Code

Credit Card Fraud Machine Learning Analysis

Yongheng Zan

Part 1: Introduction

Code

Credit card fraud is an inclusive term for fraud committed using a payment card, such as a credit card or debit card. The purpose may be to obtain goods or services or to make payment to another account, which is controlled by a criminal(https://en.wikipedia.org/wiki/Credit_card_fraud#Artificial_and_Computational_intelligence[8] (https://en.wikipedia.org/wiki/Credit_card_fraud#Artificial_and_Computational_intelligence%5B8%5D)). More important, as a victim, my cards have been used many times by frauds. So I'm going to construct a machine learning model to detect fraud. I will use Logistic Regression, Nearly Neighbors,Decision tree, Bagging, RandomForest, and Boosted tree to achieve this project.

What is Credit Card Fraud?

Credit card fraud is a form of identity theft that involves an unauthorized taking of another's credit card information for the purpose of charging purchases to the account or removing funds from it.

For detailed introduction, please watch this short video:

Code

An overview of dataset

The dataset contains transactions made by credit cards in September 2013 by European cardholders. This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. (https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud (https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud))

Loading Data and Packages

Loading Data

Hide

```
# read in the data
raw_data <- read.csv("creditcard.csv")
head(raw_data)
```

```
##   Time         V1          V2        V3         V4          V5          V6
## 1    0 -1.3598071 -0.07278117 2.5363467  1.3781552 -0.33832077  0.46238778
## 2    0  1.1918571  0.26615071 0.1664801  0.4481541  0.06001765 -0.08236081
## 3    1 -1.3583541 -1.34016307 1.7732093  0.3797796 -0.50319813  1.80049938
## 4    1 -0.9662717 -0.18522601 1.7929933 -0.8632913 -0.01030888  1.24720317
## 5    2 -1.1582331  0.87773675 1.5487178  0.4030339 -0.40719338  0.09592146
## 6    2 -0.4259659  0.96052304 1.1411093 -0.1682521  0.42098688 -0.02972755
##            V7          V8         V9         V10        V11         V12
## 1  0.23959855  0.09869790  0.3637870  0.09079417 -0.5515995 -0.61780086
## 2 -0.07880298  0.08510165 -0.2554251 -0.16697441  1.6127267  1.06523531
## 3  0.79146096  0.24767579 -1.5146543  0.20764287  0.6245015  0.06608369
## 4  0.23760894  0.37743587 -1.3870241 -0.05495192 -0.2264873  0.17822823
## 5  0.59294075 -0.27053268  0.8177393  0.75307443 -0.8228429  0.53819555
## 6  0.47620095  0.26031433 -0.5686714 -0.37140720  1.3412620  0.35989384
##          V13        V14        V15        V16         V17         V18
## 1 -0.9913898 -0.3111694  1.4681770 -0.4704005  0.20797124  0.02579058
## 2  0.4890950 -0.1437723  0.6355581  0.4639170 -0.11480466 -0.18336127
## 3  0.7172927 -0.1659459  2.3458649 -2.8900832  1.10996938 -0.12135931
## 4  0.5077569 -0.2879237 -0.6314181 -1.0596472 -0.68409279  1.96577500
## 5  1.3458516 -1.1196698  0.1751211 -0.4514492 -0.23703324 -0.03819479
## 6 -0.3580907 -0.1371337  0.5176168  0.4017259 -0.05813282  0.06865315
##           V19         V20          V21          V22         V23         V24
## 1  0.40399296  0.25141210 -0.018306778  0.277837576 -0.11047391  0.06692807
## 2 -0.14578304 -0.06908314 -0.225775248 -0.638671953  0.10128802 -0.33984648
## 3 -2.26185710  0.52497973  0.247998153  0.771679402  0.90941226 -0.68928096
## 4 -1.23262197 -0.20803778 -0.108300452  0.005273597 -0.19032052 -1.17557533
## 5  0.80348692  0.40854236 -0.009430697  0.798278495 -0.13745808  0.14126698
## 6 -0.03319379  0.08496767 -0.208253515 -0.559824796 -0.02639767 -0.37142658
##          V25        V26          V27         V28 Amount Class
## 1  0.1285394 -0.1891148  0.133558377 -0.02105305 149.62     0
## 2  0.1671704  0.1258945 -0.008983099  0.01472417   2.69     0
## 3 -0.3276418 -0.1390966 -0.055352794 -0.05975184 378.66     0
## 4  0.6473760 -0.2219288  0.062722849  0.06145763 123.50     0
## 5 -0.2060096  0.5022922  0.219422230  0.21515315  69.99     0
## 6 -0.2327938  0.1059148  0.253844225  0.08108026   3.67     0
```

Check the dimension

Hide

```
dimension <- dim(raw_data)
dimension
```

```
## [1] 284807     31
```

This dataset includes 31 columns and 284807 observations. There are 1 response variable and 30 predictor variables. And 30 of them are numeric and 1 of them is binary. Feature ‘Class’ is the response variable and it takes value 1 in case of fraud and 0 otherwise. Detail see codebook.

Loading Packages

Part 2: Data Cleaning

Clean name

Hide

```
ccard_data <- raw_data %>%
  clean_names()
```

Convert class to factor

Hide

```
ccard_data <- ccard_data %>%
  mutate(class = factor(class, levels = c("1", "0"))) %>%
  select(-time,)
```

Check missing value

Hide

```
sum(is.na(ccard_data))
```

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

Summary data

Hide

```
summary(ccard_data$amount)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##     0.00     5.60    22.00    88.35    77.17 25691.16
```

Hide

```
# check variance again
var(ccard_data$amount)
```

```
## [1] 62560.07
```

Scale the amount

As we can see, the amount has huge variance so we are going to apply `scale()` function to remove extreme value that might interfere with the functioning of our model.

Hide

```
ccard_data$amount <- scale(ccard_data$amount)
head(ccard_data)
```

```
##           v1          v2        v3         v4          v5          v6
## 1 -1.3598071 -0.07278117 2.5363467  1.3781552 -0.33832077  0.46238778
## 2  1.1918571  0.26615071 0.1664801  0.4481541  0.06001765 -0.08236081
## 3 -1.3583541 -1.34016307 1.7732093  0.3797796 -0.50319813  1.80049938
## 4 -0.9662717 -0.18522601 1.7929933 -0.8632913 -0.01030888  1.24720317
## 5 -1.1582331  0.87773675 1.5487178  0.4030339 -0.40719338  0.09592146
## 6 -0.4259659  0.96052304 1.1411093 -0.1682521  0.42098688 -0.02972755
##            v7          v8         v9         v10        v11         v12
## 1  0.23959855  0.09869790  0.3637870  0.09079417 -0.5515995 -0.61780086
## 2 -0.07880298  0.08510165 -0.2554251 -0.16697441  1.6127267  1.06523531
## 3  0.79146096  0.24767579 -1.5146543  0.20764287  0.6245015  0.06608369
## 4  0.23760894  0.37743587 -1.3870241 -0.05495192 -0.2264873  0.17822823
## 5  0.59294075 -0.27053268  0.8177393  0.75307443 -0.8228429  0.53819555
## 6  0.47620095  0.26031433 -0.5686714 -0.37140720  1.3412620  0.35989384
##          v13        v14        v15        v16         v17         v18
## 1 -0.9913898 -0.3111694  1.4681770 -0.4704005  0.20797124  0.02579058
## 2  0.4890950 -0.1437723  0.6355581  0.4639170 -0.11480466 -0.18336127
## 3  0.7172927 -0.1659459  2.3458649 -2.8900832  1.10996938 -0.12135931
## 4  0.5077569 -0.2879237 -0.6314181 -1.0596472 -0.68409279  1.96577500
## 5  1.3458516 -1.1196698  0.1751211 -0.4514492 -0.23703324 -0.03819479
## 6 -0.3580907 -0.1371337  0.5176168  0.4017259 -0.05813282  0.06865315
##           v19         v20          v21          v22         v23         v24
## 1  0.40399296  0.25141210 -0.018306778  0.277837576 -0.11047391  0.06692807
## 2 -0.14578304 -0.06908314 -0.225775248 -0.638671953  0.10128802 -0.33984648
## 3 -2.26185710  0.52497973  0.247998153  0.771679402  0.90941226 -0.68928096
## 4 -1.23262197 -0.20803778 -0.108300452  0.005273597 -0.19032052 -1.17557533
## 5  0.80348692  0.40854236 -0.009430697  0.798278495 -0.13745808  0.14126698
## 6 -0.03319379  0.08496767 -0.208253515 -0.559824796 -0.02639767 -0.37142658
##          v25        v26          v27         v28      amount class
## 1  0.1285394 -0.1891148  0.133558377 -0.02105305  0.24496383     0
## 2  0.1671704  0.1258945 -0.008983099  0.01472417 -0.34247394     0
## 3 -0.3276418 -0.1390966 -0.055352794 -0.05975184  1.16068389     0
## 4  0.6473760 -0.2219288  0.062722849  0.06145763  0.14053401     0
## 5 -0.2060096  0.5022922  0.219422230  0.21515315 -0.07340321     0
## 6 -0.2327938  0.1059148  0.253844225  0.08108026 -0.33855582     0
```

Since we have done the cleaning part, we can start exploring data now.

Part 3: Data Split

The data was split in a 70% training, 30% testing split. And stratified sampling by *class*.

Hide

```
set.seed(2022)
ccard_split <- initial_split(ccard_data, prop = 0.70, strata = class)
ccard_train <- training(ccard_split)
ccard_test <- testing(ccard_split)

# check dimension
dim(ccard_train)
```

```
## [1] 199364     30
```

Hide

```
dim(ccard_test)
```

```
## [1] 85443    30
```

The training data has 199364 observations and testing has 85443 observations.

Part 4: Exploratory Data Analysis

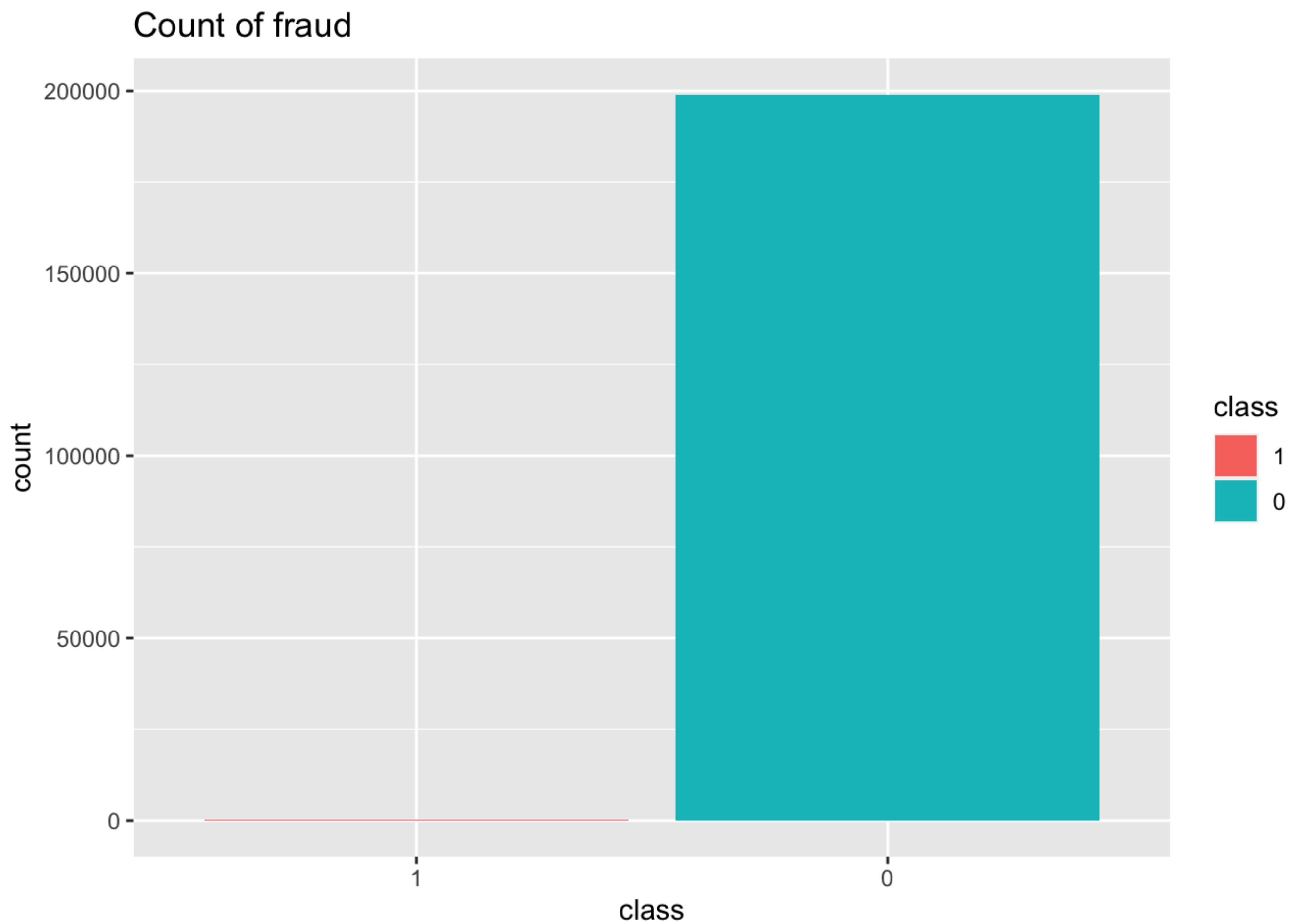
Bar Plot and Table

Hide

```
table(ccard_train$class)
```

```
##
##      1      0
##    358 199006
```

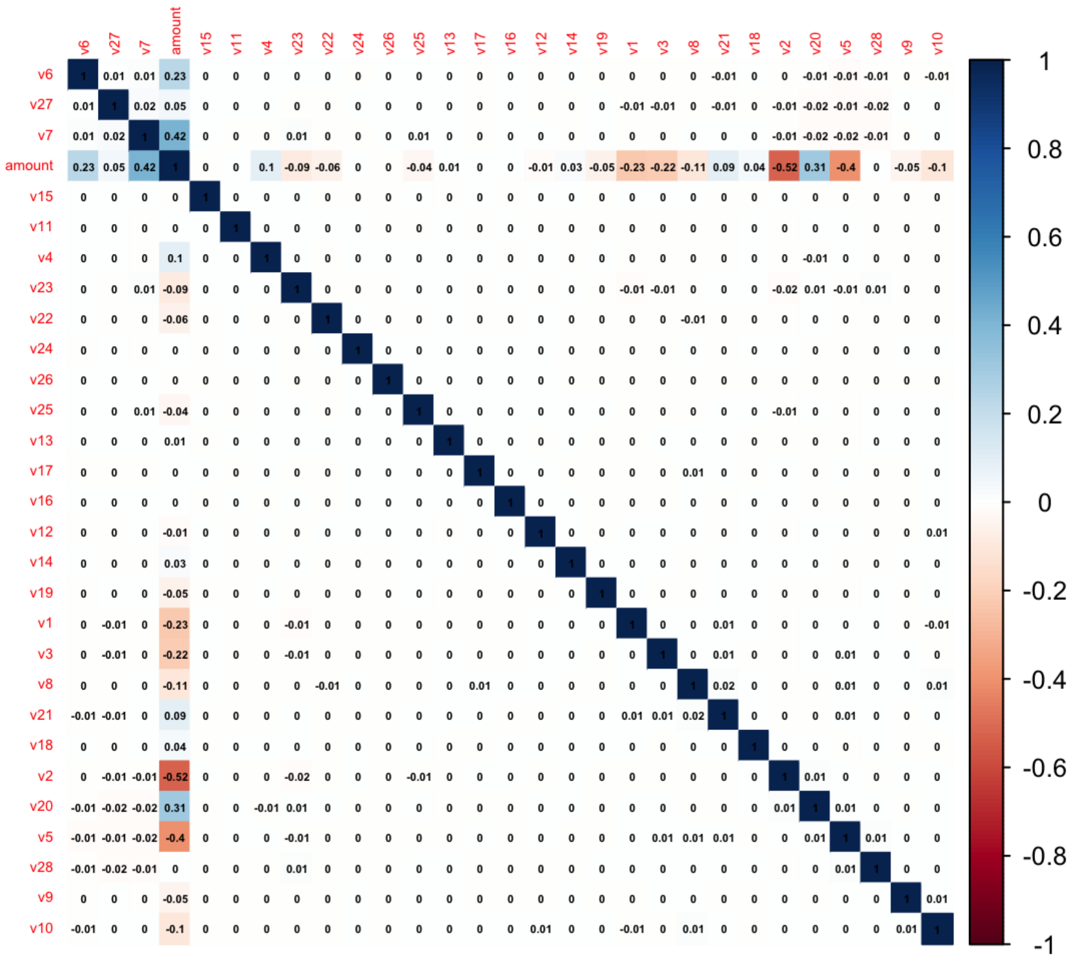
Code



From the table and plot, we can see that the number of card fraud is highly unbalanced.

Correlation matrix

Code



We observe that most of variables are not correlated since those were presented to a Principal Component Analysis (PCA) algorithm, so we do not know if the numbering of the variables reflects the importance of the Principal Components.

Part 5: Model fitting

Create Recipe

Hide

```
ccard_recipe <- recipe(class ~ ., ccard_train) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_normalize(all_predictors())
```

Model 1: Logistic Regression, LDA/QDA

For this part, I'm going to use three different models to find the best one.

Logistic regression model for classification using the *glm* engine.

Hide

```
log_reg <- logistic_reg() %>%
  set_engine("glm") %>%
  set_mode("classification")

log_wkflow <- workflow() %>%
  add_model(log_reg) %>%
  add_recipe(ccard_recipe)

log_fit <- fit(log_wkflow, ccard_train)
```

Linear discriminant analysis model for classification using the *MASS* engine.

Hide

```
lda_mod <- discrim_linear() %>%
  set_mode("classification") %>%
  set_engine("MASS")

lda_wkflow <- workflow() %>%
  add_model(lda_mod) %>%
  add_recipe(ccard_recipe)

lda_fit <- fit(lda_wkflow, ccard_train)
```

Quadratic discriminant analysis model for classification using the *MASS* engine.

Hide

```
qda_mod <- discrim_quad() %>%
  set_mode("classification") %>%
  set_engine("MASS")

qda_wkflow <- workflow() %>%
  add_model(qda_mod) %>%
  add_recipe(ccard_recipe)

qda_fit <- fit(qda_wkflow, ccard_train)
```

Comparing three models

Code

```
## # A tibble: 3 × 2
##   model    .estimate
##   <chr>        <dbl>
## 1 QDA          0.976
## 2 Logistic     0.999
## 3 LDA          0.999
```

Since the LDA model has the highest training accuracy with 0.9994031, so I'm going to apply LDA model to the testing data.

Fitting testing data

Code

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.999
```

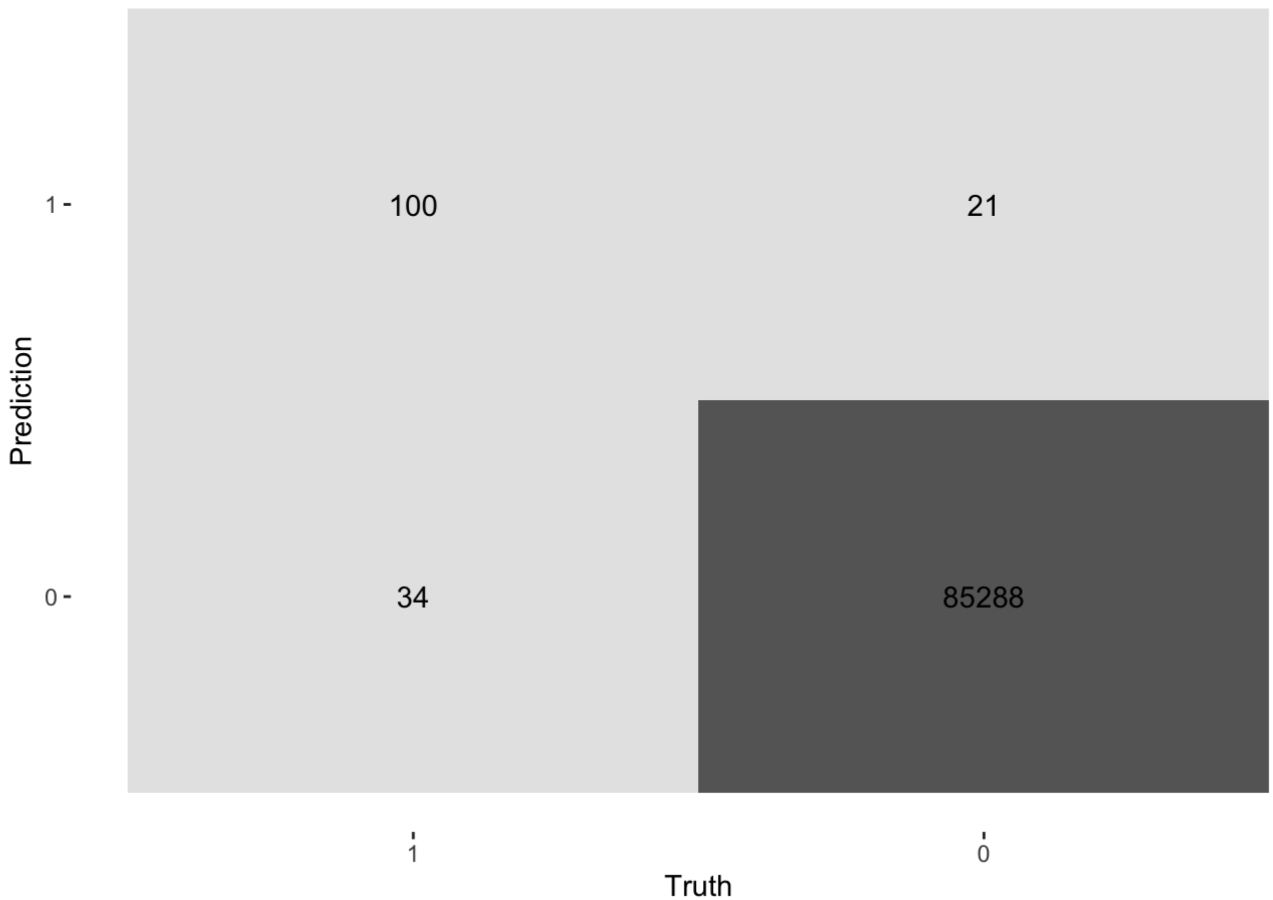
We can see that the LDA model did a great job that has 0.9993563 accuracy.

Confusion matrix and ROC

We also can check it by using visualization:

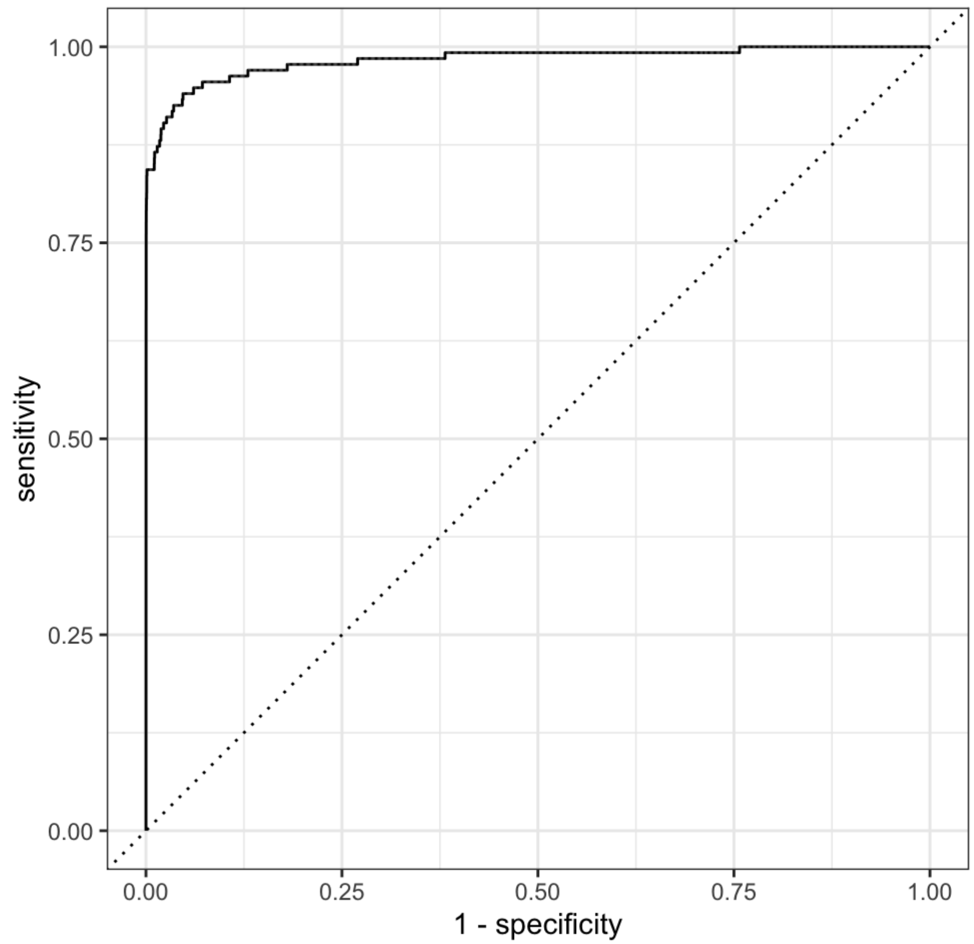
Matrix

Code



ROC

Code



Hide

```r
# Calculate AUC
augment(lda_test, new_data = ccard_test) %>%
  roc_auc(class, .pred_1)
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.983
```

The confusion matrix and ROC have a good performance which also verify our model's accuracy. We have successful predicted 100 of 134 observations from the matrix and the curve almost reach the left-top corner.

Model 2: Nearest Neighbors

Then, we start using the Nearest Neighbor model. We begin at Folding the training data. Use k-fold cross-validation, with k=5.

Hide

```
ccard_fold <- vfold_cv(ccard_train, v = 5, strata = class)
```

Set up

Hide

```
knn_model <- nearest_neighbor(neighbors = tune(),
            mode = "classification") %>%
            set_engine("kknn")

knn_workflow <- workflow() %>%
  add_model(knn_model) %>%
  add_recipe(ccard_recipe)

# set-up tuning grid
knn_params <- parameters(knn_model)

# define grid
knn_grid <- grid_regular(knn_params, levels = 2)
```

Tune the model

Hide

```
knn_tune <- knn_workflow %>%
  tune_grid(resamples = ccard_fold,
            grid = knn_grid)
```

Hide

```
arrange(collect_metrics(knn_tune),desc(mean))
```

```
## # A tibble: 4 × 7
##   neighbors .metric  .estimator  mean     n   std_err .config
##       <int> <chr>    <chr>      <dbl> <int>     <dbl> <chr>
## 1         1 accuracy binary     0.999     5 0.0000639 Preprocessor1_Model1
## 2        15 accuracy binary     0.999     5 0.0000670 Preprocessor1_Model2
## 3        15 roc_auc  binary     0.931     5 0.0130    Preprocessor1_Model2
## 4         1 roc_auc  binary     0.901     5 0.0144    Preprocessor1_Model1
```

Fit the nearest model

We using the best parameter to fit the model.

Hide

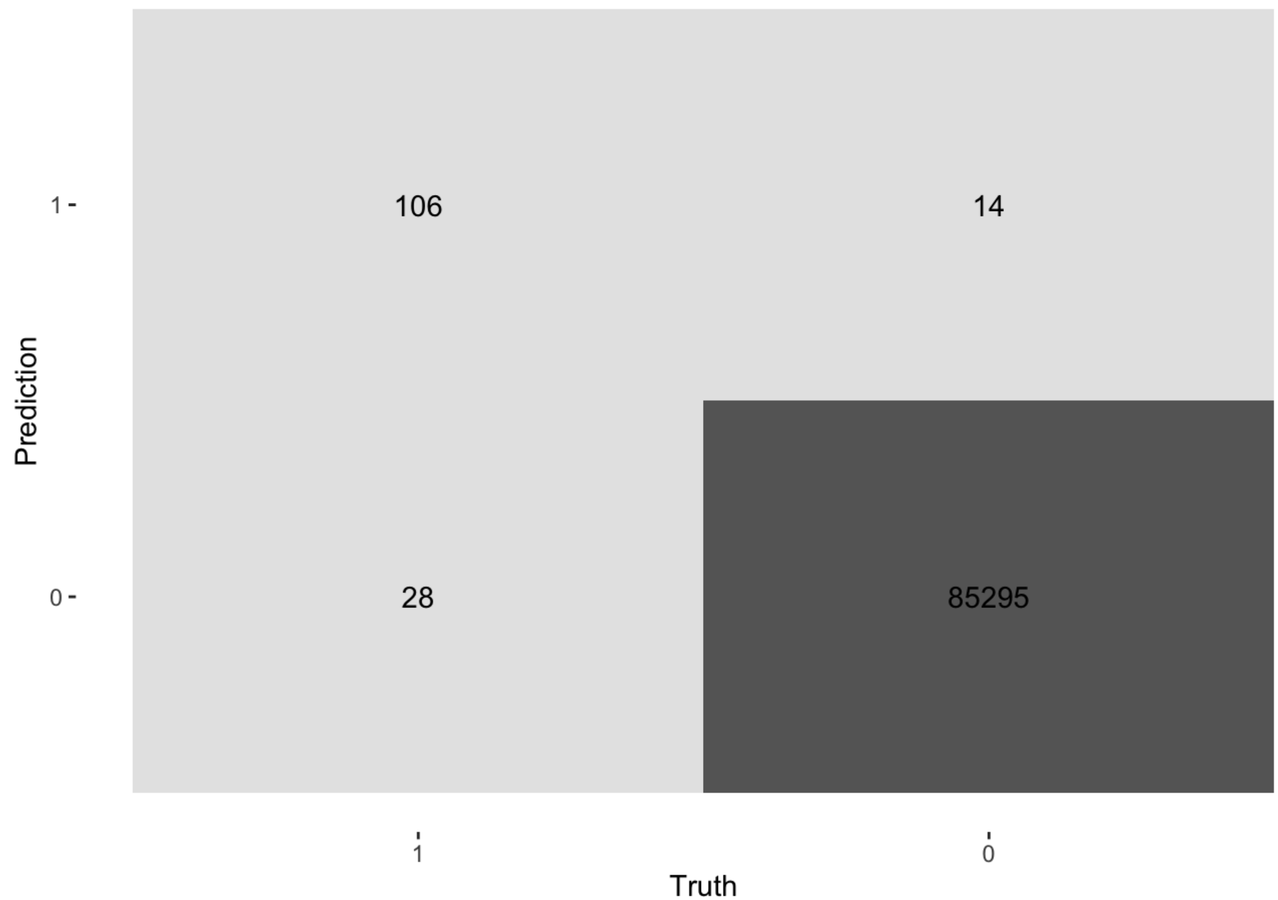
```
best_complexity <- select_best(knn_tune, metric = "roc_auc")
ccard_final <- finalize_workflow(knn_workflow, best_complexity)
knn_fit <- fit(ccard_final,data = ccard_train)

augment(knn_fit, new_data = ccard_test) %>%
  accuracy(truth = class, estimate = .pred_class)
```

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary          1.00
```

Heat map

We can use the heat map to clearly see the prediction.

Code



We can see the Nearest Neighbors have high accuracy with 0.9995084 and have successful predicted 106 of 134 observations from the matrix.

AUC

Hide

```
# Calculate AUC
augment(knn_fit, new_data = ccard_test) %>%
  roc_auc(class, .pred_1)
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.921
```

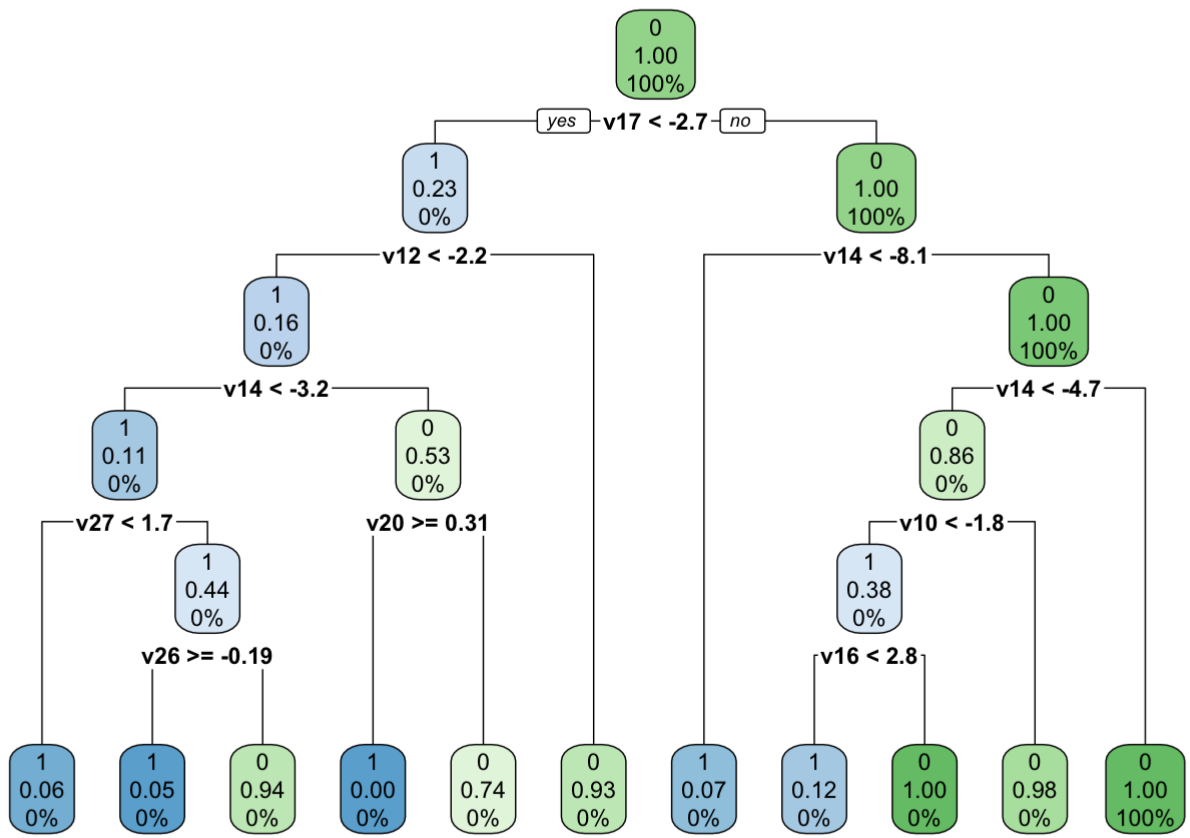
Model 3: Decision tree

Then, I’m going to set up decision tree model.

Set up and `rpart.plot()`

Hide

```
# set up model and workflow
tree_spec <- decision_tree() %>%
  set_engine("rpart")

class_tree_spec <- tree_spec %>%
  set_mode("classification")

class_tree_fit <- class_tree_spec %>%
  fit(class ~ ., data = ccard_train)

class_tree_fit %>%
  extract_fit_engine() %>%
  rpart.plot(roundint=FALSE)
```



Fit decision tree

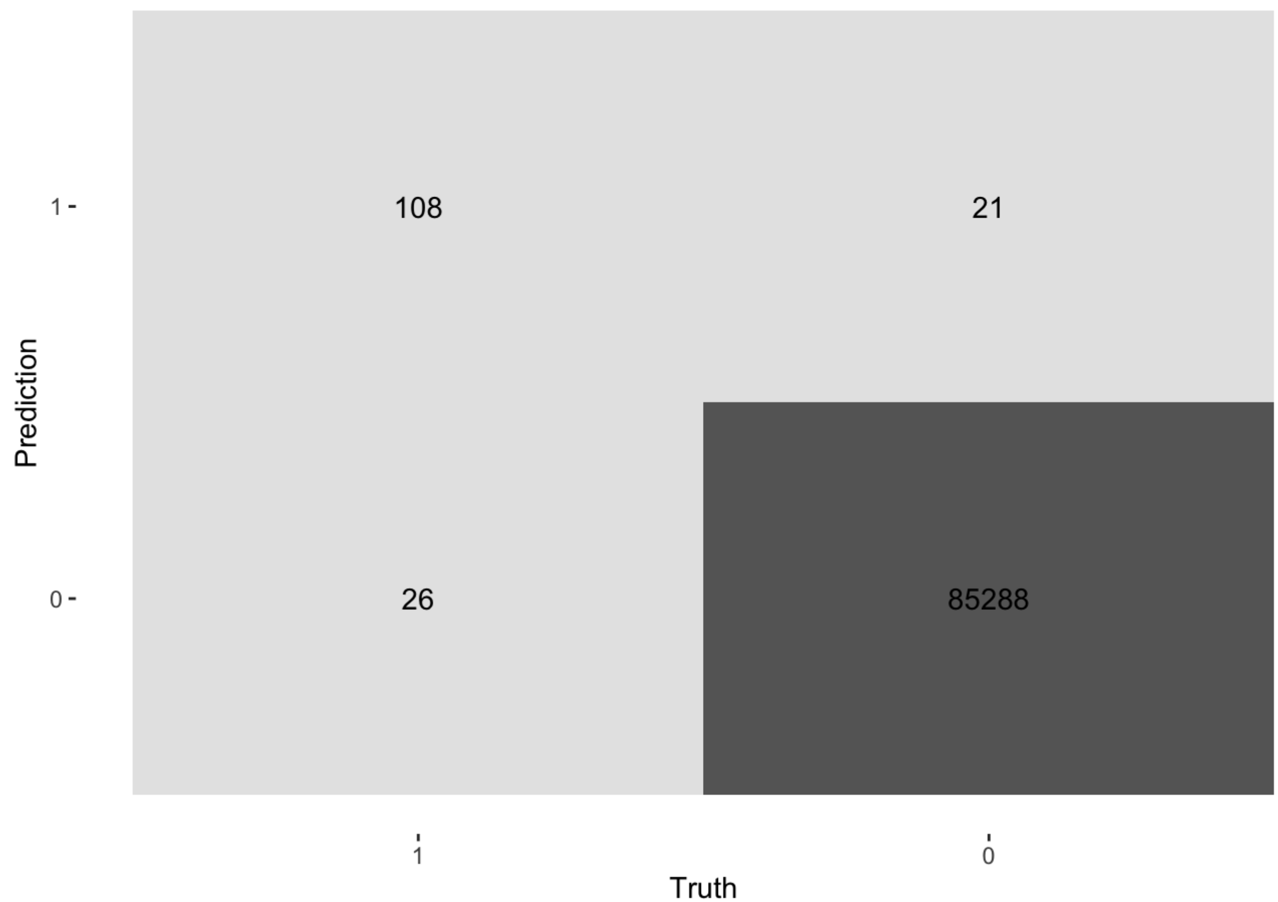
Code

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.999
```

Confusion matrix

Let us take a look at the confusion matrix:

Code



Hide

```
# Calculate AUC
augment(class_tree_fit, new_data = ccard_test) %>%
  roc_auc(class, .pred_1)
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.910
```

We can see decision tree have high accuracy with 0.9994499 and have successful predicted 108 of 134 observations from the matrix.

Model 4: Random forest

Next, I'm going to set up a random forest model and workflow.

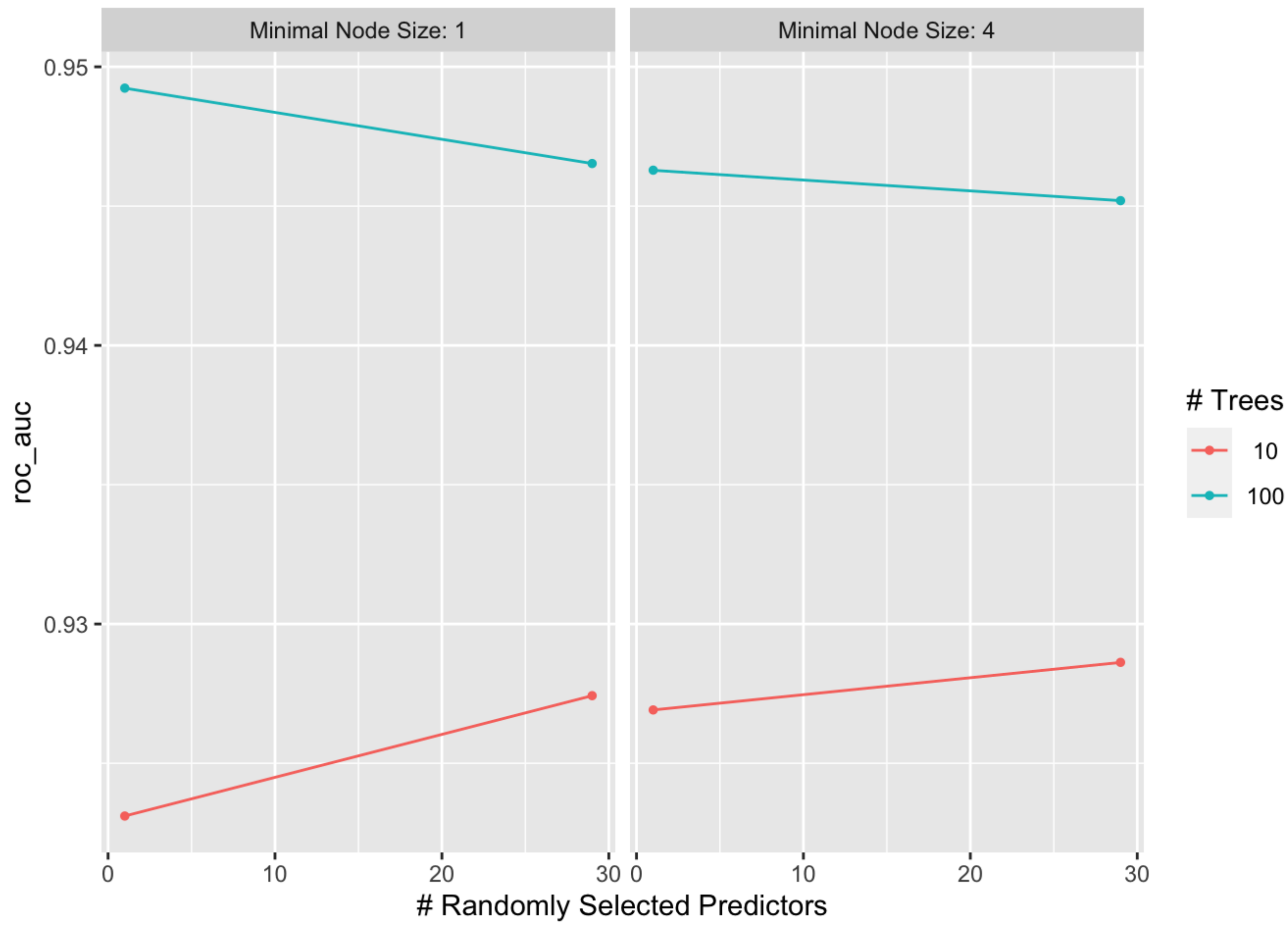
Set up

Hide

```
rf_spec <- rand_forest(mtry = tune(),trees = tune(), min_n = tune()) %>%
  set_engine("randomForest", importance = TRUE) %>%
  set_mode("classification")

rf_wf <- workflow() %>%
  add_model(rf_spec) %>%
  add_recipe(ccard_recipe)

param_grid_rf <- grid_regular(mtry(range = c(1, 29)), # mtry should not be smaller than 1 or larger than 29, since
we only have 29 predictors.
                           trees(range = c(10, 100)), # Due to we have a huge dataset, I chose 100 trees as maximu
m
                           min_n(range = c(1, 4)),
                           levels = 2)
```

Tune the model and print an `autoplot()` of the results.

Code



Code

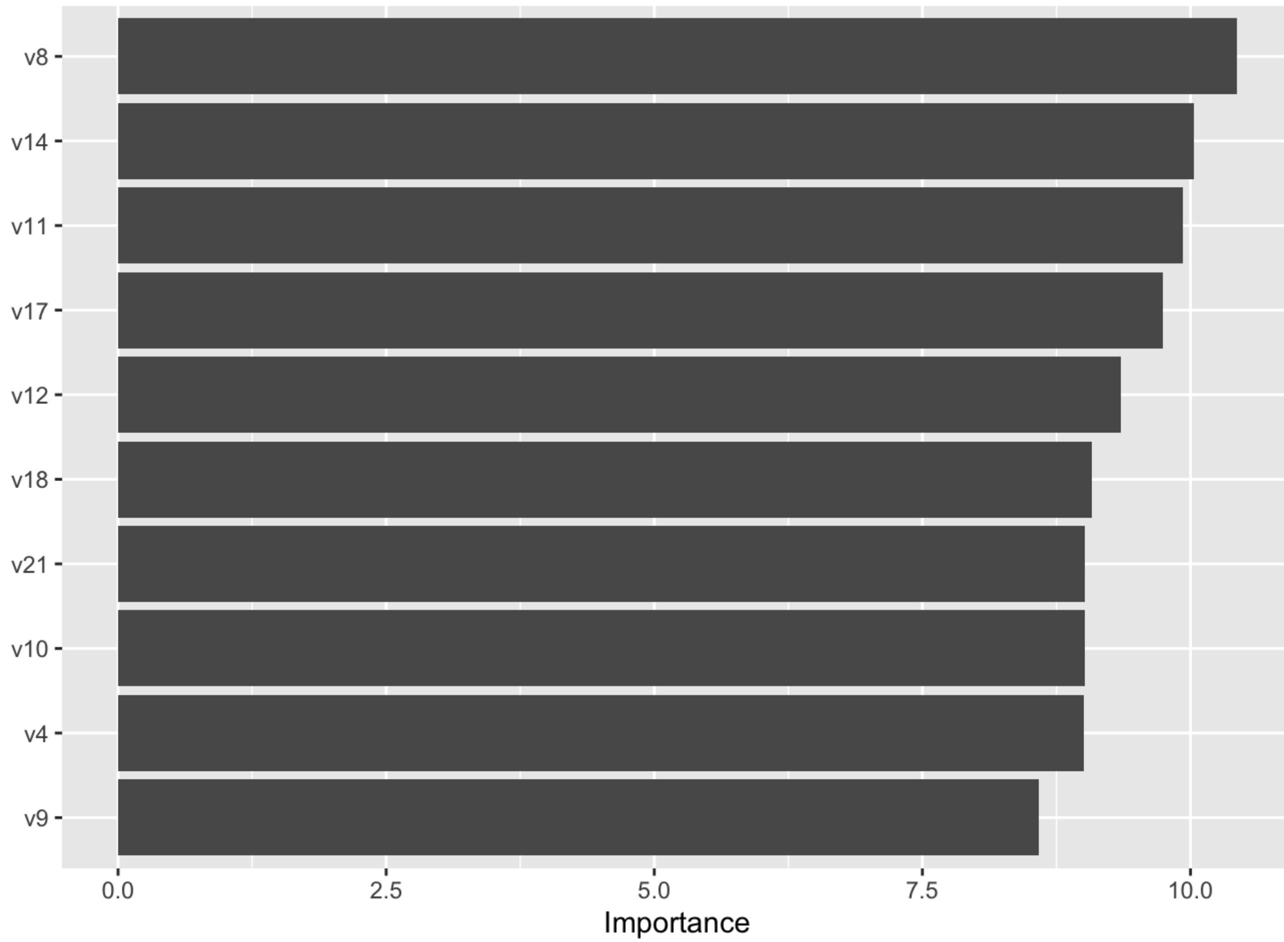
```
## # A tibble: 8 × 9
##    mtry trees min_n .metric .estimator  mean     n std_err .config
##   <int> <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1     1   100     1 roc_auc binary     0.949     5 0.00975 Preprocessor1_Model3
## 2    29   100     1 roc_auc binary     0.947     5 0.0106  Preprocessor1_Model4
## 3     1   100     4 roc_auc binary     0.946     5 0.0123  Preprocessor1_Model7
## 4    29   100     4 roc_auc binary     0.945     5 0.00984 Preprocessor1_Model8
## 5    29    10     4 roc_auc binary     0.929     5 0.00386 Preprocessor1_Model6
## 6    29    10     1 roc_auc binary     0.927     5 0.00860 Preprocessor1_Model2
## 7     1    10     4 roc_auc binary     0.927     5 0.00897 Preprocessor1_Model5
## 8     1    10     1 roc_auc binary     0.923     5 0.00842 Preprocessor1_Model1
```

In general, The more trees we add the better performance we have.

Important plot

Create a variable importance plot, using `vip()` , with the best-performing random forest model fit on the training set.

Code



We can see the variable v9 is the most important. However, all variables are play a important role in this model.

Fit random forest model

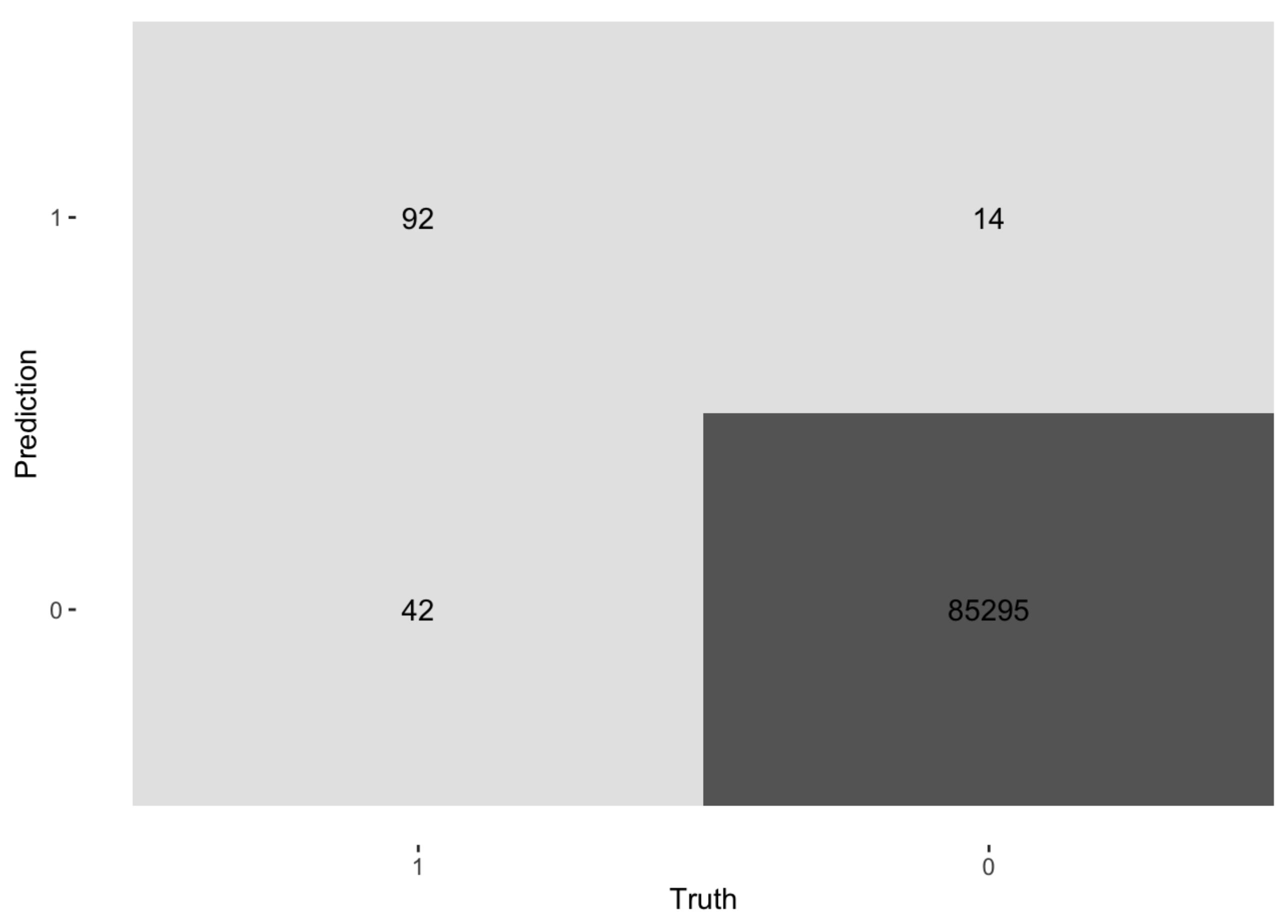
Code

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.999
```

The Random forest model has 0.9993329 accuracy.

Heat map

Code



We can see desision tree have high accuracy with 0.9993329 and have successful predicted 90 of 134 observations from the matrix.

AUC

Hide

```
# Calculate AUC
augment(rf_fit, new_data = ccard_test) %>%
  roc_auc(class, .pred_1)
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.953
```

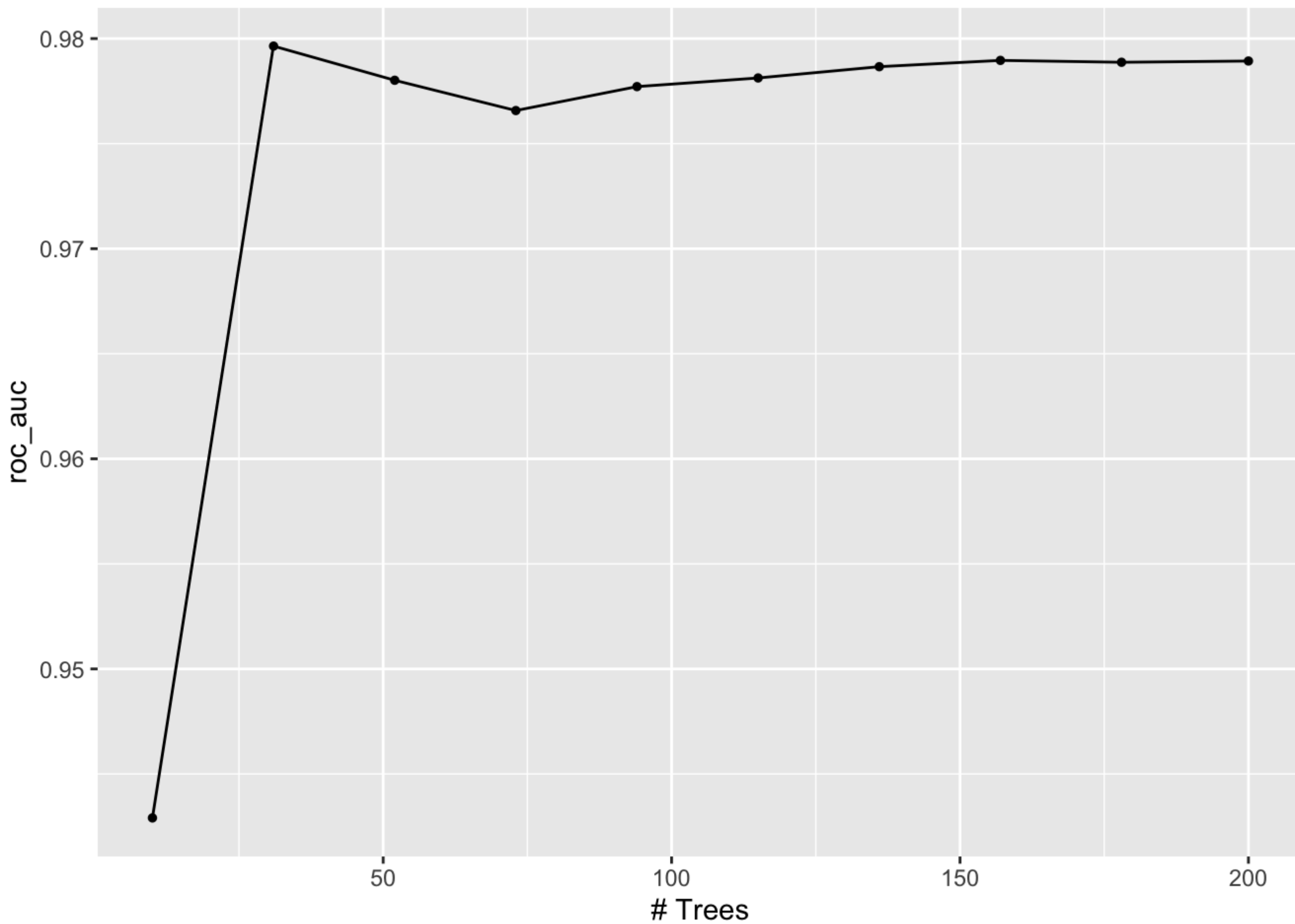
Model 5: Boost tree

At last, I’m going to using boost tree model.

Set up

Hide

```
boost_tree_spec <- boost_tree(trees = tune()) %>%
  set_engine("xgboost") %>%
  set_mode("classification")

boost_tree_grid <- grid_regular(trees(c(10,200)),levels = 10)

boost_tree_wf <- workflow() %>%
  add_model(boost_tree_spec) %>%
  add_recipe(ccard_recipe)

boost_tune_res <- tune_grid(
  boost_tree_wf,
  resamples = ccard_fold,
  grid = boost_tree_grid,
  metrics = metric_set(roc_auc),
)

autoplot(boost_tune_res)
```



The roc_auc keep increasing and reach the peak around 0.98 with 31 tress.

Select best tree

Hide

```
best_boost_tree <- select_best(boost_tune_res)
boost_tree_final <- finalize_workflow(boost_tree_wf, best_boost_tree)
boost_tree_final_fit <- fit(boost_tree_final, data = ccard_train)
```

Fit tree

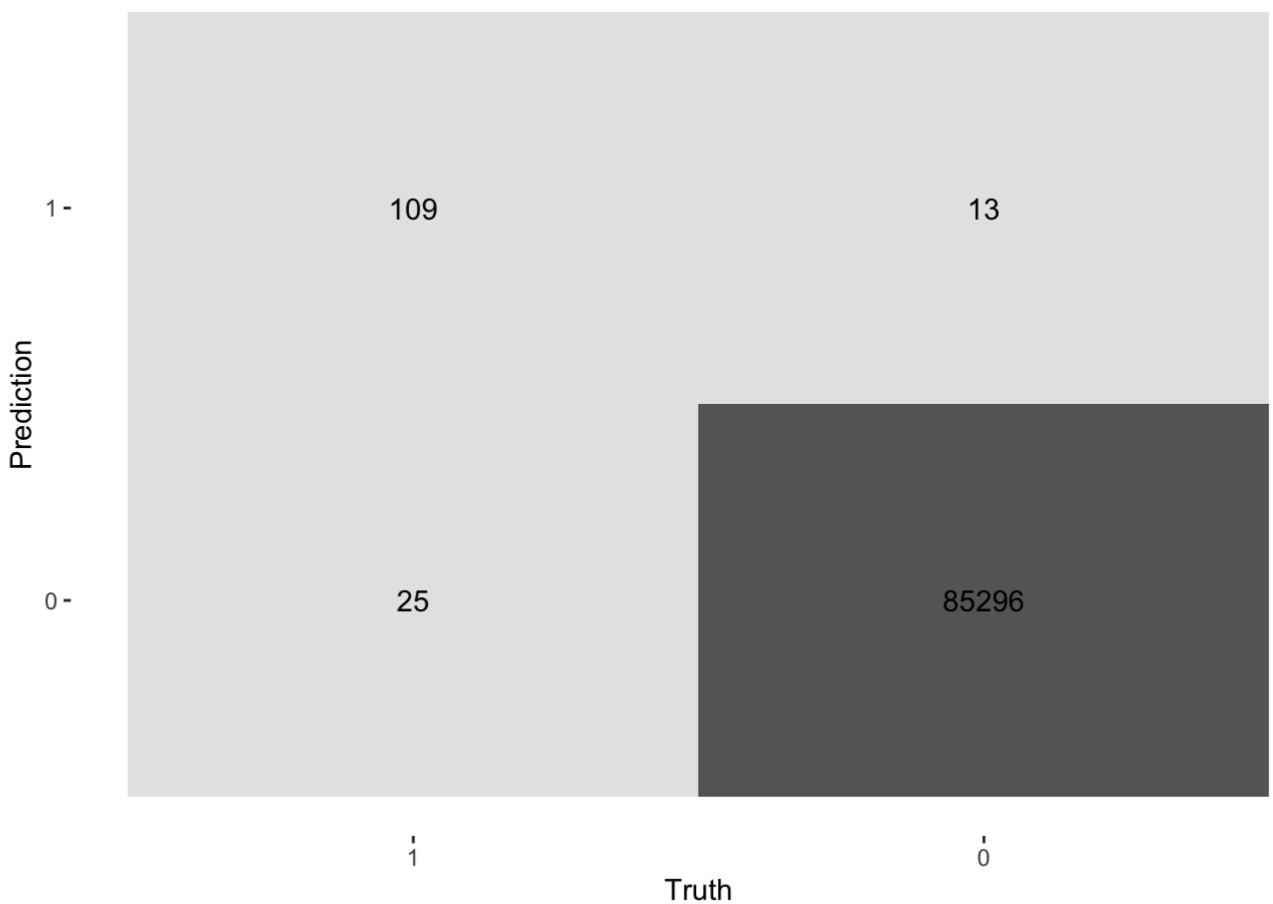
Code

```
## # A tibble: 1 × 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary          1.00
```

The best boost tree model achieve 0.9995553 accuracy!

Heat map

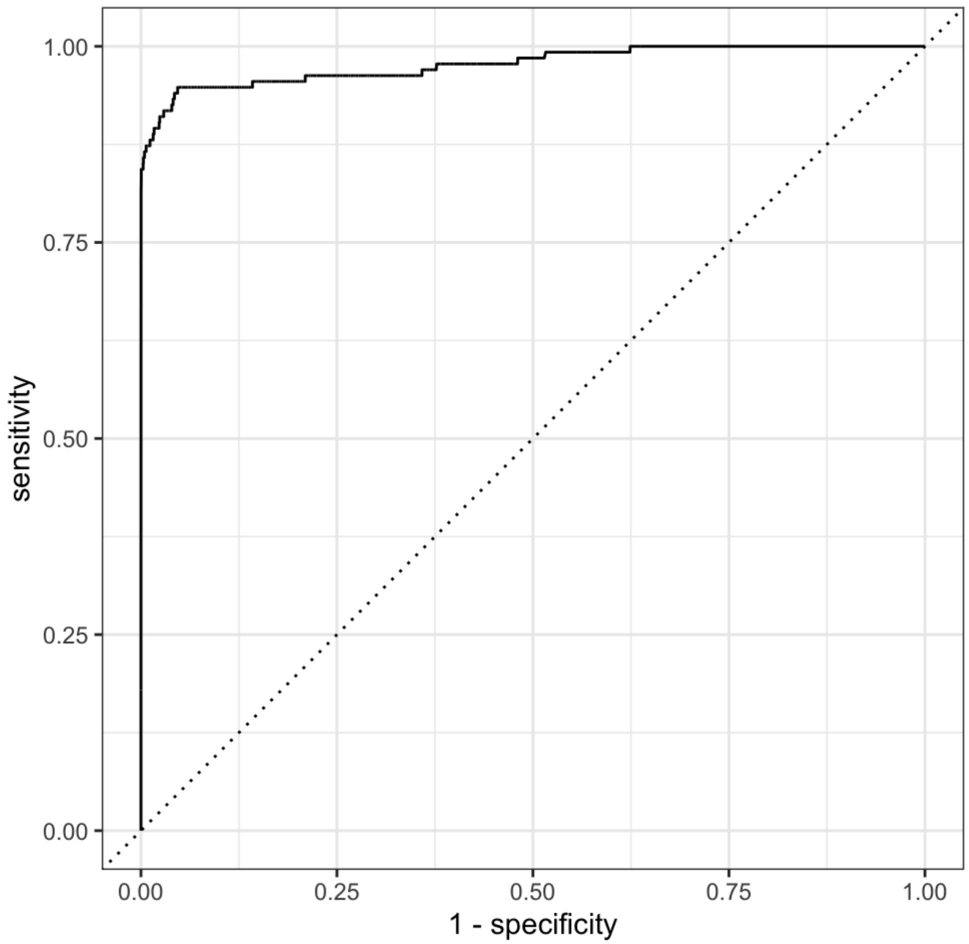
Code



We can see desision tree have high accuracy with 0.9995553 and have successful predicted 109 of 134 observations from the matrix.

ROC

Code



AUC

Hide

```
# Calculate AUC
augment(boost_tree_final_fit, new_data = ccard_test) %>%
  roc_auc(class, .pred_1)
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.977
```

Part 6: Conclusion

In this project we have tried to show mainly 4 type methods to dealing this unbalanced datasets. The performance display below:

Method/Model	Accuracy	AUC
LDA	0.9993563	0.9828842
K Nearest neighbor	0.9995084	0.9212963
Decision tree	0.9994499	0.9102373
Random Forest	0.9993329	0.930354
Boosted tree	0.9995553	0.977474

From the heat map, we can see the transaction we predicted dataset where the instances of fraudulent case is few compared to the instances of normal transactions. We have a better accuracy after using resample technology. The best score of 0.999553 was achieved using an Boost tree model though other models performed well too. It is likely that by further tuning the BOOST TREE model parameters we can achieve even better performance.