P8106: Data Science II, Homework #4

Zachary Katz (UNI: zak2132)

4/13/2022

Contents

Question 1

Set-Up and Data Preprocessing

```
set.seed(2132)

# Load data, clean column names, eliminate rows containing NA entries
data = read_csv("./Data/College.csv") %>%
  janitor::clean_names() %>%
  na.omit() %>%
  relocate("outstate", .after = "grad_rate") %>%
  select(-college)
```

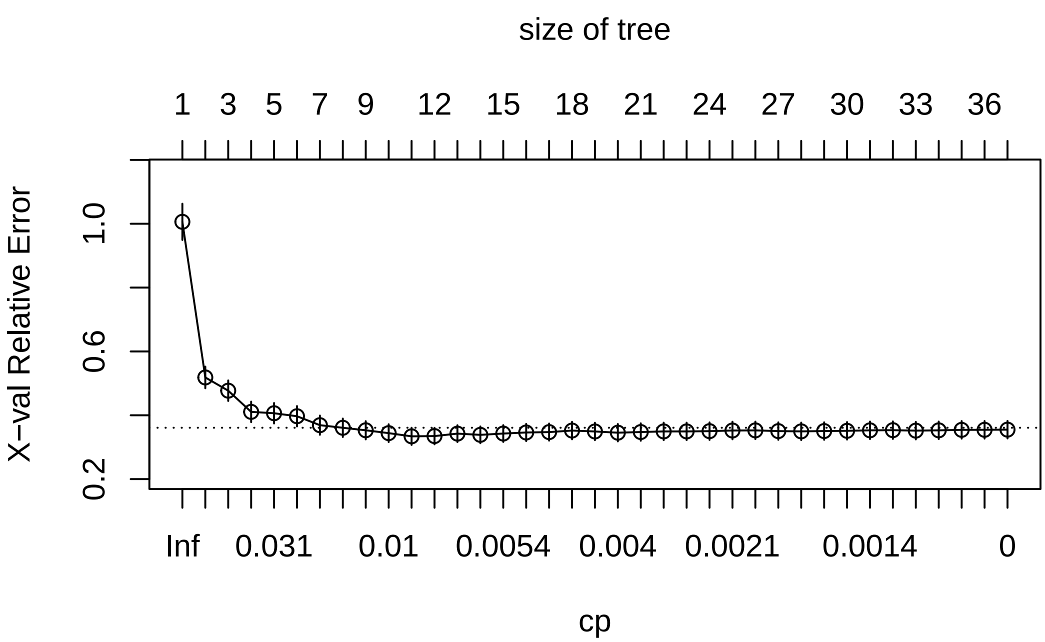
```r
# Partition data into training/test sets
indexTrain = createDataPartition(y = data$outstate,
                                 p = 0.8,
                                 list = FALSE)

training_df = data[indexTrain, ]

testing_df = data[-indexTrain, ]
```

Part (a): Regression Tree

Here, we build two regression trees: one based on the cp value that minimizes MSE, and one based on the 1SE rule. Below, we include one visualization of each tree. The tree based on the minimum MSE rule is much more complex than the one based on the 1SE rule, which only has 7 splits (8 terminal nodes). On average, the predictions between both models when applied to test data are quite close, differing by no more than a few percent.

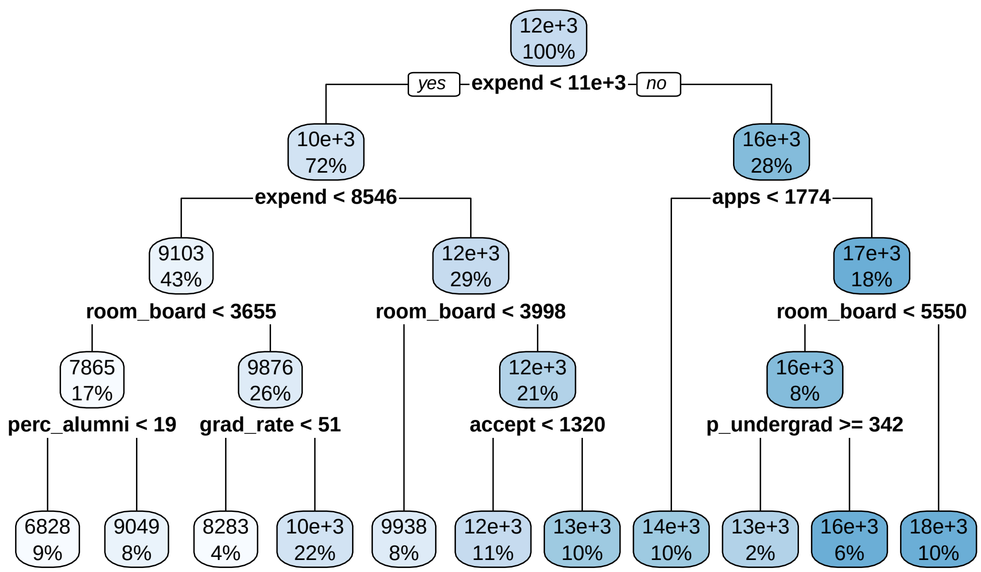
Minimum MSE Rule

```r
# Build a regression tree on the training data to predict the response
set.seed(2132)

regression_tree = rpart(formula = outstate ~ . ,
               data = training_df, control = rpart.control(cp = 0))

# Cross-validation plot
regression_cptable = regression_tree$cptable
plotcp(regression_tree)
```



```r
# Cost-complexity pruning
minimum_MSE = which.min(regression_cptable[,4])
final_regression_tree = prune(regression_tree,
                              cp = regression_cptable[minimum_MSE,1])

# Summary of final tree
# summary(final_regression_tree)
```

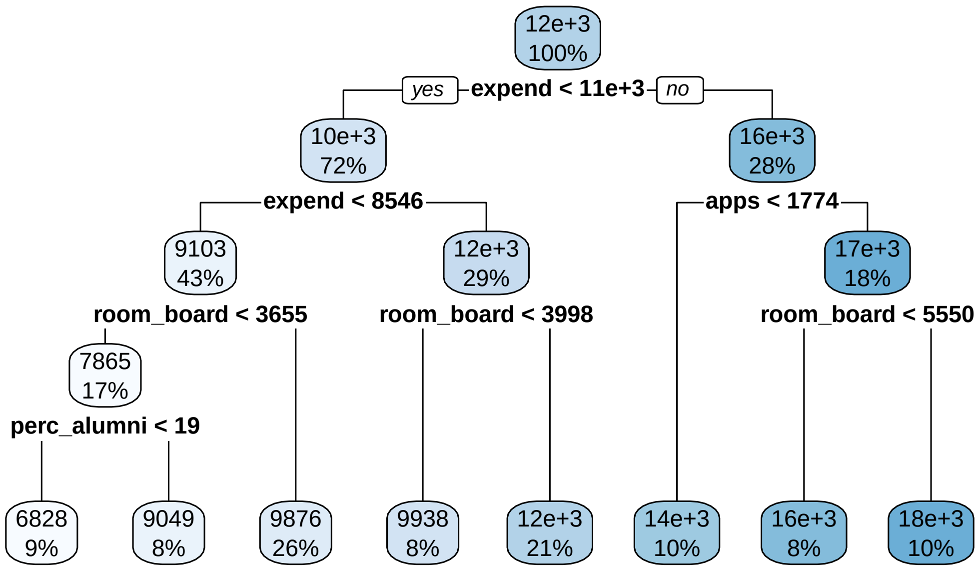
```r
# Plot of final tree
# plot(as.party(final_regression_tree))
rpart.plot(final_regression_tree)
```



1SE Rule

```r
# Alternatively, cost-complexity pruning using 1SE rule
final_regression_tree_1SE = prune(regression_tree,
    cp = regression_cptable[regression_cptable[,4]<regression_cptable[minimum_MSE,4]
                            +regression_cptable[minimum_MSE,5],1][1])
```

```r
# Plot of 1SE tree
# plot(as.party(final_regression_tree_1SE))
rpart.plot(final_regression_tree_1SE)
```



Comparison of Predictions

```
# For fun, compare predictions on first few observations in testing data set
reg_predict = predict(final_regression_tree, newdata = testing_df)
oneSE_predict = predict(final_regression_tree_1SE, newdata = testing_df)

# Compare predictions in data table
cbind(reg_predict, oneSE_predict) %>%
  as.data.frame() %>%
  head() %>%
  mutate(
    perc_diff = abs((reg_predict - oneSE_predict) * 100 / oneSE_predict)
  ) %>%
  knitr::kable(col.names = c("Prediction: Min MSE", "Prediction: 1SE", "Perc Diff"))
```

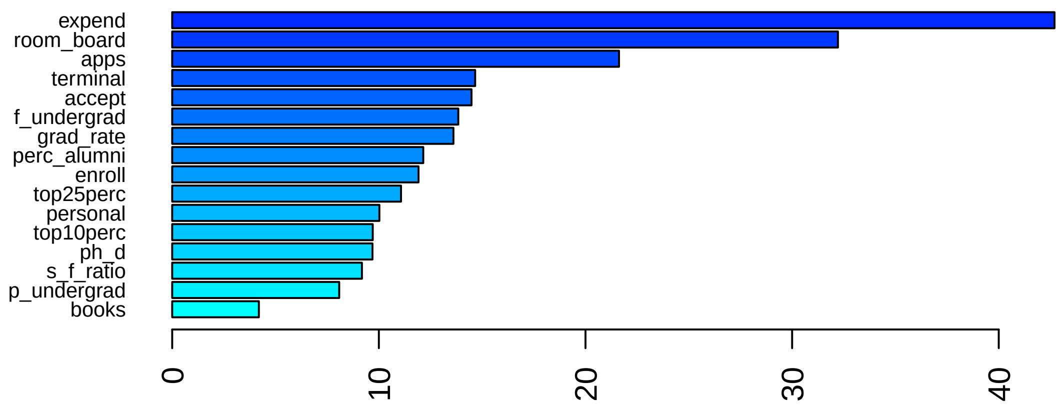
Prediction: Min MSE	Prediction: 1SE	Perc Diff
6827.90	6827.900	0.000000
11729.66	12488.737	6.078091
10194.94	9876.242	3.226919
10194.94	9876.242	3.226919
14146.09	14146.089	0.000000
10194.94	9876.242	3.226919

Part (b): Random Forest

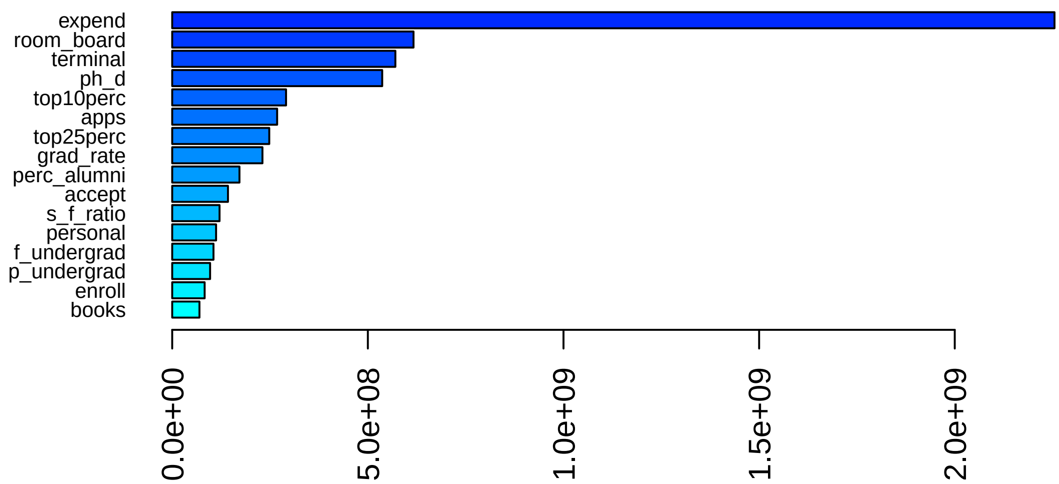
We then use random forest modeling on the training data to predict `outstate` using `caret` in conjunction with `ranger`. Using the importance method, our most important variables are `expend`, `room_board`, and `apps`, whereas with the impurity method, our most important variables are `expend`, `room_board`, and `terminal`. When we apply the model to make predictions on our testing data, our RMSE is 1992.244.

```
# Train caret random forest model

set.seed(2132)

# Grid of tuning parameters
rf_grid = expand.grid(mtry = seq(1, 16, 3),
                      splitrule = "variance",
                      min.node.size = 1:10)

# 10-fold cross-validation repeated 5 times
ctrl = trainControl(method = "repeatedcv", number = 10, repeats = 5)

# Find best-fitting model after model fitting to optimize computational efficiency
rf_college_fit = train(outstate ~ .,
                       data = training_df,
                       method = "ranger",
                       tuneGrid = rf_grid,
                       trControl = ctrl)
```

```
# Obtain variable importance using permutation method
set.seed(2132)
permutation_var_imp_rf = ranger(outstate ~ . ,
                        data = training_df,
                        mtry = rf_college_fit$bestTune[[1]],
                        splitrule = "variance",
                        min.node.size = rf_college_fit$bestTune[[3]],
                        importance = "permutation",
                        scale.permutation.importance = TRUE)

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




```r
# Obtain variable importance using impurity method
set.seed(2132)
impurity_var_imp_rf = ranger(outstate ~ . ,
                        data = training_df,
                        mtry = rf_college_fit$bestTune[[1]],
                        splitrule = "variance",
                        min.node.size = rf_college_fit$bestTune[[3]],
                        importance = "impurity",
                        scale.permutation.importance = TRUE)

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




```
# Report test error for caret
rf_college_preds_caret = predict(rf_college_fit, newdata = testing_df)
RMSE(rf_college_preds_caret, testing_df$outstate)
```

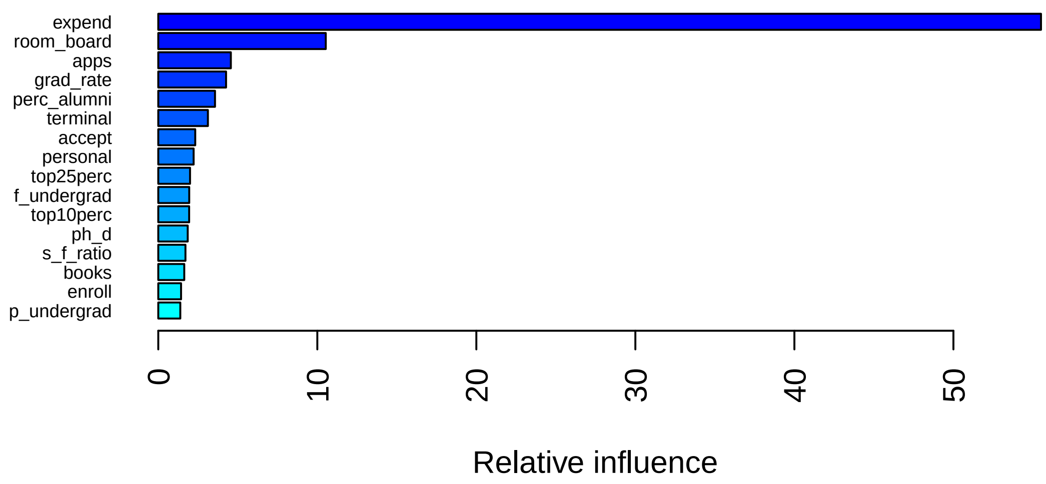
```
## [1] 1992.244
```

Part (c): Boosting

We train our model using gradient boosting as implemented with `gbm` in `caret`. After finding our optimal tuning parameters, we determine that our most important variables are once again `expend`, `room_board`, and `apps`, as we saw with random forest as well. When we apply the optimal model to the testing data, we obtain an RMSE of 1917.113, which is better performance than the random forest model.

```
# Train model using gbm in caret with grid of tuning parameters
set.seed(2132)

boost_grid = expand.grid(n.trees = seq(1, 5500, 500),
                         interaction.depth = 1:5,
                         shrinkage = c(0.001, 0.003, 0.005),
                         n.minobsinnode = 1)

college_boost_caret = train(outstate ~ .,
                            data = training_df,
                            method = "gbm",
                            tuneGrid = boost_grid,
                            trControl = ctrl,
                            verbose = FALSE)
```

```r
# Report the variable importance
summary(college_boost_caret$finalModel, las = 2, cBars = 19, cex.names = 0.6)
```




```
##                     var   rel.inf
## expend           expend 55.507059
## room_board   room_board 10.532704
## apps               apps  4.564322
## grad_rate     grad_rate  4.262154
## perc_alumni perc_alumni  3.567125
## terminal       terminal  3.120454
## accept           accept  2.324316
## personal       personal  2.222359
## top25perc     top25perc  1.997200
## f_undergrad f_undergrad  1.946269
## top10perc     top10perc  1.942414
## ph_d               ph_d  1.851104
## s_f_ratio     s_f_ratio  1.709427
## books             books  1.634280
## enroll           enroll  1.432573
## p_undergrad p_undergrad  1.386239
```

```r
# Report the test error
boost_college_preds = predict(college_boost_caret, newdata = testing_df)
RMSE(boost_college_preds, testing_df$outstate)
```

```
## [1] 1917.113
```

Question 2

Set-Up and Data Preprocessing

```r
set.seed(2132)

# Load data, clean column names, eliminate rows containing NA entries, factor outcome
data(OJ)
OJ_data = OJ %>%
  janitor::clean_names() %>%
  na.omit() %>%
  relocate("purchase", .after = "store") %>%
  mutate(
    purchase = as.factor(purchase)
  )

# Partition data into training/test sets (700 obs in training data)
OJ_indexTrain = createDataPartition(y = OJ_data$purchase,
                                    p = 0.653,
                                    list = FALSE)

OJ_training_df = OJ_data[OJ_indexTrain, ]

OJ_testing_df = OJ_data[-OJ_indexTrain, ]
```

Part (a): Classification Tree

Using `rpart`, we can build a classification tree on the OJ training data to predict `purchase` class. As with the regression tree, we can do so using either the model that minimizes cross-validation error or based on the 1SE rule; here, we do both for completeness.

Minimum MSE Rule

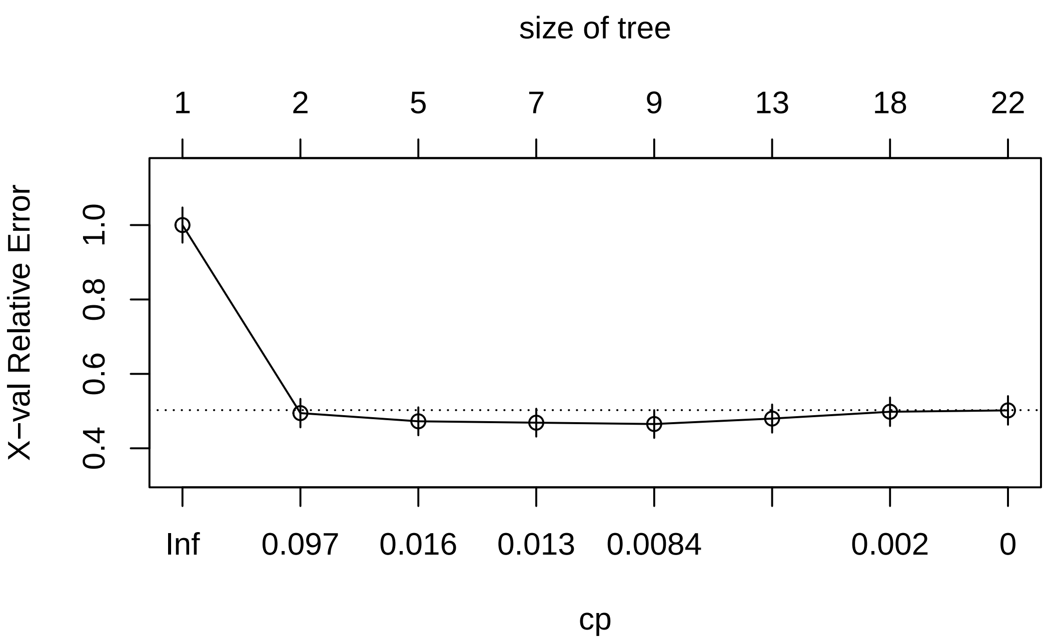
The tree that minimizes cross-validation error has 8 splits, leading to 9 terminal nodes (i.e. size 9).

```r
# Build classification tree using training data
set.seed(2132)

class_tree = rpart(formula = purchase ~ . ,
                   data = OJ_training_df,
                   control = rpart.control(cp = 0))

# Obtain cp table and plot vs cross-validation error
OJ_cp_table = printcp(class_tree)
```

```
##
## Classification tree:
## rpart(formula = purchase ~ ., data = OJ_training_df, control = rpart.control(cp = 0))
##
## Variables actually used in tree construction:
```

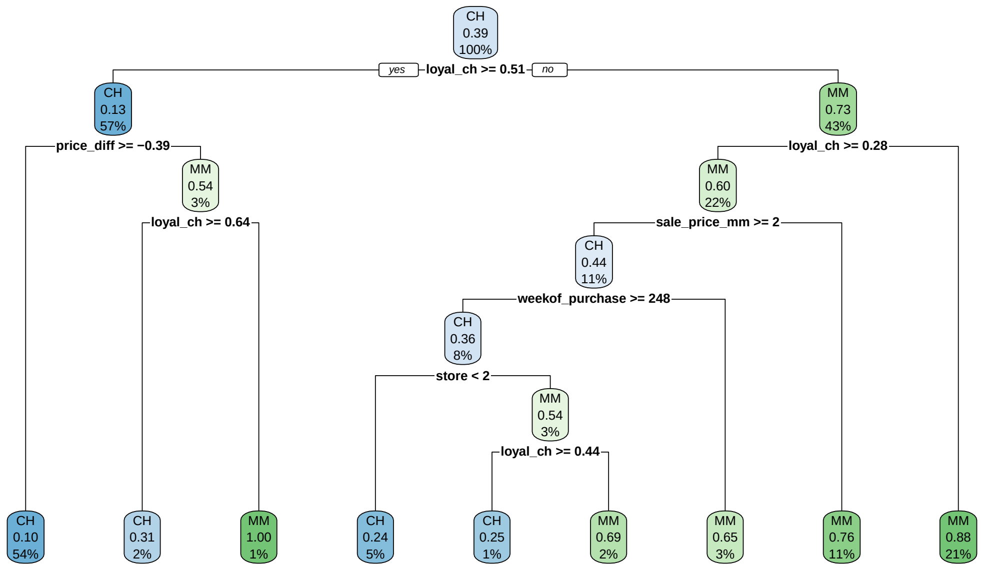
```
## [1] list_price_diff loyal_ch        price_diff      sale_price_mm
## [5] store           store_id        weekof_purchase
##
## Root node error: 273/700 = 0.39
##
## n= 700
##
##           CP nsplit rel error  xerror     xstd
## 1 0.5164835      0   1.00000 1.00000 0.047270
## 2 0.0183150      1   0.48352 0.49451 0.038237
## 3 0.0146520      4   0.42491 0.47253 0.037575
## 4 0.0109890      6   0.39560 0.46886 0.037462
## 5 0.0064103      8   0.37363 0.46520 0.037348
## 6 0.0021978     12   0.34799 0.47985 0.037799
## 7 0.0018315     17   0.33700 0.49817 0.038344
## 8 0.0000000     21   0.32967 0.50183 0.038451
```

```r
plotcp(class_tree)
```



The final tree appears as follows:

```r
# Obtain and plot final tree using min MSE rule
OJ_min_MSE = which.min(OJ_cp_table[,4])
final_class_tree = prune(class_tree, cp = OJ_cp_table[OJ_min_MSE,1])
rpart.plot(final_class_tree)
```

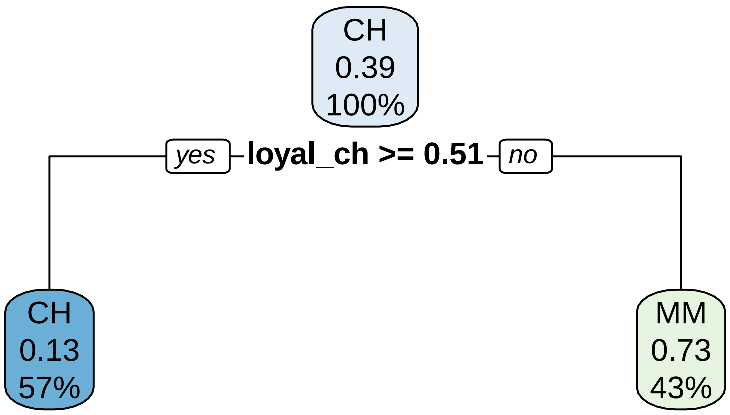



```
# plot(as.party(final_class_tree))
```

1SE Rule

Based on the 1SE rule, the final tree has only one split, which is based on the `loyal_ch` predictor, and two terminal nodes (size 2). This is quite a bit simpler and smaller than the tree that minimized cross-validation error, but is also significantly easier to interpret.

```
# Obtain and plot final tree using 1SE rule
final_class_tree_1SE = prune(class_tree,
      cp = OJ_cp_table[OJ_cp_table[,4]<OJ_cp_table[OJ_min_MSE,4]+OJ_cp_table[OJ_min_MSE,5],1][1])

# Plot of 1SE tree
rpart.plot(final_class_tree_1SE)
```



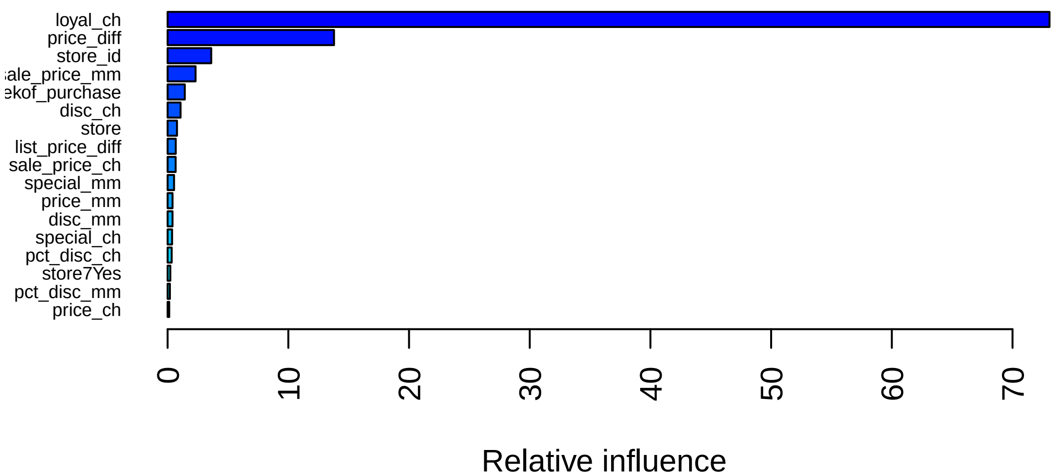

```
# plot(as.party(final_class_tree_1SE))
```

Part (b): Boosting

As in the regression case, we use `gbm`'s implementation in `caret`, except with "adaboosting" for classification of our `purchase` outcome variable. We find that our most important variable is `loyal_ch` by quite a bit, followed by `price_diff`, and then by `store_id`. When applied to our test data, the model gives an 18.9% error rate.

```
set.seed(2132)

# Fit optimal adaboost model for classification using training data
boost_grid_OJ = expand.grid(n.trees = seq(1, 5000, 500),
                          interaction.depth = 1:6,
                          shrinkage = c(0.001, 0.003, 0.005),
                          n.minobsinnode = 1)

ctrl_class = trainControl(method = "repeatedcv", number = 10, repeats = 5,
                          classProbs = TRUE,
                          summaryFunction = twoClassSummary)

OJ_boost_caret = train(purchase ~ .,
                       data = OJ_training_df,
                       method = "gbm",
                       tuneGrid = boost_grid_OJ,
                       trControl = ctrl_class,
                       distribution = "adaboost",
                       metric = "ROC",
                       verbose = FALSE)
```

```r
# Variable importance
# Method 2
summary(OJ_boost_caret$finalModel, las = 2, cBars = 19, cex.names = 0.6)
```



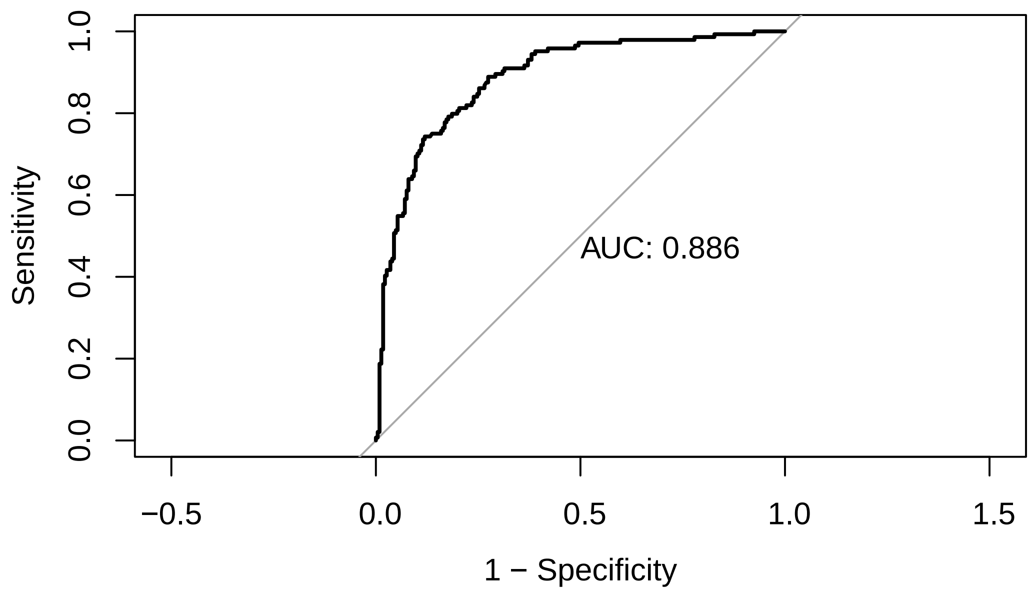

```
##                             var    rel.inf
## loyal_ch               loyal_ch 73.0636746
## price_diff           price_diff 13.7866503
## store_id               store_id  3.6127731
## sale_price_mm     sale_price_mm  2.3169757
## weekof_purchase weekof_purchase  1.4240288
## disc_ch                 disc_ch  1.0755046
## store                     store  0.7771457
## list_price_diff list_price_diff  0.6731588
## sale_price_ch     sale_price_ch  0.6661480
## special_mm           special_mm  0.5334967
## price_mm               price_mm  0.4090946
## disc_mm                 disc_mm  0.4074160
## special_ch           special_ch  0.3707729
## pct_disc_ch         pct_disc_ch  0.3373376
## store7Yes             store7Yes  0.2235972
## pct_disc_mm         pct_disc_mm  0.1886238
## price_ch               price_ch  0.1336018
```

```r
# Test error rate
# Method 2 only for now
boost_OJ_preds = predict(OJ_boost_caret, newdata = OJ_testing_df)
error_rate = mean(boost_OJ_preds != OJ_testing_df$purchase)*100
error_rate
```

```
## [1] 18.91892
```

Just for fun, we can visualize the ROC and confusion matrix as well.

```
boost_preds_prob = predict(OJ_boost_caret, newdata = OJ_testing_df, type = "prob")[, 1]
boost_roc_curve = roc(OJ_testing_df$purchase, boost_preds_prob)
plot(boost_roc_curve, legacy.axes = TRUE, print.auc = TRUE)
```




```
confusionMatrix(boost_OJ_preds, OJ_testing_df$purchase)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##         CH 192  36
##         MM  34 108
##
##                Accuracy : 0.8108
##                  95% CI : (0.7671, 0.8494)
##     No Information Rate : 0.6108
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.6011
##
##  Mcnemar's Test P-Value : 0.9049
##
##             Sensitivity : 0.8496
##             Specificity : 0.7500
##          Pos Pred Value : 0.8421
##          Neg Pred Value : 0.7606
##              Prevalence : 0.6108
##          Detection Rate : 0.5189
##    Detection Prevalence : 0.6162
```

```
##       Balanced Accuracy : 0.7998
##
##        'Positive' Class : CH
##
```