F.P. MODEL 1 RF

THEDION V. DIAM JR.

2022-12-15

LOAD PACKAGES

```
# Helper packages
library(dplyr)    # for data wrangling
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(ggplot2)  # for awesome graphics
library(tidyverse)
```

```
## ── Attaching packages
## ─────────────────────────────────────
## tidyverse 1.3.2 ──
```

```
## ✓ tibble  3.1.8     ✓ purrr   0.3.5
## ✓ tidyr   1.2.1     ✓ stringr 1.4.1
## ✓ readr   2.1.3     ✓ forcats 0.5.2
## ── Conflicts ──────────────────────────────────────
tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
```

```
library(rsample)
library(ROCR)
library(pROC)
```

```
## Type 'citation("pROC")' for a citation.
##
## Attaching package: 'pROC'
##
## The following objects are masked from 'package:stats':
```

```
##
##     cov, smooth, var

# Modeling packages
library(ranger)   # a c++ implementation of random forest
library(h2o)      # a java-based implementation of random forest

##
## ----------------------------------------------------------------------
##
## Your next step is to start H2O:
##     > h2o.init()
##
## For H2O package documentation, ask for help:
##     > ??h2o
##
## After starting H2O, you can use the Web UI at http://localhost:54321
## For more information visit https://docs.h2o.ai
##
## ----------------------------------------------------------------------
##
##
## Attaching package: 'h2o'
##
## The following object is masked from 'package:pROC':
##
##     var
##
## The following objects are masked from 'package:stats':
##
##     cor, sd, var
##
## The following objects are masked from 'package:base':
##
##     %*%, %in%, &&, ||, apply, as.factor, as.numeric, colnames,
##     colnames<-, ifelse, is.character, is.factor, is.numeric, log,
##     log10, log1p, log2, round, signif, trunc

h2o.init()

##  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         33 minutes 36 seconds
##     H2O cluster timezone:       Asia/Taipei
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.38.0.1
##     H2O cluster version age:    2 months and 27 days
##     H2O cluster name:           H2O_started_from_R_REY_hvw787
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   3.86 GB
```

```
##     H2O cluster total cores:    16
##     H2O cluster allowed cores:  16
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     R Version:                  R version 4.2.2 (2022-10-31 ucrt)
```

Load Data Sets

The data contains 197 rows and 431 columns with *Failure.binary* binary output.

```
library(readr)
rawd <- read_csv("D:/DIAM/FP-DATA.csv")

## Rows: 197 Columns: 431
## ── Column specification
──────────────────────────────────────────────────────
## Delimiter: ","
## chr   (1): Institution
## dbl (430): Failure.binary, Failure, Entropy_cooc.W.ADC, GLNU_align.H.PET,
Mi...
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this
message.
```

#=============== Reprocessing the Raw Data ================================#

```
library(tidyverse)

## ── Attaching packages ──────────────────────────────────── tidyverse
1.3.2 ──
## ✓ ggplot2 3.4.0      ✓ dplyr   1.0.10
## ✓ tibble  3.1.8      ✓ stringr 1.4.1
## ✓ tidyr   1.2.1      ✓ forcats 0.5.2
## ✓ purrr   0.3.5
## ── Conflicts ─────────────────────────────────────────
tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()

library(bestNormalize)
```

Check for null and missing values

Using *anyNA()* function, We can determine if any missing values in our data.

```
anyNA(rawd)

## [1] FALSE

#The result shows either *True* or *False*. If True, omit the missing values
using *na.omit()*

#[1] FALSE

#Thus, our data has no missing values.
```

Check for Normality of the Data

We used *Shapiro-Wilk's Test* to check the normality of the data.

```
rd <- rawd%>%select_if(is.numeric)
rd <- rd[,-1]
test <- apply(rd,2,function(x){shapiro.test(x)})
```

To have the list of p-value of all variables, the *unlist()* function is used and convert a list to vector.

```
pvalue_list <- unlist(lapply(test, function(x) x$p.value))

sum(pvalue_list<0.05)  # not normally distributed

## [1] 428

sum(pvalue_list>0.05)  # normally distributed

## [1] 1

test$Entropy_cooc.W.ADC

##
##  Shapiro-Wilk normality test
##
## data:  x
## W = 0.98903, p-value = 0.135

# [1] 428
# [1] 1

#  Thus, we have 428 variables that are not normally distributed and
Entropy_cooc.W.ADC is normally distributed.
```

We use *orderNorm()* function, the *x.t* is the elements of orderNorm() function transformed original data.Using the *Shapiro-Wilk's Test*

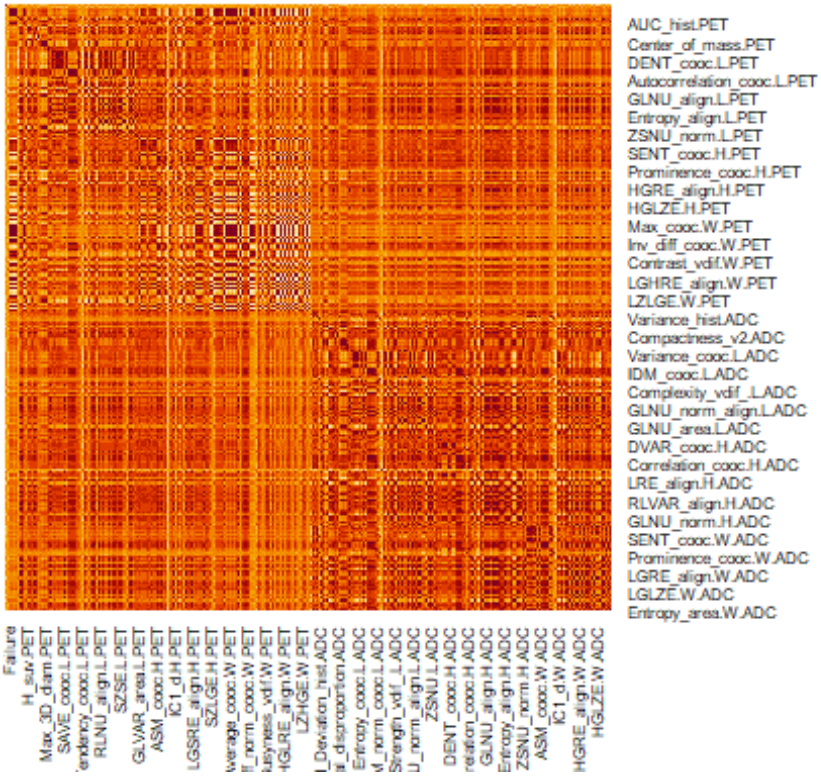
```
TRDrawd=rawd[,c(3,5:length(names(rawd)))]

TRDrawd=apply(TRDrawd,2,orderNorm)
TRDrawd=lapply(TRDrawd, function(x) x$x.t)
TRDrawd=TRDrawd%>%as.data.frame()
test=apply(TRDrawd,2,shapiro.test)
test=unlist(lapply(test, function(x) x$p.value))
```

#Testing Data

```
sum(test <0.05)  # not normally distributed

## [1] 0

sum(test >0.05)  # normally distributed

## [1] 428

#[1] 0
#[1] 428

# Thus, our data is normally distributed.

rawd[,c(3,5:length(names(rawd)))]=TRDrawd
```

Get the correlation of the whole data expect the categorical variables

```
CorMatrix=cor(rawd[,-c(1,2)])
heatmap(CorMatrix,Rowv=NA,Colv=NA,scale="none",revC = T)
```



#Splitting the Data Split the data into training (80%) and testing (20%).

```
rawd$Institution=as.factor(rawd$Institution)
rawd$Failure.binary=as.factor(rawd$Failure.binary)

splitter <- sample(1:nrow(rawd), round(nrow(rawd) * 0.8))
trainND <- rawd[splitter, ]
testND  <- rawd[-splitter, ]
```

The data frame output of data reprocessing will be converted into to “csv”, which will be used for entire project.

Load new Data

```
Final <- read_csv("D:/DIAM/newdat.csv")

## Rows: 197 Columns: 431
## ── Column specification
────────────────────────────────────────────────────────
## Delimiter: ","
## chr   (1): Institution
## dbl (430): Failure.binary, Failure, Entropy_cooc.W.ADC, GLNU_align.H.PET,
Mi...
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this
message.
```

```r
View(Final)

#==================RANDOM FOREST==================#



# make bootstrapping reproducible
set.seed(123)
Final$Failure.binary=as.factor(Final$Failure.binary)
split <- initial_split(Final, strata = "Failure.binary")
traindt <- training(split)
testdt <- testing(split)


# number of features
n_features <- length(setdiff(names(traindt), "Failure.binary"))

# train a default random forest model
randomForest_1 <- ranger(
  Failure.binary ~ .,
  data = traindt,
  mtry = floor(n_features / 3),
  respect.unordered.factors = "order",
  seed = 123
)

# get OOB RMSE
(default_rmse <- sqrt(randomForest_1$prediction.error))

## [1] 0.340068

# create hyperparameter grid
hyper_grid <- expand.grid(
  mtry = floor(n_features * c(.05, .15, .25, .333, .4)),
  min.node.size = c(1, 3, 5, 10),
  replace = c(TRUE, FALSE),
  sample.fraction = c(.5, .63, .8),
  rmse = NA
)

# execute full cartesian grid search
for(i in seq_len(nrow(hyper_grid))) {
  # fit model for ith hyperparameter combination
  fit <- ranger(
    formula         = Failure.binary ~ .,
    data            = traindt,
    num.trees       = n_features * 10,
    mtry            = hyper_grid$mtry[i],
    min.node.size   = hyper_grid$min.node.size[i],
    replace         = hyper_grid$replace[i],
```

```
    sample.fraction = hyper_grid$sample.fraction[i],
    verbose         = FALSE,
    seed            = 123,
    respect.unordered.factors = 'order',
  )
  # export OOB error
  hyper_grid$rmse[i] <- sqrt(fit$prediction.error)
}

# assess top 10 models
hyper_grid %>%
  arrange(rmse) %>%
  mutate(perc_gain = (default_rmse - rmse) / default_rmse * 100) %>%
  head(10)
```

```
##    mtry min.node.size replace sample.fraction      rmse perc_gain
## 1   172             1    TRUE            0.50 0.3299144   2.98575
## 2   172             3    TRUE            0.50 0.3299144   2.98575
## 3   143            10    TRUE            0.50 0.3299144   2.98575
## 4   172            10    TRUE            0.50 0.3299144   2.98575
## 5   172            10   FALSE            0.50 0.3299144   2.98575
## 6   172            10    TRUE            0.63 0.3299144   2.98575
## 7   143             1    TRUE            0.50 0.3400680   0.00000
## 8   143             3    TRUE            0.50 0.3400680   0.00000
## 9   143             5    TRUE            0.50 0.3400680   0.00000
## 10  172             5    TRUE            0.50 0.3400680   0.00000
```

```
h2o.no_progress()
h2o.init(max_mem_size = "5g")
```

```
##  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         34 minutes 22 seconds
##     H2O cluster timezone:       Asia/Taipei
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.38.0.1
##     H2O cluster version age:    2 months and 27 days
##     H2O cluster name:           H2O_started_from_R_REY_hvw787
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   3.86 GB
##     H2O cluster total cores:    16
##     H2O cluster allowed cores:  16
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     R Version:                  R version 4.2.2 (2022-10-31 ucrt)
```

```r
# convert training data to h2o object
train_h2o <- as.h2o(traindt)

# set the response column to Failure.binary
response <- "Failure.binary"

# set the predictor names
predictors <- setdiff(colnames(traindt), response)

h2o_rf1 <- h2o.randomForest(
  x = predictors,
  y = response,
  training_frame = train_h2o,
  ntrees = n_features * 10,
  seed = 123
)
```

```
## Warning in .h2o.processResponseWarnings(res): Dropping bad and constant
columns: [Institution].
```

```r
h2o_rf1
```

```
## Model Details:
## ==============
##
## H2OBinomialModel: drf
## Model ID:  DRF_model_R_1671172895981_33907
## Model Summary:
##   number_of_trees number_of_internal_trees model_size_in_bytes min_depth
## 1            4300                     4300             1129968         4
##   max_depth mean_depth min_leaves max_leaves mean_leaves
## 1        13    6.76721          7         26    16.11860
##
##
## H2OBinomialMetrics: drf
## ** Reported on training data. **
## ** Metrics reported on Out-Of-Bag training samples **
##
## MSE:  0.1335695
## RMSE:  0.3654717
## LogLoss:  0.4232881
## Mean Per-Class Error:  0.1834021
## AUC:  0.8859794
## AUCPR:  0.7688701
## Gini:  0.7719588
## R^2:  0.4048858
##
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal
threshold:
##         0  1    Error     Rate
```

```
## 0      75 22 0.226804   =22/97
## 1       7 43 0.140000    =7/50
## Totals 82 65 0.197279  =29/147
##
## Maximum Metrics: Maximum metrics at their respective thresholds
##                         metric threshold     value idx
## 1                        max f1  0.322825  0.747826  64
## 2                        max f2  0.230769  0.842105  84
## 3                  max f0point5  0.683418  0.760870  21
## 4                  max accuracy  0.436593  0.823129  43
## 5                 max precision  0.683418  0.954545  21
## 6                    max recall  0.132872  1.000000 114
## 7                max specificity  0.872517  0.989691   0
## 8               max absolute_mcc  0.322825  0.604012  64
## 9   max min_per_class_accuracy  0.369259  0.793814  59
## 10 max mean_per_class_accuracy  0.322825  0.816598  64
## 11                      max tns  0.872517 96.000000   0
## 12                      max fns  0.872517 50.000000   0
## 13                      max fps  0.031170 97.000000 146
## 14                      max tps  0.132872 50.000000 114
## 15                      max tnr  0.872517  0.989691   0
## 16                      max fnr  0.872517  1.000000   0
## 17                      max fpr  0.031170  1.000000 146
## 18                      max tpr  0.132872  1.000000 114
##
## Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or
`h2o.gainsLift(<model>, valid=<T/F>, xval=<T/F>)`
```

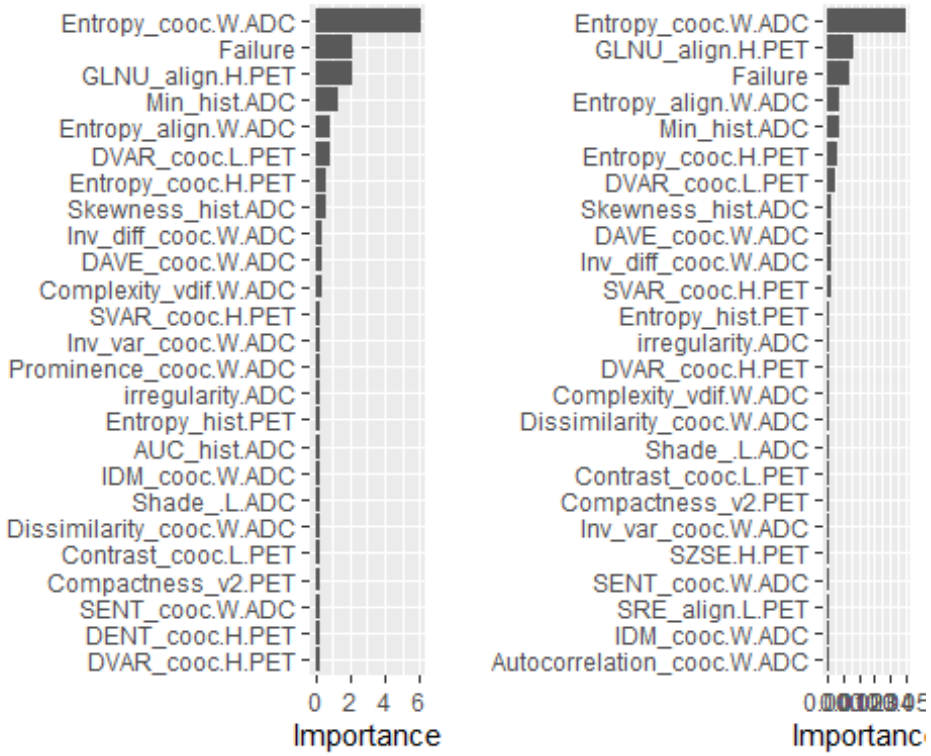
```
# hyperparameter grid
hyper_grid <- list(
  mtries = floor(n_features * c(.05, .15, .25, .333, .4)),
  min_rows = c(1, 3, 5, 10),
  max_depth = c(10, 20, 30),
  sample_rate = c(.55, .632, .70, .80)
)

# random grid search strategy
search_criteria <- list(
  strategy = "RandomDiscrete",
  stopping_metric = "mse",
  stopping_tolerance = 0.001,   # stop if improvement is < 0.1%
  stopping_rounds = 10,         # over the last 10 models
  max_runtime_secs = 60*5      # or stop search after 5 min.
)

# perform grid search
random_grid <- h2o.grid(
  algorithm = "randomForest",
  grid_id = "rf_random_grid",
  x = predictors,
```

```
  y = response,
  training_frame = train_h2o,
  hyper_params = hyper_grid,
  ntrees = n_features * 10,
  seed = 123,
  stopping_metric = "RMSE",
  stopping_rounds = 10,           # stop if last 10 trees added
  stopping_tolerance = 0.005,     # don't improve RMSE by 0.5%
  search_criteria = search_criteria
)

# collect the results and sort by our model performance metric
# of choice
random_grid_perf <- h2o.getGrid(
  grid_id = "rf_random_grid",
  sort_by = "mse",
  decreasing = FALSE
)
random_grid_perf
```

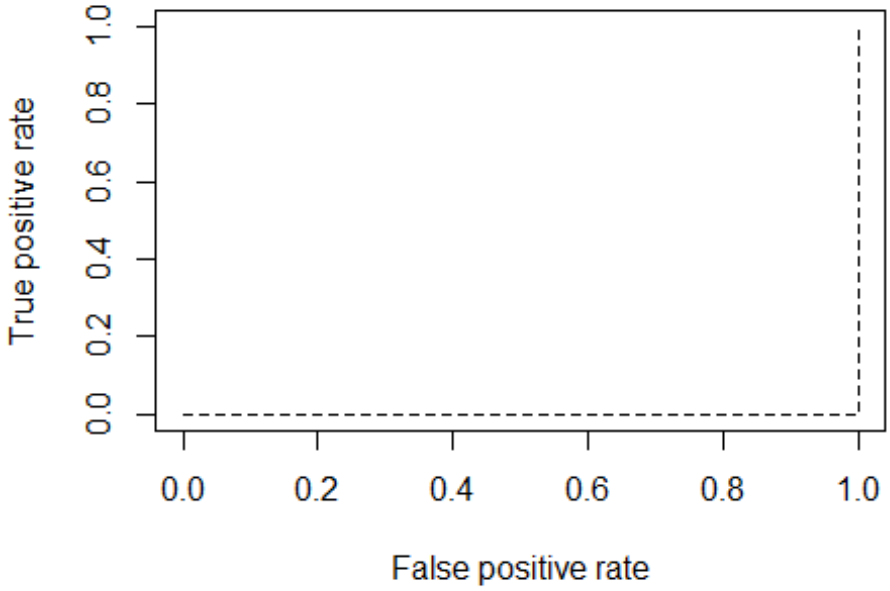
```
## H2O Grid Details
## ================
##
## Grid ID: rf_random_grid
## Used hyper parameters:
##   -  max_depth
##   -  min_rows
##   -  mtries
##   -  sample_rate
## Number of models: 240
## Number of failed models: 0
##
## Hyper-Parameter Search Summary: ordered by increasing mse
##   max_depth min_rows    mtries sample_rate                model_ids
mse
## 1  10.00000  1.00000 143.00000     0.80000 rf_random_grid_model_191
0.08344
## 2  30.00000  1.00000 143.00000     0.80000  rf_random_grid_model_20
0.08344
## 3  20.00000  1.00000 143.00000     0.80000  rf_random_grid_model_82
0.08344
## 4  20.00000  3.00000 143.00000     0.63200 rf_random_grid_model_104
0.08621
## 5  10.00000  3.00000 143.00000     0.63200 rf_random_grid_model_127
0.08621
##
## ---
##     max_depth min_rows   mtries sample_rate                 model_ids
mse
## 235  20.00000 10.00000 21.00000     0.80000 rf_random_grid_model_164
```

```
0.15089
## 236  10.00000 10.00000 21.00000     0.80000 rf_random_grid_model_203
0.15089
## 237  30.00000 10.00000 21.00000     0.80000  rf_random_grid_model_25
0.15089
## 238  30.00000  1.00000 21.00000     0.80000 rf_random_grid_model_121
0.15812
## 239  20.00000  1.00000 21.00000     0.80000 rf_random_grid_model_237
0.15812
## 240  10.00000  1.00000 21.00000     0.80000 rf_random_grid_model_160
0.15950
```

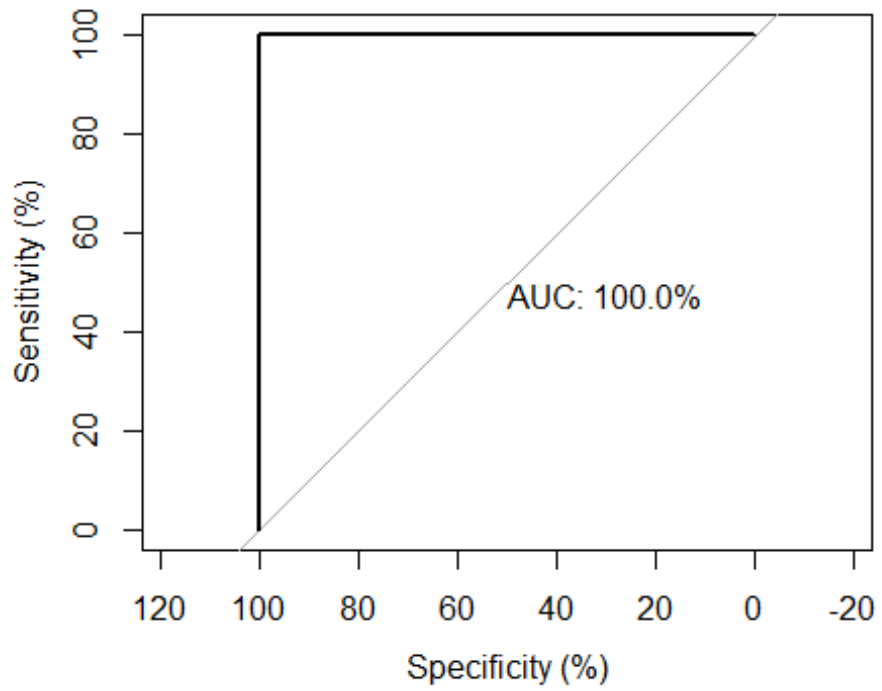
```r
# re-run model with impurity-based variable importance
rf_impurity <- ranger(
  formula = Failure.binary ~ .,
  data = trainDF,
  num.trees = 2000,
  mtry = 32,
  min.node.size = 1,
  sample.fraction = .80,
  replace = FALSE,
  importance = "impurity",
  respect.unordered.factors = "order",
  verbose = FALSE,
  seed  = 123
)

# re-run model with permutation-based variable importance
rf_permutation <- ranger(
  formula = Failure.binary ~ .,
  data = trainDF,
  num.trees = 2000,
  mtry = 32,
  min.node.size = 1,
  sample.fraction = .80,
  replace = FALSE,
  importance = "permutation",
  respect.unordered.factors = "order",
  verbose = FALSE,
  seed  = 123
)

p1 <- vip::vip(rf_impurity, num_features = 25, bar = FALSE)
p2 <- vip::vip(rf_permutation, num_features = 25, bar = FALSE)

gridExtra::grid.arrange(p1, p2, nrow = 1)
```




```
# Compute predicted probabilities on training data
m1_prob <- predict(h2o_rf1, train_h2o, type = "prob")
m1_prob=as.data.frame(m1_prob)[,2]
train_h2o=as.data.frame(train_h2o)
# Compute AUC metrics for cv_model1,2 and 3
perf1 <- prediction(m1_prob,train_h2o$Failure.binary) %>%
  performance(measure = "tpr", x.measure = "fpr")


# Plot ROC curves for cv_model1,2 and 3
plot(perf1, col = "black", lty = 2)
```

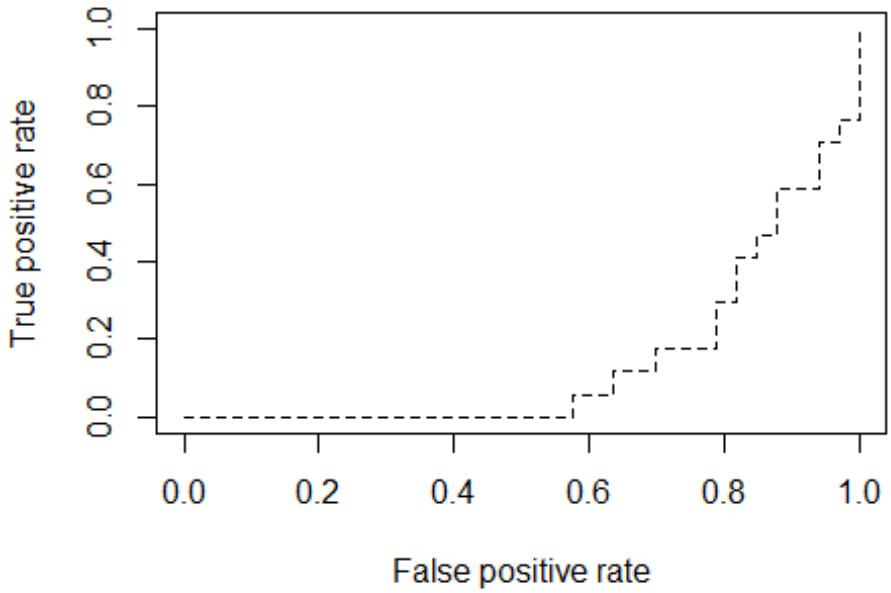



```
# ROC plot for training data
roc( train_h2o$Failure.binary ~ m1_prob, plot=TRUE, legacy.axes=FALSE,
    percent=TRUE, col="black", lwd=2, print.auc=TRUE)

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

## Setting direction: controls > cases
```

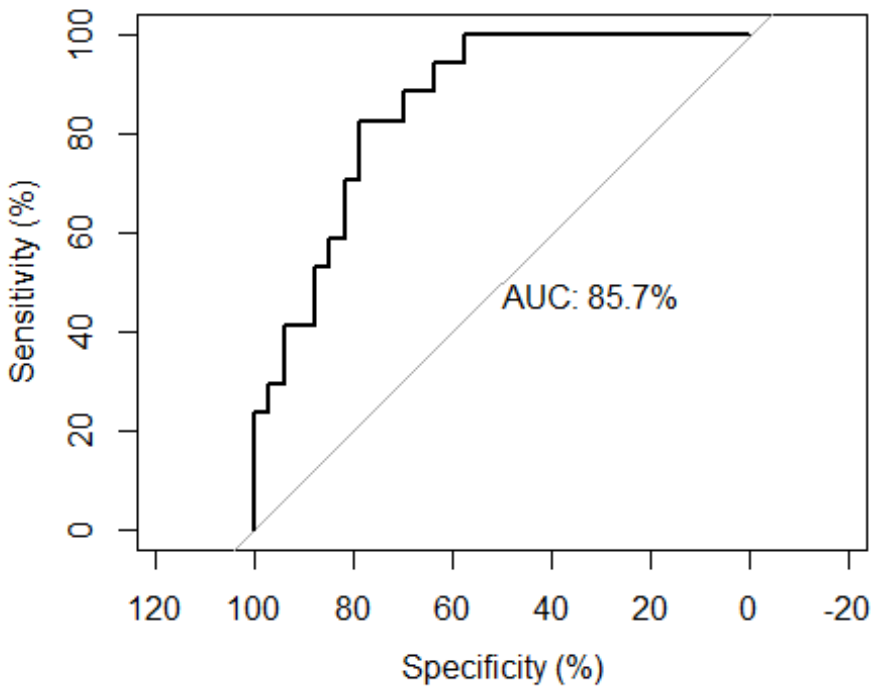



```
##
## Call:
## roc.formula(formula = train_h2o$Failure.binary ~ m1_prob, plot = TRUE,
legacy.axes = FALSE, percent = TRUE, col = "black", lwd = 2,     print.auc =
TRUE)
##
## Data: m1_prob in 97 controls (train_h2o$Failure.binary 0) > 50 cases
(train_h2o$Failure.binary 1).
## Area under the curve: 100%

#
# #Feature Interpretation
# vip(cv_model3, num_features = 20)

# Compute predicted probabilities on training data
test_h2o=as.h2o(testdt)

m2_prob <- predict(h2o_rf1, test_h2o, type = "prob")

m2_prob=as.data.frame(m2_prob)[,2]

test_h2o=as.data.frame(test_h2o)

# Compute AUC metrics for cv_model1,2 and 3
perf2 <- prediction(m2_prob,test_h2o$Failure.binary) %>%
  performance(measure = "tpr", x.measure = "fpr")
```

```r
# Plot ROC curves for cv_model1,2 and 3
plot(perf2, col = "black", lty = 2)
```




```r
# ROC plot for training data
roc( test_h2o$Failure.binary ~ m2_prob, plot=TRUE, legacy.axes=FALSE,
    percent=TRUE, col="black", lwd=2, print.auc=TRUE)

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




```
##
## Call:
## roc.formula(formula = test_h2o$Failure.binary ~ m2_prob, plot = TRUE,
legacy.axes = FALSE, percent = TRUE, col = "black", lwd = 2,     print.auc =
TRUE)
##
## Data: m2_prob in 33 controls (test_h2o$Failure.binary 0) > 17 cases
(test_h2o$Failure.binary 1).
## Area under the curve: 85.74%
```