DA5030.Proj.Sui

Xin (Sue) Sui

2020-04-15

CRISP-DM Business Understanding ——————————————————

Bank churning is defined as movement of customers from one bank to another due to several reasons such as: low interest rates and fees, customer service, latest technology, store hours and locations. As we know, it is more costly to sign in new clients than retaining an existing one. By the time customers churn, banks lose money and it is too late to know the reason.

The goal is to build a model to predict whether a customer will churn or not based on customers' background and financial status.

- Predict whether a customer will churn or not.
- Help banks determine and develop a churn prevention plan to prevent customers from churning.
- Reduce loses to banks.
- Better serve customers and retain them.

CRISP-DM Data Understanding ———————————————————–

1. Data Acquisition:

- acquisition of data (e.g., CSV or flat file)

This is the data set used in the section "ANN (Artificial Neural Networks)" of the Udemy course from Kirill Eremenko (Data Scientist & Forex Systems Expert) and Hadelin de Ponteves (Data Scientist), called Deep Learning A-Z™: Hands-On Artificial Neural Networks.

This data was obtained from Kaggle: https://www.kaggle.com/adammaus/predicting-churn-for-bank-customers (https://www.kaggle.com/adammaus/predicting-churn-for-bank-customers)

```
# 1. Load the data into dataframe
df <- read.csv("Churn_Modelling.csv", header = TRUE, stringsAsFactors = TRUE)
```

2. Data Exploration:

- Exploratory data plots

- Detection of outliers

- Correlation/collinearity analysis

```
# 1. Check data
head(df)
```

```
str(df)
```

```
## 'data.frame':    10000 obs. of  14 variables:
##  $ RowNumber      : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ CustomerId     : int  15634602 15647311 15619304 15701354 15737888 15574012 15592531 15656148 15792365 1559
2389 ...
##  $ Surname        : Factor w/ 2932 levels "Abazu","Abbie",..: 1116 1178 2041 290 1823 538 178 2001 1147 1082
...
##  $ CreditScore    : int  619 608 502 699 850 645 822 376 501 684 ...
##  $ Geography      : Factor w/ 3 levels "France","Germany",..: 1 3 1 1 3 3 1 2 1 1 ...
##  $ Gender         : Factor w/ 2 levels "Female","Male": 1 1 1 1 1 2 2 1 2 2 ...
##  $ Age            : int  42 41 42 39 43 44 50 29 44 27 ...
##  $ Tenure         : int  2 1 8 1 2 8 7 4 4 2 ...
##  $ Balance        : num  0 83808 159661 0 125511 ...
##  $ NumOfProducts  : int  1 1 3 2 1 2 2 4 2 1 ...
##  $ HasCrCard      : int  1 0 1 0 1 1 1 1 0 1 ...
##  $ IsActiveMember : int  1 1 0 0 1 0 1 0 1 1 ...
##  $ EstimatedSalary: num  101349 112543 113932 93827 79084 ...
##  $ Exited         : int  1 0 1 0 0 1 0 1 0 0 ...
```

```
summary(df)
```

```
##    RowNumber       CustomerId           Surname      CreditScore
##  Min.   :    1   Min.   :15565701   Smith   :  32   Min.   :350.0
##  1st Qu.: 2501   1st Qu.:15628528   Martin  :  29   1st Qu.:584.0
##  Median : 5000   Median :15690738   Scott   :  29   Median :652.0
##  Mean   : 5000   Mean   :15690941   Walker  :  28   Mean   :650.5
##  3rd Qu.: 7500   3rd Qu.:15753234   Brown   :  26   3rd Qu.:718.0
##  Max.   :10000   Max.   :15815690   Genovese:  25   Max.   :850.0
##                                     (Other) :9831
##    Geography       Gender          Age            Tenure          Balance
##  France :5014   Female:4543   Min.   :18.00   Min.   : 0.000   Min.   :     0
##  Germany:2509   Male  :5457   1st Qu.:32.00   1st Qu.: 3.000   1st Qu.:     0
##  Spain  :2477                 Median :37.00   Median : 5.000   Median : 97199
##                               Mean   :38.92   Mean   : 5.013   Mean   : 76486
##                               3rd Qu.:44.00   3rd Qu.: 7.000   3rd Qu.:127644
##                               Max.   :92.00   Max.   :10.000   Max.   :250898
##
##  NumOfProducts    HasCrCard      IsActiveMember   EstimatedSalary
##  Min.   :1.00   Min.   :0.0000   Min.   :0.0000   Min.   :    11.58
##  1st Qu.:1.00   1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.: 51002.11
##  Median :1.00   Median :1.0000   Median :1.0000   Median :100193.91
##  Mean   :1.53   Mean   :0.7055   Mean   :0.5151   Mean   :100090.24
##  3rd Qu.:2.00   3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:149388.25
##  Max.   :4.00   Max.   :1.0000   Max.   :1.0000   Max.   :199992.48
##
##      Exited
##  Min.   :0.0000
##  1st Qu.:0.0000
##  Median :0.0000
##  Mean   :0.2037
##  3rd Qu.:0.0000
##  Max.   :1.0000
##
```

```
table(df$Exited)
```

```
##
##    0    1
## 7963 2037
```

```
# Has 10000 observations and 14 variables (11 are usable)

# Note: RowNumber, CustomerID and Surname are not useable, and will be eliminated
# Numerical features: CreditScore, Age, Tenure, Balance,  NumOfProducts, EstimatedSalary
# Categorical features: Geography, Gender, HasCrCard, isActiveMember and Exited

numeric <- c("CreditScore", "Age", "Tenure", "Balance", "NumOfProducts", "EstimatedSalary")
categorical <- c("Geography", "Gender", "HasCrCard", "IsActiveMember", "Exited")


# 2. Check for missing and NA values, there are none
sum(is.na(df))
```

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

```
length(which(df == "?"))
```

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

```
length(which(df == "NA"))
```

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

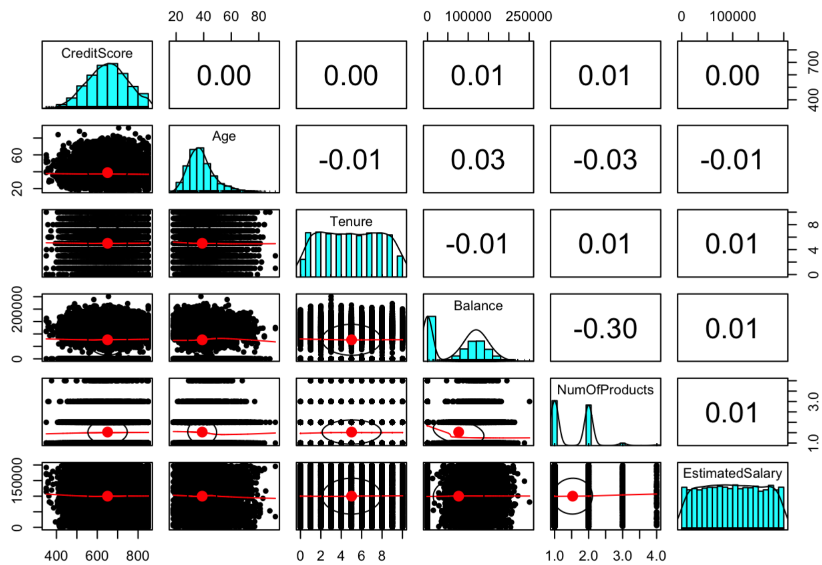
```
length(which(df == "N/A"))
```

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

```
# 3. Check for outliers
# Create z-score standarzation function.
z_normalize <- function(x) {
  return ((x - mean(x)) / sd(x))
}

# Normalize the numerical features.
numeric <- df[c(4,7:10,13)]
norm <- apply(numeric, 2, z_normalize)

# Find the outliers (outliers are considered 3 standard deviations away from the mean).
outliers <- abs(norm) > 3
sum(outliers) # 201 outliers
```

```
## [1] 201
```

```
outliers_column <- which(apply(outliers, 1, function(x) sum(x)!=0))


# 4. Check for correlation and collinearity:
pairs.panels(numeric)
```




```
# Comment: In the my dataset, there are no missing values. If there were missing data and not much, I would remov
e them and state it. If there were a lot of missing data, I would impute them with either a value between min and
max, but this might cause high variance and poor fit. Or impute with average, or median of similar data by cluste
ring. There are 201 outliers which I will remove which might increase variability when training the models. The a
lgorithms I plan to use are Naive Bayes, Decision Trees, Neural Network and SVM, removing the outliers will help
improve model performance overall. Since none of the four models I will be building are statistical learners, fol
lowing a Gaussian distribution for the data is not required.
```

CRISP-DM Data Understanding and Data Preparation

——————

1. Data Cleaning & Shaping:

- Data imputation

- Normalization/standardization of feature values

- Feature engineering: dummy codes

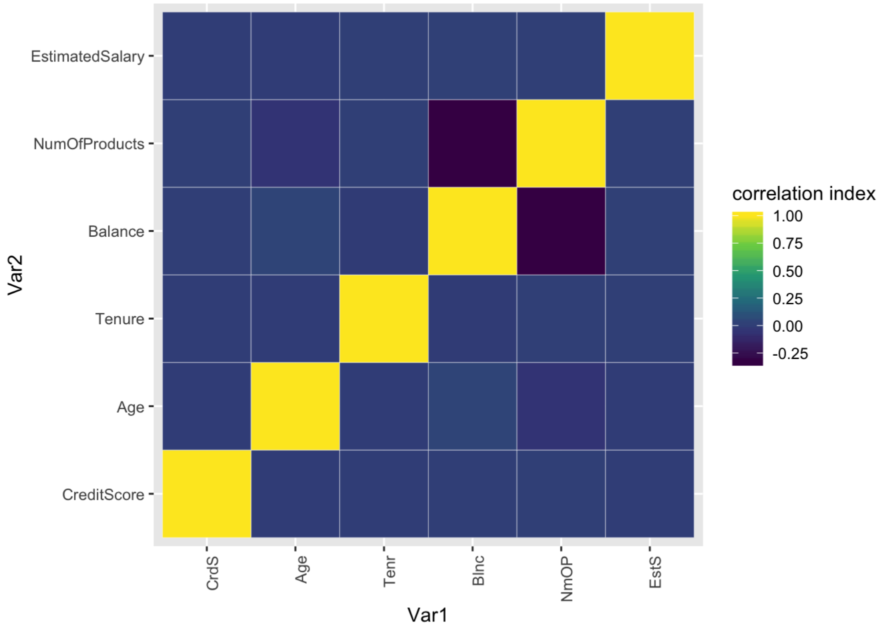
- Feature engineerring: PCA

- Feature engineering: new derived features

```
# Data cleaning

# 1. Check for useful features: First three columns are unique row number,
# customer id and their surname, I will exclude from the data.
df <- df[c(-1:-3)]


# 2. Convert categorical features to factors
df$Exited <- ifelse(df$Exited == 0, "no", "yes")
df[categorical] <- lapply(df[categorical], factor)
head(df)
```

```
# 3. Data imputation: Remove outliers
df <- df[c(-outliers_column), ]



# Data shaping: Here I will shape the data based on each model I use, some
# require normalization of numerical features; some require conversion to
# catergorical features; some require conversion to numerical features.

# 4. Normalization/standarzation of feature values:

# 4.1 Normalization: Normalize for Neural Network (min-max) it works best when
# input data are scaled.
numeric <- c("CreditScore", "Age", "Tenure", "Balance", "NumOfProducts", "EstimatedSalary")
normalize <- function(x) {
    return((x - min(x))/(max(x) - min(x)))
}
nn_norm <- as.data.frame(lapply(df[numeric], normalize))

nn_df <- cbind(nn_norm, y = df$Exited)


# 4.3 Normalization: Normalize for SVM (min-max): it works best when input data
# are scaled.
normalize <- function(x) {
    return((x - min(x))/(max(x) - min(x)))
}
svm_norm <- as.data.frame(lapply(df[numeric], normalize))



# 5. Feature engineering:

# 5.1 Dummy code categorical features to numeric for SVM
dum <- df[categorical] %>% select(-Exited)
dmy <- dummyVars("~.", dum)
svm_dmy <- data.frame(predict(dmy, newdata = dum))

svm_final_df <- cbind(svm_norm, svm_dmy, y = df$Exited)


# 5.2 New derived features for Naive Bayes by binning as it uses only categorical
# features
nb_df <- df
nb_df$CreditScore <- bin(nb_df$CreditScore, nbins = 5, labels = c("1", "2", "3",
    "4", "5"))
nb_df$Age <- bin(nb_df$Age, nbins = 5, labels = c("1", "2", "3", "4", "5"))
nb_df$Tenure <- bin(nb_df$Tenure, nbins = 5, labels = c("1", "2", "3", "4", "5"))
nb_df$Balance <- bin(nb_df$Balance, nbins = 2, labels = c("1", "2"))
nb_df$NumOfProducts <- bin(nb_df$Tenure, nbins = 4, labels = c("1", "2", "3", "4"))
nb_df$EstimatedSalary <- bin(nb_df$EstimatedSalary, nbins = 5, labels = c("1", "2",
    "3", "4", "5"))
head(nb_df)
```

```
# 5.3. Feature engineering: PCA
cor_df <- df
pca_num_df <- cor_df[c("CreditScore", "Age", "Tenure", "Balance", "NumOfProducts",
    "EstimatedSalary")]
pca_num <- apply(pca_num_df, 2, function(x) as.numeric(as.character(x)))  # Convert to all numbers

cor_num <- cor(pca_num)

ggplot(data = melt(cor_num), aes(Var1, Var2, fill = value)) + geom_tile(colour = "white") +
    scale_fill_viridis_c(name = "correlation index") + theme(axis.text.x = element_text(angle = 90,
    hjust = 1)) + scale_x_discrete(labels = abbreviate)  # Age
```




```
pca_numeric <- prcomp(pca_num_df, center = TRUE, scale = TRUE)
summary(pca_numeric)
```

```
## Importance of components:
##                           PC1    PC2    PC3    PC4    PC5    PC6
## Standard deviation     1.1595 1.0082 1.0006 0.9959 0.9895 0.8168
## Proportion of Variance 0.2241 0.1694 0.1668 0.1653 0.1632 0.1112
## Cumulative Proportion  0.2241 0.3935 0.5603 0.7256 0.8888 1.0000
```

```
# In this case, I did PCA on numerical features and found that Credit Score and
# Age are the important two factors, they explain about 40% of the total
# variation in the data.
```

CRISP-DM Data Modeling

In This section, holdout method is used to construct four different models, and each model is tuned to see whether or not it improves performance. Models built are then used to predict test data set and accuracy is calculated.

Model Construction:

1. Creation of training & validation subsets

2. Construction of at least three related models:

```
  2.1 - Neural Network (Parametric)
  2.2 - Decision Trees (Non-parametric)
  2.3 - Support Vector Machine (Parametric)
  2.4 - Naive Bayes (Parametric)
```

2.1 - I chose Neural Network because it can be used for both regression and classification.

2.2 - I chose SVM because it is good for binary classification, it attempts to find a hyper-plane separating the different classes of the training instances, with the maximum error margin.

2.3 - I choose Decision Trees because it doesn't require transformation of data and accepts both categorical and numerical features, it is very simple, fast and efficient.

2.4 - I choose Naive Bayes as a comparison to the previous three and see how binning the numerical features perform in the overall model.

1. Creation of training & validation subsets

```
# set.seed for reproducibility.
set.seed(123)

# Hold out method using stratified holdout sampling:
split <- createDataPartition(df$Exited, p = 0.75, list = FALSE)

training <- df[split, ]
testing <- df[-split, ]

table(training$Exited) %>% prop.table
```

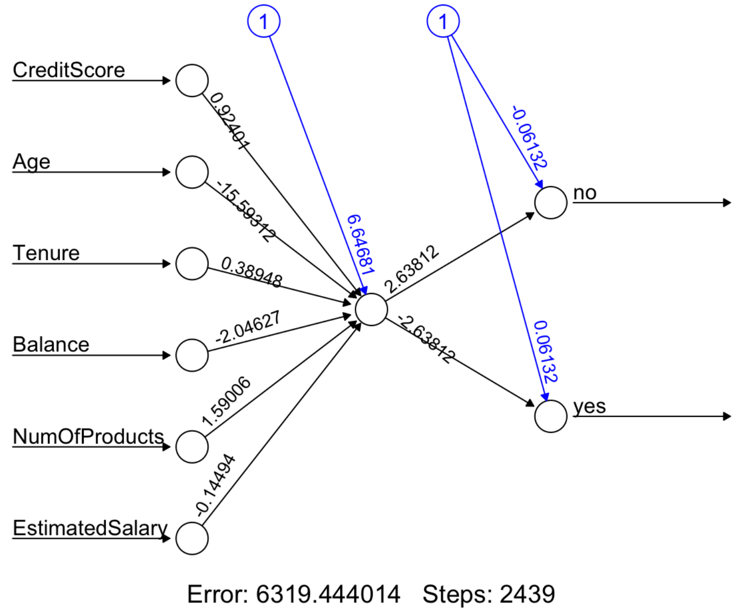
```
##
##        no       yes
## 0.8001361 0.1998639
```

```
table(testing$Exited) %>% prop.table
```

```
##
##        no       yes
## 0.8003267 0.1996733
```

2.1. Construction of Neural Network

```
set.seed(123)

# 1. Split the data (Holdout method)
train <- nn_df[split, ]
test <- nn_df[-split, ]

# 2. Construct Neural Network classifier (for binary classification, act.fct =
# logistic is used here)
nn <- neuralnet(y ~ ., data = train, hidden = 1, linear.output = FALSE, err.fct = "ce",
    act.fct = "logistic", likelihood = TRUE)

plot(nn, rep = "best")
```



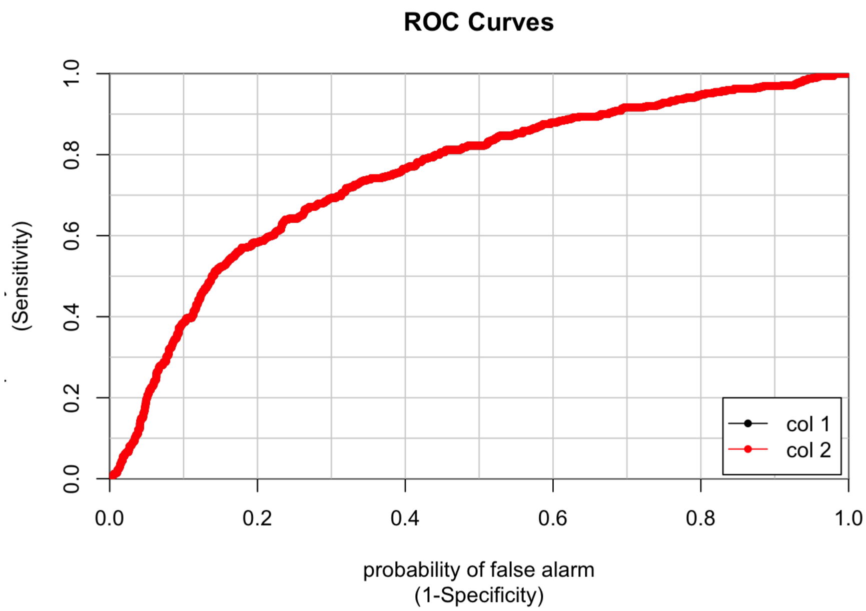

```
result <- compute(nn, test[-7])
nn_prediction <- as.factor(ifelse(result$net.result[, 1] < 0.5, "yes", "no"))

confusionMatrix(nn_prediction, test$y, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1863  395
##        yes   97   94
##
##                Accuracy : 0.7991
##                  95% CI : (0.7827, 0.8148)
##     No Information Rate : 0.8003
##     P-Value [Acc > NIR] : 0.5721
##
##                   Kappa : 0.1851
##
##  Mcnemar's Test P-Value : <2e-16
##
##               Precision : 0.8251
##                  Recall : 0.9505
##                      F1 : 0.8834
##              Prevalence : 0.8003
##          Detection Rate : 0.7607
##    Detection Prevalence : 0.9220
##       Balanced Accuracy : 0.5714
##
##        'Positive' Class : no
##
```

```
# Accuracy : 0.7991 Precision : 0.8251 Recall : 0.9505 F1 : 0.8834

# Accuracy: measure of all the correctly identified cases.  Precision: measure of
# the correctly identified positive cases from all the predicted positive cases.
# Recall: measure of the correctly identified positive cases from all the actual
# positive cases.  F-score: measure of test accuracy. It combines both the
# precision and the recall using the harmonic mean, it describes the model
# performance. It gives a better measure of the incorrectly classified cases than
# the Accuracy Metric.


# AUC--------------------------------------------------------------------------------
prob_nn <- result$net.result
colAUC(prob_nn, test$y)
```

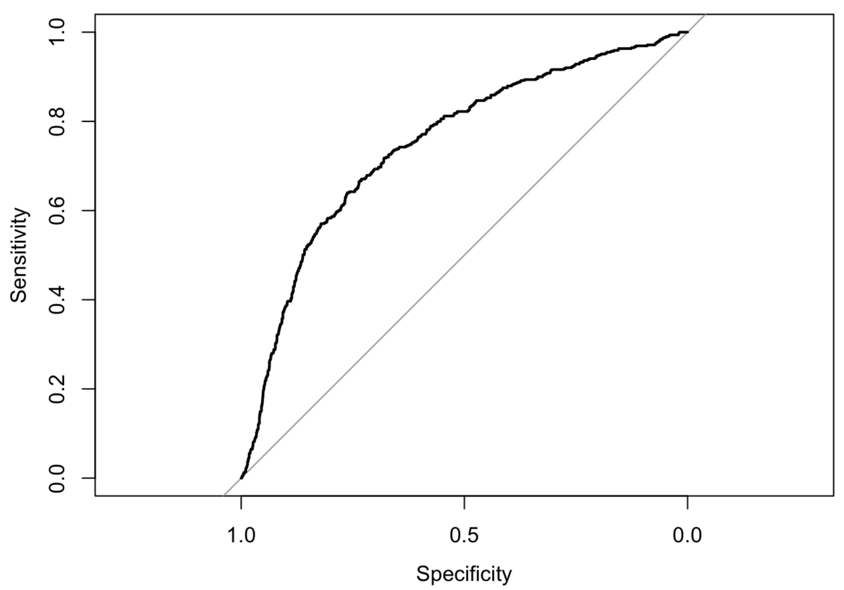
```
##                  [,1]      [,2]
## no vs. yes 0.7485216 0.7485216
```

```
colAUC(prob_nn, test$y, plotROC = TRUE)
```




```
##                  [,1]      [,2]
## no vs. yes 0.7485216 0.7485216
```

```
auc_nn <- roc(response = test$y, predictor = result$net.result[, 1])
plot(auc_nn)
```




```
auc_nn$auc  # 74.85%
```

```
## Area under the curve: 0.7485
```

```
# Comment: Here I choose to use all numeric features from the data to build
# neural network classifier because the with the the rest of categorical
# features, the computer runs so much and takes up all the CPU and takes a very
# long run time. (I've tried to use both numerical and categorical features, but
# the has a very long run time and uses a lot of computer memory). According to
# the book and Professor's notes, Neural Network can take in both numerical and
# categorical features and automatically converts them to dummy code, and assigns
# bias and weights to each input and back-propagates, it is considered as a black
# box, NP-complete problem which takes an exponential amount of time to run with
# the amount of input data, it takes up all the CPU and very computationally
# expensive. I will demonstrate tuning parameter for Neural Network in the
# following k-fold cross validation step. Accuracy for improved model is 83.54%,
# F-score is 88.34% and AUC is 80.15%.
```

2.2. Construction of Support Vector Machine

```
set.seed(123)

# Hold out method using stratified holdout sampling:
split <- createDataPartition(df$Exited, p = 0.75, list = FALSE)

train <- svm_final_df[split,]
test <- svm_final_df[-split,]


svm_classifier_l <- svm(y ~ ., data = train, kernel="linear", scaled = TRUE, probability = TRUE)

pred_svm_l <- predict(svm_classifier_l, test[-16], decision.values = TRUE, probability = TRUE)

svm_prediction_l <- predict(svm_classifier_l, test[-16], type = "prob")


confusionMatrix(svm_prediction_l, test$y, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1960  489
##        yes    0    0
##
##                Accuracy : 0.8003
##                  95% CI : (0.7839, 0.816)
##     No Information Rate : 0.8003
##     P-Value [Acc > NIR] : 0.5121
##
##                   Kappa : 0
##
##  Mcnemar's Test P-Value : <2e-16
##
##               Precision : 0.8003
##                  Recall : 1.0000
##                      F1 : 0.8891
##              Prevalence : 0.8003
##          Detection Rate : 0.8003
##    Detection Prevalence : 1.0000
##       Balanced Accuracy : 0.5000
##
##        'Positive' Class : no
##
```

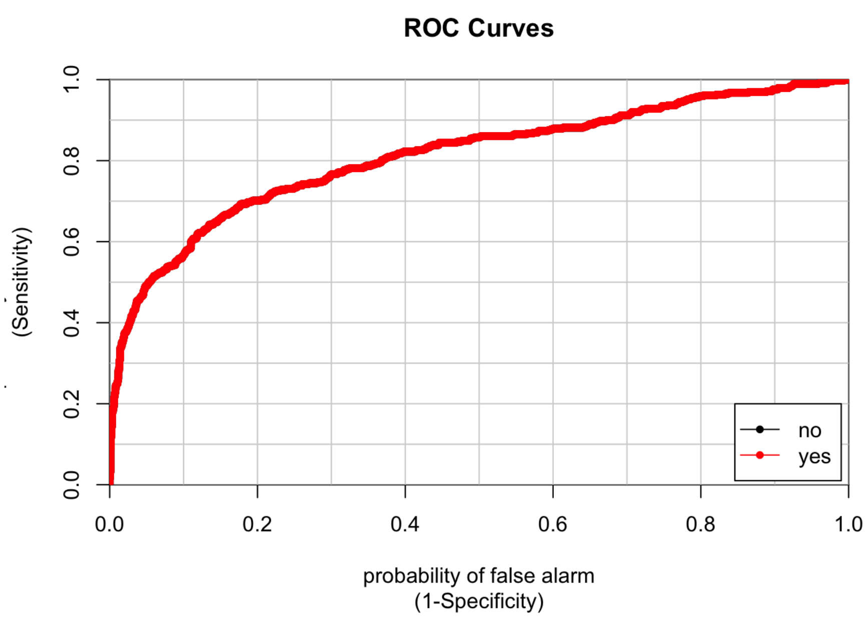
```
# Accuracy : 0.8003
# Precision : 0.8003
# Recall : 1.0000


# Improvement with kernel = radial
svm_classifier_k <- svm(y ~ ., data = train, kernel="radial", scaled = TRUE, probability = TRUE)

pred_svm_k <- predict(svm_classifier_k, test[-16], decision.values = TRUE, probability = TRUE)

svm_prediction_k <- predict(svm_classifier_k, test[-16], type = "prob")

confusionMatrix(svm_prediction_k, test$y, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1917  305
##        yes   43  184
##
##                Accuracy : 0.8579
##                  95% CI : (0.8434, 0.8715)
##     No Information Rate : 0.8003
##     P-Value [Acc > NIR] : 7.141e-14
##
##                   Kappa : 0.4435
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##               Precision : 0.8627
##                  Recall : 0.9781
##                      F1 : 0.9168
##              Prevalence : 0.8003
##          Detection Rate : 0.7828
##    Detection Prevalence : 0.9073
##       Balanced Accuracy : 0.6772
##
##        'Positive' Class : no
##
```

```
# Accuracy : 0.8579
# Precision : 0.8627
# Recall : 0.9781
# F1 : 0.9168


# AUC-----------------------------------------------------------------------------------------
prob_svm_k <- attr(pred_svm_k, "probabilities")

colAUC(prob_svm_k, test$y)
```

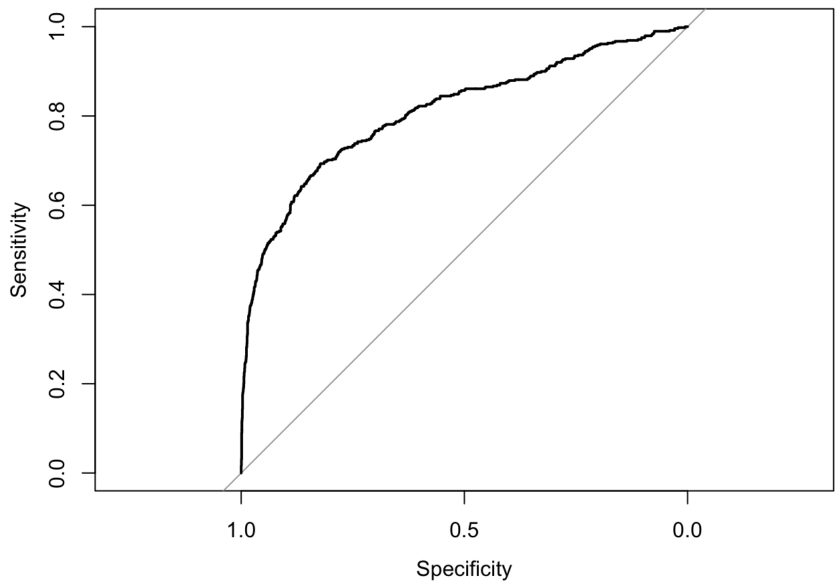
```
##                    no       yes
## no vs. yes 0.8107508 0.8107508
```

```
colAUC(prob_svm_k, test$y, plotROC = "TRUE")
```




```
##                    no       yes
## no vs. yes 0.8107508 0.8107508
```

```
auc_svm_k <- roc(response=test$y, predictor=prob_svm_k[,1])
plot(auc_svm_k)
```




```
auc_svm_k$auc # 81.08%
```

```
## Area under the curve: 0.8108
```

```
# Comment: I chose SVM because it is mostly understood when used for binary classication. It is a distance based
algorithm, it creates a flat boundary known as hyperplane which divides the space to create farily homogeneous pa
rtitions on either side. It combines kNN and linear gressions. It is very powerful, and and model highly complex
relationships. SVM model did improve after setting the kernal to radial, as linear did not perform as well. Radia
l works better here because the underlying data is not linearly separable. After model improvement, the accuracy
is 85.79%, F-score is 91.68% and AUC is 81.08% which is the second highest compare to models previously built.
```

2.3. Construction of Decision Trees

```
# Decision Trees:
# Training using data frame without dummy code since decision tree takes both categorical and numeric variables.
decision_tree <- C5.0(training[-11], training$Exited)
summary(decision_tree)
```

```
##
## Call:
## C5.0.default(x = training[-11], y = training$Exited)
##
##
## C5.0 [Release 2.07 GPL Edition]      Wed Apr 15 12:21:41 2020
## -------------------------------
##
## Class specified by attribute `outcome'
##
## Read 7350 cases (11 attributes) from undefined.data
##
## Decision tree:
##
## NumOfProducts > 2:
## :...Balance > 50194.59: yes (113/4)
## :   Balance <= 50194.59:
## :   :...Age > 42: yes (31/2)
## :       Age <= 42:
## :       :...EstimatedSalary <= 153064.9: no (38/13)
## :           EstimatedSalary > 153064.9: yes (10)
## NumOfProducts <= 2:
## :...Age <= 41: no (4969/480)
##     Age > 41:
##     :...NumOfProducts > 1:
##         :...IsActiveMember = 1: no (532/61)
##         :   IsActiveMember = 0:
##         :   :...Age <= 50: no (289/43)
##         :       Age > 50:
##         :       :...Gender = Female: yes (30/5)
##         :           Gender = Male:
##         :           :...Age <= 53: no (12/3)
##         :               Age > 53: yes (21/4)
##         NumOfProducts <= 1:
##         :...IsActiveMember = 0:
##             :...Age > 47: yes (319/53)
##             :   Age <= 47:
##             :   :...Geography = Germany: yes (109/34)
##             :       Geography in {France,Spain}:
##             :       :...Balance > 97086.4: no (117/36)
##             :           Balance <= 97086.4:
##             :           :...Geography = Spain: yes (48/16)
##             :               Geography = France:
##             :               :...Tenure <= 5: yes (30/9)
##             :                   Tenure > 5: no (36/14)
##             IsActiveMember = 1:
##             :...Geography in {France,Spain}: no (467/138)
##                 Geography = Germany:
##                 :...Balance <= 87347.7: no (15/1)
##                     Balance > 87347.7:
##                     :...CreditScore > 718: yes (35/6)
##                         CreditScore <= 718:
##                         :...Gender = Male: no (64/27)
##                             Gender = Female:
##                             :...HasCrCard = 0: yes (24/7)
##                                 HasCrCard = 1:
##                                 :...Balance <= 119565.9: yes (19/4)
##                                     Balance > 119565.9: no (22/8)
##
##
## Evaluation on training data (7350 cases):
##
##      Decision Tree
##    ----------------
##    Size      Errors
##
##      23  968(13.2%)   <<
##
##
##     (a)   (b)    <-classified as
##    ----  ----
##    5737   144    (a): class no
##     824   645    (b): class yes
##
##
##  Attribute usage:
##
##  100.00% NumOfProducts
```

```
##   98.46% Age
##   29.78% IsActiveMember
##   13.41% Geography
##    8.19% Balance
##    2.61% Gender
##    2.23% CreditScore
##    0.90% Tenure
##    0.88% HasCrCard
##    0.65% EstimatedSalary
##
##
## Time: 0.0 secs
```

```
decision_tree_pred <- predict(decision_tree, testing[-11])

confusionMatrix(testing$Exited, decision_tree_pred, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1898   62
##        yes  278  211
##
##                Accuracy : 0.8612
##                  95% CI : (0.8468, 0.8746)
##     No Information Rate : 0.8885
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.4793
##
##  Mcnemar's Test P-Value : <2e-16
##
##               Precision : 0.9684
##                  Recall : 0.8722
##                      F1 : 0.9178
##              Prevalence : 0.8885
##          Detection Rate : 0.7750
##    Detection Prevalence : 0.8003
##       Balanced Accuracy : 0.8226
##
##        'Positive' Class : no
##
```

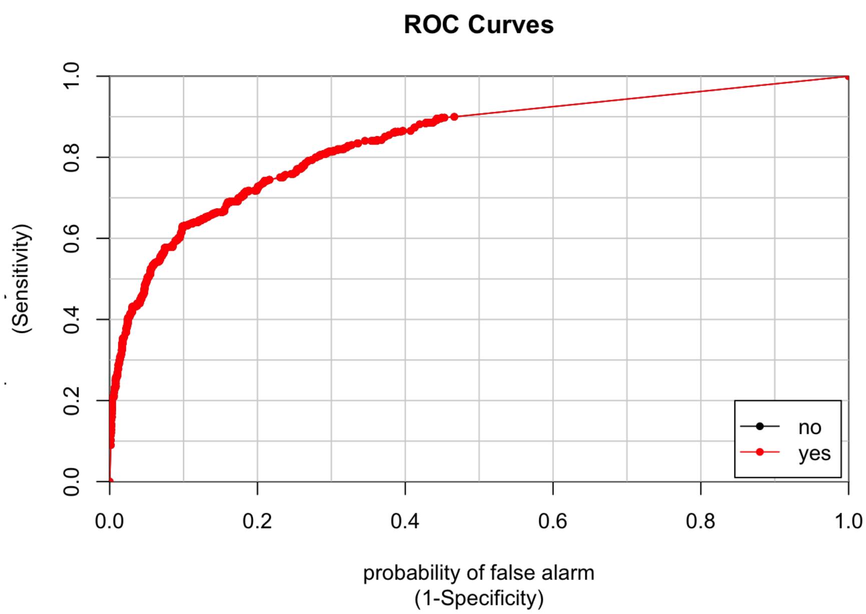
```
# Accuracy : 0.8612
# Precision : 0.9684
# Recall : 0.8722
# F1: 0.9178


# Tuning Improvement-------------------------------------------------------------------------
# Trials = 10 is boosting technique, that the algorithm will stop adding trees if the desired overall error rate
is reached or performance no longer improves with additional of trials

decision_tree_10 <- C5.0(training[-11], training$Exited, trials = 10)

decision_tree_pred_10 <- predict(decision_tree_10, testing[-11])

confusionMatrix(testing$Exited, decision_tree_pred_10, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1869   91
##        yes  262  227
##
##                Accuracy : 0.8559
##                  95% CI : (0.8413, 0.8695)
##     No Information Rate : 0.8702
##     P-Value [Acc > NIR] : 0.9825
##
##                   Kappa : 0.4809
##
##  Mcnemar's Test P-Value : <2e-16
##
##               Precision : 0.9536
##                  Recall : 0.8771
##                      F1 : 0.9137
##              Prevalence : 0.8702
##          Detection Rate : 0.7632
##    Detection Prevalence : 0.8003
##       Balanced Accuracy : 0.7954
##
##        'Positive' Class : no
##
```

```
# Accuracy : 0.8559
# Precision : 0.9536
# Recall : 0.8771
# F1 : 0.9178
# After increasing the number of trials for decision tree, model performance remained the same.


# AUC-------------------------------------------------------------------------------------------
pred_dt <- predict(decision_tree_10, testing[-11], type = "prob")

colAUC(pred_dt, test$y)
```

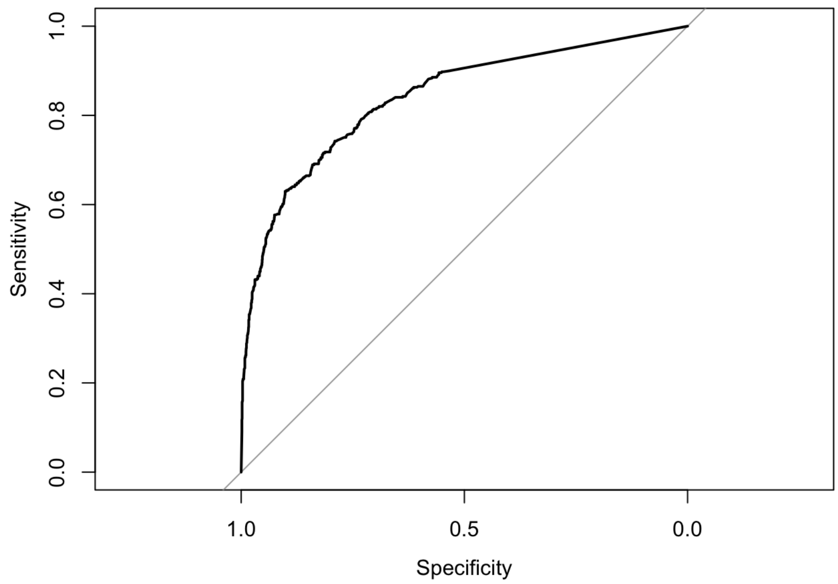
```
##                   no       yes
## no vs. yes 0.8401371 0.8401371
```

```
colAUC(pred_dt, test$y, plotROC = TRUE)
```




```
##                   no       yes
## no vs. yes 0.8401371 0.8401371
```

```
auc_dt <- roc(response=test$y, predictor=pred_dt[,1])
plot(auc_dt)
```




```
auc_dt$auc # 84.01%
```

```
## Area under the curve: 0.8401
```

```
# Comment: I chose decision tree here because it can handle numeric or categorical features. Decision Trees does
n't need dummy codes for categorical variables as it makes if-then branches; like a tree with many branches. And
for numerical features, the split is done with the elements higher than a threshold. The accuracy for improved de
cision tree is 86.12%, F-score is 92%, AUC is 84.01%, this is so far the best performing model.
```

2.4. Construction of Navie Bayes:

```
set.seed(123)

train_nb <- nb_df[split,]
test_nb <- nb_df[-split,]


m1 <- naiveBayes(train_nb[-11], train_nb$Exited)
pred1 <- predict(m1, test_nb[-11])

confusionMatrix(pred1, test_nb$Exit, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1880  361
##        yes   80  128
##
##                Accuracy : 0.8199
##                  95% CI : (0.8041, 0.835)
##     No Information Rate : 0.8003
##     P-Value [Acc > NIR] : 0.007623
##
##                   Kappa : 0.2817
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##               Precision : 0.8389
##                  Recall : 0.9592
##                      F1 : 0.8950
##              Prevalence : 0.8003
##          Detection Rate : 0.7677
##    Detection Prevalence : 0.9151
##       Balanced Accuracy : 0.6105
##
##        'Positive' Class : no
##
```

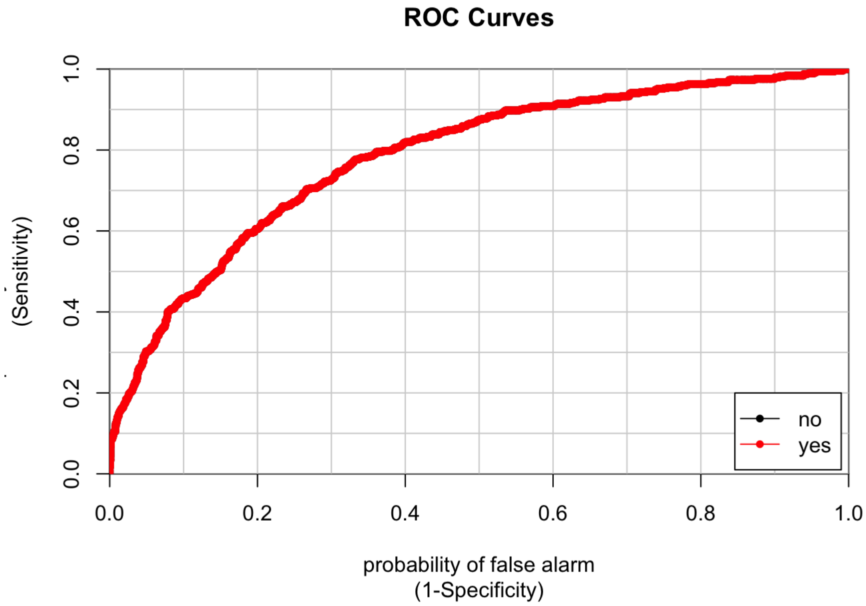
```
# Accuracy : 0.8199
# Precision : 0.8389
# Recall : 0.9592


# Improve model with laplace = 1 -------------------------------------------------------
m2 <- naiveBayes(train_nb[-11], train_nb$Exited, laplace = 1)
pred2 <- predict(m2, test_nb[-11])

confusionMatrix(pred2, test_nb$Exit, mode = "prec_recall") # 81.99% accuracy rate and kappa score of 0.2817
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1880  361
##        yes   80  128
##
##                Accuracy : 0.8199
##                  95% CI : (0.8041, 0.835)
##     No Information Rate : 0.8003
##     P-Value [Acc > NIR] : 0.007623
##
##                   Kappa : 0.2817
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##               Precision : 0.8389
##                  Recall : 0.9592
##                      F1 : 0.8950
##              Prevalence : 0.8003
##          Detection Rate : 0.7677
##    Detection Prevalence : 0.9151
##       Balanced Accuracy : 0.6105
##
##        'Positive' Class : no
##
```

```
# Accuracy : 0.8199
# Precision : 0.8389
# Recall : 0.9592
# F1 : 0.8950


# AUC-----------------------------------------------------------------------------------
pred_nb <- predict(m2, test_nb[-11], type = "raw")
colAUC(pred_nb, test$y)
```

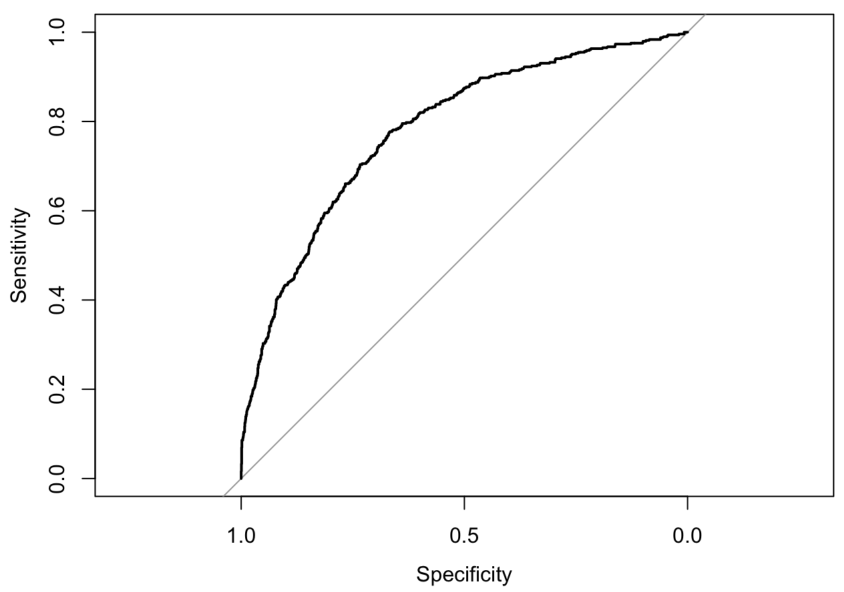
```
##                    no       yes
## no vs. yes 0.7822863 0.7822863
```

```
colAUC(pred_nb, test$y, plotROC = TRUE)
```




```
##                    no       yes
## no vs. yes 0.7822863 0.7822863
```

```
auc_nb <- roc(response=test$y, predictor=pred_nb[,1])
plot(auc_nb)
```




```
auc_nb$auc # 78.23%
```

```
## Area under the curve: 0.7823
```

```
# Comment: I chose Naive Bayes because it is a probabilistic learner that is used for classification. Naive Bayes
accepts only categorical features as it calculates probabilities, so I binned my numerical features. After tuning
the Laplace estimator, the model performance remained the same. Accuracy for Naive Bayes improved model is 81.9
9%, F-score is 89.50% and AUC is 78.23%.
```

CRISP-DM Evaluation

---

1. Model Evaluation:

- evaluation of fit of models with holdout method

- evaluation with k-fold cross-validation

- tuning of models

- comparison of models

- interpretation of results/prediction with interval

- construction of stacked ensemble model

```
# According to the above holdout methods of 4 methods:
# 1. Decision Trees:  Accuracy : 0.8559    Precision : 0.9536    Recall : 0.8771   F1 : 0.9178

# 2. SVM: Accuracy : 0.8579   Precision : 0.8627    Recall : 0.9781    F1 : 0.9168

# 3. Neural Network: Accuracy : 0.7991   Precision : 0.8251     Recall : 0.9505   F1 : 0.8834

# 4. Naive Bayes: Accuracy : 0.8199   Precision : 0.8389   Recall : 0.9592   F1 : 0.8950

# Decision Tree model has the best performance, following by SVM, Neural Network and Naive Bayes.
```

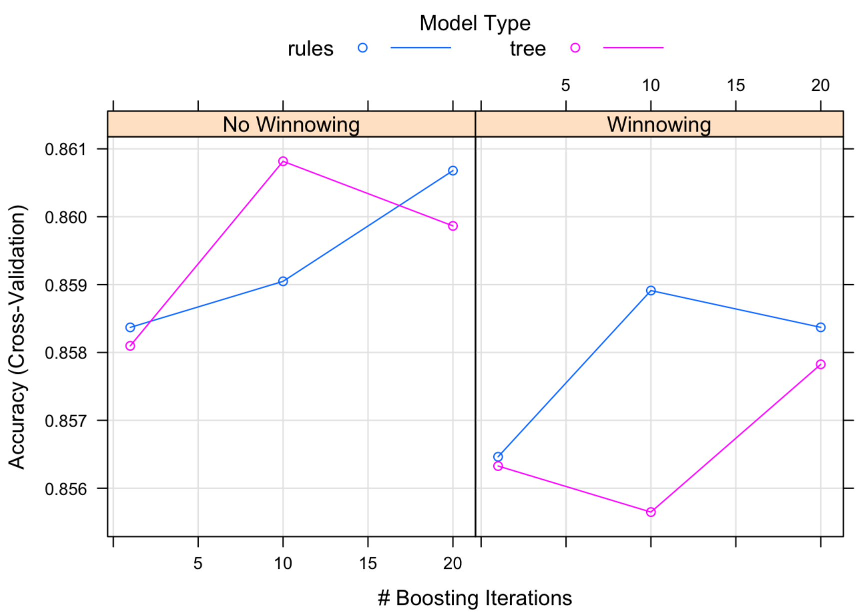
In This section, k-fold cross validation method is used to construct four different models, and each model is tuned to see whether or not it improves performance. Models built are then used to predict test data set and accuracy is calculated.

1. Decision Tree

```
set.seed(1)

# Decision Trees CV and Tuning

# 1. Evaluation of k-folds cross-validation-------------------------------------------
train <- df[split,]
test <- df[-split,]

# Create a control object that uses 10-fold cross validation
ctrl <- trainControl(method="cv", number=10, classProbs = TRUE)

dtFit_cv <- train(Exited ~ ., data = train, method = "C5.0", trControl = ctrl, preProcess = c("center","scale"))

dtFit_cv # Best model is trials = 20, model = rules and winnow = FALSE.
```

```
## C5.0
##
## 7350 samples
##   10 predictor
##    2 classes: 'no', 'yes'
##
## Pre-processing: centered (11), scaled (11)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 6616, 6615, 6615, 6614, 6615, 6615, ...
## Resampling results across tuning parameters:
##
##   model  winnow  trials  Accuracy   Kappa
##   rules  FALSE    1      0.8583695  0.4833332
##   rules  FALSE   10      0.8590485  0.4859355
##   rules  FALSE   20      0.8606797  0.4860022
##   rules   TRUE    1      0.8564633  0.4730960
##   rules   TRUE   10      0.8589126  0.4750426
##   rules   TRUE   20      0.8583703  0.4770603
##   tree   FALSE    1      0.8580976  0.4857486
##   tree   FALSE   10      0.8608172  0.4848202
##   tree   FALSE   20      0.8598650  0.4885103
##   tree    TRUE    1      0.8563272  0.4729849
##   tree    TRUE   10      0.8556484  0.4733458
##   tree    TRUE   20      0.8578253  0.4827831
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were trials = 10, model = tree and winnow
##  = FALSE.
```

```
plot(dtFit_cv)
```



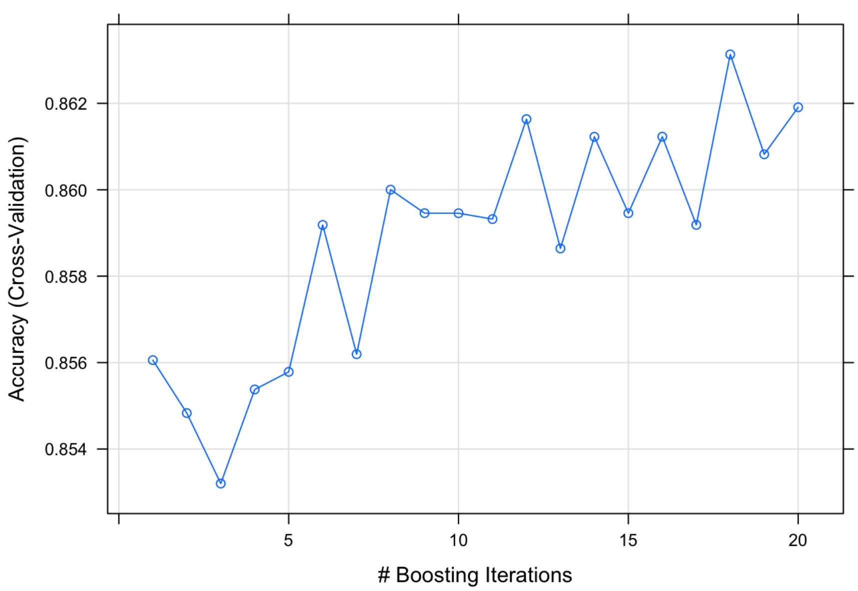

```
# Predict testing set
p_dt <- predict(dtFit_cv, test[-11])
confusionMatrix(p_dt, test$Exited, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1874  257
##        yes   86  232
##
##                Accuracy : 0.8599
##                  95% CI : (0.8456, 0.8735)
##     No Information Rate : 0.8003
##     P-Value [Acc > NIR] : 8.61e-15
##
##                   Kappa : 0.4956
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##               Precision : 0.8794
##                  Recall : 0.9561
##                      F1 : 0.9162
##              Prevalence : 0.8003
##          Detection Rate : 0.7652
##    Detection Prevalence : 0.8702
##       Balanced Accuracy : 0.7153
##
##        'Positive' Class : no
##
```

```
# Accuracy : 0.8599
# Precision : 0.8794
# Recall : 0.9561


# 2. Tuning of model--------------------------------------------------------------------
dtGrid <- expand.grid(model="tree", trials = c(1:20), winnow = FALSE)

dtFit_tune <- train(Exited ~ ., data = train,
                    method = "C5.0", metric = "ROC",
                    preProc = c("center", "scale"),
                    trControl = ctrl, tuneGrid = dtGrid)
dtFit_tune # After tunning, best model is trials = 19, model = tree and winnow = FALSE
```

```
## C5.0
##
## 7350 samples
##   10 predictor
##    2 classes: 'no', 'yes'
##
## Pre-processing: centered (11), scaled (11)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 6615, 6615, 6616, 6615, 6615, 6615, ...
## Resampling results across tuning parameters:
##
##   trials  Accuracy   Kappa
##    1      0.8560575  0.4792764
##    2      0.8548317  0.4623808
##    3      0.8532013  0.4759507
##    4      0.8553780  0.4676654
##    5      0.8557861  0.4924734
##    6      0.8591860  0.4885957
##    7      0.8561943  0.4932071
##    8      0.8600025  0.4924969
##    9      0.8594593  0.5012111
##   10      0.8594574  0.4890303
##   11      0.8593215  0.4997558
##   12      0.8616345  0.4980926
##   13      0.8586418  0.4965608
##   14      0.8612265  0.5005665
##   15      0.8594589  0.5004112
##   16      0.8612280  0.4972572
##   17      0.8591875  0.4964910
##   18      0.8631324  0.5030884
##   19      0.8608207  0.4977209
##   20      0.8619088  0.4988650
##
## Tuning parameter 'model' was held constant at a value of tree
## Tuning
##  parameter 'winnow' was held constant at a value of FALSE
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were trials = 18, model = tree and winnow
##  = FALSE.
```

```
plot(dtFit_tune)
```



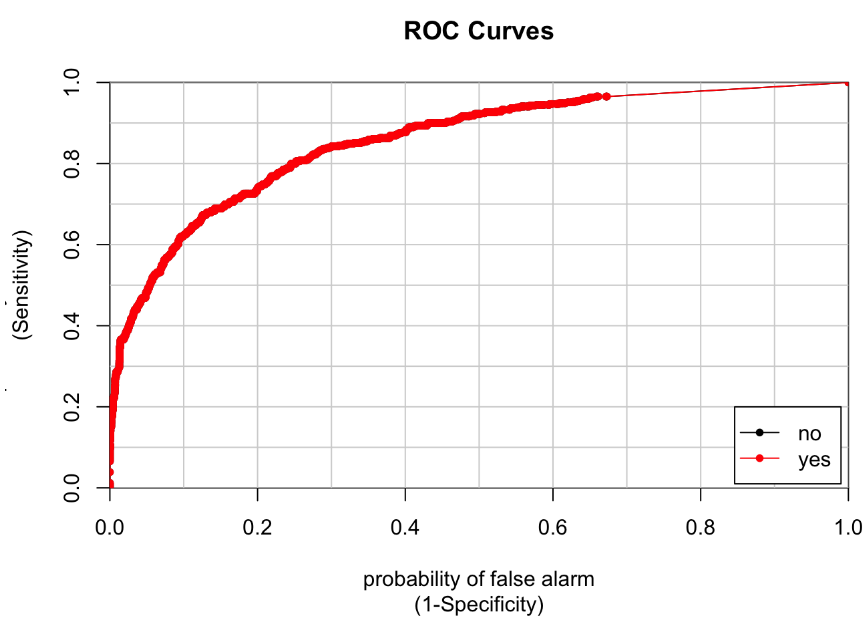

```
p_dt_tune <- predict(dtFit_tune, test[-11]) #

confusionMatrix(p_dt_tune, test$Exited, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1875  261
##        yes   85  228
##
##                Accuracy : 0.8587
##                  95% CI : (0.8443, 0.8723)
##     No Information Rate : 0.8003
##     P-Value [Acc > NIR] : 3.095e-14
##
##                   Kappa : 0.4889
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##               Precision : 0.8778
##                  Recall : 0.9566
##                      F1 : 0.9155
##              Prevalence : 0.8003
##          Detection Rate : 0.7656
##    Detection Prevalence : 0.8722
##       Balanced Accuracy : 0.7114
##
##        'Positive' Class : no
##
```

```
# Accuracy : 0.8587
# Precision : 0.8778
# Recall : 0.9566
# F1 : 0.9162


# 3. AUC-----------------------------------------------------------------------------------
pred_dt_cv <- predict(dtFit_tune, test[-11], type = "prob")
colAUC(pred_dt_cv, test$Exited)
```

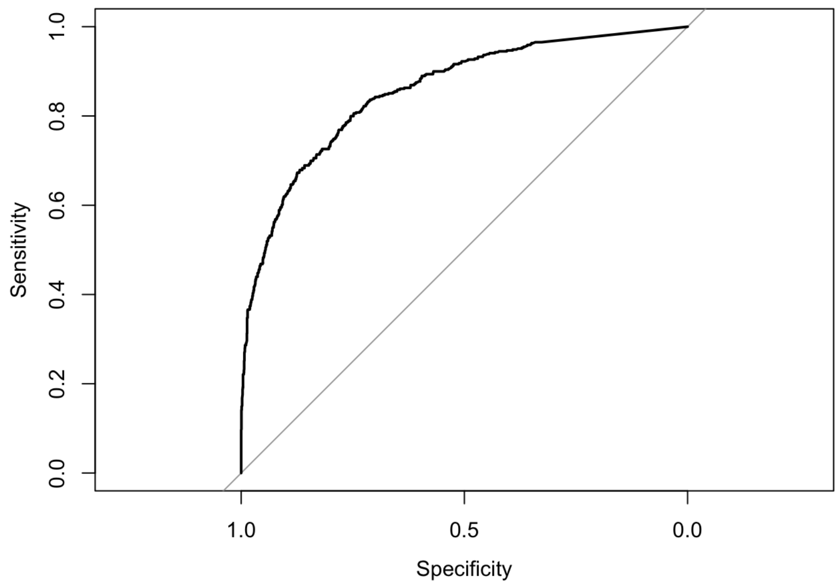
```
##                    no       yes
## no vs. yes 0.8556592 0.8556592
```

```
colAUC(pred_dt_cv, test$Exited, plotROC = TRUE)
```




```
##                    no       yes
## no vs. yes 0.8556592 0.8556592
```

```
auc_dt <- roc(response=test$Exited, predictor=pred_dt_cv[,1])
plot(auc_dt)
```




```
auc_dt$auc # AUC 85.57%
```

```
## Area under the curve: 0.8557
```

```
# Comment: k-fold cross validation did improve the model performance compare to holdout method. Tuning the parame
ters did not improve the k-fold cross validation method.
```

2. SVM

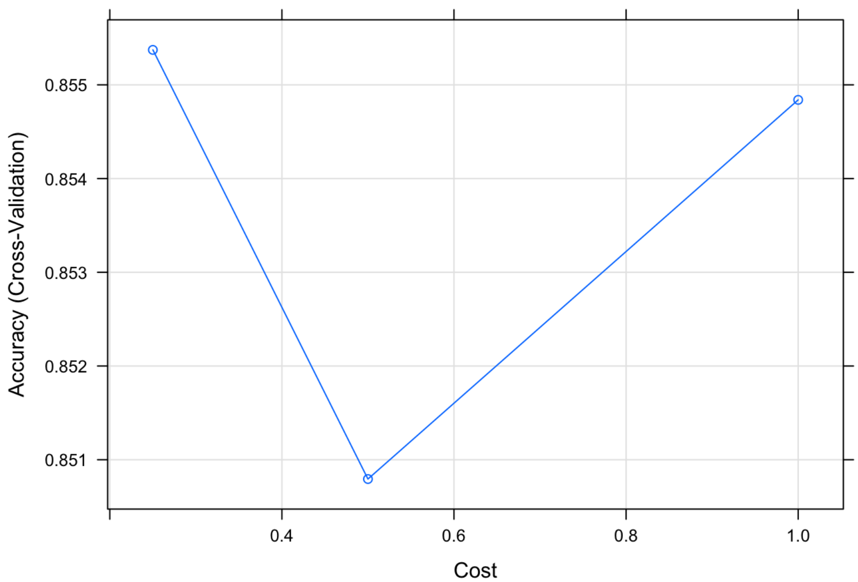
```
set.seed(1)

# SVM CV and Tuning

# 1. Evaluation of k-folds cross-validation-------------------------------------------
train <- svm_final_df[split,]
test <- svm_final_df[-split,]

# Create a control object that uses 10-fold cross validation
ctrl <- trainControl(method="cv", number=10, classProbs = TRUE)

svmFit_cv <- train(y~ ., data = train, method = "svmRadial",
                   trControl = ctrl, preProcess = c("center","scale"))
```

```
## line search fails -2.224957 -0.4069394 1.132849e-05 -6.518471e-06 -2.945737e-08 -1.491488e-08 -2.364854e-13lin
e search fails -2.552532 -0.7434487 1.246619e-05 -8.933555e-06 -3.202841e-08 -1.414955e-08 -2.728664e-13line sear
ch fails -2.286336 -0.4590789 1.100164e-05 -7.368471e-06 -2.595644e-08 -1.090137e-08 -2.052369e-13line search fai
ls -2.604757 -0.7718162 1.498019e-05 -9.761285e-06 -4.581502e-08 -2.433552e-08 -4.487719e-13line search fails -2.
346276 -0.50947 1.274194e-05 -9.079457e-06 -2.915443e-08 -1.124573e-08 -2.693788e-13line search fails -2.561674 -
0.7535846 2.377577e-05 -1.726806e-05 -6.063815e-08 -2.651396e-08 -9.838738e-13line search fails -2.603044 -0.7876
605 1.680958e-05 -1.210135e-05 -4.503906e-08 -2.088394e-08 -5.043637e-13
```

```
svmFit_cv # Cost parameter = 1.00  accuracy = 0.8527891 kappa = 0.4137305
```

```
## Support Vector Machines with Radial Basis Function Kernel
##
## 7350 samples
##   15 predictor
##    2 classes: 'no', 'yes'
##
## Pre-processing: centered (15), scaled (15)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 6616, 6615, 6615, 6614, 6615, 6615, ...
## Resampling results across tuning parameters:
##
##   C     Accuracy   Kappa
##   0.25  0.8553728  0.4644343
##   0.50  0.8507942  0.4344641
##   1.00  0.8548395  0.4432295
##
## Tuning parameter 'sigma' was held constant at a value of 0.04215211
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were sigma = 0.04215211 and C = 0.25.
```

```
plot(svmFit_cv)
```




```
# Predict testing set
p_svm <- predict(svmFit_cv, test[-16])
confusionMatrix(p_svm, test$y, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1880  268
##        yes   80  221
##
##                Accuracy : 0.8579
##                  95% CI : (0.8434, 0.8715)
##     No Information Rate : 0.8003
##     P-Value [Acc > NIR] : 7.141e-14
##
##                   Kappa : 0.4804
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##               Precision : 0.8752
##                  Recall : 0.9592
##                      F1 : 0.9153
##              Prevalence : 0.8003
##          Detection Rate : 0.7677
##    Detection Prevalence : 0.8771
##       Balanced Accuracy : 0.7056
##
##        'Positive' Class : no
##
```

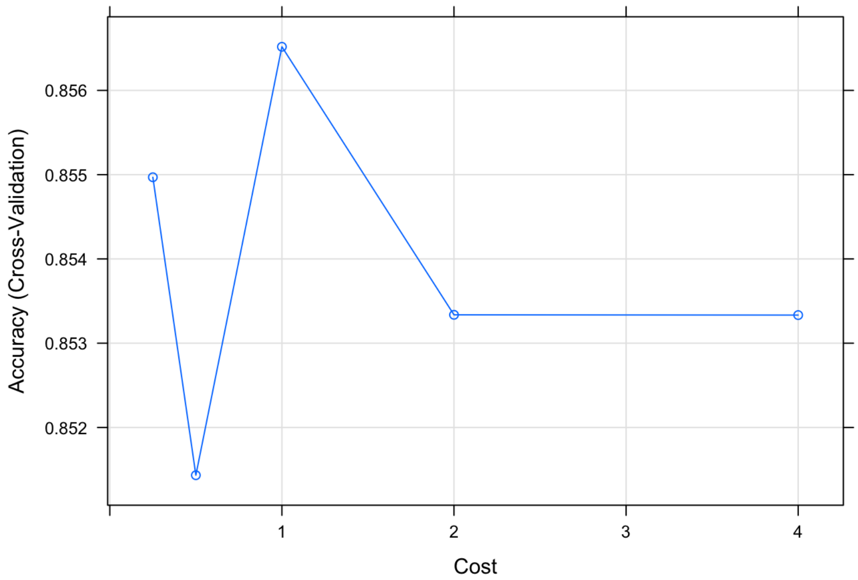
```
# Accuracy : 0.8579
# Precision : 0.8608
# Recall : 0.9811


# 2. Tuning of model------------------------------------------------------------------

svmFit_tune <- train(y ~ ., data = train, method = "svmRadial",
                     trControl = ctrl, preProcess = c("center","scale"),
                     metric = "ROC", tuneLength = 5, mode = "prec_recall")
```

```
## line search fails -2.557268 -0.7373992 1.588295e-05 -1.121988e-05 -4.262727e-08 -1.987492e-08 -4.540527e-13lin
e search fails -2.282037 -0.4563801 1.022401e-05 -6.753859e-06 -2.471233e-08 -1.085317e-08 -1.793582e-13line sear
ch fails -1.942558 -0.1452485 1.966157e-05 -1.289466e-05 -3.492063e-08 -9.710205e-09 -5.613847e-13line search fai
ls -2.522419 -0.7121024 1.806333e-05 -1.261992e-05 -4.753435e-08 -2.196979e-08 -5.813717e-13line search fails -2.
624628 -0.8041854 1.520834e-05 -1.125373e-05 -4.042016e-08 -1.833607e-08 -4.083744e-13line search fails -2.584894
-0.760945 1.436787e-05 -1.020536e-05 -3.907872e-08 -1.835519e-08 -3.741564e-13line search fails -1.918491 -0.1131
688 1.182858e-05 -7.94808e-06 -2.062597e-08 -5.348212e-09 -2.014679e-13line search fails -1.916621 -0.1253765 1.1
74893e-05 -7.818997e-06 -2.048405e-08 -5.434034e-09 -1.981769e-13line search fails -1.876527 -0.10071 1.366534e-0
5 -8.718469e-06 -2.391744e-08 -7.023157e-09 -2.656087e-13line search fails -2.559769 -0.7350673 2.473825e-05 -1.7
84633e-05 -6.369844e-08 -2.767537e-08 -1.081884e-12line search fails -1.877302 -0.08906397 1.200454e-05 -7.721373
e-06 -2.07815e-08 -5.81883e-09 -2.045428e-13
```

```
svmFit_tune # After tuning best model is Cost parameter = 2  accuracy = 0.8552371  kappa = 0.4533225
```

```
## Support Vector Machines with Radial Basis Function Kernel
##
## 7350 samples
##   15 predictor
##    2 classes: 'no', 'yes'
##
## Pre-processing: centered (15), scaled (15)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 6615, 6614, 6615, 6615, 6615, 6616, ...
## Resampling results across tuning parameters:
##
##   C     Accuracy   Kappa
##   0.25  0.8549693  0.4630186
##   0.50  0.8514333  0.4278080
##   1.00  0.8565174  0.4503513
##   2.00  0.8533366  0.4381417
##   4.00  0.8533333  0.4403914
##
## Tuning parameter 'sigma' was held constant at a value of 0.04320167
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were sigma = 0.04320167 and C = 1.
```

```
plot(svmFit_tune)
```



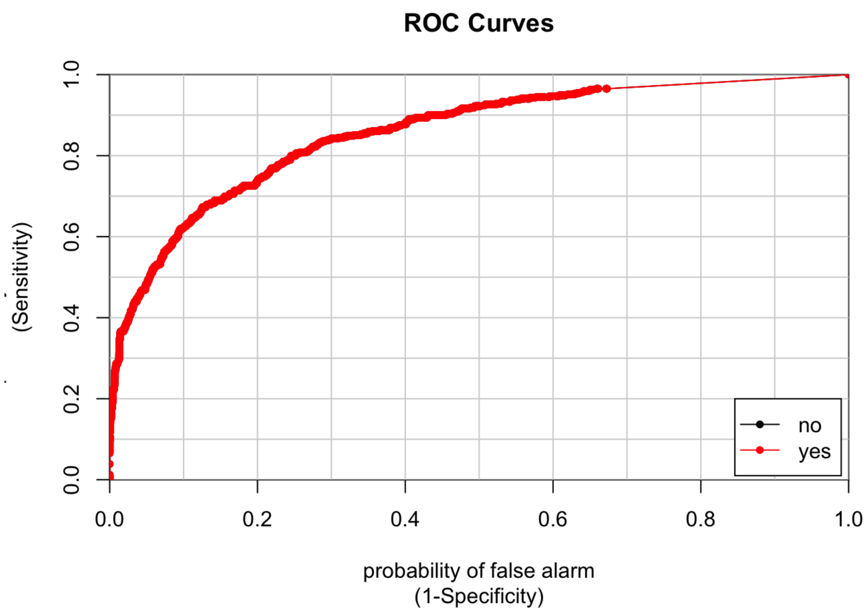

```
# Predict testing set
p_svm_tune <- predict(svmFit_tune, test[-16])
confusionMatrix(p_svm_tune, test$y, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1904  295
##        yes   56  194
##
##                Accuracy : 0.8567
##                  95% CI : (0.8422, 0.8703)
##     No Information Rate : 0.8003
##     P-Value [Acc > NIR] : 2.442e-13
##
##                   Kappa : 0.4508
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##               Precision : 0.8658
##                  Recall : 0.9714
##                      F1 : 0.9156
##              Prevalence : 0.8003
##          Detection Rate : 0.7775
##    Detection Prevalence : 0.8979
##       Balanced Accuracy : 0.6841
##
##        'Positive' Class : no
##
```

```
# Accuracy : 0.8591
# Precision : 0.8682
# Recall : 0.9714
# F1 : 0.9169

# 3. AUC------------------------------------------------------------------------------------
pred_svm_cv <- predict(svmFit_tune, test[-16], type = "prob")
colAUC(pred_svm_cv, test$y)
```

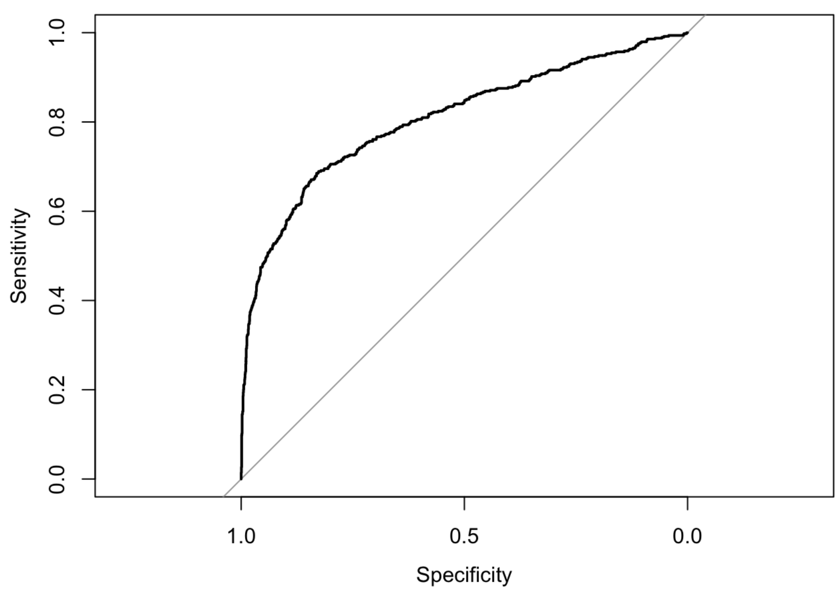
```
##                   no       yes
## no vs. yes 0.8076583 0.8076583
```

```
colAUC(pred_dt_cv, test$y, plotROC = TRUE)
```




```
##                   no       yes
## no vs. yes 0.8556592 0.8556592
```

```
auc_svm <- roc(response=test$y, predictor=pred_svm_cv[,1])
plot(auc_svm)
```




```
auc_svm$auc # AUC 81.07%
```

```
## Area under the curve: 0.8077
```

```
# Comment: k-fold cross validation did not improve the model performance compare to holdout method. Tuning the pa
rameters did improve the k-fold cross validation method.
```

3. Neural Network

```
set.seed(1)

# Neural Network CV and Tuning

# 1. Evaluation of k-folds cross-validation--------------------------------------------
train <- nn_df[split,]
test <- nn_df[-split,]

# Create a control object that uses 10-fold cross validation
ctrl <- trainControl(method="cv", number=10, classProbs = TRUE)

nnFit_cv <- train(y ~ ., data = train, method = "nnet", metric = "ROC",
                  trControl = ctrl, preProcess = c("center","scale"))
```

```
## # weights:  9
## initial  value 4129.335656
## iter  10 value 2930.582291
## iter  20 value 2838.375747
## iter  30 value 2837.459079
## final  value 2837.456903
## converged
## # weights:  25
## initial  value 7138.466066
## iter  10 value 2887.821117
## iter  20 value 2699.477166
## iter  30 value 2536.736406
## iter  40 value 2487.233982
## iter  50 value 2483.208641
## iter  60 value 2476.919006
## iter  70 value 2472.068799
## iter  80 value 2469.458414
## iter  90 value 2467.949952
## iter 100 value 2467.435854
## final  value 2467.435854
## stopped after 100 iterations
## # weights:  41
## initial  value 5415.768682
## iter  10 value 2686.560090
## iter  20 value 2491.268362
## iter  30 value 2454.104489
## iter  40 value 2442.316841
## iter  50 value 2435.365340
## iter  60 value 2431.826271
## iter  70 value 2428.544292
## iter  80 value 2416.979696
## iter  90 value 2411.193580
## iter 100 value 2410.255082
## final  value 2410.255082
## stopped after 100 iterations
## # weights:  9
## initial  value 3652.866018
## iter  10 value 2948.726328
## iter  20 value 2840.167270
## iter  30 value 2839.195186
## final  value 2839.188799
## converged
## # weights:  25
## initial  value 4022.642634
## iter  10 value 2843.221443
## iter  20 value 2742.393599
## iter  30 value 2691.087289
## iter  40 value 2631.404544
## iter  50 value 2559.636581
## iter  60 value 2531.271213
## iter  70 value 2516.903350
## iter  80 value 2512.536424
## iter  90 value 2512.132758
## iter  90 value 2512.132738
## iter  90 value 2512.132738
## final  value 2512.132738
## converged
## # weights:  41
## initial  value 6966.016678
## iter  10 value 2774.146834
## iter  20 value 2685.658828
## iter  30 value 2652.319533
## iter  40 value 2564.712164
## iter  50 value 2494.025988
## iter  60 value 2480.961517
## iter  70 value 2469.749185
## iter  80 value 2456.115604
## iter  90 value 2448.531580
## iter 100 value 2443.341131
## final  value 2443.341131
## stopped after 100 iterations
## # weights:  9
## initial  value 3807.573169
## iter  10 value 2911.759220
## iter  20 value 2860.022228
## iter  30 value 2843.863823
## iter  40 value 2837.500237
## final  value 2837.458793
```

```
## converged
## # weights:  25
## initial  value 4043.708589
## iter  10 value 2809.110529
## iter  20 value 2649.235019
## iter  30 value 2525.761946
## iter  40 value 2460.204220
## iter  50 value 2446.303017
## iter  60 value 2445.448676
## iter  70 value 2445.039411
## iter  80 value 2443.701859
## iter  90 value 2438.517086
## iter 100 value 2436.833328
## final  value 2436.833328
## stopped after 100 iterations
## # weights:  41
## initial  value 3686.784914
## iter  10 value 2833.555137
## iter  20 value 2728.056760
## iter  30 value 2643.434203
## iter  40 value 2573.726484
## iter  50 value 2539.638436
## iter  60 value 2483.899728
## iter  70 value 2459.781317
## iter  80 value 2435.180481
## iter  90 value 2431.223707
## iter 100 value 2426.155556
## final  value 2426.155556
## stopped after 100 iterations
## # weights:  9
## initial  value 4136.822190
## iter  10 value 3038.282064
## iter  20 value 2842.798707
## iter  30 value 2837.919630
## final  value 2837.710409
## converged
## # weights:  25
## initial  value 3250.651005
## iter  10 value 2728.247141
## iter  20 value 2500.018368
## iter  30 value 2483.723331
## iter  40 value 2464.915113
## iter  50 value 2452.677426
## iter  60 value 2445.807046
## iter  70 value 2445.237461
## iter  80 value 2444.773276
## iter  90 value 2444.084787
## final  value 2444.047777
## converged
## # weights:  41
## initial  value 4642.553146
## iter  10 value 2725.852181
## iter  20 value 2602.989018
## iter  30 value 2531.911775
## iter  40 value 2482.363287
## iter  50 value 2463.172116
## iter  60 value 2453.892328
## iter  70 value 2445.998879
## iter  80 value 2444.329091
## iter  90 value 2441.793357
## iter 100 value 2440.730878
## final  value 2440.730878
## stopped after 100 iterations
## # weights:  9
## initial  value 3448.576188
## iter  10 value 2881.691362
## iter  20 value 2849.253899
## iter  30 value 2839.769003
## final  value 2839.462589
## converged
## # weights:  25
## initial  value 4938.806793
## iter  10 value 2782.097440
## iter  20 value 2614.213194
## iter  30 value 2554.925018
## iter  40 value 2494.706584
## iter  50 value 2480.133338
## iter  60 value 2475.767156
## iter  70 value 2471.668313
```

```
## iter  80 value 2466.219826
## iter  90 value 2460.458768
## final  value 2460.456998
## converged
## # weights:  41
## initial  value 6642.997952
## iter  10 value 2755.127809
## iter  20 value 2683.191651
## iter  30 value 2599.300706
## iter  40 value 2524.008540
## iter  50 value 2480.252389
## iter  60 value 2460.658313
## iter  70 value 2453.948946
## iter  80 value 2449.324505
## iter  90 value 2445.563509
## iter 100 value 2442.044874
## final  value 2442.044874
## stopped after 100 iterations
## # weights:  9
## initial  value 4733.913160
## iter  10 value 2861.884575
## iter  20 value 2840.007993
## iter  30 value 2837.880326
## final  value 2837.712196
## converged
## # weights:  25
## initial  value 3901.299224
## iter  10 value 2846.999214
## iter  20 value 2794.094910
## iter  30 value 2719.477954
## iter  40 value 2643.392256
## iter  50 value 2597.435622
## iter  60 value 2585.495959
## iter  70 value 2554.693555
## iter  80 value 2537.739832
## iter  90 value 2532.778809
## iter 100 value 2530.951609
## final  value 2530.951609
## stopped after 100 iterations
## # weights:  41
## initial  value 4944.714405
## iter  10 value 2723.987590
## iter  20 value 2606.571323
## iter  30 value 2544.992034
## iter  40 value 2475.778580
## iter  50 value 2454.701189
## iter  60 value 2446.871713
## iter  70 value 2441.211612
## iter  80 value 2438.191202
## iter  90 value 2437.444823
## iter 100 value 2437.062203
## final  value 2437.062203
## stopped after 100 iterations
## # weights:  9
## initial  value 4483.646994
## iter  10 value 3059.426147
## iter  20 value 2856.342833
## iter  30 value 2855.685702
## final  value 2855.674733
## converged
## # weights:  25
## initial  value 5698.713525
## iter  10 value 2846.109864
## iter  20 value 2784.651820
## iter  30 value 2723.286823
## iter  40 value 2655.304930
## iter  50 value 2555.671803
## iter  60 value 2480.600970
## iter  70 value 2464.529679
## iter  80 value 2451.764381
## iter  90 value 2448.817082
## iter 100 value 2448.617113
## final  value 2448.617113
## stopped after 100 iterations
## # weights:  41
## initial  value 4029.507811
## iter  10 value 2736.988087
## iter  20 value 2592.018651
## iter  30 value 2480.509680
```

```
## iter  40 value 2455.493564
## iter  50 value 2444.872245
## iter  60 value 2443.102473
## iter  70 value 2442.240892
## iter  80 value 2440.319036
## iter  90 value 2437.180595
## iter 100 value 2435.343573
## final  value 2435.343573
## stopped after 100 iterations
## # weights:  9
## initial  value 5023.298597
## iter  10 value 3203.117053
## iter  20 value 3120.047129
## iter  30 value 2895.309768
## iter  40 value 2860.154270
## iter  50 value 2858.022867
## final  value 2858.022290
## converged
## # weights:  25
## initial  value 5927.490582
## iter  10 value 2846.657589
## iter  20 value 2792.684268
## iter  30 value 2711.505863
## iter  40 value 2641.426984
## iter  50 value 2595.783759
## iter  60 value 2519.580595
## iter  70 value 2494.403354
## iter  80 value 2469.271074
## iter  90 value 2462.378788
## iter 100 value 2458.323397
## final  value 2458.323397
## stopped after 100 iterations
## # weights:  41
## initial  value 4998.440499
## iter  10 value 2766.363116
## iter  20 value 2663.964530
## iter  30 value 2614.898030
## iter  40 value 2535.407606
## iter  50 value 2480.888933
## iter  60 value 2469.812194
## iter  70 value 2463.369627
## iter  80 value 2449.456844
## iter  90 value 2444.087551
## iter 100 value 2439.779850
## final  value 2439.779850
## stopped after 100 iterations
## # weights:  9
## initial  value 4671.548908
## iter  10 value 3299.662962
## iter  20 value 2892.147271
## iter  30 value 2858.207242
## iter  40 value 2855.709171
## final  value 2855.676442
## converged
## # weights:  25
## initial  value 3369.963783
## iter  10 value 2758.224384
## iter  20 value 2682.273792
## iter  30 value 2617.309088
## iter  40 value 2496.164033
## iter  50 value 2460.251440
## iter  60 value 2455.285669
## iter  70 value 2451.761769
## iter  80 value 2446.757411
## iter  90 value 2445.486664
## iter 100 value 2444.555600
## final  value 2444.555600
## stopped after 100 iterations
## # weights:  41
## initial  value 5163.008616
## iter  10 value 2685.672610
## iter  20 value 2527.907992
## iter  30 value 2500.691125
## iter  40 value 2483.911708
## iter  50 value 2472.858070
## iter  60 value 2468.932630
## iter  70 value 2466.321105
## iter  80 value 2465.312731
## iter  90 value 2465.142810
```

```
## iter 100 value 2464.478007
## final  value 2464.478007
## stopped after 100 iterations
## # weights:  9
## initial  value 3542.520783
## iter  10 value 2912.808135
## iter  20 value 2845.475057
## iter  30 value 2842.232920
## final  value 2841.955639
## converged
## # weights:  25
## initial  value 5576.674481
## iter  10 value 2837.409927
## iter  20 value 2730.501237
## iter  30 value 2715.517631
## iter  40 value 2704.274785
## iter  50 value 2703.011529
## iter  60 value 2686.534727
## iter  70 value 2672.131310
## iter  80 value 2629.528962
## iter  90 value 2611.731855
## iter 100 value 2608.597900
## final  value 2608.597900
## stopped after 100 iterations
## # weights:  41
## initial  value 3592.917270
## iter  10 value 2703.701047
## iter  20 value 2504.751330
## iter  30 value 2446.151404
## iter  40 value 2431.935395
## iter  50 value 2426.227099
## iter  60 value 2422.317597
## iter  70 value 2420.796230
## iter  80 value 2418.666554
## iter  90 value 2416.134710
## iter 100 value 2414.828001
## final  value 2414.828001
## stopped after 100 iterations
## # weights:  9
## initial  value 5670.110733
## iter  10 value 3171.079711
## iter  20 value 3082.824500
## iter  30 value 2877.012998
## iter  40 value 2854.850328
## iter  50 value 2844.579184
## final  value 2844.472287
## converged
## # weights:  25
## initial  value 4110.694710
## iter  10 value 2828.288409
## iter  20 value 2774.974968
## iter  30 value 2725.738472
## iter  40 value 2614.240524
## iter  50 value 2493.571767
## iter  60 value 2483.075273
## iter  70 value 2474.979785
## iter  80 value 2464.739803
## iter  90 value 2460.759371
## iter 100 value 2460.671514
## final  value 2460.671514
## stopped after 100 iterations
## # weights:  41
## initial  value 3653.488087
## iter  10 value 2721.034327
## iter  20 value 2611.560932
## iter  30 value 2543.796388
## iter  40 value 2490.327822
## iter  50 value 2469.042665
## iter  60 value 2453.545707
## iter  70 value 2440.649555
## iter  80 value 2436.375024
## iter  90 value 2431.783857
## iter 100 value 2427.664211
## final  value 2427.664211
## stopped after 100 iterations
## # weights:  9
## initial  value 4653.806553
## iter  10 value 2860.250453
## iter  20 value 2844.007376
```

```
## iter  30 value 2841.985830
## final  value 2841.957592
## converged
## # weights:  25
## initial  value 4327.927093
## iter  10 value 2855.064675
## iter  20 value 2700.484485
## iter  30 value 2654.518981
## iter  40 value 2602.723690
## iter  50 value 2556.604673
## iter  60 value 2535.834192
## iter  70 value 2508.725388
## iter  80 value 2489.298711
## iter  90 value 2482.543171
## iter 100 value 2482.159453
## final  value 2482.159453
## stopped after 100 iterations
## # weights:  41
## initial  value 5108.281487
## iter  10 value 2740.182308
## iter  20 value 2586.810780
## iter  30 value 2526.373543
## iter  40 value 2474.976860
## iter  50 value 2438.758202
## iter  60 value 2434.068340
## iter  70 value 2432.527477
## iter  80 value 2431.104427
## iter  90 value 2430.071103
## iter 100 value 2429.868731
## final  value 2429.868731
## stopped after 100 iterations
## # weights:  9
## initial  value 3930.235324
## iter  10 value 3023.055795
## iter  20 value 2856.569405
## iter  30 value 2846.469416
## iter  40 value 2843.424675
## final  value 2843.424564
## converged
## # weights:  25
## initial  value 4754.282561
## iter  10 value 2789.866657
## iter  20 value 2568.012572
## iter  30 value 2511.517225
## iter  40 value 2504.019831
## iter  50 value 2503.045939
## iter  60 value 2502.965262
## iter  70 value 2502.894060
## iter  80 value 2502.510685
## iter  90 value 2501.386271
## iter 100 value 2470.401129
## final  value 2470.401129
## stopped after 100 iterations
## # weights:  41
## initial  value 3379.021157
## iter  10 value 2704.852909
## iter  20 value 2606.629758
## iter  30 value 2522.508918
## iter  40 value 2469.244663
## iter  50 value 2441.381156
## iter  60 value 2426.632817
## iter  70 value 2421.694320
## iter  80 value 2418.353655
## iter  90 value 2416.409387
## iter 100 value 2414.382669
## final  value 2414.382669
## stopped after 100 iterations
## # weights:  9
## initial  value 4070.985456
## iter  10 value 2898.595605
## iter  20 value 2853.643028
## iter  30 value 2846.149035
## final  value 2845.083439
## converged
## # weights:  25
## initial  value 5968.339753
## iter  10 value 2726.963796
## iter  20 value 2606.657946
## iter  30 value 2542.762373
```

```
## iter  40 value 2503.959403
## iter  50 value 2486.545555
## iter  60 value 2477.668039
## iter  70 value 2471.397401
## iter  80 value 2461.019923
## final  value 2460.863964
## converged
## # weights:  41
## initial  value 5845.779839
## iter  10 value 2791.966138
## iter  20 value 2693.761835
## iter  30 value 2641.503558
## iter  40 value 2567.930600
## iter  50 value 2507.838400
## iter  60 value 2482.434473
## iter  70 value 2472.123966
## iter  80 value 2466.227329
## iter  90 value 2463.100410
## iter 100 value 2457.589639
## final  value 2457.589639
## stopped after 100 iterations
## # weights:  9
## initial  value 5157.227551
## iter  10 value 2877.866662
## iter  20 value 2850.043433
## iter  30 value 2843.656312
## final  value 2843.426070
## converged
## # weights:  25
## initial  value 4343.742921
## iter  10 value 2797.846994
## iter  20 value 2713.319839
## iter  30 value 2648.949035
## iter  40 value 2533.717088
## iter  50 value 2517.089363
## iter  60 value 2508.452358
## iter  70 value 2492.585801
## iter  80 value 2472.044163
## iter  90 value 2460.484448
## iter 100 value 2454.506769
## final  value 2454.506769
## stopped after 100 iterations
## # weights:  41
## initial  value 4561.579973
## iter  10 value 2780.966264
## iter  20 value 2648.609251
## iter  30 value 2568.384536
## iter  40 value 2482.850245
## iter  50 value 2461.604481
## iter  60 value 2442.372332
## iter  70 value 2434.372900
## iter  80 value 2431.406421
## iter  90 value 2430.409249
## iter 100 value 2429.886538
## final  value 2429.886538
## stopped after 100 iterations
## # weights:  9
## initial  value 6188.571200
## iter  10 value 3308.806156
## iter  20 value 2943.996730
## iter  30 value 2880.454121
## iter  40 value 2879.802628
## iter  50 value 2877.304074
## iter  60 value 2871.680322
## iter  70 value 2851.175312
## iter  80 value 2837.246167
## final  value 2837.128571
## converged
## # weights:  25
## initial  value 5335.128292
## iter  10 value 2969.018749
## iter  20 value 2817.469231
## iter  30 value 2800.041950
## iter  40 value 2717.496760
## iter  50 value 2698.530833
## iter  60 value 2696.821318
## iter  70 value 2695.145679
## iter  80 value 2689.035338
## iter  90 value 2686.926952
```

```
## iter 100 value 2686.734580
## final  value 2686.734580
## stopped after 100 iterations
## # weights:  41
## initial  value 3777.534779
## iter  10 value 2797.690134
## iter  20 value 2704.525125
## iter  30 value 2626.273621
## iter  40 value 2587.672884
## iter  50 value 2580.219650
## iter  60 value 2578.222134
## iter  70 value 2576.068531
## iter  80 value 2555.522175
## iter  90 value 2474.059356
## iter 100 value 2448.686320
## final  value 2448.686320
## stopped after 100 iterations
## # weights:  9
## initial  value 6232.098899
## iter  10 value 3292.063378
## iter  20 value 3144.917102
## iter  30 value 2857.389206
## iter  40 value 2847.149385
## iter  50 value 2839.436079
## final  value 2839.423400
## converged
## # weights:  25
## initial  value 6140.751103
## iter  10 value 2853.440913
## iter  20 value 2665.566852
## iter  30 value 2552.814676
## iter  40 value 2496.340866
## iter  50 value 2477.380503
## iter  60 value 2474.882124
## iter  70 value 2470.249685
## iter  80 value 2456.678574
## iter  90 value 2442.445614
## iter 100 value 2441.901852
## final  value 2441.901852
## stopped after 100 iterations
## # weights:  41
## initial  value 4890.470845
## iter  10 value 2824.683406
## iter  20 value 2598.361095
## iter  30 value 2513.785913
## iter  40 value 2481.307807
## iter  50 value 2464.298631
## iter  60 value 2456.213222
## iter  70 value 2448.511878
## iter  80 value 2444.873165
## iter  90 value 2443.265533
## iter 100 value 2441.654500
## final  value 2441.654500
## stopped after 100 iterations
## # weights:  9
## initial  value 5695.796395
## iter  10 value 2891.287866
## iter  20 value 2846.613179
## iter  30 value 2838.667751
## final  value 2837.130181
## converged
## # weights:  25
## initial  value 3423.032490
## iter  10 value 2821.954110
## iter  20 value 2766.983034
## iter  30 value 2733.482677
## iter  40 value 2708.154198
## iter  50 value 2691.942012
## iter  60 value 2691.412243
## iter  70 value 2690.980239
## iter  80 value 2689.086355
## iter  90 value 2688.756975
## iter 100 value 2688.612101
## final  value 2688.612101
## stopped after 100 iterations
## # weights:  41
## initial  value 6051.599586
## iter  10 value 2730.375667
## iter  20 value 2600.620385
```

```
## iter  30 value 2525.258441
## iter  40 value 2466.129231
## iter  50 value 2441.216664
## iter  60 value 2431.203973
## iter  70 value 2421.797344
## iter  80 value 2414.999365
## iter  90 value 2412.494363
## iter 100 value 2404.850655
## final  value 2404.850655
## stopped after 100 iterations
## # weights:  9
## initial  value 4250.148786
## iter  10 value 2867.393100
## iter  20 value 2848.510512
## iter  30 value 2847.956120
## final  value 2847.940768
## converged
## # weights:  25
## initial  value 3790.239713
## iter  10 value 2829.767990
## iter  20 value 2722.205668
## iter  30 value 2692.386466
## iter  40 value 2636.485367
## iter  50 value 2594.167048
## iter  60 value 2588.940396
## iter  70 value 2581.237882
## iter  80 value 2541.973296
## iter  90 value 2505.262128
## iter 100 value 2475.860931
## final  value 2475.860931
## stopped after 100 iterations
## # weights:  41
## initial  value 6049.954476
## iter  10 value 2768.114458
## iter  20 value 2664.793723
## iter  30 value 2599.664938
## iter  40 value 2540.459300
## iter  50 value 2512.694080
## iter  60 value 2482.359481
## iter  70 value 2468.517963
## iter  80 value 2462.717495
## iter  90 value 2461.455399
## iter 100 value 2460.825276
## final  value 2460.825276
## stopped after 100 iterations
## # weights:  9
## initial  value 4846.561175
## iter  10 value 2875.009088
## iter  20 value 2850.736250
## iter  30 value 2849.593979
## final  value 2849.504600
## converged
## # weights:  25
## initial  value 4587.337501
## iter  10 value 2811.701746
## iter  20 value 2761.035848
## iter  30 value 2569.332353
## iter  40 value 2505.354702
## iter  50 value 2475.406226
## iter  60 value 2465.751139
## iter  70 value 2460.058753
## iter  80 value 2453.830435
## iter  90 value 2448.631917
## final  value 2448.597901
## converged
## # weights:  41
## initial  value 4051.600476
## iter  10 value 2901.052816
## iter  20 value 2840.884154
## iter  30 value 2787.151488
## iter  40 value 2748.830929
## iter  50 value 2667.799295
## iter  60 value 2518.068963
## iter  70 value 2461.729057
## iter  80 value 2444.781666
## iter  90 value 2438.483531
## iter 100 value 2437.693287
## final  value 2437.693287
## stopped after 100 iterations
```

```
## # weights:  9
## initial  value 3981.259217
## iter  10 value 2999.011200
## iter  20 value 2883.990040
## iter  30 value 2855.377388
## iter  40 value 2847.944697
## final  value 2847.943082
## converged
## # weights:  25
## initial  value 4403.397886
## iter  10 value 2697.748609
## iter  20 value 2600.860503
## iter  30 value 2520.932098
## iter  40 value 2490.737583
## iter  50 value 2484.386213
## iter  60 value 2481.499073
## iter  70 value 2474.517364
## iter  80 value 2464.451473
## iter  90 value 2461.410685
## iter 100 value 2457.408957
## final  value 2457.408957
## stopped after 100 iterations
## # weights:  41
## initial  value 5101.576453
## iter  10 value 2825.115509
## iter  20 value 2662.945763
## iter  30 value 2623.036929
## iter  40 value 2600.207760
## iter  50 value 2575.755528
## iter  60 value 2526.692743
## iter  70 value 2459.444579
## iter  80 value 2432.056293
## iter  90 value 2427.482210
## iter 100 value 2424.613345
## final  value 2424.613345
## stopped after 100 iterations
## # weights:  9
## initial  value 5077.536745
## iter  10 value 2876.440821
## iter  20 value 2835.115193
## iter  30 value 2833.787973
## final  value 2833.777650
## converged
## # weights:  25
## initial  value 4530.051715
## iter  10 value 2846.633831
## iter  20 value 2824.508377
## iter  30 value 2798.115459
## iter  40 value 2719.183589
## iter  50 value 2619.373944
## iter  60 value 2593.983067
## iter  70 value 2571.149733
## iter  80 value 2528.630984
## iter  90 value 2482.996095
## iter 100 value 2479.185031
## final  value 2479.185031
## stopped after 100 iterations
## # weights:  41
## initial  value 3386.212384
## iter  10 value 2720.231545
## iter  20 value 2610.777351
## iter  30 value 2556.416452
## iter  40 value 2503.524821
## iter  50 value 2468.596839
## iter  60 value 2455.934705
## iter  70 value 2449.733732
## iter  80 value 2445.803987
## iter  90 value 2443.955131
## iter 100 value 2442.176572
## final  value 2442.176572
## stopped after 100 iterations
## # weights:  9
## initial  value 5476.547420
## iter  10 value 2949.957680
## iter  20 value 2860.101808
## iter  30 value 2839.236844
## iter  40 value 2835.401671
## final  value 2835.401095
## converged
```

```
## # weights:  25
## initial  value 4172.466851
## iter  10 value 2767.096003
## iter  20 value 2691.995046
## iter  30 value 2602.185748
## iter  40 value 2522.303586
## iter  50 value 2484.541672
## iter  60 value 2476.639052
## iter  70 value 2471.835470
## iter  80 value 2467.922010
## iter  90 value 2465.330142
## final  value 2465.311899
## converged
## # weights:  41
## initial  value 3364.005642
## iter  10 value 2813.350943
## iter  20 value 2729.238907
## iter  30 value 2689.219810
## iter  40 value 2633.600427
## iter  50 value 2524.561391
## iter  60 value 2492.456339
## iter  70 value 2479.806462
## iter  80 value 2475.088391
## iter  90 value 2466.327178
## iter 100 value 2460.224145
## final  value 2460.224145
## stopped after 100 iterations
## # weights:  9
## initial  value 3703.415383
## iter  10 value 2861.509391
## iter  20 value 2834.544055
## iter  30 value 2833.842595
## final  value 2833.779289
## converged
## # weights:  25
## initial  value 4784.664880
## iter  10 value 2899.887097
## iter  20 value 2731.027008
## iter  30 value 2699.817755
## iter  40 value 2590.389655
## iter  50 value 2544.979488
## iter  60 value 2504.236977
## iter  70 value 2494.099677
## iter  80 value 2482.651700
## iter  90 value 2480.938444
## iter 100 value 2480.273051
## final  value 2480.273051
## stopped after 100 iterations
## # weights:  41
## initial  value 4783.193647
## iter  10 value 2714.633926
## iter  20 value 2542.871939
## iter  30 value 2504.998801
## iter  40 value 2471.582690
## iter  50 value 2450.880740
## iter  60 value 2445.953293
## iter  70 value 2441.400565
## iter  80 value 2439.362804
## iter  90 value 2438.629533
## iter 100 value 2438.280524
## final  value 2438.280524
## stopped after 100 iterations
## # weights:  9
## initial  value 3681.662888
## iter  10 value 2900.830729
## iter  20 value 2873.866051
## iter  30 value 2859.735196
## iter  40 value 2843.791260
## iter  50 value 2843.677087
## final  value 2843.676829
## converged
## # weights:  25
## initial  value 4593.404519
## iter  10 value 2680.757234
## iter  20 value 2571.614005
## iter  30 value 2477.579927
## iter  40 value 2447.535968
## iter  50 value 2442.203302
## iter  60 value 2440.010719
```

```
## iter  70 value 2438.686024
## iter  80 value 2436.640165
## iter  90 value 2435.403289
## iter 100 value 2435.148569
## final  value 2435.148569
## stopped after 100 iterations
## # weights:  41
## initial  value 4055.726032
## iter  10 value 2837.577158
## iter  20 value 2789.847271
## iter  30 value 2697.873015
## iter  40 value 2645.109060
## iter  50 value 2605.660410
## iter  60 value 2520.282615
## iter  70 value 2444.694365
## iter  80 value 2425.867521
## iter  90 value 2423.273204
## iter 100 value 2421.728418
## final  value 2421.728418
## stopped after 100 iterations
## # weights:  9
## initial  value 3937.009537
## iter  10 value 2940.849652
## iter  20 value 2845.995728
## iter  30 value 2845.358416
## final  value 2845.344174
## converged
## # weights:  25
## initial  value 6832.980453
## iter  10 value 2888.305476
## iter  20 value 2842.648245
## iter  30 value 2830.534149
## iter  40 value 2784.208570
## iter  50 value 2615.187936
## iter  60 value 2512.280910
## iter  70 value 2489.733996
## iter  80 value 2476.971001
## iter  90 value 2473.772780
## iter 100 value 2472.680577
## final  value 2472.680577
## stopped after 100 iterations
## # weights:  41
## initial  value 3867.544390
## iter  10 value 2780.692691
## iter  20 value 2694.018093
## iter  30 value 2664.285927
## iter  40 value 2630.174913
## iter  50 value 2540.164281
## iter  60 value 2475.210835
## iter  70 value 2458.596980
## iter  80 value 2448.694179
## iter  90 value 2445.555703
## iter 100 value 2442.018055
## final  value 2442.018055
## stopped after 100 iterations
## # weights:  9
## initial  value 5558.164908
## iter  10 value 2877.875185
## iter  20 value 2845.374275
## iter  30 value 2843.693695
## final  value 2843.678518
## converged
## # weights:  25
## initial  value 7530.850050
## iter  10 value 2822.407941
## iter  20 value 2689.734751
## iter  30 value 2612.262544
## iter  40 value 2544.154132
## iter  50 value 2503.701765
## iter  60 value 2498.660571
## iter  70 value 2485.076115
## iter  80 value 2473.141139
## iter  90 value 2471.401041
## iter 100 value 2469.358396
## final  value 2469.358396
## stopped after 100 iterations
## # weights:  41
## initial  value 3789.129651
## iter  10 value 2778.227077
```

```
## iter  20 value 2644.128597
## iter  30 value 2587.170320
## iter  40 value 2570.346601
## iter  50 value 2563.498742
## iter  60 value 2549.593288
## iter  70 value 2538.285002
## iter  80 value 2531.587801
## iter  90 value 2528.743160
## iter 100 value 2526.475452
## final  value 2526.475452
## stopped after 100 iterations
## # weights:  9
## initial  value 4502.261827
## iter  10 value 2908.987173
## iter  20 value 2860.313442
## final  value 2854.274894
## converged
## # weights:  25
## initial  value 4945.139341
## iter  10 value 2749.175677
## iter  20 value 2671.836821
## iter  30 value 2562.119746
## iter  40 value 2483.278625
## iter  50 value 2467.366322
## iter  60 value 2464.128881
## iter  70 value 2458.719612
## iter  80 value 2452.958654
## iter  90 value 2451.916864
## iter 100 value 2451.769798
## final  value 2451.769798
## stopped after 100 iterations
## # weights:  41
## initial  value 4293.305775
## iter  10 value 2704.648890
## iter  20 value 2582.135307
## iter  30 value 2498.092797
## iter  40 value 2450.311318
## iter  50 value 2431.726871
## iter  60 value 2424.895535
## iter  70 value 2419.857940
## iter  80 value 2417.483726
## iter  90 value 2415.235534
## iter 100 value 2414.504232
## final  value 2414.504232
## stopped after 100 iterations
## # weights:  9
## initial  value 4211.727693
## iter  10 value 2876.236783
## iter  20 value 2856.110623
## iter  30 value 2855.911321
## final  value 2855.910512
## converged
## # weights:  25
## initial  value 4264.108321
## iter  10 value 2838.895383
## iter  20 value 2786.334180
## iter  30 value 2654.110624
## iter  40 value 2619.207529
## iter  50 value 2574.081040
## iter  60 value 2545.470477
## iter  70 value 2517.141787
## iter  80 value 2488.349298
## iter  90 value 2472.714333
## iter 100 value 2465.694522
## final  value 2465.694522
## stopped after 100 iterations
## # weights:  41
## initial  value 4457.737004
## iter  10 value 2780.611174
## iter  20 value 2656.822358
## iter  30 value 2601.930981
## iter  40 value 2512.730038
## iter  50 value 2481.258386
## iter  60 value 2464.633476
## iter  70 value 2459.862163
## iter  80 value 2456.408478
## iter  90 value 2453.539502
## iter 100 value 2446.443204
## final  value 2446.443204
```

```
## stopped after 100 iterations
## # weights:  9
## initial  value 5407.968335
## iter  10 value 2915.722376
## iter  20 value 2859.210854
## iter  30 value 2854.834048
## final  value 2854.276551
## converged
## # weights:  25
## initial  value 5144.602684
## iter  10 value 2837.164902
## iter  20 value 2750.754857
## iter  30 value 2719.353997
## iter  40 value 2697.058960
## iter  50 value 2689.927673
## iter  60 value 2688.873843
## iter  70 value 2688.513365
## iter  80 value 2686.519015
## iter  90 value 2671.732025
## iter 100 value 2669.533095
## final  value 2669.533095
## stopped after 100 iterations
## # weights:  41
## initial  value 6362.315721
## iter  10 value 2777.985787
## iter  20 value 2668.125472
## iter  30 value 2628.196005
## iter  40 value 2579.808157
## iter  50 value 2531.874453
## iter  60 value 2468.540939
## iter  70 value 2428.692417
## iter  80 value 2421.941620
## iter  90 value 2421.705072
## iter 100 value 2421.465059
## final  value 2421.465059
## stopped after 100 iterations
## # weights:  41
## initial  value 4227.361483
## iter  10 value 3121.219838
## iter  20 value 2981.837188
## iter  30 value 2898.723770
## iter  40 value 2773.941063
## iter  50 value 2741.172714
## iter  60 value 2730.796474
## iter  70 value 2723.283799
## iter  80 value 2721.920580
## iter  90 value 2721.333267
## iter 100 value 2720.718885
## final  value 2720.718885
## stopped after 100 iterations
```

```
nnFit_cv # Best model is size = 5 and decay = 1e-04 accuracy 84.08%
```

```
## Neural Network
##
## 7350 samples
##    6 predictor
##    2 classes: 'no', 'yes'
##
## Pre-processing: centered (6), scaled (6)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 6616, 6615, 6615, 6614, 6615, 6615, ...
## Resampling results across tuning parameters:
##
##   size  decay  Accuracy   Kappa
##   1     0e+00  0.7917002  0.1631688
##   1     1e-04  0.7917002  0.1631688
##   1     1e-01  0.7912924  0.1598192
##   3     0e+00  0.8342881  0.3986396
##   3     1e-04  0.8326543  0.3900103
##   3     1e-01  0.8397310  0.4195130
##   5     0e+00  0.8408192  0.4293501
##   5     1e-04  0.8416344  0.4219078
##   5     1e-01  0.8390496  0.4127871
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were size = 5 and decay = 1e-04.
```

```
# Predict testing set
p_nn <- predict(nnFit_cv, test[-7])
confusionMatrix(p_nn, test$y, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1822  279
##        yes  138  210
##
##                Accuracy : 0.8297
##                  95% CI : (0.8142, 0.8444)
##     No Information Rate : 0.8003
##     P-Value [Acc > NIR] : 0.0001166
##
##                   Kappa : 0.4026
##
##  Mcnemar's Test P-Value : 7.09e-12
##
##               Precision : 0.8672
##                  Recall : 0.9296
##                      F1 : 0.8973
##              Prevalence : 0.8003
##          Detection Rate : 0.7440
##    Detection Prevalence : 0.8579
##       Balanced Accuracy : 0.6795
##
##        'Positive' Class : no
##
```

```
# Accuracy : 0.8297
# Precision : 0.8672
# Recall : 0.9296



# 2. Tuning of model---------------------------------------------------------------------
nnetGrid <-  expand.grid(size = seq(from = 1, to = 3, by = 1), # Number of nodes
                        decay = seq(from = 0.1, to = 0.2, by = 0.1))

nnFit_tune <- train(y ~ ., data = train, method = "nnet",
                     trControl = ctrl, preProcess = c("center","scale"),
                     metric = "ROC", tuneGrid = nnetGrid)
```

```
## # weights:  9
## initial  value 4669.851282
## iter  10 value 2942.089462
## iter  20 value 2846.709175
## iter  30 value 2844.190166
## final  value 2844.134343
## converged
## # weights:  17
## initial  value 5675.706082
## iter  10 value 2818.893897
## iter  20 value 2747.591529
## iter  30 value 2695.233697
## iter  40 value 2674.264840
## iter  50 value 2650.310505
## iter  60 value 2591.621596
## iter  70 value 2567.863068
## iter  80 value 2565.121605
## iter  90 value 2563.503152
## final  value 2563.411744
## converged
## # weights:  25
## initial  value 3679.396826
## iter  10 value 2875.366913
## iter  20 value 2833.899351
## iter  30 value 2814.600348
## iter  40 value 2806.192777
## iter  50 value 2716.754473
## iter  60 value 2513.563792
## iter  70 value 2472.982839
## iter  80 value 2453.602616
## iter  90 value 2450.103720
## iter 100 value 2447.559552
## final  value 2447.559552
## stopped after 100 iterations
## # weights:  9
## initial  value 4030.930246
## iter  10 value 2860.282545
## iter  20 value 2845.750554
## iter  30 value 2845.742443
## final  value 2845.740352
## converged
## # weights:  17
## initial  value 5325.744925
## iter  10 value 2911.683102
## iter  20 value 2794.563715
## iter  30 value 2759.406879
## iter  40 value 2755.793646
## iter  50 value 2755.528728
## final  value 2755.509552
## converged
## # weights:  25
## initial  value 4803.277691
## iter  10 value 2846.251396
## iter  20 value 2822.503053
## iter  30 value 2770.599753
## iter  40 value 2747.887941
## iter  50 value 2743.955555
## iter  60 value 2740.154479
## iter  70 value 2737.952239
## iter  80 value 2736.707485
## iter  90 value 2733.523162
## iter 100 value 2731.331571
## final  value 2731.331571
## stopped after 100 iterations
## # weights:  9
## initial  value 4421.109344
## iter  10 value 2902.415848
## iter  20 value 2851.779898
## iter  30 value 2842.935142
## final  value 2841.366446
## converged
## # weights:  17
## initial  value 5590.934959
## iter  10 value 2795.894165
## iter  20 value 2749.126552
## iter  30 value 2713.424079
## iter  40 value 2677.834096
## iter  50 value 2658.646319
```

```
## iter  60 value 2627.712424
## iter  70 value 2574.352668
## iter  80 value 2569.391752
## iter  90 value 2565.254621
## final  value 2565.054511
## converged
## # weights:  25
## initial  value 5789.807376
## iter  10 value 2748.766662
## iter  20 value 2636.596646
## iter  30 value 2531.370099
## iter  40 value 2491.183834
## iter  50 value 2467.536906
## iter  60 value 2462.547125
## iter  70 value 2454.010578
## iter  80 value 2445.845394
## iter  90 value 2445.182259
## final  value 2445.179267
## converged
## # weights:  9
## initial  value 4242.557936
## iter  10 value 2870.917881
## iter  20 value 2842.991701
## final  value 2842.964333
## converged
## # weights:  17
## initial  value 5947.145020
## iter  10 value 2879.587766
## iter  20 value 2838.299027
## iter  30 value 2825.343246
## iter  40 value 2761.521618
## iter  50 value 2711.505254
## iter  60 value 2667.604071
## iter  70 value 2649.882857
## iter  80 value 2627.602102
## iter  90 value 2597.211448
## iter 100 value 2595.292261
## final  value 2595.292261
## stopped after 100 iterations
## # weights:  25
## initial  value 7762.914695
## iter  10 value 2936.756529
## iter  20 value 2852.935640
## iter  30 value 2813.657422
## iter  40 value 2659.825302
## iter  50 value 2603.915004
## iter  60 value 2577.779413
## iter  70 value 2535.938017
## iter  80 value 2502.766804
## iter  90 value 2491.849095
## iter 100 value 2489.876045
## final  value 2489.876045
## stopped after 100 iterations
## # weights:  9
## initial  value 3506.992659
## iter  10 value 3162.108076
## iter  20 value 2879.778931
## iter  30 value 2856.768103
## iter  40 value 2850.472913
## final  value 2850.442901
## converged
## # weights:  17
## initial  value 3906.451252
## iter  10 value 2929.136256
## iter  20 value 2854.037452
## iter  30 value 2826.521966
## iter  40 value 2747.541780
## iter  50 value 2741.769309
## iter  60 value 2677.422890
## iter  70 value 2623.196148
## iter  80 value 2613.347380
## iter  90 value 2596.233157
## iter 100 value 2595.164025
## final  value 2595.164025
## stopped after 100 iterations
## # weights:  25
## initial  value 3649.029284
## iter  10 value 2885.111670
## iter  20 value 2801.240388
```

```
## iter  30 value 2744.840735
## iter  40 value 2715.937160
## iter  50 value 2701.134730
## iter  60 value 2700.038697
## iter  70 value 2603.299372
## iter  80 value 2548.556656
## iter  90 value 2525.869512
## iter 100 value 2495.770427
## final  value 2495.770427
## stopped after 100 iterations
## # weights:  9
## initial  value 5701.901058
## iter  10 value 2896.311345
## iter  20 value 2857.953741
## iter  30 value 2852.183503
## final  value 2852.036888
## converged
## # weights:  17
## initial  value 3489.533721
## iter  10 value 2784.908983
## iter  20 value 2748.537157
## iter  30 value 2693.316790
## iter  40 value 2672.916862
## iter  50 value 2654.997019
## iter  60 value 2622.922271
## iter  70 value 2620.441640
## final  value 2620.434974
## converged
## # weights:  25
## initial  value 4297.966890
## iter  10 value 2881.127793
## iter  20 value 2796.737844
## iter  30 value 2714.538034
## iter  40 value 2592.128629
## iter  50 value 2533.620423
## iter  60 value 2517.461428
## iter  70 value 2502.465620
## iter  80 value 2485.086060
## iter  90 value 2475.537376
## iter 100 value 2473.383465
## final  value 2473.383465
## stopped after 100 iterations
## # weights:  9
## initial  value 3481.569908
## iter  10 value 2869.388382
## iter  20 value 2832.003958
## iter  30 value 2831.770194
## final  value 2831.769764
## converged
## # weights:  17
## initial  value 4640.418509
## iter  10 value 2961.455365
## iter  20 value 2838.285902
## iter  30 value 2816.138362
## iter  40 value 2705.661947
## iter  50 value 2661.791454
## iter  60 value 2612.586063
## iter  70 value 2561.009631
## iter  80 value 2557.535940
## iter  90 value 2551.486976
## final  value 2551.417908
## converged
## # weights:  25
## initial  value 4120.418285
## iter  10 value 2731.493719
## iter  20 value 2647.886529
## iter  30 value 2608.176971
## iter  40 value 2571.630431
## iter  50 value 2518.352312
## iter  60 value 2472.183282
## iter  70 value 2445.665455
## iter  80 value 2435.974928
## iter  90 value 2433.232353
## final  value 2433.224107
## converged
## # weights:  9
## initial  value 5499.047685
## iter  10 value 2936.086368
## iter  20 value 2844.541465
```

```
## iter  30 value 2833.433308
## final  value 2833.418686
## converged
## # weights:  17
## initial  value 3498.749569
## iter  10 value 2853.423840
## iter  20 value 2834.092942
## iter  30 value 2811.504555
## iter  40 value 2750.031202
## iter  50 value 2565.662009
## iter  60 value 2534.299486
## iter  70 value 2529.481564
## iter  80 value 2525.742816
## final  value 2525.616829
## converged
## # weights:  25
## initial  value 4672.602071
## iter  10 value 2769.048158
## iter  20 value 2569.486497
## iter  30 value 2480.506982
## iter  40 value 2456.932361
## iter  50 value 2451.119154
## iter  60 value 2447.956324
## iter  70 value 2446.535971
## iter  80 value 2446.033413
## final  value 2446.016640
## converged
## # weights:  9
## initial  value 4674.370551
## iter  10 value 2950.966285
## iter  20 value 2875.572508
## iter  30 value 2871.950786
## final  value 2871.629993
## converged
## # weights:  17
## initial  value 6834.284516
## iter  10 value 2908.547863
## iter  20 value 2833.100926
## iter  30 value 2825.584692
## iter  40 value 2825.312032
## iter  50 value 2813.812619
## iter  60 value 2771.443407
## iter  70 value 2741.444389
## final  value 2741.287408
## converged
## # weights:  25
## initial  value 3292.068647
## iter  10 value 2738.199155
## iter  20 value 2685.278120
## iter  30 value 2654.987084
## iter  40 value 2613.261744
## iter  50 value 2580.249885
## iter  60 value 2560.141257
## iter  70 value 2540.147832
## iter  80 value 2529.851048
## iter  90 value 2511.670547
## iter 100 value 2499.389943
## final  value 2499.389943
## stopped after 100 iterations
## # weights:  9
## initial  value 3767.735859
## iter  10 value 2905.454464
## iter  20 value 2849.256647
## iter  30 value 2844.532136
## final  value 2844.480104
## converged
## # weights:  17
## initial  value 5805.913204
## iter  10 value 2959.436097
## iter  20 value 2852.284110
## iter  30 value 2776.638515
## iter  40 value 2759.547047
## iter  50 value 2759.154195
## final  value 2759.099974
## converged
## # weights:  25
## initial  value 5269.087507
## iter  10 value 2774.597057
## iter  20 value 2704.110612
```

```
## iter  30 value 2676.722447
## iter  40 value 2615.115834
## iter  50 value 2566.887360
## iter  60 value 2548.470568
## iter  70 value 2527.369956
## iter  80 value 2511.139962
## iter  90 value 2504.631649
## iter 100 value 2501.460888
## final  value 2501.460888
## stopped after 100 iterations
## # weights:  9
## initial  value 4920.537551
## iter  10 value 2886.659340
## iter  20 value 2855.726071
## iter  30 value 2851.573530
## final  value 2851.523075
## converged
## # weights:  17
## initial  value 4984.528352
## iter  10 value 2841.805501
## iter  20 value 2718.565114
## iter  30 value 2677.477486
## iter  40 value 2662.880495
## iter  50 value 2660.601955
## iter  60 value 2642.084216
## iter  70 value 2614.344525
## iter  80 value 2602.022869
## iter  90 value 2587.877177
## iter 100 value 2583.461672
## final  value 2583.461672
## stopped after 100 iterations
## # weights:  25
## initial  value 3555.716881
## iter  10 value 2807.504599
## iter  20 value 2748.387972
## iter  30 value 2694.086947
## iter  40 value 2554.529392
## iter  50 value 2502.833680
## iter  60 value 2493.617313
## iter  70 value 2486.405539
## iter  80 value 2478.289103
## iter  90 value 2468.729010
## iter 100 value 2466.192964
## final  value 2466.192964
## stopped after 100 iterations
## # weights:  9
## initial  value 6216.096833
## iter  10 value 3021.855973
## iter  20 value 2857.448246
## iter  30 value 2853.193028
## final  value 2853.174348
## converged
## # weights:  17
## initial  value 3801.746968
## iter  10 value 2869.041621
## iter  20 value 2832.071868
## iter  30 value 2820.207015
## iter  40 value 2607.067358
## iter  50 value 2571.473586
## iter  60 value 2551.320705
## final  value 2551.303348
## converged
## # weights:  25
## initial  value 5594.089998
## iter  10 value 2821.662637
## iter  20 value 2775.321763
## iter  30 value 2710.877390
## iter  40 value 2590.450921
## iter  50 value 2519.544745
## iter  60 value 2501.867233
## iter  70 value 2488.262258
## iter  80 value 2479.511563
## iter  90 value 2474.155648
## iter 100 value 2471.951516
## final  value 2471.951516
## stopped after 100 iterations
## # weights:  9
## initial  value 5869.306657
## iter  10 value 3275.237394
```

```
## iter  20 value 3130.224031
## iter  30 value 3094.877441
## iter  40 value 2868.412903
## iter  50 value 2839.856159
## iter  60 value 2836.115160
## iter  70 value 2836.103939
## final  value 2836.102748
## converged
## # weights:  17
## initial  value 5540.967200
## iter  10 value 2826.203428
## iter  20 value 2686.240005
## iter  30 value 2644.992098
## iter  40 value 2638.875177
## iter  50 value 2629.033571
## iter  60 value 2616.427500
## iter  70 value 2596.949557
## iter  80 value 2588.055893
## iter  90 value 2561.759433
## iter 100 value 2558.957807
## final  value 2558.957807
## stopped after 100 iterations
## # weights:  25
## initial  value 5185.785078
## iter  10 value 2819.848929
## iter  20 value 2758.957855
## iter  30 value 2701.724157
## iter  40 value 2652.758035
## iter  50 value 2618.733143
## iter  60 value 2603.259710
## iter  70 value 2578.409571
## iter  80 value 2496.548058
## iter  90 value 2481.368905
## iter 100 value 2465.897151
## final  value 2465.897151
## stopped after 100 iterations
## # weights:  9
## initial  value 4658.210330
## iter  10 value 2853.502441
## iter  20 value 2837.188005
## final  value 2837.187056
## converged
## # weights:  17
## initial  value 4708.026454
## iter  10 value 2942.558714
## iter  20 value 2775.492872
## iter  30 value 2707.341097
## iter  40 value 2672.882034
## iter  50 value 2644.312764
## iter  60 value 2603.176958
## iter  70 value 2595.733888
## iter  80 value 2595.296504
## iter  90 value 2595.186346
## iter  90 value 2595.186337
## iter  90 value 2595.186337
## final  value 2595.186337
## converged
## # weights:  25
## initial  value 5907.699412
## iter  10 value 2885.386912
## iter  20 value 2779.288525
## iter  30 value 2638.501049
## iter  40 value 2538.927147
## iter  50 value 2525.454119
## iter  60 value 2519.627827
## iter  70 value 2512.566175
## iter  80 value 2508.854302
## iter  90 value 2507.467579
## final  value 2507.463278
## converged
## # weights:  9
## initial  value 7380.801823
## iter  10 value 2868.520279
## iter  20 value 2846.715256
## iter  30 value 2843.165097
## final  value 2843.129646
## converged
## # weights:  17
## initial  value 3841.951686
```

```
## iter  10 value 2817.272051
## iter  20 value 2752.526361
## iter  30 value 2711.922898
## iter  40 value 2663.880869
## iter  50 value 2647.273119
## iter  60 value 2590.800206
## iter  70 value 2579.314711
## iter  80 value 2578.655730
## iter  90 value 2578.196485
## final  value 2578.193419
## converged
## # weights:  25
## initial  value 4559.385720
## iter  10 value 2850.908453
## iter  20 value 2766.217925
## iter  30 value 2705.865101
## iter  40 value 2678.596122
## iter  50 value 2652.121155
## iter  60 value 2639.537740
## iter  70 value 2622.319385
## iter  80 value 2583.413209
## iter  90 value 2553.871445
## iter 100 value 2546.767623
## final  value 2546.767623
## stopped after 100 iterations
## # weights:  9
## initial  value 4128.983546
## iter  10 value 2914.105206
## iter  20 value 2852.970737
## iter  30 value 2845.024521
## final  value 2844.800035
## converged
## # weights:  17
## initial  value 3768.927157
## iter  10 value 2869.946230
## iter  20 value 2840.914035
## iter  30 value 2804.181997
## iter  40 value 2797.115667
## iter  50 value 2752.311837
## final  value 2752.307019
## converged
## # weights:  25
## initial  value 4080.121318
## iter  10 value 2773.493503
## iter  20 value 2719.502014
## iter  30 value 2648.036361
## iter  40 value 2561.675593
## iter  50 value 2525.302870
## iter  60 value 2507.737690
## iter  70 value 2495.460385
## iter  80 value 2480.828034
## iter  90 value 2466.224826
## iter 100 value 2465.917993
## final  value 2465.917993
## stopped after 100 iterations
## # weights:  9
## initial  value 3863.920967
## iter  10 value 2896.039415
## iter  20 value 2861.445152
## iter  30 value 2855.949639
## final  value 2855.607300
## converged
## # weights:  17
## initial  value 6044.052559
## iter  10 value 2878.317839
## iter  20 value 2846.612678
## iter  30 value 2837.561426
## iter  40 value 2768.349918
## iter  50 value 2754.303988
## iter  60 value 2749.486554
## iter  60 value 2749.486529
## iter  60 value 2749.486529
## final  value 2749.486529
## converged
## # weights:  25
## initial  value 3506.614915
## iter  10 value 2861.347853
## iter  20 value 2797.623010
## iter  30 value 2652.175269
```

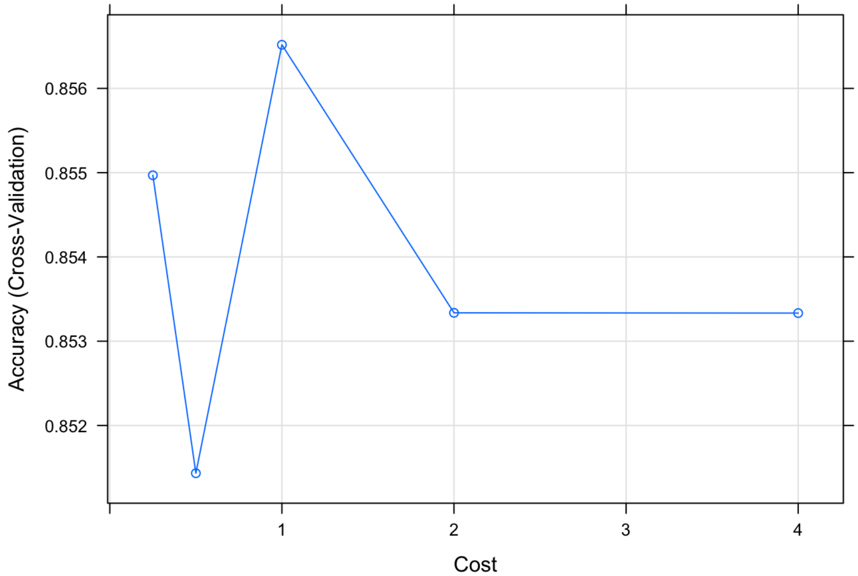
```
## iter  40 value 2586.233785
## iter  50 value 2496.718538
## iter  60 value 2473.880368
## iter  70 value 2467.942559
## iter  80 value 2460.016956
## iter  90 value 2455.546215
## iter 100 value 2455.534659
## final  value 2455.534659
## stopped after 100 iterations
## # weights:  9
## initial  value 3749.610590
## iter  10 value 3105.445082
## iter  20 value 2857.734659
## iter  30 value 2857.279664
## final  value 2857.266461
## converged
## # weights:  17
## initial  value 4366.652930
## iter  10 value 2858.571725
## iter  20 value 2830.633095
## iter  30 value 2821.758154
## iter  40 value 2817.035049
## iter  50 value 2816.156848
## iter  60 value 2814.824852
## final  value 2814.824735
## converged
## # weights:  25
## initial  value 3759.266951
## iter  10 value 2816.771080
## iter  20 value 2593.968293
## iter  30 value 2530.811334
## iter  40 value 2507.828368
## iter  50 value 2502.404613
## iter  60 value 2483.250696
## iter  70 value 2475.698256
## iter  80 value 2469.429112
## iter  90 value 2468.740548
## final  value 2468.740456
## converged
## # weights:  9
## initial  value 5662.081047
## iter  10 value 2964.418701
## iter  20 value 2853.558165
## iter  30 value 2853.001007
## final  value 2852.985689
## converged
## # weights:  17
## initial  value 4158.276093
## iter  10 value 2860.113060
## iter  20 value 2780.309626
## iter  30 value 2712.627218
## iter  40 value 2681.868194
## iter  50 value 2666.209069
## iter  60 value 2629.384472
## iter  70 value 2594.522634
## iter  80 value 2590.014270
## iter  90 value 2588.440151
## final  value 2588.415250
## converged
## # weights:  25
## initial  value 4348.577269
## iter  10 value 2840.668874
## iter  20 value 2808.012753
## iter  30 value 2646.600445
## iter  40 value 2558.967016
## iter  50 value 2520.428494
## iter  60 value 2509.574425
## iter  70 value 2501.478097
## iter  80 value 2487.376044
## iter  90 value 2468.713821
## iter 100 value 2467.362368
## final  value 2467.362368
## stopped after 100 iterations
## # weights:  9
## initial  value 3601.319992
## iter  10 value 2904.911451
## iter  20 value 2856.271452
## iter  30 value 2853.877476
## final  value 2853.787695
```

```
## converged
## # weights:  17
## initial  value 4954.619328
## iter  10 value 2898.844410
## iter  20 value 2814.572723
## iter  30 value 2730.394644
## iter  40 value 2695.464033
## iter  50 value 2675.204669
## iter  60 value 2637.822069
## iter  70 value 2619.569915
## iter  80 value 2618.492792
## iter  90 value 2618.176657
## final  value 2618.176008
## converged
## # weights:  25
## initial  value 5540.846638
## iter  10 value 2870.760042
## iter  20 value 2779.966159
## iter  30 value 2685.675736
## iter  40 value 2548.582023
## iter  50 value 2519.672972
## iter  60 value 2509.920595
## iter  70 value 2495.528496
## iter  80 value 2481.916183
## iter  90 value 2476.575566
## iter 100 value 2476.433131
## final  value 2476.433131
## stopped after 100 iterations
## # weights:  25
## initial  value 5754.220814
## iter  10 value 3112.469003
## iter  20 value 3014.412880
## iter  30 value 2942.419279
## iter  40 value 2843.059573
## iter  50 value 2769.850795
## iter  60 value 2754.535783
## iter  70 value 2745.078361
## iter  80 value 2729.817868
## iter  90 value 2726.053647
## iter 100 value 2725.940830
## final  value 2725.940830
## stopped after 100 iterations
```

```
nnFit_tune # After tunning, best model is size = 3 and decay = 0.2  Accuracy is 83.86% and Kappa = 0.4097141
```

```
## Neural Network
##
## 7350 samples
##    6 predictor
##    2 classes: 'no', 'yes'
##
## Pre-processing: centered (6), scaled (6)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 6615, 6615, 6614, 6615, 6615, 6615, ...
## Resampling results across tuning parameters:
##
##   size  decay  Accuracy   Kappa
##   1     0.1    0.7945535  0.1806682
##   1     0.2    0.7940093  0.1706728
##   2     0.1    0.8282959  0.3610426
##   2     0.2    0.8243472  0.3467579
##   3     0.1    0.8410876  0.4176904
##   3     0.2    0.8386370  0.4086754
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were size = 3 and decay = 0.1.
```

```
plot(svmFit_tune)
```



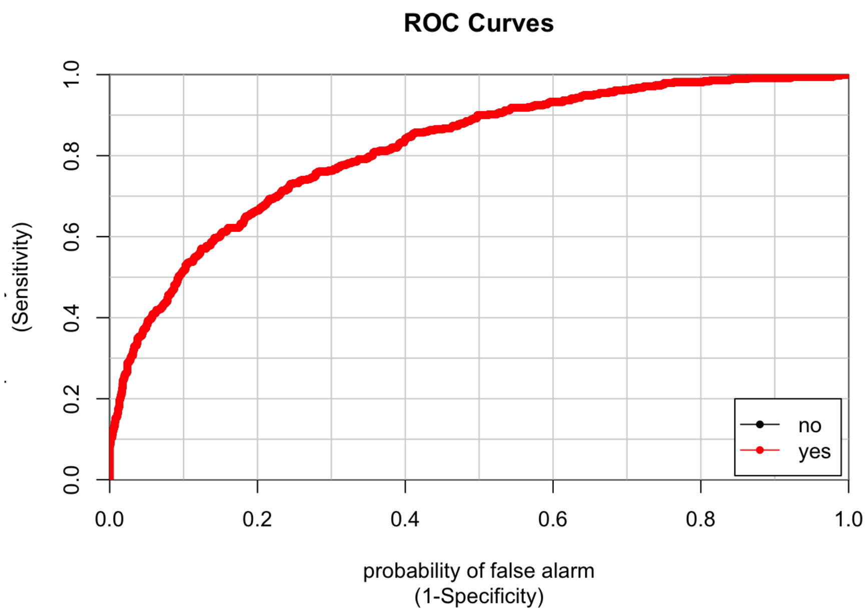

```
# Predict testing set
p_nn_tune <- predict(nnFit_tune, test[-7])
confusionMatrix(p_nn_tune, test$y, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1841  288
##        yes  119  201
##
##                Accuracy : 0.8338
##                  95% CI : (0.8185, 0.8484)
##     No Information Rate : 0.8003
##     P-Value [Acc > NIR] : 1.289e-05
##
##                   Kappa : 0.4025
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##               Precision : 0.8647
##                  Recall : 0.9393
##                      F1 : 0.9005
##              Prevalence : 0.8003
##          Detection Rate : 0.7517
##    Detection Prevalence : 0.8693
##       Balanced Accuracy : 0.6752
##
##        'Positive' Class : no
##
```

```
# Accuracy : 0.8338
# Precision : 0.8647
# Recall : 0.9393

# 3. AUC---------------------------------------------------------------------------------
pred_nn_cv <- predict(nnFit_tune, test[-7], type = "prob")
colAUC(pred_nn_cv, test$y)
```

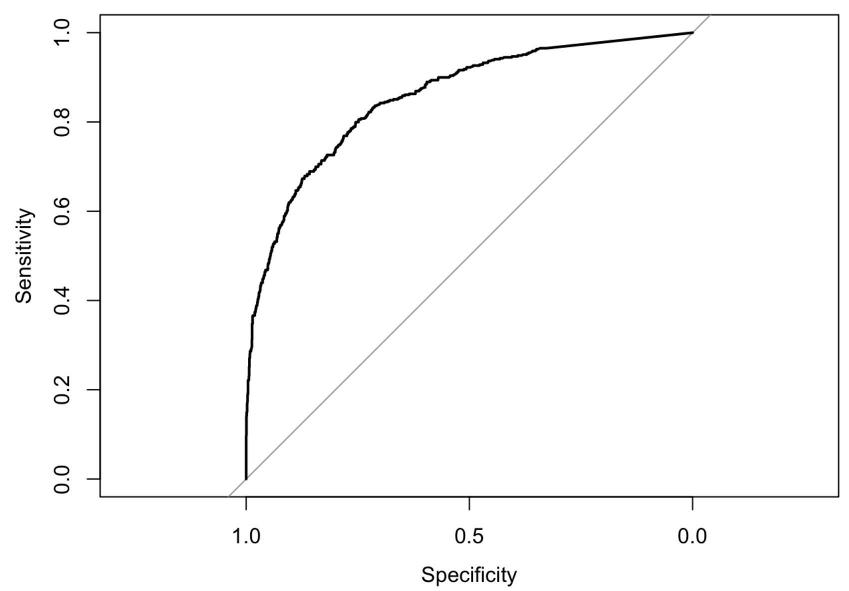
```
##                   no       yes
## no vs. yes 0.8162514 0.8162514
```

```
colAUC(pred_nn_cv, test$y, plotROC = TRUE)
```




```
##                    no       yes
## no vs. yes 0.8162514 0.8162514
```

```
auc_nn <- roc(response=test$y, predictor=pred_nn_cv[,1])
plot(auc_dt)
```




```
auc_nn$auc # AUC 81.63%
```

```
## Area under the curve: 0.8163
```

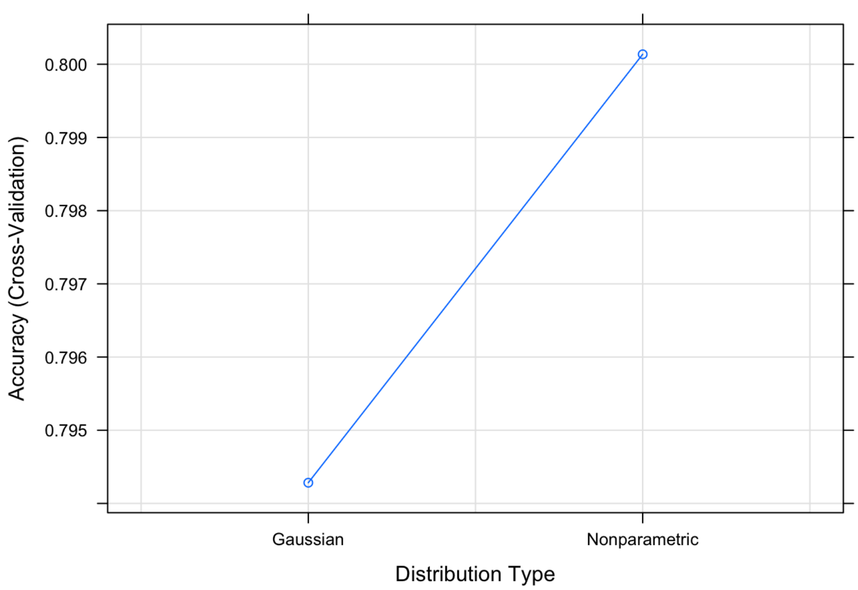
```
# Comment: k-fold cross validation did improve the model performance compare to holdout method. Tuning the parame
ters did improve the holdout method.
```

4. Naive Bayes

```
# Naive Bayes CV and Tuning

# 1. Evaluation of k-folds cross-validation-------------------------------------------

train <- nb_df[split,]
test <- nb_df[-split,]


nbFit_cv <- train(Exited ~ ., data = train, method = "nb", trControl = ctrl)

nbFit_cv
```

```
## Naive Bayes
##
## 7350 samples
##   10 predictor
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 6616, 6615, 6615, 6615, 6615, 6615, ...
## Resampling results across tuning parameters:
##
##   usekernel  Accuracy   Kappa
##   FALSE      0.7942832  0.3020774
##    TRUE      0.8001362  0.0000000
##
## Tuning parameter 'fL' was held constant at a value of 0
## Tuning
##  parameter 'adjust' was held constant at a value of 1
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were fL = 0, usekernel = TRUE and adjust
##  = 1.
```

```
plot(nbFit_cv)
```




```
# Predict testing set
p_nb <- predict(nbFit_cv, test[-11])
confusionMatrix(p_nb, test$Exited, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
## 
##           Reference
## Prediction   no  yes
##        no  1960  489
##        yes    0    0
##                                           
##                Accuracy : 0.8003          
##                  95% CI : (0.7839, 0.816) 
##     No Information Rate : 0.8003          
##     P-Value [Acc > NIR] : 0.5121          
##                                           
##                   Kappa : 0               
##                                           
##  Mcnemar's Test P-Value : <2e-16          
##                                           
##               Precision : 0.8003          
##                  Recall : 1.0000          
##                      F1 : 0.8891          
##              Prevalence : 0.8003          
##          Detection Rate : 0.8003          
##    Detection Prevalence : 1.0000          
##       Balanced Accuracy : 0.5000          
##                                           
##        'Positive' Class : no              
## 
```

```
# Accuracy : 0.8003
# Precision : 0.8003
# Recall : 1.0000


# Comment: Naive Bayes did not perform well, it has a sensitivity of 0, I did not perform a tuning and cross vali
dation for it as it did not improve anything and taking a very long time to run. The cross validation result and
holdout method produced the same result.
```

Evaluation of k-fold cross validation, not surprisingly, Decision Tree model again out performed the other models with 86% accuracy, precision of 87.78%, recall of 95.66% and F-score of 91.62% and AUC of 85.57%, follow by SVM with accuracy of 85.91%, precision of 86.82%, recall of 97.14%, F-score of 91.69% and AUC of 81.07%, follow by Neural Network with 83.38%, precision of 86.47%, recall of 93.93% and F-score of 90.05% and AUC of 81.63%. Naive Byes again performed the worst. I will exclude it from the Stacking Ensemble model.

In this section, Stacking Ensemble of top three performing models: Decision Tree, SVM and Neural Network are stacked to build an ensemble model. And Bagging Ensemble for Decision

Tree is built. The Goal is to compare bagging ensemble of Decision Tree and Stacked Ensemble model of Decision Trees, SVM and Neural Network

Construction of Stacking Ensemble Model and AUC Evaluation

```
set.seed(1)

train <- df[split,]
test <- df[-split,]

# Example of Stacking algorithms
# create submodels
control <- trainControl(method="cv", number=5, savePredictions = "final", classProbs=TRUE)

algorithmList <- c('C5.0', 'svmRadial', 'nnet')


# Stacking Algorithms - Run multiple algorithms in one call.
models <- caretList(Exited ~., data=train, trControl=control, methodList=algorithmList)
```

```
## line search fails -2.560131 -0.7218277 1.040596e-05 -6.232883e-06 -3.786098e-08 -2.16508e-08 -2.590329e-13# we
ights:  14
## initial  value 3644.996684
## final  value 2922.374580
## converged
## # weights:  40
## initial  value 3042.681029
## iter  10 value 2897.353445
## iter  20 value 2896.545037
## iter  30 value 2896.524189
## final  value 2896.523700
## converged
## # weights:  66
## initial  value 3956.641029
## iter  10 value 2897.845234
## iter  20 value 2897.370780
## final  value 2897.359498
## converged
## # weights:  14
## initial  value 3594.996252
## iter  10 value 2942.556452
## iter  20 value 2903.124056
## iter  30 value 2802.351299
## iter  40 value 2688.789559
## iter  50 value 2585.433677
## iter  60 value 2456.801272
## iter  70 value 2415.422725
## iter  80 value 2408.653611
## iter  90 value 2406.189914
## final  value 2406.076553
## converged
## # weights:  40
## initial  value 6922.481628
## iter  10 value 2942.170228
## iter  20 value 2938.568118
## iter  30 value 2897.663185
## iter  40 value 2896.310266
## iter  50 value 2894.704713
## iter  60 value 2868.921241
## iter  70 value 2744.913200
## iter  80 value 2707.366383
## iter  90 value 2672.420209
## iter 100 value 2669.988795
## final  value 2669.988795
## stopped after 100 iterations
## # weights:  66
## initial  value 3059.716257
## iter  10 value 2903.108124
## final  value 2900.275296
## converged
## # weights:  14
## initial  value 3020.482264
## final  value 2942.366658
## converged
## # weights:  40
## initial  value 3396.398731
## iter  10 value 2905.901811
## final  value 2895.037700
## converged
## # weights:  66
## initial  value 3489.015481
## iter  10 value 2938.891534
## final  value 2938.888284
## converged
## # weights:  14
## initial  value 4471.454353
## final  value 2940.979424
## converged
## # weights:  40
## initial  value 3485.456067
## iter  10 value 2903.666979
## iter  20 value 2900.176643
## iter  30 value 2899.707962
## iter  40 value 2899.644293
## iter  50 value 2899.595296
## final  value 2899.595164
## converged
## # weights:  66
```

```
## initial  value 5139.002600
## final  value 2940.979424
## converged
## # weights:  14
## initial  value 3398.520357
## iter  10 value 2905.683090
## iter  20 value 2903.821013
## final  value 2903.814374
## converged
## # weights:  40
## initial  value 5933.291285
## iter  10 value 2999.459516
## iter  20 value 2941.222852
## iter  30 value 2933.861241
## iter  40 value 2900.090032
## iter  50 value 2896.165971
## iter  60 value 2895.388141
## iter  70 value 2879.302045
## iter  80 value 2860.200670
## iter  90 value 2762.175198
## iter 100 value 2750.227587
## final  value 2750.227587
## stopped after 100 iterations
## # weights:  66
## initial  value 9814.353956
## iter  10 value 2902.830951
## iter  20 value 2902.022182
## iter  30 value 2901.973593
## iter  40 value 2897.083478
## iter  50 value 2892.539006
## iter  60 value 2828.026695
## iter  70 value 2771.060146
## iter  80 value 2756.051876
## iter  90 value 2755.001520
## iter 100 value 2753.953727
## final  value 2753.953727
## stopped after 100 iterations
## # weights:  14
## initial  value 4453.572955
## final  value 2940.979798
## converged
## # weights:  40
## initial  value 4743.659085
## iter  10 value 2925.884673
## final  value 2901.227129
## converged
## # weights:  66
## initial  value 3225.629826
## iter  10 value 2901.644784
## iter  20 value 2897.524766
## iter  30 value 2897.133476
## iter  40 value 2896.846511
## iter  50 value 2896.646811
## iter  60 value 2896.325216
## iter  70 value 2896.063990
## iter  80 value 2895.939517
## final  value 2895.938691
## converged
## # weights:  14
## initial  value 3307.919521
## final  value 2940.979424
## converged
## # weights:  40
## initial  value 3582.993964
## iter  10 value 2936.981994
## iter  20 value 2936.959984
## final  value 2936.959785
## converged
## # weights:  66
## initial  value 3150.839490
## iter  10 value 2908.861631
## iter  20 value 2907.020058
## iter  30 value 2906.744522
## iter  40 value 2905.865464
## final  value 2905.849733
## converged
## # weights:  14
## initial  value 3084.784896
## iter  10 value 2941.088651
```

```
## iter  10 value 2941.088639
## iter  10 value 2941.088634
## final  value 2941.088634
## converged
## # weights:  40
## initial  value 7342.636765
## iter  10 value 2934.503102
## iter  20 value 2928.761469
## iter  30 value 2927.512639
## iter  40 value 2927.270994
## iter  50 value 2913.815272
## iter  60 value 2905.282376
## iter  70 value 2904.311754
## iter  80 value 2904.257959
## final  value 2904.078250
## converged
## # weights:  66
## initial  value 2941.094622
## iter  10 value 2912.372729
## iter  20 value 2904.052933
## iter  30 value 2897.930149
## iter  40 value 2881.891947
## iter  50 value 2864.852810
## iter  60 value 2812.586510
## iter  70 value 2764.857582
## iter  80 value 2726.898889
## iter  90 value 2698.289311
## iter 100 value 2688.800469
## final  value 2688.800469
## stopped after 100 iterations
## # weights:  14
## initial  value 4798.486783
## final  value 2940.979769
## converged
## # weights:  40
## initial  value 3172.091126
## iter  10 value 2926.005300
## iter  20 value 2923.329500
## iter  30 value 2909.276929
## iter  40 value 2908.929334
## iter  50 value 2907.421637
## iter  60 value 2907.393902
## final  value 2907.312832
## converged
## # weights:  66
## initial  value 3910.697235
## iter  10 value 2931.379348
## iter  20 value 2929.547403
## iter  30 value 2929.525974
## iter  40 value 2928.879485
## iter  50 value 2928.581339
## iter  60 value 2928.505069
## iter  70 value 2928.484266
## iter  80 value 2928.157634
## final  value 2928.157236
## converged
## # weights:  14
## initial  value 3816.249265
## final  value 2940.979424
## converged
## # weights:  40
## initial  value 4229.004201
## iter  10 value 2931.257031
## iter  10 value 2931.257027
## iter  10 value 2931.257027
## final  value 2931.257027
## converged
## # weights:  66
## initial  value 3456.969424
## iter  10 value 2903.596157
## iter  20 value 2892.667844
## iter  30 value 2892.580766
## iter  40 value 2892.066596
## iter  50 value 2891.813048
## iter  60 value 2891.747960
## iter  70 value 2891.491882
## iter  70 value 2891.491882
## iter  70 value 2891.491882
## final  value 2891.491882
```

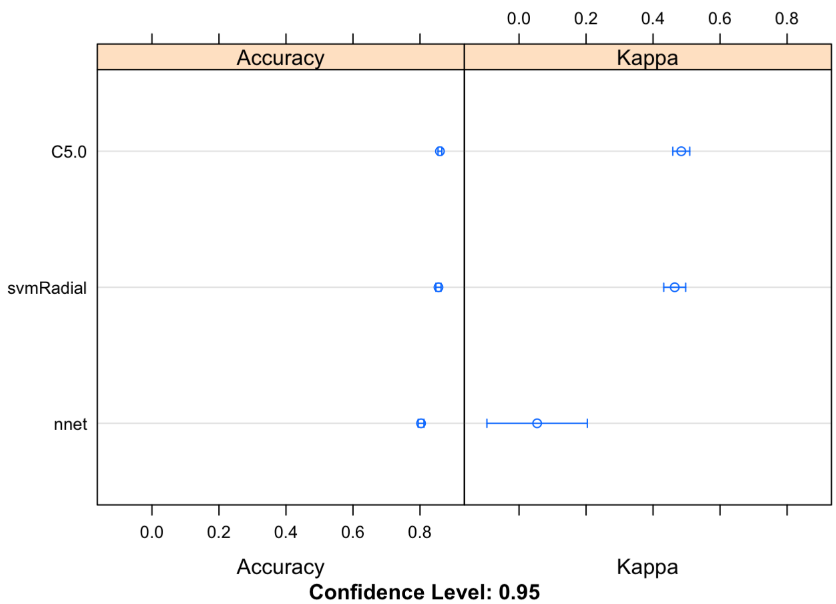
```
## converged
## # weights:  14
## initial  value 4128.272542
## iter  10 value 2940.344782
## iter  20 value 2940.003172
## iter  30 value 2939.732658
## iter  30 value 2939.732633
## iter  40 value 2939.354448
## iter  40 value 2939.354429
## iter  40 value 2939.354427
## final  value 2939.354427
## converged
## # weights:  40
## initial  value 5242.026392
## iter  10 value 2895.021968
## iter  20 value 2894.204671
## iter  30 value 2892.959206
## iter  40 value 2889.707163
## iter  50 value 2889.544350
## final  value 2889.541411
## converged
## # weights:  66
## initial  value 4120.969455
## iter  10 value 2930.379945
## iter  20 value 2904.584228
## iter  30 value 2898.169523
## iter  40 value 2892.886544
## iter  50 value 2891.945692
## iter  60 value 2891.860758
## iter  70 value 2890.915752
## iter  80 value 2890.730379
## iter  90 value 2890.318082
## iter 100 value 2889.692294
## final  value 2889.692294
## stopped after 100 iterations
## # weights:  14
## initial  value 5111.676160
## iter  10 value 2940.108268
## final  value 2940.099144
## converged
## # weights:  40
## initial  value 8655.979186
## iter  10 value 2940.732543
## iter  20 value 2939.621109
## final  value 2939.619237
## converged
## # weights:  66
## initial  value 3721.099381
## iter  10 value 2929.494101
## iter  20 value 2896.819140
## iter  30 value 2896.253020
## iter  40 value 2895.973889
## iter  50 value 2895.807613
## final  value 2895.678205
## converged
## # weights:  14
## initial  value 4461.250215
## final  value 2940.979424
## converged
## # weights:  40
## initial  value 8045.000867
## iter  10 value 2940.095573
## final  value 2940.087378
## converged
## # weights:  66
## initial  value 5669.278246
## iter  10 value 2929.807161
## iter  20 value 2929.180878
## iter  30 value 2928.993005
## iter  40 value 2928.895767
## final  value 2928.892117
## converged
## # weights:  14
## initial  value 2970.777418
## iter  10 value 2932.336053
## iter  20 value 2903.788120
## iter  30 value 2903.655832
## final  value 2903.655124
## converged
```

```
## # weights:  40
## initial  value 6809.701187
## iter  10 value 2941.502759
## iter  20 value 2890.968844
## iter  30 value 2883.721835
## iter  40 value 2845.222264
## iter  50 value 2827.150530
## iter  60 value 2766.097676
## iter  70 value 2757.340247
## iter  80 value 2744.478330
## iter  90 value 2738.626405
## iter 100 value 2734.778802
## final  value 2734.778802
## stopped after 100 iterations
## # weights:  66
## initial  value 4646.346690
## iter  10 value 2894.837941
## iter  20 value 2888.452789
## iter  30 value 2886.802723
## iter  40 value 2886.470761
## iter  50 value 2876.624431
## iter  60 value 2871.546277
## iter  70 value 2778.563734
## iter  80 value 2769.393214
## iter  90 value 2763.155713
## iter 100 value 2721.370290
## final  value 2721.370290
## stopped after 100 iterations
## # weights:  14
## initial  value 3516.356942
## final  value 2939.763891
## converged
## # weights:  40
## initial  value 3448.522018
## iter  10 value 2888.871140
## iter  10 value 2888.871140
## iter  10 value 2888.871140
## final  value 2888.871140
## converged
## # weights:  66
## initial  value 3580.646919
## iter  10 value 2926.850259
## iter  20 value 2925.757302
## iter  30 value 2899.848433
## final  value 2892.984883
## converged
## # weights:  14
## initial  value 4487.084012
## iter  10 value 3676.686843
## final  value 3676.685651
## converged
```

```
results <- resamples(models)
summary(results)
```

```
##
## Call:
## summary.resamples(object = results)
##
## Models: C5.0, svmRadial, nnet
## Number of resamples: 5
##
## Accuracy
##                Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## C5.0      0.8544218 0.8564626 0.8578231 0.8594558 0.8639456 0.8646259    0
## svmRadial 0.8469388 0.8503401 0.8578231 0.8551020 0.8585034 0.8619048    0
## nnet      0.8000000 0.8000000 0.8000000 0.8031293 0.8000000 0.8156463    0
##
## Kappa
##                Min.   1st Qu.    Median       Mean   3rd Qu.      Max. NA's
## C5.0      0.4649258 0.4709651 0.4723866 0.48417873 0.5057129 0.5069034    0
## svmRadial 0.4375593 0.4410569 0.4649258 0.46465353 0.4794795 0.5002462    0
## nnet      0.0000000 0.0000000 0.0000000 0.05397797 0.0000000 0.2698899    0
```

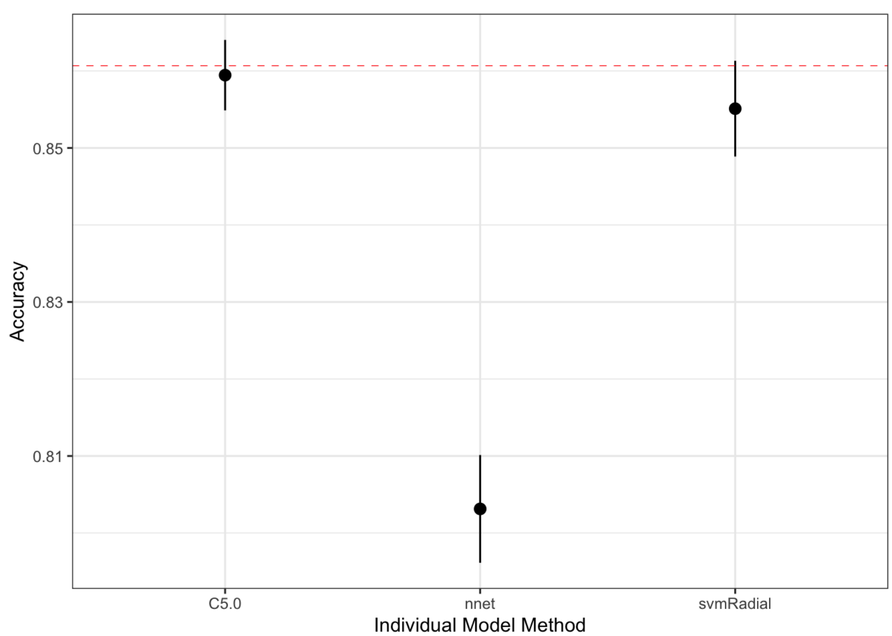
```
dotplot(results)
```




```
# Comment: As we can see decision tree has the highest performance in the ensemble model.



ensemble_1 <- caretEnsemble(models,
                            metric = "ROC",
                            trControl = control)
summary(ensemble_1)
```

```
## The following models were ensembled: C5.0, svmRadial, nnet
## They were weighted:
## 2.7757 -5.4308 -0.671 0.5265
## The resulting Accuracy is: 0.8607
## The fit for each individual model on the Accuracy is:
##     method  Accuracy  AccuracySD
##       C5.0 0.8594558 0.004578590
##  svmRadial 0.8551020 0.006216213
##       nnet 0.8031293 0.006997220
```

```
plot(ensemble_1)
```



```
# From the plot, we can see that C5.0 has the best performance.


# Combine the predictions of multiple models to form a final prediction.

# Ensemble the predictions of `models` to form a new combined prediction based on glm.
stack.glm <- caretStack(models, method = "glm", metric="Accuracy", trControl=control)

print(stack.glm)
```

```
## A glm ensemble of 3 base models: C5.0, svmRadial, nnet
##
## Ensemble results:
## Generalized Linear Model
##
## 7350 samples
##    3 predictor
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 5880, 5880, 5880, 5880, 5880
## Resampling results:
##
##   Accuracy   Kappa
##   0.8601361  0.4928501
```

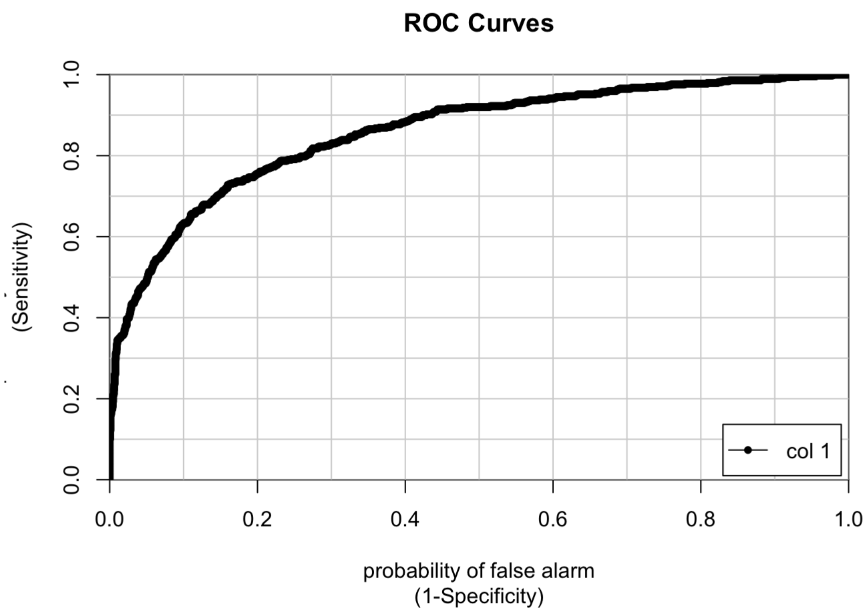
```
pred_ensemble <- predict(stack.glm, test[-11])

confusionMatrix(pred_ensemble, test$Exited, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1869  252
##        yes   91  237
##
##                Accuracy : 0.8599
##                  95% CI : (0.8456, 0.8735)
##     No Information Rate : 0.8003
##     P-Value [Acc > NIR] : 8.61e-15
##
##                   Kappa : 0.5
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##               Precision : 0.8812
##                  Recall : 0.9536
##                      F1 : 0.9160
##              Prevalence : 0.8003
##          Detection Rate : 0.7632
##    Detection Prevalence : 0.8661
##       Balanced Accuracy : 0.7191
##
##        'Positive' Class : no
##
```

```
# Accuracy : 0.8599
# Precision : 0.8812
# Recall : 0.9536
# F1 : 0.9160



# AUC-------------------------------------------------------------------------------------
pred_stack <- predict(stack.glm, test[-11], type = "prob")
colAUC(pred_stack, test$Exited)
```

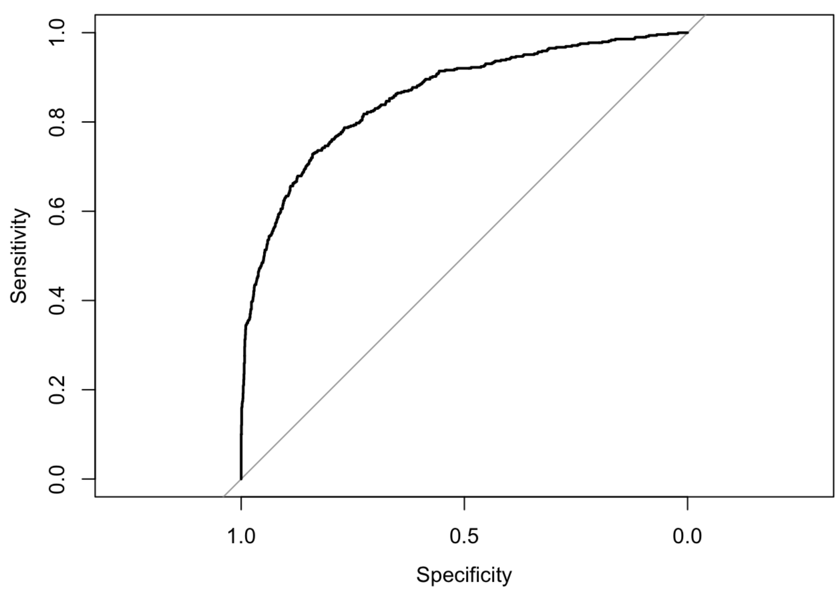
```
##                  [,1]
## no vs. yes 0.8570531
```

```
colAUC(pred_stack, test$Exited, plotROC = TRUE)
```




```
##                [,1]
## no vs. yes 0.8570531
```

```
auc_nb <- roc(response=test$Exited, predictor=pred_stack)
plot(auc_nb)
```




```
auc_nb$auc # 85.71%
```

```
## Area under the curve: 0.8571
```

```
# Comment: My stacked ensemble improved performance on average, it is the best performing model.
```

Use Boostrap and aggregating: Decision Tree Bagging and AUC Evaluation

Bagging: generates a number of training dataset by bootstrap sampling the original training data. These data sets then then used to generate a set of models using a single learning algorithm, here will be Decision Tree. The models' predictions are combined using voting for classification or averaging for numeric

prediction. Similar to bagging, boosting also resamples the training data, except it generates complimentary learners and assigns weights, the ones with better performance have greater weights over the ensemble's final prediction.

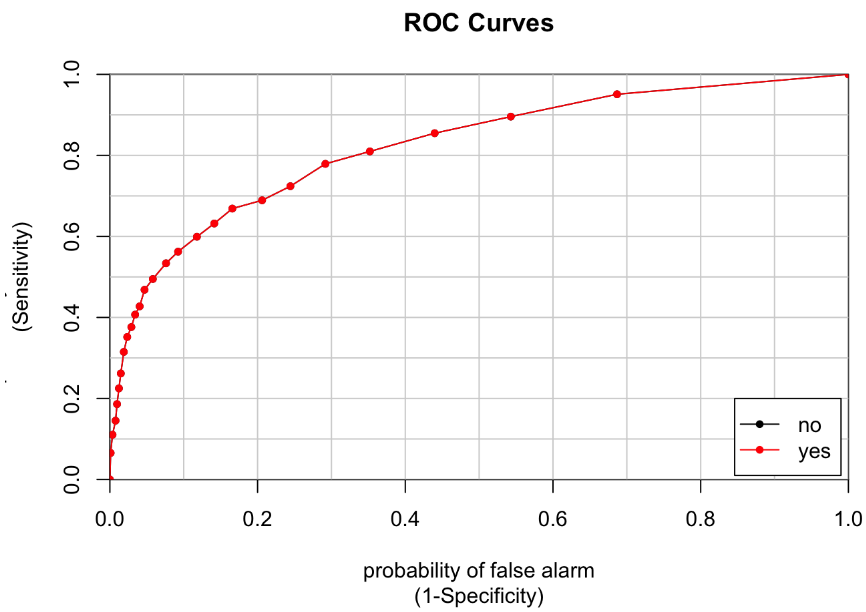
```
# Top three strong learners
# Bagging and boosting works well with decision tree models, here I chose bagging.
set.seed(1)
ctrl <- trainControl(method = "cv", number = 5) # by number of decision trees voting in the ensemble
bag <- train(Exited ~ ., data = training, method = "treebag", trControl = ctrl)

pred_bag <- predict(bag, testing[-11])
confusionMatrix(pred_bag, testing$Exited, mode = "prec_recall")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no  1846  247
##        yes  114  242
##
##                Accuracy : 0.8526
##                  95% CI : (0.8379, 0.8664)
##     No Information Rate : 0.8003
##     P-Value [Acc > NIR] : 1.191e-11
##
##                   Kappa : 0.4864
##
##  Mcnemar's Test P-Value : 3.722e-12
##
##               Precision : 0.8820
##                  Recall : 0.9418
##                      F1 : 0.9109
##              Prevalence : 0.8003
##          Detection Rate : 0.7538
##    Detection Prevalence : 0.8546
##       Balanced Accuracy : 0.7184
##
##        'Positive' Class : no
##
```

```
# Accuracy : 0.8526
# Precision : 0.8820
# Recall : 0.9418
# F1: 0.9109

# AUC-------------------------------------------------------------------------------------
pred_bag_auc <- predict(bag, testing[-11], type = "prob")
colAUC(pred_bag_auc, testing$Exited)
```

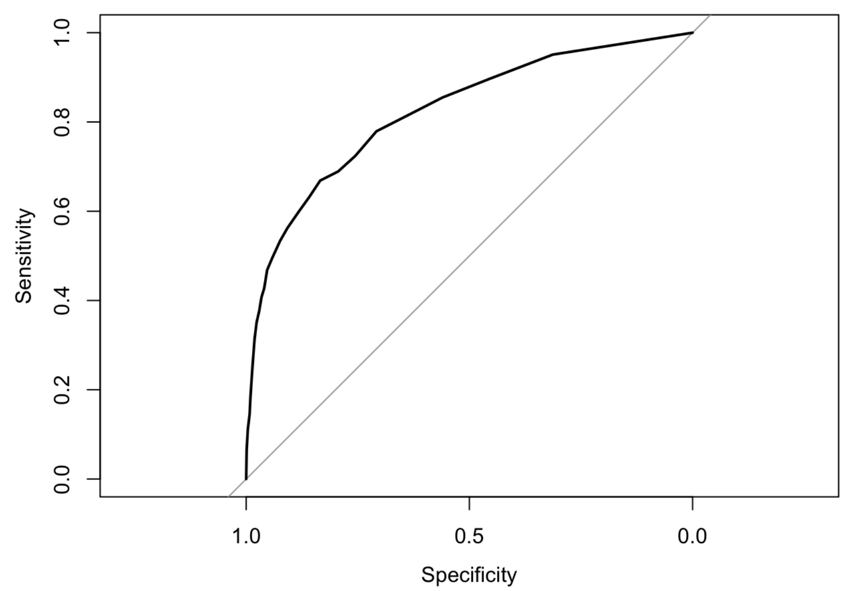
```
##                   no       yes
## no vs. yes 0.8228194 0.8228194
```

```
colAUC(pred_bag_auc, testing$Exited, plotROC = TRUE)
```




```
##                    no       yes
## no vs. yes 0.8228194 0.8228194
```

```
auc_bag <- roc(response=testing$Exited, predictor=pred_bag_auc[,1])
plot(auc_bag)
```




```
auc_bag$auc # AUC 82.28%
```

```
## Area under the curve: 0.8228
```

In conclusion, Stacked Ensemble model performed the best, it has an accuracy of 86% precision of 88.12%, recall of 95.36% and F-score of 91.60% and AUC of 85.71%. Decision tree model alone which is very close to the stacked ensemble model with accuracy of 86.12% and precision of 96.84% and recall of 87.22% and F-score of 91.78% and AUC of 84.01%, is the second best performing model. In my case, depends on the computer and its heuristic running methond, both Stacked Ensemble model and Decision Tree base model alone performed equally

well. Out of the four models built, Naive Bayes performed the worst, a lesson I learned here is perhaps kmeans should be used for clustering before binning to get the the optimal bin boundaries.

CRISP-DM Deployment

- According to the model prediction, about 86.12% accuracy, banks could use this prediction to predict whether a customer will churn or not. 86.12% will correctly predict churn movement.

- Bank could save money in the long run by targeting which type of customers to keep.

- Helps bank to develop loyalty programs and retention campaigns to keep as many customers as possible.

References

Machine Learning with R, Third Edition, Brett Lantz

https://www.dataquest.io/blog/top-10-machine-learning-algorithms-for-beginners/ (https://www.dataquest.io/blog/top-10-machine-learning-algorithms-for-beginners/)

http://www.sthda.com/english/articles/38-regression-model-validation/157-cross-validation-essentials-in-r/ (http://www.sthda.com/english/articles/38-regression-model-validation/157-cross-validation-essentials-in-r/)

https://topepo.github.io/caret/train-models-by-tag.html#boosting (//htmlpreview.github.io/?https://github.com/suesui117/da5030_final_project/blob/main/DA5030.Proj.Sui.html#boosting)

https://www.saedsayad.com/k_nearest_neighbors.htm (https://www.saedsayad.com/k_nearest_neighbors.htm)

https://medium.com/@eijaz/holdout-vs-cross-validation-in-machine-learning-7637112d3f8f (https://medium.com/@eijaz/holdout-vs-cross-validation-in-machine-learning-7637112d3f8f)

https://machinelearningmastery.com/machine-learning-ensembles-with-r/ (https://machinelearningmastery.com/machine-learning-ensembles-with-r/)

https://rpubs.com/njvijay/16444 (https://rpubs.com/njvijay/16444)

https://topepo.github.io/caret/model-training-and-tuning.html (https://topepo.github.io/caret/model-training-and-tuning.html)

https://www.wright.edu/center-for-teaching-and-learning/blog/article/creating-easy-narrated-screen-recordings-on-almost-any-mac (https://www.wright.edu/center-for-teaching-and-learning/blog/article/creating-easy-narrated-screen-recordings-on-almost-any-mac)

https://cran.r-project.org/web/packages/caretEnsemble/vignettes/caretEnsemble-intro.html (https://cran.r-project.org/web/packages/caretEnsemble/vignettes/caretEnsemble-intro.html)

https://blog.revolutionanalytics.com/2015/10/the-5th-tribe-support-vector-machines-and-caret.html (https://blog.revolutionanalytics.com/2015/10/the-5th-tribe-support-vector-machines-and-caret.html)

https://uc-r.github.io/naive_bayes (https://uc-r.github.io/naive_bayes)

https://machinelearningmastery.com/machine-learning-ensembles-with-r/ (https://machinelearningmastery.com/machine-learning-ensembles-with-r/)

https://topepo.github.io/caret/available-models.html (https://topepo.github.io/caret/available-models.html)

https://rpubs.com/zxs107020/370699 (https://rpubs.com/zxs107020/370699) CaretList and CaretStack

http://danlec.com/st4k#questions/49725934 (//htmlpreview.github.io/?https://github.com/suesui117/da5030_final_project/blob/main/DA5030.Proj.Sui.html#questions/49725934)

https://towardsdatascience.com/a-comprehensive-machine-learning-workflow-with-multiple-modelling-using-caret-and-caretensemble-in-fcbf6d80b5f2 (https://towardsdatascience.com/a-comprehensive-machine-learning-workflow-with-multiple-modelling-using-caret-and-caretensemble-in-fcbf6d80b5f2)

https://www.neuraldesigner.com/learning/examples/bank-churn (https://www.neuraldesigner.com/learning/examples/bank-churn)

https://datascience.stackexchange.com/questions/17146/does-ensemble-bagging-boosting-stacking-etc-always-at-least-increase-perfor (https://datascience.stackexchange.com/questions/17146/does-ensemble-bagging-boosting-stacking-etc-always-at-least-increase-perfor)

https://stats.libretexts.org/Bookshelves/Computing_and_Modeling/RTG%3A_Classification_Methods/4%3A_Numerical_Experiments_and_Real_Data_Analysis/Pre (https://stats.libretexts.org/Bookshelves/Computing_and_Modeling/RTG%3A_Classification_Methods/4%3A_Numerical_Experiments_and_Real_Data_Analysis/Pr

https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781788397872/1/ch01lvl1sec27/pros-and-cons-of-neural-networks (https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781788397872/1/ch01lvl1sec27/pros-and-cons-of-neural-networks) https://medium.com/analytics-vidhya/accuracy-vs-f1-score-6258237beca2 (https://medium.com/analytics-vidhya/accuracy-vs-f1-score-6258237beca2)

https://www.machinelearningplus.com/machine-learning/caret-package/ (https://www.machinelearningplus.com/machine-learning/caret-package/) (one hot encoding, caretStack)

https://www.saedsayad.com/decision_tree_reg.htm (https://www.saedsayad.com/decision_tree_reg.htm) (Decision Tree)

https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/ (https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/) (Integer encoding vs one hot encoding)