FACULTY OF ECONOMICS AND
BUSINESS ADMINISTRATION

InClass Kaggle Competition: loan default prediction

Technical report

Course: Machine Learning

Teacher: Prof. Dr. Dries Benoit
Arthur Thuy (TA)

Students: Team 02

Kasper Halewyck	01707513	kasper.halewyck@ugent.be
Stijn Heuninck	01807847	stijn.heuninck@ugent.be
Jesse Lema	01707412	jesse.lema@ugent.be
Pieter-Jan Verhelst	01807049	piverhel.verhelst@ugent.be

Academic year: 2021–2022

GHENT
UNIVERSITY

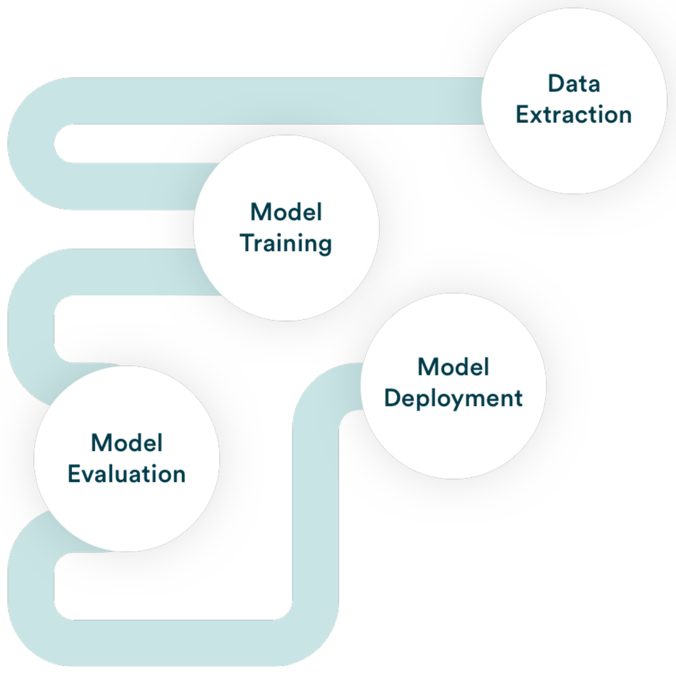
Contents

1 Introduction

Machine learning is a hot topic in literature and the business world because of its powerful predictions. In various industries and fields, machine learning is already being used to make accurate predictions. In the financial world, machine learning can also offer enormous added value. In this paper, we take a closer look at a model that makes it possible to predict the willingness to repay a loan. This is a very powerful tool for credit providers such as KBC and Deutsche Bank which can significantly increase the profitability of credit providers. First, the processing of the collected data is discussed. After this, successive different predictive models (logistic regression, GAM, SVM, tree based methods, XG-Boost and ensemble model) are discussed. At the end of the paper, a brief overview of the performance of the different models is given.

2 Machine learning pipeline

The traditional machine learning pipeline consists of 4 parts: data extraction, model training, model evaluation and model deployment. The first part data extraction consists of data cleaning and feature engineering. This is done in the bronze and silver data layer in out repository. Here all the imputations of the data are done in a way that all the models can use the correct complete data. The model training part and model evaluation part is done in the same file in our project. For each model, we first hypertune the parameters and evaluate the trained model on the validation set. This can be found in the src file. The last part is the model deployment. Since in our case it is a Kaggle competition, we can deploy our model 3 times a day.



3 Data exploration and cleaning

Before we can start working with the data, it is necessary to make sure the data is complete. Everything written in this section is done in the R file: data_cleaning.R In this first step, we started with looking at what the different columns in the data are. Now that we know what the data types are of the data set, we make a histogram of all the variables to look at how the data is distributed. This is done on line 50 until 53. The next part is looking for missing values within the data set. We have used the describe function of the package Hmisc. This function gives a handy overview of all the basic statistics for each variable. Based on the outcome of this variable, we concluded that there are missing values for the variables: annual_income, emp_length, emp_title, home_status, monthly_payment, num_bankrupts, num_mortgages, num_records, num_total_credit and revol_util.

Another conclusion of the describe function was the levels of grade and sub_grade. Since there is no general rule of how these values should be interpreted, we have chosen to follow the standard rules of the big rating agencies. This means that an A rating is the best and a G rating is the worst. And for sub_grade this means that A1 is better than A2 and A5 is better than B1.

Now that we analysed where the missing values are, we are going to fill them in. First of all, for the numerical variables, we made a new variable to show that there was a value missing in the original data set. This is because we assume the missing data is not random. Meaning that missing values could be an indication of default. Note that we only do this for the numerical values because for the categorical ones, we simply fill them in with the value Missing. Now that the flag and categorical columns are constructed we fill in the missing values. This is done by a special package named mice. The mice package is a package designed for handling missing values. Mice stands for Multiple imputation using Fully Conditional Specification (FCS) which is implemented by the MICE algorithm. Since our data is numeric and continuous we chose the pmm method of the mice package. pmm stands for predictive mean matching. pmm involves selecting a data point from the original, non-missing data which has a predicted value close to the predicted value of the missing sample. It searches similarities based on the non missing data and than chooses 1 out of the 5 chosen candidates as a donor. Therefor the missing value is filled in with a real possible value rather than just the mean or the mode of that variable. This has the result that the distribution of the data stays the same (with mean imputation, a lot more variables has the same centered value). We run the function on a rbind of the train and test set (without default which is complete). In this case there is more data for the algorithm to find the best possible donor. This also does not result in data leakage because all the distributions stay the same.

After running the algorithm we have a train and test set without missing values. These datasets are written to the silver layer as a basis for the feature engineering.

4 Feature engineering

In the feature engineering section, we will continue to pre-process the data. First, we will create new variables given the data, secondly we integer code some numeric variables, thirdly the one-hot encoding and binary encoding and to finish we scale the data and handle the class imbalance.

4.1 New variables

The first new variable that we created is the days_between variable. This variable gives us the days between earliest credit time and date funded. This variable shows how long the debtor is taking credits. A higher value means a larger difference between earliest_cr_time and date_funded. We think that new debtors might have more risk at default than more mature debtors.

The second variable is city. From the address variable we extract the state and postal code, which gives us the city the debtor lives in. We think this has an impact on default because it matters if the debtor is from a rich neighborhood or a poor neighborhood. This is done with the help of some regex code.

The third addition is actually an adaptation of the emp_title variable. Since the emp_title variable has a lot of unique values and a lot of values that are the same but are paraphrased differently, we decided to extract some job titles we think could have an impact on default. The titles we extracted are: manager, director, supervisor, senior, analyst and accountant. People having these jobs normally earn a decent amount of money, which is positive if you need to pay off a loan. For each of these titles, we made a new binary variable that indicates whether the emp_title contains a certain title.

4.2 Integer encoding

Integer encoding is used for categorical variables where there is a ranking in the levels of the variable. The variables that we encode are emp_length, grade, income_verif_status, term and sub_grade. The levels of emp_length go from missing to 10+ years. We used this ranking because a longer time of employment shows more stability resulting in a more stable income which is good to pay off a loan. For the grade and sub_grade variables, we used the ranking of the large credit companies. This means that G is the worst and A is the best. In the sub_grade, 1 is better than 5. For example B2 is better than B4 but B5 is still better than C1. The income_verif_status levels are ranked based on verification. Not verified is the worst, in the middle there is source verified and the best level is verified. The last variable is term. Here the levels are 36 months and 60 months, with 60 months better than 36 months. This is because the longer the credit goes, the smaller the monthly payments are, resulting in a higher chance the debtor can pay off his or her loan.

4.3 One-Hot encoding and binary encoding

Since for most of out models, we do not need one-hot encoded columns, the one-hot encoding of most variables is done in the file of the specific model that needs the one-hot encoding. All the flag columns, had_missing columns, are binary encoded. A true value is 1 and a false value is 0.

4.4 Scaling

Now that all the columns that needed to be numeric, are numeric, we can scale the data. We scale the data so that the difference of magnitude of values does not have an impact on the model. We first scale the train data. Afterwards we scale the test data but using the mean and standard deviation of the train data. This is necessary because else there is a loss of information, which is negative.

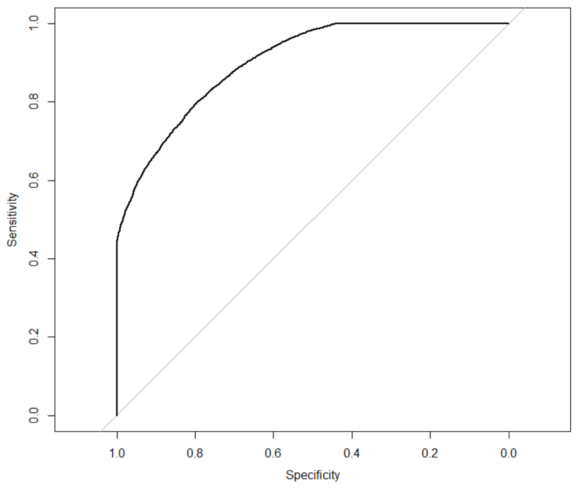
4.5 Class imbalance

The final part of the feature engineering is handling the class imbalance. Since a default is less likely, the amount of defaults is smaller than the amount of non-defaults. Therefor the model cannot train very well. To solve this problem, we impute some extra defaults in the training set. These extra imputations are duplicates of already existent rows in the train set that have a default value. In our code this is done by the upSample function of the caret package. The upSample funcion uses the oversampling technique, this means adding observations to the data set until there is a balance between default values and non-default values. The alternative is under sampling. This is deleting non-default observations until default and non-default are in balance. Here we have a downside that not all the available information is used. Therefor we chose the oversampling technique.

5 Logistic regression

The first model that we make is a logistic regression[1] model. This is a statistical model which predict the probability that a certain observation belongs to a certain class. This is one of the most basic models but it gives a good indication of the quality of our feature engineering. Since a logistic model cannot be hypertuned the model training is rather easy.

The logistic model has an AUC of 0,9012 on the validation set and a 0,89790 on the test set on Kaggle. The logistic model serves as a baseline model for the more complex models.



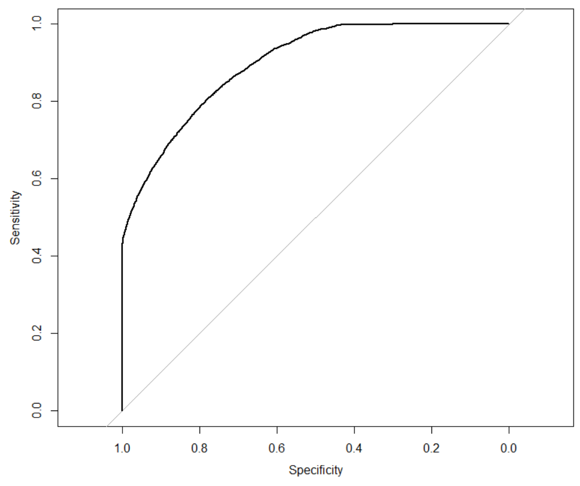
Another way to evaluate the logistic model is with an confusion matrix. In this case it can be better to modify the threshold. With a lower threshold (lower than 0,5) a lower false positive and a higher false negative. The amount of correct predictions remains the same but the amount of predictions were debtors are not predicted to default, default is lower. In this case the bank loses less money because the chance of non-default needs to be higher to receive a loan than in the standard logistic model. The downside is that there is a bigger loss in opportunity cost. Meaning we miss a bigger % of debtors that would not default. A fine balance between those 2 is optimal. This is unfortunately not the assignment since we need to evaluated based on AUC. Therefore we did not calculated the optimal threshold to have minimal loss for the company.

[1]https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/glm

6 Generalized Additive Model

The next model we use to predict whether a loan applicant would default is the Generalized Additive Model[2]. For this GAM we used smoothing splines since we didn't want to overfit the observed data, which is a danger when using for example regression splines. We first trained a linear model that included all our variables to obtain a baseline model. Then we created a model in which all numeric variables have 3 degrees of freedom, just to have an idea what the results were on the AUC on the validation set. We decided that 3 should be the maximum degrees of freedom, again to to make sure that we wouldn't overfit the model. We found that the linear model had a remarkably high AUC of 0,8974 while the basic GAM with 3 degrees of freedom had an AUC of 0,8986.

To optimise the number of degrees of freedom for a variable, we created a function that fits a GAM with the degrees of freedom for that variable ranging from 1 to 3. Then we used a series of ANOVA tests to determine which number of degrees of freedom was best, while all other (non-categorical) variables were kept linear. We found that this was computationally the most feasible way that would deliver results. We then fit a new GAM with the obtained degrees of freedom. For this fit, we also deleted the variables that had a non-significant p-value. The AUC of this model (0,8984) didn't increase compared to the basic GAM where all variables had 3 degrees of freedom, however we believe that this model would perform better on new data.



[2]https://www.rdocumentation.org/packages/gam/versions/1.20

7 Support vector machines

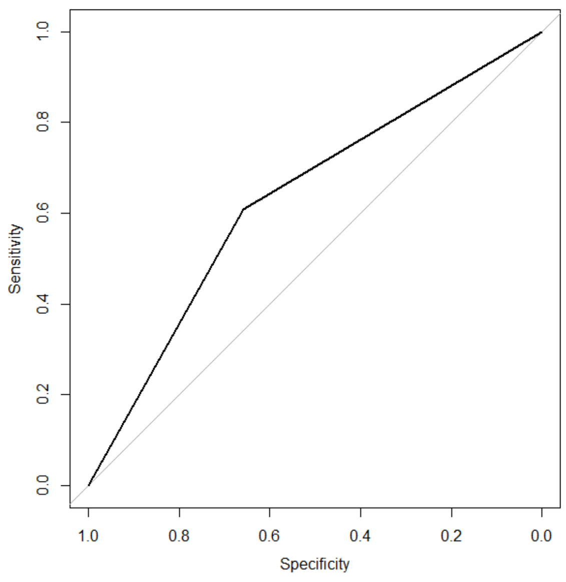
Support vector machines[3] are a set of supervised learning methods that are useful for classification and regression. The advantage of a support vector machine is the fact that this method is still useful when the dimension is higher than the number of observations. The disadvantage is that support vector machines are not suitable for large data sets. In a support vector machine, the algorithm tries to maximise the distance between two points of a different class. In the case of unsupervised data, this is referred to as support vector clustering. In the section below, we discuss how the model was developed.

The model was developed with the train dataset and afterwards evaluated with the validation dataset. The model could not be trained if there were character variables ("application type", "home status", "purpose", "earliest cr line", "city", "date funded") in the data set. Therefore, we prepare the data with one hot encoding before training the model. This results in a dataset with extra variables. In calculating the SVM, we noticed that the time to train the model was extremely long. This due the fact that we create many variables during one hot encoding. We checked the support vector machine based on the dataset with one hot encoding and the dataset where the columns with categorical values were removed. The dataset with one hot encoding performed better. We could also choose to create only binary variables for the most common categories of categorical variables such as "city" with many categories. There were no real peaks in the frequencies of each category. As a result, quite a lot of information would get lost.

The best support vector machine is finally obtained via the svm() function with cost equals 1. This parameter was optimised via the tune function. The cost parameter C gives an indication in what extend the model want to avoid misclassifying each training observation. The smaller C, the higher the chance of overfitting the model. In non-linear kernels there is also a gamma parameter. This parameter defines how far the influence of a single training example reaches. When gamma is very small, the model cannot capture the complexity of the data. The different support vector machines based on other kernels were compared. The support vector machine with a linear kernel performed better.

[3]https://www.rdocumentation.org/packages/e1071/versions/1.7-9/topics/svm

The model has an AUC of 0,68. This indicates that the predictive performance is not very great. This is due the fact that support vector machines are not very suitable for the classification of large data sets.



8 Tree based methods

We used 2 different tree based methods. We started with a random forest and then we did a boosting model.

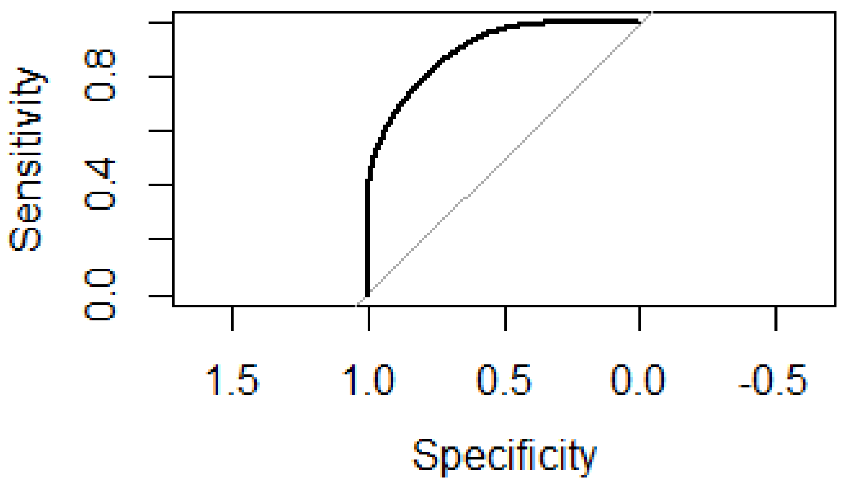
8.1 Random forest

We fitted the random forest[4] as a first model. Here we hypertuned the parameters using the OOB estimate of error rate. The first parameter was the number of trees. We found that we had a lower OOB estimate of error rate when we had more trees, fitting a lot of trees was not possible because computational reasons. It is however possible that a large value for number of trees results in overfitting, we however found no evidence of this at the values we tried (up to 1.000). The second parameter was mtry, this is the number of variables used by each tree. We again used the OOB error rate and found an optimal value of 8. We decided to use this value of 8 and set the number of trees to 1.000. We then used our model to estimate the values of our validation set and calculated the AUC, this showed that we had an AUC of 0,7. Because this is not that high, we decided to look for another method.

[4]https://www.rdocumentation.org/packages/randomForest/versions/4.6-14/topics/randomForest

8.2 Boosting

Because our first model yielded pretty average results, we fitted a second model. This model was a boosting model[5]. We had to hypertune 3 parameters. With this model, we could however use cross validation provided by R, we chose 5 folds. The parameters are number of trees, the shrinkage parameter and the interaction depth. The optimal values we found was 1.000 for number of trees, 11 for interaction depth and 0,09 for the shrinkage parameter. This model resulted in an AUC of 0,8981 on the validation set. Because of computational reasons, we had to shortcut the searching for the optimal parameters, we saw that the cv error decreased until 1.000 trees, for interaction depth fixed to 6 and shrinkage parameter fixed to 0,1 (and probably further but we did not test this). For this value we used the cv error to find the optimal value of interaction depth, this was 11 (still with shrinkage parameter set to 0,1). We then looked if changing the number of trees with the interaction depth set to 11 could further decrease the cv error, this was not the case. Then we optimized the shrinkage parameter with the optimal values for the number of tree and the interaction depth(1.000 and 11), here we found an optimal value of 0,09. We then again looked if changing another parameter further decreased our cv error. When running our completer model, we found an AUC of 0,8981 on the validation set.



[5]https://www.rdocumentation.org/packages/gbm/versions/2.1.8/topics/gbm

8.3 XGBoost

The last model is a extreme gradient boosting[6] model, XGBoost. XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way [7].

First of all, it is necessary that the data is in a xgb.DMatrix . A DMatrix is an optimized data structure that provides better memory efficiency and training speed. This is necessary for the XGBoost model to run.

The XGBoost model has 3 types of boosters: a linear one, a tree one and dart. Since our problem is of the classification type, a linear model is not useful. The difference between dart and tree is that dart uses in the predict() function random dropouts so that the chance of overfitting becomes lower. The tree booster does not do this automatically.

To make sure we use the optimal model, we hypertune the parameters. Since the XGBoost model has many parameters, hypertuning them all is computationally impossible with our laptops. Therefor we made a selection of parameters to hypertune. The parameters are: eta, max_depth, sub_sample, colsample_bytree and min_child_weight. Eta is a step size shrinkage used in update to prevent overfitting. The max_depth is the maximal depth of a tree. A higher value gives a higher degree of complexity. Subsample is the ratio of the training instances. The values shows the percentage of the training data that is used to prevent overfitting. Colsample_bytree is the ratio of columns used when constructing each tree. The min_child_weight is the minimum sum of instance weight needed in a child. This means that if the partition step in a leaf node with the sum of instance is higher than the min_child_weight, the building process will give up further partitioning[8].

There are also the parameters lambda and alpha. These are shrinkage parameters. Since we want a high AUC, we set these parameters to 0, meaning that there is no shrinkage based on the L1 and L2 regularization.

Since there are a lot of parameters, going over every possible combination, more than 2 million instances, would be computationally impossible. Therefor we make a data frame with 10.000 random combinations of parameters. This is an approximation of the true best model. Unfortunately, this is the only way we get a result using this model without having to wait a full year (300 iterations per hour), which is due to the time constraint of the group project impossible.

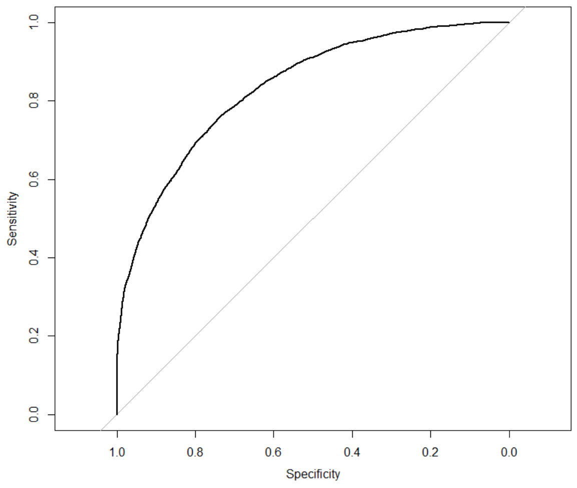
Once we have the data frame with 10.000 parameters, we train the 10.000 models with the xgb.train function of the XGBoost package. Here the model trains on the test set based

[6]https://www.rdocumentation.org/packages/xgboost/versions/0.4-4/topics/xgboost
[7]https://xgboost.readthedocs.io/en/stable/
[8]https://xgboost.readthedocs.io/en/stable/parameter.html

on the parameters of the dataframe mentioned above. To prevent overfitting, a watch list is implemented. The watch list computes the AUC of the validation set. The model looks how this AUC evolves. When for 30 rounds the validation AUC has not improved, the model stops to prevent overfitting and returns the optimal iteration. Because of this overfitting is no issue in this model.

Once we have for all the 10.000 combinations of parameters a model, we look up in the randomsearch dataframe which parameter combination results if the highest AUC. We extract these parameters and use them for the final model. The final model is used to predict the test set.

The evaluation of this model is an AUC of 0,856.



9 Ensemble model

The final model, which is not really a model, is an ensemble model. Here we have combined the output of the models described above and used soft voting to have a final prediction. Soft voting is taking a summation of all the probabilities and dividing it with the number of models[9]. This model has an AUC of 0,902 on the test set.

[9]https://www.oreilly.com/library/view/machine-learning-for/9781783980284/47c32d8b-7b01-4696-8043-3f8472e3a447.xhtml

10 General conclusion

Model	AUC
Logistic regression	0,9012
Linear model	0,8974
Basic GAM	0,8986
Evaluated GAM	0,8984
SVM	0,6869
Random Forest	0,7145
boosting	0,8981
XGBoost	0,856
Ensemble method*	0,90163

Table 1: Summary of AUC per model on the validation set

* AUC on test set

We can see that the ensemble model scores best on our chosen evaluation metric. This model is useful because every model has his downside and by combining the models, the downside of one model is minimized by all the other models.