STAT 5650 Final Project

Jonah Carlson

Due: 04/26/2025

Summary

Fraud is a financial crime that can be difficult and time consuming to detect. Machine learning techniques can aide the recognition of fraud through classification techniques. This paper tests some common classifiers on a fraud-related data set based on real-world data. We conclude that of the classifiers tested, a tuned Gradient Boosting Machine had the greatest accuracy, sensitivity, and specificity of the models.

Introduction

This project fits a variety of classifiers to data on the presence of bank account fraud. The data is synthetically generated based on real-world data using both Laplacian noise and a General Adversarial Network.[1] The data set was constructed by Sergio Jesus et. al at the the Universidade de Porto in Portugal, and was constructed for use at the Annual Conference on Neural Information Processing Systems (NeurIPS) 2022.[2]

The data set constructed is split into seven types: Base, and Variations I - VI. Each contains 1,000,000 observations with slightly different qualities. Base makes the least changes from the original data set, while the variations are designed for tackling various machine learning challenges.[3]

The data set publishers acknowledge the nature of fraud and its influence on testing classification techniques. Classifying fraud using machines demands a high level of accuracy to simultaneously catch individuals engaging in fraudulent behavior and protect the rights of individuals who are obeying the law.

The most important element of model accuracy will be sensitivity – catching individuals engaging in fraudulent behavior. Yet specificity is also crucial: in their paper discussing the construction and evaluation of this data set, they note that in some jurisdictions, such as the European Union, access to a bank account is considered a right.[4] As a result, denying somebody a bank account on grounds of fraud could cause legal issues if that classification of fraud was a false positive. As a result, both sensitivity and specificity must be valued – a model which sacrifices one to improve the other may not be as valuable as a model that classifies both well.

This paper fits 7 classifiers to the Base.csv data and compares them across overall accuracy, sensitivity, and specificity. The end of the paper examines variable importance and partial dependence plots to better gleam the most valuable variables for predicting fraud.

[1] Sérgio Jesus et al., "Turning the Tables: Biased, Imbalanced, Dynamic Tabular Datasets for ML Evaluation," arXiv.org, November 24, 2022, https://arxiv.org/abs/2211.13358, 5.

[2] Jesus et al., "Turning the Tables: Biased, Imbalanced, Dynamic Tabular Datasets for ML Evaluation." Additionally, "Bank Account Fraud Dataset Suite (NeurIPS 2022)," Kaggle, November 29, 2023, https://www.kaggle.com/datasets/sgpjesus/bank-account-fraud-dataset-neurips-2022.

[3] "Bank Account Fraud Dataset Suite (NeurIPS 2022)," Kaggle, November 29, 2023, https://www.kaggle.com/datasets/sgpjesus/bank-account-fraud-dataset-neurips-2022.

[4] Jesus et al., "Turning the Tables: Biased, Imbalanced, Dynamic Tabular Datasets for ML Evaluation", 5.

Data Input, Cleaning, and Downsampling

As previously stated, Base.csv, the fraud data set that best represents the original data, is large, containing 1,000,000 data points. Frankly, this amount of data is too great for my computer to crunch locally, so a certain level of down-sampling will be required.

Before proceeding to data reduction, we ensure that the data is in a suitable format to be used for classification. We first check the data set for the process of any nulls.

```
library(caret)

# Read in .csv "Base"
base <- read.csv("Base.csv",sep=',')

# Check if any nulls
nulls = 0
for (i in range(1,1000000)) {
  if (any(is.na(base[i,])) == TRUE) {
    nulls = nulls + 1
  }
}
nulls
```

```
## [1] 0
```

Since nulls returns 0 at the end of this for loop, we can ensure there are no null values in our data set. This result was expected, since our data set is not comprised of true real world data and is comprised for the purposes of competition.

We proceed to dummy code the 5 categorical variables in our data set: payment_type, employment_status, housing_status, source, and device_os. This process is necessary to run sine classifiers, such as k-nearest neighbors. Without undertaking this process, attempting to run k-nearest neighbors returns an error.

Additionally, we begin the process of down-sampling. This down-sampling opportunity allows us to remedy the significant class imbalance between the two classes. In base.csv, 988,971 observations (98.90%) were true absences of fraudulent behavior, while just 11,029 observations (1.10%) were true presences of fraud. Such extreme class imbalance can cause issues for classifiers: classification methods almost always bias the more frequent class at the detriment of the less frequent one. Since our goal is to successfully classify both non-fraudulent and fraudulent behavior – *especially* fraudulent behavior – remedying this class imbalance is crucial.

To remedy this imbalance, I split the data set into absences and presences and down-sampled the majority class, absences, to match the observation count of the presences. I debated whether to so significantly down-sample the absences or do a mix of down-sampling and up-sampling the absences and presences, respectively. Ultimately, despite the fact I was potentially weaning away nearly 99% of the potential information available to me in the absences, I thought it better than encouraging overfitting by significantly expanding the number of presences in the data set.

```
library(fastDummies)

# Dummy code categorical variables
base <- fastDummies::dummy_cols(base)
# Drop original categorical columns
base <- base[-c(9,16,19,26,28)]
```

```r
# Observe count for both absences and presences of fraud
# absences
absences <- base[base$fraud_bool == 0,]
nrow(absences)
```

```
## [1] 988971
```

```r
# presences
presences <- base[base$fraud_bool == 1,]
nrow(presences)
```

```
## [1] 11029
```

```r
# Down-sample absences to match presences

set.seed(0)

absences_down <- absences[sample(nrow(absences),
                                 11029,
                                 replace = FALSE,
                                 prob = NULL),]

# Generate final data set
frauddata <- rbind(absences_down,presences)
```

After down-sampling the data and recombining the absences and presences to generate a resulting data set, "frauddata", with 22,058 observations, I split the data set into two portions: a training portion and a testing portion.

```r
# Split into training and testing
splitsample <- sample(c(TRUE,FALSE), nrow(frauddata),
                replace=TRUE, prob=c(0.8,0.2))

set.seed(0)

# creating training dataset
frauddata_train <- frauddata[splitsample, ]

# creating testing dataset
frauddata_test <- frauddata[!splitsample, ]

# Observe count for both absences and presences in frauddata_train
# absences
absences <- frauddata_train[frauddata_train$fraud_bool == 0,]
nrow(absences)
```

```
## [1] 8856
```

```r
# presences
presences <- frauddata_train[frauddata_train$fraud_bool == 1,]
nrow(presences)
```

```
## [1] 8900
```

```
# Observe count for both absences and presences in frauddata_test
# absences
absences <- frauddata_test[frauddata_test$fraud_bool == 0,]
nrow(absences)
```

```
## [1] 2173
```

```
# presences
presences <- frauddata_test[frauddata_test$fraud_bool == 1,]
nrow(presences)
```

```
## [1] 2129
```

Model Running

With each classifier, a small table will be provided to summarize results. These tables will be compiled in the conclusion section to display final results.

Linear Discriminate Analysis

First, I ran Linear Discriminate Analysis and Quadratic Discriminate Analysis on the data. Despite being relatively simple methods, I immediately encountered issues with my data. I was running into errors left and right and eventually turned to ChatGPT to help troubleshoot. ChatGPT recognized the error I was receiving as the fault of multicollinearity and provided me code with which to remedy the issue.[5] I greatly simplified the code with some assistance from Stack Overflow.

At first I realized I had fallen for the "dummy trap" – which is partially true. I made sure to run the reduced dataset, multicollinear elements removed, against other models I had already ran, including logistic regression and the single classification tree. The difference between the two datasets had negligible effect, so I continued with frauddata_train.

Addressing multicollinearity allowed me to solve the issues associated with LDA. QDA proved a different matter: I have repeatedly received the error "Error in qda.default(x, grouping, ...) : rank deficiency in group 0". I again consulted both StackOverflow and ChatGPT to assist with troubleshooting.[6] The methods they provided failed; addressing the same issues with LDA's multicollinearity failed; attempting both LDA's multicollinearity removal methods and fully addressing the "dummy trap" additionally failed.

As a result of being stumped and finding no solutions to my dilemma, I have decided to about Quadratic Discriminatory Analysis and continue onto further models. In exchange for this shortcoming, I have fit both Adaboost and a Gradient Boosting Machine (the latter both tuned and untuned), to great effect.

```
library(MASS)
library(verification)
library(heplots)

# Run on training data, cross-validated, k=10
set.seed(0)

linear_combos = caret::findLinearCombos(frauddata_train)

frauddata_clean <- frauddata_train[,-c(linear_combos$remove)]

fraud_lda <- lda(fraud_bool ~ ., CV = TRUE, data = frauddata_clean)
caret::confusionMatrix(fraud_lda$class,
                       factor(frauddata_clean$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 7171 1949
##          1 1685 6951
##
```

[5] ChatGPT. "LDA Constant Variable Error," April 2025. https://chatgpt.com/share/680bee52-9118-8010-8d2d-f296c0d84df2.

[6] "LDA Constant Variable Error," ChatGPT, April 2025, https://chatgpt.com/share/680bee52-9118-8010-8d2d-f296c0d84df2.

```
##                Accuracy : 0.7953
##                  95% CI : (0.7893, 0.8013)
##     No Information Rate : 0.5012
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.5907
##
##  Mcnemar's Test P-Value : 1.284e-05
##
##             Sensitivity : 0.8097
##             Specificity : 0.7810
##          Pos Pred Value : 0.7863
##          Neg Pred Value : 0.8049
##              Prevalence : 0.4988
##          Detection Rate : 0.4039
##    Detection Prevalence : 0.5136
##       Balanced Accuracy : 0.7954
##
##        'Positive' Class : 0
##
```

```
# Run on testing data

frauddata_test_clean <- frauddata_test[,-c(linear_combos$remove)]
fraud_lda_test <- predict(lda(fraud_bool ~ ., data = frauddata_clean),
                          frauddata_test_clean,type="response")
caret::confusionMatrix(fraud_lda_test$class,
                                 factor(frauddata_test$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 1759  448
##          1  414 1681
##
##                Accuracy : 0.7996
##                  95% CI : (0.7873, 0.8115)
##     No Information Rate : 0.5051
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.5991
##
##  Mcnemar's Test P-Value : 0.261
##
##             Sensitivity : 0.8095
##             Specificity : 0.7896
##          Pos Pred Value : 0.7970
##          Neg Pred Value : 0.8024
##              Prevalence : 0.5051
##          Detection Rate : 0.4089
##    Detection Prevalence : 0.5130
##       Balanced Accuracy : 0.7995
##
```

```
##        'Positive' Class : 0
##
```

As our first classifier, LDA fittingly falls roughly in the middle of the spectrum of classifiers fit to this data set, with a cross-validated training accuracy of 79.53% and an accuracy on the testing data of 79.96%. I was surprised to see that the model fit the testing data better than the cross-validated training data – this theme would continue across multiple classifiers.

This higher accuracy is typically the result of improved specificity moving from the training data to the testing data. I expect this may be the result of the test data set having a higher proportion of absences compared to the training data set, though that may be negligible: the training data set has 49.88% absences while the 50.51% absences.

METHOD	ACCURACY	SENSITIVITY	SPECIFICITY
LDA, cv	79.53	80.97	78.10
LDA, test data	79.96	80.95	78.96

Logistic Regression

```
# Run on training data, cross-validated, k=10

set.seed(0)

fraud_logistic <- rep(0, nrow(frauddata_train))
xvs <- rep(1:10, length = nrow(frauddata_train))
xvs <- sample(xvs)
for (i in 1:10) {
  train <- frauddata_train[xvs != i, ]
  test <- frauddata_train[xvs == i, ]
  glub <- glm(fraud_bool ~ ., family = binomial, data = train)
  fraud_logistic[xvs == i] <- predict(glub, test, type = "response")
}

  # Display confusion matrix using caret
caret::confusionMatrix(as.factor(round(fraud_logistic)), as.factor(frauddata_train$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 7097 1850
##          1 1759 7050
##
##                Accuracy : 0.7967
##                  95% CI : (0.7907, 0.8026)
##     No Information Rate : 0.5012
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.5935
##
##  Mcnemar's Test P-Value : 0.1341
##
```

```
##             Sensitivity : 0.8014
##             Specificity : 0.7921
##          Pos Pred Value : 0.7932
##          Neg Pred Value : 0.8003
##              Prevalence : 0.4988
##          Detection Rate : 0.3997
##    Detection Prevalence : 0.5039
##       Balanced Accuracy : 0.7968
##
##        'Positive' Class : 0
##
```

```r
# Run on testing data

set.seed(0)

fraud_logistic_test <- predict(glm(fraud_bool ~ ., family = binomial,
                                   data = frauddata_train),
                               frauddata_test, type = "response")

print("logistic, test")
```

```
## [1] "logistic, test"
```

```r
caret::confusionMatrix(as.factor(round(fraud_logistic_test)), as.factor(frauddata_test$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 1750  440
##          1  423 1689
##
##                Accuracy : 0.7994
##                  95% CI : (0.7871, 0.8113)
##     No Information Rate : 0.5051
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.5987
##
##  Mcnemar's Test P-Value : 0.586
##
##             Sensitivity : 0.8053
##             Specificity : 0.7933
##          Pos Pred Value : 0.7991
##          Neg Pred Value : 0.7997
##              Prevalence : 0.5051
##          Detection Rate : 0.4068
##    Detection Prevalence : 0.5091
##       Balanced Accuracy : 0.7993
##
##        'Positive' Class : 0
##
```

To my surprise, the test data set outperformed the cross-validated data set in terms of absolute accuracy. Both classifiers achieved in the ballpark of 79% total accuracy, with the cross-validated training model at 79.67% and the testing model at 79.94%. Again, the testing data set performs better than the training data set, though this time as a result of both higher sensitivity (80.14% to 80.53%) and higher specificity (79.21% to 79.33%).

METHOD	ACCURACY	SENSITIVITY	SPECIFICITY
Logistic, cv	79.67	80.14	79.21
Logistic, test data	79.94	80.53	79.33

K-Nearest Neighbors

```
# Run on training data, cross-validated, k=10

library(klaR)
set.seed(0)

TrainData <- frauddata_train[,2:53]
TrainClasses <- as.factor(frauddata_train[,1])
knnFit_fraud <- train(TrainData, TrainClasses,
                  method = "knn",
                  tuneLength = 6,
                  trControl = trainControl(method = "cv",
                                           savePredictions = "final"))

k = knnFit_fraud$bestTune
k
```

```
##    k
## 6 15
```

```
# Display Confusion Matrix
caret::confusionMatrix(as.factor(knnFit_fraud$pred$pred), as.factor(knnFit_fraud$pred$obs))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 5636 3811
##          1 3220 5089
##
##                Accuracy : 0.604
##                  95% CI : (0.5968, 0.6112)
##     No Information Rate : 0.5012
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.2082
##
##  Mcnemar's Test P-Value : 1.974e-12
##
##             Sensitivity : 0.6364
```

```
##             Specificity : 0.5718
##          Pos Pred Value : 0.5966
##          Neg Pred Value : 0.6125
##              Prevalence : 0.4988
##          Detection Rate : 0.3174
##    Detection Prevalence : 0.5320
##       Balanced Accuracy : 0.6041
##
##        'Positive' Class : 0
##
```

```
set.seed(0)
# Run on testing data
fraud_knn_test <- predict(knnFit_fraud, frauddata_test[,2:53])

print("k-nearest neighbors, test")
```

```
## [1] "k-nearest neighbors, test"
```

```
caret::confusionMatrix(as.factor(fraud_knn_test),
                       as.factor(frauddata_test$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 1338  890
##          1  835 1239
##
##                Accuracy : 0.599
##                  95% CI : (0.5842, 0.6137)
##     No Information Rate : 0.5051
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.1978
##
##  Mcnemar's Test P-Value : 0.1935
##
##             Sensitivity : 0.6157
##             Specificity : 0.5820
##          Pos Pred Value : 0.6005
##          Neg Pred Value : 0.5974
##              Prevalence : 0.5051
##          Detection Rate : 0.3110
##    Detection Prevalence : 0.5179
##       Balanced Accuracy : 0.5989
##
##        'Positive' Class : 0
##
```

Of all classifiers fit to the fraud data, k-nearest neighbors performed by far the worse, performing nearly 20% worse than the rest of the cohort! k proves to be surprisingly high when tuned, yielding $k = 15$. Given that k is often a single digit number, this result surprised me.

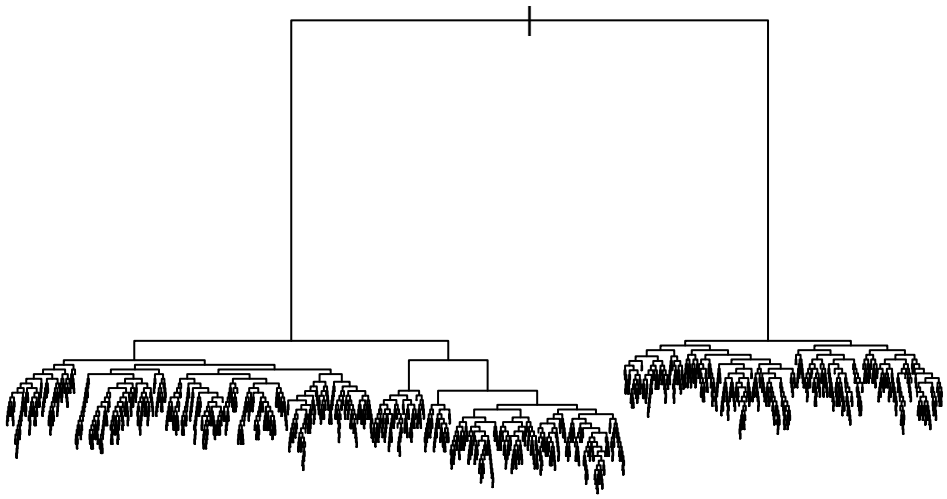
Compared to logistic, the difference between the sensitivity and specificity is notable with k-nearest neighbors. The gap between sensitivity and specificity for the logistic model on the testing data was just 1.2%, while the difference between the two for the k-nearest neighbors model on the testing data was 3.37%. While the exact values may differ from seed to seed, the gap is greater for the k-nearest neighbors model on every seed I've tested.

The model struggles to classify everyday causes as non-fraudulent. The overall failure of the model may be the result of cases on the boundary, with suspicious – but non-fraudulent – behavior being linked closer to fraud in terms of distance.

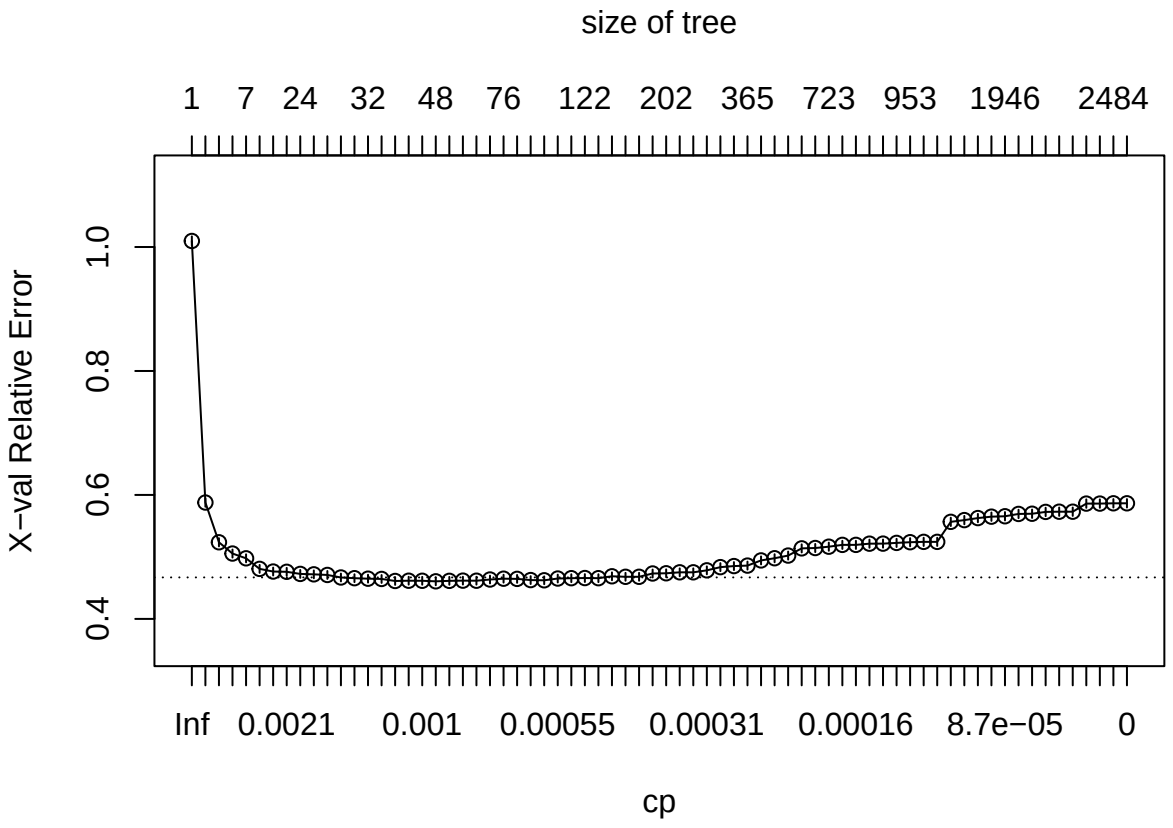
METHOD	ACCURACY	SENSITIVITY	SPECIFICITY
k-nearest, cv	60.40	63.64	57.18
k-nearest, test data	59.90	61.57	58.20

Single Classification Tree

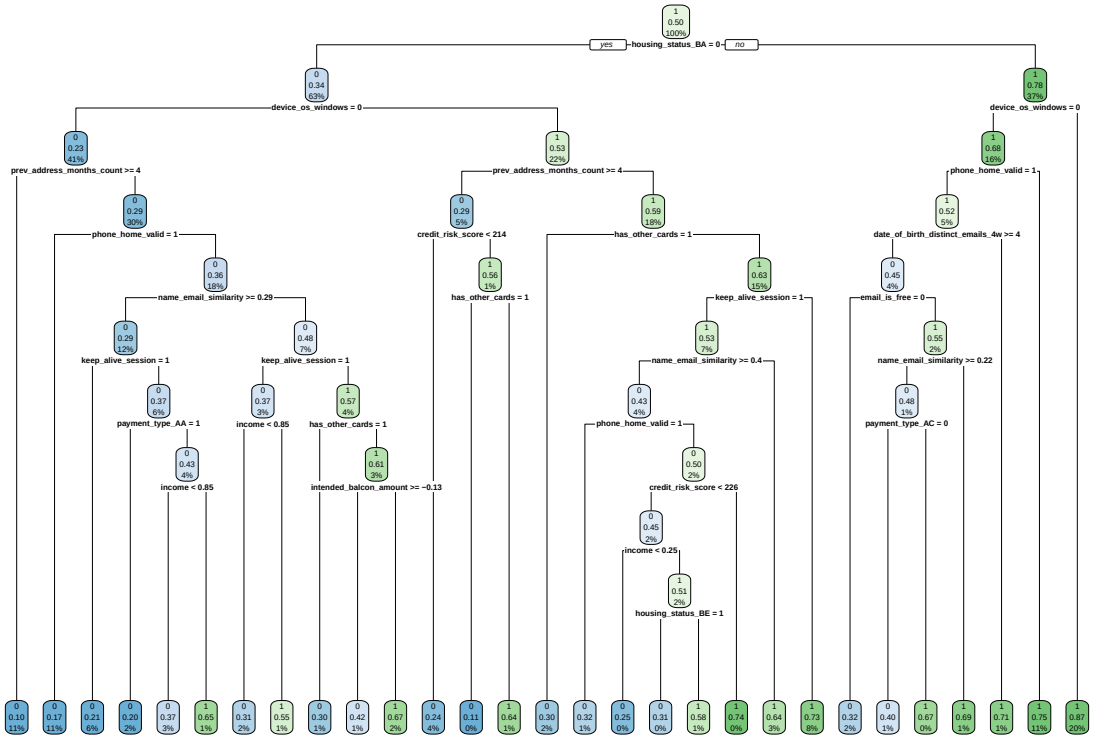
```
# Run on training data, cross-validated, k=10

library(rpart)

# PLOT A FULL TREE
set.seed(8151)

fraud_rpartfull <- rpart(fraud_bool ~ ., method = "class",
                         control = rpart.control(cp = 0.0,
                                                 minsplit = 2),
                         data = frauddata_train)

plot(fraud_rpartfull)
```



```r
plotcp(fraud_rpartfull) # Smallest tree which meets the 1SE rule: cp = 0.0015
```




```
# Plot the smallest tree meeting 1-SE rule

fraud_rpart_prune <- rpart(fraud_bool ~ ., method = "class",
                           control = rpart.control(cp = 0.0015,
                                                   minsplit = 2),
                           data = frauddata_train)

rpart.plot::rpart.plot(fraud_rpart_prune)
```




```
# Display confusion matrix for cross-validated smallest tree

  # Set up k-fold cross validation, k = 10
xvs <- rep(c(1:10), length = nrow(frauddata_train))
xvs <- sample(xvs)
fraud_rpart_prune_cv <- rep(0, length(nrow(frauddata_train)))
for (i in 1:10) {
  train <- frauddata_train[xvs != i, ]
  test <- frauddata_train[xvs == i, ]
  rp <- rpart(fraud_bool ~ ., method = "class",
                         control = rpart.control(cp = 0.0015,
                                                 minsplit = 2),
                         data = frauddata_train)
  fraud_rpart_prune_cv[xvs == i] <- predict(rp, test,
                                            type = "prob")[, 2]
}

  # Display confusion matrix using caret
caret::confusionMatrix(as.factor(round(fraud_rpart_prune_cv)),
                       as.factor(frauddata_train$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 6738 1819
```

```
##          1 2118 7081
##
##                Accuracy : 0.7783
##                  95% CI : (0.7721, 0.7844)
##     No Information Rate : 0.5012
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.5565
##
##  Mcnemar's Test P-Value : 2.041e-06
##
##             Sensitivity : 0.7608
##             Specificity : 0.7956
##          Pos Pred Value : 0.7874
##          Neg Pred Value : 0.7698
##              Prevalence : 0.4988
##          Detection Rate : 0.3795
##    Detection Prevalence : 0.4819
##       Balanced Accuracy : 0.7782
##
##        'Positive' Class : 0
##
```

```
# Run on testing data
set.seed(0)

# RUN ON MULL_TEST DATA
fraud_tree_test <- rpart(fraud_bool ~ ., method = "class",
                         control = rpart.control(cp = 0.00018, minsplit = 2),
                         data = frauddata_train)

fraud_test_tree_pred <- predict(fraud_tree_test,
                                frauddata_test, type = "prob")[,2]

print("class tree, test")
```

```
## [1] "class tree, test"
```

```
caret::confusionMatrix(as.factor(round(fraud_test_tree_pred)),
                       as.factor(frauddata_test$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 1641  504
##          1  532 1625
##
##                Accuracy : 0.7592
##                  95% CI : (0.7461, 0.7719)
##     No Information Rate : 0.5051
##     P-Value [Acc > NIR] : <2e-16
##
```

```
##                  Kappa : 0.5184
##
##  Mcnemar's Test P-Value : 0.4016
##
##             Sensitivity : 0.7552
##             Specificity : 0.7633
##          Pos Pred Value : 0.7650
##          Neg Pred Value : 0.7534
##              Prevalence : 0.5051
##          Detection Rate : 0.3815
##    Detection Prevalence : 0.4986
##       Balanced Accuracy : 0.7592
##
##        'Positive' Class : 0
##
```

While not as strong as logistic regression, a classification tree significantly outperforms k-nearest neighbors, with an overall classification accuracy of the test data set of 75.92%. On both the training and testing data, the resulting models had better specificity than sensitivity, indicating that they were better at predicting absences than presences. If this imbalance were exaggerated when applied to a full data set, a single classification tree may provide less utility to an analyst than other methods with better sensitivity.

METHOD	ACCURACY	SENSITIVITY	SPECIFICITY
class. tree, cv	77.83	76.08	79.56
class. tree, test	75.92	75.52	76.33

Random Forest

```
# Run on training data, cross-validated, k=10

library(randomForest)
set.seed(0)

fraud_rf <- randomForest(as.factor(fraud_bool) ~ .,
                         data = frauddata_train)

fraud_rf_pred <- predict(fraud_rf, frauddata_train,
                         type = "prob")[,2]

print("RF, train")
```

```
## [1] "RF, train"
```

```
caret::confusionMatrix(as.factor(fraud_rf$predicted),
                       as.factor(frauddata_train$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
```

```
##          0 7120 1813
##          1 1736 7087
##
##                Accuracy : 0.8001
##                  95% CI : (0.7942, 0.806)
##     No Information Rate : 0.5012
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.6003
##
##  Mcnemar's Test P-Value : 0.202
##
##             Sensitivity : 0.8040
##             Specificity : 0.7963
##          Pos Pred Value : 0.7970
##          Neg Pred Value : 0.8032
##              Prevalence : 0.4988
##          Detection Rate : 0.4010
##    Detection Prevalence : 0.5031
##       Balanced Accuracy : 0.8001
##
##        'Positive' Class : 0
##
```

```
# Run on testing data
set.seed(0)
fraud_rf_test_pred <- predict(fraud_rf, frauddata_test,
                              type = "prob")[,2]

print("RF, test")
```

```
## [1] "RF, test"
```

```
caret::confusionMatrix(as.factor(round(fraud_rf_test_pred)),
                       as.factor(frauddata_test$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 1745  420
##          1  428 1709
##
##                Accuracy : 0.8029
##                  95% CI : (0.7907, 0.8147)
##     No Information Rate : 0.5051
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.6057
##
##  Mcnemar's Test P-Value : 0.81
##
##             Sensitivity : 0.8030
```

```
##             Specificity : 0.8027
##          Pos Pred Value : 0.8060
##          Neg Pred Value : 0.7997
##              Prevalence : 0.5051
##          Detection Rate : 0.4056
##    Detection Prevalence : 0.5033
##       Balanced Accuracy : 0.8029
##
##        'Positive' Class : 0
##
```

Random Forest is thus far our strongest classifier on the testing data, with an overall accuracy of 80.29%. Running the model on the testing data has a similar result to the cross-validated training data, again a result of the testing data having improved specificity. This came as a surprise, given that the single decision tree had notably worse specificity on the test data set (3.23% worse than training). I had expected that tree based methods (single classification tree, Random Forest, Adaboost) may reflect similar results here.

This Random Forest model notably outperformed a single classification tree.

METHOD	ACCURACY	SENSITIVITY	SPECIFICITY
Random Forest, cv	80.01	80.40	79.63
Random Forest, test	80.29	80.30	80.27

Adaboost

```
# Run on training data, cross-validated, k=10

library(ada)
set.seed(0)

ada_fraud_untuned = rep(0,nrow(frauddata_train))
xvs = rep(1:10,length=nrow(frauddata_train))
xvs = sample(xvs)
for(i in 1:10){
  train = frauddata_train[xvs!=i,]
  test = frauddata_train[xvs==i,]
  glub = ada(as.factor(fraud_bool)~ . ,loss="exponential",
             data=train)
  ada_fraud_untuned[xvs==i] = predict(glub,newdata=test,
                                      type="prob")[,2]
}

print("ada, cv")
```

```
## [1] "ada, cv"
```

```
caret::confusionMatrix(as.factor(round(ada_fraud_untuned)), as.factor(frauddata_train$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
```

```
## Prediction    0    1
##          0 7090 1837
##          1 1766 7063
##
##                Accuracy : 0.7971
##                  95% CI : (0.7911, 0.803)
##     No Information Rate : 0.5012
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.5942
##
##  Mcnemar's Test P-Value : 0.2435
##
##             Sensitivity : 0.8006
##             Specificity : 0.7936
##          Pos Pred Value : 0.7942
##          Neg Pred Value : 0.8000
##              Prevalence : 0.4988
##          Detection Rate : 0.3993
##    Detection Prevalence : 0.5028
##       Balanced Accuracy : 0.7971
##
##        'Positive' Class : 0
##
```

```
# Run on testing data

set.seed(0)
ada_fraud_test <- predict(ada(as.factor(fraud_bool)~ .,
                          loss="exponential",
                          data=frauddata_train),
                          frauddata_test,
                          type = "response")

print("ada, test")
```

```
## [1] "ada, test"
```

```
caret::confusionMatrix(as.factor(ada_fraud_test),
                       as.factor(frauddata_test$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 1721  425
##          1  452 1704
##
##                Accuracy : 0.7961
##                  95% CI : (0.7838, 0.8081)
##     No Information Rate : 0.5051
##     P-Value [Acc > NIR] : <2e-16
##
```

```
##                  Kappa : 0.5923
##
##  Mcnemar's Test P-Value : 0.38
##
##             Sensitivity : 0.7920
##             Specificity : 0.8004
##          Pos Pred Value : 0.8020
##          Neg Pred Value : 0.7904
##              Prevalence : 0.5051
##          Detection Rate : 0.4000
##    Detection Prevalence : 0.4988
##       Balanced Accuracy : 0.7962
##
##        'Positive' Class : 0
##
```

Adaboost performed similarly to both Random Forest and logistic regression, capping out in the ballpark of 80%. The cross-validated training model had an overall accuracy of 79.71%, while the model ran on the testing data had an accuracy of 79.61%. While the cross-validated training had a weaker specificity than sensitivity, the testing data closed this gap, with specificity outpacing sensitivity by 0.84%.

METHOD	ACCURACY	SENSITIVITY	SPECIFICITY
Adaboost, cv	79.71	80.06	79.36
Adaboost, test	79.61	79.20	80.04

Gradient Boosting Machine (Untuned)

```
# Run on training data, cross-validated, k=10

library(gbm)
set.seed(0)

gbm_fraud_untuned = rep(0,nrow(frauddata_train))
xvs = rep(1:10,length=nrow(frauddata_train))
xvs = sample(xvs)
for(i in 1:10){
  train = frauddata_train[xvs!=i,]
  test = frauddata_train[xvs==i,]
  glub = gbm(fraud_bool ~ ., distribution="bernoulli",
             n.trees=5000, data=train)
  gbm_fraud_untuned[xvs==i]=predict(glub, newdata=test,
                                    type="response",
                                   n.trees=5000)
}

print("gbm, untuned, cv")
```

```
## [1] "gbm, untuned, cv"
```

```
caret::confusionMatrix(as.factor(round(gbm_fraud_untuned)), as.factor(frauddata_train$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 7203 1688
##          1 1653 7212
##
##                Accuracy : 0.8118
##                  95% CI : (0.806, 0.8176)
##     No Information Rate : 0.5012
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.6237
##
##  Mcnemar's Test P-Value : 0.5564
##
##             Sensitivity : 0.8133
##             Specificity : 0.8103
##          Pos Pred Value : 0.8101
##          Neg Pred Value : 0.8135
##              Prevalence : 0.4988
##          Detection Rate : 0.4057
##    Detection Prevalence : 0.5007
##       Balanced Accuracy : 0.8118
##
##        'Positive' Class : 0
##
```

```
# Run on testing data

set.seed(0)
gbm_fraud_test <- predict(gbm(fraud_bool ~ .,
                              distribution="bernoulli",
                              n.trees=5000,
                              data=frauddata_train),
                          frauddata_test, type = "response")

print("gbm, untuned, test")
```

```
## [1] "gbm, untuned, test"
```

```
caret::confusionMatrix(as.factor(round(gbm_fraud_test)), as.factor(frauddata_test$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 1783  420
##          1  390 1709
##
```

```
##                Accuracy : 0.8117
##                  95% CI : (0.7997, 0.8233)
##     No Information Rate : 0.5051
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.6233
##
##  Mcnemar's Test P-Value : 0.3082
##
##             Sensitivity : 0.8205
##             Specificity : 0.8027
##          Pos Pred Value : 0.8094
##          Neg Pred Value : 0.8142
##              Prevalence : 0.5051
##          Detection Rate : 0.4145
##    Detection Prevalence : 0.5121
##       Balanced Accuracy : 0.8116
##
##        'Positive' Class : 0
##
```

Of all models so far, the untuned Gradient Boosting Machine had the best performance, breaking the ~80% barrier, with both the training and testing data posting over 81% accuracy. The cross-validated training data was 81.18% accurate while the testing data was 81.17% accurate. When put to the testing data, sensitivity rose and specificity fell.

METHOD	ACCURACY	SENSITIVITY	SPECIFICITY
GBM untuned, cv	81.18	81.33	81.03
GBM untuned, test	81.17	82.05	80.27

Gradient Boosting Machine (Tuned)

The following code was used to tune the Gradient Boosting Machine. I tuned three hyperparameters: interaction.depth, n.trees, and shrinkage, each of which using the grid method. These method was undertaken thrice to hone in on the best tuning for the model.

```
fitControl = trainControl(method = "cv", number = 10)
set.seed(424)

gbmGrid = expand.grid(interaction.depth = c(12, 14, 16, 18, 20),
                      n.trees = c(25,50,75,100),
                      shrinkage = c(0.01, 0.05, 0.1, 0.2 ),
                      n.minobsinnode=10)
# BEST RESULT: 0.1 shrinkage, 20 depth, 100 trees
# 0.813

gbmFit = train(as.factor(fraud_bool)~ ., method="gbm",
               tuneGrid = gbmGrid,
               trControl = fitControl,
               data=frauddata_train)

gbmFit
```

```
gbmGrid2 = expand.grid(interaction.depth = c(18, 19, 20, 21, 22),
                       n.trees = c(95,100,105,110),
                       shrinkage = c(0.09, 0.1, 0.11, 0.12 ),
                       n.minobsinnode=10)

gbmFit2 = train(as.factor(fraud_bool)~ ., method="gbm",
                tuneGrid = gbmGrid2,
               trControl = fitControl,
               data=frauddata_train)

gbmFit2
# BEST RESULT: 0.11 shrinkage, 19 depth, 110 trees
# 0.8138



gbmGrid3 = expand.grid(interaction.depth = c(18, 19, 20),
                       n.trees = c(105,110,115,120),
                       shrinkage = c(0.105, 0.11, 0.115),
                       n.minobsinnode= c(10,15))

gbmFit3 = train(as.factor(fraud_bool)~ ., method="gbm",
                tuneGrid = gbmGrid3,
               trControl = fitControl,
               data=frauddata_train)

gbmFit3
# BEST RESULT: 0.11 shrinkage, 19 depth, 110 trees
# 0.8138
```

We proceed to run the fit model on both the cross-validated training data and the testing data:

```
# Run on training data, cross-validated, k=10

set.seed(0)

gbm_fraud_tuned = rep(0,nrow(frauddata_train))
xvs = rep(1:10,length=nrow(frauddata_train))
xvs = sample(xvs)
for(i in 1:10){
  train = frauddata_train[xvs!=i,]
  test = frauddata_train[xvs==i,]
  glub = gbm(fraud_bool ~ ., distribution="bernoulli",
             interaction.depth=20,
             shrinkage=0.11, n.trees=110,
             n.minobsinnode=10, data=train)
  gbm_fraud_tuned[xvs==i]=predict(glub, newdata=test,
                                  type="response",
                                  n.trees=5000)
}

print("gbm, tuned, cv")
```

```
## [1] "gbm, tuned, cv"
```

```
caret::confusionMatrix(as.factor(round(gbm_fraud_tuned)), as.factor(frauddata_train$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 7154 1676
##          1 1702 7224
##
##                Accuracy : 0.8098
##                  95% CI : (0.8039, 0.8155)
##     No Information Rate : 0.5012
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.6195
##
##  Mcnemar's Test P-Value : 0.6671
##
##             Sensitivity : 0.8078
##             Specificity : 0.8117
##          Pos Pred Value : 0.8102
##          Neg Pred Value : 0.8093
##              Prevalence : 0.4988
##          Detection Rate : 0.4029
##    Detection Prevalence : 0.4973
##       Balanced Accuracy : 0.8097
##
##        'Positive' Class : 0
##
```

```
# Run on testing data

set.seed(0)
gbm_fraud_test_tuned <- predict(gbm(fraud_bool ~ .,
                                    distribution="bernoulli",
                        interaction.depth=20,
                        shrinkage=0.11, n.trees=110,
                        n.minobsinnode=10, data=frauddata),
                        frauddata_test, type="response")

print("gbm, tuned, test")
```

```
## [1] "gbm, tuned, test"
```

```
caret::confusionMatrix(as.factor(round(gbm_fraud_test_tuned)), as.factor(frauddata_test$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
```

```
##          0 1841  309
##          1  332 1820
##
##                Accuracy : 0.851
##                  95% CI : (0.84, 0.8615)
##     No Information Rate : 0.5051
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.702
##
##  Mcnemar's Test P-Value : 0.3849
##
##             Sensitivity : 0.8472
##             Specificity : 0.8549
##          Pos Pred Value : 0.8563
##          Neg Pred Value : 0.8457
##              Prevalence : 0.5051
##          Detection Rate : 0.4279
##    Detection Prevalence : 0.4998
##       Balanced Accuracy : 0.8510
##
##        'Positive' Class : 0
##
```

While tuning the Gradient Boosting Machine did not improve the cross-validated model on the training data – in fact, the tuned gradient boosting machine performs slightly worse – the tuned model on the testing data makes a significant improvement. Total accuracy shoots up to 85.10%, by far the best of all the classifiers fit, with a sensitivity of 84.72% and a specificity of 85.49%! To confirm these results were not just a fluke, I ran the model again with a different random seed:

```
# Run on testing data, different seed

set.seed(1247)
gbm_fraud_test_tuned <- predict(gbm(fraud_bool ~ .,
                                    distribution="bernoulli",
                      interaction.depth=20,
                      shrinkage=0.11, n.trees=110,
                      n.minobsinnode=10, data=frauddata),
                      frauddata_test, type="response")

print("gbm, tuned, test (new seed)")
```

```
## [1] "gbm, tuned, test (new seed)"
```

```
caret::confusionMatrix(as.factor(round(gbm_fraud_test_tuned)), as.factor(frauddata_test$fraud_bool))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1
##          0 1845  301
##          1  328 1828
##
```

```
##                Accuracy : 0.8538
##                  95% CI : (0.8429, 0.8642)
##     No Information Rate : 0.5051
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.7076
##
##  Mcnemar's Test P-Value : 0.2999
##
##             Sensitivity : 0.8491
##             Specificity : 0.8586
##          Pos Pred Value : 0.8597
##          Neg Pred Value : 0.8479
##              Prevalence : 0.5051
##          Detection Rate : 0.4289
##    Detection Prevalence : 0.4988
##       Balanced Accuracy : 0.8538
##
##        'Positive' Class : 0
##
```

Testing on another random seed, the overall accuracy, sensitivity, and specificity all increased. For that seed, specificity also beat out sensitivity, though that is dependent on the seed, as I have identified seeds where sensitivity beats specificity For purposes of consistency, I kept the result generated from seed(0).

METHOD	ACCURACY	SENSITIVITY	SPECIFICITY
GBM tuned, cv	80.98	80.78	81.17
GBM tuned, test	85.10	84.72	85.49

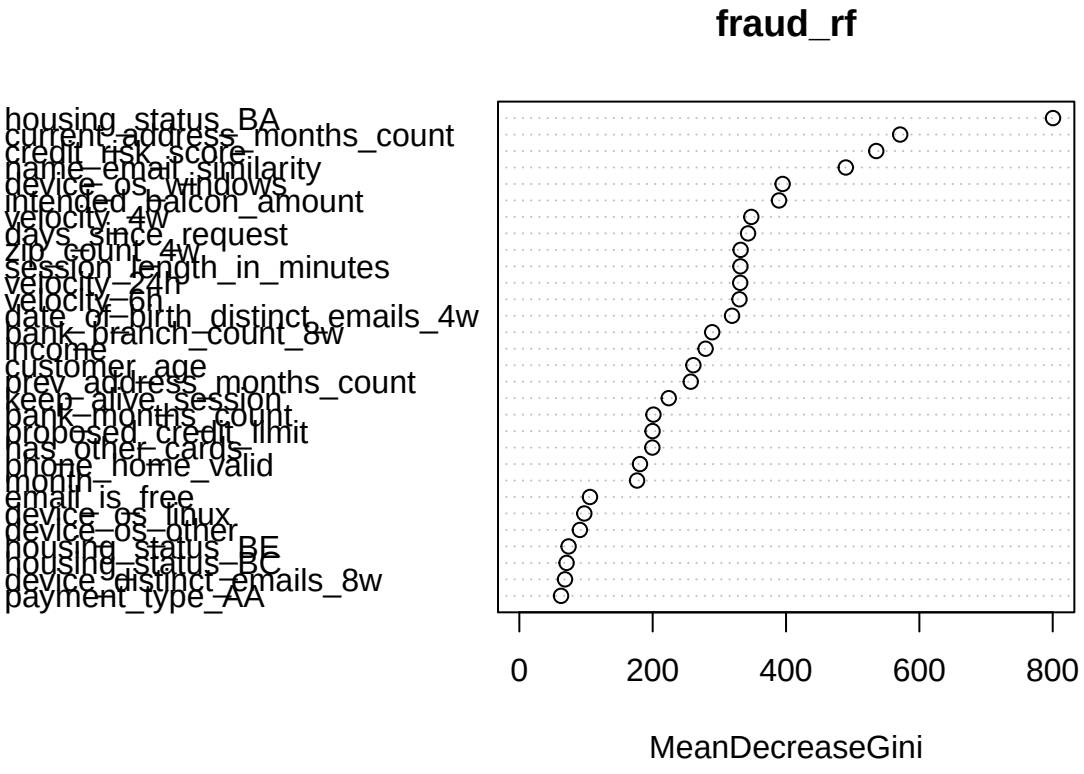
Variable Importance and Partial Dependence Plots

The Random Forest package allows for the contruction of both a variable importance plot, graphed against MeanDecreasesGini for each input feature, and partial dependence plots for each feature. These plots allow us to garner further insight into which variables are most crucial in determining classification outcomes.

Three of the top four variables did not come as a significant surprise: credit_risk_score, current_address_months_count, and name_email_similarity. A poor credit score is suspicious, and at the very least may indicate a bad investment from a bank. It appears that individuals who share bad credit scores are more prone to rash financial behavior such as fraud. The latter two variables, current_address_months_count and name_email_similarity, demonstrate stability across the account owner's life. Fraudsters are more likely to use fake addresses, both physical and e-mail, tipping off the model.

The biggest shock of all is the most influential variable, housing_status_BA. I was not expecting one of the categorical variables to tip off the model so significantly. Returning to the documentation indicates that the housing status' of individuals have been anonymized using this code structure, and this the code "BA" doesn't reflect anything tangible about the house type.[7] Perhaps making this top position even more baffling is the fact that all other housing status' fall at the very bottom of the variable importance list. Something about this specific housing type appears to be a significant indicator of fraud.

```
# Variable importance with Random Forests

varImpPlot(fraud_rf)
```



[7] "LDA Constant Variable Error," ChatGPT, April 2025, https://chatgpt.com/share/680bee52-9118-8010-8d2d-f296c0d84df2.

```
frauddata_train_features <- frauddata_train[,-1]
```

Turning to the partial dependence plots, the bland reality is that a majority of the plots are straight lines, due to the fact that they represent dummy variables which are binary. These plots demonstrate general trends among the data and illustrate how specific variables impact the outcome.

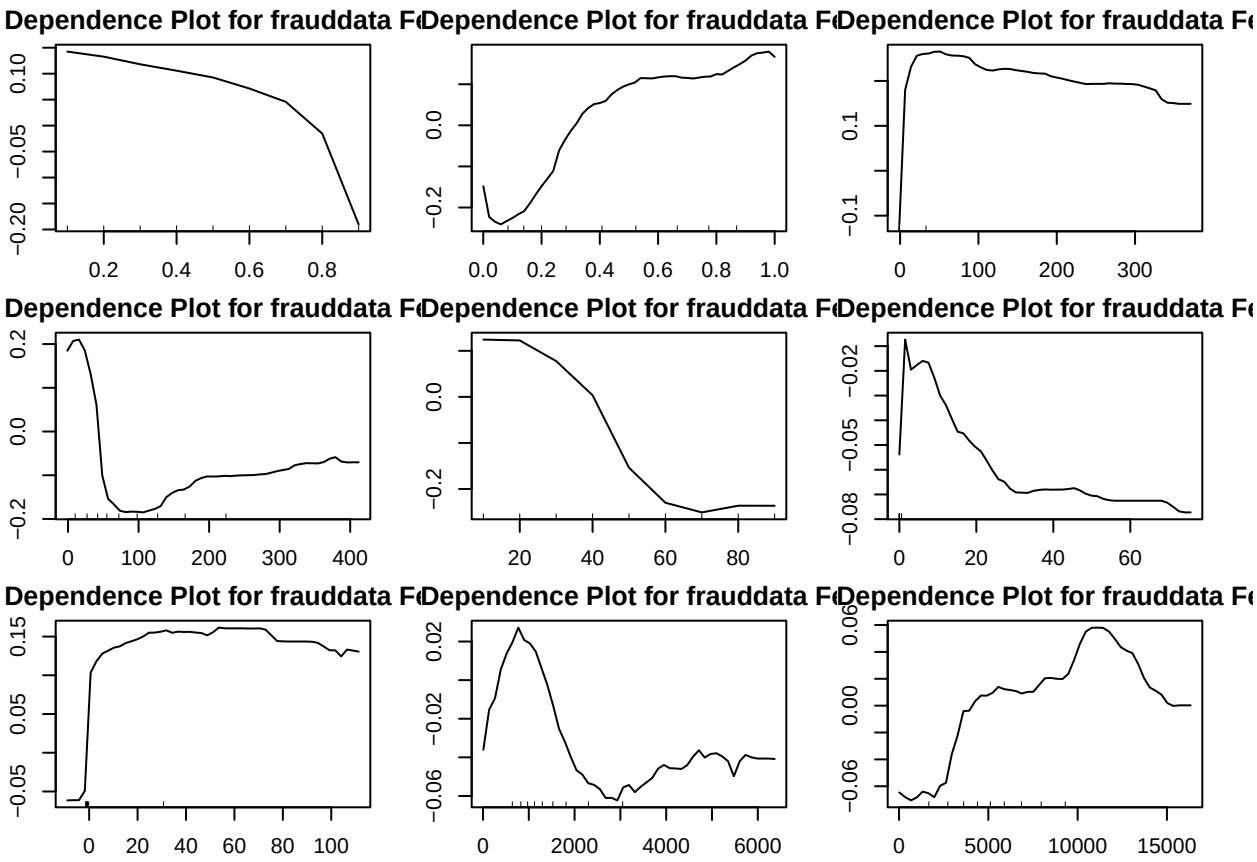
For the binary variables, while shape is not interesting, scale is. Turning our attention to housing_status_BA's plot, we see that its inclusion drops the marginal effect by 0.5 – one of the largest absolute changes across any plot. It appears that housing_status_BA notably decreases the classification of fraud. Perhaps this is some sort of stable housing, like a fully paid home, for example.

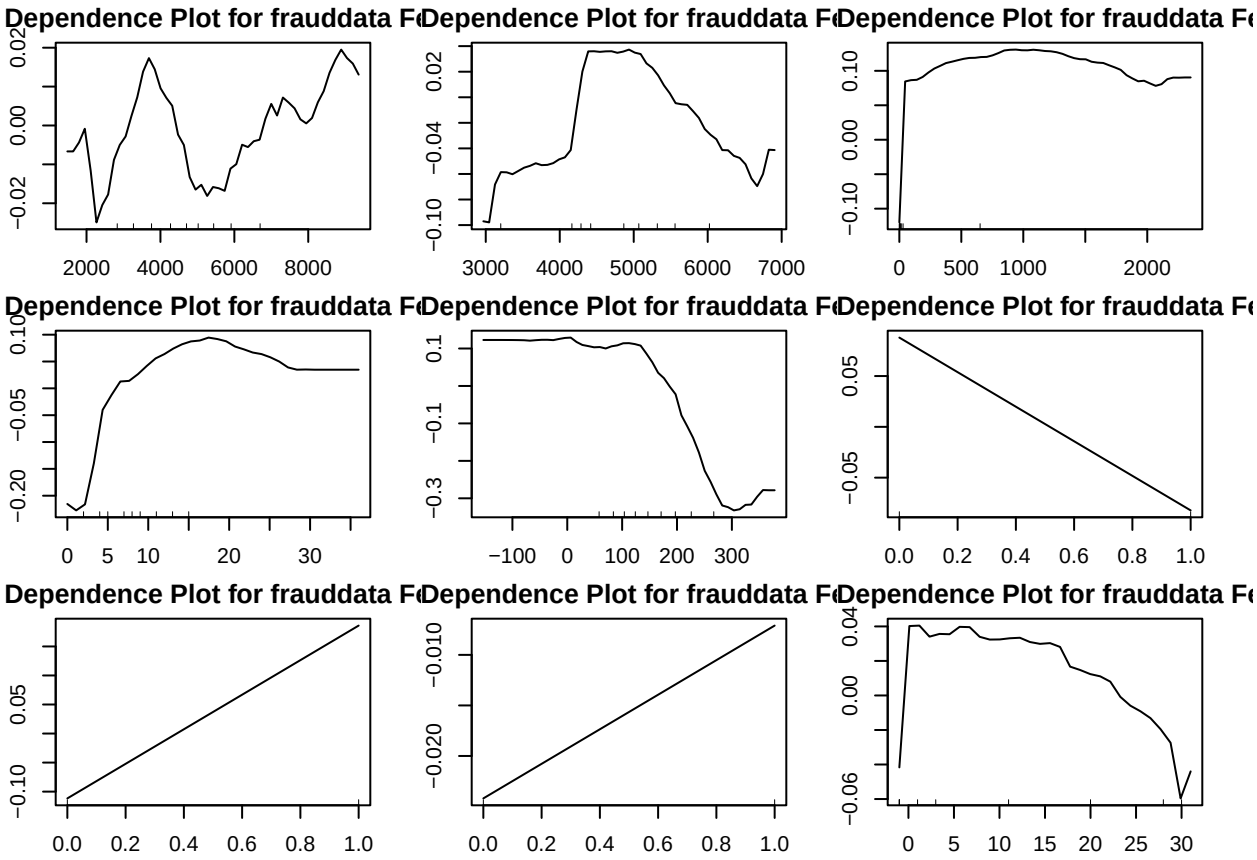
I was surprised to see that device_os_windows dropped the marginal effect by 0.4 – I did not expect there to be any correlation between OS and fraudulent behavior.

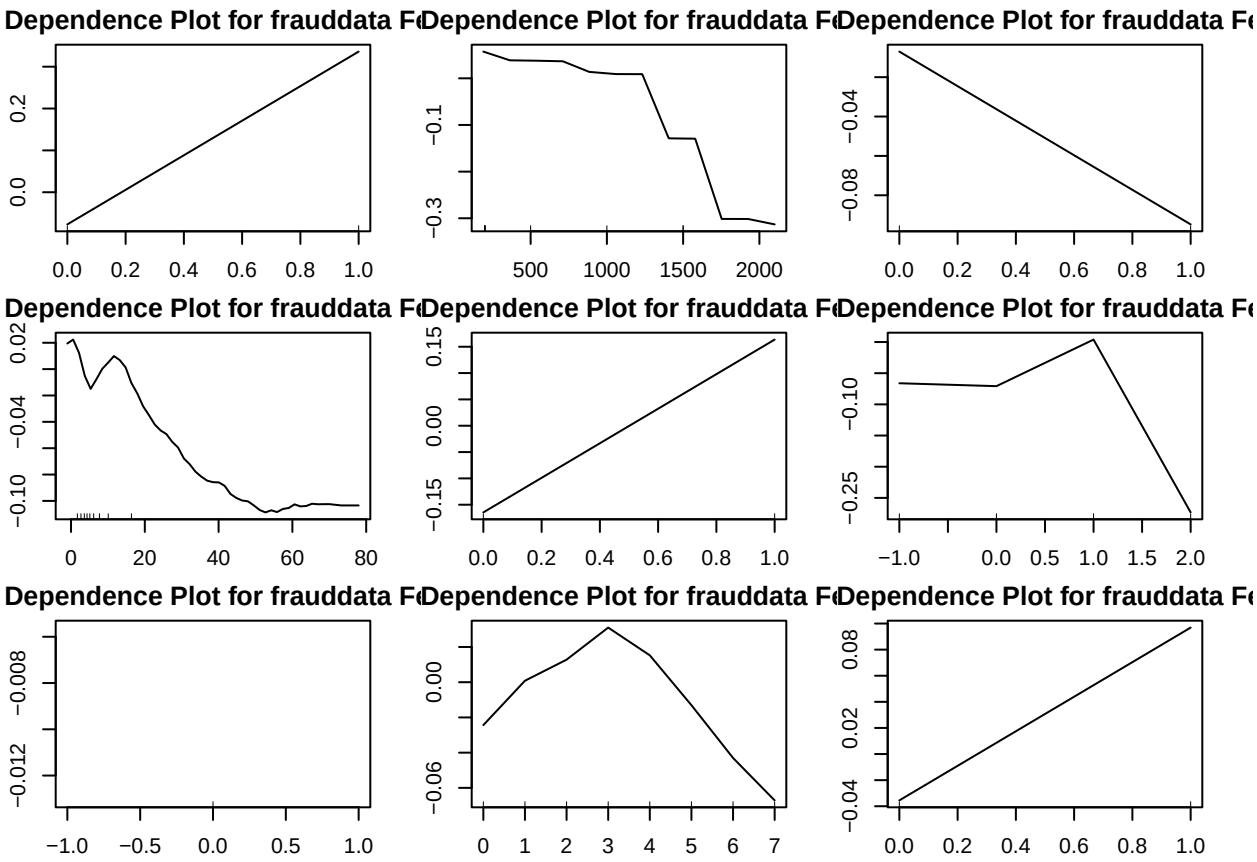
Some variables appear to have no effect when their value is low, but immediately jump when their value increases, such as prev_address_months_count and intended_balcon_amount.

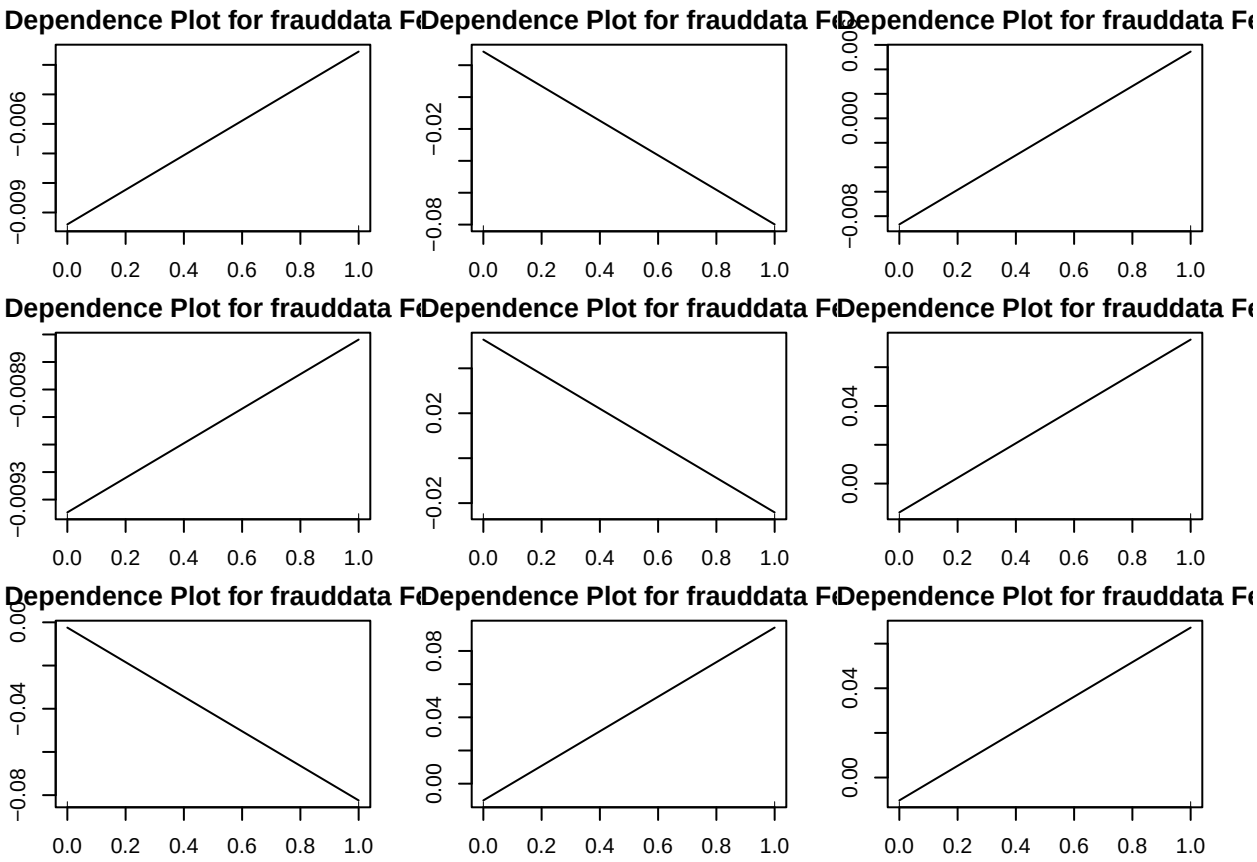
All partial dependence plots can be viewed below.

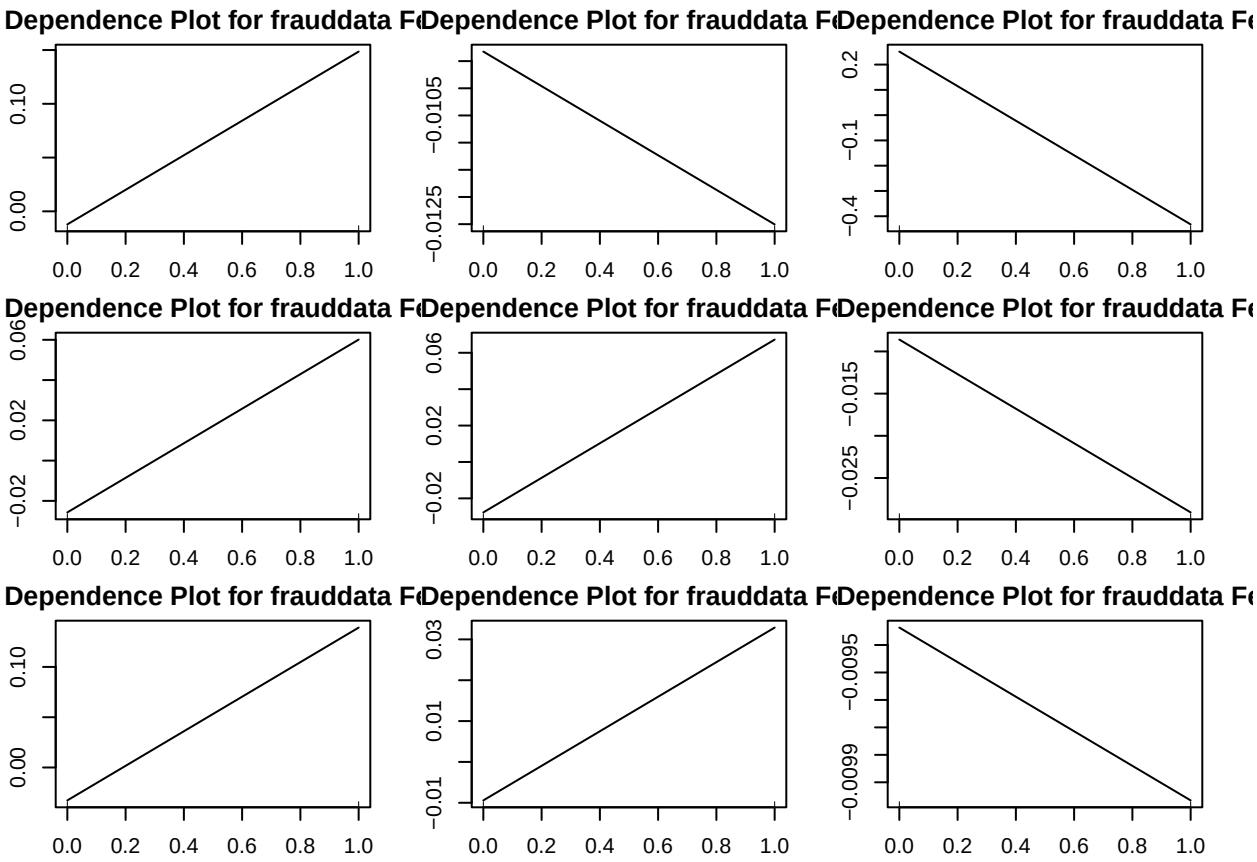
```
for (i in (1:6)) {
  par(mfrow = c(3,3))
  par(mar=c(2,2,2,2))

  end <- 9 * i
  start <- end - 8

  if (end > 52) {
    end <- 52
  }

  for (j in (start:end)) {
    partialPlot(x = fraud_rf, pred.data = frauddata_train,
                x.var = colnames(frauddata_train_features)[j],
                xlab = colnames(frauddata_train_features)[j],
                main = "Partial Dependence Plot for frauddata Features")
  }
}
```

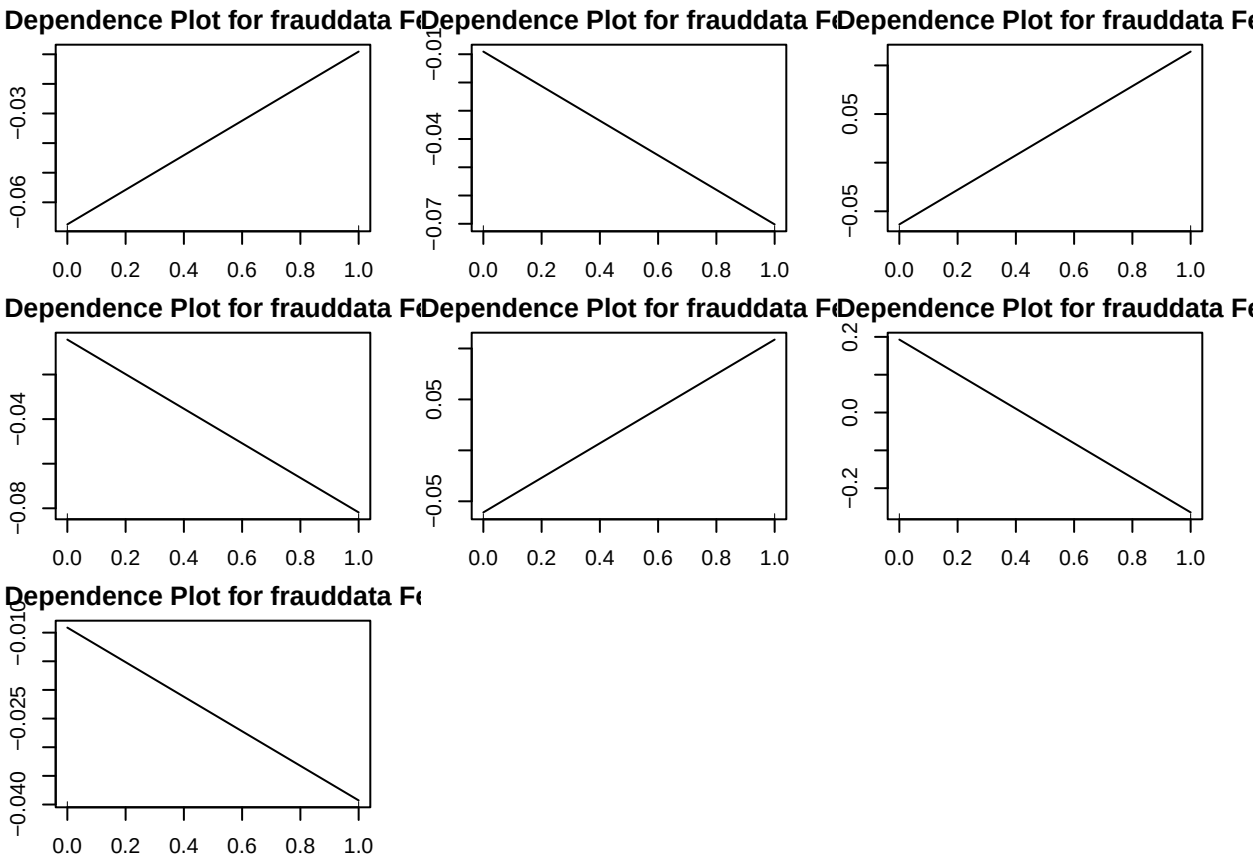

Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe


Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe


Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe


Dependence Plot for frauddata Fe
−0.009
−0.006
Dependence Plot for frauddata Fe
−0.08
−0.02
Dependence Plot for frauddata Fe
−0.008
0.000
Dependence Plot for frauddata Fe
−0.0093
−0.0089
Dependence Plot for frauddata Fe
−0.02
0.02
Dependence Plot for frauddata Fe
0.00
0.04
Dependence Plot for frauddata Fe
−0.08
−0.04
Dependence Plot for frauddata Fe
0.00
0.04
0.08
Dependence Plot for frauddata Fe
0.00
0.04
0.0
0.2
0.4
0.6
0.8
1.0


Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe
Dependence Plot for frauddata Fe


Dependence Plot for frauddata Fe
−0.06
−0.03
0.0 0.2 0.4 0.6 0.8 1.0
Dependence Plot for frauddata Fe
−0.07
−0.04
−0.01
0.0 0.2 0.4 0.6 0.8 1.0
Dependence Plot for frauddata Fe
−0.05
0.05
0.0 0.2 0.4 0.6 0.8 1.0
Dependence Plot for frauddata Fe
−0.08
−0.04
0.0 0.2 0.4 0.6 0.8 1.0
Dependence Plot for frauddata Fe
−0.05
0.05
0.0 0.2 0.4 0.6 0.8 1.0
Dependence Plot for frauddata Fe
−0.2
0.0
0.2
0.0 0.2 0.4 0.6 0.8 1.0
Dependence Plot for frauddata Fe
−0.040
−0.025
−0.010
0.0 0.2 0.4 0.6 0.8 1.0

Conclusion

METHOD	ACCURACY	SENSITIVITY	SPECIFICITY
LDA, cv	79.53	80.97	78.10
LDA, test data	79.96	80.95	78.96
Logistic, cv	79.67	80.14	79.21
Logistic, test data	79.94	80.53	79.33
k-nearest, cv	60.40	63.64	57.18
k-nearest, test data	59.90	61.57	58.20
class. tree, cv	77.83	76.08	79.56
class. tree, test	75.92	75.52	76.33
Random Forest, cv	80.01	80.40	79.63
Random Forest, test	80.29	80.30	80.27
Adaboost, cv	79.71	80.06	79.36
Adaboost, test	79.61	79.20	80.04
GBM untuned, cv	81.18	81.33	81.03
GBM untuned, test	81.17	82.05	80.27
GBM tuned, cv	80.98	80.78	81.17
GBM tuned, test	**85.10**	84.72	85.49

Common classifiers are capable of making strong predictions on whether or not fraud exists based on the input features of the Base.csv file. However, these models remain insufficient alone. An accuracy rate of 85.10% is not good enough alone. Fraud cases should be reviewed by a human being in addition to being run through a classifying model.

A tuned Gradient Boosting Machine performed the best of all models tested against the data, with an accuracy of 85.10%, sensitivity of 84.72%, and specificity of 85.49% on the testing data. Analysts can use this to assist them in determining cases of fraud. Additionally, analysts should look at the most prominent variables in terms of importance, such as housing_status, current_address_months_count, and name_email_similarity to see if they match up with similar fraud cases.