Links to my apps:

Dashboard: https://modellogreg.herokuapp.com/

API: http://modelheart.herokuapp.com/predict Caution: you don't see nothing, because it's only pickle model. All the code that serves and wraps the pickle model is in my GitHub account, in the repository "OC_P7_model_pkl" https://github.com/VeronikaBerezhnaia/OC_P7_model_pkl

Why do so: because I deploy two different applications on two different Heroku accounts, they should be linked to two different folders in Google as well (even though now I start to suspect that on my GitHub account one repository can store the files from both projects and it still works well on Heroku, but this should be tested).

Links to all the project files stored on GitHub:

https://github.com/VeronikaBerezhnaia/OC_P7_model_pkl

Code allowing to deploy the API

https://github.com/VeronikaBerezhnaia/OC_P7_credit_scoring

Files allowing to deploy the dashboard.

The side files, like presentation or methodic note, are also in this folder.

Exploratory data analysis, feature selection, values imputing

The dataset presents ten tables, linked to each other with key columns. For making the dashboard more compact and reducing the execution time on the notebook, I will use only the main table (application train), the dependent one (bureau) and the one dependent from bureau (bureau balance), in order to illustrate the principle how these tables are joined one to another. The table application_train includes the TARGET column (1 means difficulties to pay, "positive target", "target1", 0 means all other cases, "negative target", "target0"). The meaning of all variables of all tables is available in "descriptions" table.

Application_train table: The information about the client is filled to a high extent, while the information about the client's dwelling is poorly filled (probably it comes from aggregation and join from another table). I worked separately with object data type, then with integers called "Flags", where you have 0 or 1 for each (or flags from 1 to 3), then with integers that can take many values (like DAYS_BIRTH), then with floats.

For treatment of object and flag variables, I created "find_most_risky" function, that shows the percent of people bearing every value of a given variable who fail their credits. I created the function "category_to_fail_proportion" to replace categorical values with their probabilities of fail (see "limits and improvements"). When variables looked correlated (like RATING_REGION_CLIENT and RATING_REGION_CLIENT with city), I explored them with confusion matrix, and see the fail probabilities with "find_most_risky" for the lines where these values differ (less than 1% of lines) to select which of them is more useful for the failure detection.

Integer and float variables were explored with histograms. If the distribution is quasi-exponential, I apply log transform. In most cases, after log transform it looks close to the normal distribution (for instance,

INCOME_TOTAL or AMOUNT_ANNUITY), but sometimes distribution stays skewed (like TOTALAREA_MODE—maybe because it only looks like exponential, but in fact it's not). Then I used kdeplots, that show the probability of a client to have this value. I displayed the curves of client groups with targets 0 and 1 on the same plot for comparison. As the class distribution is imbalanced, I declined the common norm (so the distribution is within the group0 and group1 independently). When I see the difference in shape between curve0 and curve1, I select this variable. Surprisingly, INCOME_TOTAL has no impact on the curve shape, neither any synthetic variables derived from it. I tried some synthetic variables that looked relevant to me (ratios var1/var2) and their inverses (var2/var1). For most of them the derived synthetic variables show no difference for curve0 and curve1, except PAYMENT_RATE and CREDIT_TERM. The variables CNT_ADULTS shows the distinction on age distribution: there is a set of lonely old people who manage to pay their credits. From the columns GENDER and FAMILY_STATUS I guess it's about widows who relatively rarely fail to pay their credits.

Variables AMT_ANNUITY and AMT_CREDIT showed the biggest impact on the outcome of the model, as I see from shap summary_plot. But these impacts are always the similar amplitude/abs value but in opposite directions. I suggest that's due to the fact they are negatively correlated. When I delete one of them, the other loses its high importance as well.

The variables EXT_SOURCE_1, 2 and 3 are the most useful, but 1 and 3 have many empty cells. To maintain the relevance as much as I could, I check the probability of fail in missing values: it's merely the same as for the filled cells. So, I imputed the values from the crossing of target0 curve and target1 curve.

The table contains many variables about client dwelling, I suppose aggregates from another table, as they have marks AVG, MODE and MEDI, and plenty of empty values. For every variable the kde curves are almost identical between var_AVG, var_MODE and var_MEDI. So, I selected only those who had different curves for target0 and target1 (with AVG aggregation) and dropped the rest 40 variables.

The dependent tables: bureau balance and bureau: table bureau balance doesn't have missing values. I aggregated the balance by count (how many months the client holds an account), STATUS by get dummies (this variable is important because it shows if the client had DPD (days past due) in his credit history. When I join the bureau_balance to bureau on the given key, bureau table has some missing values. I filled the dummies with X (which is unknown status) and the balance with 0.

Bureau table has missing values in eight columns. To find out the best way to impute, I checked for each variable, how the lines where the variable has a missing value look. The proportion of the bad debt statues isn’t higher for samples with missing variables. I also looked at the lined having CREDIT_ACTIVE variable bad_debt value (there are only 21). Half of them has the debt sum 0 or overdue days 0. So, these variables aren't a good detector of a bad_debt. I filled empty cells with 0, because I aggregate these columns by max or by sum. If I aggregate anything by mean, I will replace the missing values my mean value, in order not to shift the mean to 0. When I joined the bureau to application_train on the given key, quite many values are missing. I filled the numeric columns with the median (because I'm not going to aggregate anymore and want to have a "neutral" value that doesn't shift the decision). Categorial variables were filled with 1 for "unknown", and with 0 for the rest.

Finally, I selected features iteratively: I compared the models on full set of variables, I saw which variables have high shap values and kept them (except if they are correlated). I tried to drop the variables that were not in the list and check whether it reduces performance of the model. It turns out, many features can be dropped without

harm (like CNT_FAM_MEMBERS). Some are essential (like EXT_SOURCE), and some are of smaller importance (they reduce the performance by 0.5%).

Model training methodology, optimization algorithm

Imbalanced class handling: The class distribution is highly imbalanced: only 8% of target 1. That's why I used stratified split by target, keeping 20% of samples as the test split. Class imbalance is an issue during the model training as well. You can solve it by setting the “class weight” parameter to "balanced", that assigns the higher weights to the minor class samples. Also, you can use an oversampling technique, abbreviated as SMOTE. It creates virtual samples of minor class with the values of these new samples located on the vectors connecting any chosen sample with random samples of the same class.

Model choice: I compared 5 models: Logistic Regression, Passive Aggressive Classifier, Random Forest Classifier. Logistic regression is a classic model, Passive Aggressive Classifier is known to work as good as Logistic Regression but is known to work faster and better eliminate redundant features, RandomForest is known to be an efficient predictor, though tends to overfit. The XGBClassifier and LightGBM algorithm were trained only with SMOTE class re-balancing, because these algorithms don’t have the class_weight parameter that I could set to “balanced” (see "limits and improvements").

Hyperparameters adjustment: I selected hyperparameters with exhaustive GridSearch. As a full grid-search on all parameters is time-consuming, I took only 1 or 2 key parameters, which typically have the highest impact on model performance. I compared train scores and test scores from cross-validation to a check if the model tends to overfit. The Random Forest Classifier, XGBoost and LightGBM show much higher score on train set than on test set, so it’s overfitted, even with parameters preventing overfitting (for Random Forest Classifier - low number_estimators, low max_depth, high min_impurity_decrease).

The evaluation metric and the business cost function

The evaluation metric: I used ROC_AUC (area under ROC curve) metric for hyperparameters selection, because it is more representative than accuracy for the imbalanced class distributions. Results of three tested models with 2 tested methods of class re-balancing were collected in a DataFrame and sorted with ROC AUC. LogisticRegression with class_weight = “balanced” showed the highest score, so I used it for further step: personalized metric.

The business cost function: A false negative prediction costs some money to the bank (a bad debt). A false positive prediction costs some money too (bank loses a client). We can assign the costs as coefficients and look to minimize this cost. For example, if a FN makes lose 10 times more money than a FP, we look where is the minimal loss 10* FN + FP. I applied this metric for looking the best hyperparameter C with the cross-validation. I wanted to apply it to find the best probability threshold (see "limits and improvements"). Once the C is found, I trained the model on the full train set.

Local feture importances (shap explainer)

I used the feature explainer from shap library. An explainer is an object that learns on a set of samples, treating the model as a black box, and testing what is the outcome when it replaces this or that feature, one by one. Then the explainer provides an expected value (somewhat like the intercept in a linear regression) and a set of Shapley

values (somewhat like the coefficients in a linear regression). Addition of a sample values multiplied by Shapley values gives the prediction of a shap explainer. The Shap library allows to make the graphs for one client or for a set of clients (see "limits and improvements"). I tested KernelExplainer, that is recommended as a universal for all kinds of models, and LinearExplainer. The latter has a drawback: it returns the prediction values outside of probability interval [0, 1].

API and dashboard

API (Application programming interface) means a program that you launch on the computer through the command prompt. You send the requests to this program through the command prompt and receive the returns of the program. In our case, the program is the saved model. I launched the model first on my computer, before launching them on a server. Heroku is a free server, but it provides only one host per account. So, in order to make two accounts, I had to run two accounts in parallel on Heroku, have two different repositories on Github (this is a question to be tested if I can deploy two applications from 1 github repository, but seems like no) and two local folders (each linked to its own repository GitHub and to its own account on Heroku. When I log in I see two different mails).

On the dashboard, in the frame of this stude project, I kept the addressing to local host. In real life that will probably always be a remote host. Local host works when you deploy the API in command prompt with "mlflow serve" and in another command prompt you run the dashboard on your local computer. They should be both on the same computer (your local computer), only then it works. I keep this functionality to illustrate the principle, but in real life this feature only brings mess and should be removed.

Limits and improvements

1. **The function category_to_fail_proportion** decreases the dimensionality of the table: one column instead of a dummy column for N values of a categorical variable. Worth comparing results with this function or without it. But the model can be saved only in a pipeline, and this transformation is done out of pipeline. To integrate this function into the pipeline, a new class "custom column transformer" should be created that implements this function.

2. SMOTE is claimed to perform better when followed by **undersampling techniques**, such as Tomek Links or ENN. Including this step into the pipeline would probably improve the performance. Also, the undersampling techniques are a must for the algorithms like XGBoost or lightGBM. LightGBM took 1h to be executed with empty parameters grid when using SMOTE. The XGBoost is expected to take even more time than LightGBM.

3. **LightGBM**, which is close to XGBoost by its principle of work, is an interesting algorithm to try at the **model comparison step**. But it isn't created in sklearn library, so it has different methods than sklearn algorithms (say, train instead of fit). And unlike the XGBoost, it has no sklearn wrapper. One possible workaround is to train and test a LIghtGBM algorithm outside the pipeline, to see its performance. But if the performance is ok, the limitation is to save the model. The overcome would be either write a sklearn wrapper for LightGBM, as it is done for XGBoost, or save the lightGBM model with joblib library without using sklearn pipeline and learn to load the saved model.

4. Try **other grid search strategies** (like RandomizedSearchCV) in order to compare more parameters faster.

5. Make sure that folding in cross-validation is stratified (see the documentation: When the cv argument is an

integer, cross_val_score uses the KFold or StratifiedKFold strategies by default, the latter being used if the estimator derives from ClassifierMixin) and **ensure the use StratifiedKFold in XGBoost or LIghtGBM**.

6. The probability threshold in pipeline "predict" method for a binary classification is 0.5. It would be useful to select the **threshold that provides the optimal value of the cost function**. One way to do that is to include the value of this threshold into the grid_search. For that the "predict" method of the pipeline must be rewritten in a way that it returns the probability of belonging to the class 1 instead of a class label.

7. To make group shap **plots that show a client on a background of similar clients**. For that it's needed to define what exactly similar clients are. For example, having the same values – of which variables? Or if these values are floats – what's the bin size? How to withdraw these similar clients from the dataset (if we don't load the full dataset into the dashboard).

8. To retrieve data from the full dataset to the dashboard and to the API (maybe the dataset is stored in yet another computer than API or dashboard). The command for the request must be learned, with all the parameters and specifications.