Project Overview

- Brief explanation of AUC and NDCG@k (mailto:NDCG@k) (write own function for NDCG@k (mailto:NDCG@k))
- Explain how RUSBoost works and why we use it
- Ran RUSBoost on this dataset – talk about parameters used and how you tried different numbers of trees
- Recreated some tables from the paper [discuss what these show]
- Results from RUSBoost – talk about differences between years
- Discuss weird ROC curve for 2008
- Future step includes Run RUSBoost on other datasets and Support Vector Learning

Paper Summary

This study aimed to develop a novel out-of-sample fraud prediction model by implementing ensemble learning. The authors focused on sample data of publicly traded U.S. firms over the period 1991–2008. This choice of time span could help them to avoid the influence of some major events that happened outside that time frame. To maintain the intertemporal nature of the model, they split the whole dataset into two parts: while treating year 2003 to 2008 as a test set, they took two years earlier data as train data for every test year. Besides the authors' own prediction model derived from RUSBoost method, they also evaluated two benchmark models: one is a logistic regression model from Dechow et al, and the other is a support vector machine model from Cecchini et al. In order to replicate those two models, the authors selected similar raw financial data items. By implementing AUC and NDCG@k (mailto:NDCG@k) performance evaluation metrics, they successfully demonstrated that their newly proposed model could yield more reliable predictions. Their findings are relative to the current accounting literature because accounting fraud is extremely hard to identify, and it requires a lot of resources to evaluate all firms. Therefore, a reliable model is extremely important for the government and investors.

Yang Bao, Bin Ke, Bin Li, Julia Yu, and Jie Zhang (2020). Detecting Accounting Fraud in Publicly Traded U.S. Firms Using a Machine Learning Approach. Journal of Accounting Research, 58 (1): 199-235.

Data Description

Taken from the Github:

The file "uscecchini28.csv" is our final dataset which contains the fraud labels and feature variables. The variable name of our fraud label is "misstate" (1 denotes fraud, and 0 denotes non-fraud). The variable names of the 28 raw financial data items are: act, ap, at, ceq, che, cogs, csho, dlc, dltis, dltt, dp, ib, invt, ivao, ivst, lct, lt, ni, ppegt, pstk, re, rect, sale, sstk, txp, txt, xint, prcc_f. The variable names of the 14 financial ratios are: dch_wc, ch_rsst, dch_rec, dch_inv, soft_assets, ch_cs, ch_cm, ch_roa, issue, bm, dpi, reoa, EBIT, ch_fcf. The variable new_p_aaer is used for identifying serial frauds as described in Section 3.3.

Annotated dictionary: *act* – *Current Assets, Total* ap – Account Payable, Trade *at* – *Assets, Total* ceq - - Common/Ordinary Equity, Total *che* – *Cash and Short-Term Investments* cogs – Cost of Goods Sold *csho* – *Common Shares Outstanding* dlc – Debt in Current Liabilities, Total *dltis* – *Long-Term Debt Issuance* dltt – Long-Term Debt, Total dp – Depreciation and Amortization *ib* – *Income Before Extraordinary Items* invt – Inventories, Total *ivao* – *Investment and Advances, Other* ivst – Short-Term Investments, Total *lct* – *Current Liabilities, Total* lt – Liabilities, Total ni – Net Income (Loss) *ppegt* – *Property, Plant and Equipment, Total* pstk – Preferred/Preference

Stock (Capital), Total re – Retained Earnings *rect – Receivables, Total* sale – Sales/Turnover (Net) *sstk – Sale of Common and Preferred Stock* txp – Income Taxes Payable txt – Income Taxes, Total xint – Interest and Related Expense, Total *prcc_f – Price Close, Annual, Fiscal

(Here after are 14 financial ratios) dch_wc – WC accruals ch_rsst – RSST accruals dch_rec – Change in receivables dch_inv – Change in inventory soft_assset – % Soft assets dpi – Depreciation index ch_cs – Change in cash sales ch_cm – Change in cash margin ch_roa – Change in return on assets ch_fcf – Change in free cash flows reoa – Retained earnings over total assets EBIT – Earnings before interest and taxes over total assets issue – Actual issuance bm – Book-to-market

Resources: https://github.com/JarFraud/FraudDetection (https://github.com/JarFraud/FraudDetection)

Code

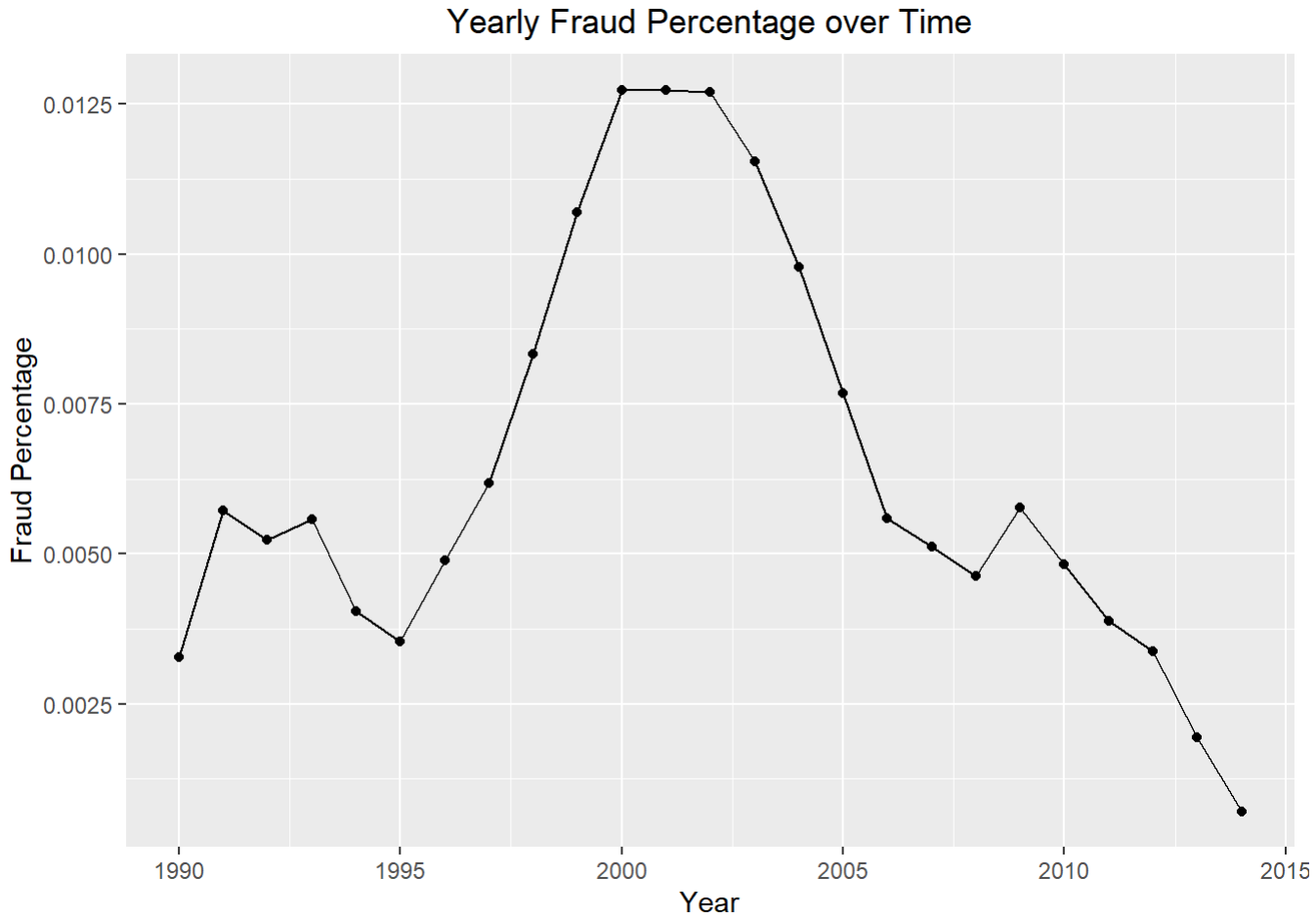
Preliminary Analysis

Load in data first. (Because it takes a lot of time to train all of my models, I won't re-train my models here; I will just load in all my saved models and results)

```
library(tidyverse)
load("C:/Users/cjy2001/OneDrive - Middlebury College/Middlebury Academic/Fall 2021/Project/Final_Data.RData")
```

Doing some preliminary analysis

```
home2 <- data %>%
  select(fyear, misstate) %>%
  group_by(fyear) %>%
  mutate(percent = sum(misstate == 1)/n())

home2 %>%
  ggplot(aes(x = fyear, y = percent)) +
  geom_line () +
  geom_point() +
  labs(x = "Year", y = "Fraud Percentage") +
  ggtitle("Yearly Fraud Percentage over Time") +
  theme(plot.title = element_text(hjust = 0.5))
```




```
# Passage of the Sarbanes-Oxley Act on July 30, 2002 (Enron Company)
```

Create Data Splitting Function

Then I need to create a function to split the data into different training and testing sets. As the paper mentions, it is a temporal data. Yao et.al require the training period to exceed 10 years to ensure the reliability of model training. In addition, they require a gap of 24 months between the financial results announcement of the last training year and the results announcement of a test year. For example, the training period is 1991–2001 for test year 2003 and 1991–2003 for test year 2005.

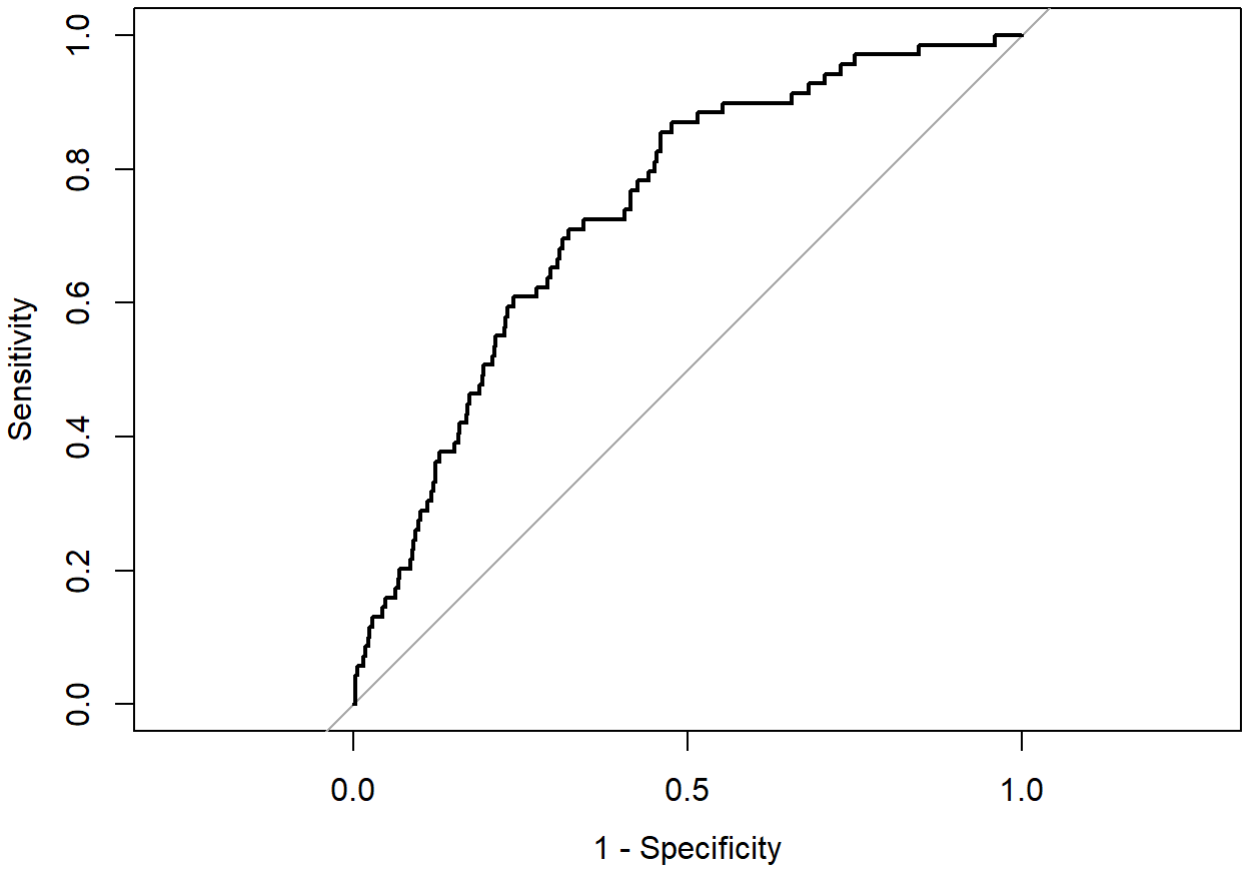
```
split_train_test <- function (year, data) {
  train <<- data %>%
    filter(fyear <= year - 2)

  test <<- data %>%
    filter(fyear == year)
}
```

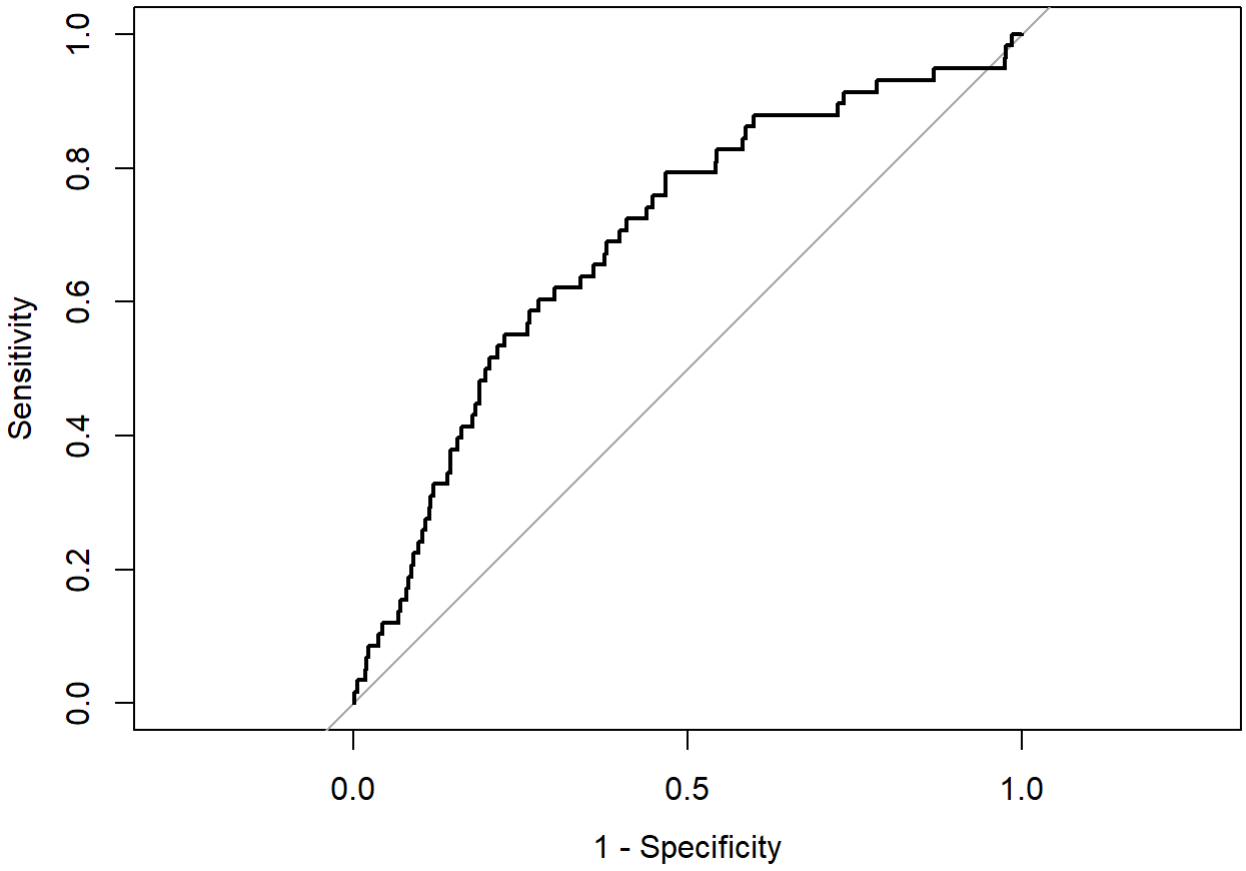
Logistic Regression

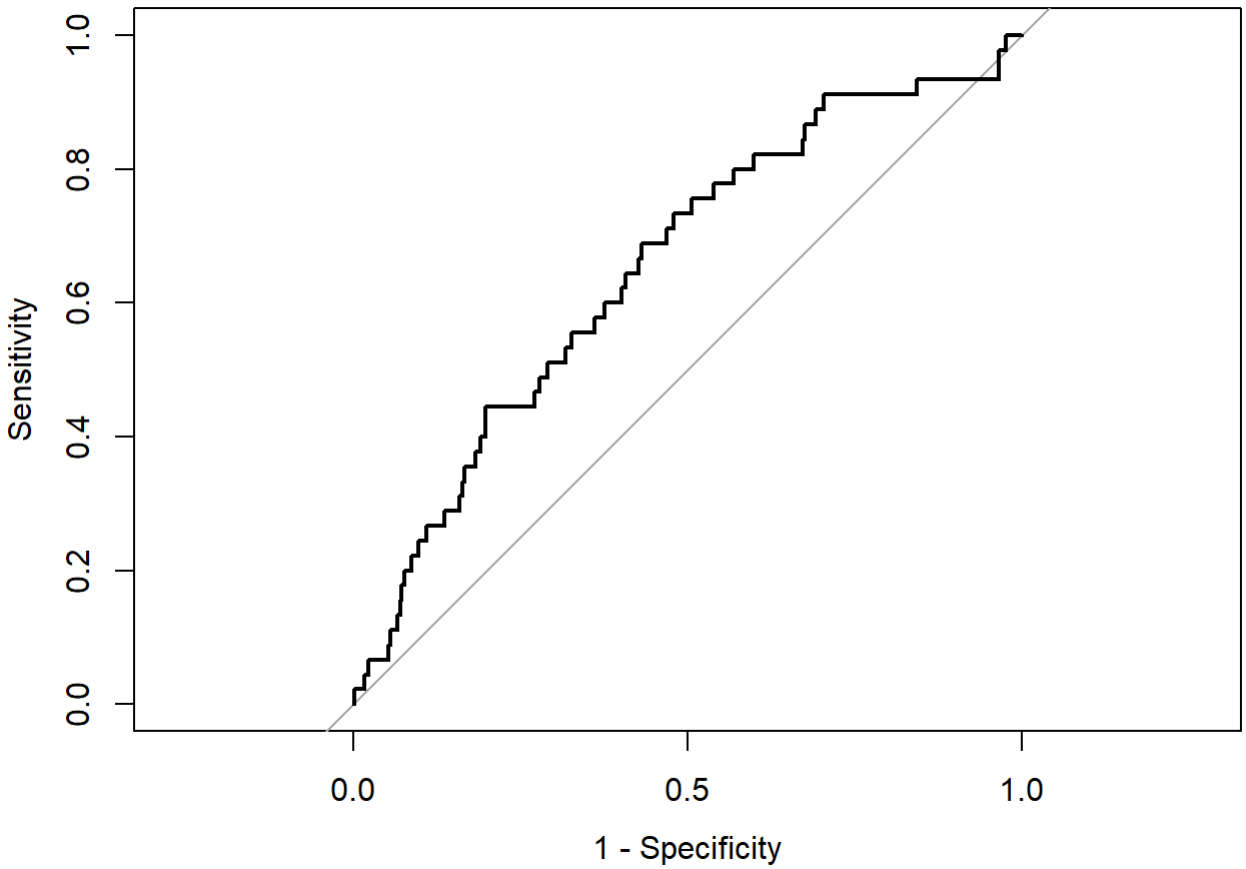
Now, I can do some logistic evaluations on my data across different years. I am using AUC score, along with the RUC curve, to evaluate my models.

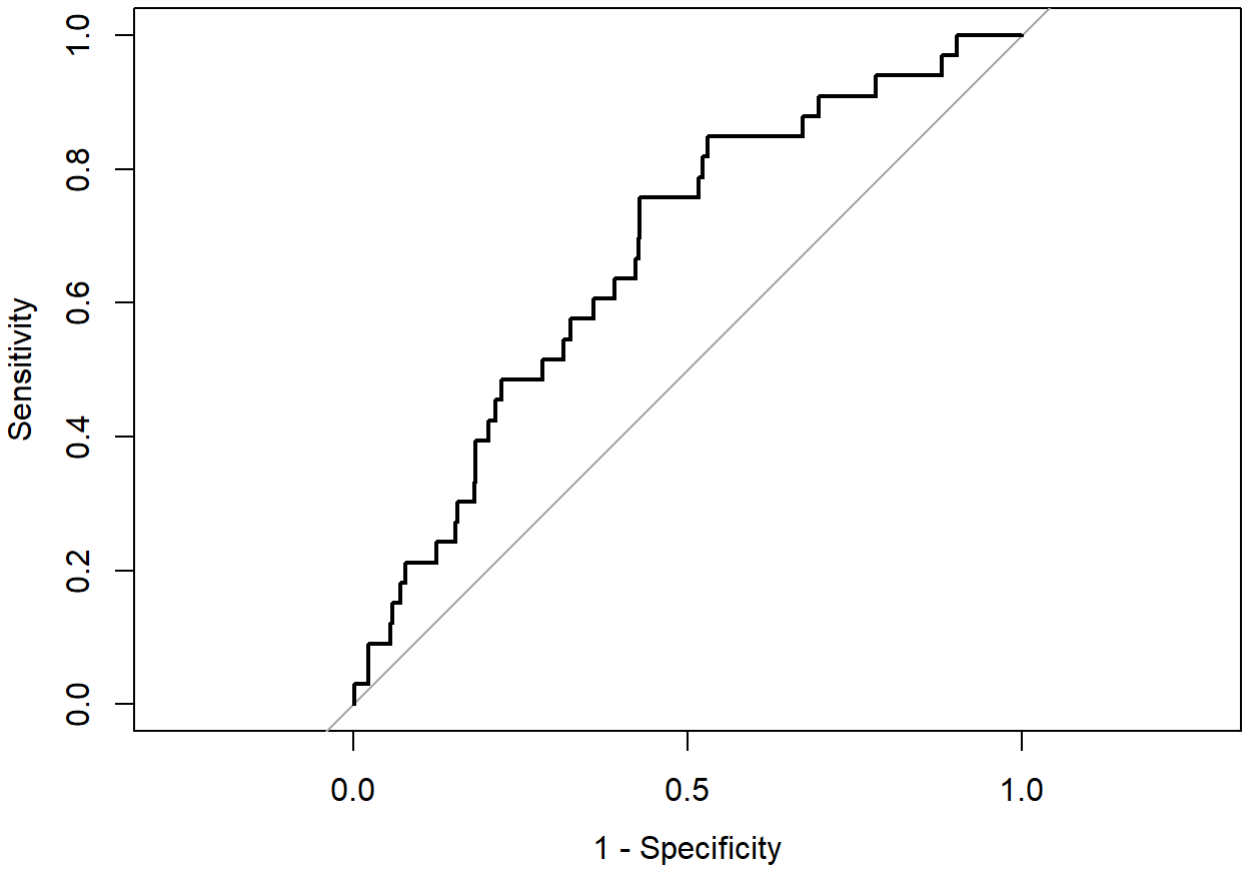
$$log(p/(1-p)) = intercept + slope1 * exp1 + slope2 * exp2 + ... + slopeP * expP$$

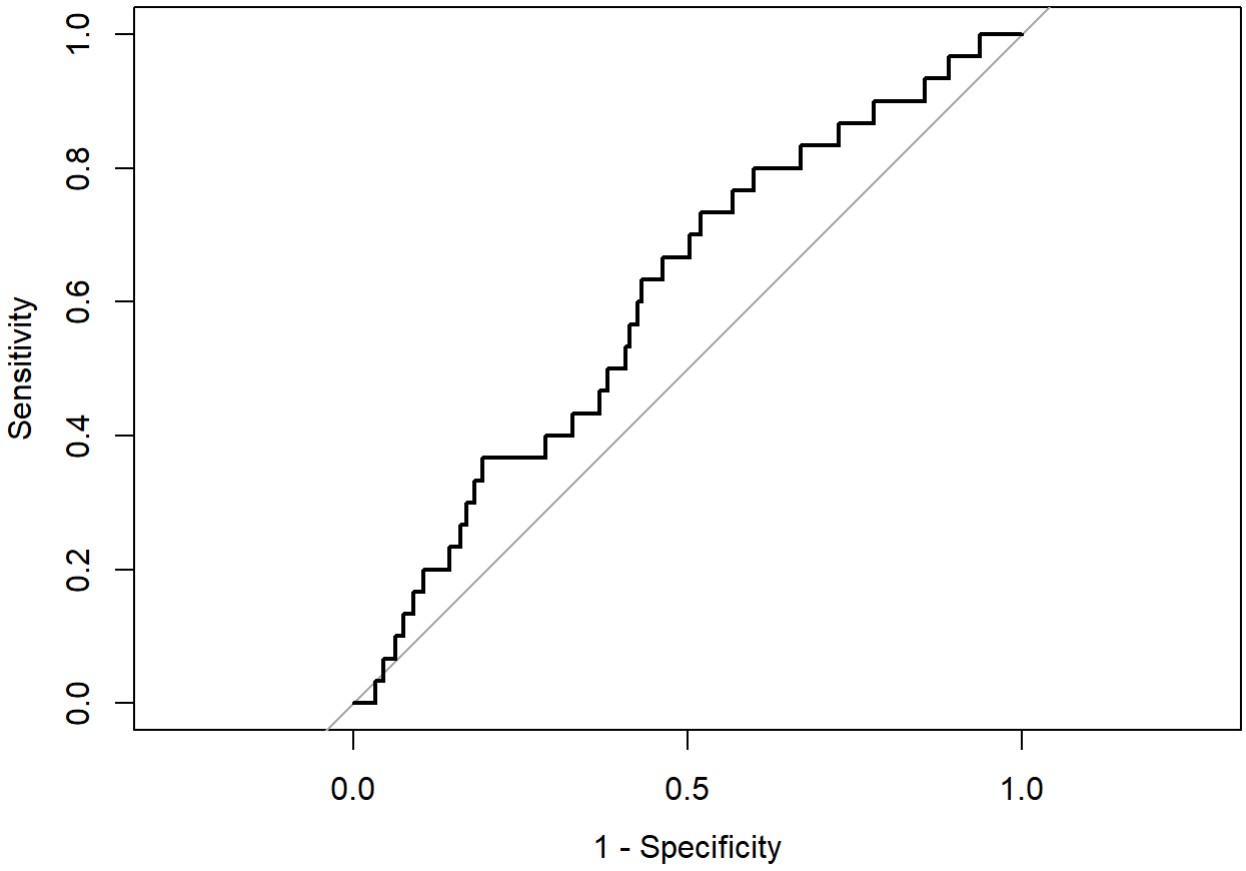
```
library(broom)
library(ISLR)
library(pROC)

result <- c(0)
i = 1

for (yr in 2003:2008){
  split_train_test(yr, data)

  default_model <- glm(formula = misstate ~ act + ap + at + ceq + che + cogs + csho +   dlc + dl
tis + dltt + ib + invt + ivao + ivst + lct + lt + ppegt + pstk + rect + sale + sstk + txp + prcc
_f, data = train, family = "binomial")

  pred_prob <- predict(default_model, newdata = test, type = "response") # predicted probabiliti
es
  pred_class_labels <- ifelse(pred_prob > 0.5, "Yes", "No") # convert to labels

roc_obj <- roc(predictor = pred_prob,
               response = test$misstate)

plot(roc_obj, legacy.axes = TRUE)

result[i] = auc(roc_obj)

i <- i + 1
}
```

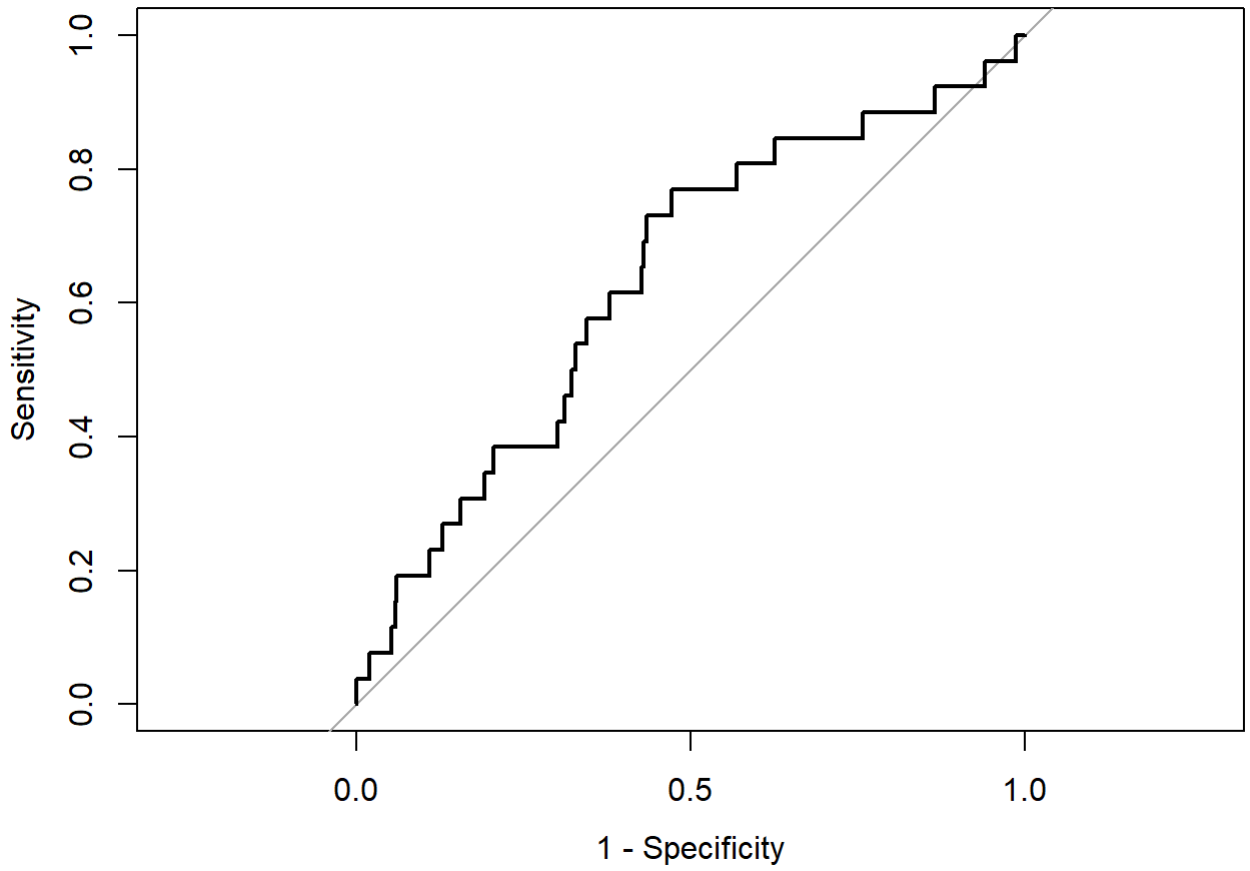

Sensitivity
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.5
1.0
1 - Specificity


Sensitivity
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.5
1.0
1 - Specificity


Sensitivity
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.5
1.0
1 - Specificity


Sensitivity
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.5
1.0
1 - Specificity


Sensitivity
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.5
1.0
1 - Specificity




```
result
```

```
## [1] 0.7381131 0.6946785 0.6517436 0.6723817 0.6061151 0.6354554
```

NDCG@k (mailto:NDCG@k)

In order to better assess a model's performance, the authors introduce another metric named "NDCG@k (mailto:NDCG@k)", which focuses more on if a model can rank its predicted probabilities correctly compared to the actual data. Here is a breif summary about its mechanism and how do my functions work.

NDCG: This function calculates the Normalized Discounted Cumulative Gain (NDCG), which is a measure of ranking quality. This metric takes position significance into account, and NDCG is computed as the ratio between Discounted Cumulative Gain(DCG) and idealized Discounted Cumulative Gain(iDCG). This function also requires the user to enter the value of k, which represents the number of top observations that should be used for calculation.

NDCG2: Similar to NDCG, this function calculates the Normalized Discounted Cumulative Gain. The difference between NDCG and NDCG2 is that the latter is more concise with the use of a new function named DCG (the formula for calculating the DCG score).

Both of them have 3 inputs: k: the number of observations that you want to include in computing NDCG score. Those observations generally have the highest predicted probabilities (or highest rankings) Relevance: an ideal list of relevance score of the document Predicted: a list of relevance score ranked by predicted probability

Output: NDCG@k (mailto:NDCG@k), which is (DCG@k (mailto:DCG@k))/(ideal DCG@k (mailto:DCG@k))

```
NDCG <- function (k, Relevance, Predicted) {
# Calculate DCG@K
  DCGk <- 0
  for (i in 1:k){
    # Assign each observation's relevance value
    rel <- Predicted[i]
    # Sum up DCG values
    DCGk = DCGk +(2^(rel) - 1)/(log2(i + 1))
  }

# Calculate ideal DCG@K
  iDCGk <- 0
  for (j in 1:k){
    # Similar to the previous for loop, we want to assign each observation's relevance value, ex
cept in this case all relevance scores are already ranked from highest to lowest (because it's i
deal)
    rel <- Relevance[j]
    iDCGk = iDCGk +(2^(rel) - 1)/(log2(j + 1))
  }

  # Print out DCG@k and iDCG@k values
  cat("DCG of the predicted order is", DCGk, "\nDCG of the ideal order(iDCG) is", iDCGk, "\n")
  # Compute NDCG@k
  return(DCGk/iDCGk)
}
```

```
# First, define a function DCG which can calculate the DCG score
DCG <- function (k, ranking){
  DCG <- 0
  for (i in 1:k){
  DCG = DCG +(2^(ranking[i]) - 1)/(log2(i + 1))
  }
  return(DCG)
}

# Then implement the DCG function into the main NDCG function
NDCG2 <- function (k, Relevance, Predicted) {
# Calculate DCG@K
  DCGk <- DCG(k, Predicted)

# Calculate ideal DCG@K
  iDCGk <- DCG(k, Relevance)

  cat("DCG of the predicted order is", DCGk, "\nDCG of the ideal order(iDCG) is", iDCGk, "\n")
  return(DCGk/iDCGk)
}
```

To test whether my function works, I manually calculate the score and compare the result with the result produced by my function, and they match each other.

```
test_predicted <- c(3, 5, 1, 3, 1)
test_ideal <- c(5, 3, 3, 1, 1)

# Compute each value manually (k = 4)
test_DCG <- (2^3 - 1)/log2(1 + 1) + (2^5 - 1)/log2(1 + 2) + (2^1 - 1)/log2(1 + 3) + (2^3 - 1)/lo
g2(1 + 4)
test_iDCG <- (2^5 - 1)/log2(1 + 1) + (2^3 - 1)/log2(1 + 2) + (2^3 - 1)/log2(1 + 3) + (2^1 - 1)/l
og2(1 + 4)
cat(test_DCG, test_iDCG, sep = "\n")
```

```
## 30.07356
## 39.34718
```

```
# Compute each value by using function
NDCG(4, test_ideal, test_predicted)
```

```
## DCG of the predicted order is 30.07356
## DCG of the ideal order(iDCG) is 39.34718
```

```
## [1] 0.7643128
```

```
NDCG2(4, test_ideal, test_predicted)
```

```
## DCG of the predicted order is 30.07356
## DCG of the ideal order(iDCG) is 39.34718
```

```
## [1] 0.7643128
```

```
# They match each other!
```

I also used another build-in function named “Evaluation.NDCG” from the StatRank package to assess my function’s correctness.

```
library(StatRank)
NDCG(length(PredictedRank), RelevanceLevel, PredictedRank)
Evaluation.NDCG(order(testpred$pred), PredictedRank)
```

Finally, I can apply my functions to the Fraud’s data by choosing year 2008 as test year to produce the corresponding NDCG@k (mailto:NDCG@k) score. Since I only have the logistic regression model, I can only evaluate its performance.

```
split_train_test(2008, data)

# Adding prediction to a test set
default_model <- glm(formula = misstate ~ act + ap + at + ceq + che + cogs + csho +   dlc + dlti
s + dltt + ib + invt + ivao + ivst + lct + lt + ppegt + pstk + rect + sale + sstk + txp + prcc_
f, data = train, family = "binomial")

# In this case, k = the number of top 1% firms in a test year
k <- floor(0.01*nrow(test))

test <- test %>%
  mutate(misstate = as.numeric(as.character(misstate)))

# Ideal order
RelevanceLevel <- sort(test$misstate, decreasing = TRUE)

# Recommendations order
testpred <- test %>%
    mutate(pred = pred_prob <- predict(default_model, newdata = test, type = "response")) %>%
    arrange(desc(pred))
PredictedRank <- testpred$misstate

NDCG(k, RelevanceLevel, PredictedRank)
```

```
## DCG of the predicted order is 0.3868528
## DCG of the ideal order(iDCG) is 8.342075
```

```
## [1] 0.04637369
```

```
NDCG2(k, RelevanceLevel, PredictedRank)
```

```
## DCG of the predicted order is 0.3868528
## DCG of the ideal order(iDCG) is 8.342075
```

```
## [1] 0.04637369
```

```
# The logit method's NDCG@k value from the paper is 0.028
```

RUSBoost

Introduction of Random Under-Sampling Boosting (RUSBoost)

-Why do we need RUSBoost?

It is common to have largely skewed training data. In our case, the fraud firms are unevenly represented, and model constructed with the goal of detecting a can achieve a correct classification rate of 99% by classifying all firms as being non-fraud. However, that model is meaningless to us as our goal is to identify fraudulent firms as much as possible.

-How does RUSBoost work?

It combines both Data Resampling and Boosting (ensemble learning) techniques. As for the Data Reseampling, RUSBoost implements Undersampling technique, which means removing examples from the majority class. Meanwhile, RUSBoost uses Adaptive Boosting (Adaboosting). The Adaboosting mechanism works as following: as initial learners are weak, subsequent ones can be tweaked in favor for those wrongly identified observations; and then it combines the result of multiple “weak learners” based on weight.

Therefore, RUSBoost reconciles undersampling’s problem by using boosting to offset the loss of information. Comparing to its brother SMOTEBoost, or as being a variant of SMOTEBoost, RUSBoost is more cost effective and less time consuming, as it provides a simple and efficient method for improving classification performance when training data is imbalanced.

RUSBoost Model Training

First, I create several training and testing data.

```
split_train_test(2003, data)
train_3 <- train
test_3 <- test
split_train_test(2004, data)
train_4 <- train
test_4 <- test
split_train_test(2005, data)
train_5 <- train
test_5 <- test
split_train_test(2006, data)
train_6 <- train
test_6 <- test
split_train_test(2007, data)
train_7 <- train
test_7 <- test
split_train_test(2008, data)
train_8 <- train
test_8 <- test
```

Next, I am going to adjust the tuning parameters to optimize our model. According to the paper, the model makes the best prediction when minsplit is set to 5 and sampleFration is set to 0.5. I will adhere to those values, but I will change the number of iterations (number of trees) to test my model’s performance. Therefore, in order to determine which number of iterations should I use, I use year 2003 as the test year and run several models to compare their prediction. The values I have tried for the number of iterations are 100, 200, 500, 1500, 3000, and 5000. I find that results from 3000 and 5000 iterations are the most reasonable (by looking at the number of true positives and AUC scores). Hence, I select 3000 as the final winner as it is relatively more computationally efficient comparing to 5000.

```
# These are all for year 2003.
set.seed(5)

time_5000 <- system.time(model_5000 <- rusb(formula = misstate ~ act + ap + at + ceq + che + cog
s + csho + dlc + dltis + dltt + dp
              + ib + invt + ivao + ivst + lct + lt + ni + ppegt + pstk + + re + rect + sale + ss
tk + txp + txt + xint +
                prcc_f,
              data = train_3,
              boot = FALSE,
              iters = 5000,
              control = rpart.control(minsplit = 5),
              sampleFraction = 0.5,
              idx = train_3$misstate == 0)
)

time_3000 <- system.time(model_3000 <- rusb(formula = misstate ~ act + ap + at + ceq + che + cog
s + csho + dlc + dltis + dltt + dp
              + ib + invt + ivao + ivst + lct + lt + ni + ppegt + pstk + + re + rect + sale + ss
tk + txp + txt + xint +
                prcc_f,
              data = train,
              boot = FALSE,
              iters = 3000,
              control = rpart.control(minsplit = 5),
              sampleFraction = 0.5,
              idx = train$misstate == 0)
)

time_1500 <- system.time(model_1500 <- rusb(formula = misstate ~ act + ap + at + ceq + che + cog
s + csho + dlc + dltis + dltt + dp
              + ib + invt + ivao + ivst + lct + lt + ni + ppegt + pstk + + re + rect + sale + ss
tk + txp + txt + xint +
                prcc_f,
              data = train,
              boot = FALSE,
              iters = 1500,
              control = rpart.control(minsplit = 5),
              sampleFraction = 0.5,
              idx = train$misstate == 0)
)

time_500 <- system.time(model_500 <- rusb(formula = misstate ~ act + ap + at + ceq + che + cogs
 + csho + dlc + dltis + dltt + dp
              + ib + invt + ivao + ivst + lct + lt + ni + ppegt + pstk + + re + rect + sale + ss
tk + txp + txt + xint +
                prcc_f,
              data = train,
              boot = FALSE,
              iters = 500,
              control = rpart.control(minsplit = 5),
              sampleFraction = 0.5,
```

```
              idx = train$misstate == 0)
)

time_200 <- system.time(model_200 <- rusb(formula = misstate ~ act + ap + at + ceq + che + cogs
 + csho + dlc + dltis + dltt + dp
              + ib + invt + ivao + ivst + lct + lt + ni + ppegt + pstk + + re + rect + sale + ss
tk + txp + txt + xint +
                prcc_f,
              data = train,
              boot = FALSE,
              iters = 200,
              control = rpart.control(minsplit = 5),
              sampleFraction = 0.5,
              idx = train$misstate == 0)
)
```

After obtaining models, I predict individual results.

```
preds_200 <- predict.rusb(model_200, data.frame(test_3))
preds_500 <- predict.rusb(model_500, data.frame(test_3))
preds_1500 <- predict.rusb(model_1500, data.frame(test_3))
preds_3000 <- predict.rusb(model_3000, data.frame(test_3))
year3_preds_5000 <- predict.rusb(year3_model_5000, data.frame(test_3))
```

Calculate their AUC, Sensitivity, and Precision.

```
auc <- NULL
x <- list(preds_200, preds_500, preds_1500, preds_3000, year3_preds_5000)

for (i in 1:5){
roc_obj <- roc(predictor = x[[i]]$prob[,2],
               response = test_3$misstate)

#plot(roc_obj, legacy.axes = TRUE)
auc[i] = auc(roc_obj)
}

# Sensitivity TP/P = TP/(TP + FN)
# Precision TP/TP+FP
sensitivity <- NULL
precision <- NULL

for (i in 1:5) {
  print(x[[i]]$confusion)
  sensitivity[i] <- x[[i]]$confusion[2,2]/sum(x[[i]]$confusion[,2])
  precision[i] <- x[[i]]$confusion[2,2]/sum(x[[i]]$confusion[2,])
}
```

```
##                Observed Class
## Predicted Class    0    1
##               0 5911   66
##               1    1    3
##                Observed Class
## Predicted Class    0    1
##               0 5907   69
##               1    5    0
##                Observed Class
## Predicted Class    0    1
##               0 5856   61
##               1   56    8
##                Observed Class
## Predicted Class    0    1
##               0 5660   54
##               1  252   15
##                Observed Class
## Predicted Class    0    1
##               0 4374   20
##               1 1538   49
```

```
auc
```

```
## [1] 0.7798656 0.7593889 0.7784474 0.7755731 0.7756884
```

```
sensitivity
```

```
## [1] 0.04347826 0.00000000 0.11594203 0.21739130 0.71014493
```

```
precision
```

```
## [1] 0.75000000 0.00000000 0.12500000 0.05617978 0.03087587
```

Therefore, I choose the number of iterations to be 3000 and apply the same tuning parameters to the rest test years.

```
set.seed(5)
#.Random.seed  Check seed is the same
year4_time_3000 <- system.time(year4_model_3000 <- rusb(formula = misstate ~ act + ap + at + ceq
+ che + cogs + csho + dlc + dltis + dltt + dp
              + ib + invt + ivao + ivst + lct + lt + ni + ppegt + pstk + + re + rect + sale + ss
tk + txp + txt + xint +
                prcc_f,
              data = train_4,
              boot = FALSE,
              iters = 3000,
              control = rpart.control(minsplit = 5),
              sampleFraction = 0.5,
              idx = train_4$misstate == 0)
)

year5_time_3000 <- system.time(year5_model_3000 <- rusb(formula = misstate ~ act + ap + at + ceq
+ che + cogs + csho + dlc + dltis + dltt + dp
              + ib + invt + ivao + ivst + lct + lt + ni + ppegt + pstk + + re + rect + sale + ss
tk + txp + txt + xint +
                prcc_f,
              data = train_5,
              boot = FALSE,
              iters = 3000,
              control = rpart.control(minsplit = 5),
              sampleFraction = 0.5,
              idx = train_5$misstate == 0)
)

year6_time_3000 <- system.time(year6_model_3000 <- rusb(formula = misstate ~ act + ap + at + ceq
+ che + cogs + csho + dlc + dltis + dltt + dp
              + ib + invt + ivao + ivst + lct + lt + ni + ppegt + pstk + + re + rect + sale + ss
tk + txp + txt + xint +
                prcc_f,
              data = train_6,
              boot = FALSE,
              iters = 3000,
              control = rpart.control(minsplit = 5),
              sampleFraction = 0.5,
              idx = train_6$misstate == 0)
)

year7_time_3000 <- system.time(year7_model_3000 <- rusb(formula = misstate ~ act + ap + at + ceq
+ che + cogs + csho + dlc + dltis + dltt + dp
              + ib + invt + ivao + ivst + lct + lt + ni + ppegt + pstk + + re + rect + sale + ss
tk + txp + txt + xint +
                prcc_f,
              data = train_7,
              boot = FALSE,
              iters = 3000,
              control = rpart.control(minsplit = 5),
              sampleFraction = 0.5,
              idx = train_7$misstate == 0)
```

```
)

year8_time_3000 <- system.time(year8_model_3000 <- rusb(formula = misstate ~ act + ap + at + ceq
+ che + cogs + csho + dlc + dltis + dltt + dp
              + ib + invt + ivao + ivst + lct + lt + ni + ppegt + pstk + + re + rect + sale + ss
tk + txp + txt + xint +
                prcc_f,
              data = train_8,
              boot = FALSE,
              iters = 3000,
              control = rpart.control(minsplit = 5),
              sampleFraction = 0.5,
              idx = train_8$misstate == 0)
)
```

Result

Since I use the system time to time each model, I can also put the time used by each model here.

model with 100 trees: 13 mins
model with 200 trees: 30 mins
model with 500 trees: 1 hour
model with 1500 trees: 2 hours
model with 3000 trees: 5 hours
model with 5000 trees: 8 hours

year3: 5 hours
year4: 4 hours
year5: 4 hours
year6: 45 mins???
year7: 8 hours???
year8: 8 hours???

It is worth mentioning that we still don't know what causes that large variation in time used by models trained by different years. For example, the time used by year 2006 only takes 45 minutes; year 2007 and 2008 both take a much longer time, but their performances are the worst. Though the number of observations increases over years, it still cannot explain why we first see a stable trend from year 2003 to 2005 and then it changes so abruptly (because the increase of observations is similar between years.)

Finally, I record the performance of each model. I can then recreate some tables from the paper such as the NDCG@k (mailto:NDCG@k), AUC, sensitivity, and precision scores.

```
yearly_test_set <- list(test_3,test_4,test_5,test_6, test_7, test_8)
yearly_preds <- list(year3_preds_3000,year4_preds_3000,year5_preds_3000,year6_preds_3000,year7_p
reds_3000,year8_preds_3000)

yearly_sensitivity <- NULL
yearly_precision <- NULL
yearly_auc <- NULL
yearly_NDCG <- NULL

for (i in 1:6) {
  print(yearly_preds[[i]]$confusion)

  roc_obj <- roc(predictor = yearly_preds[[i]]$prob[,2],
                 response = yearly_test_set[[i]]$misstate)
  yearly_auc[i] = auc(roc_obj)
  plot(roc_obj, legacy.axes = TRUE)

  yearly_sensitivity[i] <-
    yearly_preds[[i]]$confusion[2,2]/sum(yearly_preds[[i]]$confusion[,2])
  yearly_precision[i] <- yearly_preds[[i]]$confusion[2,2]/sum(yearly_preds[[i]]$confusion[2,])

  k <- floor(0.01*nrow(yearly_test_set[[i]]))
  RelevanceLevel <- as.numeric(as.character(sort(yearly_test_set[[i]]$misstate, decreasing = TRU
E)))
  testpred <- cbind(yearly_test_set[[i]], data.frame(yearly_preds[[i]]$prob[,1]))
  names(testpred)[52] <- "preds"
  testpred <- testpred %>% arrange(preds)
  PredictedRank <- as.numeric(as.character(testpred$misstate))
  yearly_NDCG[i] <- NDCG2(k, Relevance = RelevanceLevel, Predicted = PredictedRank)
}
```

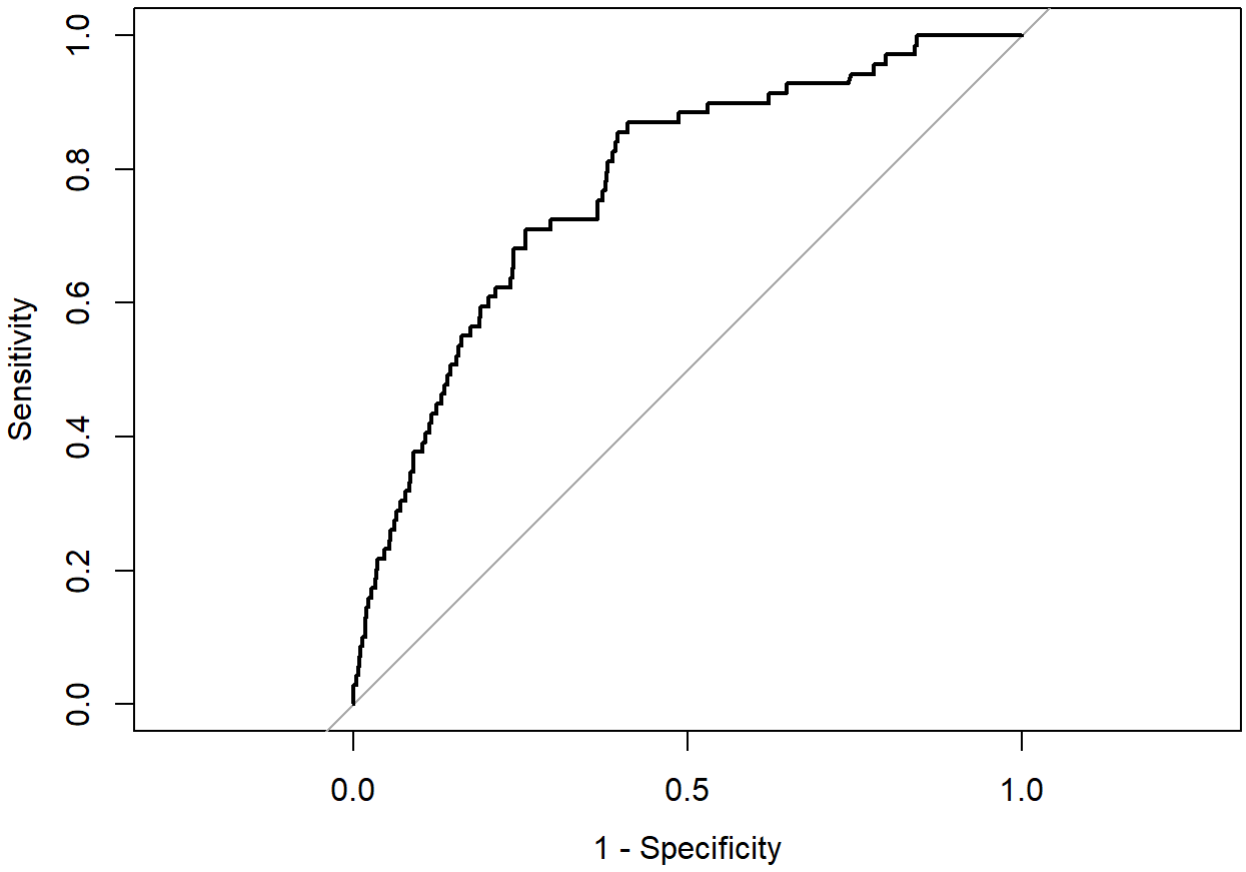
```
##                Observed Class
## Predicted Class    0    1
##               0 5660   54
##               1  252   15
```

```
## Setting levels: control = 0, case = 1
```

```
## Setting direction: controls < cases
```

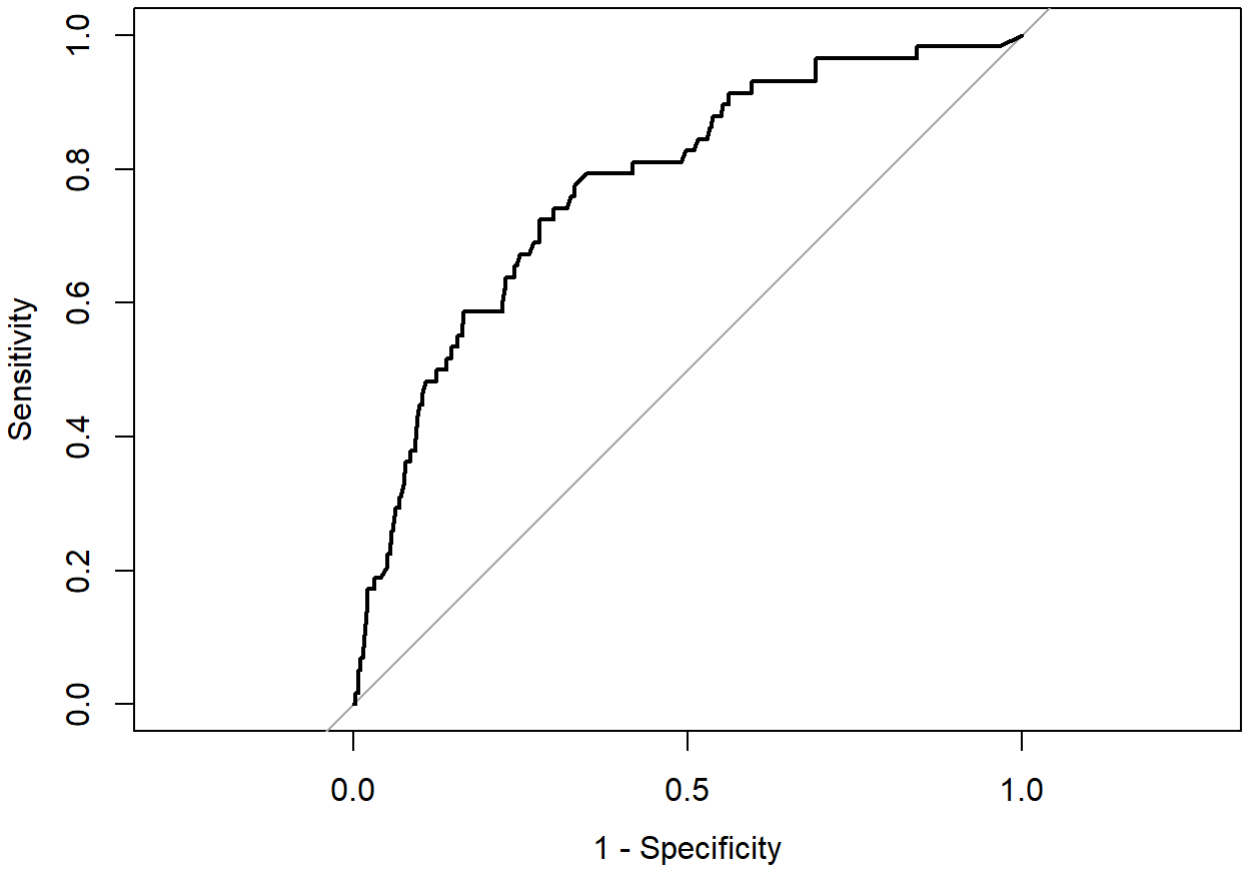
```
## DCG of the predicted order is 1.876148
## DCG of the ideal order(iDCG) is 14.44811
##                Observed Class
## Predicted Class    0    1
##               0 5670   47
##               1  206   11
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```



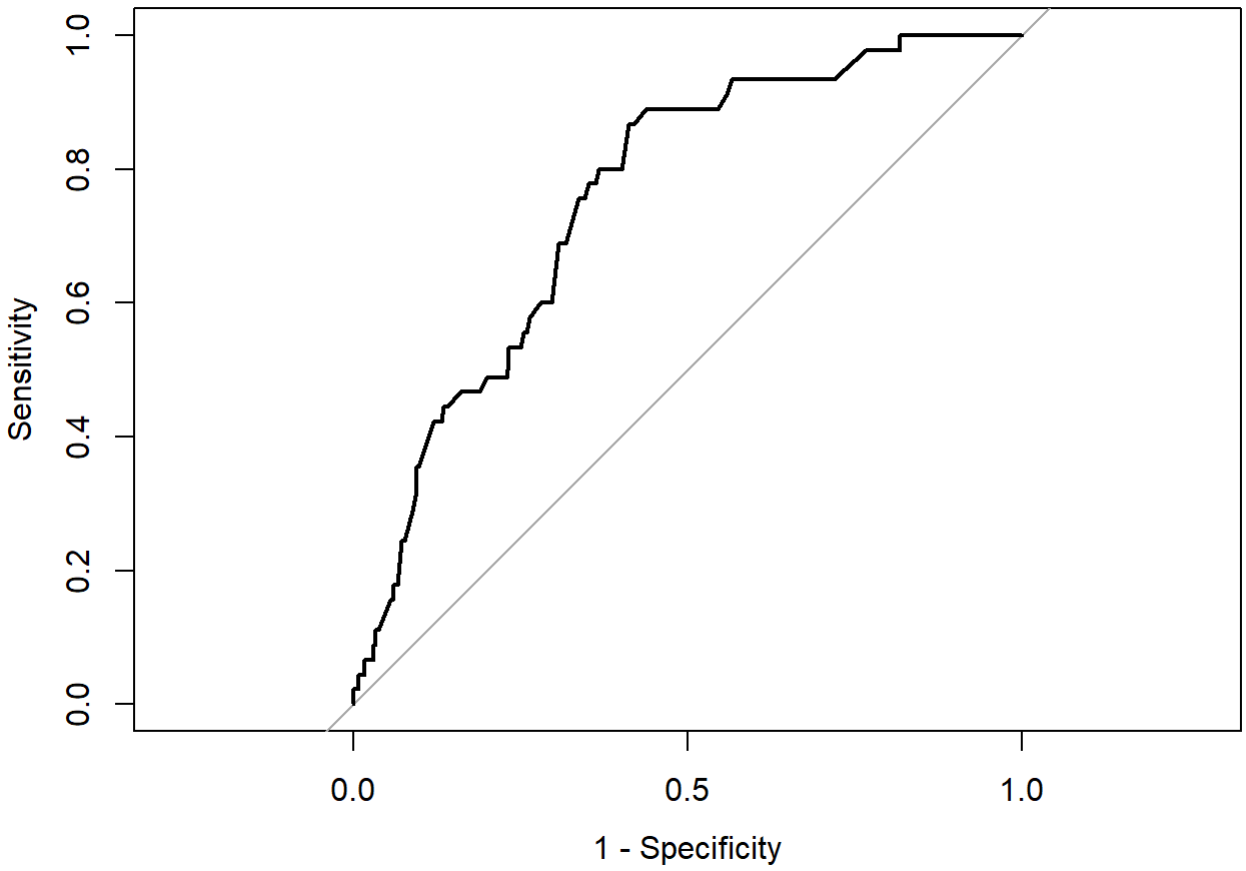

```
## DCG of the predicted order is 0.5681128
## DCG of the ideal order(iDCG) is 14.27881
##                Observed Class
## Predicted Class    0    1
##               0 5702   42
##               1  116    3
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```



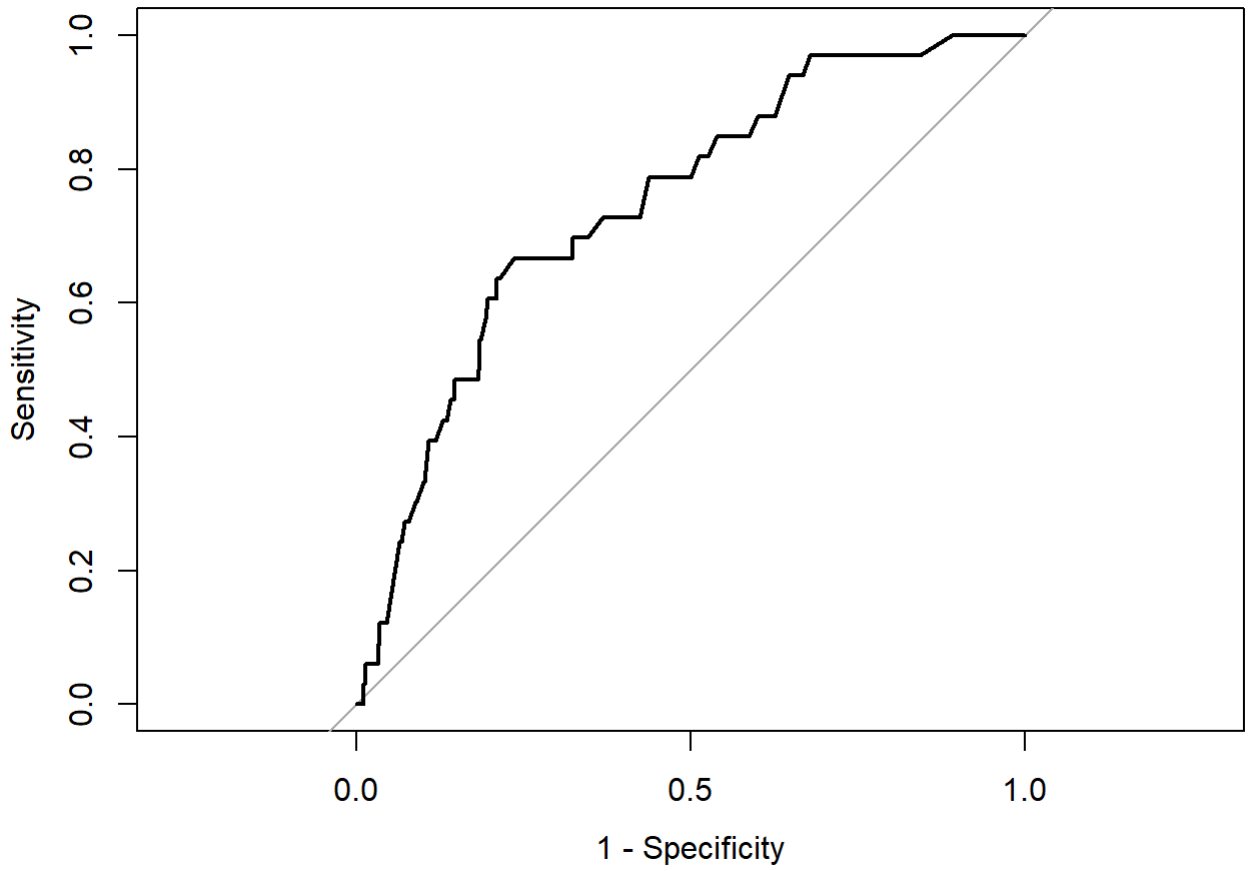

```
## DCG of the predicted order is 0.5649564
## DCG of the ideal order(iDCG) is 12.00707
##                Observed Class
## Predicted Class    0    1
##               0 4119   11
##               1 1756   22
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```



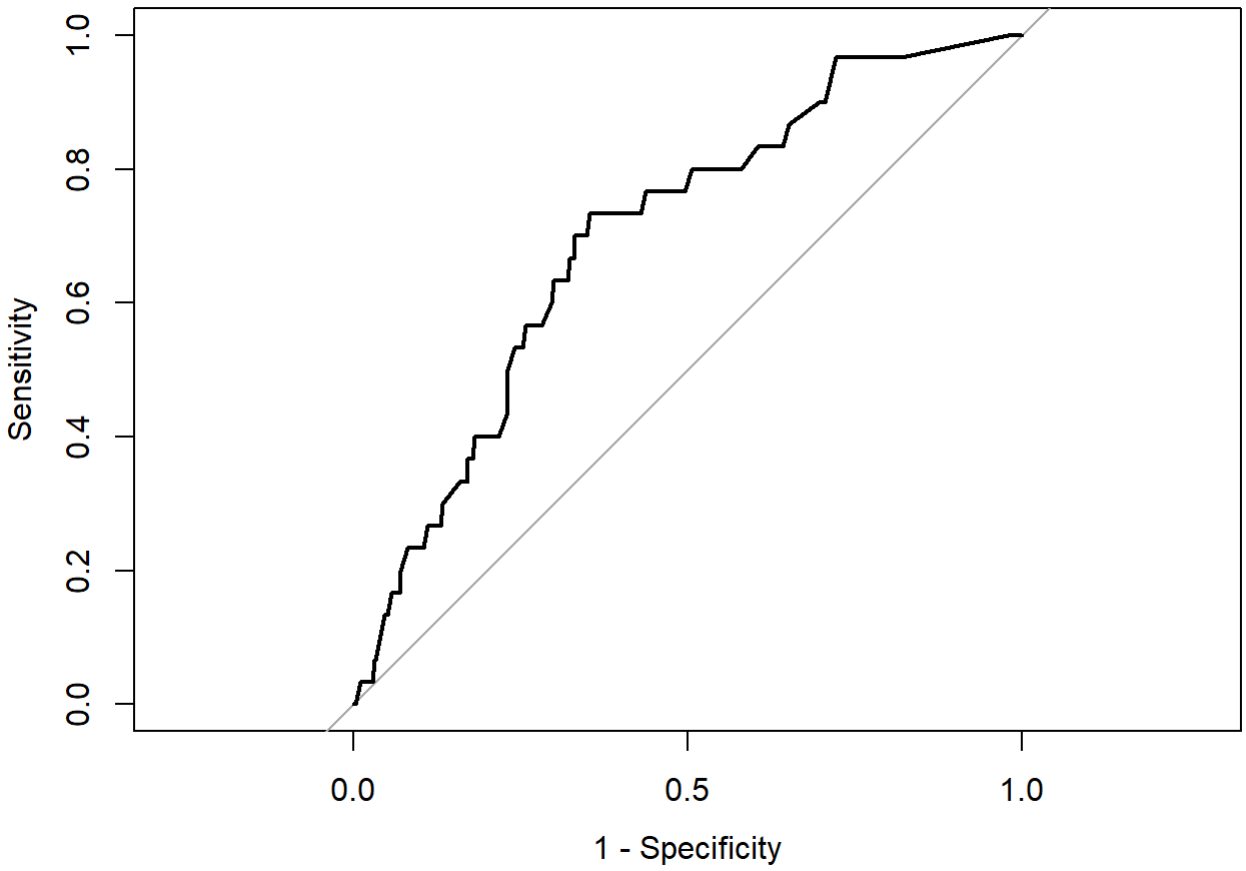

```
## DCG of the predicted order is 0
## DCG of the ideal order(iDCG) is 9.756383
##                Observed Class
## Predicted Class    0    1
##               0 5436   25
##               1  402    5
```

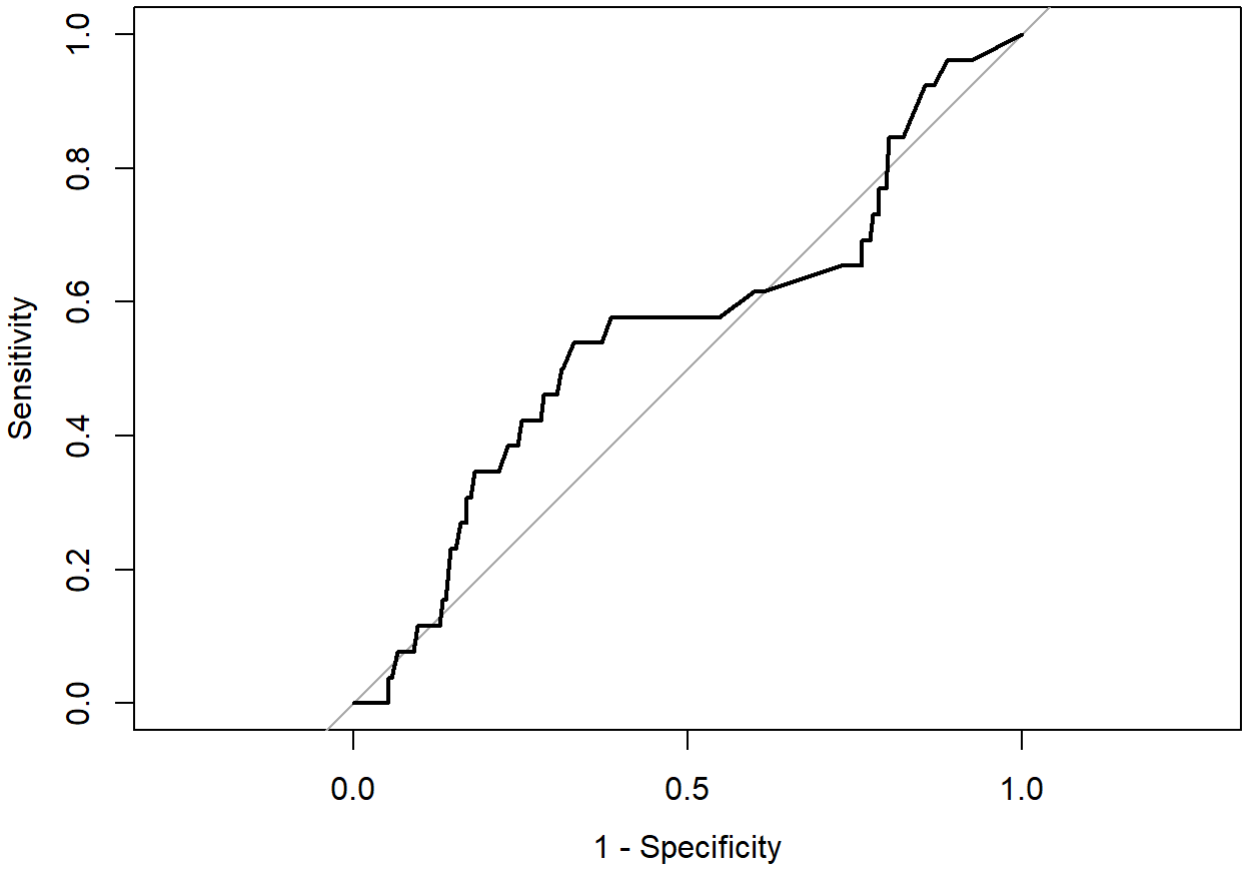
```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```




```
## DCG of the predicted order is 0.1800313
## DCG of the ideal order(iDCG) is 9.161581
##                Observed Class
## Predicted Class    0    1
##               0 5471   26
##               1  115    0
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```


Sensitivity
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.5
1.0
1 - Specificity


Sensitivity
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.5
1.0
1 - Specificity

```
## DCG of the predicted order is 0
## DCG of the ideal order(iDCG) is 8.342075
```

```
mean(yearly_auc)
```

```
## [1] 0.7183315
```

```
mean(yearly_precision)
```

```
## [1] 0.02612326
```

```
mean(yearly_sensitivity)
```

```
## [1] 0.2178411
```

```
mean(yearly_NDCG)
```

```
## [1] 0.03939067
```

Referring to the paper：“using the same 28 raw financial data items, the performance metrics averaged over the test period 2003–2008 is”:

AUC = 0.725
NDCG = 0.049
Sensitivity = 4.88%
Precision = 4.48%

Here are some intuitive findings:

1. Our models’ averaged value of AUC is similar to paper’s, except the poor performance of year 2008.
2. Except for year 2003, 2004, and 2005, our models generally don’t perform well on the NDCG@k (mailto:NDCG@k) metric.
3. Our models have a much higher averaged sensitivity. Though it is very different from the paper, a higher sensitivity indicates that our models are doing a better job.
4. Our models have a slightly lower score of precision.

The general trend is that the overall accuracy rate declines over the years, but it still cannot explain why the 2008 model is so unusual, which requires further investigation.

Discuss Weird ROC Curve for 2008

Here I want to study why the prediction for year8(2008)’s model behaves so strangely.

At first, I compute all the ROC curves from year 2003 to 2008. It seems like only year8’s curve looks so abnormal comparing to others: the ROC curve looks promising at first. However, there is a sharp turn when x approximately approaches 0.3, and the weird shape continues until x reaches 0.75, thereby reducing its AUC value largely. Therefore, I’d like to find those threshold values which cause that shape and try to find the reason behind them.

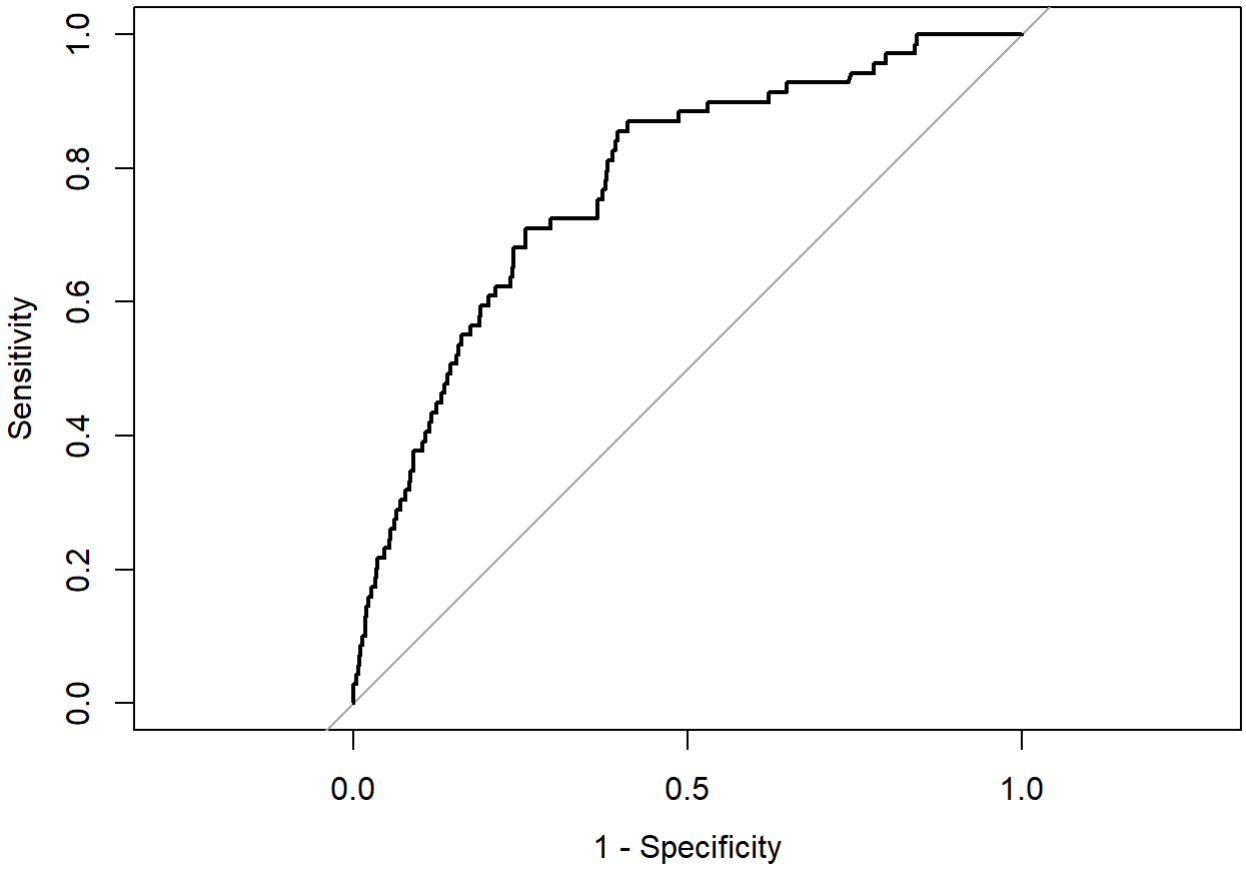
```
new_yearly_auc <- NULL

for (i in 1:6) {
  roc_obj <- roc(predictor = yearly_preds[[i]]$prob[,2],
                 response = yearly_test_set[[i]]$misstate)
  new_yearly_auc[i] = auc(roc_obj)
  plot(roc_obj, legacy.axes = TRUE)
}
```

```
## Setting levels: control = 0, case = 1
```

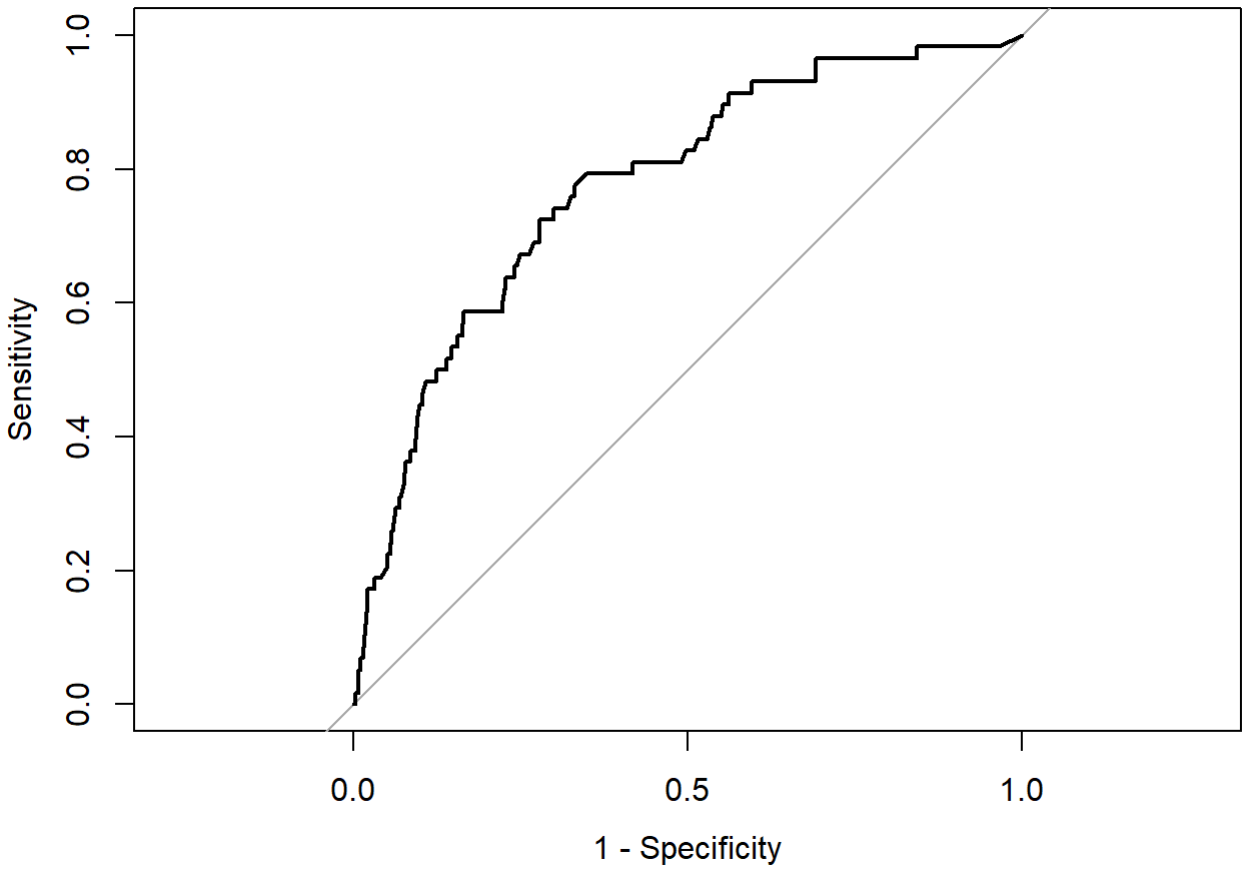
```
## Setting direction: controls < cases
```

```
## Setting levels: control = 0, case = 1
```

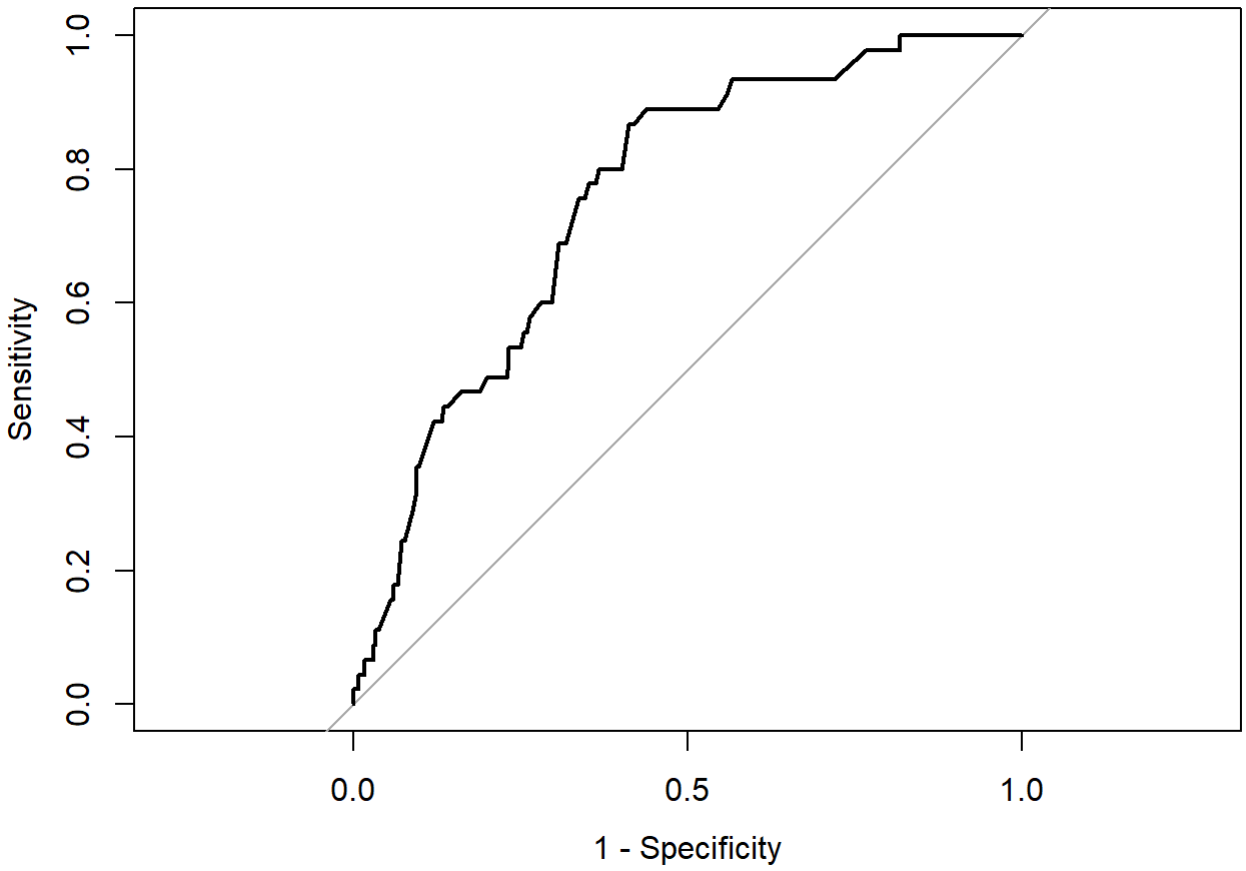
```
## Setting direction: controls < cases
```



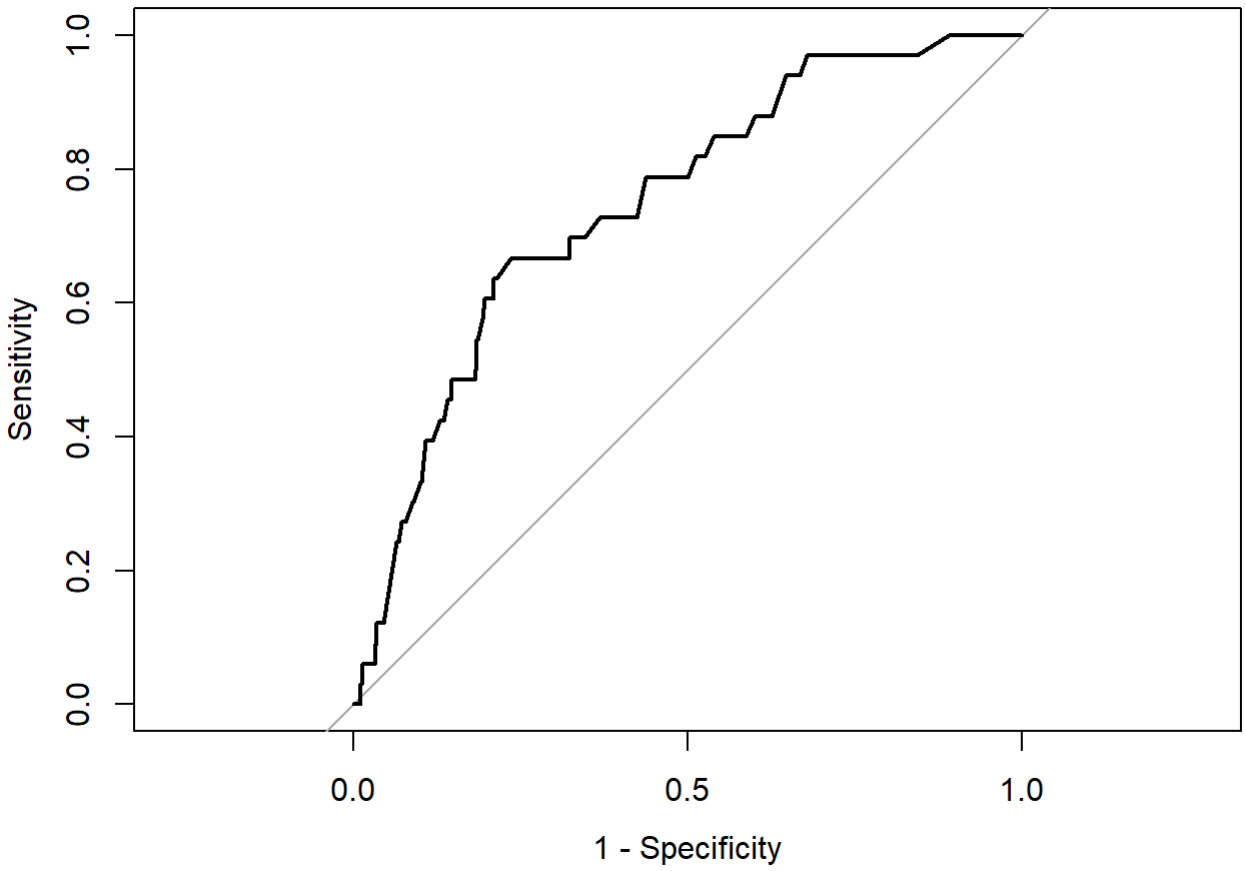

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```



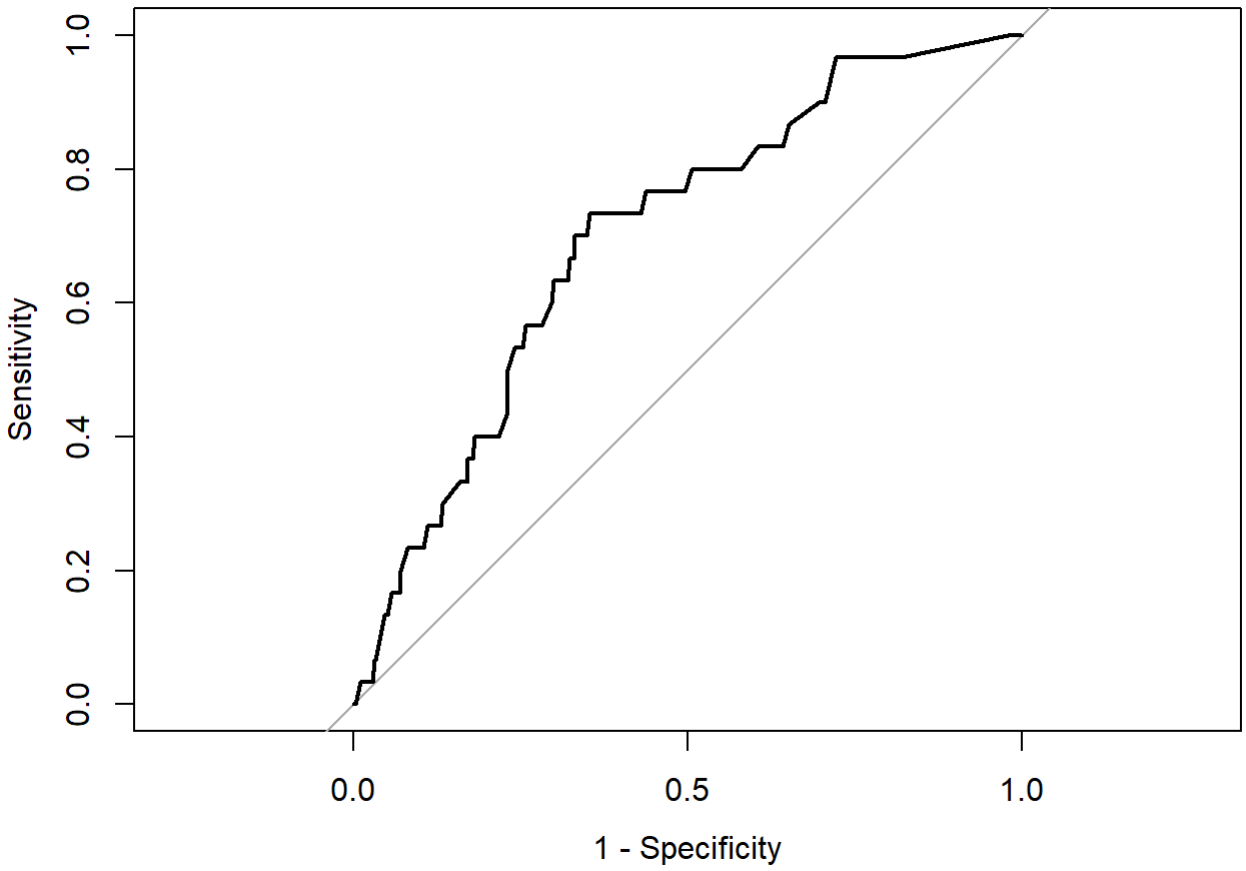

```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```

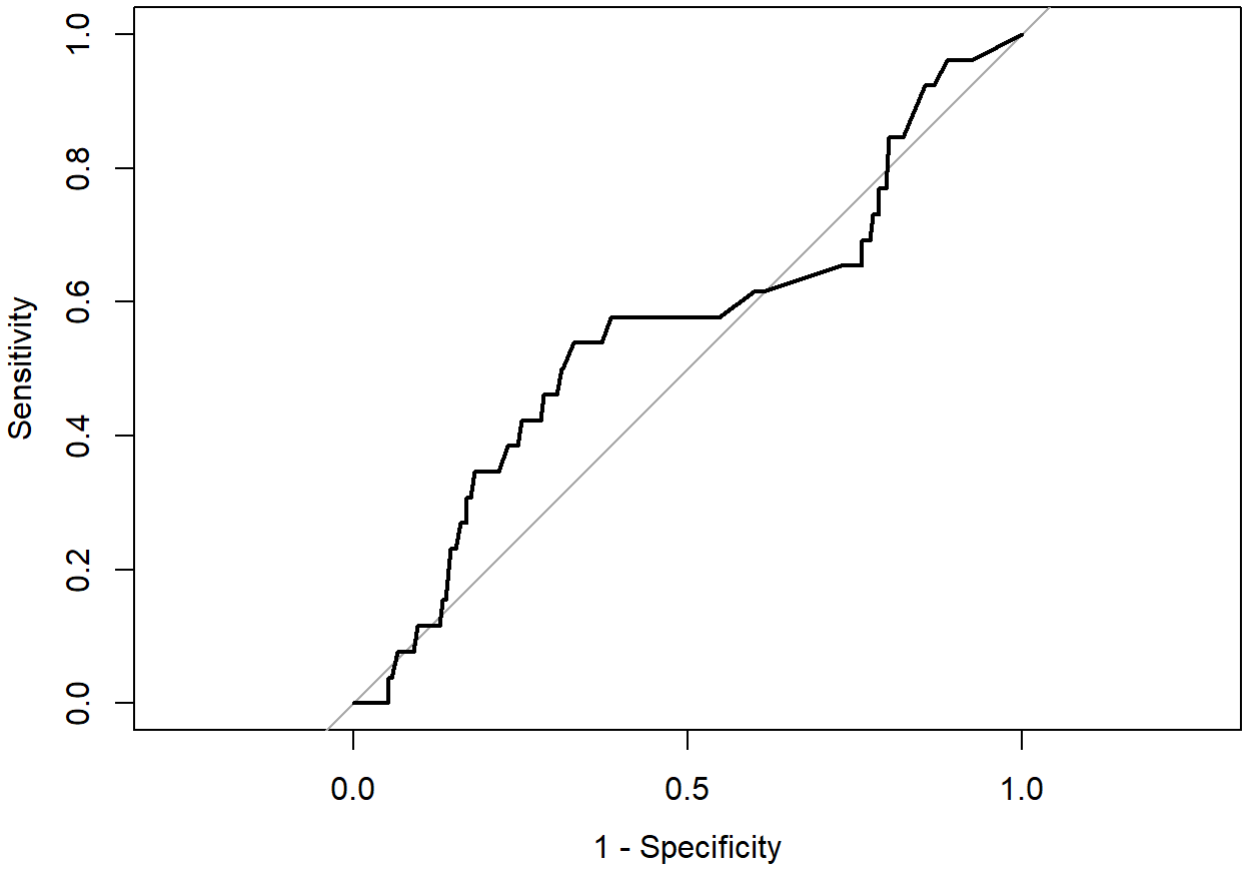



```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```




```
## Setting levels: control = 0, case = 1
## Setting direction: controls < cases
```


Sensitivity
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.5
1.0
1 - Specificity


Sensitivity
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.5
1.0
1 - Specificity

Next I am going to adjust threshold values and record each value's corresponding coordinates (1-specificity as x-axis and sensitivity as y-axis.) The loop will stop and report an error when it reaches the limit of predicted probability, which means based on that threshold and afterwards, every observation will be predicted as negative.

```
threshold <- seq(0.01, 1, by = 0.01) # Setting threshold vectors
new_sensitivity <- NULL
one_minus_specificity <- NULL

# Append observation's probability
new_test_8 <- cbind(test_8, year8_preds_3000$prob) %>%
  rename("Yes" = "2", "No" = "1")
```

```
# Change the code chunk's setting "error" to be true so R will not stop executing when there is
 an error

for (i in 1:length(threshold)) {
  print(threshold[i]) # Let user know progress of the for loop so it can indicate the threshold
 at which the for loop should stop
  new_test_8 <- new_test_8 %>%
    mutate(preds = case_when(new_test_8$Yes >= threshold[i] ~ "Yes",
                             TRUE ~ "No")) #Add predicted fraudulence based on a threshold value t
o the train data frame and relabel it (so I can use it to construct confusion matrix)

#Confusion Matrix
new_result <- table(new_test_8$misstate, new_test_8$preds)[2:1, 2:1]
# Since I reverse the matrix every time, it will report an error when there is only one single c
olumn in the table, which means at that threshold, all companies will be predicted to be non-fra
udulent. Therefore, I can use that feature along with the previous "print" function to know the
 specific stopping threshold.

# Sensitivity = TP/P
# Specificity = TN/N
new_sensitivity[i] <- new_result[1,1]/sum(new_result[1,])
one_minus_specificity[i] <- 1 - new_result[2,2]/sum(new_result[2,])
# Calculate the values on the x-axis and y-axis of the ROC curve so I can construct the curve ma
nually
}
```

```
## [1] 0.01
## [1] 0.02
## [1] 0.03
## [1] 0.04
## [1] 0.05
## [1] 0.06
## [1] 0.07
## [1] 0.08
## [1] 0.09
## [1] 0.1
## [1] 0.11
## [1] 0.12
## [1] 0.13
## [1] 0.14
## [1] 0.15
## [1] 0.16
## [1] 0.17
## [1] 0.18
## [1] 0.19
## [1] 0.2
## [1] 0.21
## [1] 0.22
## [1] 0.23
## [1] 0.24
## [1] 0.25
## [1] 0.26
## [1] 0.27
## [1] 0.28
## [1] 0.29
## [1] 0.3
## [1] 0.31
## [1] 0.32
## [1] 0.33
## [1] 0.34
## [1] 0.35
## [1] 0.36
## [1] 0.37
## [1] 0.38
## [1] 0.39
## [1] 0.4
## [1] 0.41
## [1] 0.42
## [1] 0.43
## [1] 0.44
## [1] 0.45
## [1] 0.46
## [1] 0.47
## [1] 0.48
## [1] 0.49
## [1] 0.5
## [1] 0.51
## [1] 0.52
```

```
## [1] 0.53
## [1] 0.54
## [1] 0.55
```

```
## Error in `[.default`(table(new_test_8$misstate, new_test_8$preds), 2:1, : subscript out of bo
unds
```

It seems like when threshold is set to 0.55, all observations will be predicted as non-fraudulent. Here I just want to test if that's right by creating two confusion matrices here, and it proves my finding.

```
new_test_8 <- new_test_8 %>%
  mutate(preds = case_when(new_test_8$Yes >= 0.54 ~ "Yes",
                           TRUE ~ "No"))
table(new_test_8$misstate, new_test_8$preds)
```

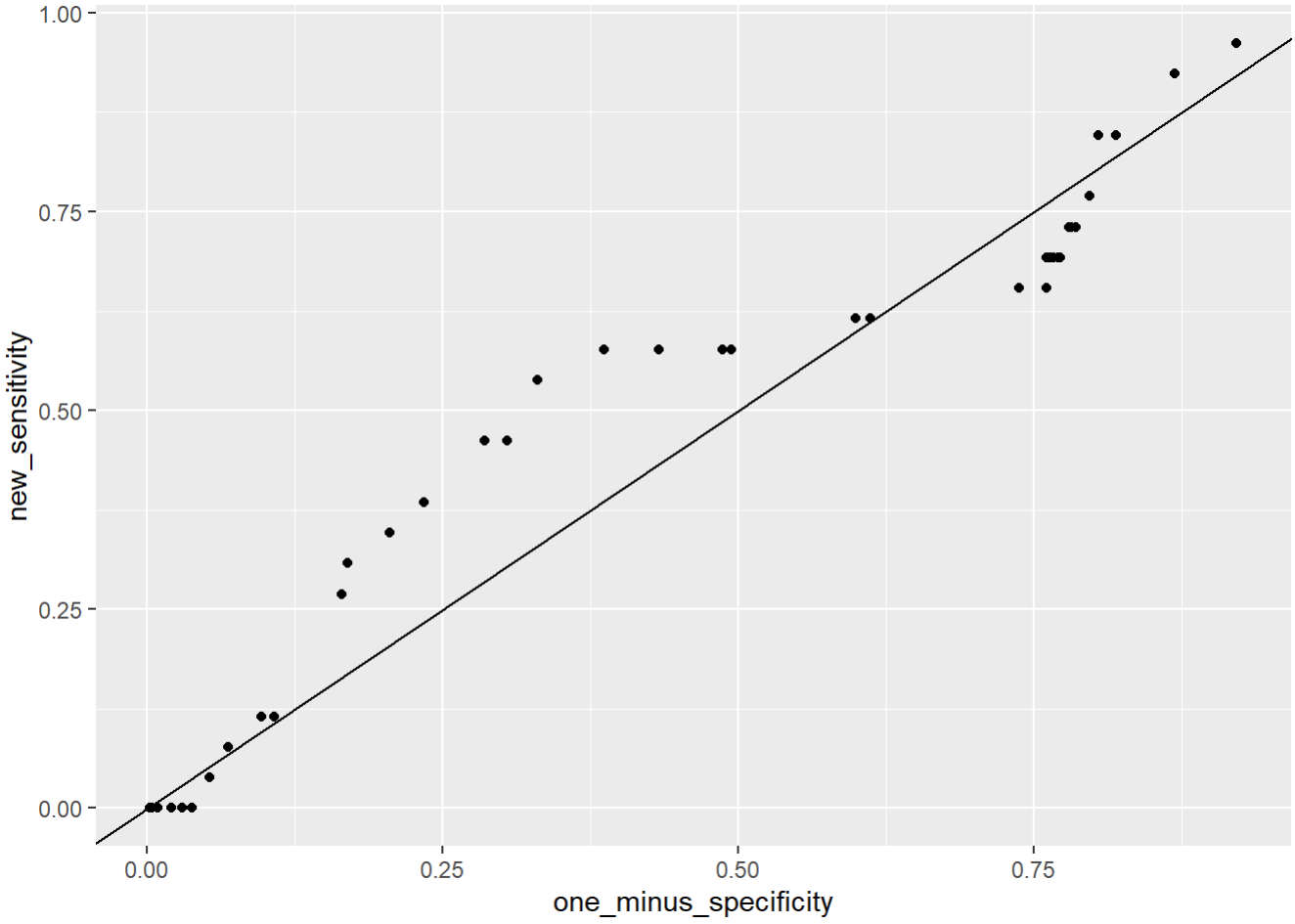
```
##
##       No  Yes
##   0 5574   12
##   1   26    0
```

```
new_test_8 <- new_test_8 %>%
  mutate(preds = case_when(new_test_8$Yes >= 0.55 ~ "Yes",
                           TRUE ~ "No"))

table(new_test_8$misstate, new_test_8$preds)
```

```
##
##       No
##   0 5586
##   1   26
```

Lastly, I plot all threshold values on the same graph based on their corresponding sensitivity and specificity values. The graph is the same as the one I get from the AUC function.

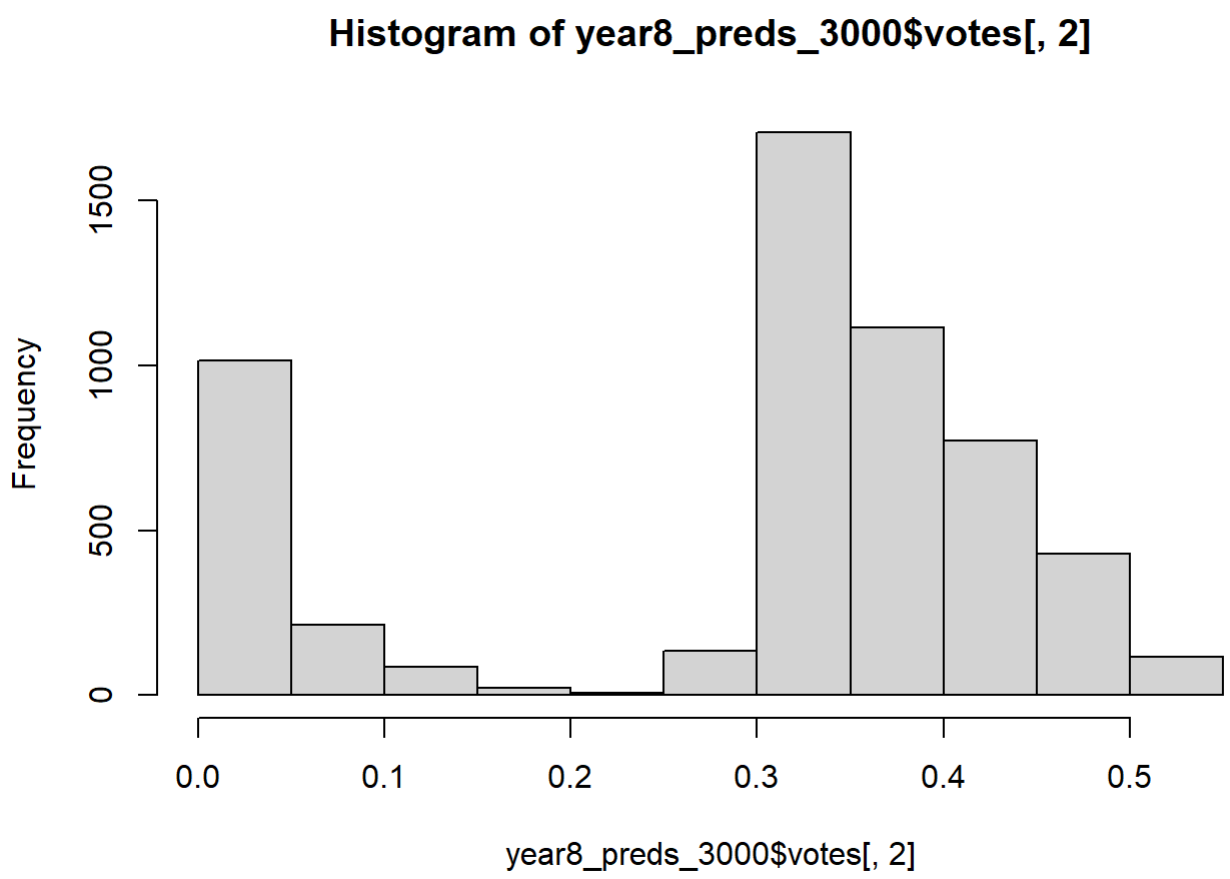
```
ggplot() +
  geom_point(aes(x = one_minus_specificity, y = new_sensitivity)) +
  geom_abline(intercept =0 , slope = 1)
```



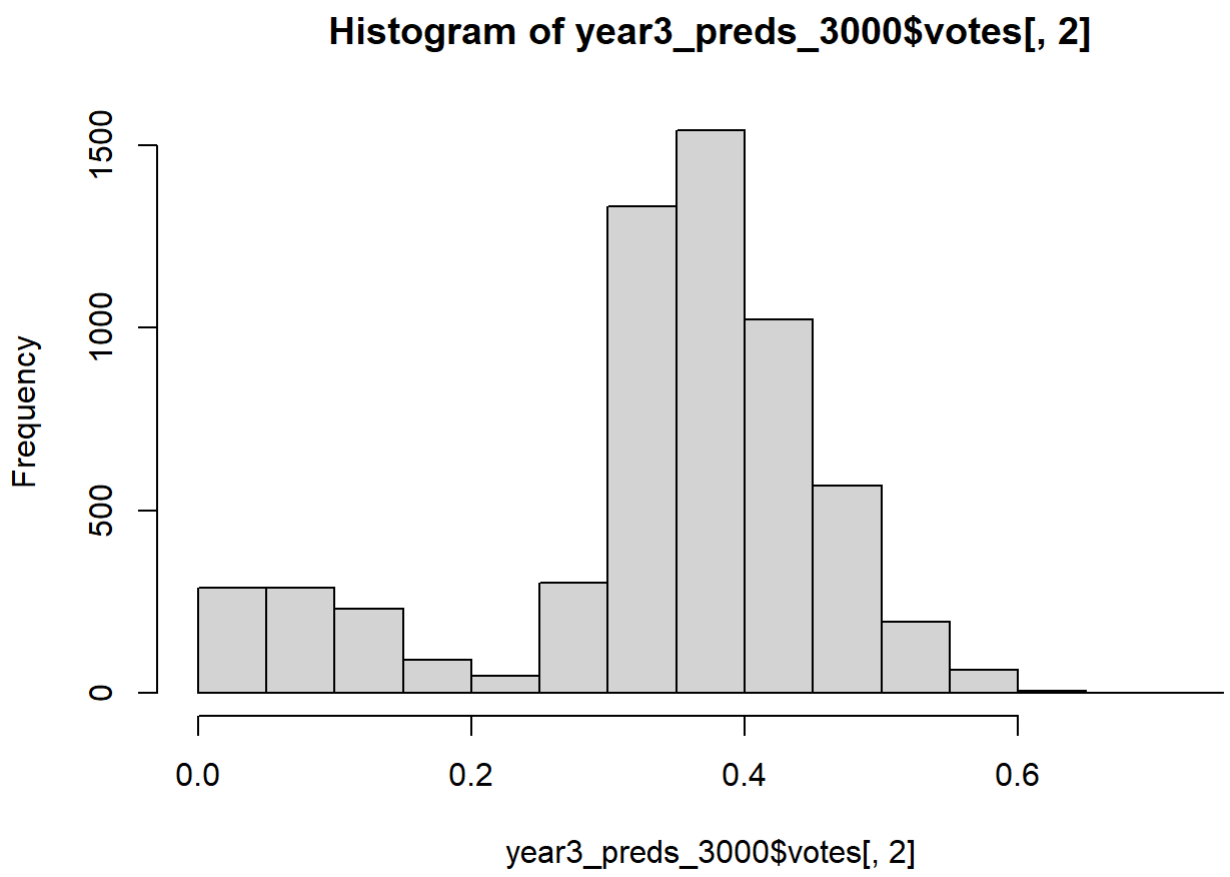

```
data.frame(threshold = threshold[1:54], one_minus_specificity, new_sensitivity) %>%
  filter(one_minus_specificity > 0.38 & one_minus_specificity < 0.875) %>%
  arrange(one_minus_specificity)
```

```
##    threshold one_minus_specificity new_sensitivity
## 1       0.36             0.3865020       0.5769231
## 2       0.35             0.4323308       0.5769231
## 3       0.34             0.4867526       0.5769231
## 4       0.33             0.4942714       0.5769231
## 5       0.32             0.5989975       0.6153846
## 6       0.31             0.6117078       0.6153846
## 7       0.30             0.7373792       0.6538462
## 8       0.29             0.7602936       0.6538462
## 9       0.28             0.7604726       0.6923077
## 10      0.27             0.7606516       0.6923077
## 11      0.26             0.7610097       0.6923077
## 12      0.24             0.7613677       0.6923077
## 13      0.25             0.7613677       0.6923077
## 14      0.18             0.7624418       0.6923077
## 15      0.19             0.7624418       0.6923077
## 16      0.20             0.7624418       0.6923077
## 17      0.21             0.7624418       0.6923077
## 18      0.22             0.7624418       0.6923077
## 19      0.23             0.7624418       0.6923077
## 20      0.16             0.7631579       0.6923077
## 21      0.17             0.7631579       0.6923077
## 22      0.14             0.7662012       0.6923077
## 23      0.15             0.7662012       0.6923077
## 24      0.13             0.7704977       0.6923077
## 25      0.12             0.7721088       0.6923077
## 26      0.11             0.7798067       0.7307692
## 27      0.10             0.7814178       0.7307692
## 28      0.09             0.7851772       0.7307692
## 29      0.08             0.7968135       0.7692308
## 30      0.07             0.8045113       0.8461538
## 31      0.05             0.8191908       0.8461538
## 32      0.06             0.8191908       0.8461538
## 33      0.03             0.8686001       0.9230769
## 34      0.04             0.8686001       0.9230769
```

```
hist(year8_preds_3000$votes[,2])
```




```
hist(year3_preds_3000$votes[,2])
```

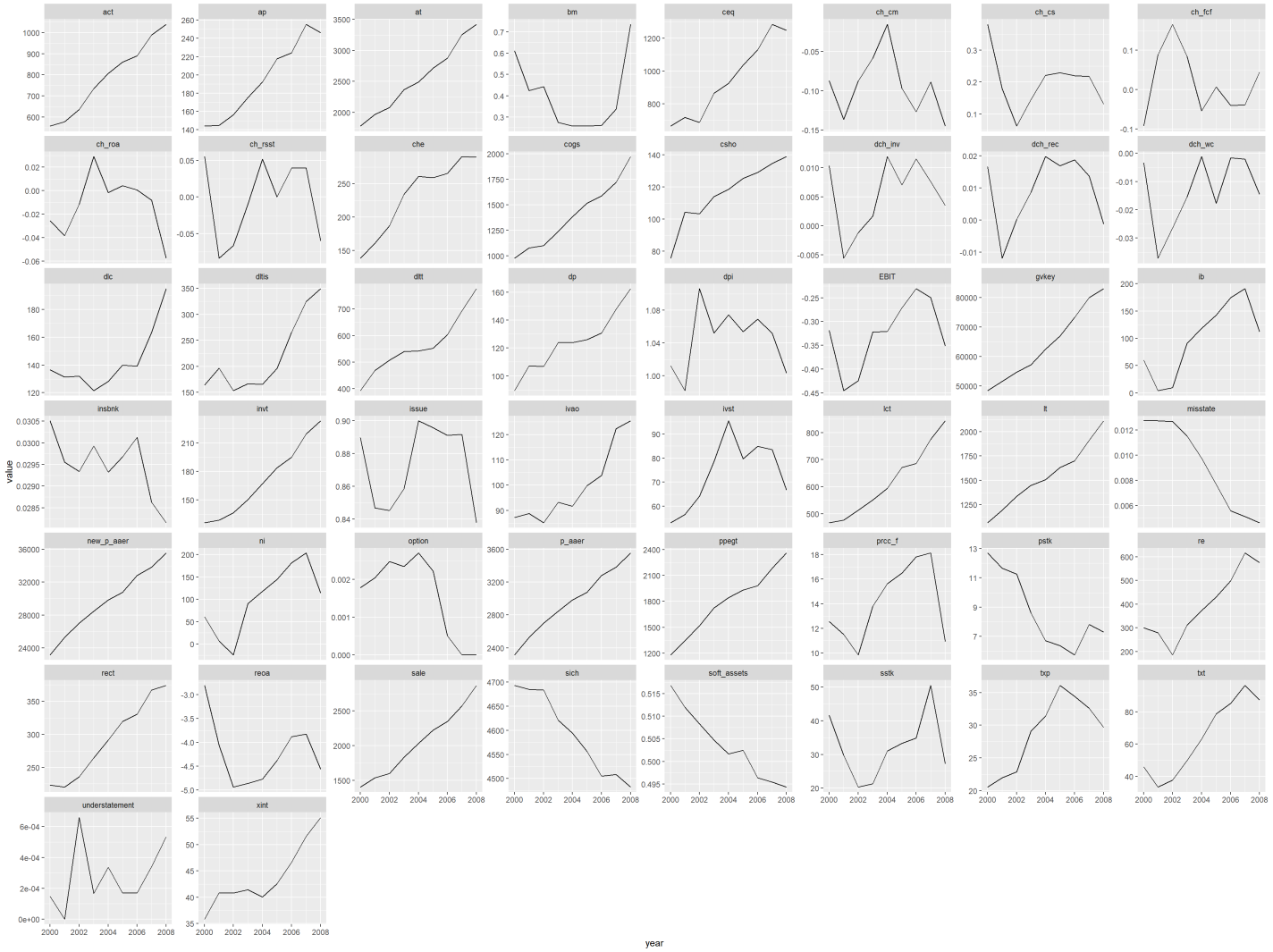


My Thoughts

So what makes this year's model not classification-threshold invariant? I think one possible explanation could be the predicted probability's distribution. According to the histogram, there are too many points distributed between 0.3 and 0.35, and that's where our ROC curve gets weird. As a comparison, the year3's (2003) distribution is more even. As the threshold value decreases from 0.36, sensitivity does not vary too much, implying that even though there are many observations distributed on that range, few of them are correctly predicted as fraudulent. In contrast, specificity is computed as TN/N, and it will keep changing because the model tends to predict more negatives. Therefore, as x value increases while y value stays the same, the ROC curve will become more flat, thereby having a lower AUC score.

What's more, I plot all variables' graphs to see if there appears any unusual lookings.

```
data2 <- data %>%
  filter(fyear < 2009 & fyear > 1999) %>%
  mutate(misstate = as.numeric(as.character(misstate)))

stats2 <- aggregate(data2[,2:51], list(data2$fyear), mean, na.rm=TRUE) %>%
  rename(year = Group.1)

stats2 %>%
  pivot_longer(cols = 2:51,
               names_to = "variable",
               values_to = "value") %>%
  ggplot() +
  geom_line(aes(x = year, y = value)) +
  facet_wrap(~variable,
             scales = "free_y") +
  scale_x_continuous(breaks = seq(2000,2008,2))
```



By examining my explanatory variables' trends in different years, I want to find if there are any sharp changes in year 2008. Possible candidates are bm, ch_roa, ch_rsst, issue, prcc_f, and sstk. All of them show different degrees of sharp decline in year 2008. Therefore, it is possible that those variables cause the unusual performance for year 2008's model.