P8106_HW4

Naomi Simon-Kumar ns3782

04/11/2025

Contents

Loading libraries 1

Part 1: Tree Based Models using College Data **2**

Partition into training and testing set 2

a) Build a regression tree on training data 2

b) Perform random forest on training data 7

c) Perform boosting on training data 13

Part 2: Classification using Auto dataset **17**

Partition into training and testing set 17

a) Classification Tree 17

b) Boosting model: AdaBoost 21

References **25**

Loading libraries

```
# Load libraries
library(tidyverse)
library(tidymodels)
library(caret)
library(ggplot2)
library(rpart)
library(rpart.plot)
library(ranger)
library(gbm)
library(rsample)
library(pROC)
```

Part 1: Tree Based Models using College Data

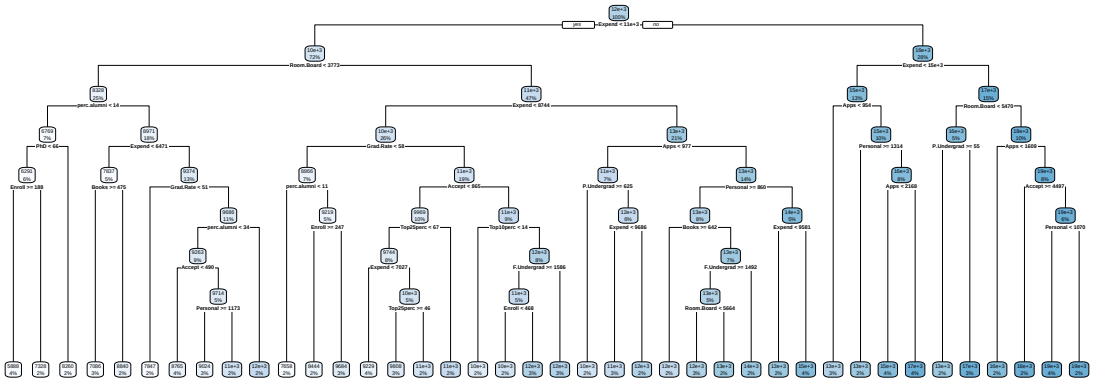
Partition into training and testing set

```
# Read in dataset
college <- read.csv("College.csv")

# Remove NAs
college <- na.omit(college)

# Set seed for reproducibility
set.seed(299)

# Split data into training and testing data
data_split_college <- initial_split(college, prop = 0.8)

# Extract the training and test data, removing college ID column
training_data_college <- training(data_split_college) %>% select(-College)
testing_data_college <- testing(data_split_college) %>% select(-College)
```

a) Build a regression tree on training data

In order to implement the CART approach implementing recursive partitioning and pruning, I first fit a regression tree using cp=0 (complexity parameter). This parameter controls the complexity pruning in the CART algorithm, i.e., how splits are undertaken. Setting cp=0 was a safe choice to ensure that the tree was sufficiently large, allowing all potential splits are considered. I also produced a plot of this tree.

```
# Set seed for reproducibility
set.seed(299)

# Fit initial tree:
initial.tree.fit <- rpart(Outstate ~ .,
                          data = training_data_college,
                          control = rpart.control(cp = 0))

# Tree plot
rpart.plot(initial.tree.fit)
```

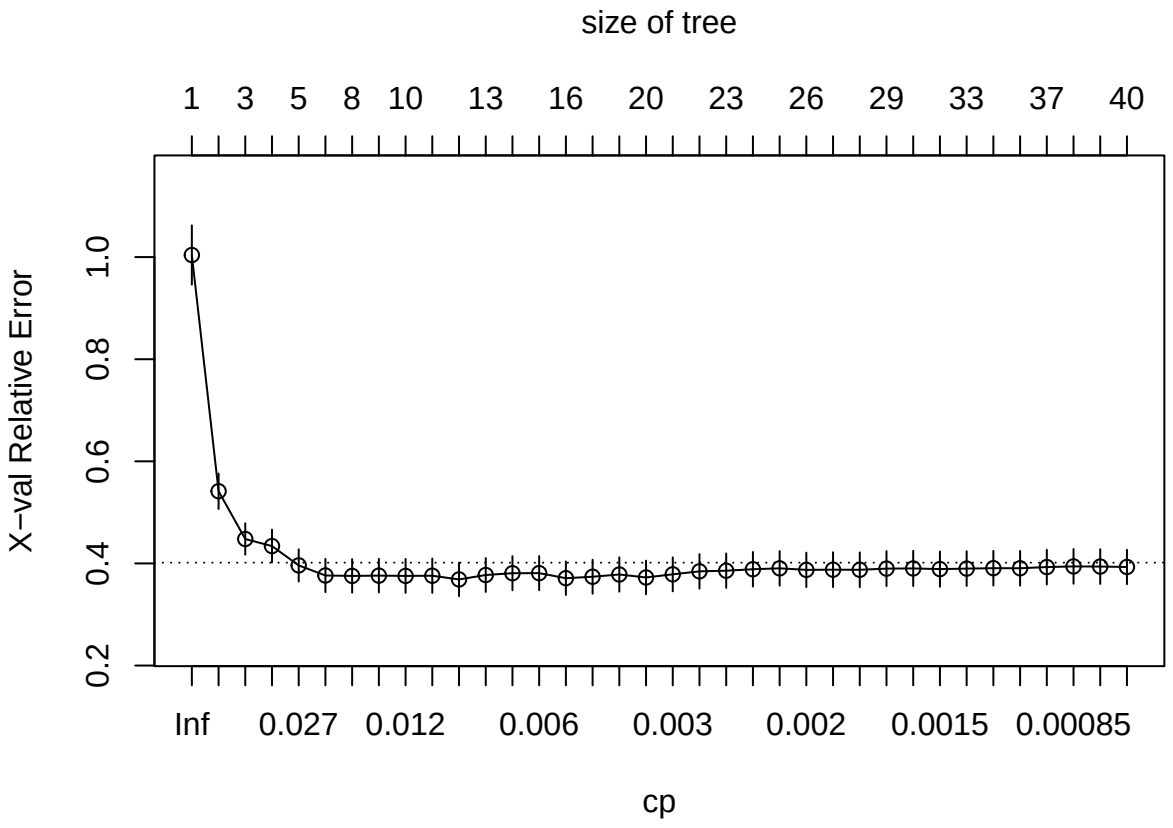


```
# Print and plot the cp table
printcp(initial.tree.fit)
```

```
##
## Regression tree:
## rpart(formula = Outstate ~ ., data = training_data_college, control = rpart.control(cp = 0))
##
## Variables actually used in tree construction:
##  [1] Accept      Apps        Books       Enroll      Expend      F.Undergrad
##  [7] Grad.Rate   P.Undergrad perc.alumni Personal    PhD         Room.Board
## [13] Top10perc   Top25perc
##
## Root node error: 6222135701/452 = 13765787
##
## n= 452
##
##            CP nsplit rel error  xerror     xstd
## 1  0.49965968      0   1.00000 1.00404 0.058031
## 2  0.10266905      1   0.50034 0.54145 0.034753
## 3  0.05071124      2   0.39767 0.44785 0.030837
## 4  0.04021488      3   0.34696 0.43401 0.032119
## 5  0.01820952      4   0.30675 0.39616 0.031612
## 6  0.01319074      5   0.28854 0.37639 0.032586
## 7  0.01255106      7   0.26215 0.37556 0.032802
## 8  0.01194445      8   0.24960 0.37611 0.032884
```

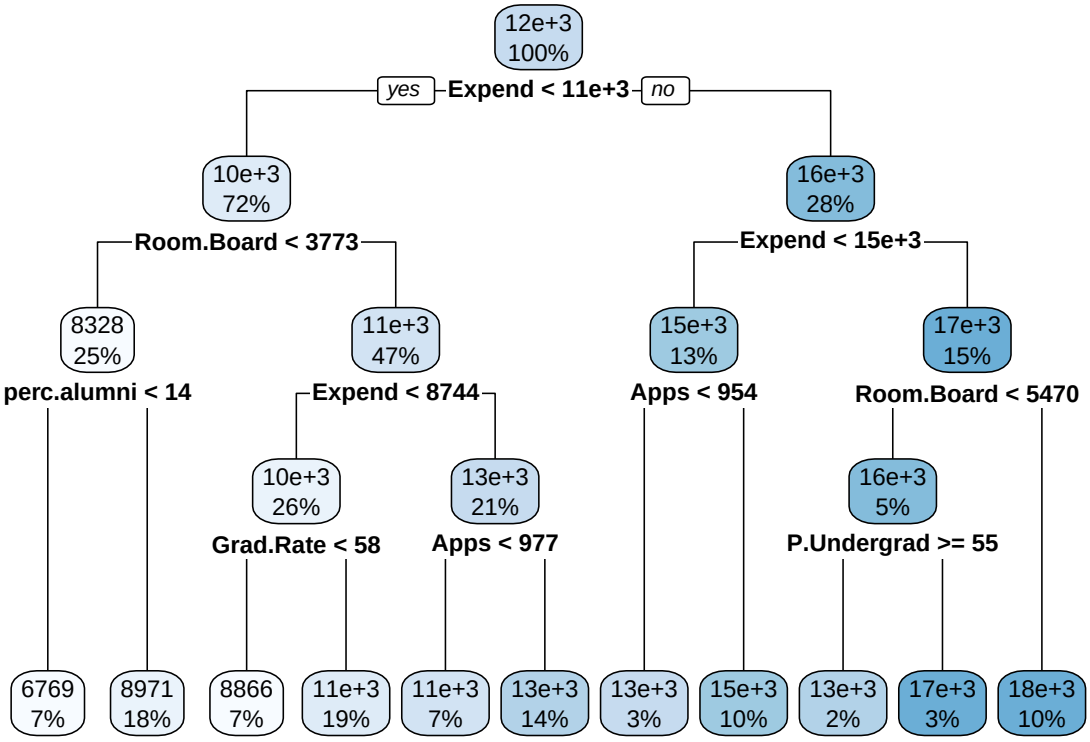
```
## 9  0.01165684      9   0.23766 0.37553 0.033331
## 10 0.00942611     10   0.22600 0.37590 0.033641
## 11 0.00770392     11   0.21658 0.36867 0.032816
## 12 0.00672841     12   0.20887 0.37730 0.033123
## 13 0.00619338     13   0.20214 0.38085 0.033378
## 14 0.00587852     14   0.19595 0.38098 0.033357
## 15 0.00538113     15   0.19007 0.37111 0.032927
## 16 0.00531018     17   0.17931 0.37395 0.033274
## 17 0.00377286     18   0.17400 0.37834 0.033610
## 18 0.00353985     19   0.17023 0.37250 0.032941
## 19 0.00254368     20   0.16669 0.37870 0.033291
## 20 0.00246805     21   0.16414 0.38447 0.033757
## 21 0.00212300     22   0.16167 0.38576 0.033736
## 22 0.00210237     23   0.15955 0.38858 0.033838
## 23 0.00207706     24   0.15745 0.39037 0.033877
## 24 0.00199280     25   0.15537 0.38740 0.033677
## 25 0.00188271     26   0.15338 0.38777 0.034094
## 26 0.00187160     27   0.15150 0.38754 0.034083
## 27 0.00179477     28   0.14963 0.38986 0.034280
## 28 0.00167899     30   0.14604 0.39016 0.034537
## 29 0.00139108     31   0.14436 0.38893 0.034479
## 30 0.00120513     32   0.14297 0.38988 0.033974
## 31 0.00119267     33   0.14176 0.39068 0.033937
## 32 0.00110541     34   0.14057 0.39051 0.033940
## 33 0.00099926     36   0.13836 0.39288 0.033968
## 34 0.00073074     37   0.13736 0.39417 0.033983
## 35 0.00031851     38   0.13663 0.39388 0.033991
## 36 0.00000000     39   0.13631 0.39305 0.033715
```

```
plotcp(initial.tree.fit)
```



To explore the impact of adjusting the complexity parameter, I also fit a second model setting cp=0.01. This model had fewer splits by comparison. When plotted, this tree was noticeably smaller than the previous tree.

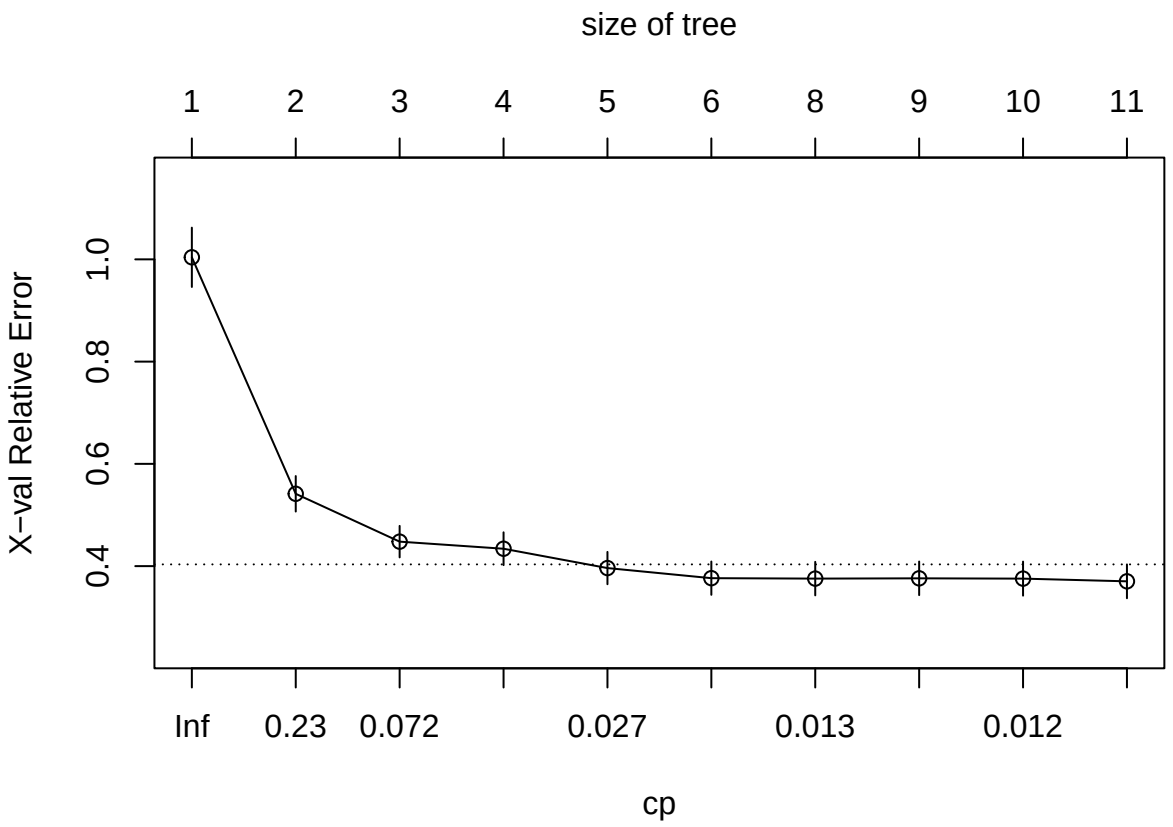
```
# Set seed for reproducibility
set.seed(299)

# Fit another tree:
tree.fit.2 <- rpart(Outstate ~ .,
                    data = training_data_college,
                    control = rpart.control(cp = 0.01))

# Tree plot
rpart.plot(tree.fit.2)
```




```
# Print and plot the cp table
printcp(tree.fit.2)
```

```
##
## Regression tree:
## rpart(formula = Outstate ~ ., data = training_data_college, control = rpart.control(cp = 0.01))
##
## Variables actually used in tree construction:
## [1] Apps        Expend      Grad.Rate   P.Undergrad perc.alumni Room.Board
##
## Root node error: 6222135701/452 = 13765787
##
## n= 452
##
##          CP nsplit rel error  xerror     xstd
## 1  0.499660      0   1.00000 1.00404 0.058031
## 2  0.102669      1   0.50034 0.54145 0.034753
## 3  0.050711      2   0.39767 0.44785 0.030837
## 4  0.040215      3   0.34696 0.43401 0.032119
## 5  0.018210      4   0.30675 0.39616 0.031612
## 6  0.013191      5   0.28854 0.37639 0.032586
## 7  0.012551      7   0.26215 0.37556 0.032802
## 8  0.011944      8   0.24960 0.37611 0.032884
## 9  0.011657      9   0.23766 0.37553 0.033331
## 10 0.010000     10   0.22600 0.37024 0.033137
```

```r
plotcp(tree.fit.2)
```



Importantly, setting cp=0 is the preferred choice, as it allowed for a large enough tree to be grown for the cost complexity table. In the cp = 0.01 model, the smallest xerror (scaled cross-validation error) was 0.37369, whereas the cp = 0 model achieved a slightly lower minimum xerror of 0.36867. This shows us that a fully grown tree in this case is better suited for selecting an optimal complexity parameter based on cross-validation.

The optimal tree selected from the cp = 0 model has a **complexity parameter of 0.00770392**, with **11 splits** (i.e., 12 terminal nodes). This was the model that minimised scaled cross-validation error (**xerror = 0.36867**). Therefore, this was chosen as the final pruned tree.

b) Perform random forest on training data

I decided to explore two tuning grids to find the optimal random forest model using cross-validation RMSE. The mtry parameter controls the number of predictors randomly selected at each split in the forest. I decided to tune mtry over the full range of possible values, from 1 to 16 (i.e., the full number of predictors in the college dataset).

For the first grid (min.node.size = 1:7), the best model had mtry = 7 and min.node.size = 5.

```r
# Set seed for reproducibility
set.seed(299)

# Set cross-validation
ctrl <- trainControl(method = "cv",
```

```
                     summaryFunction = defaultSummary)

# Define grid for tuning mtry and min.node.size
rf.grid <- expand.grid(
  mtry = 1:16, # max no. of predictors
  splitrule = "variance",
  min.node.size = c(1:7)
)

# Fit random forest using ranger via caret
rf.fit <- train(
  Outstate ~ .,
  data = training_data_college,
  method = "ranger",
  tuneGrid = rf.grid,
  trControl = ctrl
)
```

```
## Growing trees.. Progress: 21%. Estimated remaining time: 5 minutes, 54 seconds.
```

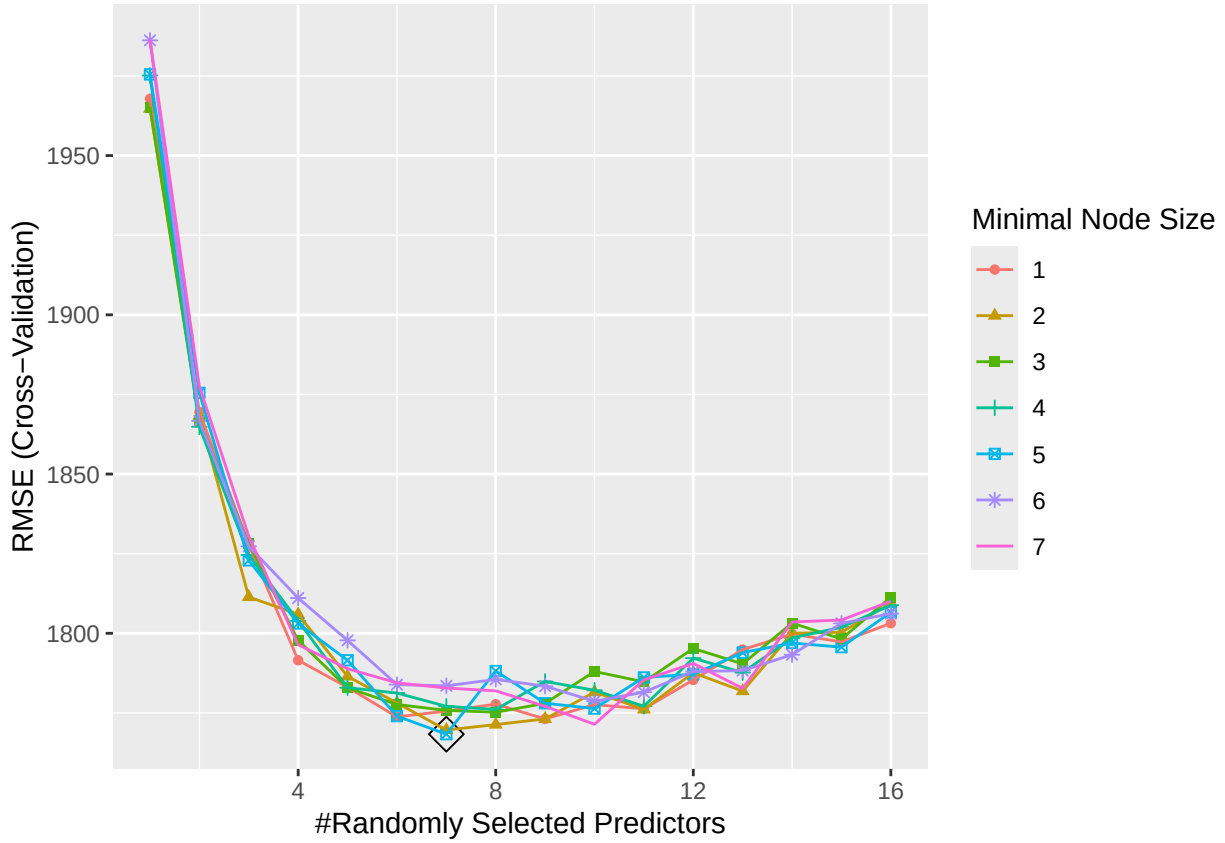
```
# Obtain optimal tuning parameters from cross-validation
rf.fit$bestTune # mtry = 7, min.node.size = 5
```

```
##    mtry splitrule min.node.size
## 47    7  variance             5
```

```
ggplot(rf.fit, highlight = TRUE)
```

```
## Warning: The shape palette can deal with a maximum of 6 discrete values because more
## than 6 becomes difficult to discriminate
## i you have requested 7 values. Consider specifying shapes manually if you need
##   that many have them.
```

```
## Warning: Removed 16 rows containing missing values or values outside the scale range
## (`geom_point()`).
```



I then decided to try an expanded grid range for min.node.size, extending it to 1:10 to check for any better performing values beyond the original range.

```
# Set seed for reproducibility
set.seed(299)

# Define another grid for tuning mtry and min.node.size
rf.grid2 <- expand.grid(
  mtry = 1:16, # max no. of predictors
  splitrule = "variance",
  min.node.size = c(1:10)
)

# Fit random forest using ranger via caret
rf.fit2 <- train(
  Outstate ~ .,
  data = training_data_college,
  method = "ranger",
  tuneGrid = rf.grid2,
  trControl = ctrl
)
```

```
## Growing trees.. Progress: 12%. Estimated remaining time: 1 hour, 47 minutes, 45 seconds.
```

```
# Optimal parameters
rf.fit2$bestTune  # mtry = 9, min.node.size = 3 (row 83)
```

```
##    mtry splitrule min.node.size
## 83    9  variance             3
```

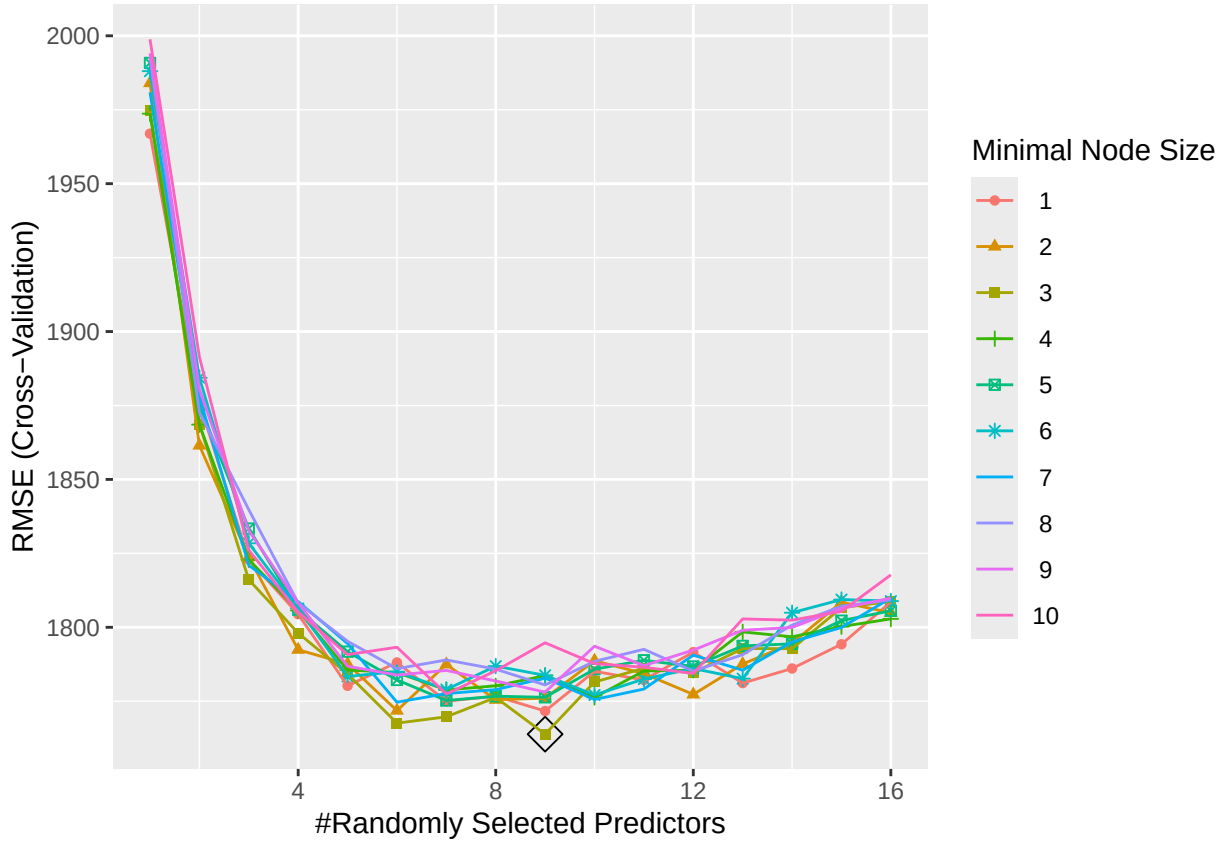
```
rf.fit2$results[83, ] # pulling the lowest RMSE: 1763.865
```

```
##    mtry splitrule min.node.size     RMSE Rsquared     MAE   RMSESD RsquaredSD
## 83    9  variance             3 1763.865 0.774242 1332.22 307.1101 0.08056556
##       MAESD
## 83 176.1794
```

```
# Plot performance for tuning grid values
ggplot(rf.fit2, highlight = TRUE)
```

```
## Warning: The shape palette can deal with a maximum of 6 discrete values because more
## than 6 becomes difficult to discriminate
## i you have requested 10 values. Consider specifying shapes manually if you need
##   that many have them.
```

```
## Warning: Removed 64 rows containing missing values or values outside the scale range
## (`geom_point()`).
```

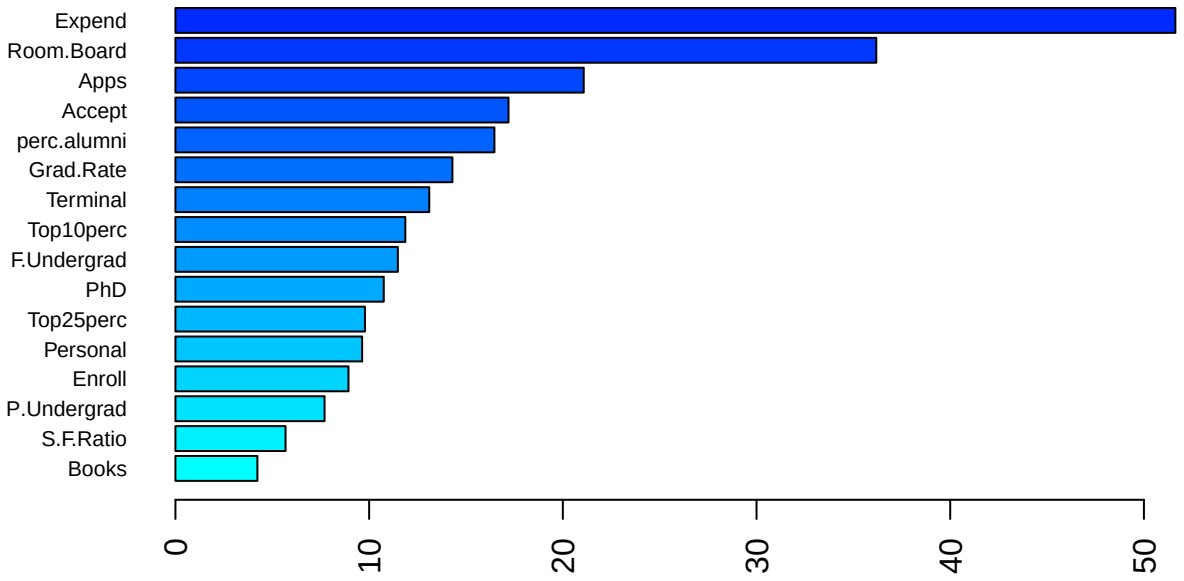


From this grid, **the optimal tuning parameters were mtry = 9 and min.node.size = 3**, selected based on the lowest cross validation RMSE (1763.865). The corresponding plot shows the lowest RMSE.

Next, reporting the variable importance and the test error for this selected model (rf.fit2):

I first examined **permutation-based variable importance** in the dataset.

```
# Set seed for reproducibility
set.seed(299)

# Fit random forest using ranger via caret
rf.fit2.per <- ranger(
  Outstate ~ .,
  data = training_data_college,
  mtry = rf.fit2$bestTune$mtry, # selecting the cross-validated parameter
  min.node.size = rf.fit2$bestTune$min.node.size, # selecting the cross-validated parameter
  splitrule = "variance",
  importance = "permutation", # selecting permutation
  scale.permutation.importance = TRUE
)

# Plotting the variable importance
barplot(sort(ranger::importance(rf.fit2.per), decreasing = FALSE),
las = 2, horiz = TRUE, cex.names = 0.7,
col = colorRampPalette(colors = c("cyan","blue"))(19))
```

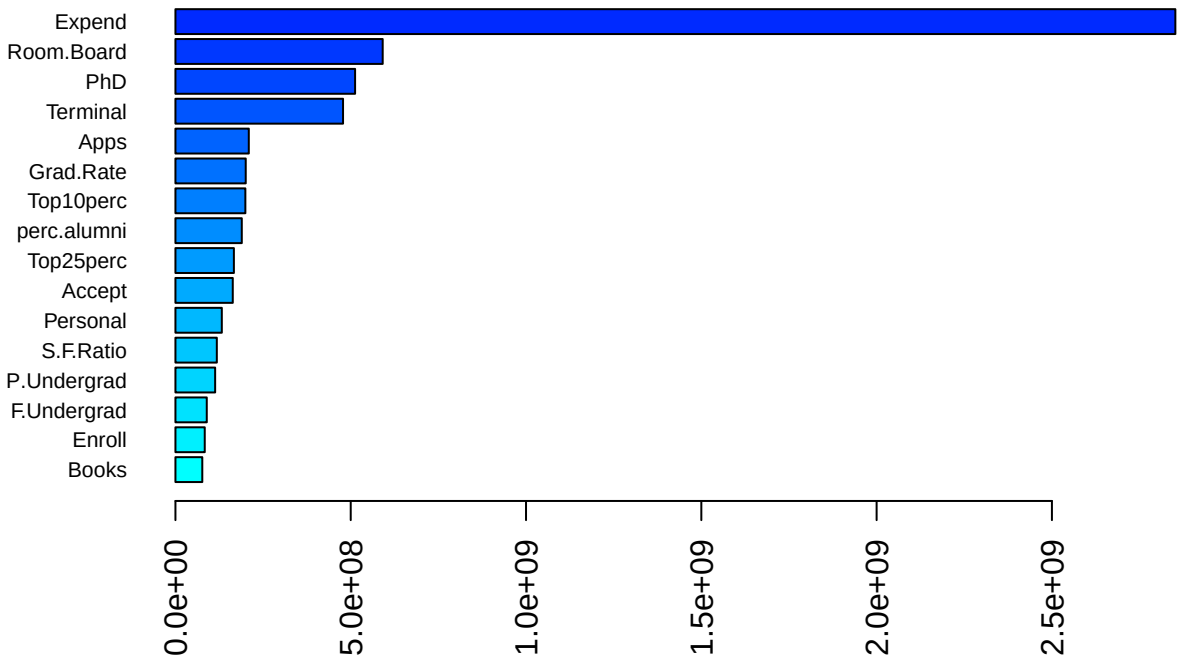


Based on this plot, **Expend** appears to have the highest variable importance, followed by **Room Board**

and **Apps**, respectively. Books appears to be the least important variable, by comparison, followed by S.F. Ratio and P.Undergrad, in that order.

I decided to explore variable importance using the impurity method for the same model.

```
# Set seed for reproducibility
set.seed(299)

# Fit random forest using ranger via caret
rf.fit2.imp <- ranger(
  Outstate ~ .,
  data = training_data_college,
  mtry = rf.fit2$bestTune$mtry, # selecting the cross-validated parameter
  min.node.size = rf.fit2$bestTune$min.node.size, # selecting the cross-validated parameter
  splitrule = "variance",
  importance = "impurity", # selecting impurity
  scale.permutation.importance = TRUE
)

# Plotting the variable importance
barplot(sort(ranger::importance(rf.fit2.imp), decreasing = FALSE),
las = 2, horiz = TRUE, cex.names = 0.7,
col = colorRampPalette(colors = c("cyan","blue"))(19))
```



As with the permutation method, the most important variable based on this plot appears to be **Expend**, followed by Room.Board. However, the impurity method shows that PhD is the third most important

variable, which is different to what was identified using the permutation method (the permutation method identified the third most important variable as Apps).

Books appears to be the least important variable using the impurity method, similar to the permutation method. However, unlike the permutation method, the subsequent least important variables are Enroll and F.Undergrad, in that order. This difference is consistent with the literature, with permutation importance avoiding biases that affect impurity measures, though impurity based variable importance rankings can be more computationally efficient and sometimes more robust to data perturbations in high-dimensional settings (Nembrini, et al., 2018).

Next, finding the **test error of the model**:

```
# Set seed for reproducibility
set.seed(299)

# Predict on the test data
pred.rf2 <- predict(rf.fit2, newdata = testing_data_college)

# RMSE based on the test data
rf_test_rmse <- RMSE(pred.rf2, testing_data_college$Outstate)

# Test error
rf_test_rmse
```

```
## [1] 1629.167
```

Therefore, the **test error (RMSE) is 1629.167**.

c) Perform boosting on training data

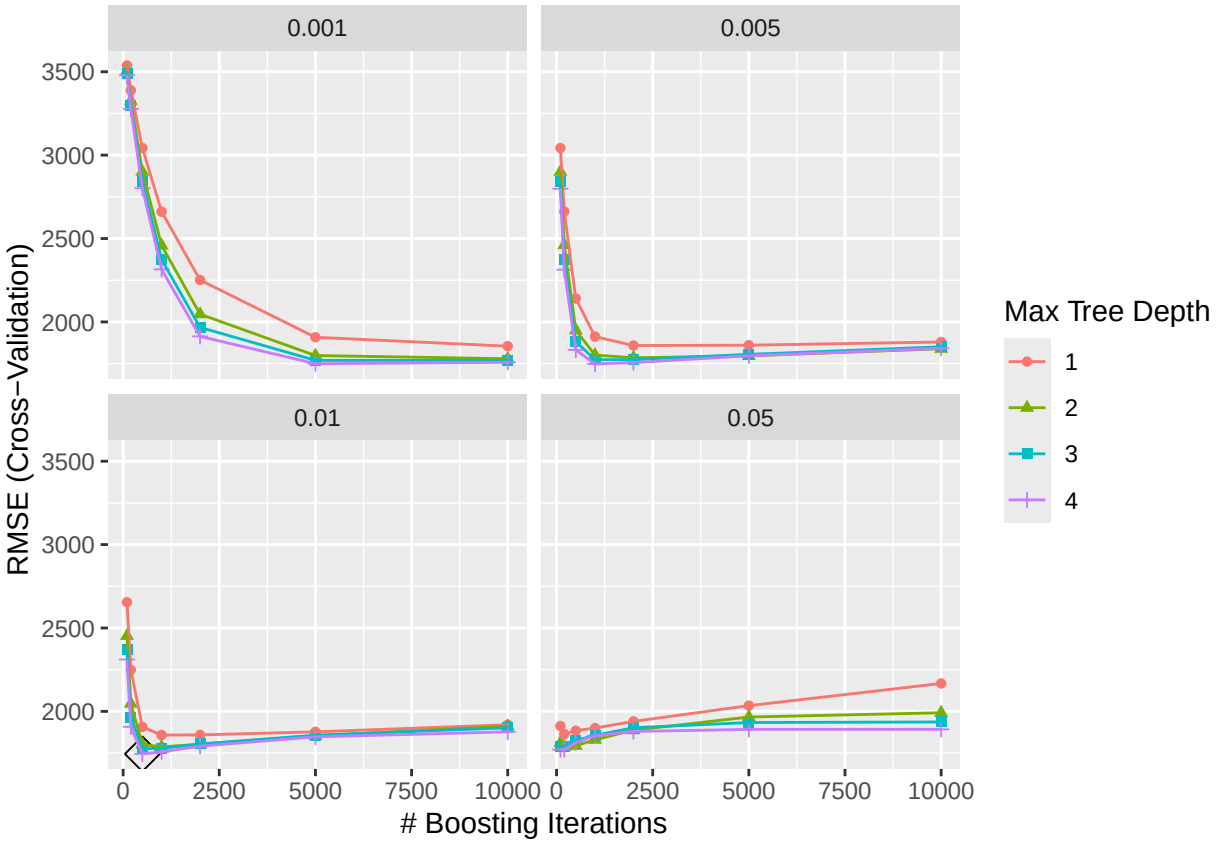
I proceeded to fit a gradient boosting model with the Gaussian loss function.

```
# Set seed for reproducibility
set.seed(299)

# Initial grid for tuning
# gbm.grid <- expand.grid(
 # n.trees = c(100, 200, 500, 1000, 2000, 5000, 10000),
#  interaction.depth = 1:3, # we want to learn slowly, so keep small
#  shrinkage = c(0.005, 0.01, 0.05), # range of lambda values
# n.minobsinnode = 10  # Based on notes, this can be fixed
#)

# Fit the GBM model
# gbm.fit <- train(
 # Outstate ~ .,
#  data = training_data_college,
#  method = "gbm",
#  tuneGrid = gbm.grid,
 # trControl = ctrl,
#  verbose = FALSE
#)
```

```r
# View best tuning parameters
# gbm.fit$bestTune
# n trees = 2000; interaction.depth=3; shrinkage=0.005; n.minobsinnode=10


# Define grid for tuning
gbm.grid <- expand.grid(
  n.trees = c(100, 200, 500, 1000, 2000, 5000, 10000),
  interaction.depth = 1:4, # increase to 4, still keeping small
  shrinkage = c(0.001, 0.005, 0.01, 0.05), # adding an additional shrinkage
  n.minobsinnode = 10
)

# Fit the GBM model
gbm.fit <- train(
  Outstate ~ .,
  data = training_data_college,
  method = "gbm",
  tuneGrid = gbm.grid,
  trControl = ctrl,
  verbose = FALSE
)

# View best tuning parameters
gbm.fit$bestTune # n.trees = 500; shrinkage=0.01, interaction.depth=4
```

```
##    n.trees interaction.depth shrinkage n.minobsinnode
## 80     500                 4      0.01             10
```

```r
# Plot the CV results
ggplot(gbm.fit, highlight = TRUE)
```

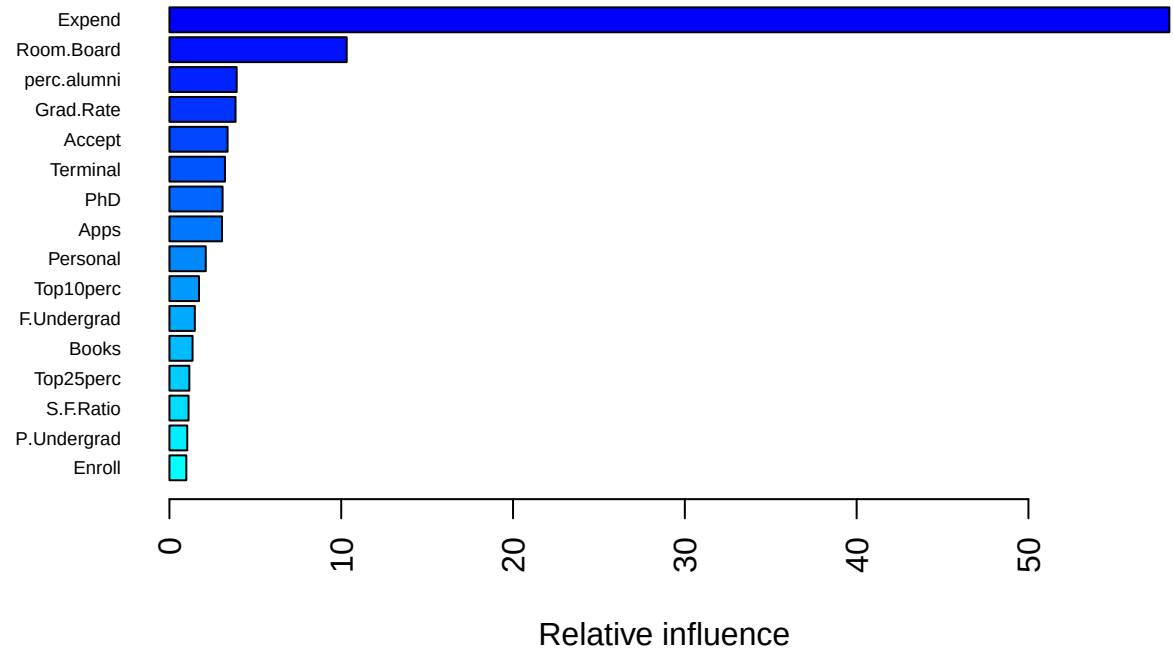


I initially tested a smaller grid when tuning the model parameters with interaction.depth = 1:3. The best model had interaction.depth = 3, which was the upper limit of that range. Even though it is acceptable for interaction.depth to be at the boundary, I decided to expand the grid to 1:4 to explore other ranges and to also explore another shrinkage parameter.

Based on the cross-validation RMSE plot showing the combinations of shrinkage (i.e., the learning rate), n.trees (number of boosting iterations), and interaction depth (tree depth), we can see that the lowest RMSE was achieved with **shrinkage = 0.01, n.minobsinnode = 10, interaction.depth = 4**, and approximately **500 boosting iterations (n.trees)**. This is consistent with the Generalized Boosting Models package documentation, which notes that the relationship between shrinkage and optimal iterations is roughly proportional, as our optimal n.trees decreased when we increased the shrinkage value (Ridgeway, 2024). Notably, while n.minobsinnode was at the boundary of the range, it was noted by the Professor that this is acceptable.

Next, finding the **variable importance**.

```
# Set seed for reproducibility
set.seed(299)

# Using summary to find variable importance for boosting
summary(gbm.fit$finalModel, las = 2, cBars = 19, cex.names = 0.6)
```




```
##                     var    rel.inf
## Expend           Expend 58.2031691
## Room.Board   Room.Board 10.3174922
## perc.alumni perc.alumni  3.9127298
## Grad.Rate     Grad.Rate  3.8474112
## Accept           Accept  3.3836026
## Terminal       Terminal  3.2318329
## PhD                 PhD  3.0933115
## Apps               Apps  3.0597074
## Personal       Personal  2.1141994
## Top10perc     Top10perc  1.7216327
## F.Undergrad F.Undergrad  1.4781997
## Books             Books  1.3455640
## Top25perc     Top25perc  1.1584877
## S.F.Ratio     S.F.Ratio  1.1122809
## P.Undergrad P.Undergrad  1.0374312
## Enroll           Enroll  0.9829476
```

From this, we can see the most important predictor appears to be **Expend**, followed by Room.Board and perc.alumni, respectively. The least important variable based on this summary of the boosting model appears to be Enroll, followed by P.Undergrad and S.F. Ratio, in that order.

Next, finding the **test error** for the boosting model:

```r
# Set seed for reproducibility
set.seed(299)

# Predict on the test data
pred.gbm.fit <- predict(gbm.fit, newdata = testing_data_college)

# RMSE based on the test data
gbm_test_rmse <- RMSE(pred.gbm.fit, testing_data_college$Outstate)

# Test error
gbm_test_rmse
```

```
## [1] 1568.585
```

The test error for the model is **1568.585**.

Part 2: Classification using Auto dataset

Partition into training and testing set

```r
# Read in dataset
auto <- read.csv("auto.csv")

# Remove NAs
auto <- na.omit(auto)

# Recode mpg_cat as binary variable (0 = "low", 1 = "high")
auto <- auto %>%
  mutate(mpg_cat = ifelse(mpg_cat == "high", 1, 0))

# Set seed for reproducibility
set.seed(299)

# Split data into training and testing data
data_split_auto <- initial_split(auto, prop = 0.7)


# Extract the training and test data
training_data_auto <- training(data_split_auto)
testing_data_auto <- testing(data_split_auto)
```

a) Classification Tree

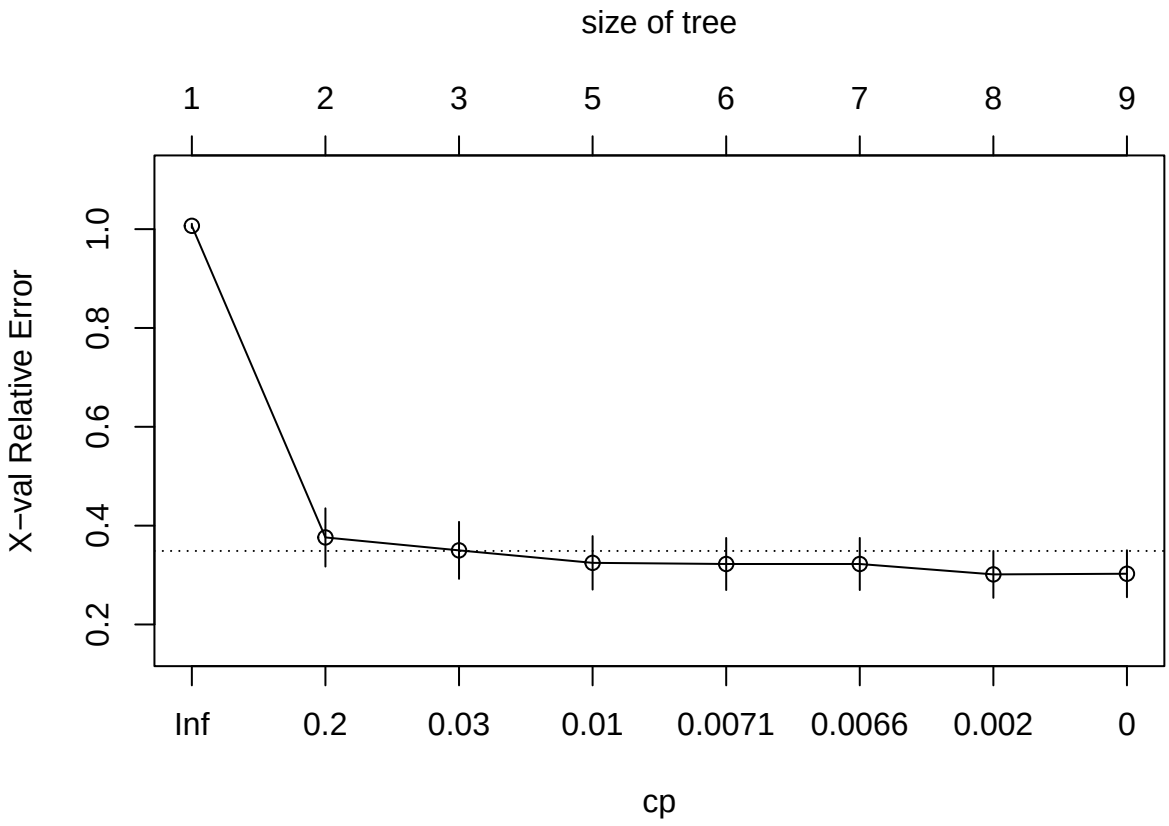
Using the rpart method, I proceeded to fit the tree:

```r
# Set seed for reproducibility
set.seed(299)

# Build tree
```

```r
rpart.class.tree.fit <- rpart(formula = mpg_cat ~ . ,
               data = training_data_auto,
               control = rpart.control(cp = 0))

# Produce cp table and plot tree
cpTable <- printcp(rpart.class.tree.fit)
```

```
##
## Regression tree:
## rpart(formula = mpg_cat ~ ., data = training_data_auto, control = rpart.control(cp = 0))
##
## Variables actually used in tree construction:
## [1] acceleration displacement horsepower   weight       year
##
## Root node error: 68.442/274 = 0.24979
##
## n= 274
##
##           CP nsplit rel error  xerror      xstd
## 1 0.68007687      0   1.00000 1.00672 0.0041924
## 2 0.06138922      1   0.31992 0.37612 0.0589406
## 3 0.01506186      2   0.25853 0.34996 0.0576476
## 4 0.00723536      4   0.22841 0.32469 0.0541093
## 5 0.00706519      5   0.22117 0.32237 0.0527038
## 6 0.00608791      6   0.21411 0.32237 0.0527038
## 7 0.00066109      7   0.20802 0.30132 0.0474196
## 8 0.00000000      8   0.20736 0.30264 0.0474997
```

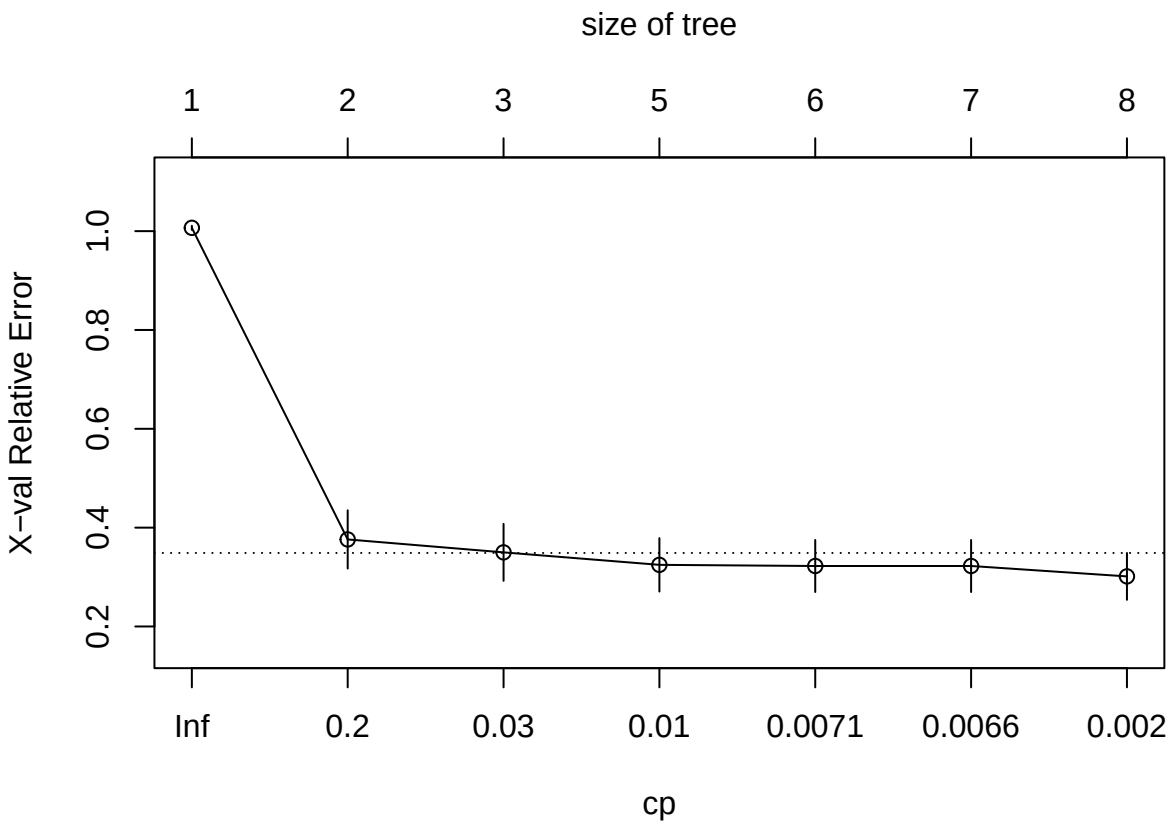
```r
plotcp(rpart.class.tree.fit)
```



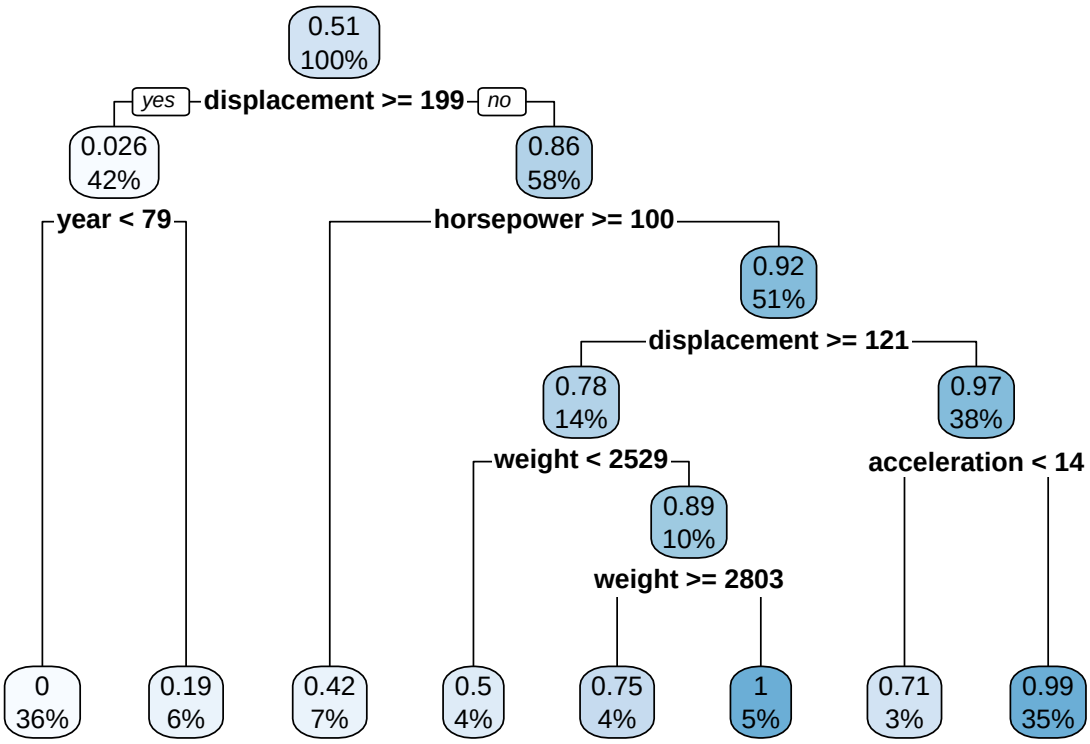
I set cp=0, to allow for a sufficiently large enough tree to be grown. The optimal tree selected from this cp = 0 model has a **complexity parameter of 0.022556**, with **1 split** (i.e, **tree size is 2 terminal nodes** in the decision tree). This was the model that minimised scaled cross-validation error (**xerror = 0.21805**). Therefore, this was chosen as the final pruned tree.

Next, using the 1SE rule to build the tree:

```
# Set seed for reproducibility
set.seed(299)

# Find row with min Cross validation error (xerror)
minErr <- which.min(cpTable[, 4])  # column 4 is xerror

# Extracts the cp parameter corresponding to the 1se rule
tree.fit.1se <- rpart::prune(rpart.class.tree.fit,
                             cp = cpTable[minErr, 1])  # column 1 is cp

# Plot cp for the 1se tree
plotcp(tree.fit.1se)
```




```
# Plot the pruned tree
rpart.plot(tree.fit.1se)
```



In applying the 1se rule, we are looking for the smallest tree below the dotted line. The 1se rule allows us to select the smallest tree whose cross-validation error is within one standard error of the minimum. In this case, the **selected tree has 1 split, with 2 terminal nodes**. Therefore, this is the same tree size when using the lowest cross-validation error.

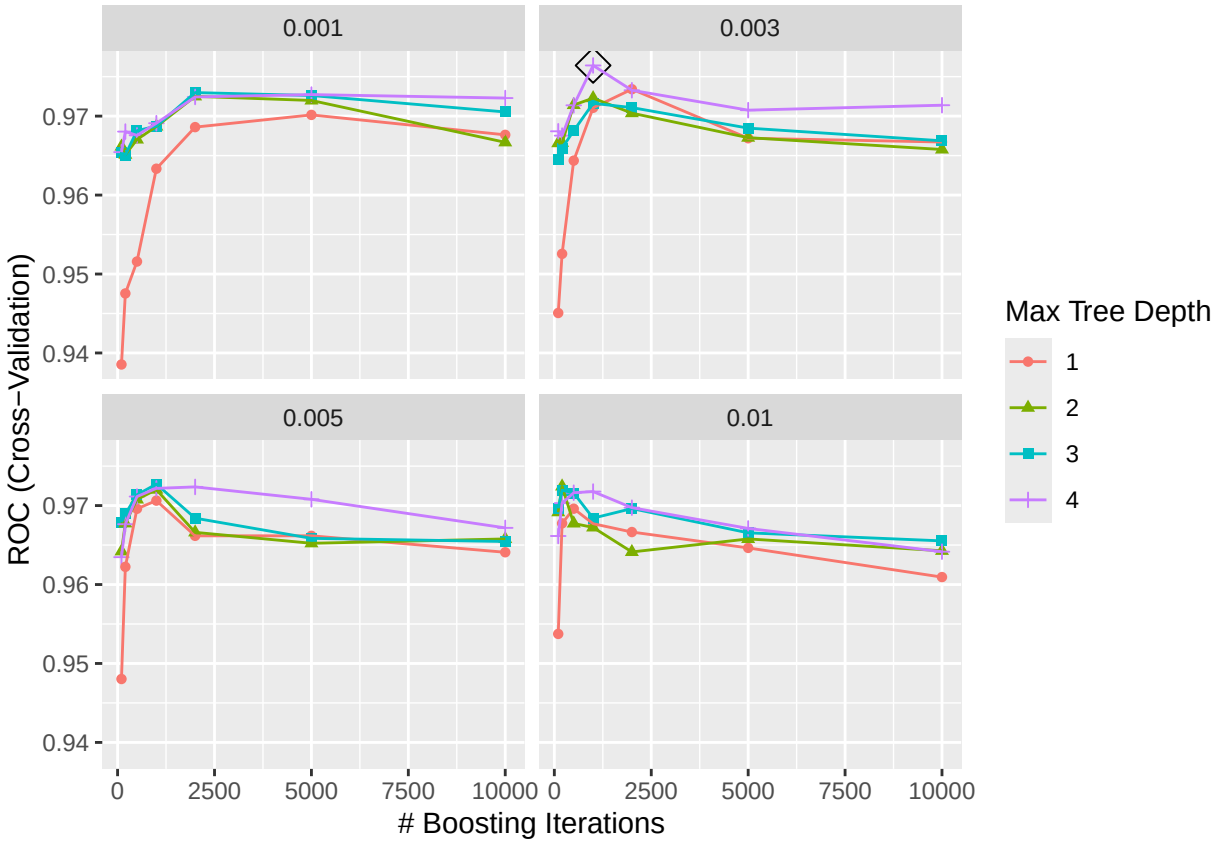
b) Boosting model: AdaBoost

```
# Set seed for reproducibility
set.seed(299)

# Check factor levels
levels(training_data_auto$mpg_cat)
```

```
## NULL
```

```
# Relevel mpg_cat
training_data_auto <- training_data_auto %>%
  mutate(mpg_cat = factor(ifelse(mpg_cat == 1, "high", "low"), levels = c("low", "high")))
testing_data_auto <- testing_data_auto %>%
  mutate(mpg_cat = factor(ifelse(mpg_cat == 1, "high", "low"), levels = c("low", "high")))


# Set cross validation
ctrl.adaboost <- trainControl(method = "cv",
```

```
                              classProbs = TRUE,
                              summaryFunction = twoClassSummary)

# Tuning grid for adaboost
gbm.ada.grid <- expand.grid(
  n.trees = c(100, 200, 500, 1000, 2000, 5000, 10000),
  interaction.depth = 1:4, # want to learn slowly, so keep small
  shrinkage = c(0.001, 0.003, 0.005, 0.01), # range of lambda values
  n.minobsinnode = 10 # based on notes, can fix this at 10
)

# Fit gbm with caret, using adaboost
gbm.ada.fit <- train(mpg_cat ~ .,
  data = training_data_auto,
  tuneGrid = gbm.ada.grid,
  trControl = ctrl.adaboost,
  method = "gbm",
  distribution = "adaboost",
  metric = "ROC",
  verbose = FALSE
)

# Plot to show best tuning parameters
ggplot(gbm.ada.fit, highlight = TRUE)
```



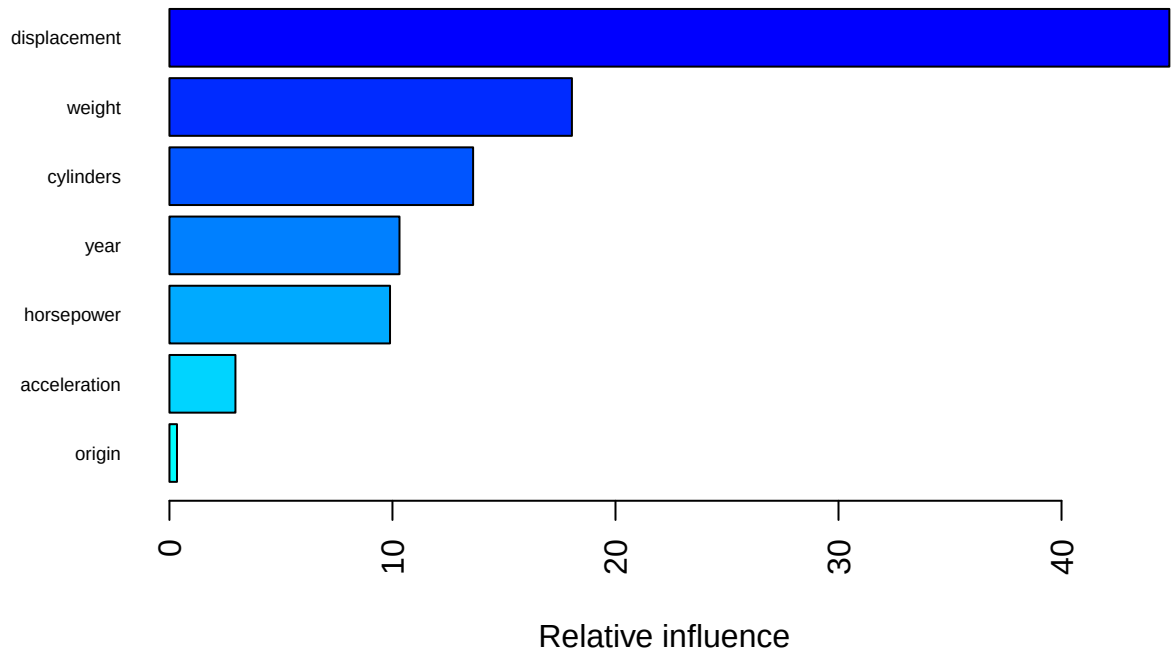
```
# Optimal tuning parameters
gbm.ada.fit$bestTune
```

```
##    n.trees interaction.depth shrinkage n.minobsinnode
## 53    1000                 4     0.003             10
```

I relevelled the outcome variable to make sure high was the second factor level, in order for it to be treated as the positive class when computing AUC using caret.

I then proceeded to peform boosting using adaboost. I initially tried a range of values for interaction depth between 1:5, however, this produced a shrinkage value at the boundary of the grid (0.01). I then tuned the grid keeping interaction.depth at 4, which was appropriate as it gave me n.trees and shrinkage values not at the boundary of their respective grids. Based on this model, the optimal tuning parameters were **n.trees = 1000, interaction.depth = 4, shrinkage = 0.003, and n.minobsinnode = 10**, based on 10 fold cross validation AUC. This is evident based on the plot, where we can see the best performance at shrinkage=0.003 and interaction.depth=4.

```
# Set seed for reproducibility
set.seed(299)

# Presenting variable importance
summary(gbm.ada.fit$finalModel, las = 2, cBars = 19, cex.names = 0.6)
```




```
##                         var    rel.inf
## displacement displacement 44.8349621
```

```
## weight             weight 18.0461779
## cylinders       cylinders 13.6164108
## year                 year 10.3148997
## horsepower     horsepower  9.8902679
## acceleration acceleration  2.9589536
## origin             origin  0.3383279
```

From this, we can see the most important predictor appears to be **displacement**, followed by **weight** and **cylinders**, respectively. The least important variable based on this model appears to be **origin**, followed by acceleration and horsepower, in that order.

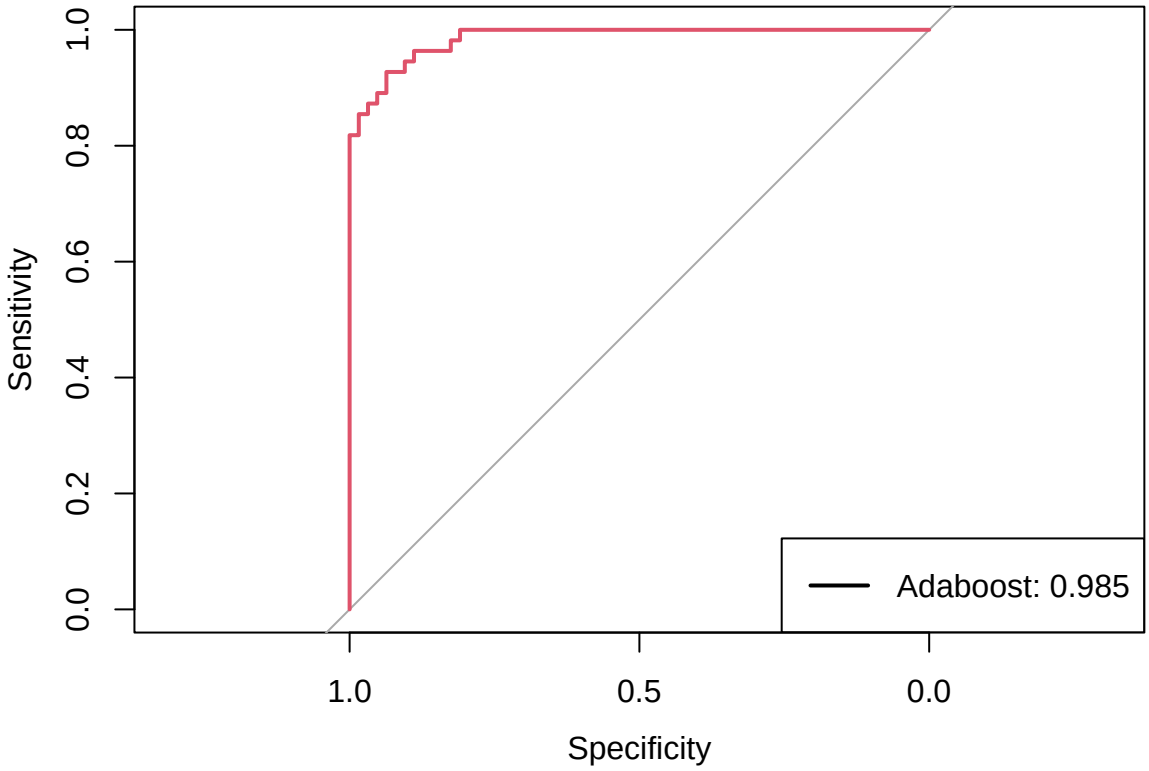
Next, obtaining test error:

```
# Set seed for reproducibility
set.seed(299)

# Predict probabilities for positive class (ie high)
gbmA.prob <- predict(gbm.ada.fit, newdata = testing_data_auto, type = "prob")[, "high"]

# Compute and plot ROC
roc.gbmA <- roc(testing_data_auto$mpg_cat, gbmA.prob)
```

```
## Setting levels: control = low, case = high
```

```
## Setting direction: controls < cases
```

```
plot(roc.gbmA, col = 2)

# Test AUC
auc <- roc.gbmA$auc[1]
modelNames <- "Adaboost"
legend("bottomright", legend = paste0(modelNames, ": ", round(auc,3)),
col = 1:2, lwd = 2)
```



To find the test performance for this model, I computed the AUC (area under the ROC curve). Based on the plot, **AUC was 0.985**, which represents excellent classification performance on the test set.

References

Ridgeway, G. (2024, June 26). Generalized Boosted Models: A guide to the gbm package.

Nembrini, S., König, I. R., & Wright, M. N. (2018). The revival of the Gini importance?. Bioinformatics, 34(21), 3711-3718.