregate additional tables by SK_ID_CURR which is the id of our current application in the main table. Merging large tables and computing aggregates is generally computationally expensive tasks. All our data in total is about **2.7 GB** and it is quite big for a personal laptop. I tried to deal with this dataset with usual tools such as *pandas*, unfortunately I faced kernels dying and a process being shut down many times because of too high load in terms of computation. OS just kills the process if it consumes too many resources and that was my case. I decided to look for a tool that is able to manage resources in a more efficient way in terms of both speed of operations and bounded consumption of resources.



Polars: High-performance Data processing

Eventually, I found a great library that was designed and highly optimized for processing large datasets on a single machine without crashes and with reasonable speed of operations which is surprisingly fast.

Polars is a high-performance DataFrame library, designed to provide fast and efficient data processing capabilities. Inspired by the reigning pandas library, Polars takes things to another level, offering a seamless experience for working with large datasets that might not fit into memory.

Key features

1. **Speed and performance:** Polars prioritizes speed through parallel processing and memory optimization, enabling faster processing of large datasets compared to traditional methods.

2. **Data manipulation capabilities**: Offering a robust toolkit, Polars covers essential data manipulation operations, including filtering, sorting, grouping, joining, and aggregating, addressing approximately 80% of common tasks in comparison to Pandas.

3. **Expressive syntax:** With an expressive and intuitive syntax reminiscent of pandas, Polars ensures easy adoption and utilization, leveraging existing Python knowledge for efficient data processing.

4. **DataFrame and series structures:** Built around DataFrame and Series structures, Polars provides a powerful and familiar abstraction for working with tabular data, allowing chained operations for concise and efficient transformations.

5. **Lazy evaluation:** Unique to Polars, support for lazy evaluation optimizes queries for performance and memory efficiency, distinguishing it from Pandas, which relies solely on eager evaluation.

How to merge tables

In our data analysis workflow, the fundamental approach revolves around a central table, namely *`application_train.csv`*, which serves as the core dataset containing primary information about client applications. However, to gain a comprehensive understanding of each client's profile, we leverage multiple auxiliary tables containing additional details such as credit card transactions history, previous applications, and information from the bureau.

Our objective is to compute various aggregates, including MEAN, MIN, MAX, and STD for numerical features, as well as MODE for categorical features. The key step involves grouping these tables by the unique identifier `SK_ID_CURR`, which represents the current application's ID in the main table.

To consolidate information effectively, we employ a LEFT JOIN operation when merging the additional tables with the main table. The choice of a LEFT JOIN is deliberate, driven by the need to preserve all records in the main table while accommodating cases where certain information might be missing in the supplementary tables for specific clients. This ensures that our analyses encompass the entire dataset, avoiding any loss of valuable data from the main table and allowing for a comprehensive examination, even in instances where details from the auxiliary tables may be absent for certain clients.

Example of using polars for grouping data

```
grouped_cred_card_df = lazy_cred_card_balance_df.group_by("SK_ID_PREV").agg([
    pl.col('MONTHS_BALANCE').mean().alias('MONTHS_BALANCE_mean'),
    pl.col('AMT_BALANCE').mean().alias('AMT_BALANCE_mean'),
    pl.col('AMT_CREDIT_LIMIT_ACTUAL').mean().alias('AMT_CREDIT_LIMIT_ACTUAL_mean'),
    pl.col('AMT_DRAWINGS_ATM_CURRENT').mean().alias('AMT_DRAWINGS_ATM_CURRENT_mean'),
    pl.col('AMT_DRAWINGS_CURRENT').mean().alias('AMT_DRAWINGS_CURRENT_mean'),
    pl.col('AMT_DRAWINGS_OTHER_CURRENT').mean().alias('AMT_DRAWINGS_OTHER_CURRENT_mean'),
    pl.col('AMT_DRAWINGS_POS_CURRENT').mean().alias('AMT_DRAWINGS_POS_CURRENT_mean'),
    pl.col('AMT_INST_MIN_REGULARITY').mean().alias('AMT_INST_MIN_REGULARITY_mean'),
    pl.col('AMT_PAYMENT_CURRENT').mean().alias('AMT_PAYMENT_CURRENT_mean'),
    pl.col('AMT_PAYMENT_TOTAL_CURRENT').mean().alias('AMT_PAYMENT_TOTAL_CURRENT_mean'),
    pl.col('AMT_RECEIVABLE_PRINCIPAL').mean().alias('AMT_RECEIVABLE_PRINCIPAL_mean'),
    pl.col('AMT_RECIVABLE').mean().alias('AMT_RECIVABLE_mean'),
    pl.col('AMT_TOTAL_RECEIVABLE').mean().alias('AMT_TOTAL_RECEIVABLE_mean'),
    pl.col('CNT_DRAWINGS_ATM_CURRENT').mean().alias('CNT_DRAWINGS_ATM_CURRENT_mean'),
    pl.col('CNT_DRAWINGS_CURRENT').mean().alias('CNT_DRAWINGS_CURRENT_mean'),
    pl.col('CNT_DRAWINGS_OTHER_CURRENT').mean().alias('CNT_DRAWINGS_OTHER_CURRENT_mean'),
    pl.col('CNT_DRAWINGS_POS_CURRENT').mean().alias('CNT_DRAWINGS_POS_CURRENT_mean'),
    pl.col('CNT_INSTALMENT_MATURE_CUM').mean().alias('CNT_INSTALMENT_MATURE_CUM_mean'),
    pl.col('SK_DPD').mean().alias('SK_DPD_mean'),
    pl.col('SK_DPD_DEF').mean().alias('SK_DPD_DEF_mean'),
    pl.col('NAME_CONTRACT_STATUS').mode().first().alias('NAME_CONTRACT_STATUS_mode')
])
```

This piece of code will build and optimize the execution graph that gives us a grouped data as a result. To retrieve the result, we need to call method *.collect()* to actually execute these steps.

```
grouped_cred_card_df = grouped_cred_card_df.collect()
```

```
grouped_cred_card_df.write_csv("processsing/grouped_cred_card.csv", float_precision=3)
```

Merged result

We finally ended up with a train dataset that contains *211 columns and 307510 rows of total size - 560 MB.* The same operations have been done for *application_test.csv and* eventually the size of test table is - 78 MB.

Preprocessing for ML

In the process of preparing the dataset for machine learning models, essential preprocessing steps were taken to enhance data quality and usability. Here are the key actions performed:

Dropping Columns with High Missing Values

To streamline the dataset and improve computational efficiency, columns with a substantial number of missing values were identified and removed from both the training and testing datasets. This ensures that the model is not burdened with irrelevant or sparse features that may not significantly contribute to the learning process.

Missing Value Imputation

The `MissingImputer` class was utilized for handling missing values in both categorical and numerical features. Categorical features were imputed with a constant value ("MISSING"), while numerical features were imputed with the median. This approach ensures a balanced treatment of missing values across diverse feature types.

Categorical Feature Encoding

The `CategoricalEncoder` class facilitated the encoding of categorical features using an Ordinal Encoder. This transformation converts categorical variables into numerical representations, enabling machine learning algorithms to effectively interpret and utilize these features during model training.

Saving Preprocessed Data

The final preprocessed datasets, labeled as `processed_train.csv` and `processed_test.csv`, were saved to separate CSV files. This step ensures that the preprocessed data can be seamlessly integrated into machine learning workflows without the need for repetitive preprocessing steps.

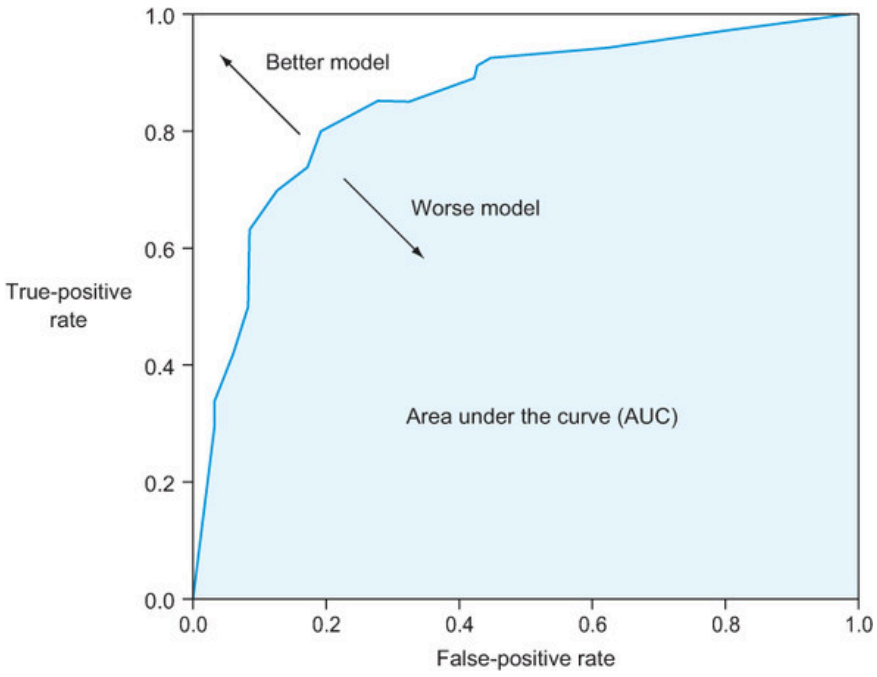
Training model

Metric

In our case, the goal is to train a model that predicts the risk of default for a given customer. As it's been mentioned before, we are dealing with a highly unbalanced dataset and therefore, we need to use appropriate algorithms and metrics that are more robust to unbalanced data.

Given the imbalanced distribution of the target variable, where only **9%** represent defaulters and **91%** non-defaulters, it is imperative to choose an evaluation metric that accounts for this skew. Metrics such as accuracy can be misleading in such scenarios, prompting the consideration of alternatives like **F1-score** and **AUC** (Area under the ROC curve). While both metrics address the balance between precision and recall, I have opted for AUC due to its ability to assess the model's discrimination across various probability thresholds. AUC evaluates the model's performance comprehensively, considering the entire range of classification thresholds and emphasizing the trade-off between sensitivity and specificity. This choice aligns with the objective of identifying potential defaulters while minimizing false-positive predictions, making AUC a robust metric for addressing the challenges posed by imbalanced datasets and ensuring a balanced approach to risk management for the bank.

ROC-AUC metric plot



Feature Engineering

Most of the time, we have many features that can be combined together in some way (product, ration, difference etc) to produce a new useful feature that is potentially more informative than initial features separately. In this project, I tried to make up new features based on my intuition and common sense. Some of the created features turned out to be useful and informative, although intuition doesn't guarantee to help us in making new informative predictors.

New features:

$$\text{EXT\_SOURCE\_AVG} = \frac{\text{EXT\_SOURCE\_1} + \text{EXT\_SOURCE\_2} + \text{EXT\_SOURCE\_3}}{3}$$

$$\text{LABOR\_PERIOD\_RATE} = \frac{\text{DAYS\_EMPLOYED}}{\text{DAYS\_BIRTH} + \epsilon}$$

$$\text{CURR\_VS\_PREV\_GOODS\_PRICE} = \frac{\text{AMT\_GOODS\_PRICE}}{\text{AMT\_GOODS\_PRICE\_mean} + \epsilon}$$

$$\text{CURR\_VS\_PREV\_ANNUITY} = \frac{\text{AMT\_ANNUITY}}{\text{AMT\_ANNUITY\_mean} + \epsilon}$$

$$\text{CONSUMPTION\_RATE} = \frac{\text{AMT\_GOODS\_PRICE}}{\text{AMT\_INCOME\_TOTAL} + \epsilon}$$

$$\text{CURR\_REGISTRATION\_PERIOD} = \frac{\text{DAYS\_REGISTRATION}}{\text{DAYS\_BIRTH} + \epsilon}$$

$$\text{CREDIT\_LOAD} = \frac{\text{AMT\_CREDIT}}{\text{AMT\_INCOME\_TOTAL} + \epsilon}$$

$$\text{CREDIT\_LOAD\_MEAN} = \frac{\text{AMT\_CREDIT\_mean}}{\text{AMT\_INCOME\_TOTAL} + \epsilon}$$

$$\text{DECISION\_ACTION\_TIME} = \text{DAYS\_DECISION\_mean} - \text{DAYS\_ENTRY\_PAYMENT\_mean}$$

$$\text{PAYMENT\_ANNUITY\_RATIO} = \frac{\text{AMT\_PAYMENT\_mean}}{\text{AMT\_ANNUITY\_mean} + \epsilon}$$

Model

Taking into account that we have many features - 177 and quite large dataset, I would like to consider **tree-based ensembles** such as:

- Random Forest
- Catboost
- Xgboost
- LightGBM

These models generally provide the best performance in such problems and most of them can be efficiently trained by utilizing GPU. An additional advantage of tree-based models is the possibility to extract feature importance that can be further used in feature selection and interpreting the results. After I define a set of tools to try, I plan to create a grid of hyperparameters for each of these models and run a hyperparameter search to come up with a good set of parameters. After all iterations of tuning for each model, we can compare performance of models and select the best one.

Hyperparameter tuning

I decided to use the **Optuna** framework for tuning hyperparameters for each model.

Optuna is an open-source hyperparameter optimization framework designed for machine learning. It provides an efficient way to search for the best set of parameters for a given model to improve its performance. Optuna uses *Bayesian Optimization* under the hood. Bayesian optimization in Optuna is a sophisticated approach to hyperparameter tuning, using probabilistic models to guide the search for the best hyperparameters. It's particularly effective for optimizing complex functions where evaluations (like training and validating a machine learning model) are expensive in terms of time and computational resources. We need to define an objective function that we want to optimize. In our case, we are interested in maximizing *AUC* on cross-validation.

Example of using Optuna for hyperparameter search of Random Forest

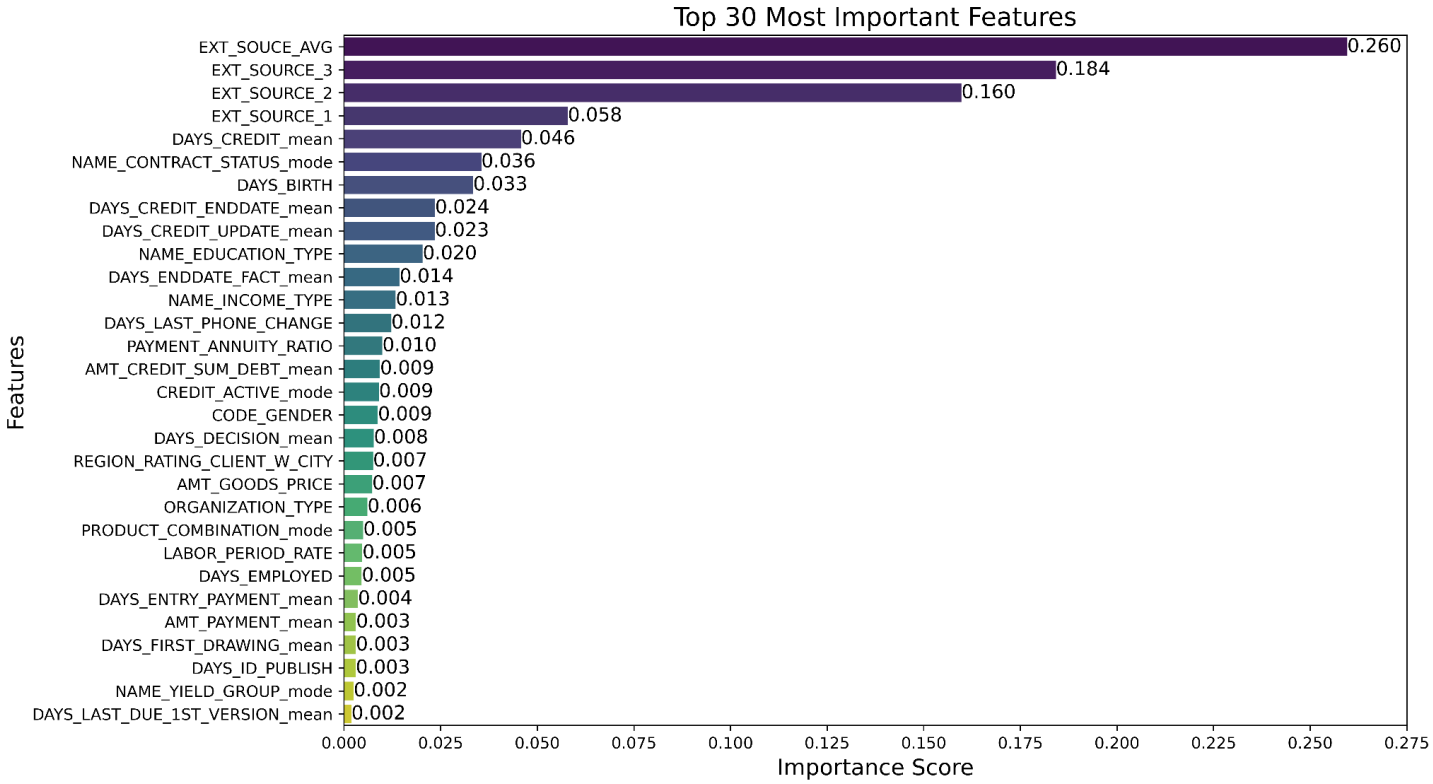
```
def rf_objective(trial):
    # Define the hyperparameter grid
    n_estimators = trial.suggest_int("n_estimators", 100, 1000)
    max_depth = trial.suggest_int("max_depth", 2, 32, log=True)
    min_samples_split = trial.suggest_int("min_samples_split", 2, 14)
    min_samples_leaf = trial.suggest_int("min_samples_leaf", 1, 14)

    clf = RandomForestClassifier(
        n_estimators=n_estimators,
        max_depth=max_depth,
        min_samples_split=min_samples_split,
        min_samples_leaf=min_samples_leaf,
        random_state=42,
        n_jobs=-1
    )

    scores = cross_val_score(clf, X_train, y_train, cv=3, scoring='roc_auc')
    return np.mean(scores)

rf_study = optuna.create_study(direction="maximize")
rf_study.optimize(rf_objective, n_trials=10, timeout= 2400)


rf_trial = rf_study.best_trial
print(f'Best trial: {rf_trial.params}')
```

In order to find best hyperparameters, , we need to define an objective *function* that we want to minimize/maximize ((in our case - cross validation AUC score). We also define a search domain/hyperparameter grid which is basically a set of parameters we can try. We train a model with a particular set of parameters and calculate AUC score. After defining an objective, we can create a study and specify in which direction we want to optimize function and specify time limit as well as number of trials. Generally, Bayesian optimization is more powerful and able to find a decent set of parameters faster since it utilizes the information of previous trials by building probabilistic models and approximations of our objective. I found this framework as a more efficient and powerful alternative to classic techniques such as GridSearch and RandomSearch that do not utilize information about previous trials.

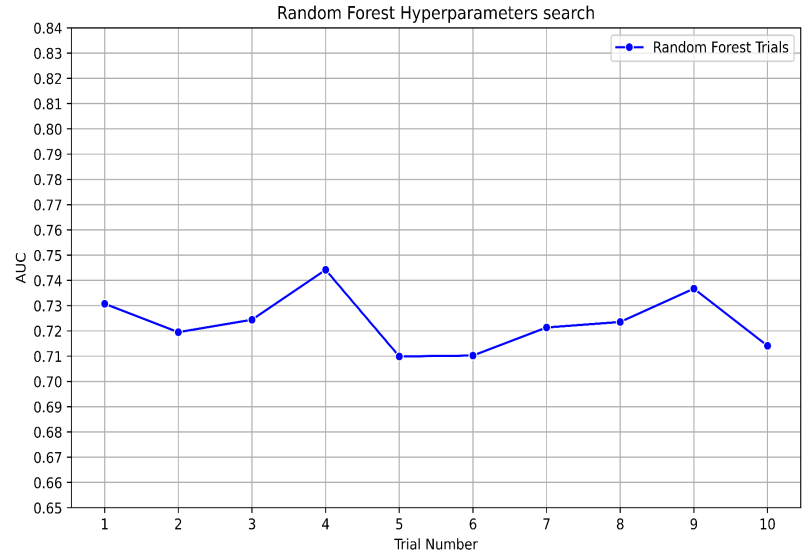
Results

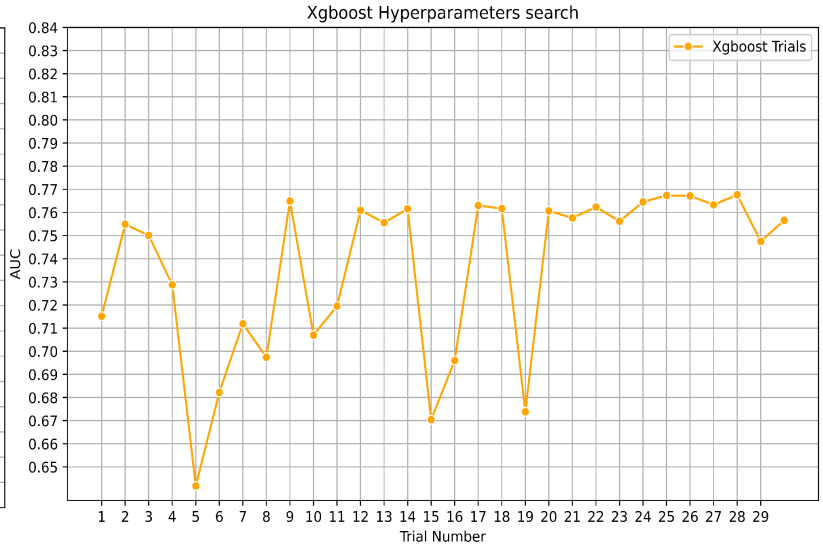
After several hours of training and searching for good hyperparameters for each model, I can provide the following leaderboard that corresponds to performance of each algorithm with the best set of parameters found. The AUC scores were computed using cross validation with 3 folds. Boosting methods turned out to work better and achieved more decent performance than Random Forest. The best model turned out to be Catboost.

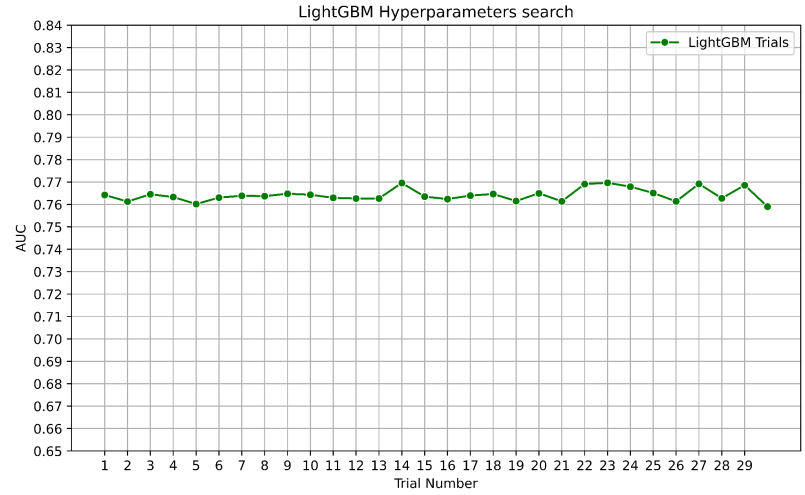
Model	AUC score
Catboost	0.770134
LightGBM	0.769648
XGboost	0.767651
Random Forest	0.744196

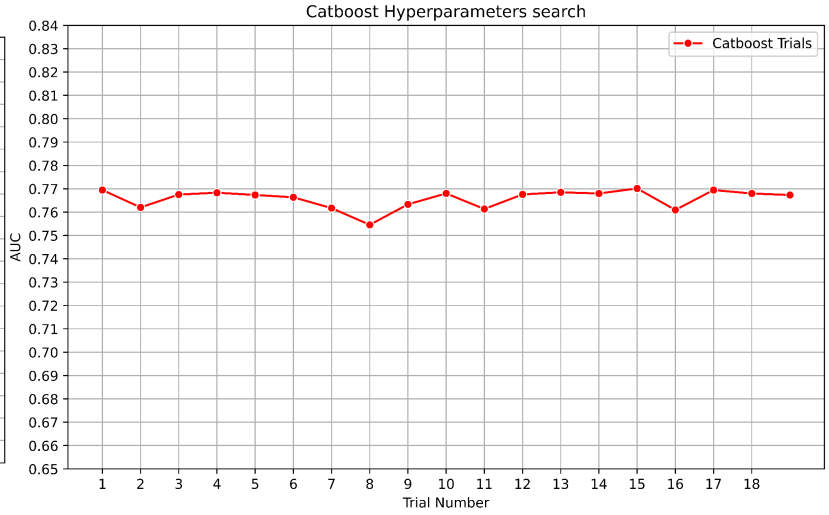


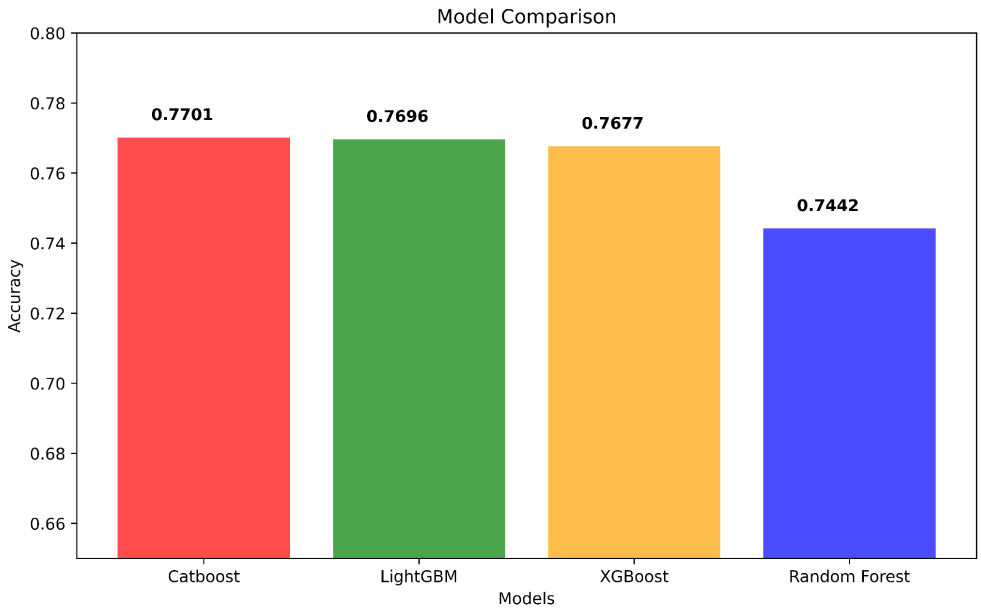
Optimization history











Conclusion

Having meticulously progressed through the entire pipeline, from handling raw data in separate tables to fine-tuning models with optimal parameters, the pivotal next step for the bank is deploying the model into its production system, underscored by the need for vigilant monitoring with entirely new customer-generated data. The incorporation of hand-crafted features has proven to be advantageous, notably exemplified by the most influential feature identified in the importance plots – the average external scoring, which is a hand-crafted feature. This manual feature engineering significantly contributes to the overall model effectiveness. The chosen models for evaluation exhibit relevance to the problem and consistently demonstrate commendable performance. Notably, the CatBoost model stands out with the highest cross-validation score, solidifying its position as the preferred model for deployment in the production environment. This decision is supported not only by its exceptional performance but also by a thorough evaluation of various models tailored to the specific requirements of the problem at hand.