Assignment 2 – Kaggle Challenge

Team - OtiTheCat

Georgios Markou B00813525
Robin Warrot B00809879
Alexandra Pricop B00806068
Federico Bianchi B00805255
Théo Siouffi B00813442
Roberto Forner B00803830

1. SECTION 1: FEATURE ENGINEERING

1.1 Motivation and Intuition behind features

To begin with, let's see how we preprocessed, selected, and created new features.

1.1.1 Numerical features

First, before talking about intuition behind features, we saw that some features were highly skewed, which can create problems with outliers. Hence, we applied log transform to the most intense (Total EMI per month and Amount invested monthly) and square root to the less intense (Annual Income and Monthly Inhand Salary). During testing, we saw that this led to a significant impact on our results! As a side note, we also tried to deal with outliers by removing 5% of them, but it turned out to be a complete mistake.

Let's move on to the intuition behind features. After inspecting the features, we noticed very interesting patterns that we wanted to capture in each feature: for some features, beyond a certain threshold, we don't have any "Good" credit score. This is the case for Outstanding debt, Delay from due date, Interest rate, Changed Credit Limit, and to a lesser extent Num Credit Inquiries, Num credit card, Number of loan and Number of bank accounts. We therefore wanted to keep these features in our dataset, as they captured important insights. We thought that the insights brought by these features would be very well captured by the splits decided by decision trees!

1.1.2 Categorical features

Now, let's have a look at how we encoded the categorical features and how we tried to have them capture the most insights.

1.1.2.1 Occupation

As professions are not ordinal, we have simply one hot encoded them to see if the profession had an impact on the labels.

1.1.2.2 Payment of Min Amount

Since we thought that there was no sense of "order" in this feature ("not mentioned" is not really comparable to "Yes" or "No"), we separated this data into 2 parts. One column shows whether we know if a payment of min amount has been made, and if known, the second column states, whether it was paid or not.

1.1.2.3 Credit Mix

As we wanted to keep the ordinality of the credit mix, we simply converted "Bad", "Standard" and "Good" values to ordinal values (0,1,2).

1.1.2.4 Payment Behavior

As we found that payment behavior contained information on both the scale of individual transactions, and customers' overall spending habits, we decided to divide this information into two columns. We applied ordinal encoding to these columns to capture their ordinality.

1.1.2.5 Type of Loan

As the "Type of Loan" column contained the different types of loan that users have at the same time, we have created individual binary columns for each loan type. This approach makes it easier for algorithms to identify patterns, as opposed to deciphering combinations from a single column.

1.1.2.6 Label: Credit Score

We investigated whether recognizing the ordinal nature of Credit Scores (Good > Standard > Bad) impacts performance.

To this end, we experimented with three encoding methods: One Hot, Ordinal Label, and a new type of encoding "Ordinal Hot." The latter value representations is [1, 1] to embody the principle of not only possessing the equivalent of [1, 0], but exceeding it.

However, our testing on Neural Networks revealed that accounting for ordinality had a minimal effect, altering performance by less than 1%.

In addition to describing our motivations behind them, note that each feature has been integrated into a correlation matrix to determine their usefulness (strong correlation with the label) and the redundancy (strong correlation with another feature).

1.1.3 New features

After carefully preprocessing our features, we decided to take a deeper dive into our dataset and features and try to construct new features using the already existing ones. After rigorous analysis, we realized a very interesting fact about the dataset. For each unique client, defined by their "Customer_ID", we are given their credit score for 8 different months (1-8). Some of those months are included in the train set and some others are part of the test set. Also, all clients of the test set, apart from 2, exist in the train set. Thus, we could use the past and future credit scores of our clients to predict their credit score for a given month. This completely changes the nature of the problem, since instead of predicting the credit score for any given client, now our goal is to predict the credit score of pre-existing customers for some specific months.

1.1.3.1 Mean Credit Score

Following the idea described above we implemented a basic metric to understand if the general credit score of a client is a good predictor of their credit score for a specific month. The metric we came up with is the Mean Credit Score where we averaged the credit score of each client from the available data of the train set. When we tested this metric in practice, it's influence on accuracy was impactful. Thus, we decided to continue exploring this concept.

1.1.3.2 Exponential Centered Moving Average Credit Score

Following the same idea, we also created a more complex credit score variable that increases the importance of credit scores close to the predicted value. Indeed, what better indicators of a given month's value than the months preceding and following it, and perhaps a little memory of other months?

To create this feature on each line, we simply lookup the Customer_ID in the train_csv, sort by the month, and then apply a centered moving average of the credit scores of the other months (of course not using the current month score, as we don't have them for the test set). When we tried to incorporate this Exponential Centered Moving Average Credit Score feature into our models, we encountered varying responses. Interestingly, some models used this feature seamlessly, experiencing no issues during training. However, other models stumbled and could not train effectively. This discrepancy was traced back to a specific issue: for two clients in our dataset, the credit score calculations resulted in NaN (Not a Number) values. We hence chose to replace those NaN values by 1, as it is both the highest occurring value and an average value, which shouldn't impact the combinations with other features too much.

Note: While creating the EMA feature, we understood that the EMA had an enormous potential for high scores in the Kaggle competition. Indeed, by using only the Exponential Centered Moving Average, we can reach an accuracy level of 83.14% **without any algorithm**.

However, we are aware that beyond this exercise, in a real-word case, it would be strange to predict month 5 knowing that we know months before and after, so our algorithm might overfit to our current dataset and perform badly for new customers for which we don't have any data.

Nevertheless, since the task at hand is to reach the best accuracy, we used this new feature. In addition, our method could still work if the train/test split was shuffled so that months 1 to 6 were in the train set, and we had to predict months 7 and 8 from the test set: instead of using a centered average, we would just a normal exponential moving average, putting more emphasis on previous months.

1.1.3.3 Polynomial Features

The final thing that we tried is to use polynomial features of our existing features and see if it would increase the accuracy of the model. We did that by creating the squared representation of our numerical features and testing their impact on "Credit Score" using logistic regression. Sadly, our experiment did not prove fruitful, and we did not use those features for our final model.

1.2 Feature Importance and Selections

To understand which features were relevant for predicting the Credit Score of a client, we created a correlation matrix between all our features. This matrix showed us that many of our features did not have any correlation with our target variable. We decided to remove those features and see how our models performed without them. To our surprise, the accuracy without those features was lower in every single experiment compared to when we added them. Thus, we decided to keep them. Finally, we saw that "Monthly_Inhand_Salary" is an exact multiple of "Annual_Income", so we chose to discard it.

2. SECTION 2: MODEL TUNING AND COMPARISON

In this section we are to walk you through how we selected, tuned, and evaluated different classification models. For each model, we used a well-defined pipeline to carry out this task. First, we created a parameter grid for each model. Then we used Randomized Grid Search with Cross-Validation to find the best parameters. We chose a Randomized Grid Search since the execution time tends to be extremely long for most of the models. Also, instead of the normal Cross-Validation algorithm we opted for a Stratified CV. We chose this version of CV since we are dealing with a classification problem and if we randomly split our data, we might get an uneven distribution of labels. Stratified CV deals with that issue and provides more trustworthy measures of accuracy. The last step of the pipeline includes storing the best performing parameters for each model, training our models using the whole training data set and the best performing parameters, and submitting our results to measure our accuracy on the test set. Note that we used random seeds for our non-deterministic models to achieve reproducibility. For the results, look at table 1.

2.1 Classification Models

2.1.1 Logistic Regression

A "staple" in machine learning classification, we decided to use logistic regression since it is one of the most interpretable methods, it's fast to train, and it would give us a nice benchmark to start our model evaluation journey. For the implementation of this model, we used the library "Sklearn". Since our problem is not binary classification, we set the parameters to "multinomial". Also note that we do not need to define the exact number of classes that we need to predict, since in the multinomial approach, the model uses the One-vs-Rest approach. We also accounted for regularizing our model, so we added a regularization parameter to prevent our model from overfitting. The final results, as expected, were not great, and interestingly enough, the cross-validation score of the best model was worse than our accuracy on the test set. We suppose that this is happening because the problem is too complex for logistic regression to solve.

2.1.2 Random Forests

Next, we decided to use decision trees. As mentioned previously, by observing our data we saw a lot of possible ways to "split" our features that would lead to predicting the correct label. Following this observation, we decided to use different decision tree methods, starting with Random Forests. Specifically Random forests, when it comes to classifying, is an assembled method of learning that combines multiple decision trees, that then "vote" on their predictions to produce the final prediction. For the hyper parameter tuning we tuned 'min_samples_split' to determine the number of samples required to split a node, trying various values to prevent overfitting. Also, we accounted for the number of estimators and the max_depth of the tree to balance performance with overfitting. Finally, even though our best performing CV model did not use bootstrap methods, we found out, through research, that bootstrapping tends to increase the performance of the model. In our case since we used a Randomized Grid Search maybe it happened that it randomly selected it to be false while the other parameters indeed boost performance. After evaluating our model by adding bootstrap the model's performance increased but by a slight margin (0.001 increase). The results were significantly better than logistic regression, but also was the training time.

2.1.3 *XGBoost*

After Random Forests, we decided to use the XGBoost model, provided by the "xgboost" library, which is an advanced implementation of the gradient boosting algorithm. Contrary to the Random forests, XGBoost uses boosting method, that means that one tree is created at a time, and then a new tree is created to help correct the mistakes of the previous one. For this model we specified that we are dealing with a multiclass problem by setting the objective to 'multi:softmax' and the number of classes to 3. Also, for the learning rate and the number of estimators, we tried to use small learning rates with high numbers of estimators to increase our model's accuracy. Finally, we adjusted and added L1 and L2 regularization parameters to prevent overfitting. What is interesting is that the model performed better with high L2 and low L1 regularization parameters. From our understanding, this happens because the significance of our new feature is too high, and the L2 parameter helps to even it out. Overall, the XGBoost model outperforms the random forests method, while keeping the training time shorter.

2.1.4 *LightGBM*

Finally, we tried the LightGBM framework, using the "lightgbm" library. Specifically, we used a LightGBM classifier, and researched the way hyperparameters interact with each other to achieve a balance between overfitting and underfitting. Some notable points that we found are that we tested between 'Dart' and "gbdt" boosting methods and the 'Dart' method took both more time to train and produced worse results. Also, we tried different metrics, such as multilogloss and mulit_error, and multi_error outperformed multilogloss in every single hyperparameter tuning experiment conducted. Also, in the beginning to get a sense of the good hyperparameters, we tried to minimize our training time and experiment faster with different values. We chose a bigger learning rate and a relatively small number of iterations, while adding a stopping round parameter that was 10% of the number of iterations. Also, since the models tended to overfit, apart from using a small number of iterations we also introduced regularization parameters. The LightGBM model outperformed all other models and provided us with our best solution. Overall, the great accuracy of LightGBM in addition to its very fast training time make it one of the most attractive models we used.

2.1.5 *Neural Networks*

Finally, we wanted to see whether a Neural Network would be able to capture patterns and combinations of features even if we hadn't explicitly created them. We hence chose to create and fine tune a Neural Network without any feature engineering like the Exponential Moving Average feature.

After creating our algorithm with different numbers of layers, we initially tried to use a grid search methodology for hyperparameter optimization. This approach, though comprehensive, proved to be computationally intensive. To mitigate this, we opted for a sequential hyperparameter tuning process. While simultaneous adjustment of all hyperparameters is theoretically ideal, we discovered that modifying them sequentially, while keeping others fixed, was still effective in enhancing model performance.

We hence evaluated various hyperparameters over a wide range of epochs, implementing an early stopping mechanism when the validation loss showed negligible reduction. The hyperparameters examined included model size (quantified by the number of hidden layers and neurons), learning rate, learning rate decay, batch size and the lambda coefficient for L2 regularization, in addition to the three distinct model architectures adapted to each of the three Credit Score encoding schemes.

We systematically plotted learning curves in order to compare how quickly the models were learning and how they were overfitting. The early stopping mechanism helped us to stay away from overfitting too much, but even though we tried to use methods like L2 regularization or dropouts, we still didn't manage to have the model not overfit. We say that adding L2 regularization worsened the results, as we can see in the table of results.

We also used confusion matrices to monitor whether the algorithm was maybe very good or bad at predicting a specific score, but it turned out that there wasn't any problem on this side.

The table with the results of the 50 tests can be found in the notebook.

2.2 Cross-validated & test performance

For the performance of our model, we would like to address some concerns. As discussed in the feature engineering part, we created a feature that completely changes the way we view and solve this classification problem. This particular feature is in fact so good at predicting the credit score of a customer that without employing any machine learning algorithms it reaches an accuracy of 83%. Nevertheless, our goal for this project along with learning and trying various models is to achieve the greatest result. Thus, we decided to use it for all of our models.

Our best solution came from the LightGBM model. We highlight it with bold letters. For your convenience, in our Notebook file there is a section called "Best Submission Reproduction" where you can run the code and produce our best solution.

Below we present the Cross-Validated results, along with our prediction accuracy in the test set:

Models tested	CV Accuracy	Test Accuracy
Random Forest	85.49%	85.47%
Logistic Regression	82.79%	83.26%
XGBoost	86%	85.7%
LightGBM	**86.14%**	**86.56%**
Shallow Neural Network	73%	73%
Deep Neural Network	78%	77%

Table 1. Results of our algorithms in Cross Validation and Test-set.

3. References

[1] "How to set up neural network to output ordinal data?" (Source: StackExchange, 2015)

[2] https://neptune.ai/blog/lightgbm-parameters-guid

[3] https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74

[4] Optimizing XGBoost: A Guide to Hyperparameter Tuning | by RITHP | Medium

[5] sklearn.model_selection.RandomizedSearchCV — scikit-learn 1.3.2 documentation