Statistical Methods Illustration - Classification Methods

Daniel Shang

```r
# Load the necessary packages
library(tidylog)
```

```
##
## Attaching package: 'tidylog'

## The following object is masked from 'package:stats':
##
##     filter
```

```r
library(e1071)
```

```
## Warning: package 'e1071' was built under R version 4.0.3
```

```r
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.0.3
```

```r
library(party)
```

```
## Warning: package 'party' was built under R version 4.0.3

## Loading required package: grid

## Loading required package: mvtnorm

## Warning: package 'mvtnorm' was built under R version 4.0.3

## Loading required package: modeltools

## Warning: package 'modeltools' was built under R version 4.0.3

## Loading required package: stats4

## Loading required package: strucchange

## Warning: package 'strucchange' was built under R version 4.0.3

## Loading required package: zoo
```

```
##
## Attaching package: 'zoo'

## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric

## Loading required package: sandwich

## Warning: package 'sandwich' was built under R version 4.0.3
```

```r
library(rpart)
library(rpart.plot)
```

```
## Warning: package 'rpart.plot' was built under R version 4.0.3
```

```r
library(randomForest)
```

```
## randomForest 4.6-14

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

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:ggplot2':
##
##     margin
```

```r
library(caret)
```

```
## Loading required package: lattice
```

```r
library(ggthemes)
```

```
## Warning: package 'ggthemes' was built under R version 4.0.3
```

```r
library(car)
```

```
## Warning: package 'car' was built under R version 4.0.3

## Loading required package: carData

##
## Attaching package: 'car'

## The following object is masked from 'package:modeltools':
##
##     Predict
```

```r
library(pROC)
```

```
## Type 'citation("pROC")' for a citation.

##
## Attaching package: 'pROC'

## The following objects are masked from 'package:stats':
##
##     cov, smooth, var
```

```r
library(naivebayes)
```

```
## Warning: package 'naivebayes' was built under R version 4.0.3

## naivebayes 0.9.7 loaded
```

```r
library(psych)
```

```
## Warning: package 'psych' was built under R version 4.0.3

##
## Attaching package: 'psych'

## The following object is masked from 'package:car':
##
##     logit

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

## The following objects are masked from 'package:ggplot2':
##
##     %+%, alpha
```

```r
library(readxl)
```

```r
# Load the data, remove missing values (if any), and convert columns to proper formats
## based on the documentation of the dataset
data = read.csv('C:/Users/34527/Desktop/heart.csv', )
data_clean = na.omit(mutate_all(data,
                      ~ifelse(. %in% c("N/A", "null", "", NULL),  NA, .)))
```

```
## mutate_all: no changes
```

```r
colnames(data_clean)[1] = 'age'
data_clean$sex = as.factor(data_clean$sex)
data_clean$cp = as.factor(data_clean$cp)
data_clean$fbs = as.factor(data_clean$fbs)
data_clean$restecg = as.factor(data_clean$restecg)
data_clean$exang = as.factor(data_clean$exang)
data_clean$slope = as.factor(data_clean$slope)
data_clean$ca = as.factor(data_clean$ca)
data_clean$thal = as.factor(data_clean$thal)
data_clean$target = as.factor(data_clean$target)
```

```r
# Set up a train/test split for later model evaluation
set.seed(111)
index_train_test = sample(x = 2, size = nrow(data_clean), replace = TRUE, prob = c(0.8, 0.2))
train_data = data_clean[index_train_test == 1, ]
test_data = data_clean[index_train_test == 2, ]
```

——————————————- Support Vector Machine ——————————————-

```r
# Build a support vector machine (SVM) to predict the 'target' and summarize the model
svm_model = svm(target ~ ., data = train_data, kernel = 'linear')
summary(svm_model)
```

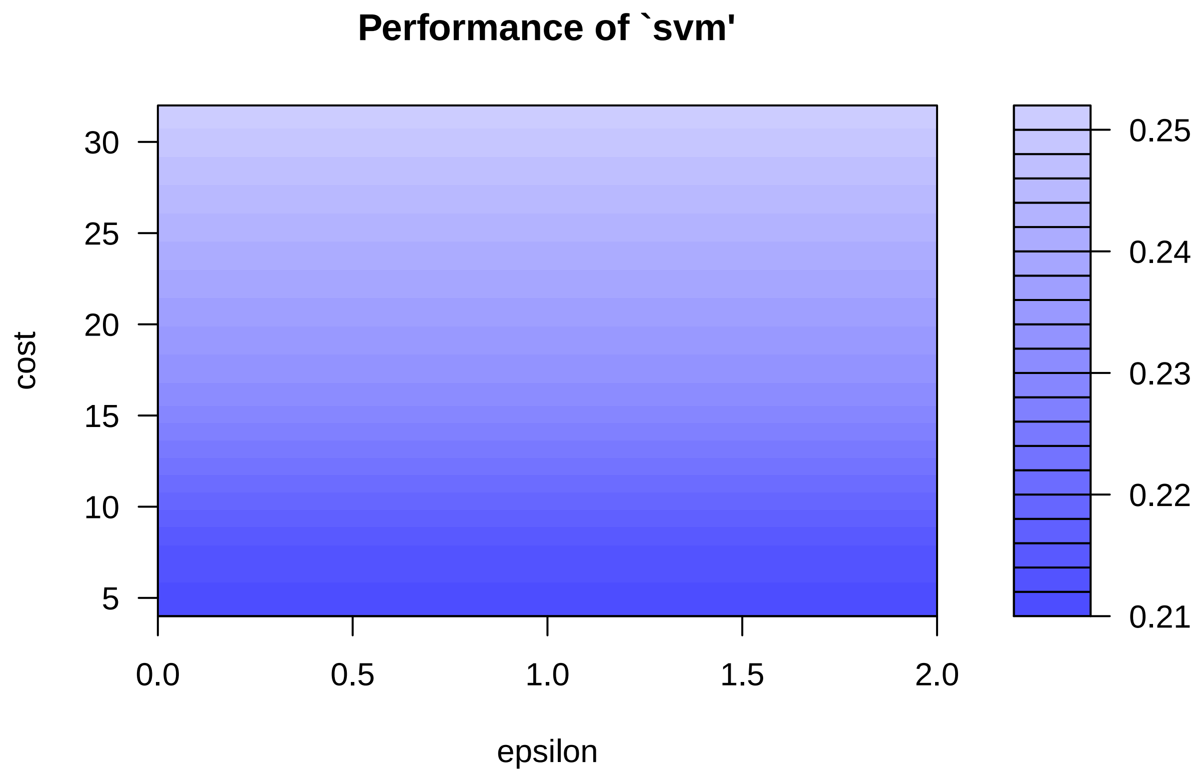
```
##
## Call:
## svm(formula = target ~ ., data = train_data, kernel = "linear")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  1
##
## Number of Support Vectors:  95
##
##  ( 49 46 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1
```

```r
# Make predictions using the model and compare the outcome with the 'target' values
## in the dataset. Summarize the prediction accuracy and related statistics using
## a confusion matrix
set.seed(123)
prediction_svm1 = predict(svm_model, newdata = test_data)
confusionMatrix(prediction_svm1, test_data$target)
```

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

```
##           Reference
## Prediction  0  1
##          0 29  3
##          1  4 29
##
##                Accuracy : 0.8923
##                  95% CI : (0.7906, 0.9556)
##     No Information Rate : 0.5077
##     P-Value [Acc > NIR] : 4.663e-11
##
##                   Kappa : 0.7847
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.8788
##             Specificity : 0.9062
##          Pos Pred Value : 0.9062
##          Neg Pred Value : 0.8788
##              Prevalence : 0.5077
##          Detection Rate : 0.4462
##    Detection Prevalence : 0.4923
##       Balanced Accuracy : 0.8925
##
##        'Positive' Class : 0
##
```

```r
# Use the 'tune' formula to figure out the best parameters for the SVM model to
## boost the model performance. The darker the color is, the better the model will
## perform, as indicated by the 'cost' y-axis label.
set.seed(123)
tune_svm = tune(svm, target ~ ., data = train_data, range = list(epsilon = seq(0, 2, 0.1), cost = 2^(2:
plot(tune_svm)
```




```r
# Summarize the tuned model
summary(tune_svm)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  epsilon cost
##        0    4
##
## - best performance: 0.2101449
##
## - Detailed performance results:
##    epsilon cost     error dispersion
## 1      0.0    4 0.2101449 0.06250510
## 2      0.1    4 0.2101449 0.06250510
## 3      0.2    4 0.2101449 0.06250510
## 4      0.3    4 0.2101449 0.06250510
## 5      0.4    4 0.2101449 0.06250510
## 6      0.5    4 0.2101449 0.06250510
## 7      0.6    4 0.2101449 0.06250510
## 8      0.7    4 0.2101449 0.06250510
## 9      0.8    4 0.2101449 0.06250510
## 10     0.9    4 0.2101449 0.06250510
```

```
## 11     1.0    4 0.2101449 0.06250510
## 12     1.1    4 0.2101449 0.06250510
## 13     1.2    4 0.2101449 0.06250510
## 14     1.3    4 0.2101449 0.06250510
## 15     1.4    4 0.2101449 0.06250510
## 16     1.5    4 0.2101449 0.06250510
## 17     1.6    4 0.2101449 0.06250510
## 18     1.7    4 0.2101449 0.06250510
## 19     1.8    4 0.2101449 0.06250510
## 20     1.9    4 0.2101449 0.06250510
## 21     2.0    4 0.2101449 0.06250510
## 22     0.0    8 0.2141304 0.06293997
## 23     0.1    8 0.2141304 0.06293997
## 24     0.2    8 0.2141304 0.06293997
## 25     0.3    8 0.2141304 0.06293997
## 26     0.4    8 0.2141304 0.06293997
## 27     0.5    8 0.2141304 0.06293997
## 28     0.6    8 0.2141304 0.06293997
## 29     0.7    8 0.2141304 0.06293997
## 30     0.8    8 0.2141304 0.06293997
## 31     0.9    8 0.2141304 0.06293997
## 32     1.0    8 0.2141304 0.06293997
## 33     1.1    8 0.2141304 0.06293997
## 34     1.2    8 0.2141304 0.06293997
## 35     1.3    8 0.2141304 0.06293997
## 36     1.4    8 0.2141304 0.06293997
## 37     1.5    8 0.2141304 0.06293997
## 38     1.6    8 0.2141304 0.06293997
## 39     1.7    8 0.2141304 0.06293997
## 40     1.8    8 0.2141304 0.06293997
## 41     1.9    8 0.2141304 0.06293997
## 42     2.0    8 0.2141304 0.06293997
## 43     0.0   16 0.2309783 0.05255532
## 44     0.1   16 0.2309783 0.05255532
## 45     0.2   16 0.2309783 0.05255532
## 46     0.3   16 0.2309783 0.05255532
## 47     0.4   16 0.2309783 0.05255532
## 48     0.5   16 0.2309783 0.05255532
## 49     0.6   16 0.2309783 0.05255532
## 50     0.7   16 0.2309783 0.05255532
## 51     0.8   16 0.2309783 0.05255532
## 52     0.9   16 0.2309783 0.05255532
## 53     1.0   16 0.2309783 0.05255532
## 54     1.1   16 0.2309783 0.05255532
## 55     1.2   16 0.2309783 0.05255532
## 56     1.3   16 0.2309783 0.05255532
## 57     1.4   16 0.2309783 0.05255532
## 58     1.5   16 0.2309783 0.05255532
## 59     1.6   16 0.2309783 0.05255532
## 60     1.7   16 0.2309783 0.05255532
## 61     1.8   16 0.2309783 0.05255532
## 62     1.9   16 0.2309783 0.05255532
## 63     2.0   16 0.2309783 0.05255532
## 64     0.0   32 0.2516304 0.07470590
```

```
## 65     0.1   32 0.2516304 0.07470590
## 66     0.2   32 0.2516304 0.07470590
## 67     0.3   32 0.2516304 0.07470590
## 68     0.4   32 0.2516304 0.07470590
## 69     0.5   32 0.2516304 0.07470590
## 70     0.6   32 0.2516304 0.07470590
## 71     0.7   32 0.2516304 0.07470590
## 72     0.8   32 0.2516304 0.07470590
## 73     0.9   32 0.2516304 0.07470590
## 74     1.0   32 0.2516304 0.07470590
## 75     1.1   32 0.2516304 0.07470590
## 76     1.2   32 0.2516304 0.07470590
## 77     1.3   32 0.2516304 0.07470590
## 78     1.4   32 0.2516304 0.07470590
## 79     1.5   32 0.2516304 0.07470590
## 80     1.6   32 0.2516304 0.07470590
## 81     1.7   32 0.2516304 0.07470590
## 82     1.8   32 0.2516304 0.07470590
## 83     1.9   32 0.2516304 0.07470590
## 84     2.0   32 0.2516304 0.07470590
```

```r
# Use the best model tuned by the function and set it as our final model
set.seed(123)
final_svm = tune_svm$best.model
summary(final_svm)
```

```
##
## Call:
## best.tune(method = svm, train.x = target ~ ., data = train_data,
##     ranges = list(epsilon = seq(0, 2, 0.1), cost = 2^(2:5)))
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  4
##
## Number of Support Vectors:  120
##
##  ( 62 58 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1
```

```r
# Use the tuned model to make prediction and compare the accuracy with the previous
## model. Since the prediction accuracy increased from 0.8779 to 0.9241, we can
## conclude that the 'tune' function did a great job identifying the best model
prediction_svm2 = predict(final_svm, newdata = test_data)
confusionMatrix(prediction_svm2, test_data$target)
```

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

```
##
##           Reference
## Prediction  0  1
##          0 29  5
##          1  4 27
##
##                Accuracy : 0.8615
##                  95% CI : (0.7534, 0.9347)
##     No Information Rate : 0.5077
##     P-Value [Acc > NIR] : 2.107e-09
##
##                   Kappa : 0.7229
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.8788
##             Specificity : 0.8438
##          Pos Pred Value : 0.8529
##          Neg Pred Value : 0.8710
##              Prevalence : 0.5077
##          Detection Rate : 0.4462
##    Detection Prevalence : 0.5231
##       Balanced Accuracy : 0.8613
##
##        'Positive' Class : 0
##
```

—————————————- Classification Tree —————————————-

```r
# Build a classification tree model to predict the target. I used a tree control
## parameter 'mincriterion.' The value of this parameter will be considered as
## 1 - p-value that must be exceeded in order to implement a node split.
tree_model1 = ctree(target~., data = train_data, controls = ctree_control(mincriterion = 0.95))
summary(tree_model1)
```

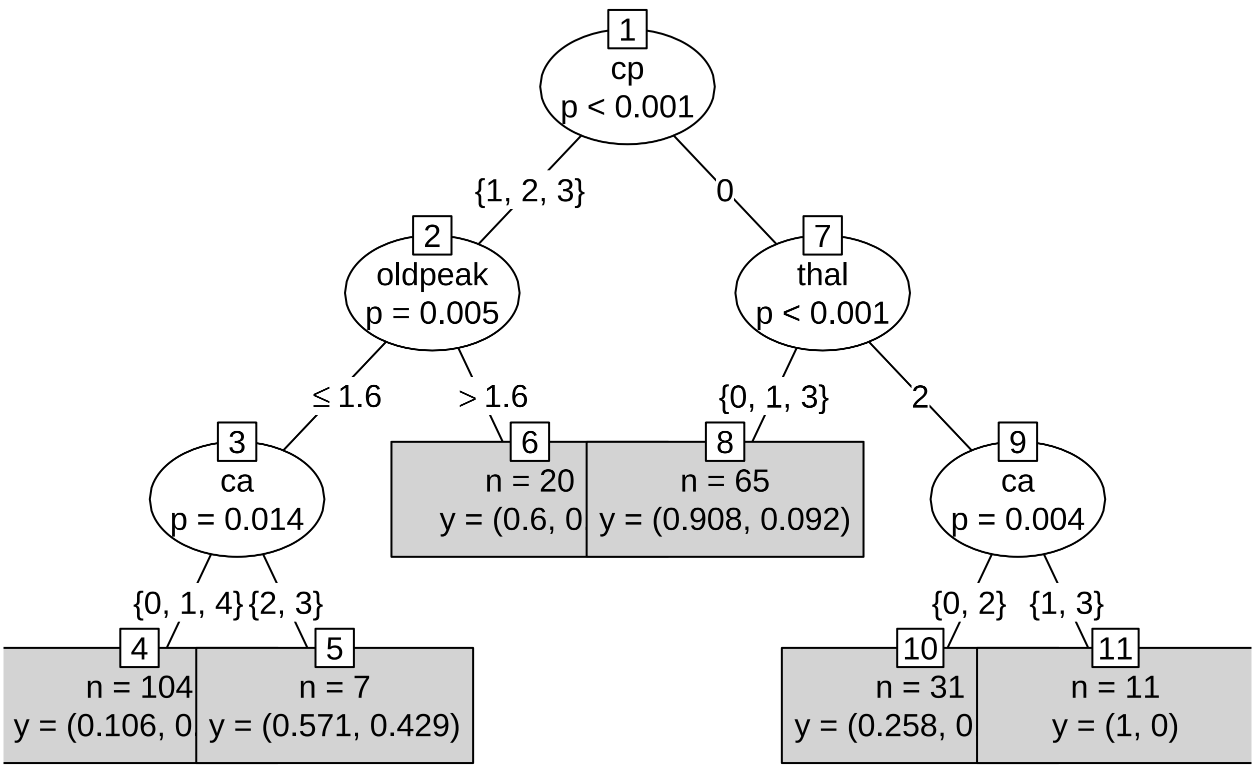
```
##     Length      Class       Mode
##          1 BinaryTree         S4
```

```r
tree_model1
```

```
##
##   Conditional inference tree with 6 terminal nodes
##
## Response:  target
## Inputs:  age, sex, cp, trestbps, chol, fbs, restecg, thalach, exang, oldpeak, slope, ca, thal
## Number of observations:  238
##
## 1) cp == {1, 2, 3}; criterion = 1, statistic = 65.752
##   2) oldpeak <= 1.6; criterion = 0.995, statistic = 19.873
##     3) ca == {0, 1, 4}; criterion = 0.986, statistic = 18.274
##       4)*  weights = 104
##     3) ca == {2, 3}
##       5)*  weights = 7
```

```
##   2) oldpeak > 1.6
##     6)*  weights = 20
## 1) cp == {0}
##   7) thal == {0, 1, 3}; criterion = 1, statistic = 26.741
##     8)*  weights = 65
##   7) thal == {2}
##     9) ca == {0, 2}; criterion = 0.996, statistic = 18.67
##       10)*  weights = 31
##     9) ca == {1, 3}
##       11)*  weights = 11
```

```r
# Plot the tree model built
plot(tree_model1, type = 'simple')
```



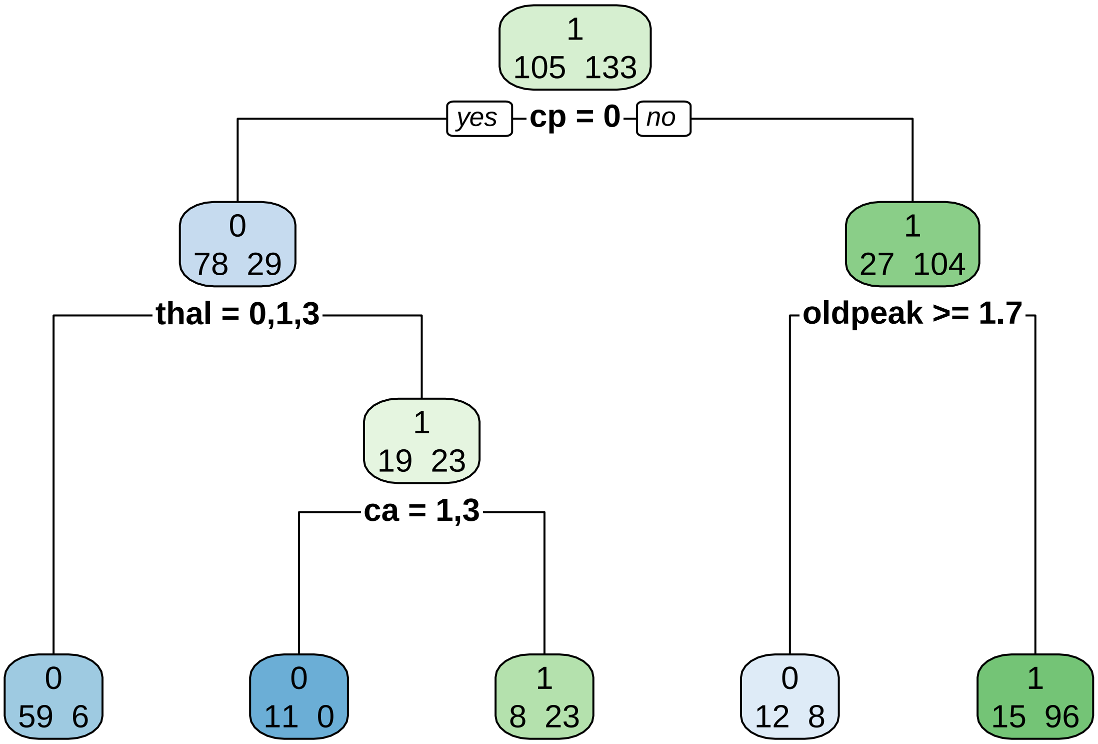

```r
# Make prediction using the tree model and build a confusion matrix to evaluate
## its prediction accuracy
prediction_tree1 = predict(tree_model1, newdata = test_data)
confusionMatrix(prediction_tree1, test_data$target)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 24 10
##          1  9 22
```

```
##
##                Accuracy : 0.7077
##                  95% CI : (0.5817, 0.814)
##     No Information Rate : 0.5077
##     P-Value [Acc > NIR] : 0.0008348
##
##                   Kappa : 0.415
##
##  Mcnemar's Test P-Value : 1.0000000
##
##             Sensitivity : 0.7273
##             Specificity : 0.6875
##          Pos Pred Value : 0.7059
##          Neg Pred Value : 0.7097
##              Prevalence : 0.5077
##          Detection Rate : 0.3692
##    Detection Prevalence : 0.5231
##       Balanced Accuracy : 0.7074
##
##        'Positive' Class : 0
##
```

```r
# Build another tree model using a different package
tree_model2 = rpart(target ~ ., data = train_data)
```

```r
# Plot the tree at a certain level of detail
rpart.plot(tree_model2, extra = 1)
```




```r
# Make prediction using the second tree model and build a confusion matrix to evaluate
## the prediction accuracy
prediction_tree2 = predict(tree_model2, newdata = test_data, type = 'class')
confusionMatrix(prediction_tree2, test_data$target)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 24  8
##          1  9 24
##
##                Accuracy : 0.7385
##                  95% CI : (0.6146, 0.8397)
##     No Information Rate : 0.5077
##     P-Value [Acc > NIR] : 0.0001234
##
##                   Kappa : 0.477
##
##  Mcnemar's Test P-Value : 1.0000000
##
##             Sensitivity : 0.7273
##             Specificity : 0.7500
##          Pos Pred Value : 0.7500
##          Neg Pred Value : 0.7273
##              Prevalence : 0.5077
```

```
##          Detection Rate : 0.3692
##    Detection Prevalence : 0.4923
##       Balanced Accuracy : 0.7386
##
##        'Positive' Class : 0
##
```

Random Forest

```r
# Build a random forest model to predict the 'target' variable in the dataset. I
## started with a huge number of trees (ntree) so that, based on the plot later,
## we can easily identify the number of trees that leads to least prediction error
set.seed(123)
rf_model1 = randomForest(target ~ ., data = train_data, ntree = 2000)
print(rf_model1)
```

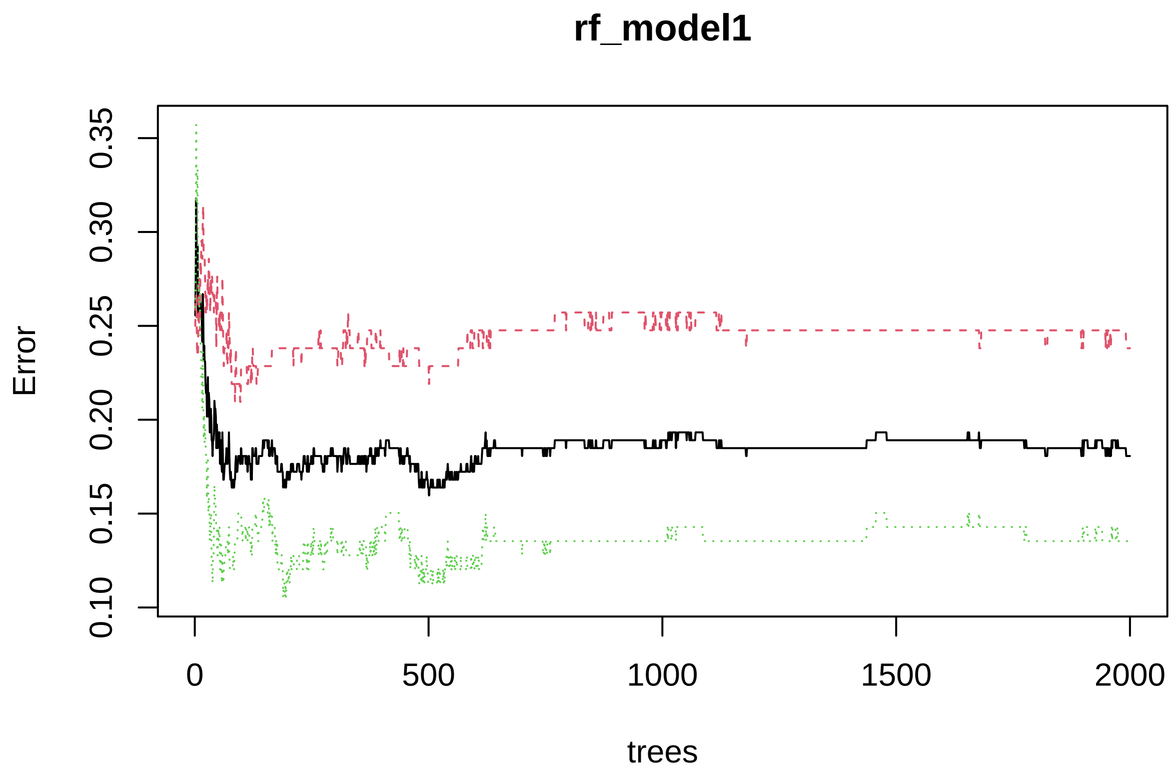
```
##
## Call:
##  randomForest(formula = target ~ ., data = train_data, ntree = 2000)
##                Type of random forest: classification
##                      Number of trees: 2000
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 18.07%
## Confusion matrix:
##    0   1 class.error
## 0 80  25   0.2380952
## 1 18 115   0.1353383
```

```r
# Use the random forest model to make prediction and build a confusion matrix to
## evaluate the prediction accuracy
prediction_rf1 = predict(rf_model1, newdata = test_data)
confusionMatrix(prediction_rf1, test_data$target)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 30  4
##          1  3 28
##
##                Accuracy : 0.8923
##                  95% CI : (0.7906, 0.9556)
##     No Information Rate : 0.5077
##     P-Value [Acc > NIR] : 4.663e-11
##
##                   Kappa : 0.7845
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.9091
##             Specificity : 0.8750
##          Pos Pred Value : 0.8824
```

```
##          Neg Pred Value : 0.9032
##              Prevalence : 0.5077
##          Detection Rate : 0.4615
##    Detection Prevalence : 0.5231
##       Balanced Accuracy : 0.8920
##
##        'Positive' Class : 0
##
```

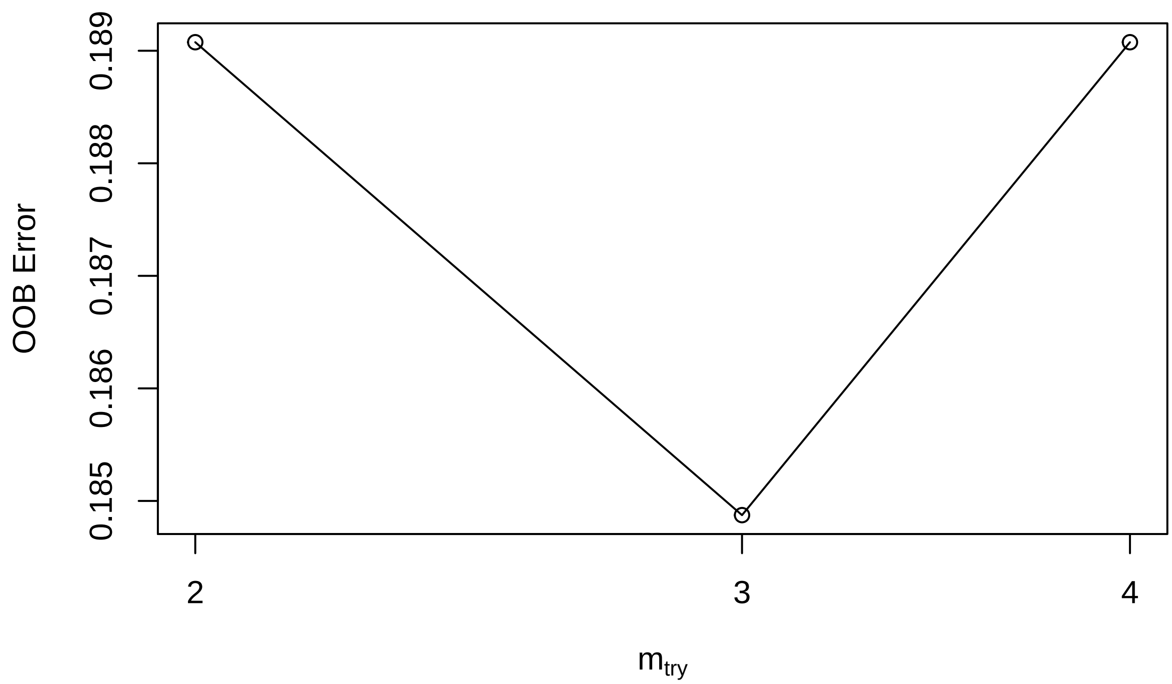
```r
# Plot the relationship between the number of trees and the prediction error. We
## can see that the error reaches the lowest point when the number of trees is
## around 750. Therefore, I will use this number to build a new model later to see
## if it does a great job predicting
plot(rf_model1)
```




```r
# Use the 'tuneRF' function to figure out the 'mtry' parameter that leads to least
## prediction error. 'mtry' is the number of variables randomly sampled as candidates
## at each split of node. According to the plot, an 'mtry' of three leads to the
## random forest model the predicts most accurately
set.seed(123)
tune_rf1 = tuneRF(train_data[, -14], train_data[, 14], stepFactor = 1.5, plot = TRUE, ntreeTry = 750, t
```

```
## mtry = 3  OOB error = 18.49%
## Searching left ...
## mtry = 2     OOB error = 18.91%
```

```
## -0.02272727 0.01
## Searching right ...
## mtry = 4     OOB error = 18.91%
## -0.02272727 0.01
```



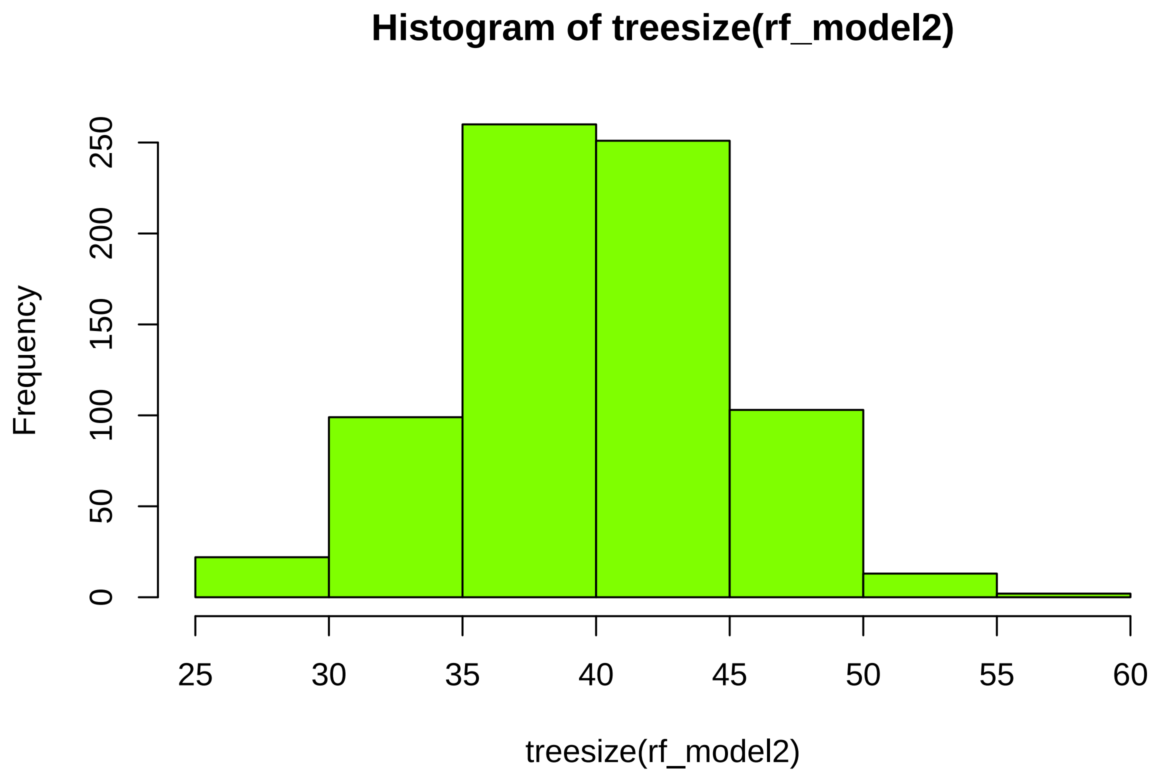

```
# Build a new model using the parameters we just figured out. We can see that the
## Out Of Bag (OBB) estimate of error rate decreases from 16.17% to 15.84%, meaning
## that the functions did a great job identifying the best parameters
set.seed(123)
rf_model2 = randomForest(target ~ ., data = train_data, ntree = 750, mtry = 3, importance = TRUE, proxi
print(rf_model2)
```

```
##
## Call:
##  randomForest(formula = target ~ ., data = train_data, ntree = 750,      mtry = 3, importance = TRUE
##                Type of random forest: classification
##                      Number of trees: 750
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 18.49%
## Confusion matrix:
##    0   1 class.error
## 0 79  26   0.2476190
## 1 18 115   0.1353383
```

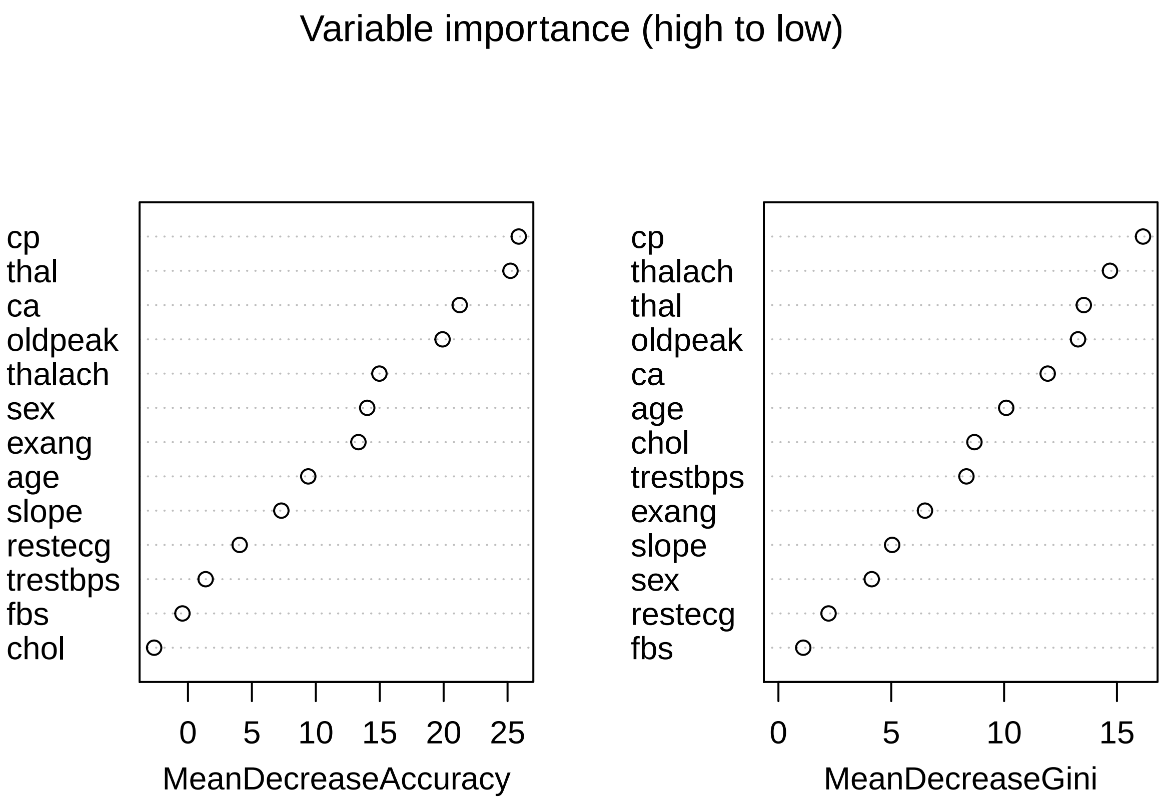
```r
# Build a confusion matrix for more detailed statistics about the model performance
prediction_rf2 = predict(rf_model2, newdata = test_data)
confusionMatrix(prediction_rf2, test_data$target)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 31  4
##          1  2 28
##
##                Accuracy : 0.9077
##                  95% CI : (0.8098, 0.9654)
##     No Information Rate : 0.5077
##     P-Value [Acc > NIR] : 5.586e-12
##
##                   Kappa : 0.8152
##
##  Mcnemar's Test P-Value : 0.6831
##
##             Sensitivity : 0.9394
##             Specificity : 0.8750
##          Pos Pred Value : 0.8857
##          Neg Pred Value : 0.9333
##              Prevalence : 0.5077
##          Detection Rate : 0.4769
##    Detection Prevalence : 0.5385
##       Balanced Accuracy : 0.9072
##
##        'Positive' Class : 0
##
```

```r
# Plot the distribution of tree size to better understand the model
hist(treesize(rf_model2), col = 'chartreuse1')
```



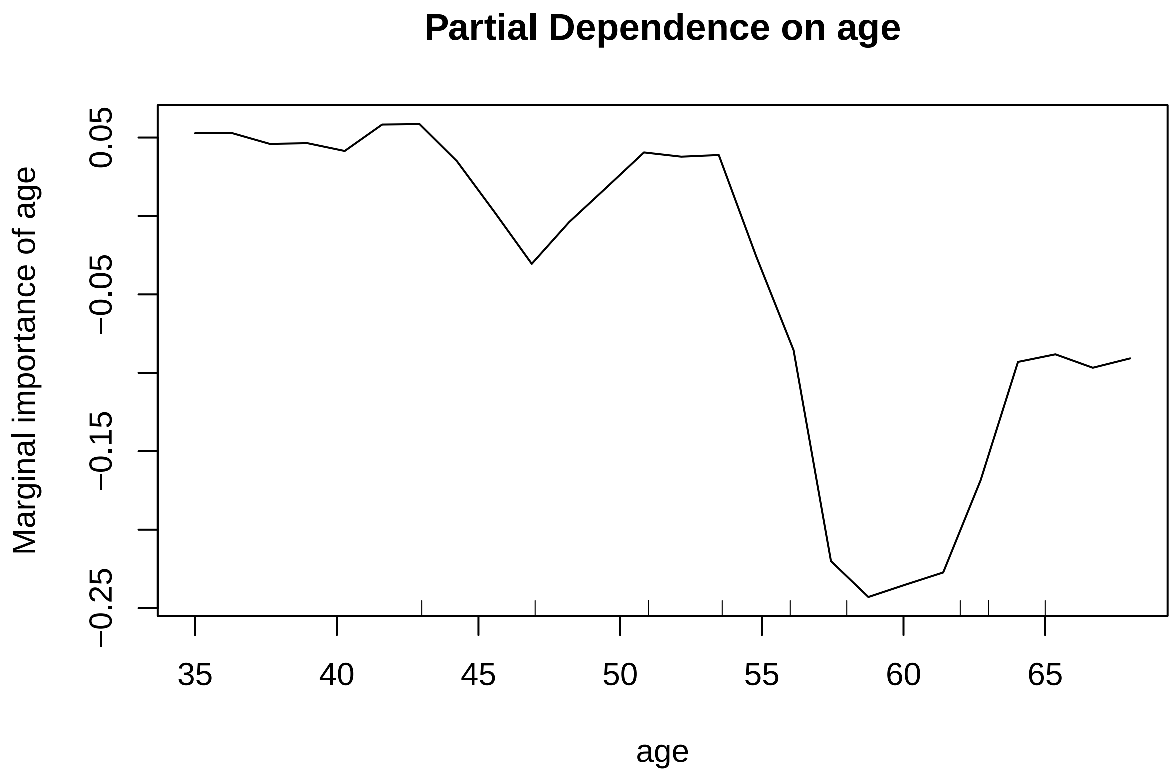

```r
# Plot all the variables in the dataset and sort them based on their relative
## importance when making the prediction. The first plot gives information about
## how much prediction accuracy will decrease if we remove the variable. For example,
## if we remove 'ca,' the prediction accuracy will decrease by 30%. The second plot
## shows how pure the nodes are at the end of the tree, if the variable is removed.
varImpPlot(rf_model2, main = 'Variable importance (high to low)')
```




```r
# To know how many times each column is used in the entire random forest, we can
## use the 'varUsed' function
varUsed(rf_model2)
```

```
##  [1] 3678 1165 2346 3420 3586  611 1187 3873 1064 3399 1411 2116 1666
```

```r
# To understand the marginal effect of a variable on the final prediction result,
## we can use the partial dependence plot. For example, this plot shows that, when
## age is greater than 53, the random forest is much less likely to predict
## 1 as the target for that record.
partialPlot(x = rf_model2, pred.data = test_data, x.var = age, which.class = '1',
            ylab = 'Marginal importance of age')
```



K-Nearest Neighbor (KNN)

```
# Build a KNN model to predict the 'target'. I started by defining the training
## controls. Here, I will evaluate the KNN model using repeated 10-fold cross
## validation repeated for three times.
trCrl1 = trainControl(method = 'repeatedcv', number = 10, repeats = 5)
```

```
# Then, we train the model using the KNN method. Since different fields in the data
## may have different unit, we pre-processed the data by first minus the mean value
## from each single value. Then, we divide the result by the standard deviation.
## This standardization should result in data that falls within a scope from -3 to 3.
## Only numeric values, not factors, are standardized. The tuneLength means the
## number of K we want to test. To figure out the best number of K, we set it to
## a fairly large number.
set.seed(123)
knn1 = train(data = train_data, target ~ ., method = 'knn', tuneLength = 100,
             trControl = trCrl1, preProcess = c('center', 'scale'))
```

```
# Here, we summarize the model. The result shows that, when k equals 57, the model
## has the highest prediction accuracy. Therefore, if we were to judge the model
## by its prediction accuracy, we get the best model when k equals to 57.
knn1
```

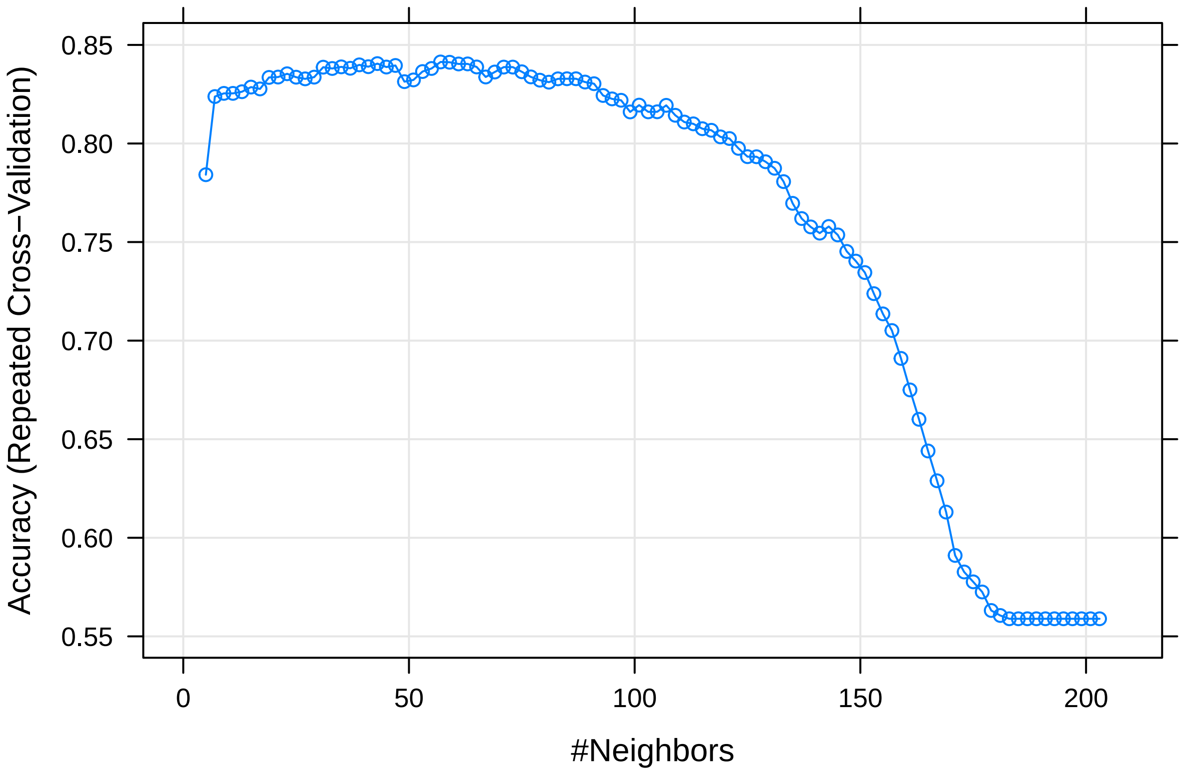
```
## k-Nearest Neighbors
##
## 238 samples
```

```
##  13 predictor
##   2 classes: '0', '1'
##
## Pre-processing: centered (22), scaled (22)
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 214, 214, 214, 215, 213, 215, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    5  0.7841580  0.562981114
##    7  0.8238188  0.641790464
##    9  0.8254159  0.645005378
##   11  0.8254159  0.644930992
##   13  0.8262522  0.647451481
##   15  0.8286159  0.651469667
##   17  0.8276768  0.649410796
##   19  0.8335188  0.660993153
##   21  0.8336942  0.661566143
##   23  0.8353638  0.665085408
##   25  0.8336246  0.661614973
##   27  0.8327913  0.660115738
##   29  0.8336580  0.662125729
##   31  0.8387029  0.672004181
##   33  0.8380420  0.670245143
##   35  0.8388754  0.671625625
##   37  0.8381478  0.669711389
##   39  0.8398870  0.672744217
##   41  0.8389536  0.670226663
##   43  0.8405174  0.673177674
##   45  0.8387783  0.668593700
##   47  0.8395783  0.669984230
##   49  0.8313754  0.653150418
##   51  0.8322449  0.655019275
##   53  0.8364870  0.663586244
##   55  0.8380507  0.666534996
##   57  0.8413536  0.673185172
##   59  0.8412145  0.672501357
##   61  0.8403449  0.669951063
##   63  0.8403783  0.670405329
##   65  0.8388145  0.667054945
##   67  0.8337362  0.656236811
##   69  0.8362449  0.661295058
##   71  0.8387449  0.666572805
##   73  0.8387449  0.666430792
##   75  0.8363812  0.661247492
##   77  0.8337754  0.655324557
##   79  0.8321058  0.651262718
##   81  0.8310971  0.648947998
##   83  0.8328029  0.651978988
##   85  0.8328362  0.652072070
##   87  0.8328362  0.652300325
##   89  0.8311696  0.648662858
##   91  0.8303333  0.646826961
##   93  0.8243580  0.633226628
```

```
##    95  0.8226188  0.629575182
##    97  0.8219246  0.628024262
##    99  0.8160130  0.615396474
##   101  0.8194551  0.622762950
##   103  0.8160493  0.615353243
##   105  0.8160826  0.615190963
##   107  0.8193493  0.621921074
##   109  0.8143130  0.610752711
##   111  0.8108710  0.602598153
##   113  0.8099652  0.600286490
##   115  0.8075014  0.594830925
##   117  0.8066986  0.593031288
##   119  0.8033623  0.585777498
##   121  0.8025261  0.583287862
##   123  0.7975203  0.572160327
##   125  0.7933145  0.563137689
##   127  0.7932812  0.562836731
##   129  0.7907420  0.557274126
##   131  0.7874362  0.549800447
##   133  0.7806609  0.534464094
##   135  0.7696464  0.510226951
##   137  0.7619290  0.492395437
##   139  0.7576870  0.482781327
##   141  0.7545174  0.474263361
##   143  0.7578928  0.480523673
##   145  0.7536145  0.470044369
##   147  0.7452420  0.450230819
##   149  0.7403391  0.438753681
##   151  0.7345362  0.425152332
##   153  0.7238304  0.400410188
##   155  0.7135768  0.376289224
##   157  0.7051319  0.356686692
##   159  0.6909841  0.322001547
##   161  0.6750000  0.284400667
##   163  0.6600855  0.248535848
##   165  0.6440290  0.210305717
##   167  0.6289087  0.174375843
##   169  0.6130275  0.134540074
##   171  0.5910290  0.080125606
##   173  0.5826565  0.059812423
##   175  0.5776536  0.047490311
##   177  0.5725087  0.034455960
##   179  0.5631188  0.010731698
##   181  0.5605826  0.004249969
##   183  0.5589159  0.000000000
##   185  0.5589159  0.000000000
##   187  0.5589159  0.000000000
##   189  0.5589159  0.000000000
##   191  0.5589159  0.000000000
##   193  0.5589159  0.000000000
##   195  0.5589159  0.000000000
##   197  0.5589159  0.000000000
##   199  0.5589159  0.000000000
##   201  0.5589159  0.000000000
```

```
##   203  0.5589159  0.000000000
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 57.
```

```r
# We plot the relationship between k and the model's prediction accuracy. We can
## see that, when k equals to 57, the model performs the best.
plot(knn1)
```




```r
# To better understand the model in terms of which field plays the most important role,
## we use the varImp function to sort the importance of different fields in a
## descending order.
varImp(knn1)
```

```
## ROC curve variable importance
##
##          Importance
## thalach      100.00
## cp            96.01
## oldpeak       88.17
## exang         80.28
## ca            79.05
## thal          77.59
## slope         67.04
## sex           53.90
```

```
## age           52.49
## restecg       26.93
## trestbps      24.56
## chol          12.35
## fbs            0.00
```

```r
# We predict the target using the model and build a confusion matrix. The result
## shows a prediction accuracy of 84.62%.
confusionMatrix(predict(knn1, newdata = test_data), test_data$target)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 25  2
##          1  8 30
##
##                Accuracy : 0.8462
##                  95% CI : (0.7352, 0.9237)
##     No Information Rate : 0.5077
##     P-Value [Acc > NIR] : 1.173e-08
##
##                   Kappa : 0.6931
##
##  Mcnemar's Test P-Value : 0.1138
##
##             Sensitivity : 0.7576
##             Specificity : 0.9375
##          Pos Pred Value : 0.9259
##          Neg Pred Value : 0.7895
##              Prevalence : 0.5077
##          Detection Rate : 0.3846
##    Detection Prevalence : 0.4154
##       Balanced Accuracy : 0.8475
##
##        'Positive' Class : 0
##
```

```r
# Other than accuracy, ROC is another common way to evaluate the predictive performance
## of models. I will use ROC to evaluate our KNN model to see if a different K is
## chosen.

set.seed(123)
train_data1 = train_data
test_data1 = test_data
train_data1$target = as.integer(train_data1$target)
test_data1$target = as.integer(test_data1$target)
train_data1$target[train_data1$target == 1] = 'No'
train_data1$target[train_data1$target == 2] = 'Yes'
test_data1$target[test_data1$target == 1] = 'No'
test_data1$target[test_data1$target == 2] = 'Yes'

trCrl2 = trainControl(method = 'repeatedcv', number = 10, repeats = 5,
```

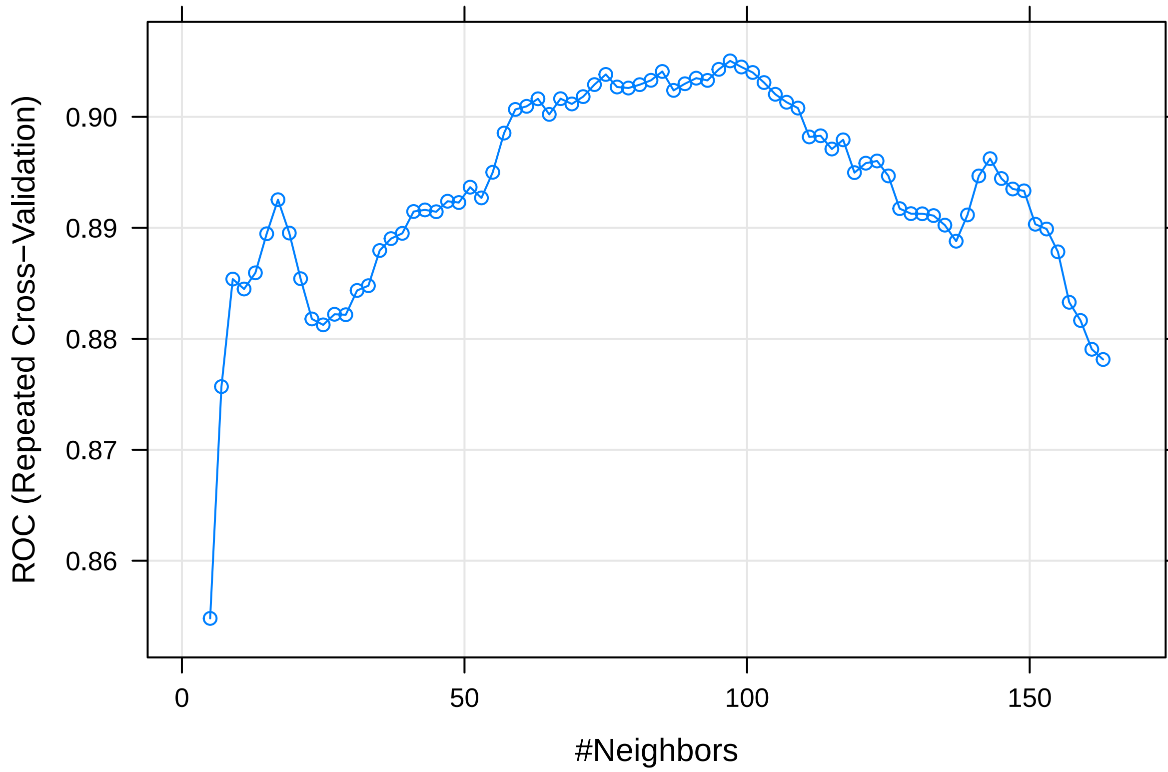
```r
                        classProbs = TRUE, summaryFunction = twoClassSummary)

knn2 = train(target ~ ., data = train_data1, method = 'knn', tuneLength = 80,
             trControl = trCrl2, preProcess = c('center', 'scale'), metric = 'ROC')
```

```r
# The larger the area under the ROC curve, the better the model performs. Therefore,
## we can see that when K is 97, the model performs the best.
knn2
```

```
## k-Nearest Neighbors
##
## 238 samples
##  13 predictor
##   2 classes: 'No', 'Yes'
##
## Pre-processing: centered (22), scaled (22)
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 214, 214, 214, 215, 213, 215, ...
## Resampling results across tuning parameters:
##
##   k    ROC        Sens       Spec
##     5  0.8547987  0.7576364  0.8054945
##     7  0.8756908  0.7849091  0.8552747
##     9  0.8853786  0.7827273  0.8612088
##    11  0.8844855  0.7807273  0.8626374
##    13  0.8859431  0.7885455  0.8567033
##    15  0.8894635  0.7809091  0.8671429
##    17  0.8925305  0.7792727  0.8668132
##    19  0.8895285  0.7812727  0.8756044
##    21  0.8854131  0.7852727  0.8728571
##    23  0.8817862  0.7872727  0.8742857
##    25  0.8812537  0.7852727  0.8727473
##    27  0.8822068  0.7834545  0.8727473
##    29  0.8821663  0.7872727  0.8713187
##    31  0.8843541  0.7894545  0.8787912
##    33  0.8847827  0.7854545  0.8805495
##    35  0.8879525  0.7854545  0.8820879
##    37  0.8890255  0.7778182  0.8867033
##    39  0.8895005  0.7778182  0.8897802
##    41  0.8914680  0.7701818  0.8941758
##    43  0.8916134  0.7700000  0.8970330
##    45  0.8914481  0.7583636  0.9029670
##    47  0.8923946  0.7583636  0.9043956
##    49  0.8922752  0.7454545  0.9000000
##    51  0.8936623  0.7454545  0.9015385
##    53  0.8926938  0.7474545  0.9089011
##    55  0.8950045  0.7490909  0.9089011
##    57  0.8985365  0.7510909  0.9118681
##    59  0.9006668  0.7452727  0.9174725
##    61  0.9009476  0.7392727  0.9204396
##    63  0.9016269  0.7414545  0.9190110
##    65  0.9002208  0.7360000  0.9204396
##    67  0.9016344  0.7263636  0.9190110
```

```
##    69  0.9011608  0.7281818  0.9221978
##    71  0.9018212  0.7320000  0.9236264
##    73  0.9029051  0.7301818  0.9250549
##    75  0.9038222  0.7227273  0.9267033
##    77  0.9026878  0.7149091  0.9281319
##    79  0.9025959  0.7052727  0.9310989
##    81  0.9028971  0.7050909  0.9309890
##    83  0.9032912  0.7014545  0.9370330
##    85  0.9040879  0.7014545  0.9371429
##    87  0.9023821  0.7034545  0.9356044
##    89  0.9029825  0.6978182  0.9356044
##    91  0.9034845  0.6976364  0.9340659
##    93  0.9032822  0.6803636  0.9370330
##    95  0.9042802  0.6801818  0.9370330
##    97  0.9050420  0.6767273  0.9385714
##    99  0.9044890  0.6649091  0.9371429
##   101  0.9039960  0.6687273  0.9372527
##   103  0.9030904  0.6612727  0.9386813
##   105  0.9020315  0.6592727  0.9403297
##   107  0.9013157  0.6629091  0.9447253
##   109  0.9007897  0.6478182  0.9462637
##   111  0.8981903  0.6400000  0.9478022
##   113  0.8982937  0.6320000  0.9507692
##   115  0.8971004  0.6267273  0.9521978
##   117  0.8979261  0.6229091  0.9521978
##   119  0.8949710  0.6172727  0.9507692
##   121  0.8958232  0.6096364  0.9552747
##   123  0.8960230  0.5983636  0.9552747
##   125  0.8946768  0.5889091  0.9552747
##   127  0.8917268  0.5869091  0.9567033
##   129  0.8912757  0.5810909  0.9567033
##   131  0.8912632  0.5736364  0.9567033
##   133  0.8911004  0.5567273  0.9581319
##   135  0.8902393  0.5338182  0.9581319
##   137  0.8887912  0.5163636  0.9581319
##   139  0.8911613  0.5010909  0.9595604
##   141  0.8946738  0.4880000  0.9656044
##   143  0.8962318  0.4840000  0.9761538
##   145  0.8944401  0.4685455  0.9791209
##   147  0.8934980  0.4440000  0.9850549
##   149  0.8933317  0.4309091  0.9851648
##   151  0.8903247  0.4140000  0.9880220
##   153  0.8898906  0.3896364  0.9880220
##   155  0.8878352  0.3643636  0.9895604
##   157  0.8832902  0.3470909  0.9895604
##   159  0.8816489  0.3074545  0.9953846
##   161  0.8790554  0.2710909  0.9954945
##   163  0.8781314  0.2336364  0.9969231
##
## ROC was used to select the optimal model using the largest value.
## The final value used for the model was k = 97.
```

```r
# The plot also shows that a K equals to 97 results in the greatest area under ROC.
plot(knn2)
```




```r
# The confusion matrix shows that the prediction accuracy is 80%. In this case, ROC
## does a worse job figuring out the model with the strongest predictive power than
## the accuracy.
confusionMatrix(predict(knn2, newdata = test_data1), as.factor(test_data1$target))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction No Yes
##        No  22   2
##        Yes 11  30
##
##                Accuracy : 0.8
##                  95% CI : (0.6823, 0.889)
##     No Information Rate : 0.5077
##     P-Value [Acc > NIR] : 1.067e-06
##
##                   Kappa : 0.6016
##
##  Mcnemar's Test P-Value : 0.0265
##
##             Sensitivity : 0.6667
##             Specificity : 0.9375
##          Pos Pred Value : 0.9167
##          Neg Pred Value : 0.7317
```

```
##             Prevalence : 0.5077
##         Detection Rate : 0.3385
##   Detection Prevalence : 0.3692
##      Balanced Accuracy : 0.8021
##
##       'Positive' Class : No
##
```

Naive Bayes

```
# Load the data and print the summary and structure of the dataset
data_n = read_xlsx('C:/Users/34527/Desktop/Admission.xlsx')
summary(data_n)
```

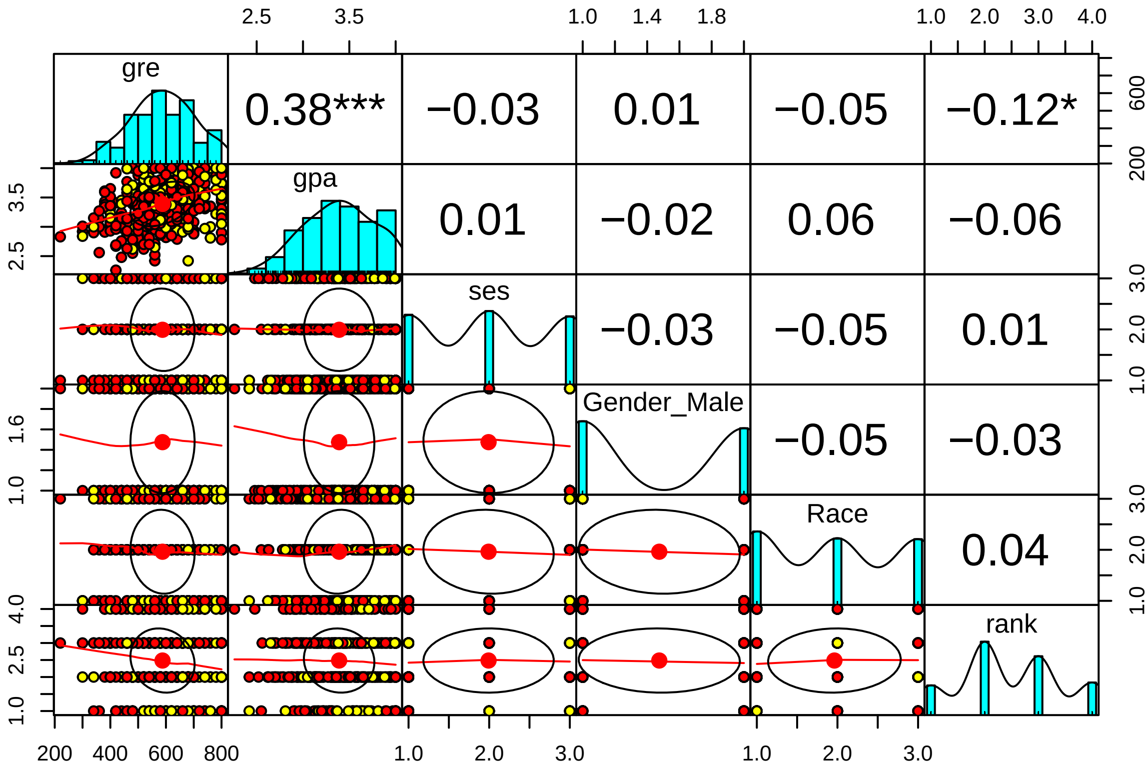
```
##      admit             gre             gpa             ses       
##  Min.   :0.0000   Min.   :220.0   Min.   :2.260   Min.   :1.000  
##  1st Qu.:0.0000   1st Qu.:520.0   1st Qu.:3.130   1st Qu.:1.000  
##  Median :0.0000   Median :580.0   Median :3.395   Median :2.000  
##  Mean   :0.3175   Mean   :587.7   Mean   :3.390   Mean   :1.992  
##  3rd Qu.:1.0000   3rd Qu.:660.0   3rd Qu.:3.670   3rd Qu.:3.000  
##  Max.   :1.0000   Max.   :800.0   Max.   :4.000   Max.   :3.000  
##   Gender_Male         Race            rank      
##  Min.   :0.000   Min.   :1.000   Min.   :1.000  
##  1st Qu.:0.000   1st Qu.:1.000   1st Qu.:2.000  
##  Median :0.000   Median :2.000   Median :2.000  
##  Mean   :0.475   Mean   :1.962   Mean   :2.485  
##  3rd Qu.:1.000   3rd Qu.:3.000   3rd Qu.:3.000  
##  Max.   :1.000   Max.   :3.000   Max.   :4.000
```

```
str(data_n)
```

```
## tibble [400 x 7] (S3: tbl_df/tbl/data.frame)
##  $ admit      : num [1:400] 0 1 1 1 0 1 1 0 1 0 ...
##  $ gre        : num [1:400] 380 660 800 640 520 760 560 400 540 700 ...
##  $ gpa        : num [1:400] 3.61 3.67 4 3.19 2.93 3 2.98 3.08 3.39 3.92 ...
##  $ ses        : num [1:400] 1 2 2 1 3 2 2 2 1 1 ...
##  $ Gender_Male: num [1:400] 0 0 0 1 1 1 1 0 1 0 ...
##  $ Race       : num [1:400] 3 2 2 2 2 1 2 2 1 2 ...
##  $ rank       : num [1:400] 3 3 1 4 4 2 1 2 3 2 ...
```

```
# Change columns to factor class
data_n$admit = as.factor(data_n$admit)
data_n$ses = as.factor(data_n$ses)
data_n$Gender_Male = as.factor(data_n$Gender_Male)
data_n$Race = as.factor(data_n$Race)
data_n$rank = as.factor(data_n$rank)
```

```
# Plot the distribution between each pair of variables to get a better understanding
## of the data
pairs.panels(data_n[, -1], gap = 0, stars = TRUE, pch = 21,
             bg = c('red', 'yellow', 'blue')[data_n$admit])
```




```r
# Split the data set into train and test sets for further model evaluation
set.seed(123)
index_n = sample(2, nrow(data_n), replace = TRUE, prob = c(0.8, 0.2))
train_n = data_n[index_n == 1, ]
test_n = data_n[index_n == 2, ]
```

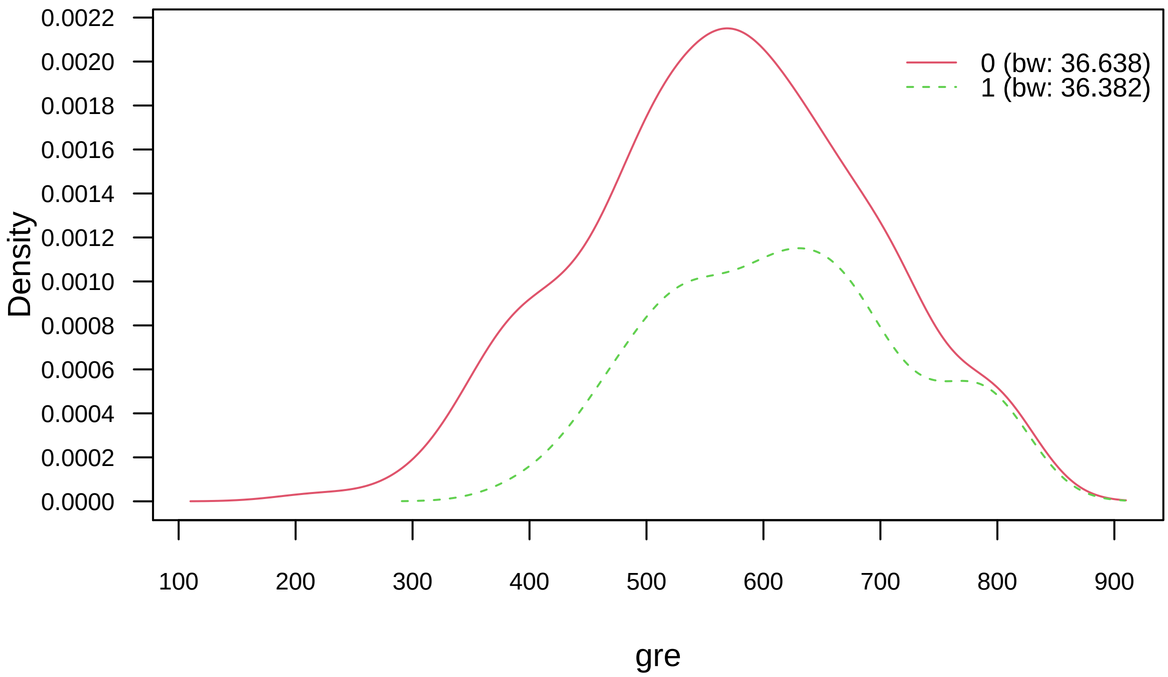
```r
# Build a Naive Bayes model using the train set and show the model
nb1 = naive_bayes(admit ~ ., data = train_n, usekernel = TRUE)
nb1
```

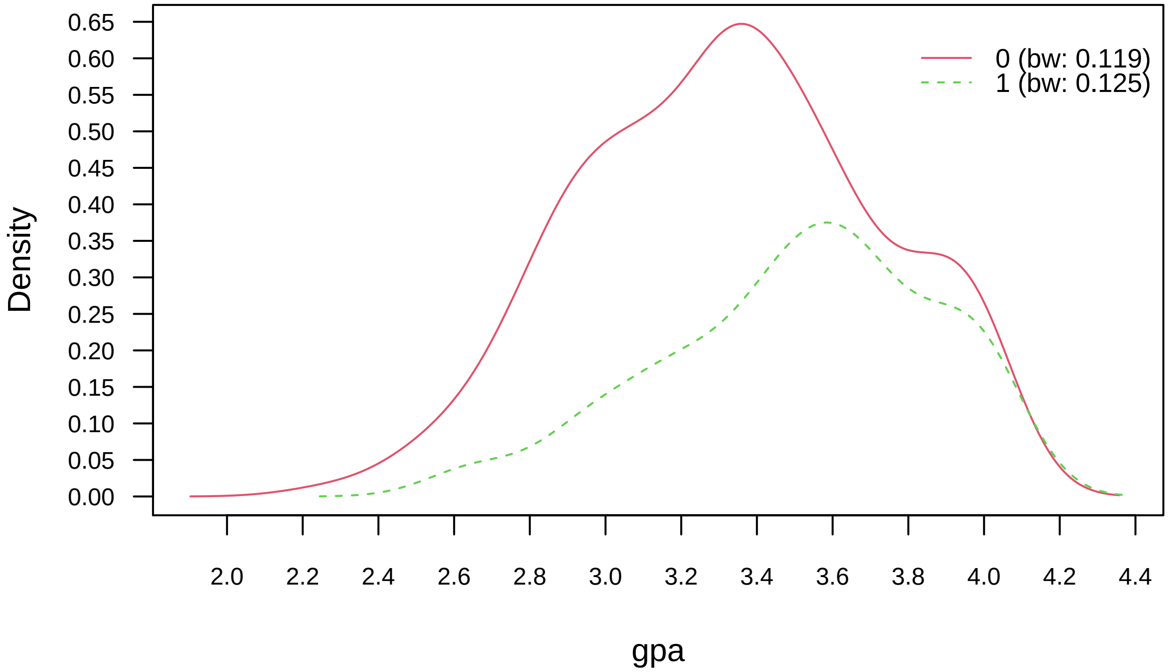
```
##
## ================================== Naive Bayes ==================================
##
##  Call:
## naive_bayes.formula(formula = admit ~ ., data = train_n, usekernel = TRUE)
##
## ---------------------------------------------------------------------------------
##
## Laplace smoothing: 0
##
## ---------------------------------------------------------------------------------
##
##  A priori probabilities:
##
##          0         1
```

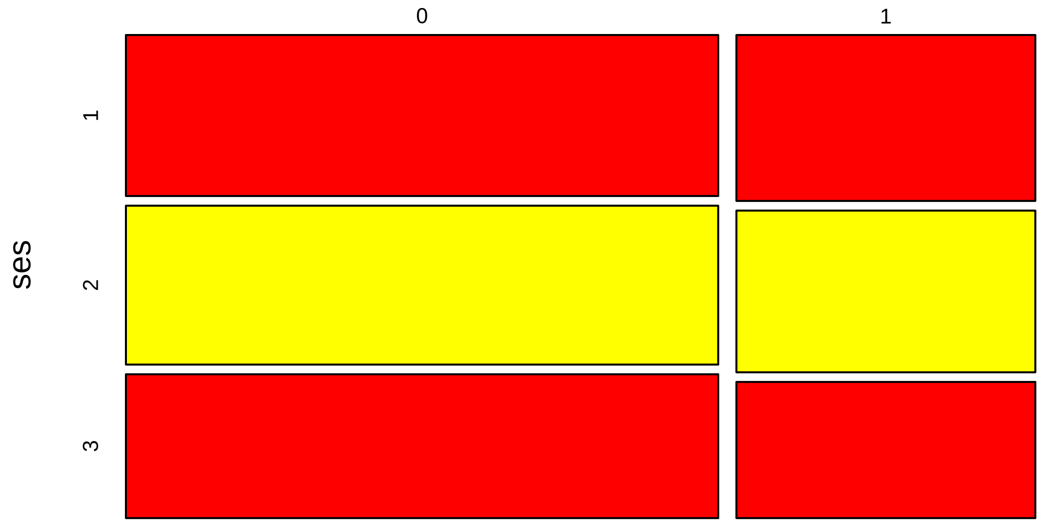
```
## 0.6646154 0.3353846
##
## ---------------------------------------------------------------------------------
##
##  Tables:
##
## ---------------------------------------------------------------------------------
##  ::: gre::0 (KDE)
## ---------------------------------------------------------------------------------
##
## Call:
##  density.default(x = x, na.rm = TRUE)
##
## Data: x (216 obs.);  Bandwidth 'bw' = 36.64
##
##        x               y
##  Min.   :110.1   Min.   :5.680e-07
##  1st Qu.:310.0   1st Qu.:1.202e-04
##  Median :510.0   Median :1.021e-03
##  Mean   :510.0   Mean   :1.249e-03
##  3rd Qu.:710.0   3rd Qu.:2.207e-03
##  Max.   :909.9   Max.   :3.236e-03
##
## ---------------------------------------------------------------------------------
##  ::: gre::1 (KDE)
## ---------------------------------------------------------------------------------
##
## Call:
##  density.default(x = x, na.rm = TRUE)
##
## Data: x (109 obs.);  Bandwidth 'bw' = 36.38
##
##        x               y
##  Min.   :290.9   Min.   :2.316e-06
##  1st Qu.:445.4   1st Qu.:3.602e-04
##  Median :600.0   Median :1.629e-03
##  Mean   :600.0   Mean   :1.616e-03
##  3rd Qu.:754.6   3rd Qu.:2.880e-03
##  Max.   :909.1   Max.   :3.432e-03
##
## ---------------------------------------------------------------------------------
##  ::: gpa::0 (KDE)
## ---------------------------------------------------------------------------------
##
## Call:
##  density.default(x = x, na.rm = TRUE)
##
## Data: x (216 obs.);  Bandwidth 'bw' = 0.1189
##
##        x               y
##  Min.   :1.903   Min.   :0.0001762
##  1st Qu.:2.517   1st Qu.:0.0542374
##  Median :3.130   Median :0.4168159
##  Mean   :3.130   Mean   :0.4071837
```

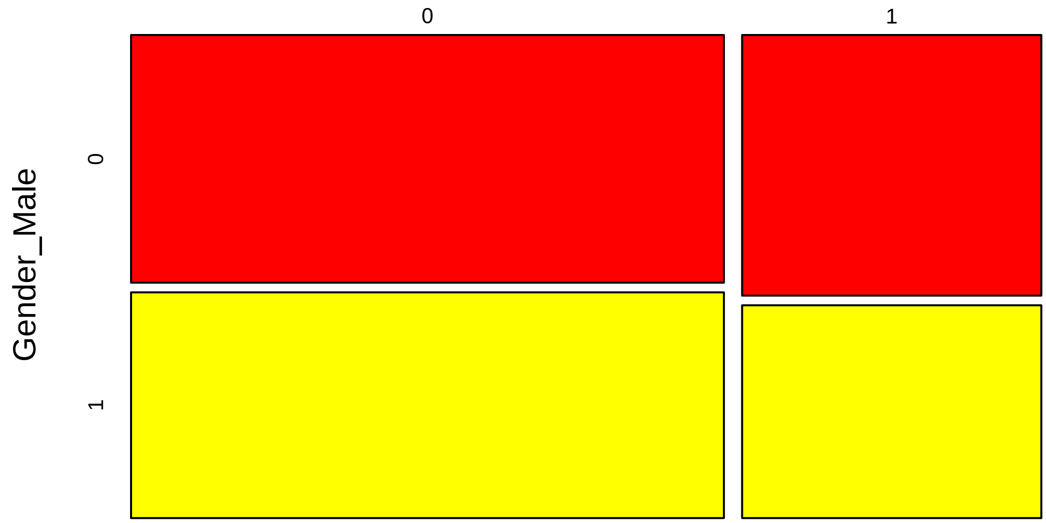
```
##  3rd Qu.:3.743   3rd Qu.:0.7184947
##  Max.   :4.357   Max.   :0.9738248
##
## --------------------------------------------------------------------------------
##  ::: gpa::1 (KDE)
## --------------------------------------------------------------------------------
##
## Call:
##  density.default(x = x, na.rm = TRUE)
##
## Data: x (109 obs.);  Bandwidth 'bw' = 0.125
##
##        x               y
##  Min.   :2.245   Min.   :0.0006475
##  1st Qu.:2.778   1st Qu.:0.1186863
##  Median :3.310   Median :0.4376137
##  Mean   :3.310   Mean   :0.4689858
##  3rd Qu.:3.842   3rd Qu.:0.7882761
##  Max.   :4.375   Max.   :1.1184635
##
## --------------------------------------------------------------------------------
##  ::: ses (Categorical)
## --------------------------------------------------------------------------------
##
## ses         0         1
##   1 0.3472222 0.3577982
##   2 0.3425926 0.3486239
##   3 0.3101852 0.2935780
##
## --------------------------------------------------------------------------------
##  ::: Gender_Male (Bernoulli)
## --------------------------------------------------------------------------------
##
## Gender_Male         0         1
##           0 0.5231481 0.5504587
##           1 0.4768519 0.4495413
##
## --------------------------------------------------------------------------------
##  ::: Race (Categorical)
## --------------------------------------------------------------------------------
##
## Race         0         1
##    1 0.3333333 0.4495413
##    2 0.3472222 0.2385321
##    3 0.3194444 0.3119266
##
## --------------------------------------------------------------------------------
##
## # ... and 1 more table
##
## --------------------------------------------------------------------------------
```

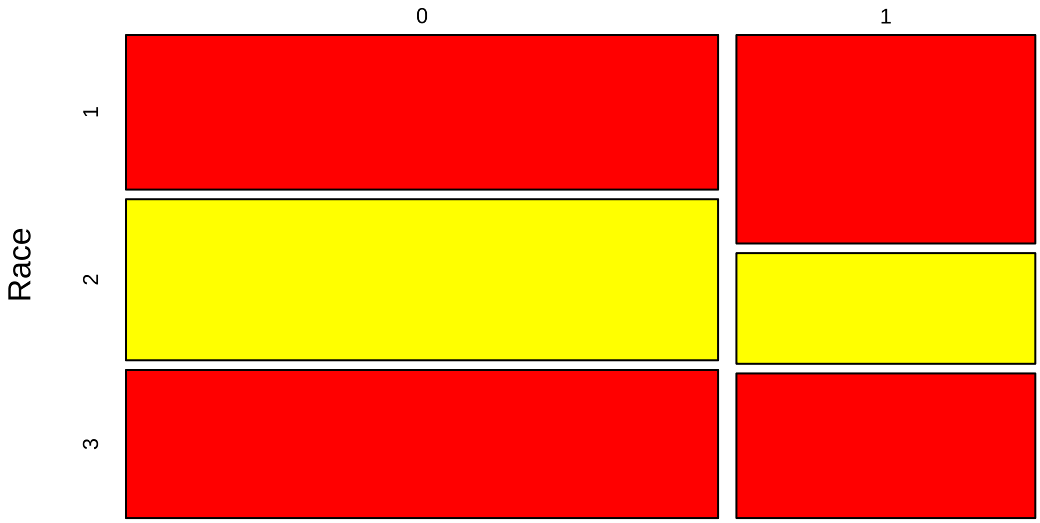
```r
# Plot the model and see how each variable is related to the target variable
plot(nb1)
```

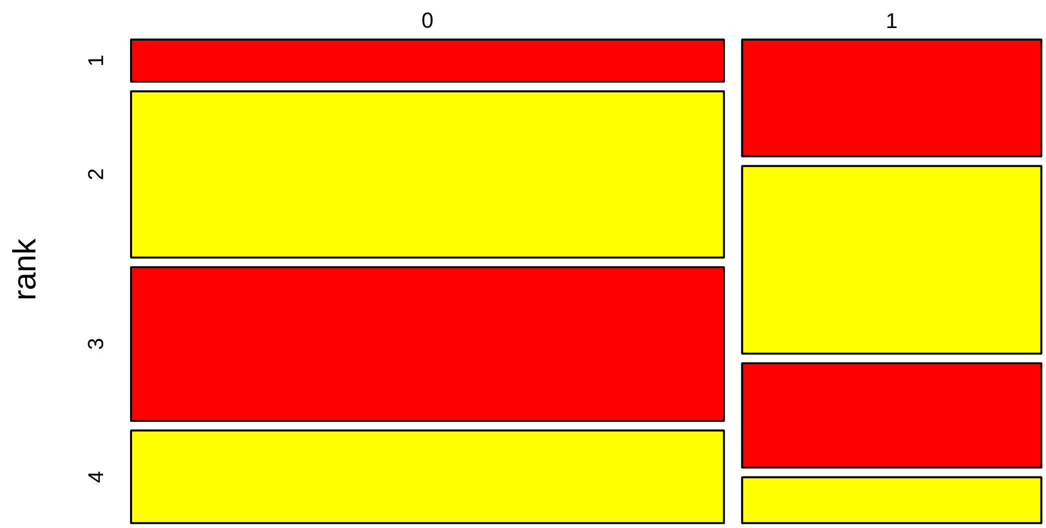

0.0022
0.0020
0.0018
0.0016
0.0014
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
Density
0 (bw: 36.638)
1 (bw: 36.382)
100
200
300
400
500
600
700
800
900
gre


0 (bw: 0.119)
1 (bw: 0.125)
Density
gpa
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0.45
0.50
0.55
0.60
0.65
2.0
2.2
2.4
2.6
2.8
3.0
3.2
3.4
3.6
3.8
4.0
4.2
4.4


0
1
ses
1
2
3


0
1
Gender_Male
0
1


0
1
Race
1
2
3




```r
# Make the prediction using the Naive Beyes model and combine the result with the
## original data set
pred_n = predict(nb1, train_n, type = 'prob')
```

```
## Warning: predict.naive_bayes(): more features in the newdata are provided as
## there are probability tables in the object. Calculation is performed based on
## features to be found in the tables.
```

```r
pred_n = round(pred_n, digits = 3)
head(cbind(pred_n, train_n), 30)
```

```
##        0     1 admit gre  gpa ses Gender_Male Race rank
## 1  0.885 0.115     0 380 3.61   1           0    3    3
## 2  0.631 0.369     1 660 3.67   2           0    2    3
## 3  0.238 0.762     1 800 4.00   2           0    2    1
## 4  0.605 0.395     1 760 3.00   2           1    1    2
## 5  0.673 0.327     1 560 2.98   2           1    2    1
## 6  0.724 0.276     1 540 3.39   1           1    1    3
## 7  0.545 0.455     0 700 3.92   1           0    2    2
## 8  0.682 0.318     0 440 3.22   3           0    2    1
## 9  0.308 0.692     1 760 4.00   3           1    2    1
## 10 0.750 0.250     0 700 3.08   2           0    2    2
## 11 0.209 0.791     1 700 4.00   2           1    1    1
## 12 0.573 0.427     0 780 3.87   2           0    3    4
## 13 0.977 0.023     0 360 2.56   3           1    3    3
```

```
## 14 0.365 0.635     0 800 3.75   1           1    3    2
## 15 0.352 0.648     1 660 3.63   1           0    1    2
## 16 0.893 0.107     0 600 2.82   1           0    3    4
## 17 0.657 0.343     1 760 3.35   2           0    2    2
## 18 0.149 0.851     1 800 3.66   2           1    1    1
## 19 0.209 0.791     1 620 3.61   2           0    1    1
## 20 0.693 0.307     1 520 3.74   2           0    3    4
## 21 0.487 0.513     1 780 3.22   1           0    1    2
## 22 0.408 0.592     0 520 3.29   1           0    1    1
## 23 0.692 0.308     0 600 3.40   3           0    1    3
## 24 0.411 0.589     1 800 4.00   3           0    1    3
## 25 0.917 0.083     0 360 3.14   1           1    2    1
## 26 0.923 0.077     0 400 3.05   3           0    2    2
## 27 0.577 0.423     0 580 3.25   1           0    2    1
## 28 0.895 0.105     0 520 2.90   2           0    2    3
## 29 0.787 0.213     1 500 3.13   2           0    2    2
## 30 0.807 0.193     1 520 2.68   2           0    1    3
```

```r
# Build a confusion matrix using the test set to evaluate the model. As shown, the
## prediction accuracy is 73.33%.
confusionMatrix(predict(nb1, test_n), test_n$admit)
```

```
## Warning: predict.naive_bayes(): more features in the newdata are provided as
## there are probability tables in the object. Calculation is performed based on
## features to be found in the tables.
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 51 14
##          1  6  4
##
##                Accuracy : 0.7333
##                  95% CI : (0.6186, 0.8289)
##     No Information Rate : 0.76
##     P-Value [Acc > NIR] : 0.7544
##
##                   Kappa : 0.1379
##
##  Mcnemar's Test P-Value : 0.1175
##
##             Sensitivity : 0.8947
##             Specificity : 0.2222
##          Pos Pred Value : 0.7846
##          Neg Pred Value : 0.4000
##              Prevalence : 0.7600
##          Detection Rate : 0.6800
##    Detection Prevalence : 0.8667
##       Balanced Accuracy : 0.5585
##
##        'Positive' Class : 0
##
```