3\_Report\_RandomForest.R

setwd("")  
  
########################################################################  
#Definition Random Forest  
########################################################################  
#What is the main difference between bagged trees and the Random Forest algorithm?  
#In Random Forest, only a subset of features are selected at random at each split in a decision tree.   
#In bagging, all features are used.  
credit <- read.csv("credit.csv", stringsAsFactors = TRUE)  
########################################################################  
#Split data in 80% 20%  
########################################################################  
# Total number of rows in the credit data frame  
n <- nrow(credit)  
# Number of rows for the training set (80% of the dataset)  
n\_train <- round(0.8 \* n)   
# Create a vector of indices which is an 80% random sample  
set.seed(123)  
train\_indices <- sample(1:n, n\_train)  
# Subset the credit data frame to training indices only  
credit\_train <- credit[train\_indices, ]   
# Exclude the training indices to create the test set  
credit\_test <- credit[-train\_indices, ]  
  
########################################################################  
#Train a Random Forest  
########################################################################  
library(randomForest)

## randomForest 4.6-14

## Type rfNews() to see new features/changes/bug fixes.

# Train a Random Forest  
set.seed(1) # for reproducibility  
credit\_model <- randomForest(formula = default ~ .,   
 data = credit\_train)  
  
# Print the model output   
print(credit\_model)

##   
## Call:  
## randomForest(formula = default ~ ., data = credit\_train)   
## Type of random forest: classification  
## Number of trees: 500  
## No. of variables tried at each split: 4  
##   
## OOB estimate of error rate: 23.62%  
## Confusion matrix:  
## no yes class.error  
## no 518 44 0.07829181  
## yes 145 93 0.60924370

########################################################################  
#Evaluate out-of-bag error  
########################################################################  
#plot the OOB error as a function of the number of trees trained  
#extract the final OOB error of the Random Forest model from the trained model object.  
# Grab OOB error matrix & take a look  
err <- credit\_model$err.rate  
head(err)

## OOB no yes  
## [1,] 0.3414634 0.2657005 0.5375000  
## [2,] 0.3183761 0.2255193 0.5572519  
## [3,] 0.3136594 0.2099057 0.5739645  
## [4,] 0.3318519 0.2092050 0.6294416  
## [5,] 0.3073713 0.1699605 0.6338028  
## [6,] 0.3012048 0.1628352 0.6222222

# Look at final OOB error rate (last row in err matrix)  
oob\_err <- err[nrow(err), "OOB"]  
print(oob\_err)

## OOB   
## 0.23625

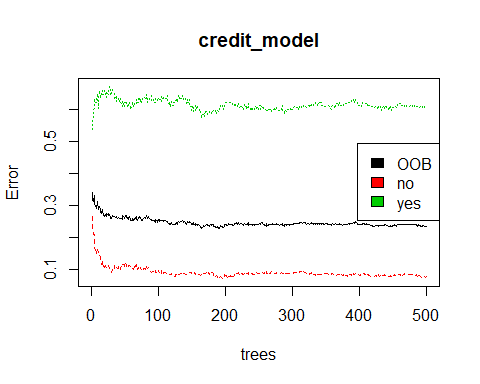
# Plot the model trained in the previous exercise  
plot(credit\_model)  
  
# Add a legend since it doesn't have one by default  
legend(x = "right",   
 legend = colnames(err),  
 fill = 1:ncol(err))  
  
########################################################################  
#Evaluate model performance on a test set  
########################################################################  
#Use the caret::confusionMatrix() function to compute test set accuracy   
#Compare the test set accuracy to the OOB accuracy.  
library(caret)

## Loading required package: lattice

## Loading required package: ggplot2

##   
## Attaching package: 'ggplot2'

## The following object is masked from 'package:randomForest':  
##   
## margin



# Generate predicted classes using the model object  
class\_prediction <- predict(object = credit\_model, # model object   
 newdata = credit\_test, # test dataset  
 type = "class") # return classification labels  
  
# Calculate the confusion matrix for the test set  
cm <- confusionMatrix(data = class\_prediction, # predicted classes  
 reference = credit\_test$default) # actual classes  
print(cm)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction no yes  
## no 130 41  
## yes 8 21  
##   
## Accuracy : 0.755   
## 95% CI : (0.6894, 0.8129)  
## No Information Rate : 0.69   
## P-Value [Acc > NIR] : 0.02614   
##   
## Kappa : 0.329   
## Mcnemar's Test P-Value : 4.844e-06   
##   
## Sensitivity : 0.9420   
## Specificity : 0.3387   
## Pos Pred Value : 0.7602   
## Neg Pred Value : 0.7241   
## Prevalence : 0.6900   
## Detection Rate : 0.6500   
## Detection Prevalence : 0.8550   
## Balanced Accuracy : 0.6404   
##   
## 'Positive' Class : no   
##

#Accuracy = 0.755  
  
# Compare test set accuracy to OOB accuracy  
paste0("Test Accuracy: ", cm$overall[1])

## [1] "Test Accuracy: 0.755"

paste0("OOB Accuracy: ", 1 - oob\_err)

## [1] "OOB Accuracy: 0.76375"

########################################################################  
#Advantage of OOB error  
########################################################################  
#QUESTION : What is the main advantage of using OOB error instead of validation or test error?  
#REPSONSE : If you evaluate your model using OOB error, then you don't need to create a separate test set.  
#COMMENT : This allows you to use all of rows in your original dataset for training  
  
########################################################################  
#Evaluate test set AUC  
########################################################################  
# Generate predictions on the test set  
pred <- predict(object = credit\_model,  
 newdata = credit\_test,  
 type = "prob")  
  
# `pred` is a matrix  
class(pred)

## [1] "matrix" "votes"

# Look at the pred format  
head(pred)

## no yes  
## 3 0.926 0.074  
## 10 0.302 0.698  
## 11 0.394 0.606  
## 14 0.820 0.180  
## 27 0.748 0.252  
## 28 0.646 0.354

library(Metrics)

##   
## Attaching package: 'Metrics'

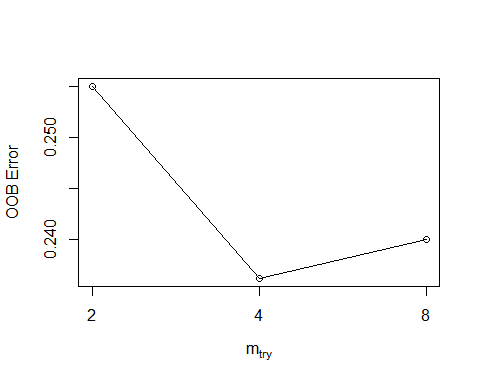
## The following objects are masked from 'package:caret':  
##   
## precision, recall

# Compute the AUC (`actual` must be a binary 1/0 numeric vector)  
auc(actual = ifelse(credit\_test$default == "yes", 1, 0),   
 predicted = pred[,"yes"])

## [1] 0.7989715

#AUC = 0.7989  
  
########################################################################  
#Tuning a Random Forest via mtry  
########################################################################  
#randomForest::tuneRF() to tune mtry (by training several models).  
  
#This function is a specific utility to tune the mtry parameter based on OOB error  
#which is helpful when you want a quick & easy way to tune your model  
#ntreeTry that defaults to 50   
# Execute the tuning process  
set.seed(1)   
res <- tuneRF(x = subset(credit\_train, select = -default),  
 y = credit\_train$default,  
 ntreeTry = 500)

## mtry = 4 OOB error = 23.62%   
## Searching left ...  
## mtry = 2 OOB error = 25.5%   
## -0.07936508 0.05   
## Searching right ...  
## mtry = 8 OOB error = 24%   
## -0.01587302 0.05



# Look at results  
print(res)

## mtry OOBError  
## 2.OOB 2 0.25500  
## 4.OOB 4 0.23625  
## 8.OOB 8 0.24000

# Find the mtry value that minimizes OOB Error  
mtry\_opt <- res[,"mtry"][which.min(res[,"OOBError"])]  
print(mtry\_opt)

## 4.OOB   
## 4

# If you just want to return the best RF model (rather than results)  
# you can set `doBest = TRUE` in `tuneRF()` to return the best RF model  
# instead of a set performance matrix.  
  
#Mtry which minimize the OOB error (0.23) is 4 mtry.  
  
########################################################################  
#Tuning a Random Forest via tree depth  
########################################################################  
#In Chapter 2, we created a manual grid of hyperparameters using the expand.grid()  
#In this exercise, you will create a grid of mtry, nodesize and sampsize values  
  
#In this example, we will identify the "best model" based on OOB error.  
#The best model is defined as the model from our grid which minimizes OOB error.  
  
#Keep in mind that there are other ways to select a best model from a grid, such as choosing the best model based on validation AUC  
  
# Establish a list of possible values for mtry, nodesize and sampsize  
mtry <- seq(4, ncol(credit\_train) \* 0.8, 2)  
nodesize <- seq(3, 8, 2)  
sampsize <- nrow(credit\_train) \* c(0.7, 0.8)  
  
# Create a data frame containing all combinations   
hyper\_grid <- expand.grid(mtry = mtry, nodesize = nodesize, sampsize = sampsize)  
  
# Create an empty vector to store OOB error values  
oob\_err <- c()  
  
# Write a loop over the rows of hyper\_grid to train the grid of models  
for (i in 1:nrow(hyper\_grid)) {  
   
 # Train a Random Forest model  
 model <- randomForest(formula = default ~ .,   
 data = credit\_train,  
 mtry = hyper\_grid$mtry[i],  
 nodesize = hyper\_grid$nodesize[i],  
 sampsize = hyper\_grid$sampsize[i])  
   
 # Store OOB error for the model   
 oob\_err[i] <- model$err.rate[nrow(model$err.rate), "OOB"]  
}  
  
# Identify optimal set of hyperparmeters based on OOB error  
opt\_i <- which.min(oob\_err)  
print(hyper\_grid[opt\_i,])

## mtry nodesize sampsize  
## 19 10 3 640

########################################################################  
#Evaluate BEST MODEL based on tuning hyperparamters on OOB erros  
########################################################################  
  
#Train Best Random Forest  
Best\_model <- randomForest(formula = default ~.,  
 data = credit\_train,  
 mtry = 10,  
 nodesize = 3,  
 sampsize = 640)  
Best\_pred <- predict(object = Best\_model,  
 newdata = credit\_test,  
 type = "prob")  
  
#Saved Positive prediction  
saveRDS(Best\_pred[,"yes"],file = "rf\_preds")  
  
  
# Compute the AUC (`actual` must be a binary 1/0 numeric vector)  
auc(actual = ifelse(credit\_test$default == "yes", 1, 0),   
 predicted = Best\_pred[,"yes"])

## [1] 0.8121786

#AUC = 0.81121