# Predicting Company Bankruptcy

DSC 540 Final Project Presentation

**Group 1:** John Castano, Jared Hebda, Vismay Patel, Evelina Ramoskaite

# Table of Contents

**INTRODUCTION** ..... 2
**EXPLORATORY DATA ANALYSIS** ..... 3
Data ..... 3
Attributes ..... 3
Empirical Cumulative Distribution Function plot: ..... 5
Feature Distributions Histogram: ..... 5
Correlation Analysis: ..... 7
Simple kernel density estimation plot: ..... 8
Pair plot: ..... 9
Class Balance ..... 10
Missing Values ..... 10
**MODEL BUILDING** ..... 11
Methods ..... 11
Dimensionality Reduction- PCA ..... 11
Machine Learning Algorithms implemented/performed on bankruptcy data. ..... 12
**Overview** ..... 12
Logistic Regression ..... 13
Naive Bayes ..... 13
Evaluation metrics from these methods: ..... 14
Gaussian Naive Bayes Confusion matrix and classification report: ..... 14
Bernoulli Naive Bayes Confusion Matrix and classification report: ..... 14
Decision Tree: ..... 15
Random Forest ..... 17
**Boosting Algorithms** ..... 17
XGBoost: ..... 18
AdaBoost: ..... 19
CatBoost: ..... 20
Light GBM: ..... 21
**Support Vector Machine** ..... 22
SVM using linear Kernel: ..... 23
SVM using rbf kernel: ..... 23
**Feature Importances** ..... 24
**MODEL MANAGEMENT** ..... 26
**CONCLUSION** ..... 34

# INTRODUCTION

The purpose of this project is to better predict whether a corporation will go bankrupt, given publicly available stock exchange data. We worked with data from the Taiwan Economic Journal for the years 1999 to 2009, which contained bankruptcy records for companies traded on the Taiwan Stock Exchange.

In this report, we aim to discover features or latent factors that may be indicative of a company's propensity to go bankrupt by attempting a variety of classification algorithms and tuning parameters. We first describe the data by looking at the different distributions of each of the variables as well as looking at potential correlations. Then, we utilize synthetic oversampling to account for a heavily imbalanced target class distribution. After processing the data, we perform hyperparameter tuning on different classification algorithms and compare the results. Finally, we discuss the optimal deployment and management of our best model.

# EXPLORATORY DATA ANALYSIS

## Data

The raw dataset consists of 6,819 observations and 95 different numeric attributes. The sample was collected from the Taiwan Economic Journal, which had Taiwan Stock Exchange data on publicly traded firms. Company bankruptcy status was defined based on the business regulations of the exchange. The dataset was already cleaned and had no missing values.

## Attributes

Y - Bankrupt?: Class label
X1 - ROA(C) before interest and depreciation before interest: Return On Total Assets(C)
X2 - ROA(A) before interest and % after tax: Return On Total Assets(A)
X3 - ROA(B) before interest and depreciation after tax: Return On Total Assets(B)
X4 - Operating Gross Margin: Gross Profit/Net Sales
X5 - Realized Sales Gross Margin: Realized Gross Profit/Net Sales
X6 - Operating Profit Rate: Operating Income/Net Sales
X7 - Pre-tax net Interest Rate: Pre-Tax Income/Net Sales
X8 - After-tax net Interest Rate: Net Income/Net Sales
X9 - Non-industry income and expenditure/revenue: Net Non-operating Income Ratio
X10 - Continuous interest rate (after tax): Net Income-Exclude Disposal Gain or Loss/Net Sales
X11 - Operating Expense Rate: Operating Expenses/Net Sales
X12 - Research and development expense rate: (Research and Development Expenses)/Net Sales
X13 - Cash flow rate: Cash Flow from Operating/Current Liabilities
X14 - Interest-bearing debt interest rate: Interest-bearing Debt/Equity
X15 - Tax rate (A): Effective Tax Rate
X16 - Net Value Per Share (B): Book Value Per Share(B)
X17 - Net Value Per Share (A): Book Value Per Share(A)
X18 - Net Value Per Share (C): Book Value Per Share(C)
X19 - Persistent EPS in the Last Four Seasons: EPS-Net Income
X20 - Cash Flow Per Share
X21 - Revenue Per Share (Yuan ¥): Sales Per Share
X22 - Operating Profit Per Share (Yuan ¥): Operating Income Per Share
X23 - Per Share Net profit before tax (Yuan ¥): Pretax Income Per Share
X24 - Realized Sales Gross Profit Growth Rate
X25 - Operating Profit Growth Rate: Operating Income Growth
X26 - After-tax Net Profit Growth Rate: Net Income Growth
X27 - Regular Net Profit Growth Rate: Continuing Operating Income after Tax Growth
X28 - Continuous Net Profit Growth Rate: Net Income-Excluding Disposal Gain or Loss Growth
X29 - Total Asset Growth Rate: Total Asset Growth
X30 - Net Value Growth Rate: Total Equity Growth
X31 - Total Asset Return Growth Rate Ratio: Return on Total Asset Growth
X32 - Cash Reinvestment %: Cash Reinvestment Ratio
X33 - Current Ratio
X34 - Quick Ratio: Acid Test
X35 - Interest Expense Ratio: Interest Expenses/Total Revenue
X36 - Total debt/Total net worth: Total Liability/Equity Ratio
X37 - Debt ratio %: Liability/Total Assets
X38 - Net worth/Assets: Equity/Total Assets

X39 - Long-term fund suitability ratio (A): (Long-term Liability+Equity)/Fixed Assets
X40 - Borrowing dependency: Cost of Interest-bearing Debt
X41 - Contingent liabilities/Net worth: Contingent Liability/Equity
X42 - Operating profit/Paid-in capital: Operating Income/Capital
X43 - Net profit before tax/Paid-in capital: Pretax Income/Capital
X44 - Inventory and accounts receivable/Net value: (Inventory+Accounts Receivables)/Equity
X45 - Total Asset Turnover
X46 - Accounts Receivable Turnover
X47 - Average Collection Days: Days Receivable Outstanding
X48 - Inventory Turnover Rate (times)
X49 - Fixed Assets Turnover Frequency
X50 - Net Worth Turnover Rate (times): Equity Turnover
X51 - Revenue per person: Sales Per Employee
X52 - Operating profit per person: Operation Income Per Employee
X53 - Allocation rate per person: Fixed Assets Per Employee
X54 - Working Capital to Total Assets
X55 - Quick Assets/Total Assets
X56 - Current Assets/Total Assets
X57 - Cash/Total Assets
X58 - Quick Assets/Current Liability
X59 - Cash/Current Liability
X60 - Current Liability to Assets
X61 - Operating Funds to Liability
X62 - Inventory/Working Capital
X63 - Inventory/Current Liability
X64 - Current Liabilities/Liability
X65 - Working Capital/Equity
X66 - Current Liabilities/Equity
X67 - Long-term Liability to Current Assets
X68 - Retained Earnings to Total Assets
X69 - Total income/Total expense
X70 - Total expense/Assets
X71 - Current Asset Turnover Rate: Current Assets to Sales
X72 - Quick Asset Turnover Rate: Quick Assets to Sales
X73 - Working capitcal Turnover Rate: Working Capital to Sales
X74 - Cash Turnover Rate: Cash to Sales
X75 - Cash Flow to Sales
X76 - Fixed Assets to Assets
X77 - Current Liability to Liability
X78 - Current Liability to Equity
X79 - Equity to Long-term Liability
X80 - Cash Flow to Total Assets
X81 - Cash Flow to Liability
X82 - CFO to Assets
X83 - Cash Flow to Equity
X84 - Current Liability to Current Assets
X85 - Liability-Assets Flag: 1 if Total Liability exceeds Total Assets, 0 otherwise
X86 - Net Income to Total Assets
X87 - Total assets to GNP price
X88 - No-credit Interval
X89 - Gross Profit to Sales
X90 - Net Income to Stockholder's Equity
X91 - Liability to Equity

X92 - Degree of Financial Leverage (DFL)
X93 - Interest Coverage Ratio (Interest expense to EBIT)
X94 - Net Income Flag: 1 if Net Income is Negative for the last two years, 0 otherwise
X95 - Equity to Liability

## Empirical Cumulative Distribution Function plot:

The empirical cumulative distribution function is closely related to cumulative frequency, the ecdf creates an estimate of the cumulative distribution for a data point. The estimate is based on the sample data. An estimate is created by assigning probability to each point and then adding the probabilities, point by point, from minimum to maximum value. This produces the cumulative probability for each point, the probability assigned to a sample value is the estimate of the proportion of times the value occurs in the data. Here empirical stands for observations.
Let us explore a few graphs generated for Bankruptcy data.

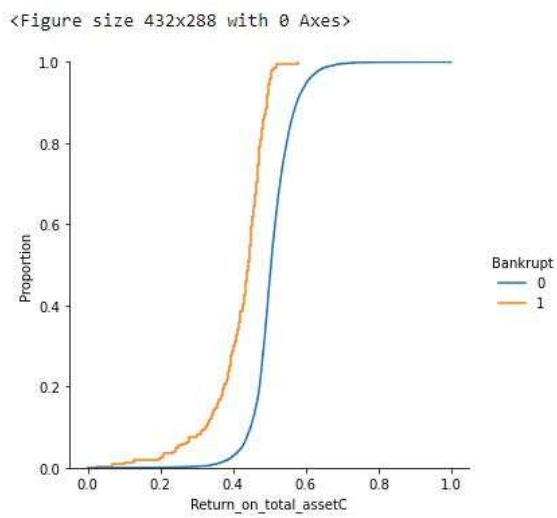



**Figure 0.1**

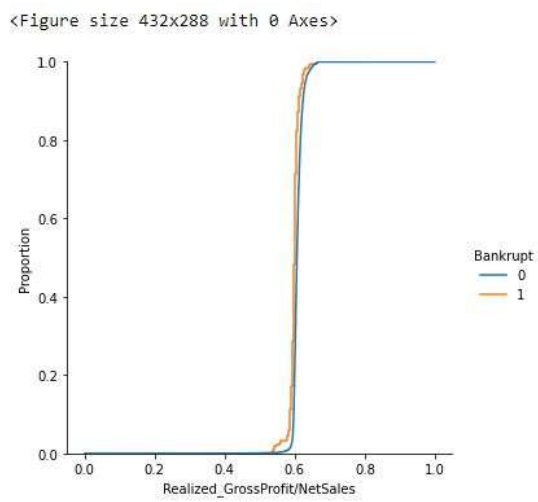



**Figure 0.2**

The above ecdf plots are generated from seaborn library, using a step graph pattern, the first graph is for Return_on_total_assetC and the second graph is on Realized_GrossProfitNet Sales. The blue line in the graph represents non-bankrupt companies and the yellow line represents bankrupt companies. The lines in the graphs show the imbalance nature of data points.

## Feature Distributions Histogram:

We first explored the distributions of each attribute in the dataset.

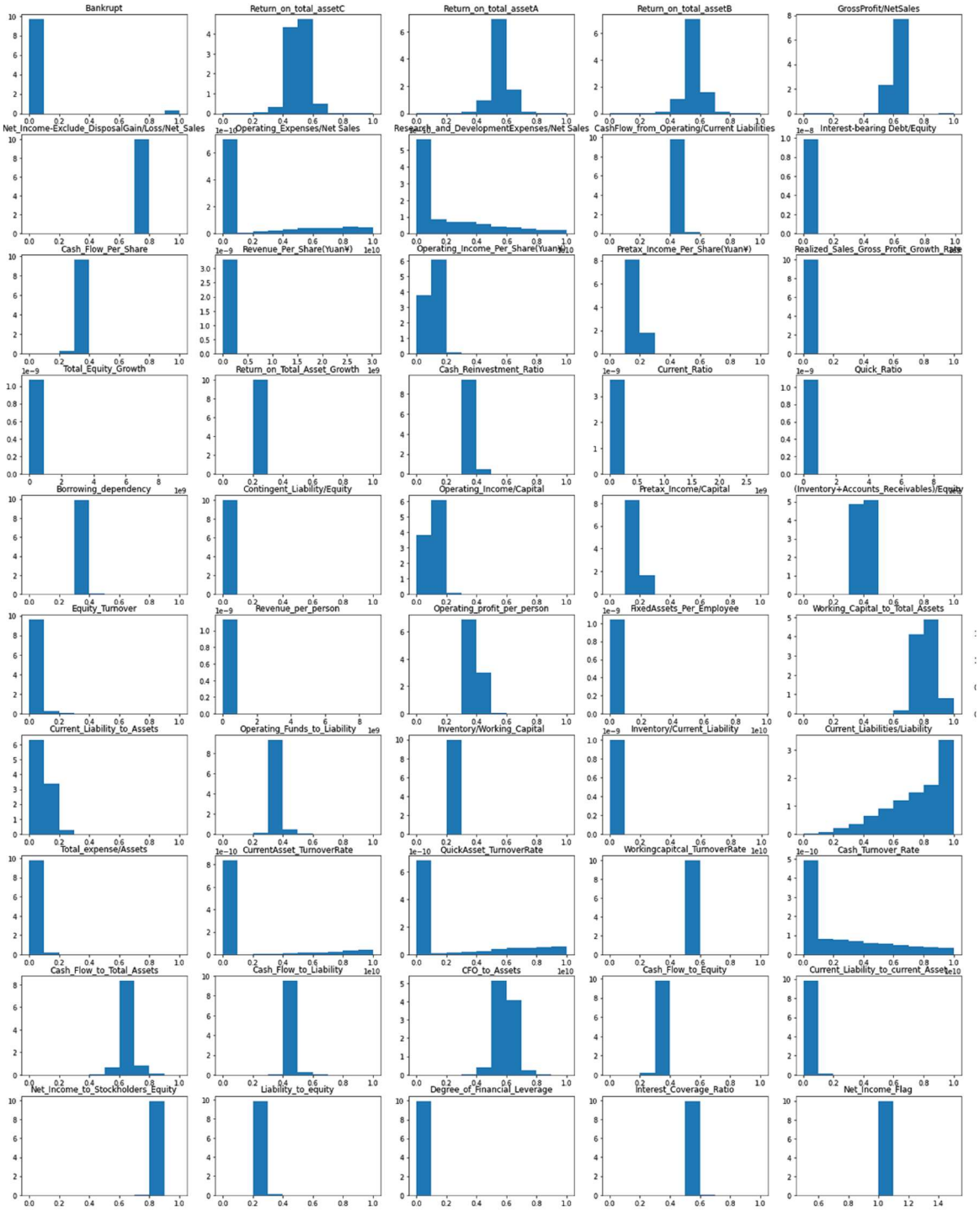



**Figure 0.3**

From the histograms above we can observe that the data points are either extremely skewed or biased towards a specific data point, resulting in imbalance plots/graphs.

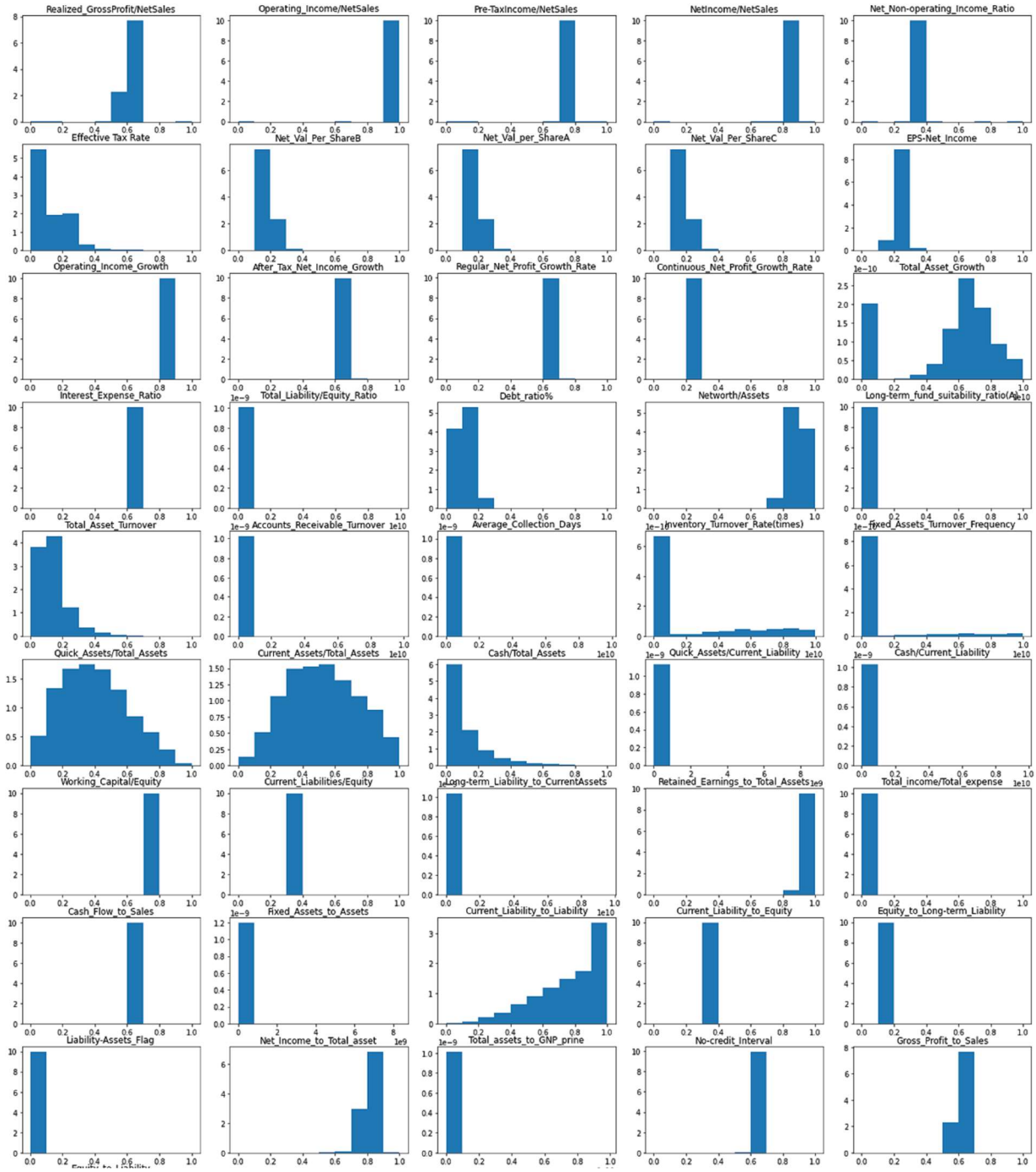



Figure 0.4

Similar trend can be followed in this portion of histograms. We are visualizing all the 95 variables here.

# Correlation Analysis:

We explored the correlations in the dataset next by creating a correlation plot filtered for the most relevant values, with a threshold of 0.3. Attributes with weaker correlations were excluded to create a more readable heatmap with 48 of the 95 features. Bankruptcy was the most heavily correlated with a low ROA, lower operating gross margins, lower net value per share, lower operating profits,lower retained earnings, and a higher expense/asset ratio.

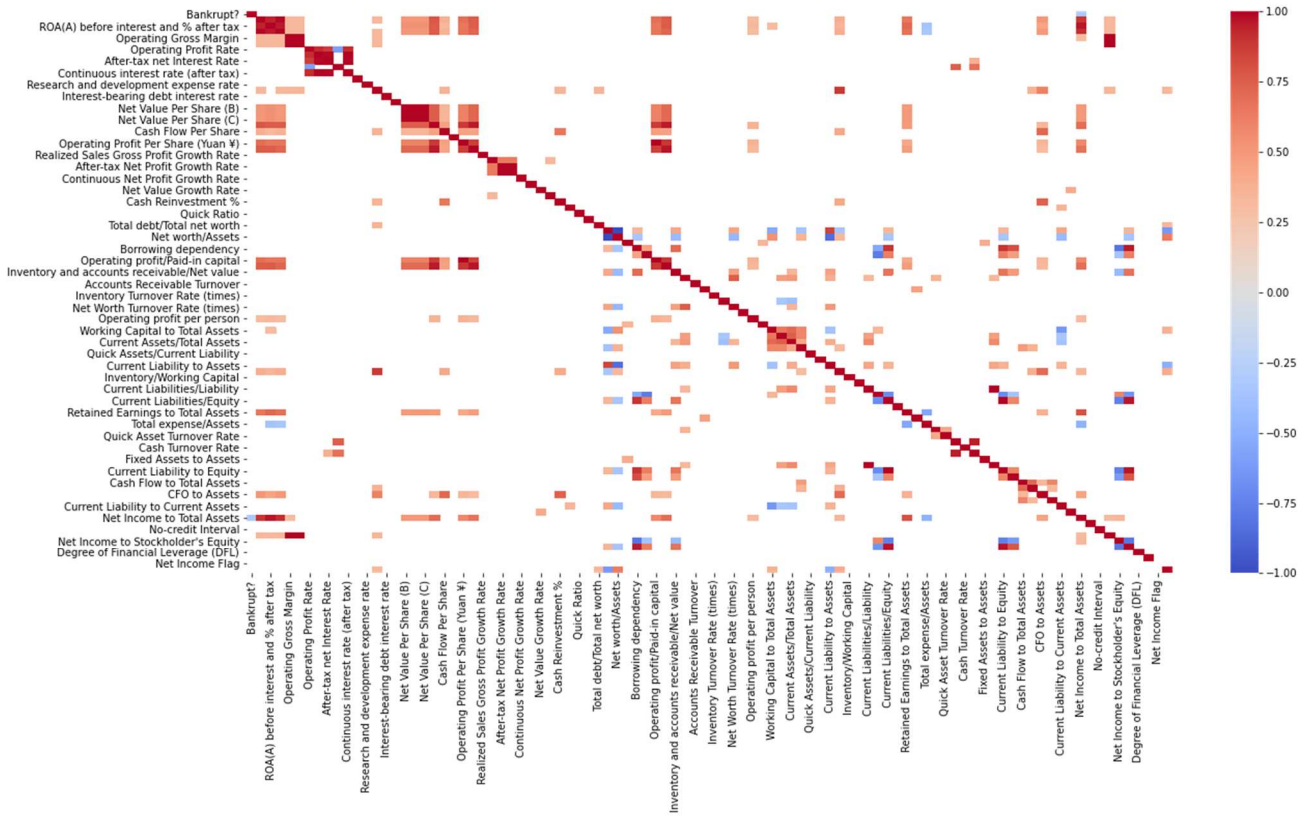



**Figure 0.5**

# Simple kernel density estimation plot:

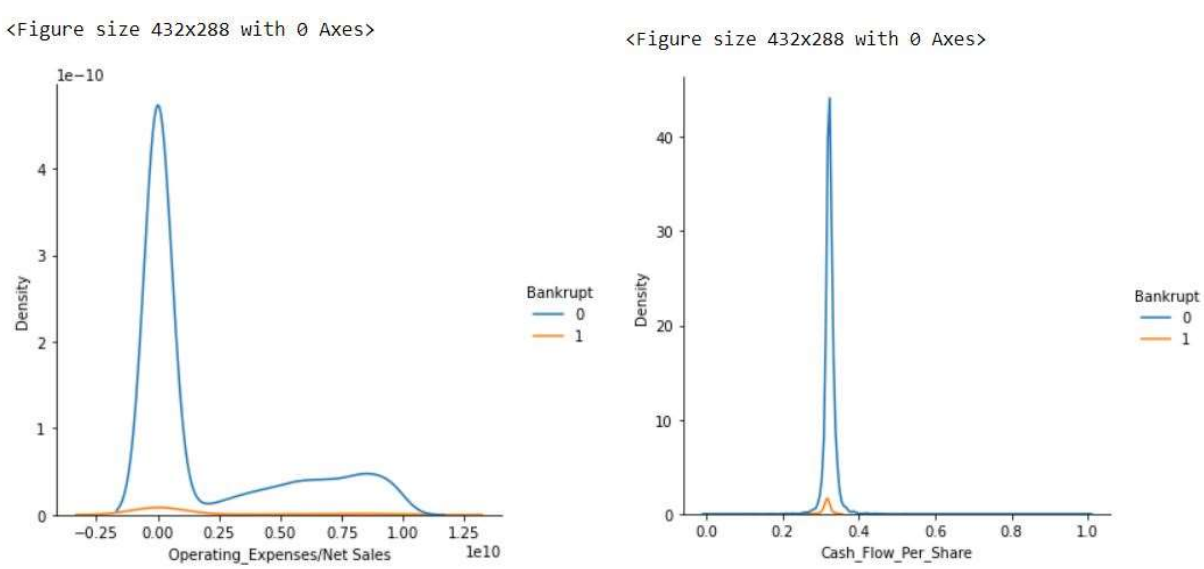



**Figure 0.6**

Kernel density plots are used to observe the distribution of the dataset,  KDE represents the data using continuous probability curves in one or more dimensions.

## Pair plot:

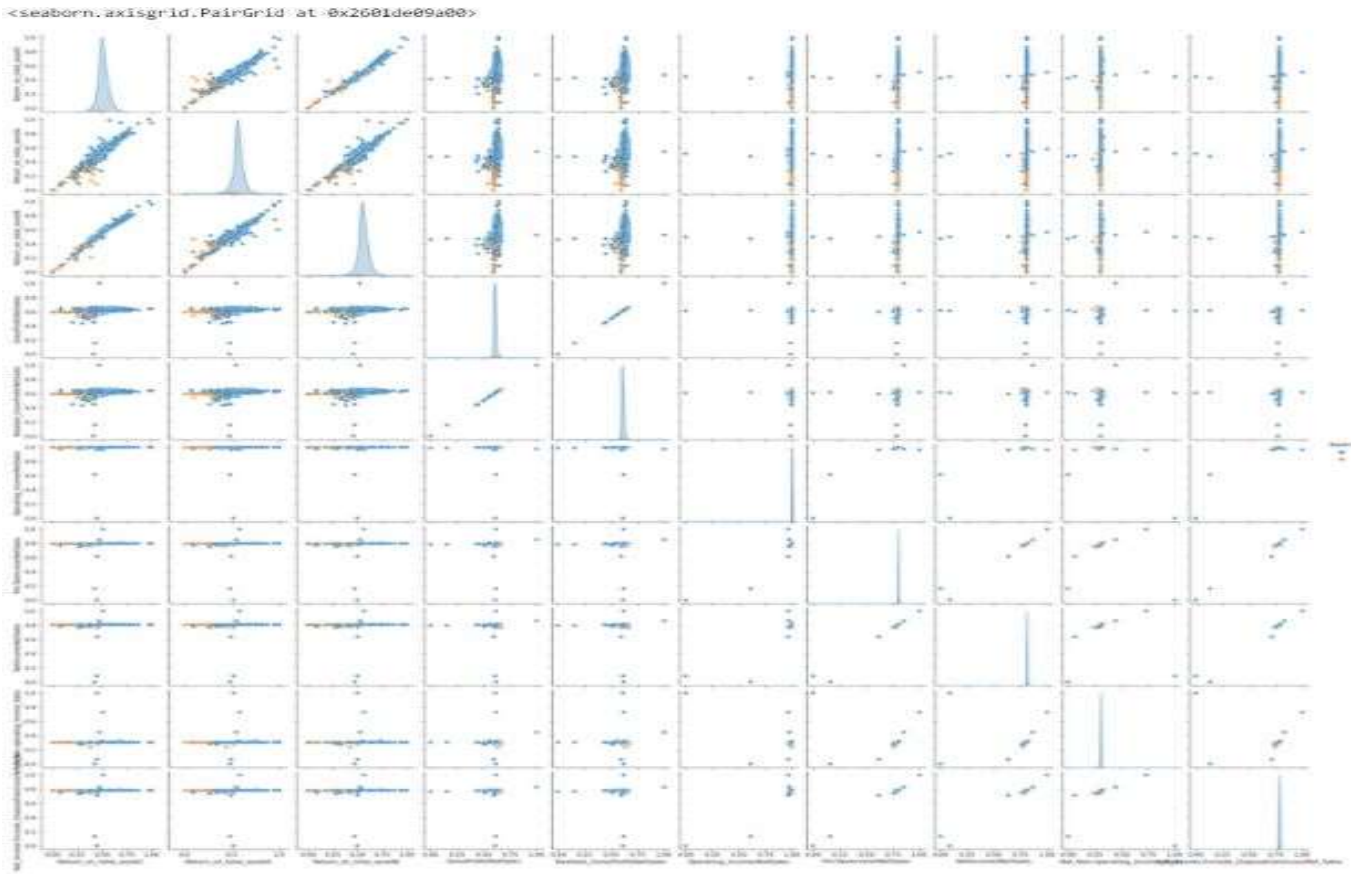
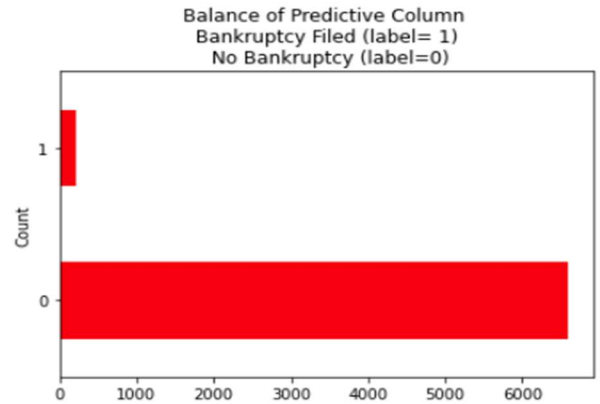



**Figure 0.7**

Pair plot describes the relationship between  variables, and in over case there are only a few variables which are correlated with each other, and this helps to analyse the nature of the data that we have, and perform the results accordingly.
Since the data is unbalanced we will perform SMOTE analysis below to balance the data.

# Class Balance




```
Bankrupt?
0    6599
1     220
dtype: int64
Companies that have not filed for Bankruptcy: 6599
Companies that filed for Bankruptcy): 220
```

**Figure 0.6**

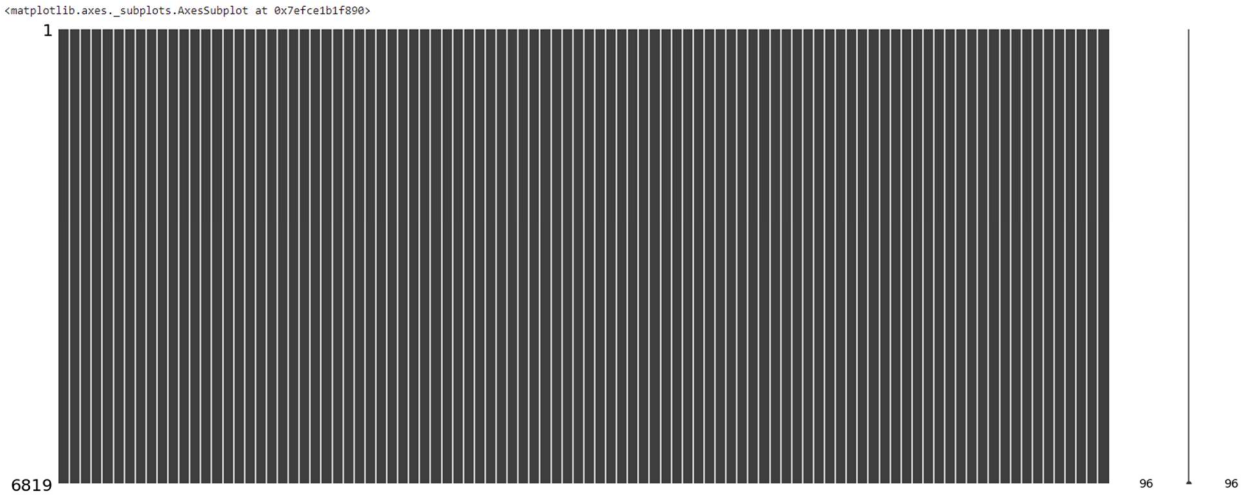
The class distribution of the data set is imbalanced. The Bankrupt class is heavily underrepresented in the dataset. Only 220, or 3.3% of the companies in the exchange were bankrupt. This presented a unique challenge, as there were relatively few minority class samples to train our models with. To counteract the imbalance, we applied a synthetic minority oversampling technique (SMOTE) to the training set.

# Missing Values

Our dataset was very complete and was not missing any data.  Therefore, we were fortunate we did not have to do any imputation on the dataset in order to fill out any missing values.

```
#!pip install missingno
import missingno as msno
msno.matrix(dataset)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7efce1b1f890>
```



# MODEL BUILDING

## Methods

Since our dataset was heavily imbalanced, we utilized Synthetic Minority Oversampling Technique (SMOTE) to better represent the minority class (bankruptcies). Using SMOTE, we were able to train on data that was perfectly balanced. We also applied scaling to our data for normalization and used a grid search to find optimal hyper-parameters. Each grid search utilized 5-fold cross validation to evaluate each combination of parameters. After finding the best parameters, we tested this model on the hold-out set for model evaluation. We utilized logistic regression as a benchmark for the performance of our other models.

## Dimensionality Reduction- PCA

Since our dataset contained 95 features, we experimented with Principal Component Analysis to see if we could reduce dimensionality and improve our classification of bankruptcies. After running PCA, we settled on 17 components, which explain 96.2% of the variance in the original data. Looking at the scree-plot below,there are marginal increases of variance explained after component 17, showing this is a good number of components to select.

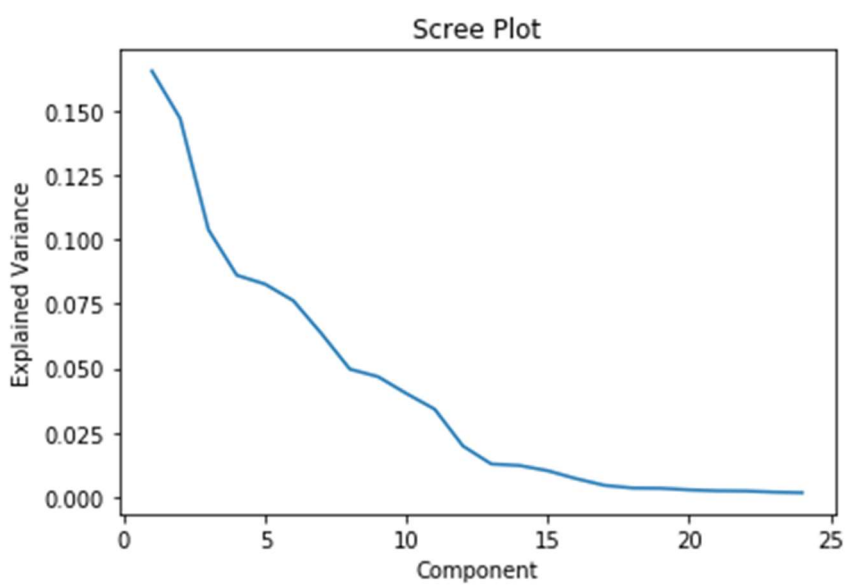



**Figure 0.8**

Below you can see the composition of the seventeen principal components. Overall, the distribution of the importance is scattered across many attributes and components.

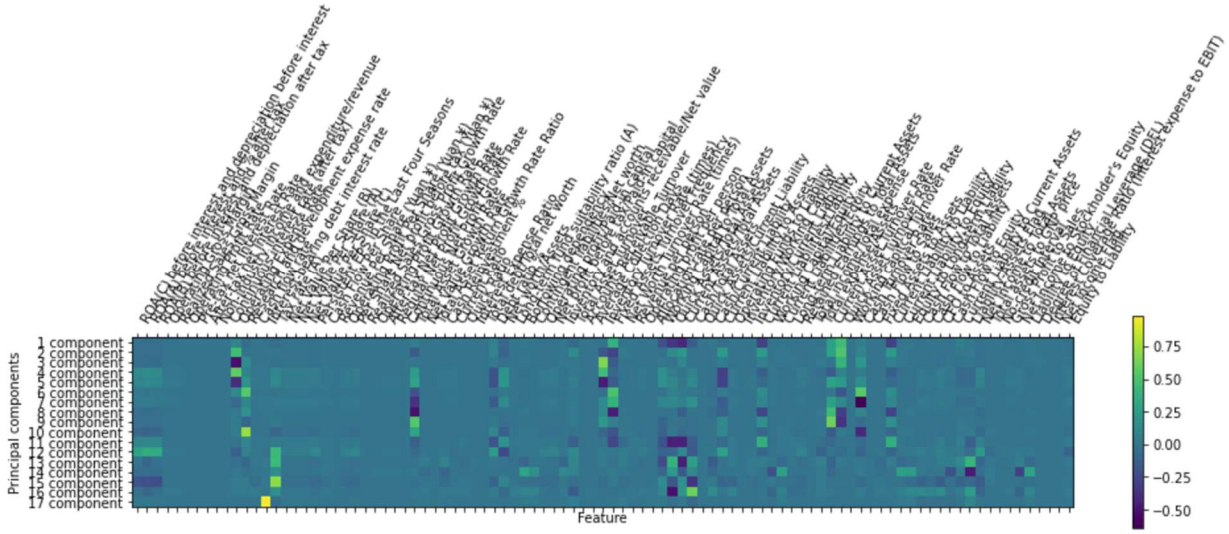



**Figure 0.9**

In most cases, PCA did not improve the classification when compared to using the non transformed dataset. From the feature importance analysis done below, we see that there are a few features that are heavily important. Since a subset of the 95 features are good predictors of bankruptcy, this may explain why PCA does not improve our classification.

## Machine Learning Algorithms implemented/performed on bankruptcy data.

### Overview

| Model | Accuracy Score | Precision | Recall |
|---|---|---|---|
| Logistic Regression | 88.7% | 96% | 88% |
| Naive Bayes | 77.0% | 96% | 77% |
| Decision Tree | 92.4% | 96% | 92% |
| Random Forest | 95.0% | 96% | 95% |
| XGBoost | 96.6% | 96% | 97% |
| AdaBoost | 95.2% | 96% | 95% |
| CatBoost | 96.8% | 97% | 97% |
| LightGBM | 97.1% | 97% | 97% |
| Support Vector Classifier | 93% | 96% | 93% |

**Table 1.0**

## Logistic Regression

As a benchmark, we trained a logistic regression model to compare other more complex models to. After training and tuning with a grid-search, this model had an accuracy of 96.11% and a recall of 18.75% on the hold-out set. This logistic regression model performed very poorly on predicting the minority class (bankruptcies), even though it had a high accuracy. Next, we balanced the dataset using SMOTE, which improved our recall on logistic regression to 81.25% but decreased our accuracy to 88.78%. Moving forward, we tried to run more complex models to see if we could improve upon these scores and better predict bankruptcies. It was also clear that SMOTE was allowing us to train better by balancing the data, so we continued using this method to make an even split between the training classes.

Logistic Regression Classification Report using SMOTE:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.9924 | 0.8906 | 0.9387 | 1316 |
| 1 | 0.2131 | 0.8125 | 0.3377 | 48 |
| accuracy | | | 0.8878 | 1364 |
| macro avg | 0.6027 | 0.8515 | 0.6382 | 1364 |
| weighted avg | 0.9650 | 0.8878 | 0.9176 | 1364 |

Confusion Matrix:

```
[[1172  144]
 [   9   39]]
```

## Naive Bayes

Naive Bayes method is a supervised learning algorithm based on applying Bayes' theorem with "naive" assumption of conditional independence between every pair of features given the value of the class variable.One of the differences of naive Bayes classifier is it makes assumptions regarding the distribution of independent and dependent variables.

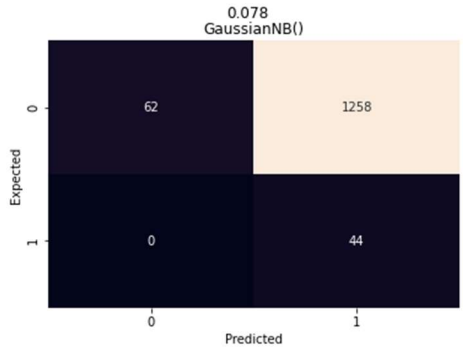
**Perks** of using this method is, the naive bayes learners and classifiers can be extremely fast compared to other sophisticated methods. This is because, each distribution can be independently estimated as one dimensional distribution, which helps to alleviate problems from the curse of dimensionality.

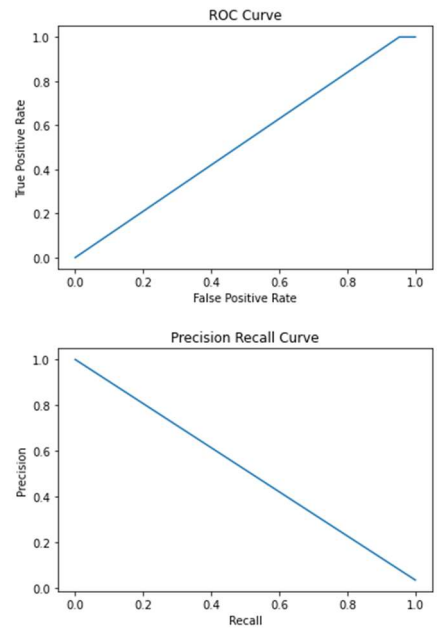
**Cons** of naive Bayes classifier is, it is generally an inferior estimator. It also assumes that all features are independent, which is not always the case.

In this project we have applied Naive Bayes using two classification algorithms, i.e., Gaussian Naive Bayes algorithm for classification and Bernoulli Naive Bayes classification algorithm. The Gaussian classifier estimates the mean and standard deviation making it easier to work with, and Bernoulli classifier is designed for boolean/binary data. This makes both the models suitable for over dataset.

# Evaluation metrics from these methods:

## Gaussian Naive Bayes Confusion matrix and classification report:



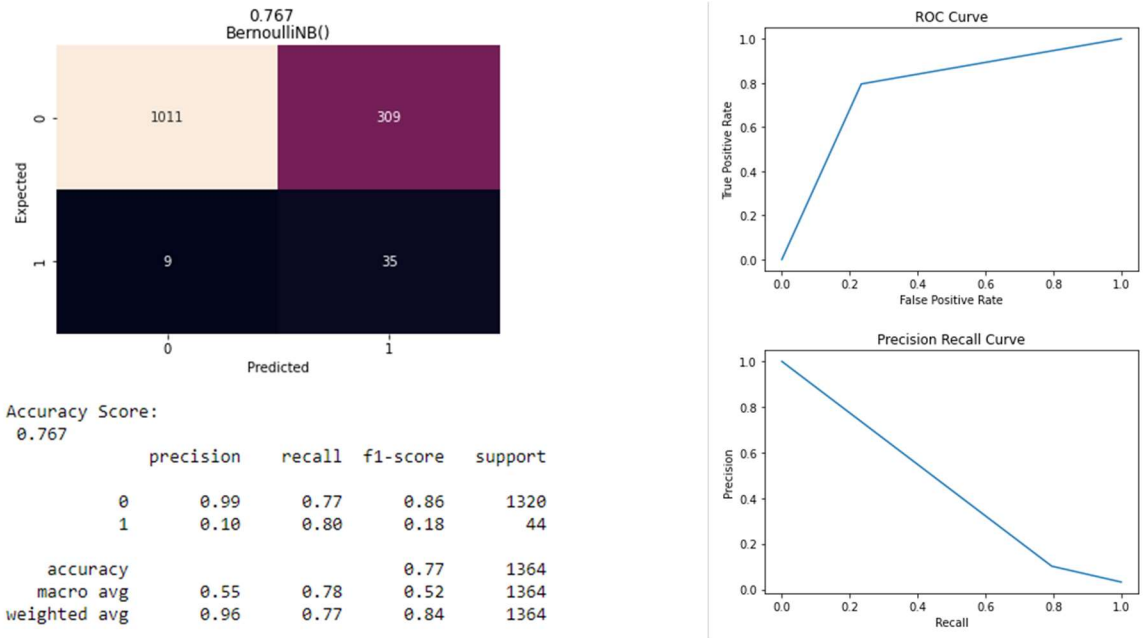




```
Accuracy Score:
 0.078
              precision    recall  f1-score   support

           0       1.00      0.05      0.09      1320
           1       0.03      1.00      0.07        44

    accuracy                           0.08      1364
   macro avg       0.52      0.52      0.08      1364
weighted avg       0.97      0.08      0.09      1364
```

**Figure 1.0**

Based on the evaluation matrix and roc curves we can observe that the model performs with a mere 8% accuracy, and the recall values / graph show less accuracy in the test. The Gaussian model correctly identified all of the minority class, but its utility was minimal because almost all of the majority class was mislabeled as bankrupt.

## Bernoulli Naive Bayes Confusion Matrix and classification report:




```
Accuracy Score:
 0.767
              precision    recall  f1-score   support

           0       0.99      0.77      0.86      1320
           1       0.10      0.80      0.18        44

    accuracy                           0.77      1364
   macro avg       0.55      0.78      0.52      1364
weighted avg       0.96      0.77      0.84      1364
```

**Figure 1.1**

Since Bernoulli Naye Bayes classifier is for binary / boolean data, it shows better performance on the bankruptcy prediction than the GaussianNB model. The accuracy scores and recall / precision values are better than that of the GaussianNB model with 77% accuracy and 77% recall and 96% precision.

## Decision Tree:

We also used a decision tree classifier, to see how the data performs with decision trees. A decision tree is a flow chart like tree structure where an internal node represents features (or attributes), the branch represents a decision rule, and each leaf node represents the outcome. The top most node is called the root node. The attribute selection process takes place by measuring Information gain, gain ratio, gini index.

The feature selection method done for the decision tree was different, we used chi-square for feature selection in order to see if there is a difference we observe in the results. so , we used feature_selection method from scikit-learn to select top 10% attributes from 95 attributs. The first model (**Model 1**) was generated from the top 10% attributes selected by the method. The number of attributes selected were 10.
Classification report with top 10% features is shown below:

```
Accuracy:0.901

Classification report
              precision    recall  f1-score   support

           0       0.98      0.92      0.95      1992
           1       0.10      0.35      0.16        54

    accuracy                           0.90      2046
   macro avg       0.54      0.63      0.55      2046
weighted avg       0.96      0.90      0.93      2046


Confussion matrix
[[1824  168]
 [  35   19]]
```

**Figure 1.2**

As we can see the model accuracy was 90% with precision and recall value of 96% and 90%.

Next (**Model 2**) we changed the method of criterion to see how the results vary, using gini criterion and Entropy criterion on the training set and use cv = 5. The accuracy on the training set was as follows.

```
Entropy criterion accuracy on cv: 0.959
Gini criterion accuracy on cv: 0.956
```

On the training set entropy criterion performs better so we will run test set on that, Testing accuracy and classification report is as follows:

```
Accuracy:0.924

Classification report
              precision    recall  f1-score   support

           0       0.99      0.94      0.96      1992
           1       0.17      0.50      0.26        54

    accuracy                           0.92      2046
   macro avg       0.58      0.72      0.61      2046
weighted avg       0.96      0.92      0.94      2046
```

**Figure 1.3**

**Model 3**, we change the depth of the tree and minimum samples of the leaf to see the accuracy scores, parameters were max_depth=3 and min_samples_leaf=10 with classification parameter criterion = ‘entropy’.

```
Accuracy:0.865

Classification report
              precision    recall  f1-score   support

           0       1.00      0.86      0.93      1992
           1       0.15      0.85      0.25        54

    accuracy                           0.86      2046
   macro avg       0.57      0.86      0.59      2046
weighted avg       0.97      0.86      0.91      2046
```

**Figure 1.4**

Evaluating all the three results the best accuracy score was achieved by the second model of decision tree with entropy criterion and cv=5 , i.e. 92% accuracy.

### Random Forest

The results of the random forest showed that the best classifier criteria used a gini criterion, a max-depth of 50, 2 as the minimum samples to split on, and 100 estimators (trees). The results on the testing set showed this model improved upon logistic regression, having an overall accuracy of 95.0% with a recall of 50.0% and precision of 35.3% on the minority class. Compared to logistic regression, random forest was able to improve upon the accuracy and precision scores, but suffered in recall. Accurately predicting bankruptcies seemed challenging with random forest since this classifier overfit the training data. While training, random forest had accuracies around 98-99% for each fold, while it didn't perform this well on the testing set.

Classification Report for Random Forest:

```
              precision    recall  f1-score   support

           0     0.9815    0.9666    0.9740      1316
           1     0.3529    0.5000    0.4138        48

    accuracy                         0.9501      1364
   macro avg     0.6672    0.7333    0.6939      1364
weighted avg     0.9594    0.9501    0.9543      1364
```

**Figure 1.5**

## Boosting Algorithms

We applied several boosting algorithms on the dataset to predict bankruptcy The following are the models we used: XGBoost, AdaBoost, CatBoost and a Light GBM

model. For the evaluation of algorithms a custom function was built to generate the ROC curves, precision, recall and confusion matrix, to get faster results at one go.

## XGBoost:

XGBoost implements machine learning algorithms under the Gradient Boosting framework. It is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. The parallel tree boosting feature of XGBoost helps solve data science problems in a fast and accurate way. Moreover, the code can run on different / major distributed environments. However, XGBoost lacks scalability.

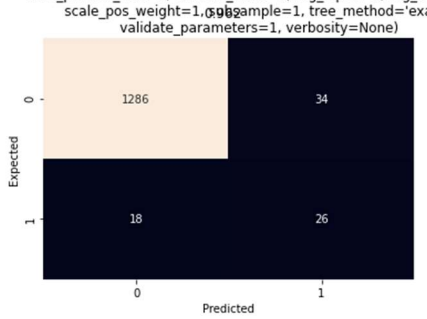
We selected the following optimizing parameters while training the model, n_estimators, max_depth, min_samplws_split, min_samples_leaf, eval_metric:'logloss'.
The model's best score for this metric was: 0.986. I.e., **98.6%** for the **training set**.

Model Parameters were as follows:

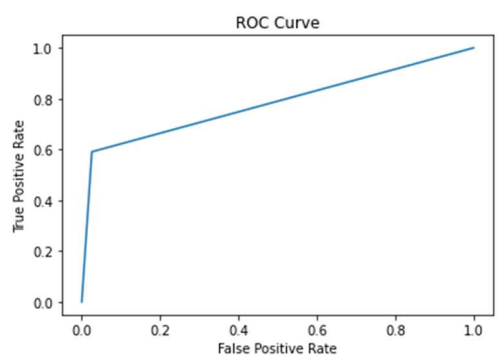
```
Model Best Score:
 0.9862663560153969
Model Best Parameters:
 {'n_estimators': 926, 'min_samples_split': 7, 'min_samples_leaf': 5, 'max_depth': 20, 'eval_metric': 'logloss', 'alpha:': 5}
```

Below are the results obtained from XGBoost. Evaluation metrics used are the Classification report and confusion matrix.

```
XGBClassifier(alpha=5, base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, eval_metric='logloss',
              gamma=0, gpu_id=-1, importance_type='gain',
              interaction_constraints='', learning_rate=0.300000012,
              max_delta_step=0, max_depth=20, min_child_weight=1,
              min_samples_leaf=5, min_samples_split=7, missing=nan,
              monotone_constraints='()', n_estimators=926, n_jobs=16,
              num_parallel_tree=1, random_state=0, reg_alpha=5, reg_lambda=1,
              scale_pos_weight=1, subsample=1, tree_method='exact',
              validate_parameters=1, verbosity=None)
```




```
Accuracy Score:
 0.962
              precision    recall  f1-score   support

           0       0.99      0.97      0.98      1320
           1       0.43      0.59      0.50        44

    accuracy                           0.96      1364
   macro avg       0.71      0.78      0.74      1364
weighted avg       0.97      0.96      0.96      1364
```



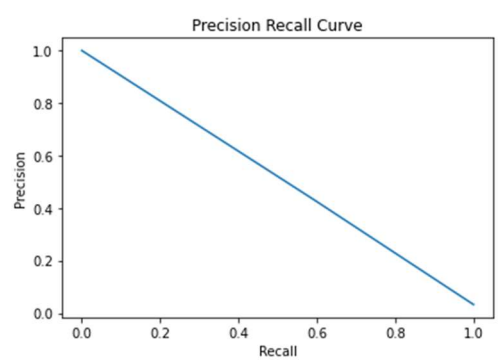



**Figure 1.6**

The tuned XGBoost model was the second highest performing classifier with a 96.2% accuracy score on the test set. It has 96% recall and 97% precision and F1 score of 96%. The recall for the minority class was 59%. The Roc graph is towards the axis showing good fit and accurate results on the test data.

## AdaBoost:

AdaBoost stands for Adaptive Boosting, it is a meta machine learning algorithm. It is a meta-estimator that fits the classifier on the original dataset and then fits additional copies of the classifier on the same dataset. The incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.
The algorithm is implemented as AdaBoost-SAMME.R, here SAMME.R algorithm achieves lower test error with fewer boosting iterations.

**Pros:** Fast, flexible to combine with any machine learning algorithm, simple and easy to implement. It requires minimal tuning.

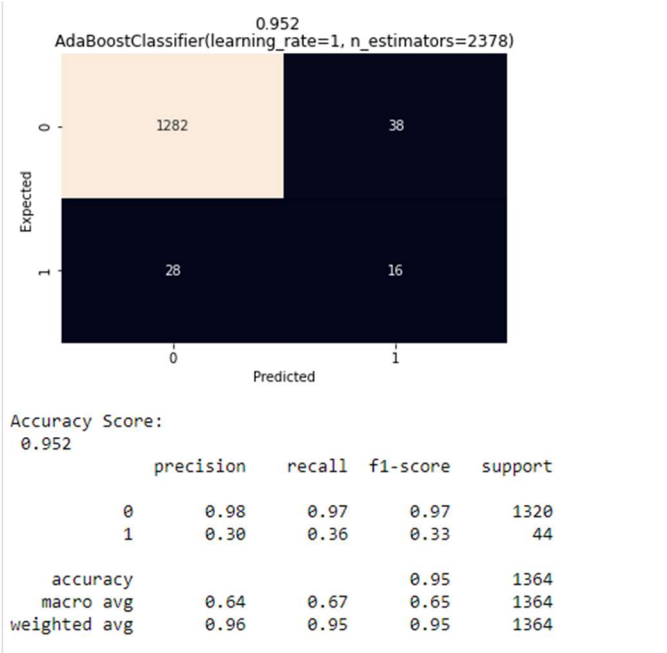
**Cons:** It is vulnerable to uniform noise and weak classifiers being too weak can lead to low margins and overfitting.

Let us look at the results and performance of the algorithm on bankruptcy data.
The training model was built on AdaBoostClassifier() and the following parameters: n_estimators, learning_rate an algorithm (SAMME/SAMME.R).

**Model estimate best score** on the training set is 0.977 i.e., 97.7%

```
Model Best Score:
 0.977553185305743
Model Best Parameters:
 {'n_estimators': 2378, 'learning_rate': 1, 'algorithm': 'SAMME.R'}
```

**Evaluation parameters, confusion matrices and ROC curve, precision-recall, F1 score.**



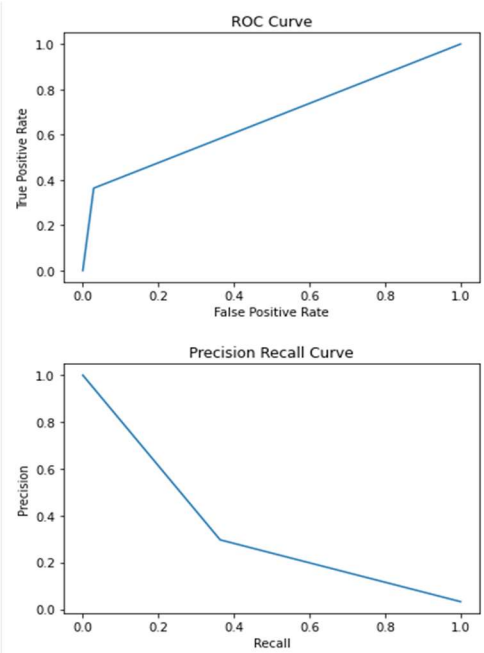




**Figure 1.7**

The above evaluation matrices shows that the model has performed well with 96% precision and 95% recall value, and showing 95.2% accuracy on F1-scores. The Roc curve is inclined towards the axis showing test accuracy. Also, the test sample uses learning_rate=1 and n_estimates = 2378 trees. Only 36% of the minority class was correctly identified as bankrupt.

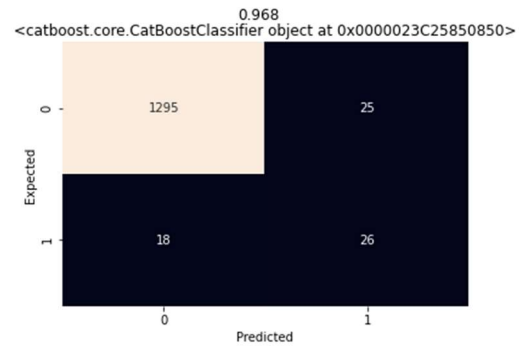
## CatBoost:

CatBoost stands for Category Gradient Boosting, this is because it supports categorical input variables. It is used for training and applying models on classification problems, and uses scikit-learn tools for stability. It supports binomial and continuous target attributes.
Exploring the training model and parameter selection, the model is trained on CatBoostClassifier(), with learning_rate, depth, l2_lear_reg, iterations, parameters.

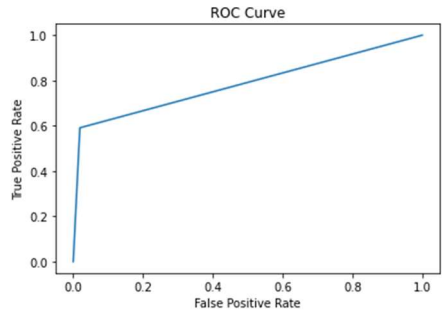
**The model best score on the training set** is 0.977, i.e., **98%** below are the training models selected by the classifier.

```
Model Best Score:
 0.9821934713718602
Model Best Parameters:
 {'learning_rate': 0.1, 'l2_leaf_reg': 1, 'iterations': 150, 'depth': 10}
```

**Evaluation parameters, confusion matrices and ROC curve, precision-recall, F1 score.**




```
Accuracy Score:
 0.968
              precision    recall  f1-score   support

           0       0.99      0.98      0.98      1320
           1       0.51      0.59      0.55        44

    accuracy                           0.97      1364
   macro avg       0.75      0.79      0.77      1364
weighted avg       0.97      0.97      0.97      1364
```



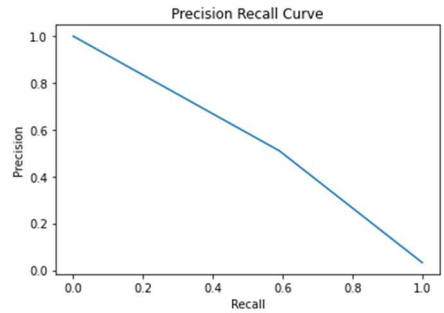




**Figure 1.8**

The above evaluation results show, the model has given 96.8% accuracy and the ROC curve aligns on the axis showing accurate test results. Recall and precision values are 97%.
The final model on the boosting algorithm list is LightGBM, which is mentioned below.

## Light GBM:

LightGBM is a gradient boosting framework that uses tree based learning algorithms. It is designed to be distributed and has the following advantages:

- Faster training speed and higher efficiency.
- Lower memory usage.
- Better accuracy.
- Support of parallel, distributed and GPU learning.
- Capable of handling large-scale data.

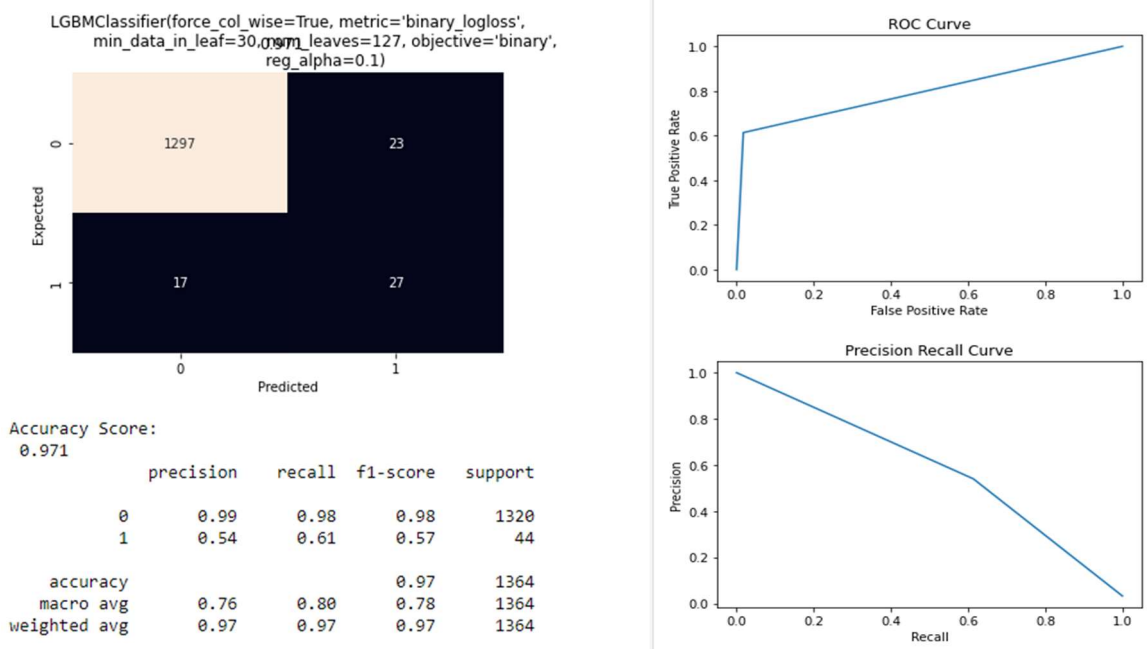
LightGBM is used in regression , classification, predictive modeling tasks with tabular data. This algorithm also takes support of scikit-learn algorithm. Most boosting models, including XGBoost, grow by tree level, while LightGBM grows trees leaf-wise.

Let us discuss the parameters used for the model building in the bankruptcy project.
The LGBMClassifier() classifier is used to build the training model, parameters for the training model are num_leaves, reg_alpha, min_data_in_leaf, lambda_l1, lambda_l2.

**Model Best Score on Training Set**: 0.986, i.e., 98.6%

```
Model Best Score:
 0.9861716590457
Model Best Parameters:
 {'reg_alpha': 0.1, 'num_leaves': 127, 'min_data_in_leaf': 30, 'lambda_l2': 0, 'lambda_l1': 0}
```

**Evaluation parameters, confusion matrix, roc curve, precision - recall and accuracy value**




```
Accuracy Score:
 0.971
              precision    recall  f1-score   support

           0       0.99      0.98      0.98      1320
           1       0.54      0.61      0.57        44

    accuracy                           0.97      1364
   macro avg       0.76      0.80      0.78      1364
weighted avg       0.97      0.97      0.97      1364
```

**Figure 1.9**

From the above evaluation metrics and roc curves we can easily see that LightGBM gives the most accurate results. LightGBM mode gives accuracy of 97.1%, with precision-recall values of 97%. The recall for the minority class was 61%. Also, the roc curve is aligned to the axis giving accuracy for test data. The LightGBM classifier outperformed other boosting methods, and it had a faster computation time.

# Support Vector Machine

Support Vector Machines are supervised learning methods used for classification, regression and outlier detection. Advantages of svm are as follows:

- It is effective with high dimensional spaces.
- Effective when the number of data samples are less than the number of dimensions.
- Memory efficient
- Different custom kernels can be specified based on the decision functions.

Disadvantages:

- When the features are greater than the number of samples, avoiding overfitting in kernel functions and regularization term is crucial.
- We have to use cross validation to provide probability estimates which can be expensive.

- We need to use cross validation to get probability estimates, as SVM does not provide it directly.

To predict bankruptcy, we also tried using SVM using PCA, for SVM we used two methods or kernels. Linear kernel and rbf kernel. Following are the results and evaluation metrics for both the kernel methods using SVM.

## SVM using linear Kernel:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.99 | 0.85 | 0.92 | 1992 |
| 1 | 0.13 | 0.83 | 0.23 | 54 |
| accuracy | | | 0.85 | 2046 |
| macro avg | 0.56 | 0.84 | 0.57 | 2046 |
| weighted avg | 0.97 | 0.85 | 0.90 | 2046 |

**Figure 2.0**

From the classification report, we can observe the linear kernel gives the accuracy of 85%  percent with precision and recall values of 97% and 85%.

## SVM using rbf kernel:

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.98 | 0.94 | 0.96 | 1992 |
| 1 | 0.18 | 0.46 | 0.26 | 54 |
| accuracy | | | 0.93 | 2046 |
| macro avg | 0.58 | 0.70 | 0.61 | 2046 |
| weighted avg | 0.96 | 0.93 | 0.94 | 2046 |

**Figure 2.1**

Accuracy of the model using rbf is 93%, with precision and recall value of 96% and 93%. From the classification report it is clear that the rbf kernel gives better accuracy than the linear kernel, we will select the SVM rbf model to compare with other models in the project.

# Feature Importances

In order to provide more explanation to our models, we analyzed the feature importance of logistic regression (benchmark) and XGBoost, our best classification method.

Overall, the 5 most important features by coefficient values for logistic regression are as follows: Continuous interest rate (after tax) (negative coef), Pre-tax net Interest Rate, Cash Flow to Sales, Operating Profit Rate, and Working capital Turnover Rate (negative coef). Below is the graph for all features.

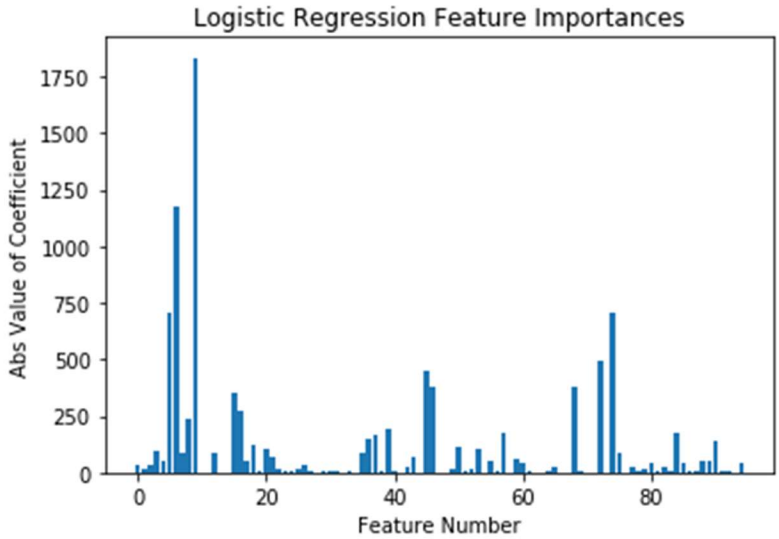



**Figure 2.2**

For XGBoost, one of the best classifiers, a single feature (Persistent EPS in the Last Four Seasons, negative coef) dominated the model. Other important features (all positive coef) were Net Income to Stickholder's Equity, Borrowing dependency, Net Value Growth Rate, and Non-industry income and expenditure/revenue.

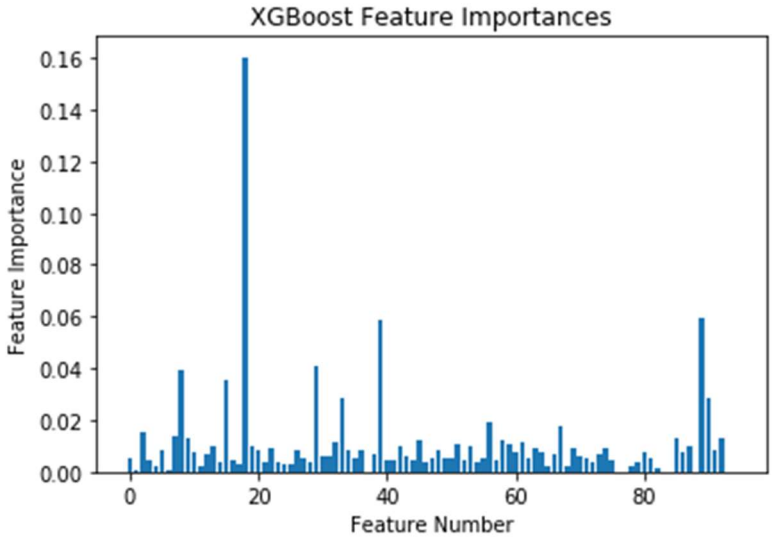



**Figure 2.3**

For Random Forest, there was a much wider spread of important features. Instead of having a few features dominate the model, there was a much wider spread of feature importances in this classification.

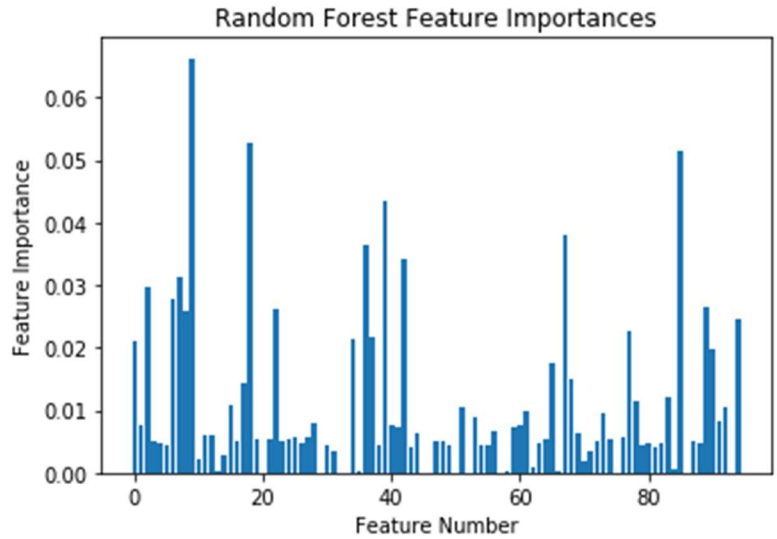



**Figure 2.4**

For LGBM, our best classifier, we had an even wider spread of features that were important. The most important features for this classifier were Interest-bearing debt interest rate, Cash Turnover Rate, Research and development expense rate, Inventory Turnover Rate (times), and Current Asset Turnover Rate.

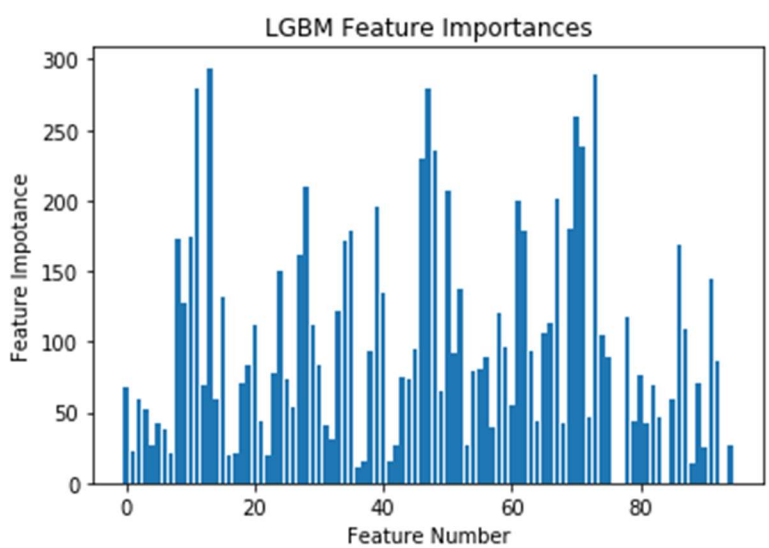



**Figure 2.5**

Overall, each of our models gave different features importances after training. Our best classifier, LGBM, had a wide spread of feature importances. Other classifiers, such as Logistic Regression and XGBoost, showed a few sets of features were very important.

# MODEL MANAGEMENT

In order to accelerate the large amount of models that we wanted to train/test, we implemented a 3-node cluster using Google Cloud Platform (GCP). This cluster was quite powerful, with 32 CPU with 64 gigs of RAM and two storage buckets. We encountered many networking issues in order to maintain and optimize the network and this led to very high initial upfront costs.

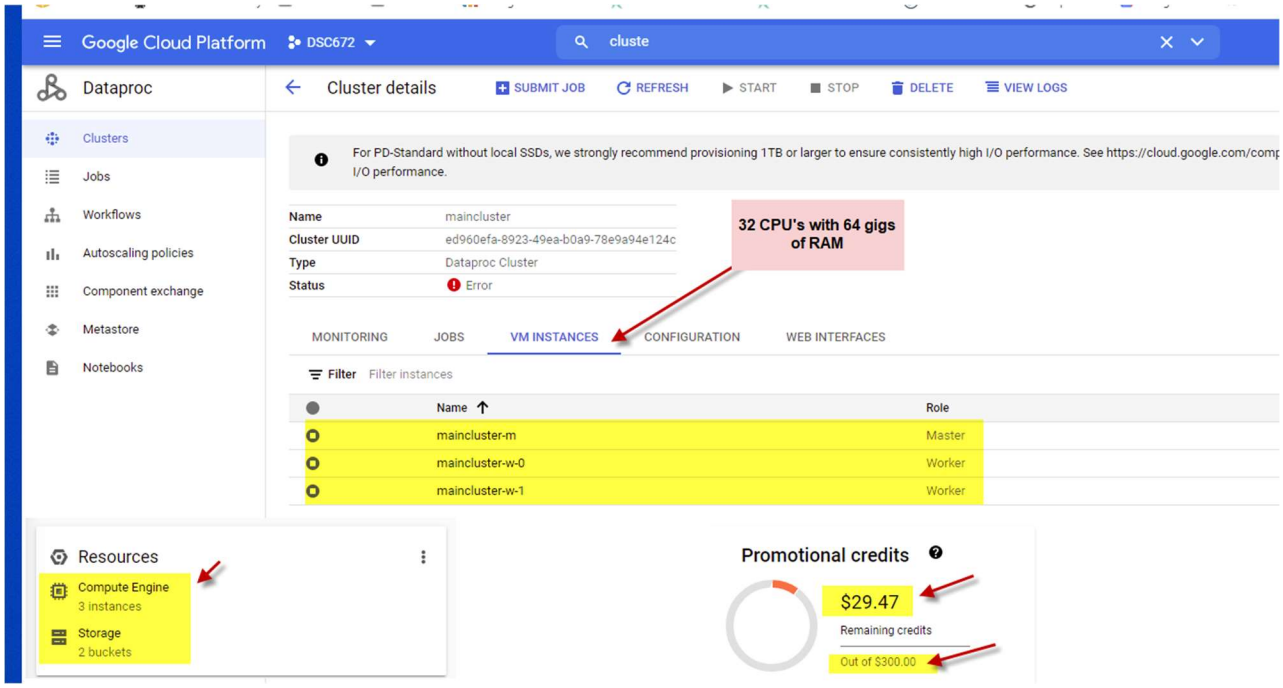



**Figure 2.6**

It took us around 5 hours to train/test fifteen of the most popular models (with and without PCA/Hyperparameter tuning). We also included two powerful stack models in our analysis. Overall, using GCP was very expensive and time-consuming.

**Below you can see the final results of our runs in GCP:**

**NO PCA - No Hyperparamerter Optimization**

10-Fold Cross Validation

| | Model ID | Model | accuracy | f1_score | precision | recall | specificity | roc_auc | pr_auc |
|---|---|---|---|---|---|---|---|---|---|
| 1 | STACK1 | Stack Model 1 | 0.9685 | 0.3175 | 0.6250 | 0.2128 | 0.9954 | 0.8266 | 0.3073 |
| 2 | STACK2 | Stack Model 2 | 0.9516 | 0.1951 | 0.2286 | 0.1702 | 0.9795 | 0.8504 | 0.2030 |
| 3 | SVC | SupportVectorClassifier | 0.8717 | 0.2616 | 0.1632 | 0.6596 | 0.8793 | 0.8923 | 0.2289 |
| 4 | LR | LogisticRegression | 0.8688 | 0.2811 | 0.1733 | 0.7447 | 0.8732 | 0.8999 | 0.2142 |
| 5 | LDA | LinearDiscriminantAnalysis | 0.8504 | 0.2500 | 0.1511 | 0.7234 | 0.8550 | 0.9094 | 0.2148 |
| 6 | KNN | KNeighborsClassifier | 0.8827 | 0.2381 | 0.1534 | 0.5319 | 0.8952 | 0.7660 | 0.1916 |
| 7 | CART | DecisionTreeClassifier | 0.9106 | 0.2179 | 0.1560 | 0.3617 | 0.9301 | 0.6459 | 0.2698 |
| 8 | NB | GaussianNaiveBayes | 0.7097 | 0.1750 | 0.0970 | 0.8936 | 0.7031 | 0.8736 | 0.3500 |
| 9 | NN | MLPClassifier | 0.9230 | 0.2657 | 0.1979 | 0.4043 | 0.9415 | 0.8775 | 0.2247 |
| 10 | AB | AdaBoostClassifier | 0.9113 | 0.3388 | 0.2279 | 0.6596 | 0.9203 | 0.8905 | 0.2523 |
| 11 | GBM | GradientBoostingClassifier | 0.9230 | 0.3396 | 0.2411 | 0.5745 | 0.9355 | 0.9192 | 0.3159 |
| 12 | CAT | CatBoostClassifier | 0.9641 | 0.4731 | 0.4783 | 0.4681 | 0.9818 | 0.9325 | 0.4503 |
| 13 | XGB | XGBClassifier #XGBoost | 0.9619 | 0.4091 | 0.4390 | 0.3830 | 0.9825 | 0.9375 | 0.3757 |
| 14 | LGBM | LGBMClassifier #LightBoost | 0.9589 | 0.4167 | 0.4082 | 0.4255 | 0.9780 | 0.9355 | 0.3592 |
| 15 | RF | RandomForestClassifier | 0.9457 | 0.3621 | 0.3043 | 0.4468 | 0.9636 | 0.9159 | 0.3470 |
| 16 | ETR | ExtraTreesClassifier | 0.9450 | 0.3243 | 0.2813 | 0.3830 | 0.9651 | 0.9088 | 0.3011 |
| 17 | BAG | BaggingClassifier(KNeighborsClassifier) | 0.8856 | 0.2500 | 0.1615 | 0.5532 | 0.8975 | 0.7761 | 0.2004 |

**With PCA - No Hyperparamerter Optimization**

10-Fold Cross Validation

| | Model ID | Model | accuracy | f1_score | precision | recall | specificity | roc_auc | pr_auc |
|---|---|---|---|---|---|---|---|---|---|
| 1 | STACK1 | Stack Model 1 | 0.9597 | 0.0984 | 0.2143 | 0.0638 | 0.9916 | 0.7508 | 0.2068 |
| 2 | STACK2 | Stack Model 2 | 0.9611 | 0.2535 | 0.3750 | 0.1915 | 0.9886 | 0.7825 | 0.2083 |
| 3 | SVC | SupportVectorClassifier | 0.9223 | 0.3026 | 0.2190 | 0.4894 | 0.9377 | 0.8683 | 0.2303 |
| 4 | LR | LogisticRegression | 0.8592 | 0.2615 | 0.1596 | 0.7234 | 0.8641 | 0.8914 | 0.1921 |
| 5 | LDA | LinearDiscriminantAnalysis | 0.8358 | 0.2329 | 0.1388 | 0.7234 | 0.8398 | 0.8751 | 0.1690 |
| 6 | KNN | KNeighborsClassifier | 0.8937 | 0.2328 | 0.1549 | 0.4681 | 0.9089 | 0.7612 | 0.2164 |
| 7 | CART | DecisionTreeClassifier | 0.9047 | 0.1558 | 0.1122 | 0.2553 | 0.9279 | 0.5916 | 0.1966 |
| 8 | NB | GaussianNaiveBayes | 0.7940 | 0.1808 | 0.1047 | 0.6596 | 0.7988 | 0.7909 | 0.1506 |
| 9 | NN | MLPClassifier | 0.9318 | 0.2439 | 0.1974 | 0.3191 | 0.9537 | 0.8399 | 0.1964 |
| 10 | AB | AdaBoostClassifier | 0.8695 | 0.2583 | 0.1606 | 0.6596 | 0.8770 | 0.8645 | 0.1942 |
| 11 | GBM | GradientBoostingClassifier | 0.8930 | 0.2843 | 0.1847 | 0.6170 | 0.9028 | 0.8619 | 0.2100 |
| 12 | CAT | CatBoostClassifier | 0.9472 | 0.3077 | 0.2807 | 0.3404 | 0.9689 | 0.8636 | 0.2018 |
| 13 | XGB | XGBClassifier #XGBoost | 0.9501 | 0.3200 | 0.3019 | 0.3404 | 0.9719 | 0.8670 | 0.1993 |
| 14 | LGBM | LGBMClassifier #LightBoost | 0.9377 | 0.2735 | 0.2286 | 0.3404 | 0.9590 | 0.8641 | 0.1948 |
| 15 | RF | RandomForestClassifier | 0.9465 | 0.2316 | 0.2292 | 0.2340 | 0.9719 | 0.8805 | 0.1895 |
| 16 | ETR | ExtraTreesClassifier | 0.9567 | 0.2133 | 0.2857 | 0.1702 | 0.9848 | 0.8801 | 0.2044 |
| 17 | BAG | BaggingClassifier(KNeighborsClassifier) | 0.8930 | 0.2474 | 0.1633 | 0.5106 | 0.9066 | 0.9066 | 0.2097 |

**Figure 2.7**

**NO PCA - With Hyperparamerter Tuning (using 10-Fold cross-validations)**

| | Model ID | Model | accuracy | f1_score | precision | recall | specificity | roc_auc | pr_auc | Time | HyperParameters |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SVC | SupportVectorClassifier | 0.9054 | 0.2873 | 0.1940 | 0.5532 | 0.9180 | 0.8767 | 0.2372 | 3min 26s | {'C': 20, 'kernel': 'poly'} |
| 2 | LR | LogisticRegression | 0.8768 | 0.2881 | 0.1799 | 0.7234 | 0.8823 | 0.8991 | 0.2513 | 28min 20s | {'C': 20, 'dual': False, 'max_iter': 100, 'penalty': 'l1', 'solver': 'liblinear'} |
| 3 | LDA | LinearDiscriminantAnalysis | 0.7691 | 0.1463 | 0.0839 | 0.5745 | 0.7760 | 0.7712 | 0.1385 | 1.1 s | {'shrinkage': 1, 'solver': 'lsqr'} |
| 4 | KNN | KNeighborsClassifier | 0.8834 | 0.2392 | 0.1543 | 0.5319 | 0.8960 | 0.7688 | 0.1965 | 1min 12s | {'algorithm': 'auto', 'leaf_size': 10, 'weights': 'distance'} |
| 5 | CART | DecisionTreeClassifier | 0.9267 | 0.2537 | 0.1954 | 0.3617 | 0.9468 | 0.6538 | 0.2881 | 12 s | {'criterion': 'gini', 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 3, 'splitter': 'best'} |
| 6 | NB | GaussianNaiveBayes | 0.8424 | 0.2561 | 0.1529 | 0.7872 | 0.8443 | 0.8759 | 0.2629 | 2.26 s | {'var_smoothing': 0.0001} |
| 7 | NN | MLPClassifier | 0.9069 | 0.2983 | 0.2015 | 0.5745 | 0.9188 | 0.8907 | 0.2095 | 6min 43s | {'activation': 'relu', 'alpha': 0.0001, 'learning_rate': 'adaptive', 'solver': 'adam'} |
| 8 | AB | AdaBoostClassifier | 0.9172 | 0.3392 | 0.2339 | 0.6170 | 0.9279 | 0.8972 | 0.2920 | 4min 23s | {'learning_rate': 0.5, 'n_estimators': 95} |
| 9 | GBM | GradientBoostingClassifier | 0.9443 | 0.3770 | 0.3067 | 0.4894 | 0.9605 | 0.8777 | 0.3108 | 10min 13s | {'learning_rate': 1.0, 'loss': 'deviance', 'max_depth': 2, 'n_estimators': 100} |
| 10 | CAT | CatBoostClassifier | 0.9421 | 0.2883 | 0.2500 | 0.3404 | 0.9636 | 0.8735 | 0.2400 | 2min 15s | {'border_count': 10, 'depth': 2, 'iterations': 100, 'l2_leaf_reg': 100, 'learning_rate': 0.9} |
| 11 | XGB | XGBClassifier #XGBoost | 0.9619 | 0.3500 | 0.4242 | 0.2979 | 0.9856 | 0.9183 | 0.3360 | 2h 35min 32s | {'gamma': 0.01, 'learning_rate': 0.1, 'max_depth': 2, 'n_estimators': 3000} |
| 12 | LGBM | LGBMClassifier #LightBoost | 0.9563 | 0.3960 | 0.3704 | 0.4255 | 0.9742 | 0.9164 | 0.2810 | 1min 20s | {'learning_rate': 1.0, 'max_depth': 2, 'n_estimators': 95} |
| 13 | RF | RandomForestClassifier | 0.7742 | 0.1852 | 0.1057 | 0.7447 | 0.7752 | 0.8797 | 0.3642 | 14.6 s | {'criterion': 'entropy', 'max_depth': 1, 'min_samples_leaf': 1, 'n_estimators': 25} |
| 14 | ETR | ExtraTreesClassifier | 0.8409 | 0.2439 | 0.1458 | 0.7447 | 0.8443 | 0.8952 | 0.2266 | 25.9 s | {'max_depth': 2, 'min_samples_leaf': 1, 'n_estimators': 65} |
| 15 | BAG | BaggingClassifier(KNeighborsClassifier) | 0.9399 | 0.3692 | 0.2892 | 0.5106 | 0.9552 | 0.8930 | 0.2457 | 10min 55s | {'max_features': 0.5, 'max_samples': 1.0, 'n_estimators': 40} |

**With PCA - With Hyperparamerter Tuning (using 10-Fold cross-validations)**

| | Model ID | Model | accuracy | f1_score | precision | recall | specificity | roc_auc | pr_auc | Time | HyperParameters |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | SVC | SupportVectorClassifier | 0.9450 | 0.2857 | 0.2586 | 0.3191 | 0.9674 | 0.8400 | 0.2023 | 53.5 s | {'C': 20, 'kernel': 'rbf'} |
| 2 | LR | LogisticRegression | 0.8636 | 0.2677 | 0.1643 | 0.7234 | 0.8686 | 0.8940 | 0.2032 | 8.86 s | {'C': 5, 'dual': True, 'max_iter': 100, 'penalty': 'l2', 'solver': 'liblinear'} |
| 3 | LDA | LinearDiscriminantAnalysis | 0.8167 | 0.1830 | 0.1081 | 0.5957 | 0.8246 | 0.8338 | 0.1792 | 354 ms | {'shrinkage': 1, 'solver': 'lsqr'} |
| 4 | KNN | KNeighborsClassifier | 0.8988 | 0.2660 | 0.1773 | 0.5319 | 0.9119 | 0.7691 | 0.2397 | 12.3 s | {'algorithm': 'auto', 'leaf_size': 10, 'weights': 'distance'} |
| 5 | CART | DecisionTreeClassifier | 0.9128 | 0.2013 | 0.1471 | 0.3191 | 0.9339 | 0.6260 | 0.2439 | 7.84 s | {'criterion': 'entropy', 'max_features': 'log2', 'min_samples_leaf': 1, 'min_samples_split': 3, 'splitter': 'best'} |
| 6 | NB | GaussianNaiveBayes | 0.7845 | 0.1923 | 0.1104 | 0.7447 | 0.7859 | 0.8177 | 0.1940 | 790 ms | {'var_smoothing': 0.08111308307896872} |
| 7 | NN | MLPClassifier | 0.9435 | 0.2376 | 0.2222 | 0.2553 | 0.9681 | 0.8501 | 0.2201 | 4min 42s | {'activation': 'relu', 'alpha': 0.0001, 'learning_rate': 'adaptive', 'solver': 'lbfgs'} |
| 8 | AB | AdaBoostClassifier | 0.8768 | 0.2500 | 0.1582 | 0.5957 | 0.8869 | 0.8555 | 0.2181 | 1min 1s | {'learning_rate': 0.5, 'n_estimators': 95} |
| 9 | GBM | GradientBoostingClassifier | 0.9142 | 0.2642 | 0.1875 | 0.4468 | 0.9309 | 0.8518 | 0.1882 | 2min 6s | {'learning_rate': 1.0, 'loss': 'exponential', 'max_depth': 2, 'n_estimators': 100} |
| 10 | CAT | CatBoostClassifier | 0.8871 | 0.2451 | 0.1592 | 0.5319 | 0.8998 | 0.8436 | 0.1742 | 1min 7s | {'border_count': 10, 'depth': 2, 'iterations': 100, 'l2_leaf_reg': 100, 'learning_rate': 0.9} |
| 11 | XGB | XGBClassifier #XGBoost | 0.8409 | 0.2439 | 0.1458 | 0.7447 | 0.8443 | 0.8952 | 0.2266 | 25.9 s | {'max_depth': 2, 'min_samples_leaf': 1, 'n_estimators': 65} |
| 12 | LGBM | LGBMClassifier #LightBoost | 0.9289 | 0.2815 | 0.2159 | 0.4043 | 0.9476 | 0.8431 | 0.1703 | 20.3 s | {'learning_rate': 1.0, 'max_depth': 2, 'n_estimators': 95} |
| 13 | RF | RandomForestClassifier | 0.8152 | 0.2125 | 0.1245 | 0.7234 | 0.8185 | 0.8224 | 0.1544 | 9.92 s | {'criterion': 'gini', 'max_depth': 1, 'min_samples_leaf': 1, 'n_estimators': 75} |
| 14 | ETR | ExtraTreesClassifier | 0.7881 | 0.1623 | 0.0940 | 0.5957 | 0.7950 | 0.8089 | 0.1447 | 13.4 s | {'max_depth': 2, 'min_samples_leaf': 2, 'n_estimators': 100} |
| 15 | BAG | BaggingClassifier(KNeighborsClassifier) | 0.9289 | 0.1983 | 0.1622 | 0.2553 | 0.9529 | 0.8321 | 0.1437 | 3min 27s | {'max_features': 0.3, 'max_samples': 1.0, 'n_estimators': 50} |

**Figure 2.**

In order to test out the results we obtained via our GCP cluster, we compared it to the results from DataRobot. This platform is very user friendly and gives you many benefits:

1) It is very simple to enter your data. You can provide a csv file, an url, or connect it to a database
2) It is also very easy to analyze and explore your datasets. It runs up to 20 different threads simultaneously and it takes a few minutes to analyse a model set like ours (instead of 5 hours).
3) The environment has health checks on your connection to tell you if something is going on with your connection. GCP was very bad in proving accurate health checks on the network and this costs time and money.
4) You can run your Pyton/R/SAS models against an inventory of other 75 models that are proprietary to DataRobot.
5) Once all the best algorithms are compared, you can drill down into each model and choose the best one
6) If you wish, you can deploy this model to production and then connect it with a corresponding data connection so that it can continue providing predictions.
7) The deployment tool keeps alerting the users if the deployed model is suffering from data drifting or accuracy loss

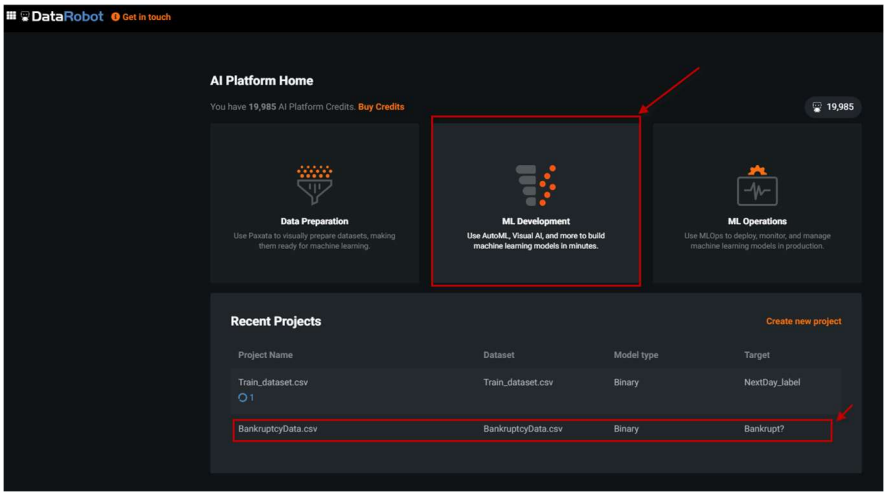



**Figure 2.9**

Once DataRobot compares all the models that you provide (and the ones provided by DataRobot) then you can drill down into the characteristics of each model.

For example, DataRobot's winning model below was able to suggest a stronger model that the one we calculated. You can see how you have the abily to drill down into the winning model and check out all the characteristics inside of it.

## Our winning Model using GCP:

| NO PCA - No Hyperparamerter Optimization | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 10-Fold Cross Validation | | | | | | | | | |
| | Model ID | Model | accuracy | f1_score | precision | recall | specificity | roc_auc | pr_auc |
| 12 | CAT | CatBoostClassifier | 0.9641 | 0.4731 | 0.4783 | 0.4681 | 0.9818 | 0.9325 | 0.4503 |

**Figure 3.0**

## Our winning Model using DataRobot:

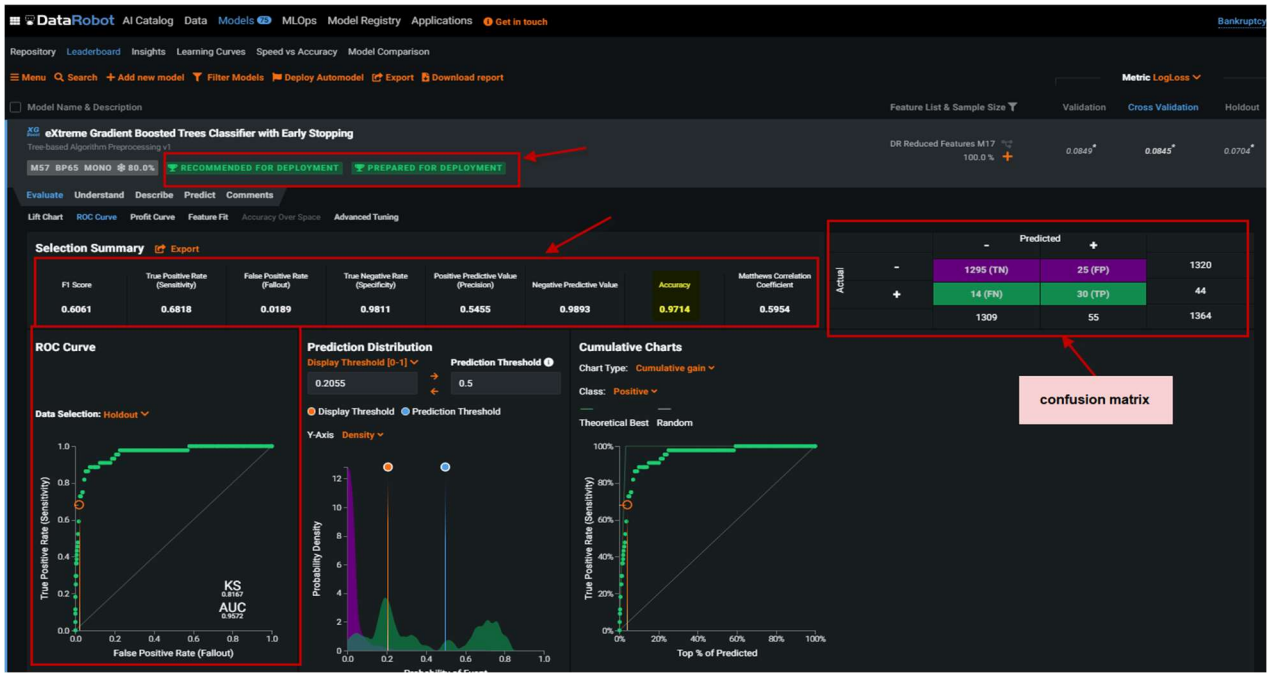



**Figure 3.1**

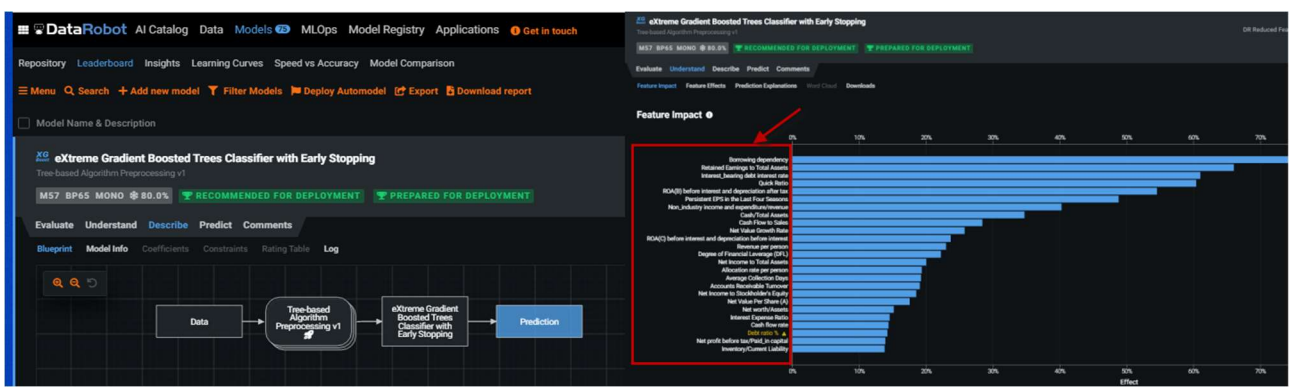



**Figure 3.2**

Once you ensure that the best model has been chosen, you can easily deploy that model to production as shown below:

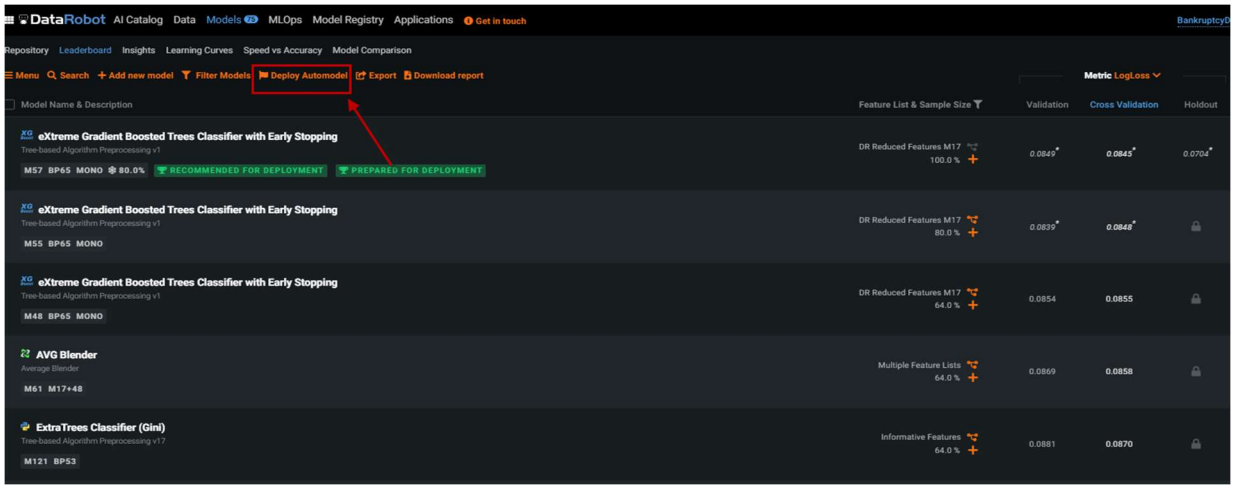



**Figure 3.3**

You can also register your own model by either entering it in Python directly. Or you can use this screen to continue to fine tune your model.

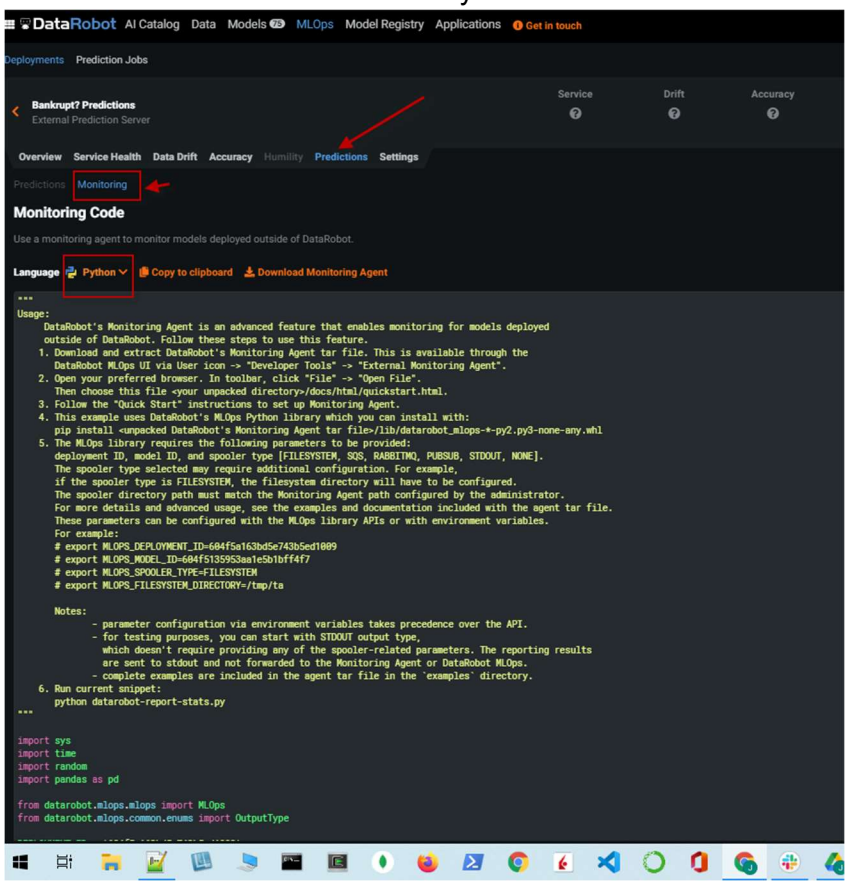



**Figure 3.4**

Or once you register the model you can go out to an external or internal location to pick up the code of the model. Here the model goes out to GitHub to pick up our internal code

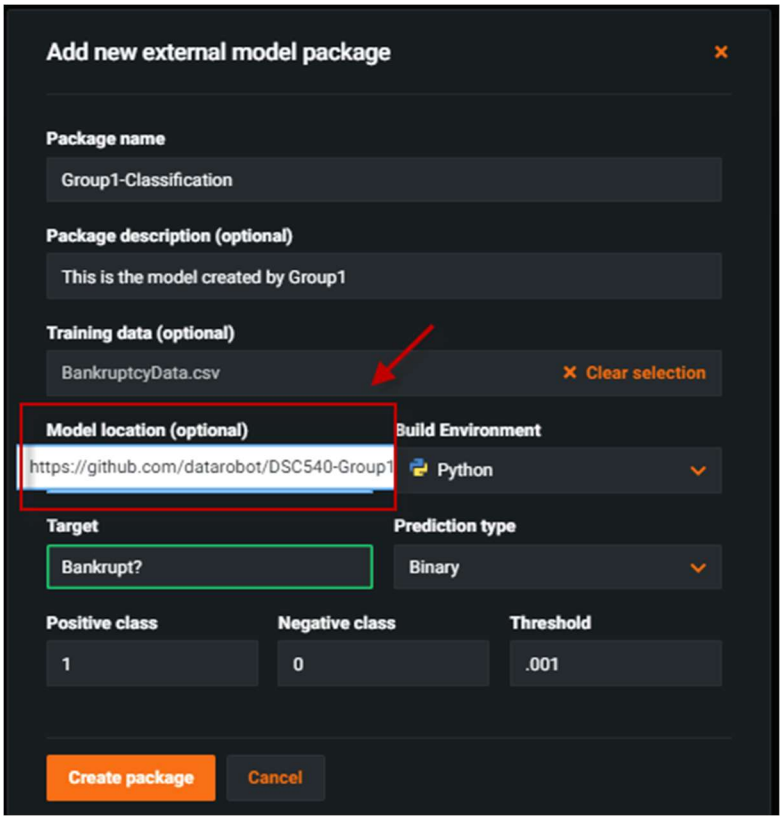



**Figure 3.5**

If anyone in the team wishes to fine tune the parameters of the model the can go to the "Evaluate" link and then click on "Advanced Tuning". Once they are done, they can deploy the model once again.

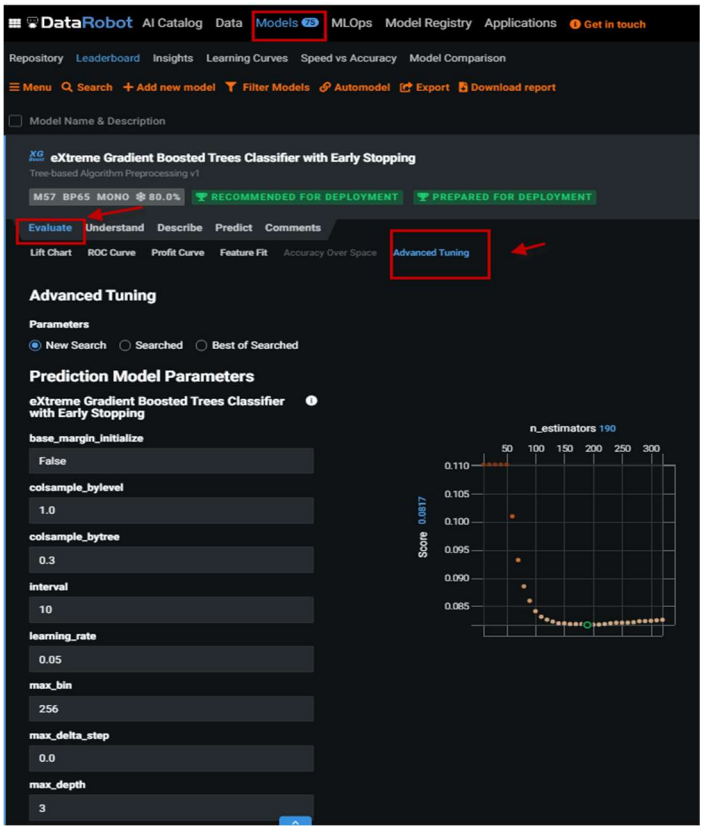


**Figure 3.6**

Once you deploy it you will see it registered under the “MLOps” hyperlink. Here you can check to check on the health of the models, the accuracy, and data drift as the characteristics of your dataset change with time.

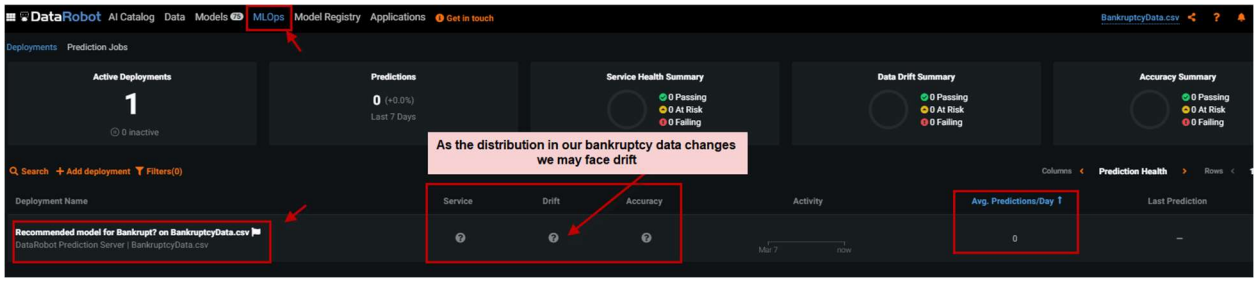



**Figure 3.7**

In order to integrate your model to a production dataset you can provide it the following:

1) You continue to manually provide it with csv files with your new data
2) You can connected to a data source such as Microsoft SQL Server, Tableau, or Snowflake
3) You can connected to another datasource via a REST api

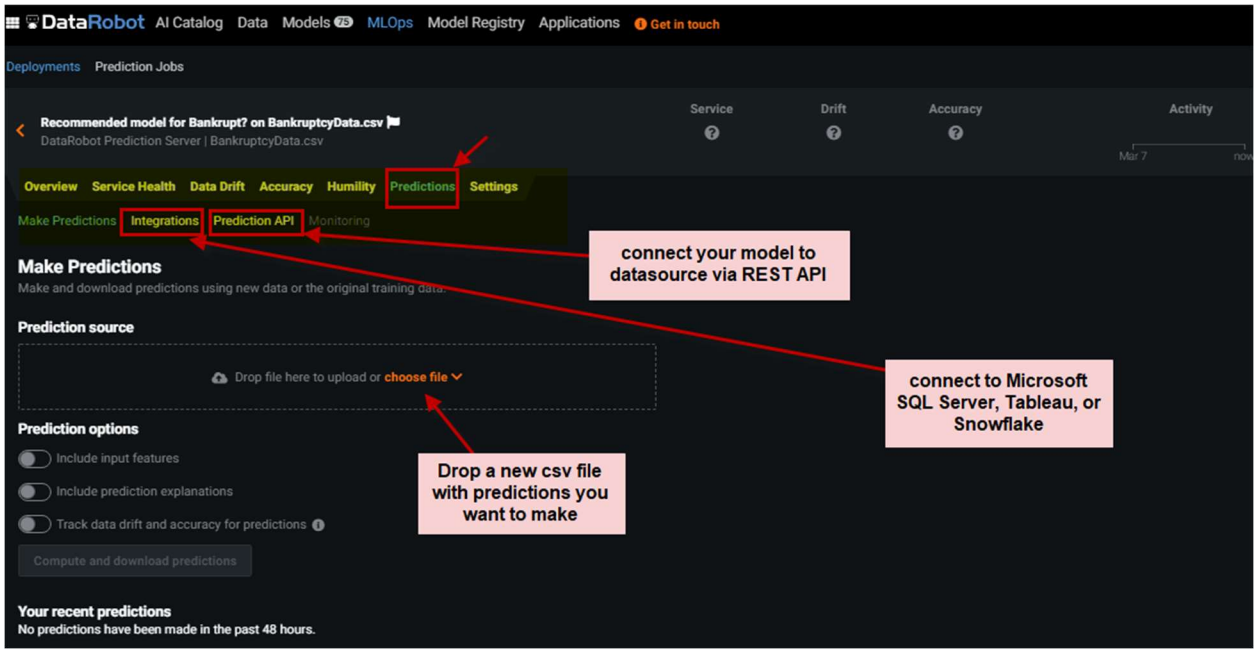



**Figure 3.8**

Finally, anyone in the team has the ability to make as many optimizations, predictions, and what-if analysis using the “Applications” tab.

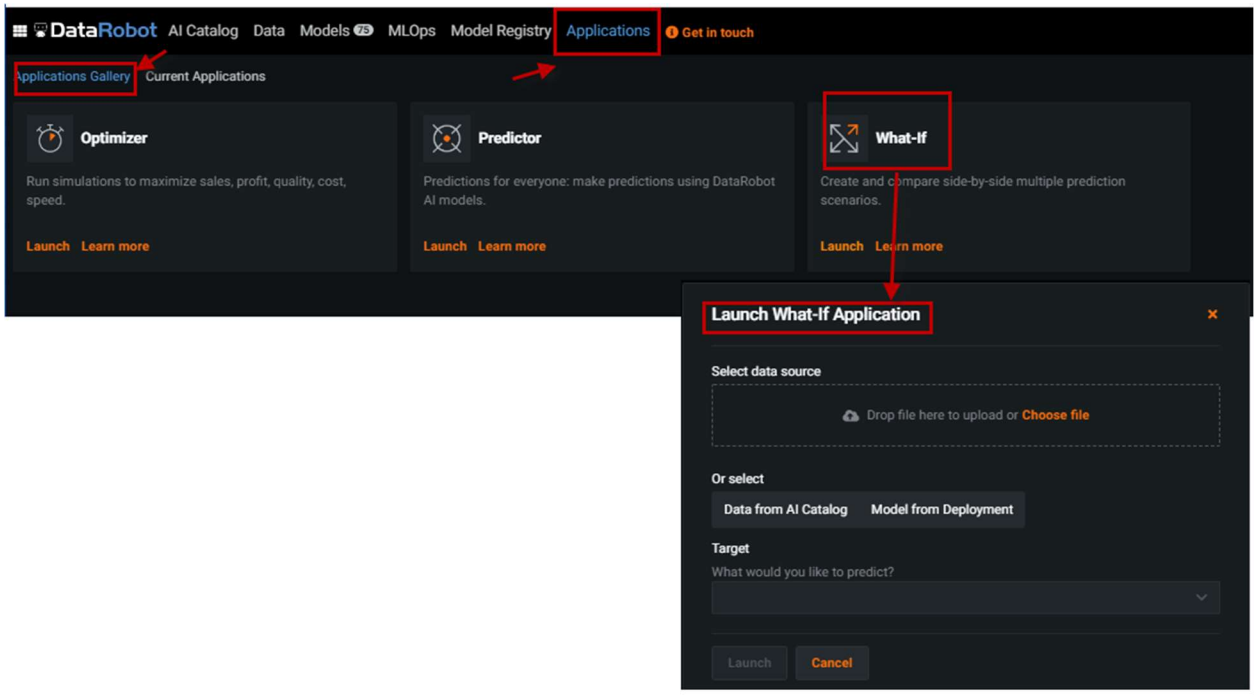



**Figure 3.9**

# CONCLUSION

The boosted models performed the best on the test set, followed by the tuned random forest. LightGBM has a slight edge over CatBoost and XGBoost, with an accuracy rate of 97.1% on the test set compared to 96.8% and 96.6% respectively. LightGBM had the shortest training and computation time of the set as well. Our best model correctly classified 61.4% of bankruptcy cases in the test set.