Long Term Credit Rating Projection – Abstract

Data

Gathering Data

The primary resource of data is the Wharton Research Data Services data base [WRDS] (2019). Thanks to the prepared queries and the company list we received through OLAT; the desired datasets were gathered quickly. Additional data, like inflation data or US-treasury bond rates were obtained from the Federal Reserve Economic Data's [FRED] website (2019). We chose to consider quarterly data over the observation period of seven years for all the companies in the S&P 500 Index.

The first part of the project was dedicated to data gathering. After importing each .csv-file, we dropped obsolete columns (such as PERMNO and CUSIP) immediately. The date columns and some ratios were formatted and renamed to obtain a uniform format.

As a next step, we categorized the sector column "gsector" of the WRDS credit rating data, using the "get_dummies" function of the panda library. In doing so, we obtained a dummy variable for each sector.

We merged the WRDS financial ratios data with the company list and the WRDS credit rating data via the ticker, year and month. The dataset was then extended by adding the FRED's data (1Y, 5Y and 10Y US-treasury bond rates as well as yearly US-Dollar inflation).

Furthermore, we calculated a log-return variable on share prices, using a lambda function [np.log(x) - np.log(x.shift())] where we grouped our data set by ticker and share price.

The response values, in our case the long-term credit ratings, consisted of 18 different integer values. Rating "D" only featured 17 observations, so we decided to drop it. The remaining 17 values were factorized into five numerical classes by mapping them as follows:

Rating	AAA – AA-	A+ – A-	BBB+ – BBB-	BB+ – BB-	B+ – CCC+
Meaning	Prime & High Grade	Upper Medium Grade	Lower Medium Grade	Non-Investment Speculative Grade	Highly Speculative & Substantial Risk
Numerical Value	4	3	2	1	0
No. of observations	1421	7438	11445	3257	662

This split made sure each class had enough observations, while still differing between investment grade (AAA- BBB-) and speculative grade (BB+ or worse) (Standard & Poor Financial Services, 2019).

Handling missing values

As to decide how to handle our missing values, we first checked how much of the data was missing. Therefore, we calculated missing values in the columns. All features (columns) had more than 65% non-missing values, that is why we decided to keep all the columns. We proceeded by checking for zero values, keeping in mind that the dummy variables had a lot of zeros that are deliberate. However, three of the remaining features had too much zero values in our opinion (more than 45%). That is why we dropped: "staff_sale", "rd_sale" and "adv_sale". Second, we checked the rows for missing values. Here, we decided to drop all rows with more than 70% of the features missing by specifying a threshold in the dropna() function.

To impute our missing values, we decided to use the imputing with "Multiple Imputation through Chained Equations" [MICE]. For that we used the MICE Imputer from the statsmodels package, which calculates imputed values that strongly resemble our real values. As a first step, a certain number of rows is bootstrapped from the dataset and all the missing values in this bootstrap sample are imputed with the mean of the respective column. One after another imputed missing value is then cleared again (the mean serves mainly as “place holder”) and estimated separately with a regression, where all the other columns are used as regressor variables. This is done until all the missing values are imputed with this procedure and then this whole process is repeated for multiple cycles. Here, the ideal number of cycles lies between 5-10. We decided to limit on 5 iteration to avoid a too time consuming data processing. The idea behind this multiple iteration is, that through the repeated process the distribution of the parameters will converge, and the final imputed values shouldn’t depend on the order of imputation anymore (Azur, Stuart, Frangakis & Leaf, 2011). We decided to use this imputer on all columns but the ticker, date columns, bond rates, inflation and our credit rating column (response values). After imputing the remaining columns with the MICE imputer, we merged our imputed columns with the remaining, not yet imputed columns from mentioned above.

To deal with our missing bond data we decided to use the fillna() function were we used the ffill method as the values of the interest rates are relatively stable.

We applied the fillna() function with the median method for missing ratings, where we grouped ratings by ticker. Hence, we ensured that the average rating of a firm is inserted and effectively rounded up to only allow for integers values. This provided an appropriate way to fill individual missing ratings of firms for the rather constant measure over time. For firms that lack too many rating observations, we wouldn’t get any useful information thus we dropped the values.

We got rid of all the missing values in our dataset and we saved it as a .csv file for further use.

Preparing the Dataset for the machine learning algorithms

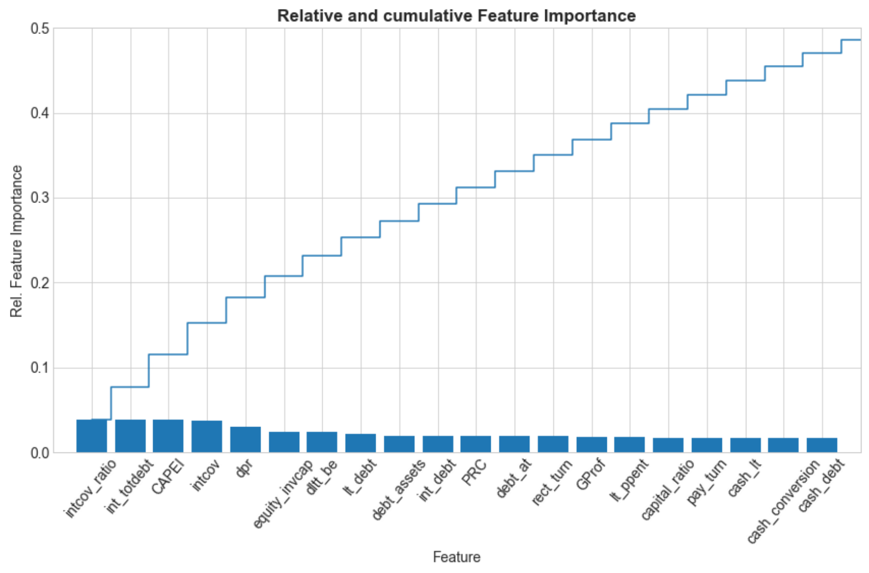
First, we split the dataset into the features (X) and responses (Y, our ratings column). We dropped the ticker and date columns as they will not be used by the algorithms.

We encountered class imbalance in our ratings column (see table) by using the Synthetic Minority Over-sampling Technique [SMOTE]. Technically one could address this problem of class imbalance in different ways: either over-sampling the minority classes, under-sampling the majority class, penalties in the algorithm (so that it makes misclassifications of minority instances count more than misclassifications of majority instances) or by synthesizing new minority class instances. This last option is exactly what SMOTE does. It takes the vector from an observation to one of its k-nearest neighbors and then creates new artificial minority class observations by multiplying this vector by a random number x which lies between 0 and 1. The SMOTE function does this until all the classes are up sampled to the highest available sample size, that is in our case 10’611 observations in each class (Kunert, 2017). We decided to use this SMOTE function rather than the simpler resample function because our algorithms achieve better scores by doing so.

The features (X) and the responses (Y) were then further split into train and test sets. We’ve done this, using the train_test_split() function specified with a test size of 20%.

We decided deliberately to only put the StandardScaler()-function into the pipelines of our algorithms because GridSearch uses cross validation and there it is crucial to not scale the train and test set in advance. (Side note: Random Forest is immune to issues considering scaling)

To identify which of the features are most important we used a random forest classifier and estimated the following relative and cumulative feature importance:



Source: own figure

We found that the most important features are the Interest Coverage Ratio, Interest/Average Total Debt, Shillers Cyclically Adjusted P/E ratio, After-tax Interest Coverage and the Dividend Payout ratio. Overall, we see that the most important features are mainly ratios with no ratio standing out, as expected. By investigating the code, we noticed that the least important features include the sector dummies, bond rates and our return on share price. Apparently, these explain only a fraction and should therefore rather be dropped to avoid overfitting.

In the beginning, we deliberately decided to include all the available features from our different data sources and even added some additional ones like bond rates and the return on share price to our dataset. Keeping in mind that by later evaluating the feature importance, we could specify a certain threshold for the features to be included or not. This is why we decided on specifying the threshold for the feature importance at 1% (drop all variables that explain less than 1%). While with this threshold 48 of the features are used, with for instance a 0.5% threshold 67 features would be used. Such a practice would increase the risk of overfitting. Therefore, we specified this threshold of 0.01 in all our algorithms.

After having finished all the preprocessing of the data as described, we saved it to a new csv-file, that can then be used by our chosen algorithms.

Machine Learning Algorithms

In the subsequent part our chosen algorithms are explained, and their performance is summarized.

Extreme Gradient Bosting Algorithm: The XGBoost algorithm is very popular at the moment and amongst the most preferred by many data scientists (Bhaskar Sundaram, 2018; Jain, 2016). It is an ensemble learning method, what implies that it combines different “weak learner” models into one “strong learner” model with the combined predictive power. The algorithm uses boosting. This boosting typically makes use of small decision trees (fewer splits compared to random forest) that are sequentially built and that aim to learn from their respective predecessors and reduce the residual errors of the model. The final model then reduces both variance as well as the bias (Bhaskar Sundaram, 2019). XGBoost has many advantages like regularization (that helps reducing overfitting), parallel processing and high flexibility because of the many different hyperparameters and built-in cross validation. Generally, it has a very high accuracy but doesn’t need too much time (Jain, 2016). We based upon common values for different hyperparameter in order to maximize the training score. In the end the algorithm worked convincingly well and gave us a training score of 99.3851% and a test score of 99.6796 %.

Random Forest: The Random Forest Algorithm grows numerous decision trees from a bootstrapped subsample of the training set. The predictions of the different trees are then summed up and the algorithm assigns the class label that the majority of trees predicted. Decision trees are a straightforward method, where a certain number of questions is asked (each node represents one question) to decide on the final class of our observations. Hence, postures a high risk of overfitting. In Random Forest, each tree is randomized in the sense that at each node a random subset of the features is preselected, and the node is then split according to the best performing feature of this random subset, what reduces the risk of overfitting. Random Forest often performs already pretty well with the default hyperparameters and even better after our hyperparameter tuning. A possible drawback is, that the higher the number of trees used, which increases the predictive power; the higher are the computational costs and the longer it takes to get a prediction (Zimmermann, 2019). After considering all this and tuning the hyperparameters we got a training score of 98.2188% and a test score of 98.6146%.

Support Vector Machine: As Support Vector Machines [SVM] are quite intuitive to understand, we decided to include this algorithm as well. A hyperplane (=decision boundary) is placed in the space to discriminate between the classes. This while maximizing the margin between the hyperplane and the observations closest by. Again, we used GridSearch to find the best parameters ("kernel" and for the chosen radial basis function kernel, "C" and "gamma") and finetune them. An advantage of SVM is, that they are running at a considerably faster speed than e.g. the XGBoost algorithm, while still performing well. We reached a training score of 97.5968% and a test score of 98.1340%.

As mentioned, we used the GridSearch function to find the best settings for the hyperparameters for each algorithm. The GridSearch function, maximized the training score while k-fold (we decided for k=5 after Breiman and Spector (1992) or Kohavi (1995), as we had a big dataset) validating its results. While tuning the parameters, we only looked at the training score to avoid overfitting. Only after reaching the maximum training score, we examined the test score, to check how well it performed on our test set. In the final version of our code we left our GridSearch empty and noted the sequence of the hyperparameter tuning and the best hyperparameters below the code, to reduce the running time.

We also did a principal component analysis [PCA] in random forest. The PCA tries to decrease complexity by reducing the dimensions of a dataset while still trying to capture as much as possible of the information, more precisely of the variation in the dataset. This is done by only choosing the "principal components" of the dataset that explain most of the variation in the dataset. This allows to shrink large datasets into several components. The output then shows how well the chosen "principal components" explain the variance ratio. Our PCAs did indeed show a consistent image trough all algorithms: With around 15 components, you could already explain 50% of the variance ratio. But compared to the polish bankruptcy data as shown by Zimmermann (2019), where one principal component explains more than 80% of the variance ratio, this is a low number.

Conclusion

We knew the preprocessing of the data was the most time-consuming part in our project, so we started immediately with the gathering and preparation of the data. We met at the very beginning to arrange who was going to do which task and plan further steps. After gathering all the necessary data, what didn't take too much time, we decided that we all focus on preprocessing the data and handling missing values in order to have the data ready for further processing as fast as possible. We then met on a weekly basis to discuss further actions. Thanks to this, our working process was very well structured and there were almost no duplications.

After merging the data, where we didn't have a lot of difficulties but needed to pay attention to the different column names in the different datasets to merge, it took us almost until the end of March to preprocess the data, where we had quite some trouble with deciding how to impute our values, with making the MICE imputer work and with deciding how to deal with class imbalance. After spending a lot of time and effort experimenting with the SimpleImputer, MICE, the resampler and SMOTE, we finally got MICE and SMOTE running and also decided how to impute the columns where we didn't want to use MICE. We also spent quite some time thinking of a good split for our classes and about how many classes we should use in order to have large enough samples. In the end after some discussion, we settled on using 5 different classes, also because we didn't have enough observations for the low rating classes.

After having finished the preparation of the dataset we could finally start trying out the algorithms. We already knew that we needed to use Random Forest, so we had to determine our other two algorithms. For that, we focused on two algorithms we didn't encounter in the lecture, namely the XGBoost and Neural Network. For the Neural Network we followed the book "Make Your Own Neural Nework" by Tariq Rashid (2016).
In early April, we got the first results and started to conclude our work. After playing around for some time, we achieved a good score for XGBoost, but we decided to stop our attempt of making Neural Network work, because we could see that it would take too much time to focus on two new algorithms, and we progressed faster with XBGoost. That's why we decided to instead use another one of the algorithms that we came across in the lecture and chose to use SVM. It performed way better than our early Neural Network tryouts and didn't use as much time to run as XGBoost did, that is why it was a very good third algorithm for us. Here, we spent most of the time tuning the hyperparameters of our algorithms with GridSearch and finding the best specifications in order to maximize our score. Especially for the XGBoost algorithm we needed a long time to finetune it, because it took a lot of computational power and time to run.

Eventually, our scores came out even better than we expected for all our algorithms, XGBoost being the best performing algorithm with a train score of more than 98%. One reason for these high scores could be, that we spent lots of time with good imputation methods and we also up sampled our dataset enormously with SMOTE. Still, our test scores don't show any sign of overfitting, that's why we're pretty confident with our results.

In the end we spent the rest of the available time on optimizing our scores, presenting our results in an appealing way by preparing PCA in Random Forest and a performance report and a confusion matrix for each of our algorithms and by writing the abstract and preparing our presentation. If we would have had more time, we would have definitely tried out to somehow plot the ROC-Curve and calculate the AUC for our multiclass problem. Additionally, we would have spent more time with getting good train scores for Neural Networks, because we are pretty sure that this algorithm would work well for a project like that if more time was available for engaging ourselves with this method.

Overall, we worked well together as a team, especially when one struggled with coding, the others would try to help and overcome the difficulty. This worked out well and we didn't spend too much time on trivialities. Everyone did their part and we never had any time pressure. Also, the team members where very motivated to do more than just the minimal requirements and to try and include stuff that we didn't encounter in the lecture. The most we struggled with the MICE imputer, where it took us several weeks to make it work, but in the end, it was very rewarding. Also, in the aggregate, we spent several hours seeking the internet for solutions to debug small errors in our code. If the time would've had allowed it, we would have tried to implement additional algorithms and methods into our model.

In closing, we can say that we put a fair amount of work into our project, but we feel like we learnt a lot from the task that might come in handy in future projects. Now we have a good overview of different existing algorithms and other machine learning tools and we were able to practice coding in Python quite extensively with all the difficulties that it involves.

Appendix

Following the performance reports for the three algorithms as shown in the code.

Performance Report Extreme Gradient Boosting Algorithm

```
#################################################################

PERFORMANCE REPORT XGBOOST

#################################################################

-----------------------------------------------------------------
METRICS
-----------------------------------------------------------------
Overall Accuracy score:  99.68%
Overall Error rate:      0.32%
-----------------------------------------------------------------
------------  ------  ------  ------  ------  ------
Rating Class  0       1       2       3       4
Score         0.9996  0.9992  0.998   0.9975  0.9992
Error-rate    0.0004  0.0008  0.002   0.0025  0.0008
Specifit      0.9998  0.9996  0.9992  0.9979  0.9995
Sensivity     0.9991  0.9976  0.9935  0.9962  0.9977
Precision     0.9991  0.9986  0.9967  0.9914  0.9981
F1 (beta=1)   0.9991  0.9981  0.9951  0.9938  0.9979
------------  ------  ------  ------  ------  ------
-----------------------------------------------------------------
CONFUSION MATRIX
-----------------------------------------------------------------
[[2123    2    0    0    0]
 [   2 2091    3    0    0]
 [   0    1 2125   13    0]
 [   0    0    4 2087    4]
 [   0    0    0    5 2151]]
-----------------------------------------------------------------
CLASSIFICATION REPORT
-----------------------------------------------------------------
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00      2125
         1.0       1.00      1.00      1.00      2096
         2.0       1.00      0.99      1.00      2139
         3.0       0.99      1.00      0.99      2095
         4.0       1.00      1.00      1.00      2156

   micro avg       1.00      1.00      1.00     10611
   macro avg       1.00      1.00      1.00     10611
weighted avg       1.00      1.00      1.00     10611

#################################################################
```

Performance Report Random Forest

```
#################################################################

PERFORMANCE REPORT RANDOM FOREST

#################################################################

-----------------------------------------------------------------
METRICS
-----------------------------------------------------------------
Overall Accuracy score:  98.61%
Overall Error rate:      1.39%
-----------------------------------------------------------------
------------  ------  ------  ------  ------  ------
Rating Class  0       1       2       3       4
Score         0.9985  0.9941  0.9888  0.9926  0.9983
Error-rate    0.0015  0.0059  0.0112  0.0074  0.0017
Specifit      0.9993  0.9966  0.9933  0.9947  0.9988
Sensivity     0.9953  0.9838  0.971   0.9842  0.9963
Precision     0.9972  0.9861  0.9733  0.9786  0.9954
F1 (beta=1)   0.9962  0.985   0.9722  0.9814  0.9958
------------  ------  ------  ------  ------  ------
-----------------------------------------------------------------
CONFUSION MATRIX
-----------------------------------------------------------------
[[2115   10    0    0    0]
 [   3 2062   30    1    0]
 [   3   16 2077   39    4]
 [   0    3   24 2062    6]
 [   0    0    3    5 2148]]
-----------------------------------------------------------------
CLASSIFICATION REPORT
-----------------------------------------------------------------
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00      2125
         1.0       0.99      0.98      0.98      2096
         2.0       0.97      0.97      0.97      2139
         3.0       0.98      0.98      0.98      2095
         4.0       1.00      1.00      1.00      2156

   micro avg       0.99      0.99      0.99     10611
   macro avg       0.99      0.99      0.99     10611
weighted avg       0.99      0.99      0.99     10611

#################################################################
```

Performance Report Support Vector Machine

```
################################################################

PERFORMANCE REPORT Support Vector Machine

################################################################

----------------------------------------------------------------
METRICS
----------------------------------------------------------------
Overall Accuracy score:  98.13%
Overall Error rate:      1.87%
----------------------------------------------------------------
------------  ------  ------  ------  ------  ------
Rating Class  0       1       2       3       4
Score         0.9991  0.9941  0.984   0.9873  0.9983
Error-rate    0.0009  0.0059  0.016   0.0127  0.0017
Specifit      0.9993  0.9958  0.9911  0.9919  0.9986
Sensivity     0.9981  0.9871  0.9556  0.9685  0.9972
Precision     0.9972  0.9829  0.9646  0.9671  0.9944
F1 (beta=1)   0.9976  0.985   0.9601  0.9678  0.9958
------------  ------  ------  ------  ------  ------
----------------------------------------------------------------
CONFUSION MATRIX
----------------------------------------------------------------
[[2121    3    1    0    0]
 [   5 2069   20    2    0]
 [   0   30 2044   65    0]
 [   1    2   51 2029   12]
 [   0    1    3    2 2150]]
----------------------------------------------------------------
CLASSIFICATION REPORT
----------------------------------------------------------------
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00      2125
         1.0       0.98      0.99      0.99      2096
         2.0       0.96      0.96      0.96      2139
         3.0       0.97      0.97      0.97      2095
         4.0       0.99      1.00      1.00      2156

   micro avg       0.98      0.98      0.98     10611
   macro avg       0.98      0.98      0.98     10611
weighted avg       0.98      0.98      0.98     10611

################################################################
```

Bibliography

Azur, M. J., Stuart, E. A., Frangakis, C., & Leaf, P. J. (2011). Multiple imputation by chained equations: what is it and how does it work?. *International journal of methods in psychiatric research*, *20*(1), 40–49. doi:10.1002/mpr.329.

Bhaskar Sundaram, R. (September 6, 2018). *An End-to-End Guide to Understand the Math behind XGBoost.* Retrieved April 10, 2019, from https://www.analyticsvidhya.com/blog/2018/09/an-end-to-end-guide-to-understand-the-math-behind-xgboost/.

Board of Governors of the Federal Reserve System. (2019). *1-Year Treasury Constant Maturity Rate*. Retrieved April 9, 2019, from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/DGS1.

Board of Governors of the Federal Reserve System. (2019). 5-Year Treasury Constant Maturity Rate, Retrieved April 9, 2019, from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/DGS5.

Board of Governors of the Federal Reserve System. (2019). 10-Year Treasury Constant Maturity Rate, Retrieved April 9, 2019, from FRED, Federal Reserve Bank of St. Louis; https://fred.stlouisfed.org/series/DGS10.

Breiman, L., & Spector, P. (1992). Submodel Selection and Evaluation in Regression. The X-Random Case, *International Statistical Review* 60, 291-319.

Jain, A. (March 1, 2016). *Complete Guide to Parameter Tuning in XGBoost (with codes in Python).* Retrieved April 10, 2019, from https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/.

Kohavi, R. (1995). A study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection, in *International Joint Conference on Artificial Intelligence (IJCAI),* 1137-1145, Stanford, CA.

Kunert, R. (2017). *SMOTE explained for noobs – Synthetic Minority Over-sampling TEchnique line by line.* Retrieved April 9, 2019, from http://rikunert.com/SMOTE_explained.

Standard & Poor’s Financial Services. (2019). *Understanding Ratings.* Retrieved April 10, 2019, from https://www.spratings.com/en_US/understanding-ratings?rd=understandingratings.com#firstPage.

Tariq, R. (2016). *Make Your Own Neural Network*. North Charleston: Createspace Independent Publishing Platform.

World Bank. (2019). *Inflation, consumer prices for the United State*s. Retrieved April 9, 2019, from FRED, Federal Reserve Bank of St. Louis; *https://fred.stlouisfed.org/series/FPCPITOTLZGUSA.*

Zimmermann, B. (2019). *Machine Learning in Finance – A gentle Introduction with a Focus on Application in Python.*