Ryan Herrin, Luke Stodgel
DS7333
2/24/2023

Bankruptcy Prediction using Random Forest and Extreme Gradient Boosting

Introduction

When a company goes bankrupt it can impact a lot of people: the company's employees and their families, investors and entire communities. Therefore predicting bankruptcy can be a useful tool for all interested parties and businesses.

In this project, we created Random Forest and Extreme Gradient Boosting models to predict bankruptcy.

Methods

Our dataset contained a range of economic attributes. The shape of the dataset was 43,406 rows by 65 columns. Our target variable, named "target", represented whether or not a company went bankrupt. A 0 value for our target variable meant the company did not go bankrupt, and a 1 meant the company went bankrupt.

Here is a list of a few of the columns and their meanings (please find the full list of features in the appendix):

X1 net profit / total assets
X2 total liabilities / total assets
X3 working capital / total assets
X4 current assets / short-term liabilities
X5 [(cash + short-term securities + receivables - short-term liabilities) / (operating expenses - depreciation)] 365

Preface on Our Model Building and Testing

In total we created six models, three Random Forest and three XGBoost. We did this so that we could compare and contrast the models and find a "final, best" model. We started by making RF and XGB models that only used the default parameters so that we could get baseline metrics. Having baseline metrics allowed us to know if our parameter tuning helped our models' performance or not. Next, after we found the best parameters for both RF and XGB using RandomGridSearchCV, we made two models for each algorithm. One that used a best_threshold and predicted probabilities and another that used traditional prediction functions. We will report on the results of all of these models in the results section but in the conclusion we will only talk about our best RF and XGB models.

Creating the Dataset

The data we started with was in 5 separate files where each file represented 1 year of data. The top of each file had the attributes and what type the attribute was, but the attribute name did not describe what the feature was. Only the last attribute was defined as ‘class {0,1}’ and we used this as our target class. For the target class attributes, 0 represented a company that did not go bankrupt and 1 represented a company that went bankrupt. To create the initial dataset we read in each file as a python list. Then, we searched through the list to find the index value that contained ‘@data’ which is where the actual financial data started. Then, all lines after that were added to a master list containing all values for each year. Then we transformed that master list of financial data into a Pandas Dataframe.

Addressing the Missing Data

After creating our dataframe, we addressed the missing values in the data. Every feature had missing data and from an initial investigation the data seemed to be missing completely at random. The most notable feature was X21 (sales (n) / sales (n-1)) with 13.5% missing and X37 ((current assets - inventories) / long-term liabilities) with 43.7% missing. We decided to impute the data using scikit-learn's KNNImputer function since all of the data was numeric. The KNNImputer attempts to replace the missing values with the mean of the surrounding neighbors in the dataset. We only ended up having to remove one row because the target did not have a value, and KNN would have given us an estimate instead of the binary classification that we needed. This KNN imputation method worked out very well for us as shown through our model metrics in the results section.

RandomSearchCV vs. GridSearchCV

We opted to use RandomSearchCV over GridSearchCV for a few different reasons. One reason was because there were several different parameters to test for both the Random Forest and XGBoost models. With that many parameters, GridSearchCV would have had to fit many more models and would have taken significantly longer to run than RandomSearchCV. Another reason why we opted to use RandomSearchCV was because of the size of our data set. Because our data was 43,406 rows by 65 columns, the models took a very long time to train. This combination of having to test several different parameters and also having a long and wide data set made RandomSearchCV the obvious winning option in this project.

We understand that GridSearchCV would have probably found a better parameter combination than RandomSearchCV, but the amount of time we saved looking for those parameters is worth having our models’ performance decrease slightly.

RandomSearchCV for our Random Forest Model

We used RandomSearchCV to find the best parameters for our models. Our random forest randomsearchcv parameter grid looked like this:

```
param_grid = {
    'criterion': ['gini','entropy'],
    'n_estimators': [50, 100, 200, 500],
    'max_features': ['sqrt', 'log2', None],
    'max_depth': [3, 5, 10, 15],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'class_weight': ['balanced', None]
}
```

We also used the parameters: scoring='accuracy', n_iter=5, cv=5 and n_jobs=-1. We picked accuracy as our scoring metric because we wanted our accuracy as high as possible to compete with the models from the research paper we were given, titled, "Ensemble boosted trees with synthetic features generation in application to bankruptcy prediction".

The best parameters returned by our RF randomsearchcv were: 'n_estimators': 100, 'min_samples_split': 10, 'min_samples_leaf': 1, 'max_features': None, 'max_depth': 15, 'criterion': 'entropy', 'class_weight': None. Lastly, using the best model returned by the randomsearchcv along with a score(X_test, y_test) function we received a score (average accuracy) of 0.963.

RandomSearchCV for XGBoost

For our XGBoost RandomSearchCV, our parameter grid looked like this:

```
param_grid = {
      'eta': [0.1, 0.3, 0.5, 0.7, 1.0],
      'max_depth': [3, 5, 7, 9],
      'learning_rate': [0.05, 0.10, 0.15, 0.20, 0.25, 0.30],
      'subsample': [0.6, 0.7, 0.8, 0.9, 1.0],
      'n_estimators': [50, 100, 500],
      'gamma': [0, 1, 10, 50, 100],
      'eval_metric': ['auc', 'error'],
      'objective':['binary:logistic'],
      'colsample_bytree': [0.5, 0.7, 1.0],
      'reg_alpha': [0.0, 0.1, 0.5, 1.0],
      'reg_lambda': [0.0, 0.1, 0.5, 1.0],
      'scale_pos_weight': [1, 5, 10, 50]
}
```

We also used parameters n_iter=5, scoring='accuracy', cv=5, n_jobs=-1. We picked accuracy as our scoring metric for the same reason as our RF randomsearchcv, to get the highest accuracy to compete with the models described in the research paper we were given in class.

The best parameters returned by our XGB randomsearchcv were: 'subsample': 0.8, 'scale_pos_weight': 10, 'reg_lambda': 1.0, 'reg_alpha': 1.0, 'objective': 'binary:logistic', 'n_estimators': 500, 'max_depth': 7, 'learning_rate': 0.05, 'gamma': 1, 'eval_metric': 'error', 'eta': 0.5, 'colsample_bytree': 0.5. Using the best model returned by the randomsearchcv, we used a score(X_test, y_test) function and we received a score (average accuracy) of 0.969.

Scaling our Data

We scaled our data using StandardScaler because for XGB models it can improve convergence of the optimization algorithm, speed up the training process, normalize the data to make it more suitable for the assumptions of XGB and will reduce the impact of outliers if there are any. Creating RF and XGB models using scaled data can be beneficial in terms of improving stability and the interpretability of them and it allows for easier and more consistent reproducibility of these types of models.

We understand that RF models are non-parametric and don't usually benefit from using scaled data, but we still decided to use the same scaled data we used for our XGB model with our RF model. In the end, scaling the data didn't end up making any difference on the performance of any of the models.

Prediction With or Without using Predicted Probabilities?

In our project we tuned four models: random forest using predicted probabilities, xgb using predicted probabilities, random forest using traditional binary prediction, and xgb using traditional binary prediction. Out of these four models we picked the best performing random forest and xgb models and used them as our final models.

To find the best thresholds for our predicted probabilities models, we created separate find_best_threshold functions, one for RF and another for XGB. Within them we created train-test splits, used the best_params from our RandomGridSearchCV, found predicted probabilities and then tested thresholds from 0.01 to 1.0, in increments of 0.01. We picked the best threshold based on the model that returned the highest accuracy. The reason why we picked the model based on accuracy as opposed to any other metric is because our problem statement says to tune our models to maximize accuracy and also when we tested finding the best threshold based on f1_score, the model metrics did not noticeably change.

Unbalanced Target Variable Classes

In our dataset, the target variable was extremely unbalanced. 95% or 41315/43406 entries were marked "0" meaning no bankruptcy, while the other 5% were marked "1" meaning yes bankruptcy. This is important to mention because it explains why the recall in some of our models were so low; the models struggled to predict true positives. A few of our models had less than 10% recall while having 90% precision and 95% accuracy. Some of our other models were able to achieve 60% recall and 60% precision while still maintaining 95% accuracy. Because of the massive imbalance of the classes in the target variable, it is expected that our models would guess "0" almost all of the time, but because of how serious bankruptcy is, we opted to pick our final models as the models with higher recall, even if it meant picking the model with 1% lower accuracy score.

Results

As shown in Figure 1 we obtained a best_threshold of 0.298 for our Random Forest model. This is important because this threshold allowed us to create a higher performing model than our RF model that made predictions without using predicted probabilities. We picked the highest threshold based on the highest accuracy which, again, was threshold = 0.298.

Prediction Probabilities Threshold Scores for RF

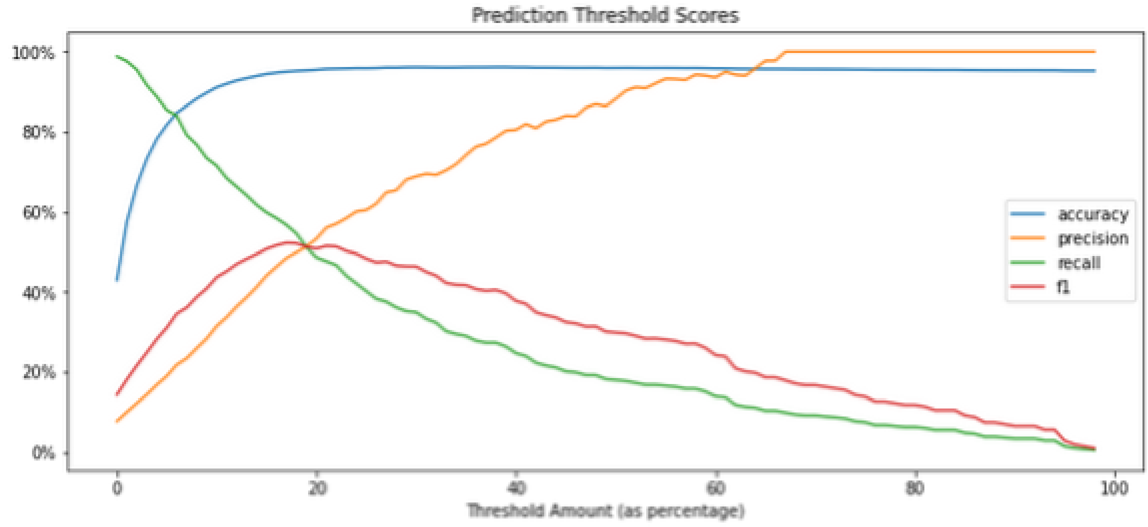


Figure 1 - Prediction Threshold Scores for Random Forest

In Figure 2 we obtained a best_threshold of 0.564 for our XGBoost model. Our XGB model that used predicted probabilities also outperformed our XGB model that used a traditional binary prediction function. We picked the highest threshold based on the highest accuracy which, again, was threshold = 0.564.

Prediction Probabilities Threshold Scores for XGB

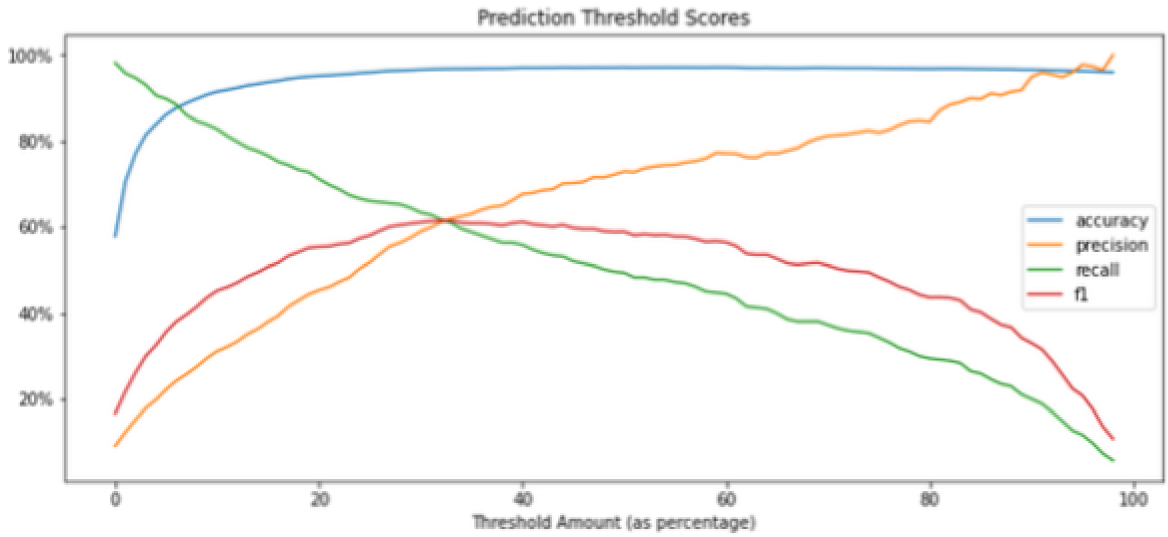
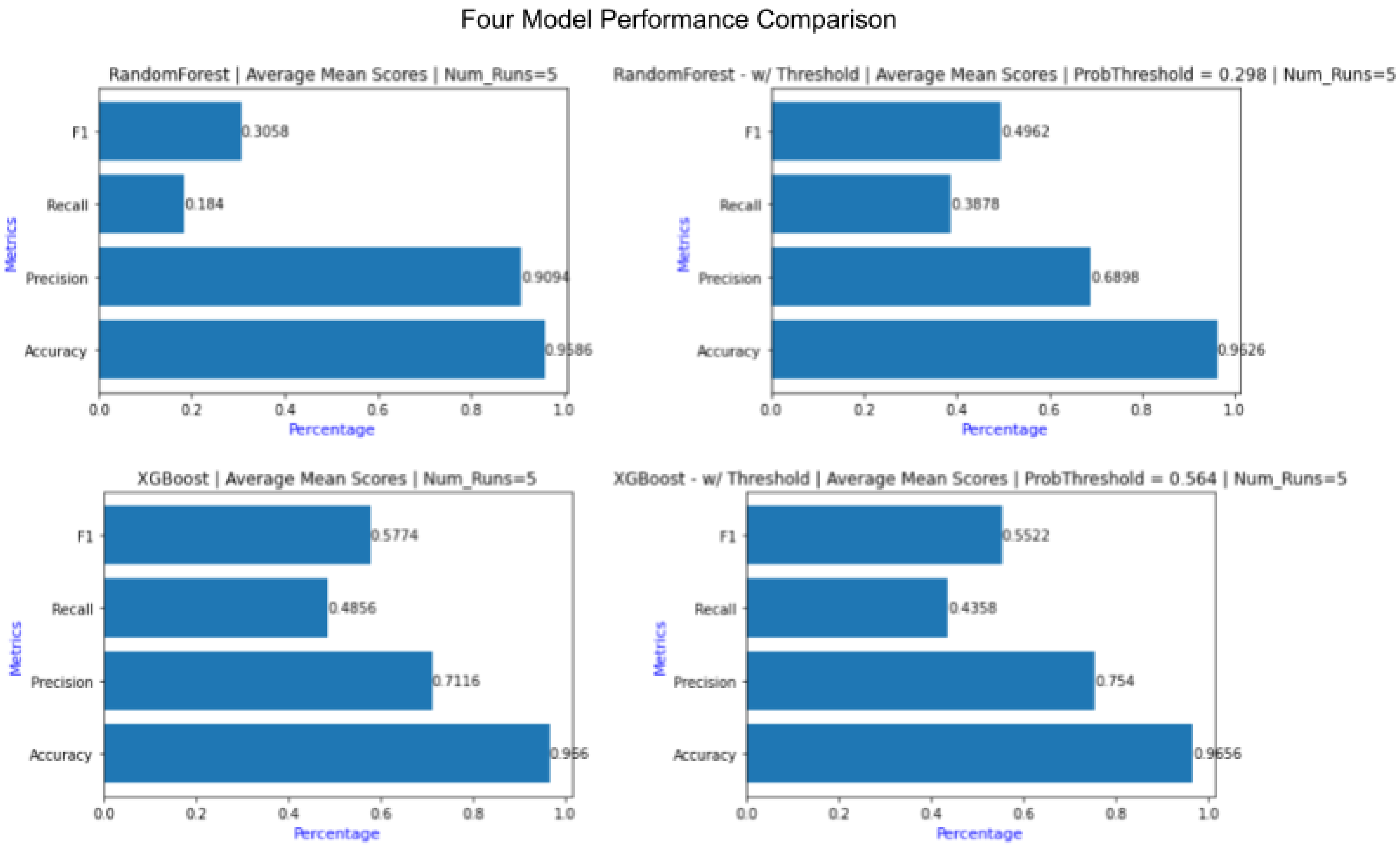


Figure 2 - Prediction Threshold Scores for XGBoost

Below in Figure 3 we have a comparison of mean metrics of the four models we built. As you can see, the accuracy between the RF and XGB models were very similar. But, the XGB models certainly had higher recalls by 5-10% and f1_scores by 5-8% compared to the RF models, making them overall better models. In this study, achieving a high recall without causing precision to suffer was a real challenge. We believe this challenge was caused due to the large imbalance of positive and negative classes in our target variable. We tried to mitigate the imbalance in our XGBoost model by utilizing the scale_pos_weight parameter, but the results only increased slightly. And for our RF model we included class_weight in the RandomSearchCV parameter grid, but it ended up suggesting we use 'None' instead of 'balanced' for that parameter. It was equally as difficult to obtain a high f1_score for these same reasons.



Four Model Performance Comparison

RandomForest | Average Mean Scores | Num_Runs=5

RandomForest - w/ Threshold | Average Mean Scores | ProbThreshold = 0.298 | Num_Runs=5

XGBoost | Average Mean Scores | Num_Runs=5

XGBoost - w/ Threshold | Average Mean Scores | ProbThreshold = 0.564 | Num_Runs=5

Figure 3 - RF and XGB Predict() vs Predict_proba() Models

Figure 4 (below) shows boxplots describing model results from fitting 5 models of each type with different train/test splits each time. The orange lines within the boxes represent each metric’s mean, while the box and whiskers represent each metric’s variance. The smaller the variance, the more confident we can be in our model’s consistency. As you can see, the variance for each metric for each of the models is very small, hinting that our models had consistent results.

Some notable points here are that every model’s accuracy is 95% or above and the models that used predicted probabilities had stronger recall and f1_scores than the models that used traditional binary prediction functions.

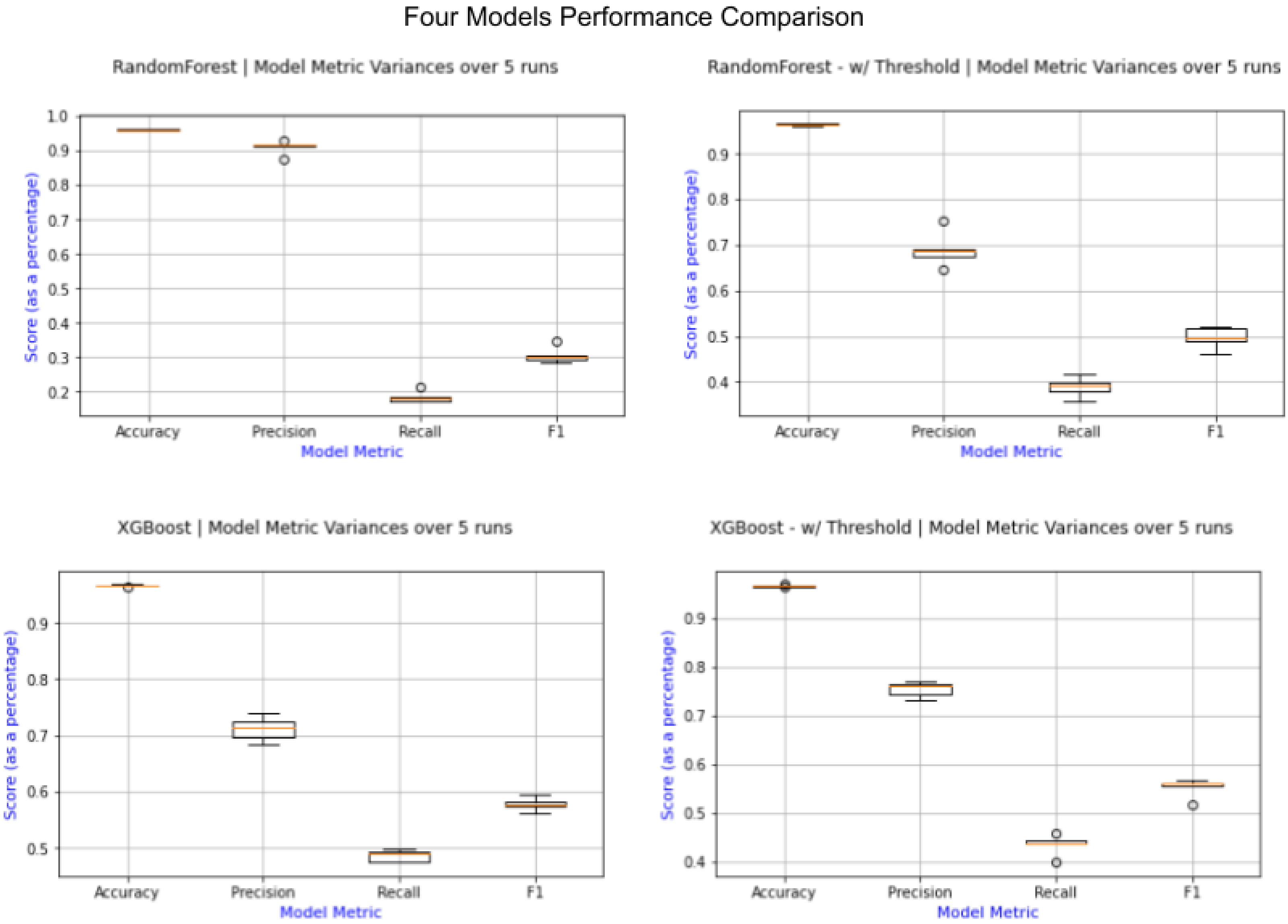


Figure 4 - RF and XGB Predict() vs Predict_proba() Models

Figure 5 (below) shows the metrics for our final models. The metrics of the RF and XGB models that used predicted probabilities and an optimal threshold outperformed both other RF and XGB models that used traditional binary prediction functions.

Our final RandomForest model received an average f1_score of 0.496, recall of 0.388, precision of 0.689 and an accuracy of 0.963.

Our final XGB model received an average f1_score of 0.552, recall of 0.436, precision of 0.754, and an accuracy of 0.966.

To our surprise, the metrics of these models were very similar but the XGB still had stronger metrics with 5.6% higher f1_score, 4.8% higher recall, 6.5% higher precision and 0.3% higher accuracy.

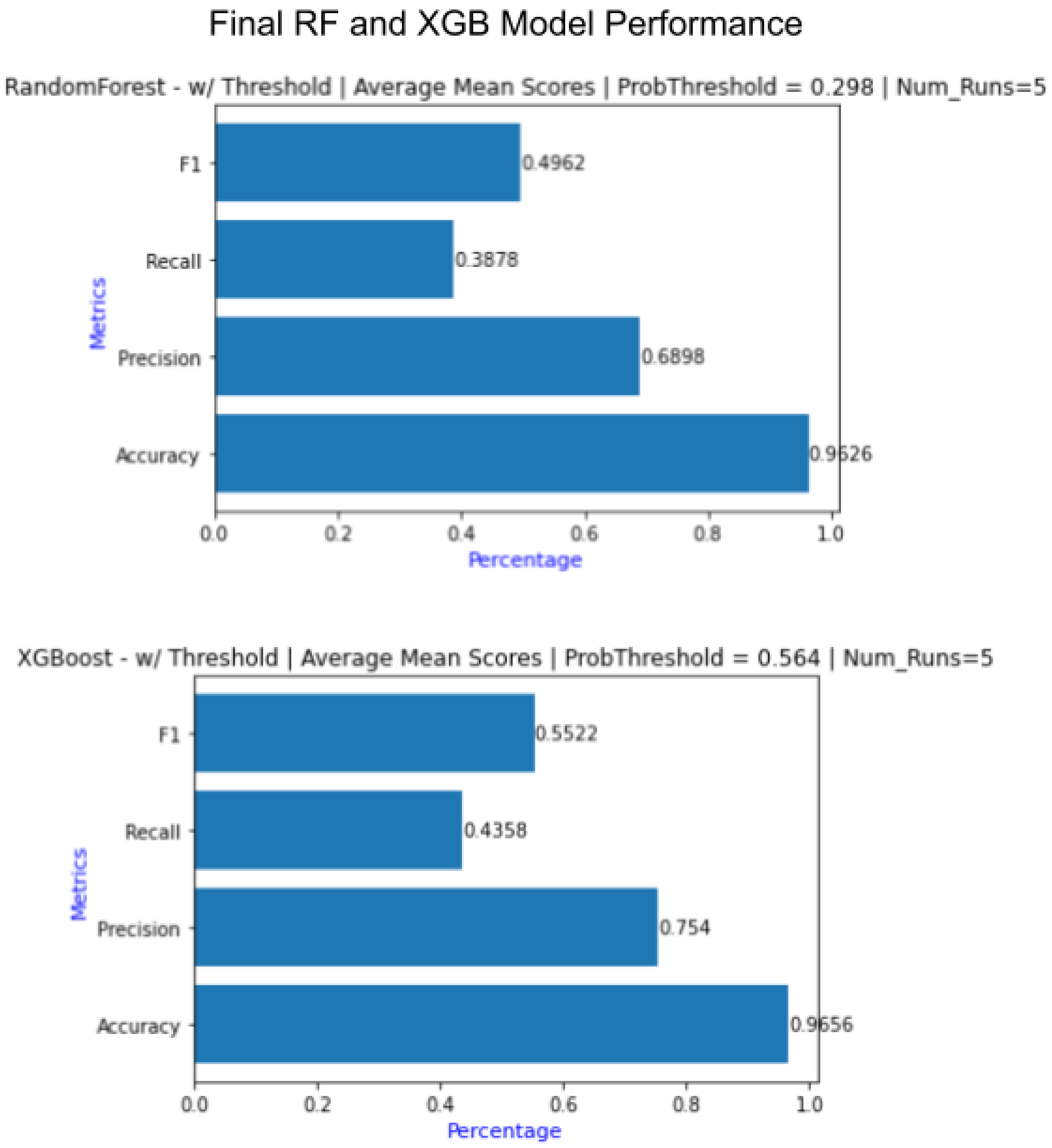


Figure 5 - RF and XGB Predict_proba() Models

Below are descriptions of the results based on Figure 4 for our final models only:

Accuracy: RF: Mean = 96.3% | Variance = <1% | XGB: Mean = 96.6% | Variance = <1%

Accuracy is determined by the number of correctly predicted instances of the total number of instances. Our model had very high accuracy, and made very accurate predictions. We tuned our parameters around the model that returned the highest accuracy score. We also tuned our best threshold around the model that returned the highest accuracy. Our purpose behind this was because we were trying to get the highest accuracy possible so that we could results to the research paper, “Ensemble boosted trees with synthetic features generation in application to bankruptcy prediction” by Maciej Zieba, Sebastian K. Tomczak, and Jakub M. Tomczak.

Precision: RF: Mean = 68.9% | Variance = <1% | XGB: Mean = 75.4% | Variance = <1%

The equation for precision is: (true positives / (true positives / false positives)). The more false positives there are, the lower the precision score. In this project we tried to find the best balance between precision and recall and in doing so, precision typically decreased as recall increased. Our final XGB model outperformed our final RF model by 6.5%.

Recall: RF: Mean = 38.8% | Variance = <1% | XGB: Mean = 43.6% | Variance = <1%

Recall is similar to precision except it measures false negatives rates instead of false positive rates. The equation for recall is: (true positives / (true positives + false negatives)). In the context of this project, a lower recall would mean that more companies that would go bankrupt (1) were predicted to not go bankrupt (0) instead. Of our two final models, the XGB model performed better than the RF model by 4.8%.

F1_Score: RF: Mean = 49.6% | Variance = <1% | XGB: Mean = 55.2% | Variance = <1%

F1_score is the harmonic mean of precision and recall. The equation for f1_score is this: 2 * (precision * recall) / (precision + recall). The XGB model outperformed the RF model overall in precision and in recall so it stands to reason that the F1 score of the XGB model was better. Again, our objective was to achieve the highest accuracy, but with both models having close accuracy metrics the F1 score shows the XGB model outperformed the RF model.

Conclusion

In the end our Random Forest and XGB models performed very well. The mean accuracy for our final XGB model was .966 and the mean accuracy for our final RF model was .953. Our accuracy scores were higher than the scores of the models from the paper, “Ensemble boosted trees with synthetic features generation in application to bankruptcy prediction” by Maciej Zieba, Sebastian K. Tomczak, and Jakub M. Tomczak. Their mean accuracy score for their models were: 0.934 for their XGB model and 0.854 for their RF model.

The parameters we used to create our final XGB model were: 'subsample': 0.8, 'scale_pos_weight': 10, 'reg_lambda': 1.0, 'reg_alpha': 1.0, 'objective': 'binary:logistic', 'n_estimators': 500, 'max_depth': 7, 'learning_rate': 0.05, 'gamma': 1, 'eval_metric': 'error', 'eta': 0.5, 'colsample_bytree': 0.5.

The parameters we used to create our final RF model were: 'n_estimators': 100, 'min_samples_split': 10, 'min_samples_leaf': 1, 'max_features': None, 'max_depth': 15, 'criterion': 'entropy', 'class_weight': None.

Since the accuracy of our RF model was on par with our XGB model it makes us think that we didn't obtain the optimal parameters through using RandomSearchCV. In the future, if anyone were to try to replicate this procedure, and if time permits it, we recommend using GridsearchCV to find the optimal parameters before building your models.

Appendix A

Link to main script:
https://github.com/Abillelatus/QTW-Case-Studies/blob/main/CaseStudy_4/MSDS-7333-CaseStudy-4.py

Link to full CaseStudy_4 full contents:
https://github.com/Abillelatus/QTW-Case-Studies/tree/main/CaseStudy_4

Full list of columns:
X1 net profit / total assets
X2 total liabilities / total assets
X3 working capital / total assets
X4 current assets / short-term liabilities
X5 [(cash + short-term securities + receivables - short-term liabilities) / (operating expenses - depreciation)] 365
X6 retained earnings / total assets
X7 EBIT / total assets
X8 book value of equity / total liabilities
X9 sales / total assets
X10 equity / total assets
X11 (gross profit + extraordinary items + financial expenses) / total assets
X12 gross profit / short-term liabilities
X13 (gross profit + depreciation) / sales
X14 (gross profit + interest) / total assets
X15 (total liabilities * 365) / (gross profit + depreciation)
X16 (gross profit + depreciation) / total liabilities
X17 total assets / total liabilities
X18 gross profit / total assets
X19 gross profit / sales
X20 (inventory * 365) / sales
X21 sales (n) / sales (n-1)
X22 profit on operating activities / total assets
X23 net profit / sales
X24 gross profit (in 3 years) / total assets
X25 (equity - share capital) / total assets
X26 (net profit + depreciation) / total liabilities
X27 profit on operating activities / financial expenses
X28 working capital / fixed assets

X29 logarithm of total assets
X30 (total liabilities - cash) / sales
X31 (gross profit + interest) / sales
X32 (current liabilities * 365) / cost of products sold
X33 operating expenses / short-term liabilities
X34 operating expenses / total liabilities
X35 profit on sales / total assets
X36 total sales / total assets
X37 (current assets - inventories) / long-term liabilities
X38 constant capital / total assets
X39 profit on sales / sales
X40 (current assets - inventory - receivables) / short-term liabilities
X41 total liabilities / ((profit on operating activities + depreciation) * (12/365))
X42 profit on operating activities / sales
X43 rotation receivables + inventory turnover in days
X44 (receivables * 365) / sales
X45 net profit / inventory
X46 (current assets - inventory) / short-term liabilities
X47 (inventory * 365) / cost of products sold
X48 EBITDA (profit on operating activities - depreciation) / total assets
X49 EBITDA (profit on operating activities - depreciation) / sales
X50 current assets / total liabilities
X51 short-term liabilities / total assets
X52 (short-term liabilities * 365) / cost of products sold)
X53 equity / fixed assets
X54 constant capital / fixed assets
X55 working capital
X56 (sales - cost of products sold) / sales
X57 (current assets - inventory - short-term liabilities) / (sales - gross profit - depreciation)
X58 total costs /total sales
X59 long-term liabilities / equity
X60 sales / inventory
X61 sales / receivables
X62 (short-term liabilities * 365) / sales
X63 sales / short-term liabilities
X64 sales / fixed assets