Predicting Bankruptcy using Random Forest and Extreme Gradient Boosting

1. Introduction

Using supervised machine learning to assist in predicting bankruptcies advances one of the most fundamental tasks in risk assessment that ensures business continuity with partners, investors, lenders, and other sector related entities. Since the economic collapse in 2008, there has been a greater need to predict and avoid bankruptcy. As evident from that crisis, bankruptcies of large and even small corporations can trigger a domino effect that can bring financial markets to a halt.

2. Data

The Finance Department gave the Data Science Department a collection of historical data with 64 financial data attributes and 1 response variable named "class". The task at hand given to the team is to predict in the future that a company might go bankrupt, so that precautions can be taken ahead of time. The team was also informed that this historical data was collected over five years, separated into five .arff files, which consists of 43,405 entries, and the response variable is binary and conveys either bankruptcy or non-bankruptcy. The head of the department was asked for advice, to which two powerful supervised models Random Forest and Extreme Gradient Boosting were recommended to be built for a classification problem such as this one, where the latter is proved to insensitive to and forces proper ordering of imbalanced data (Zieba, K. Tomczak & M. Tomczak, 2016).

To simulate data science in real life, the test set needs to be clean of all data processing (e.g., imputations, and normalizing) and must be left alone until the validation stage (e.g., scoring by cross-validation). The common ratio of 80:20 will be used in this project (80% training, 20% test), and the meaning of life is set as the seed for the entirety of the project.

2.1. Imputation

After some exploratory analysis, two problems have been found with this data so far. Fortunately, there are no attributes with near 90% missing values, however "Attr37" and "Attr21" represent 43.74% and 13.49% missing data, respectively. No information was given on any attribute, i.e., we know nothing about those two attributes despite them representing such a percentage of missing data and removing them without knowing their importance domain wise is a bad idea.

It is difficult to implement the most efficient imputation method for the size of this data. However, for the sake of simplicity which will expedite the launch of this project's results, it makes more sense to apply median imputations for missing data related to economics such as the unknown attributes given in this data, or data that does not have a limit e.g., income. The counter argument would be that it's more fitting to apply mean imputations for missing data that has a fixed range, like wear of clothing or temperature. With that said, the given data is most likely to have outliers due to said nature, therefore `SimpleImputer(strategy='median')` and `RobustScaler()` are used to address the missing data.

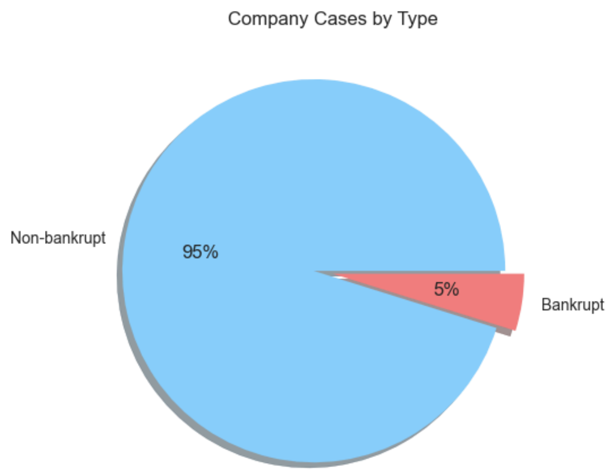


Figure 1. This distribution of company cases by type is extremely imbalanced. This can be solved by using interventions to fix weights such as `class_weight` in RandomForest and `scale_pos_weight` in XGBoost.

3. Random Forest Classifier

A randomized search cross-validation is initially performed and fitted for the 80% training data.

3.1. Hyperparameters

The user-defined cross-validation strategy is set to `RepeatedStratifiedKFold()` class with 10 folds and 3 repeats. The list below are the input parameters used in `RandomizedSearchCV()` which ran for 60 iterations, and the following Table 1 shows the chosen hyperparameters that produced the best cross-validated Random Forest model in terms of Area Under the Curve (AUC) out of the 60 randomized parameter settings. These chosen hyperparameters are then used as input parameters for `RandomForestClassifier()`. Likewise, the resulting fitted model is evaluated again by `RepeatedStrafifiedKFold()` with 10 folds and 3 repeats via `cross_val_score()` on the 20% test set, which was mentioned earlier to be left out until this stage, in terms of mean and standard deviation AUC, as seen in Section 5 Table 2, separately for each of the five years.

Initial Hyperparameters:

- "n_estimators": [1, 2, 3, 4, 5, 7, 10, 12, 13, 14, 15, 17, 20]
- "criterion": ["gini", "entropy", "log_loss"]
- "max_depth": [3, 5, 7, None]
- "min_samples_split": sp_randint(2, 11)
- "min_samples_leaf": sp_randint(1, 5)

- "max_features": sp_randint(1, 11)
- "bootstrap": [True, False]
- "class_weight": [“balanced”, “balanced_subsample”]

RandomizedSearchCV Results

Hyperparameters	*Values*
n_estimators	20
criterion	gini
max_depth	None
min_samples_split	5
min_samples_leaf	2
max_features	10
bootstrap	True
class_weight	balanced

Table 1. The resulting hyperparameters for RandomForest RandomizedSearchCV that followed the preprocessing methods of this project.

3.2. Finetuning

The best performing Random Forest model can have its results further finetuned by adjusting the prediction threshold. However, this was done with the entire data of all five years collective for sake of simplicity. To get a starting point, a graph was produced to observe the tradeoff between precision and recall seen in Figure 2, where the most optimal range of thresholds that favor recall can be seen nearer 0.0 than 0.2. A trial-and-error process is done for each sequential run of the finetuning, starting from threshold 0.1 with increments of 0.05. The sequential finetuning process for Random Forest can be seen in Table 3.

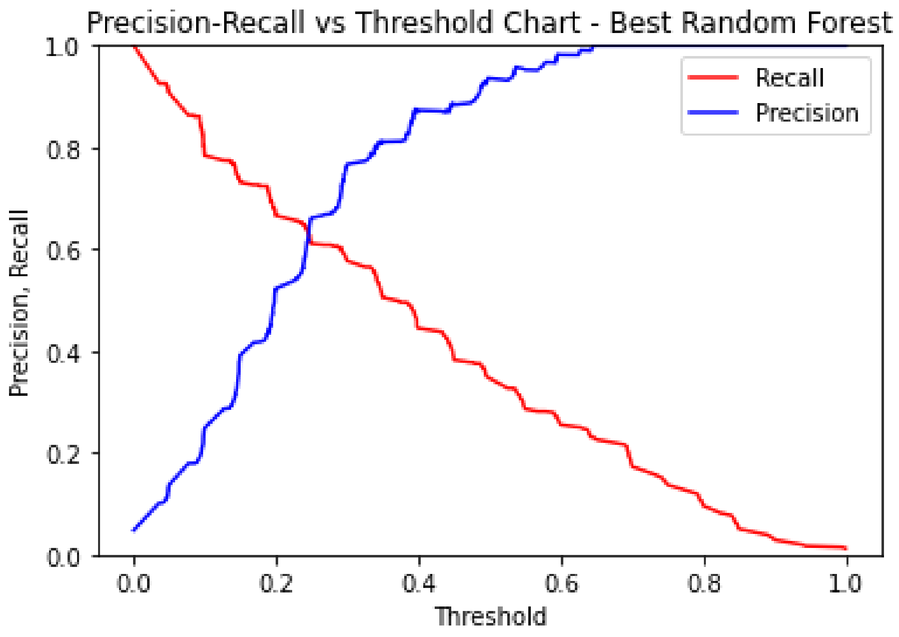


Figure 2. Peak recall is observed to be nearer 0.0 than 0.2. A trade-off between precision and recall is observed at around upper threshold 0.2.

Classification Report – Random Forest					
Threshold	*Precision*	*Recall*	*Specificity*	*G-mean*	*AUC*
0.10	**0.248**	**0.834**	**0.873**	**0.854**	**0.854**
0.15	0.368	0.714	0.939	0.819	0.826
0.05	0.133	0.894	0.707	0.795	0.800
0.11	0.273	0.788	0.895	0.400	0.842
0.09	0.177	0.868	0.797	0.832	0.833

Table 2. The accuracy metrics of Random Forest model adjusted for threshold and favoring recall. In bold is the most optimal threshold.

Confusion Matrix – Random Forest

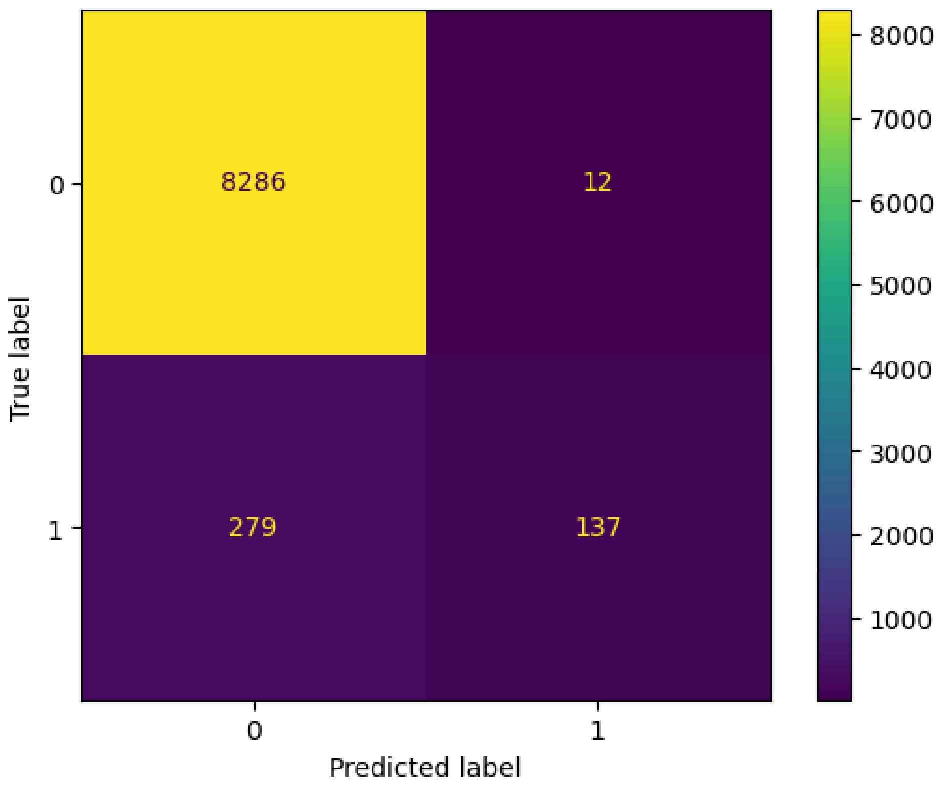


Figure 3. Confusion Matrix of Random Forest adjusted with the most optimal threshold of 0.1.

4. Extreme Gradient Boosting

XGBoost has its own cross-validation function via its library in Python called `xgb.cv()`. This function requires a data matrix for training its cross-validated models, where the 80% training data is used.

4.1. Hyperparameters (Manually)

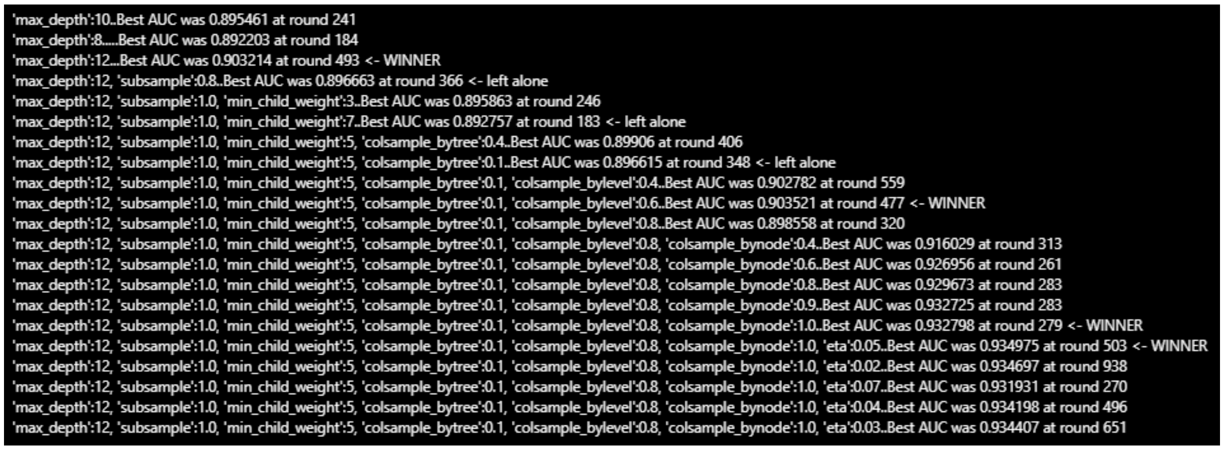
The user-defined cross-validation strategy is the same one used for Random Forest for sake of simplicity (stratified 10 folds repeated 3 times). Cross-validation and fitting the data are both done within the function. The list below are the initial input parameters used in `xgb.cv()`, which ran for 100,000 rounds and was also evaluated with AUC. One important parameter for the algorithm is called early stopping rounds, which is set to 50. The way it works is if early stopping rounds is N, the algorithm will halt before reaching the end of the large number of rounds mentioned earlier, if it has gone N rounds without an improvement in AUC. In short, the purpose of this large number of rounds is not for hypertuning, but rather for the team to observe the one round that the test AUC produced by the algorithm started to decrease, however long it may take to behave so.

Initial Hyperparameters:

- 'eta':0.1
- 'max_depth':10
- 'subsample':1.0
- 'min_child_weight':5
- colsample_bytree':0.2
- colsample_bylevel':0.2
- colsample_bynode':0.2
- objective':'binary:logistic'
- 'eval_metric':'auc'
- ' scale_pos_weight ':20

XGBoost can be trained with many other parameters, but the ones chosen above are due to recommendations from the head of the department. Having said that the values in the parameter setting above is initially set, the test AUC of the one round that caused the algorithm to halt is recorded, and the algorithm is run again once a sequential parameter is changed. Those sequential parameters are the following: `max_depth` (chosen first because it is generally invariant to other parameters), `subsample`, `min_child_weight`, `colsample_bytree`, `colsample_bylevel`, `colsample_bynode` and lastly `eta`. These were changed with either increments or decrements of 2 or 0.2, depending on if the parameter is an integer or a float. Smaller increments or decrements are introduced once there is a gap of knowledge between the changes. The direction of the manually changed values, either increase or decrease, is continued as long as the produced metrics keep improving, otherwise the direction will be switched. In short, each parameter is tested with both an increase and decrease before keeping or changing direction and finally moving on to the next parameter. A more visual sample can be seen in Figure 4, and the following Table 3 showcases the best parameter settings that were picked manually.

Visual Sample of Manual Cross-Validation



```
'max_depth':10..Best AUC was 0.895461 at round 241
'max_depth':8.....Best AUC was 0.892203 at round 184
'max_depth':12...Best AUC was 0.903214 at round 493 <- WINNER
'max_depth':12, 'subsample':0.8..Best AUC was 0.896663 at round 366 <- left alone
'max_depth':12, 'subsample':1.0, 'min_child_weight':3..Best AUC was 0.895863 at round 246
'max_depth':12, 'subsample':1.0, 'min_child_weight':7..Best AUC was 0.892757 at round 183 <- left alone
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.4..Best AUC was 0.89906 at round 406
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.1..Best AUC was 0.896615 at round 348 <- left alone
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.1, 'colsample_bylevel':0.4..Best AUC was 0.902782 at round 559
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.1, 'colsample_bylevel':0.6..Best AUC was 0.903521 at round 477 <- WINNER
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.1, 'colsample_bylevel':0.8..Best AUC was 0.898558 at round 320
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.1, 'colsample_bylevel':0.8, 'colsample_bynode':0.4..Best AUC was 0.916029 at round 313
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.1, 'colsample_bylevel':0.8, 'colsample_bynode':0.6..Best AUC was 0.926956 at round 261
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.1, 'colsample_bylevel':0.8, 'colsample_bynode':0.8..Best AUC was 0.929673 at round 283
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.1, 'colsample_bylevel':0.8, 'colsample_bynode':0.9..Best AUC was 0.932725 at round 283
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.1, 'colsample_bylevel':0.8, 'colsample_bynode':1.0..Best AUC was 0.932798 at round 279 <- WINNER
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.1, 'colsample_bylevel':0.8, 'colsample_bynode':1.0, 'eta':0.05..Best AUC was 0.934975 at round 503 <- WINNER
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.1, 'colsample_bylevel':0.8, 'colsample_bynode':1.0, 'eta':0.02..Best AUC was 0.934697 at round 938
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.1, 'colsample_bylevel':0.8, 'colsample_bynode':1.0, 'eta':0.07..Best AUC was 0.931931 at round 270
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.1, 'colsample_bylevel':0.8, 'colsample_bynode':1.0, 'eta':0.04..Best AUC was 0.934198 at round 496
'max_depth':12, 'subsample':1.0, 'min_child_weight':5, 'colsample_bytree':0.1, 'colsample_bylevel':0.8, 'colsample_bynode':1.0, 'eta':0.03..Best AUC was 0.934407 at round 651
```

Figure 4. Visual representation of the manual cross-validation method used in this project.

RandomizedSearchCV Results

Hyperparameters	*Control*
max_depth	12
subsample	1.0
min_child_weight	5
colsample_bytree	0.1
colsample_bylevel	0.8
colsample_bynode	1.0
eta	0.05

Table 3. The resulting hyperparameters for XGBoost manual cross-validation that followed the preprocessing methods of this project.

4.2. Finetuning

The best performing XGBoost model was also adjusted by threshold the same method used for the best Random Forest method. The sequential finetuning process for XGBoost can be seen in Table 4.

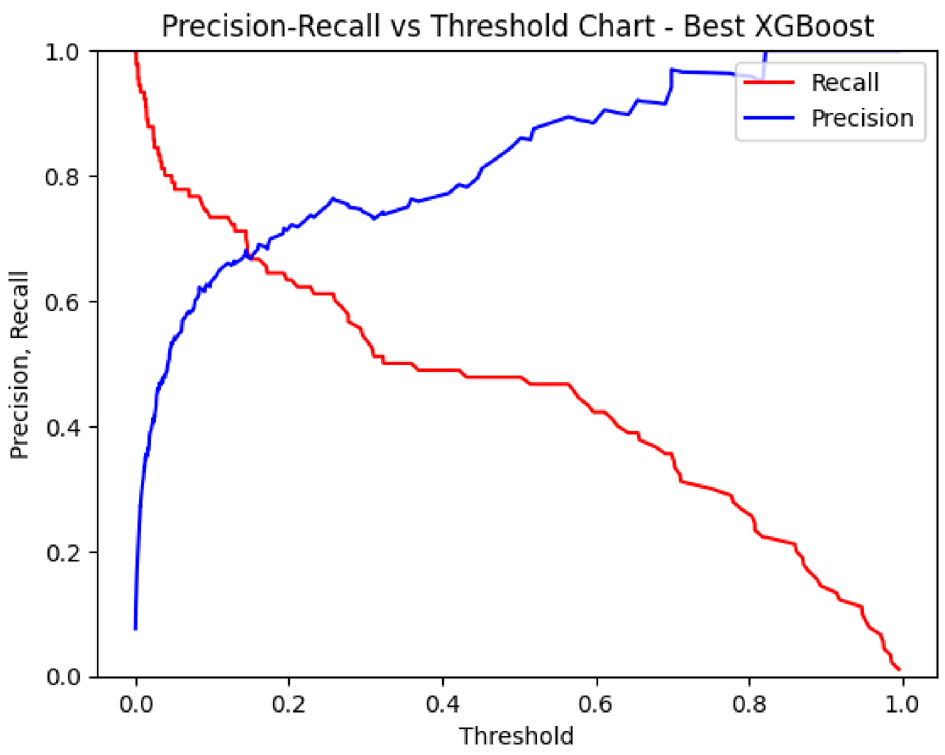


Figure 5. Peak recall is observed to be nearer 0.0 than 0.2. A trade-off between precision and recall is observed at around lower threshold 0.2.

Classification Report – XGBoost					
Threshold	*Precision*	*Recall*	*Specificity*	*G-mean*	*AUC*
0.10	0.635	0.733	0.965	0.841	0.849
0.15	0.667	0.667	0.973	0.805	0.820
0.05	**0.538**	**0.789**	**0.944**	**0.863**	**0.867**
0.04	0.486	0.800	0.830	0.863	0.865
0.06	0.556	0.778	0.949	0.859	0.863

Table 4. The accuracy metrics of XGBoost model adjusted for threshold and favoring recall. In bold is the most optimal threshold.

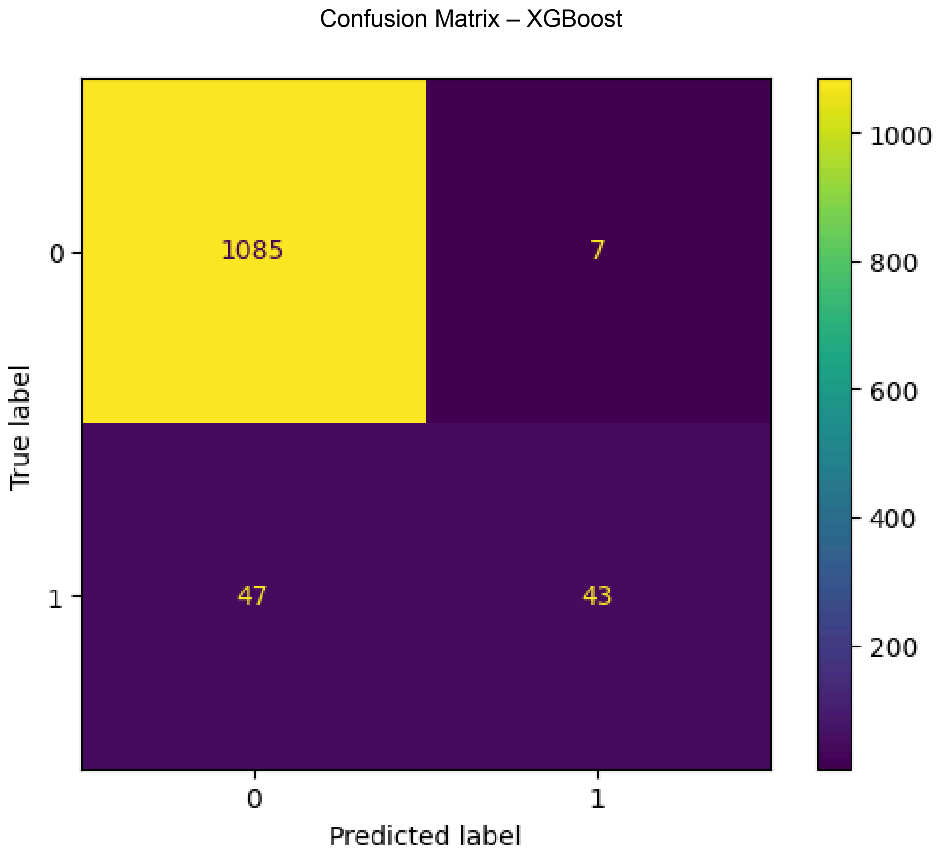


Figure 6. Confusion Matrix of XGBoost adjusted with the most optimal threshold of 0.05.

5. Results

Model					
	1st Year	2nd Year	3rd Year	4th Year	5th Year

	MN	STD	MN	STD	MN	STD	MN	STD	MN	STD
RF	0.798	0.109	0.743	0.091	0.788	0.065	0.765	0.069	0.883	0.060
XGB	0.859	0.094	0.821	0.067	0.859	0.065	0.835	0.067	0.925	0.033

Table 5. This table represents the mean (MN) and standard deviation (STD) of AUC from 10 stratified cross-validation folds repeated 3 times for each of the 5 years.

5.1. Accuracy Metrics

We are favoring the negative target, meaning that we want to correctly predict all non-bankruptcy cases. Therefore, recall is the most important accuracy metric as opposed to precision; it is okay to falsely assign a normal company with bankruptcy where further analysis can be done on such company for correction, however it is unacceptable to assign a failing company with non-bankruptcy, because said company is at high risk and will (by compromising for more false positives, we can minimize false negatives).

Inversely, in a previous precision-focused project, we favored the positive target by attempting to correctly predict all spam emails, since any time-sensitive emails or the like sent to the spam folder cannot be easily identified again.

Specificity is the 2nd most important because we also want to correctly predict all the non-bankruptcy cases as well. And to compare our competing thresholds more accurately, the G-mean, geometric mean of recall and specificity, and AUC, the tradeoff between true positives and false positives, are also calculated and considered.

6. Conclusion

Our study has concluded that both Machine Learning techniques, Random Forest (RF) and Extreme Gradient Boosting (XGBoost) produce very accurate results when predicting whether a company will go bankrupt. We utilized weights to account for the imbalance between classes within the provided dataset. A cross-validation strategy consisting of stratified 10 folds repeated 3 times is used to calculate mean and standard deviation AUC; the Random Forest model produced 0.846 and 0.034 respectively on the out-of-sample 20% test set, and the XGBoost model produced 0.925 and 0.033 respectively on the same test set.

7. References

1. Zięba, M., Tomczak, S. K., & Tomczak, J. M. (2016). Ensemble boosted trees with synthetic features generation in application to bankruptcy prediction. Expert Systems with Applications, 58, 93–101. https://doi.org/10.1016/j.eswa.2016.04.001

8. Appendix

```
import pandas as pd

import numpy as np
```

```
from scipy.io import arff

from sklearn.model_selection import train_test_split

from sklearn.impute import SimpleImputer

from sklearn.preprocessing import RobustScaler

from sklearn.ensemble import RandomForestClassifier

import xgboost as xgb

from xgboost import XGBClassifier

from sklearn.model_selection import cross_val_score, RepeatedStratifiedKFold

from sklearn.metrics import confusion_matrix




def do_my_study():



    files = [

        '1year.arff',

        '2year.arff',

        '3year.arff',

        '4year.arff',

        '5year.arff'

    ]



    df = import_data(files)



    impute_strategy = "median"
    X_train, X_test, y_train, y_test = preproc(df, impute_strategy)


    rf_model = do_RF(X_train, y_train)
    threshold = .1
    display_class_report(threshold, rf_model, X_test, y_test)

    xgb_model = do_XGB(X_train, y_train)
    threshold = .05
    display_class_report(threshold, xgb_model, X_test, y_test)



def import_data(files):


    df = pd.DataFrame(arff.loadarff(files[0])[0])


    for f in files[1:]:
        data_temp = arff.loadarff(f)
        df_temp = pd.DataFrame(data_temp[0])
        df = df.merge(df_temp, how='outer')


    return df
```

```python
def preproc(df, impute_strategy):
    # Fixing response variable
    df['class'] = df['class'].replace([b'0', b'1'], [0, 1])

    # Splitting data
    X = df.loc[:, df.columns != 'class'].values
    y = df['class'].values
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=42)

    # Impute
    X_train = SimpleImputer(strategy=impute_strategy).fit_transform(X_train)
    X_test = SimpleImputer(strategy=impute_strategy).fit_transform(X_test)

    # Normalize the data
    X_train = RobustScaler().fit_transform(X_train)
    X_test = RobustScaler().fit_transform(X_test)

    return X_train, X_test, y_train, y_test


def resample(method, X_train, y_train):
    method_holder = method
    X_train, y_train = method_holder.fit_resample(X_train, y_train)


    return X_train, y_train



def do_RF(X_train, y_train):
    rf_model = RandomForestClassifier(
        n_estimators=20,
        bootstrap=True,
        criterion="gini",
        max_depth=None,
        max_features=10,
        min_samples_leaf=2,
        min_samples_split=5,
        class_weight="balanced")
    rf_model.fit(X_train, y_train)


    return rf_model


def do_XGB(X_train, y_train):
    xgb_model = XGBClassifier(
        subsample=1.0,
        max_depth=12,
        min_child_weight=5,
        colsample_bytree=0.1,
        colsample_bylevel=0.8,
        colsample_bynode=1.0,
        eta=0.05,
        eval_metric='auc',
        objective='binary:logistic',
        n_estimators=1000,
        verbosity=1)
    xgb_model.fit(X_train, y_train)


    return xgb_model



def display_AUC(model, X_train, y_train):
    cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=42)


    AUC_holder = cross_val_score(
        model, X_train, y_train, scoring="roc_auc", cv=cv)
    print("\n", model, "\n")
    print("\nMean AUC: {0:.3f} \nStd AUC: {1:.3f}".format(
        AUC_holder.mean(), AUC_holder.std()))



def display_class_report(threshold, model, X_test, y_test):
    predicted_proba = model.predict_proba(X_test)
    y_preds = (predicted_proba[:, 1] >= threshold).astype('int')

    print("------------------------------")
    print("\nClassification Report \nThreshold = ", threshold,
          "\nAlgorithm = ", model, "):"
          )
    tn, fp, fn, tp = confusion_matrix(y_test, y_preds).ravel()
    precision = tp/(tp+fp)
    recall = tp/(tp+fn)
    specificity = tn / (tn+fp)
    G_mean = np.sqrt(recall*specificity)
    AUC = (recall+specificity)/2
    print("Precision: \t", precision.round(3))
    print("Recall: \t", recall.round(3))
    print("Specificity: \t", specificity.round(3))
    print("G-mean: \t", G_mean.round(3))
    print("AUC: \t\t", AUC.round(3))


if __name__ == "__main__":

    do_my_study()
```