HW3 Markdown

Santanu Mukherjee

04/13/2022

R Markdown

Q 1

Friedman (1991) introduced several benchmark data sets create by simulation. One of these simulations used the following nonlinear equation to create data:

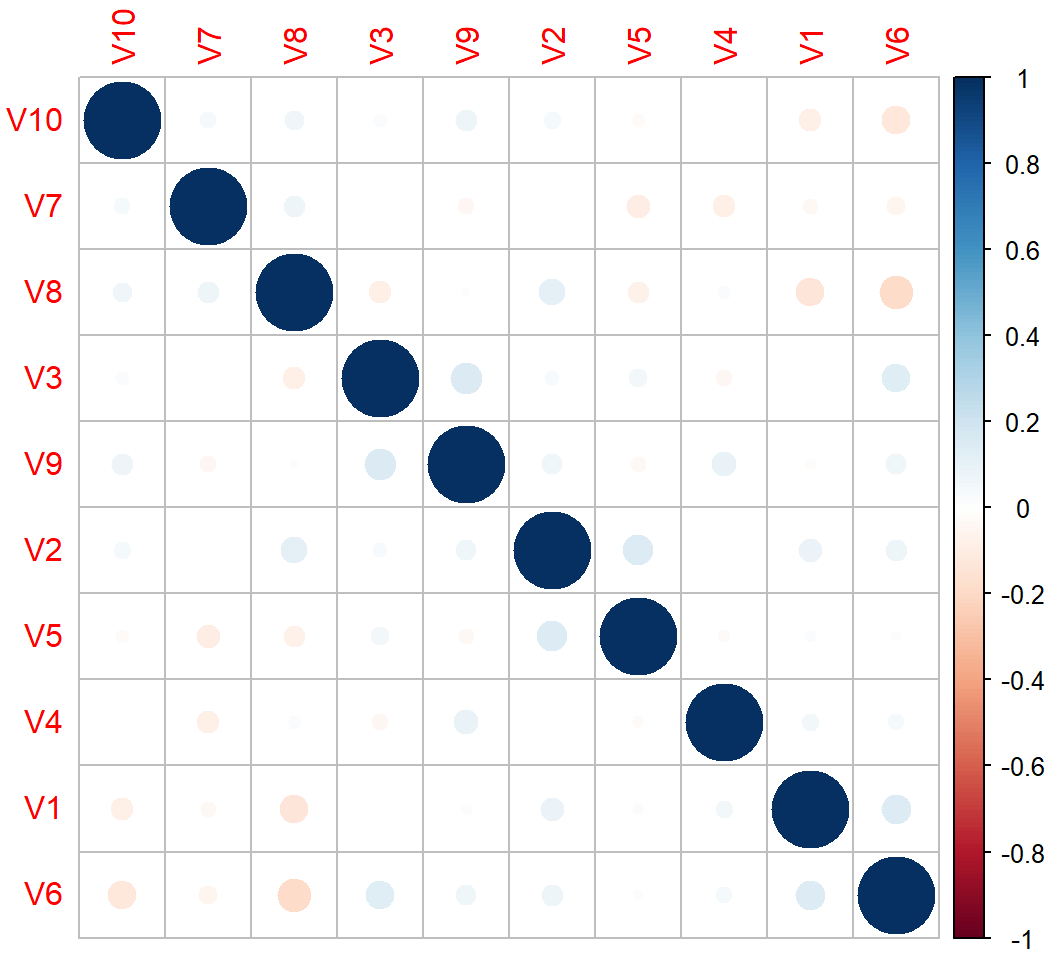
$y = 10\ sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + N(0, \sigma^2)$

where the x values are random variables uniformly distributed between $[0, 1]$ (there are also 5 other non-informative variables also created in the simulation). The package **mlbench** contains a function called **mlbench.friedman1** that simulates these data:

```
set.seed(200)
simulated <- mlbench.friedman1(200, sd = 1)
simulated <- cbind(simulated$x, simulated$y)
simulated <- as.data.frame(simulated)
colnames(simulated)[ncol(simulated)] <- "y"

str(simulated)
```

```
## 'data.frame':    200 obs. of  11 variables:
##  $ V1 : num  0.534 0.584 0.59 0.691 0.667 ...
##  $ V2 : num  0.648 0.438 0.588 0.226 0.819 ...
##  $ V3 : num  0.8508 0.6727 0.4097 0.0334 0.7168 ...
##  $ V4 : num  0.1816 0.6692 0.3381 0.0669 0.8032 ...
##  $ V5 : num  0.929 0.1638 0.8941 0.6374 0.0831 ...
##  $ V6 : num  0.3618 0.4531 0.0268 0.525 0.2234 ...
##  $ V7 : num  0.827 0.649 0.179 0.513 0.664 ...
##  $ V8 : num  0.421 0.845 0.35 0.797 0.904 ...
##  $ V9 : num  0.5911 0.9282 0.0176 0.6899 0.397 ...
##  $ V10: num  0.589 0.758 0.444 0.445 0.55 ...
##  $ y  : num  18.5 16.1 17.8 13.8 18.4 ...
```

```
Cor = round(cor(simulated[,1:10]),4)
corrplot(Cor, order = "hclust")
```



Q 1 a

Fit a random forest model to all of the predictors, then estimate the variable importance scores:

```
model1 <- randomForest(y ~ ., data = simulated, importance = TRUE, ntree = 5000)
rfImp1 <- varImp(model1, scale = FALSE)
```

Did the random forest model significantly use the uninformative predictors $(V6 – V10)$?

Answer 1 (a)

```
rfImp1 = rfImp1[ order(-rfImp1), , drop=FALSE ]
print("randomForest (no correlated predictor)")
```

```
## [1] "randomForest (no correlated predictor)"
```

```
print(rfImp1)
```

```
##         Overall
## V1   8.79515634
## V4   7.64635222
## V2   6.56654060
## V5   2.15996803
## V3   0.73300417
## V6   0.13519686
## V7   0.06891977
## V10 -0.02833177
## V9  -0.08615019
## V8  -0.14606085
```

```
cor(simulated$V6, simulated$V1)
```

```
## [1] 0.1447624
```

```
cor(simulated$V7, simulated$V1)
```

```
## [1] -0.03741029
```

```
cor(simulated$V8, simulated$V1)
```

```
## [1] -0.1344975
```

```
cor(simulated$V9, simulated$V1)
```

```
## [1] -0.01926865
```

```
cor(simulated$V10, simulated$V1)
```

```
## [1] -0.08128228
```

No. the model did not use the uninformative predictors $(V6–V10)$ because these uninformative predictors $(V6–V10)$ are very low correlation with $V1$. Also based on the importance scores, these uninformative predictors $(V6–V10)$ has very low important scores.

Q 1 b

Now add an additional predictor that is highly correlated with one of the informative predictors. For example:

```
simulated$duplicate1 <- simulated$V1 + rnorm(200) * .1
cor(simulated$duplicate1, simulated$V1)
```

```
## [1] 0.953051
```

Fit another random forest model to these data. Did the importance score for $V1$ change? What happens when you add another predictor that is also highly correlated with $V1$?

Answer 1 (b)

```
model2 = randomForest( y ~ ., data=simulated, importance=TRUE, ntree=5000 )
rfImp2 = varImp(model2, scale=FALSE)
rfImp2 = rfImp2[ order(-rfImp2), , drop=FALSE ]
print("randomForest (one correlated predictor)")
```

```
## [1] "randomForest (one correlated predictor)"
```

```
print(rfImp2)
```

```
##                Overall
## V4          6.93143877
## V1          6.17739898
## V2          5.97111496
## duplicate1  3.81659740
## V5          2.07678953
## V3          0.59650620
## V6          0.18118973
## V7          0.05271119
## V10         0.02412532
## V9         -0.05854792
## V8         -0.09553865
```

```
simulated$duplicate2 = simulated$V1 + rnorm(200) * 0.1
cor(simulated$duplicate2,simulated$V1)
```

```
## [1] 0.9371348
```

```
model3 = randomForest( y ~ ., data=simulated, importance=TRUE, ntree=5000 )
rfImp3 = varImp(model3, scale=FALSE)
rfImp3 = rfImp3[ order(-rfImp3), , drop=FALSE ]
print("randomForest (two correlated predictors)")
```

```
## [1] "randomForest (two correlated predictors)"
```

```
print(rfImp3)
```

```
##                Overall
## V4          7.24371697
## V2          6.18361651
## V1          5.52439122
## duplicate1  3.08165186
## V5          2.00756558
## duplicate2  1.63334672
## V3          0.49887005
## V6          0.18779180
## V7          0.01315613
## V10        -0.01824960
## V8         -0.06994458
## V9         -0.08607670
```

So, when the first highly correlated additional predictor ($duplicate1$) to $V1$ was added, the importance score for V1 reduced from **8.795** to **6.177**. But, when another highly correlated additional predictor ($duplicate2$) to $V1$ was added, the importance score for V1 reduced to **5.524**. So, based on the trend we can say that as we keep adding highly correlated predictors to $V1$, the importance score of $V1$ keeps reducing.

Q 1 c

Use the **cforest** function in the party package to fit a random forest model using conditional inference trees. The party package function **varimp** can calculate predictor importance. Do these importances show the same pattern as the traditional random forest model in part (a)?

Answer 1 (c)

```
library(party)

use_conditional_true = T # whether to use the conditional argument in the cforest function call

simulated$duplicate1 = NULL
simulated$duplicate2 = NULL

model1 = cforest( y ~ ., data=simulated )
cfImp1 = as.data.frame(varimp(model1),conditional=use_conditional_true)
cfImp1 = cfImp1[ order(-cfImp1), , drop=FALSE ]
print(sprintf("cforest (no correlated predictor); varimp(*,conditional=%s)",use_conditional_true))
```

```
## [1] "cforest (no correlated predictor); varimp(*,conditional=TRUE)"
```

```
print(cfImp1)
```

```
##     varimp(model1)
## V1     8.956529026
## V4     8.490407094
## V2     6.800602352
## V5     1.888119636
## V7     0.053420891
## V3     0.034244559
## V9     0.001405308
## V6    -0.011297698
## V8    -0.020218385
## V10   -0.047113025
```

```
# Now we add correlated predictors one at a time
simulated$duplicate1 = simulated$V1 + rnorm(200) * 0.1

model2 = cforest( y ~ ., data=simulated )
cfImp2 = as.data.frame(varimp(model2),conditional=use_conditional_true)
cfImp2 = cfImp2[ order(-cfImp2), , drop=FALSE ]
print(sprintf("cforest (one correlated predictor); varimp(*,conditional=%s)",use_conditional_true))
```

```
## [1] "cforest (one correlated predictor); varimp(*,conditional=TRUE)"
```

```
print(cfImp2)
```

```
##            varimp(model2)
## V4            7.405686843
## V2            6.138661300
## V1            5.075786734
## duplicate1    4.366938094
## V5            1.849685831
## V6            0.030234563
## V7            0.029129547
## V3            0.009901651
## V10           0.002260404
## V8           -0.010117523
## V9           -0.036704211
```

```
simulated$duplicate2 = simulated$V1 + rnorm(200) * 0.1

model3 = cforest( y ~ ., data=simulated )
cfImp3 = as.data.frame(varimp(model3),conditional=use_conditional_true)
cfImp3 = cfImp3[ order(-cfImp3), , drop=FALSE ]
print(sprintf("cforest (two correlated predictor); varimp(*,conditional=%s)",use_conditional_true))
```

```
## [1] "cforest (two correlated predictor); varimp(*,conditional=TRUE)"
```

```
print(cfImp3)
```

```
##            varimp(model3)
## V4            7.304614091
## V2            5.545043888
## V1            4.416334604
## duplicate1    3.581017934
## V5            1.900738242
## duplicate2    1.289941157
## V3            0.036341993
## V7            0.011429188
## V6           -0.005362661
## V10          -0.032135337
## V8           -0.033245603
## V9           -0.043173228
```

Now, with **cforest**, we see a similar trend. that the importance score of $V1$ keeps reducing as we introduce additional predictors in the model which are highly correlated to $V1$.

Q 1 d

Repeat this process with different tree models that we discussed in the class and also you need to consider two new **Cubist** and **XGBoost** Regression methods (You may use **Google** to search and study). Does the same pattern occur?

Answer 1 (d)

Boosting Method

```
# Lets try the same experiment but using boosted trees:

library(gbm)

simulated$duplicate1 = NULL
simulated$duplicate2 = NULL

model1 = gbm( y ~ ., data=simulated, distribution="gaussian", n.trees=5000 )
print(sprintf("gbm (no correlated predictor)"))
```

```
## [1] "gbm (no correlated predictor)"
```

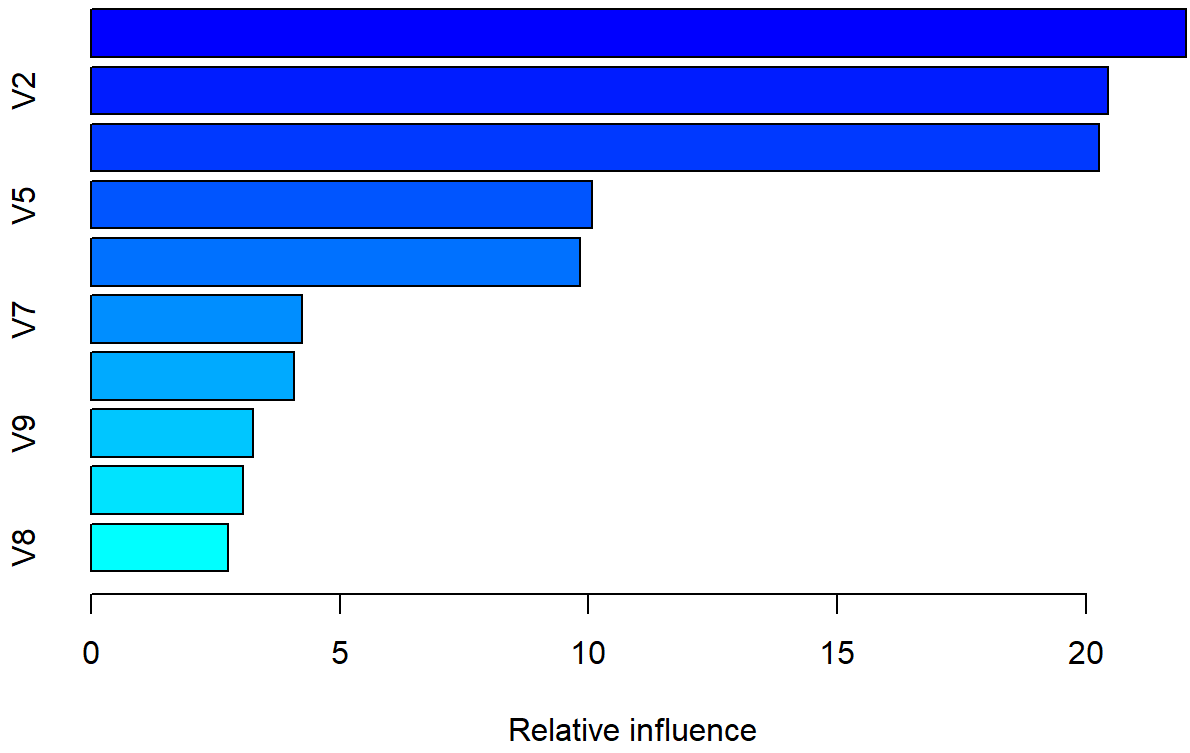
```
print(summary(model1,plotit=F)) # the summary method gives variable importance ...
```

```
##     var   rel.inf
## V4   V4 22.004938
## V2   V2 20.435410
## V1   V1 20.265030
## V5   V5 10.070295
## V3   V3  9.829996
## V7   V7  4.249849
## V6   V6  4.080398
## V9   V9  3.254157
## V10 V10  3.049548
## V8   V8  2.760380
```

```
rel_inf = relative.influence(model1, n.trees=5000)
rel_inf = sort(rel_inf, decreasing=T)/sum(rel_inf)*100
sum(rel_inf[1:2])
```

```
## [1] 42.44035
```

```
## Plot of Important Variables
summary(model1)
```




```
##     var   rel.inf
## V4   V4 22.004938
## V2   V2 20.435410
## V1   V1 20.265030
## V5   V5 10.070295
## V3   V3  9.829996
## V7   V7  4.249849
## V6   V6  4.080398
## V9   V9  3.254157
## V10 V10  3.049548
## V8   V8  2.760380
```

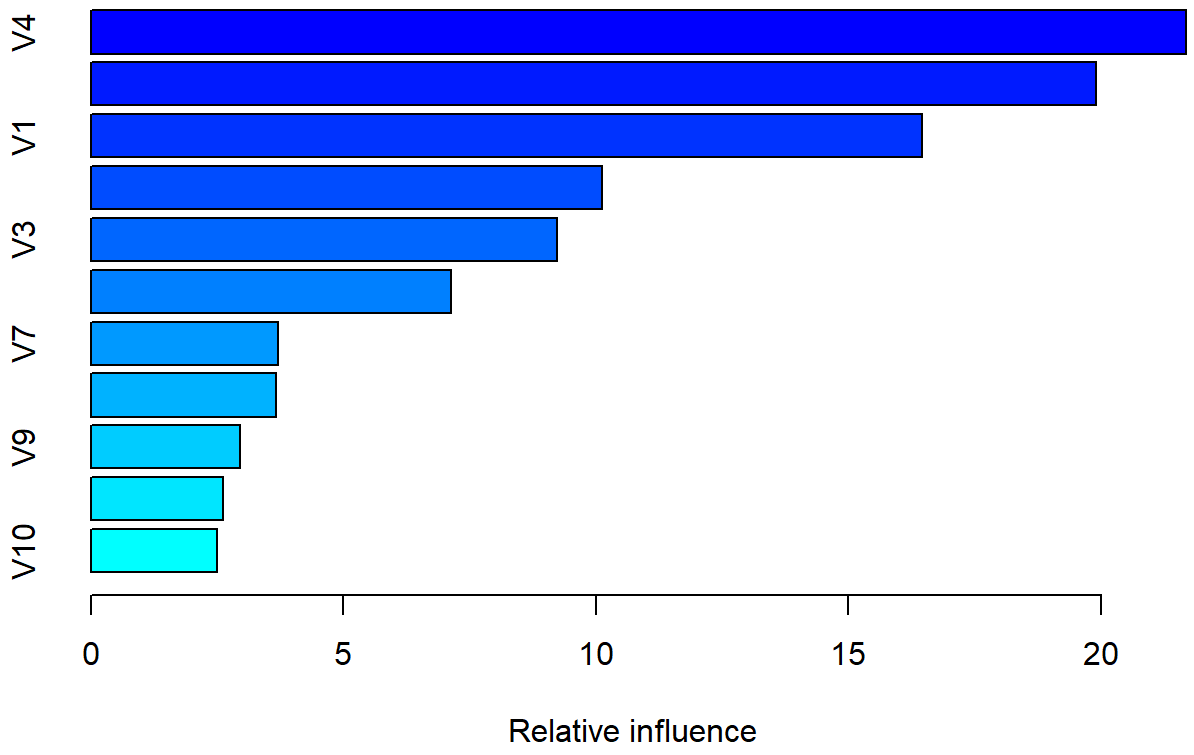
```
# Now we add correlated predictors one at a time
simulated$duplicate1 = simulated$V1 + rnorm(200) * 0.1

model2 = gbm( y ~ ., data=simulated, distribution="gaussian", n.trees=5000 )
print(sprintf("gbm (one correlated predictor)"))
```

```
## [1] "gbm (one correlated predictor)"
```

```
print(summary(model2,plotit=F))
```

```
##                    var   rel.inf
## V4                  V4 21.688696
## V2                  V2 19.910103
## V1                  V1 16.469519
## V5                  V5 10.118159
## V3                  V3  9.238498
## duplicate1 duplicate1  7.142462
## V7                  V7  3.700452
## V6                  V6  3.658865
## V9                  V9  2.951151
## V8                  V8  2.626449
## V10                V10  2.495646
```

```
## Plot of Important Variables
summary(model2)
```



```
##                   var   rel.inf
## V4                 V4 21.688696
## V2                 V2 19.910103
## V1                 V1 16.469519
## V5                 V5 10.118159
## V3                 V3  9.238498
## duplicate1 duplicate1  7.142462
## V7                 V7  3.700452
## V6                 V6  3.658865
## V9                 V9  2.951151
## V8                 V8  2.626449
## V10               V10  2.495646
```

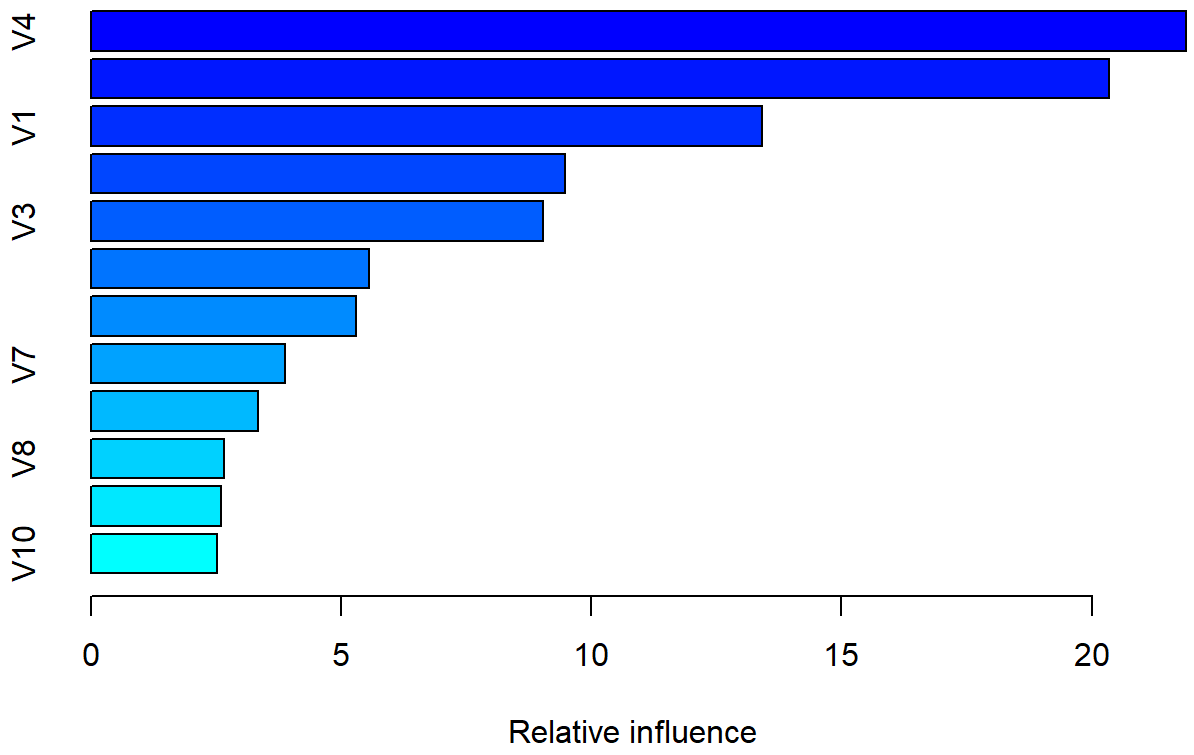
```
simulated$duplicate2 = simulated$V1 + rnorm(200) * 0.1

model3 = gbm( y ~ ., data=simulated, distribution="gaussian", n.trees=5000 )
print(sprintf("gbm (two correlated predictor)"))
```

```
## [1] "gbm (two correlated predictor)"
```

```
print(summary(model3,plotit=F))
```

```
##                   var   rel.inf
## V4                 V4 21.884202
## V2                 V2 20.348369
## V1                 V1 13.406902
## V5                 V5  9.466042
## V3                 V3  9.031222
## duplicate2 duplicate2  5.554191
## duplicate1 duplicate1  5.303957
## V7                 V7  3.876792
## V6                 V6  3.331377
## V8                 V8  2.667903
## V9                 V9  2.610123
## V10               V10  2.518920
```

```
## Plot of Important Variables
summary(model3)
```




```
##                   var   rel.inf
## V4                 V4 21.884202
## V2                 V2 20.348369
## V1                 V1 13.406902
## V5                 V5  9.466042
## V3                 V3  9.031222
## duplicate2 duplicate2  5.554191
## duplicate1 duplicate1  5.303957
## V7                 V7  3.876792
## V6                 V6  3.331377
## V8                 V8  2.667903
## V9                 V9  2.610123
## V10               V10  2.518920
```

Using **Cubist**

```
##
# Lets try the same experiment but using boosted trees:

library(Cubist)


simulated$duplicate1 = NULL
simulated$duplicate2 = NULL

pred = simulated[-11]
resp = simulated$y
model1.cubist = cubist(x = pred, y = resp)
model1.cubist
```

```
##
## Call:
## cubist.default(x = pred, y = resp)
##
## Number of samples: 200
## Number of predictors: 10
##
## Number of committees: 1
## Number of rules: 1
```

```
## Plot of Important Variables
summary(model1.cubist)
```

```
##
## Call:
## cubist.default(x = pred, y = resp)
##
##
## Cubist [Release 2.07 GPL Edition]  Tue Apr 12 23:17:28 2022
## ---------------------------------
##
##     Target attribute `outcome'
##
## Read 200 cases (11 attributes) from undefined.data
##
## Model:
##
##   Rule 1: [200 cases, mean 14.416183, range 3.55596 to 28.38167, est err 1.944664]
##
##  outcome = 0.183529 + 8.9 V4 + 7.9 V1 + 7.1 V2 + 5.3 V5
##
##
## Evaluation on training data (200 cases):
##
##     Average  |error|           2.224012
##     Relative |error|               0.55
##     Correlation coefficient        0.84
##
##
##  Attribute usage:
##    Conds  Model
##
##           100%    V1
##           100%    V2
##           100%    V4
##           100%    V5
##
##
## Time: 0.0 secs
```

```
model1.cubist.imp = caret::varImp(model1.cubist)



# Now we add correlated predictors one at a time - Duplicate1
simulated$duplicate1 = simulated$V1 + rnorm(200) * 0.1

pred = simulated[-11]
resp = simulated$y
model2.cubist = cubist(x = pred, y = resp)
model2.cubist
```

```
##
## Call:
## cubist.default(x = pred, y = resp)
##
## Number of samples: 200
## Number of predictors: 11
##
## Number of committees: 1
## Number of rules: 1
```

```
## Plot of Important Variables
summary(model2.cubist)
```

```
##
## Call:
## cubist.default(x = pred, y = resp)
##
##
## Cubist [Release 2.07 GPL Edition]  Tue Apr 12 23:17:28 2022
## -------------------------------
##
##     Target attribute `outcome'
##
## Read 200 cases (12 attributes) from undefined.data
##
## Model:
##
##   Rule 1: [200 cases, mean 14.416183, range 3.55596 to 28.38167, est err 1.941464]
##
##  outcome = 0.619511 + 9 V4 + 8.2 V1 + 7.2 V2 + 5.2 V5 - 1.6 V6
##
##
## Evaluation on training data (200 cases):
##
##     Average  |error|           2.154252
##     Relative |error|               0.53
##     Correlation coefficient        0.85
##
##
##  Attribute usage:
##    Conds  Model
##
##           100%    V1
##           100%    V2
##           100%    V4
##           100%    V5
##           100%    V6
##
##
## Time: 0.0 secs
```

```
model2.cubist.imp = caret::varImp(model2.cubist)


## Next we add another correlated predictor - Duplicate2
simulated$duplicate2 = simulated$V1 + rnorm(200) * 0.1

pred = simulated[-11]
resp = simulated$y
model3.cubist = cubist(x = pred, y = resp)
model3.cubist
```

```
##
## Call:
## cubist.default(x = pred, y = resp)
##
## Number of samples: 200
## Number of predictors: 12
##
## Number of committees: 1
## Number of rules: 1
```

```
## Plot of Important Variables
summary(model3.cubist)
```

```
##
## Call:
## cubist.default(x = pred, y = resp)
##
##
## Cubist [Release 2.07 GPL Edition]  Tue Apr 12 23:17:29 2022
## ---------------------------------
##
##     Target attribute `outcome'
##
## Read 200 cases (13 attributes) from undefined.data
##
## Model:
##
##   Rule 1: [200 cases, mean 14.416183, range 3.55596 to 28.38167, est err 1.941464]
##
##  outcome = 0.619511 + 9 V4 + 8.2 V1 + 7.2 V2 + 5.2 V5 - 1.6 V6
##
##
## Evaluation on training data (200 cases):
##
##     Average  |error|           2.076192
##     Relative |error|               0.51
##     Correlation coefficient        0.86
##
##
##  Attribute usage:
##    Conds  Model
##
##           100%    V1
##           100%    V2
##           100%    V4
##           100%    V5
##           100%    V6
##
##
## Time: 0.0 secs
```

```
model3.cubist.imp = caret::varImp(model3.cubist)
```

As we add on correlated predictors , the error rate (MSE on training data) decreases.

Q 2

The **"churn"** data set in the $MLC++$ software package was developed to predict telecom customer churn based on information about their account. The data files state that the data are *"artificial based on claims similar to real world."* The data consist of **19** predictors related to the customer account, such as the number of customer service calls, the area code, and the number of minutes. The outcome is whether the customer churned.

The data are contained in the C50 package and can be loaded using

```
library(modeldata)
data(mlc_churn)
## Two objects are loaded: churnTrain and churnTest
str(mlc_churn)
```

```
## tibble [5,000 x 20] (S3: tbl_df/tbl/data.frame)
##  $ state                         : Factor w/ 51 levels "AK","AL","AR",..: 17 36 32 36 37 2 20 25 19 50
...
##  $ account_length                : int [1:5000] 128 107 137 84 75 118 121 147 117 141 ...
##  $ area_code                     : Factor w/ 3 levels "area_code_408",..: 2 2 2 1 2 3 3 2 1 2 ...
##  $ international_plan            : Factor w/ 2 levels "no","yes": 1 1 1 2 2 2 1 2 1 2 ...
##  $ voice_mail_plan               : Factor w/ 2 levels "no","yes": 2 2 1 1 1 1 2 1 1 2 ...
##  $ number_vmail_messages         : int [1:5000] 25 26 0 0 0 0 24 0 0 37 ...
##  $ total_day_minutes             : num [1:5000] 265 162 243 299 167 ...
##  $ total_day_calls               : int [1:5000] 110 123 114 71 113 98 88 79 97 84 ...
##  $ total_day_charge              : num [1:5000] 45.1 27.5 41.4 50.9 28.3 ...
##  $ total_eve_minutes             : num [1:5000] 197.4 195.5 121.2 61.9 148.3 ...
##  $ total_eve_calls               : int [1:5000] 99 103 110 88 122 101 108 94 80 111 ...
##  $ total_eve_charge              : num [1:5000] 16.78 16.62 10.3 5.26 12.61 ...
##  $ total_night_minutes           : num [1:5000] 245 254 163 197 187 ...
##  $ total_night_calls             : int [1:5000] 91 103 104 89 121 118 118 96 90 97 ...
##  $ total_night_charge            : num [1:5000] 11.01 11.45 7.32 8.86 8.41 ...
##  $ total_intl_minutes            : num [1:5000] 10 13.7 12.2 6.6 10.1 6.3 7.5 7.1 8.7 11.2 ...
##  $ total_intl_calls              : int [1:5000] 3 3 5 7 3 6 7 6 4 5 ...
##  $ total_intl_charge             : num [1:5000] 2.7 3.7 3.29 1.78 2.73 1.7 2.03 1.92 2.35 3.02 ...
##  $ number_customer_service_calls: int [1:5000] 1 1 0 2 3 0 3 0 1 0 ...
##  $ churn                         : Factor w/ 2 levels "yes","no": 2 2 2 2 2 2 2 2 2 2 ...
```

```
table(mlc_churn$churn)
```

```
##
##  yes   no
##  707 4293
```

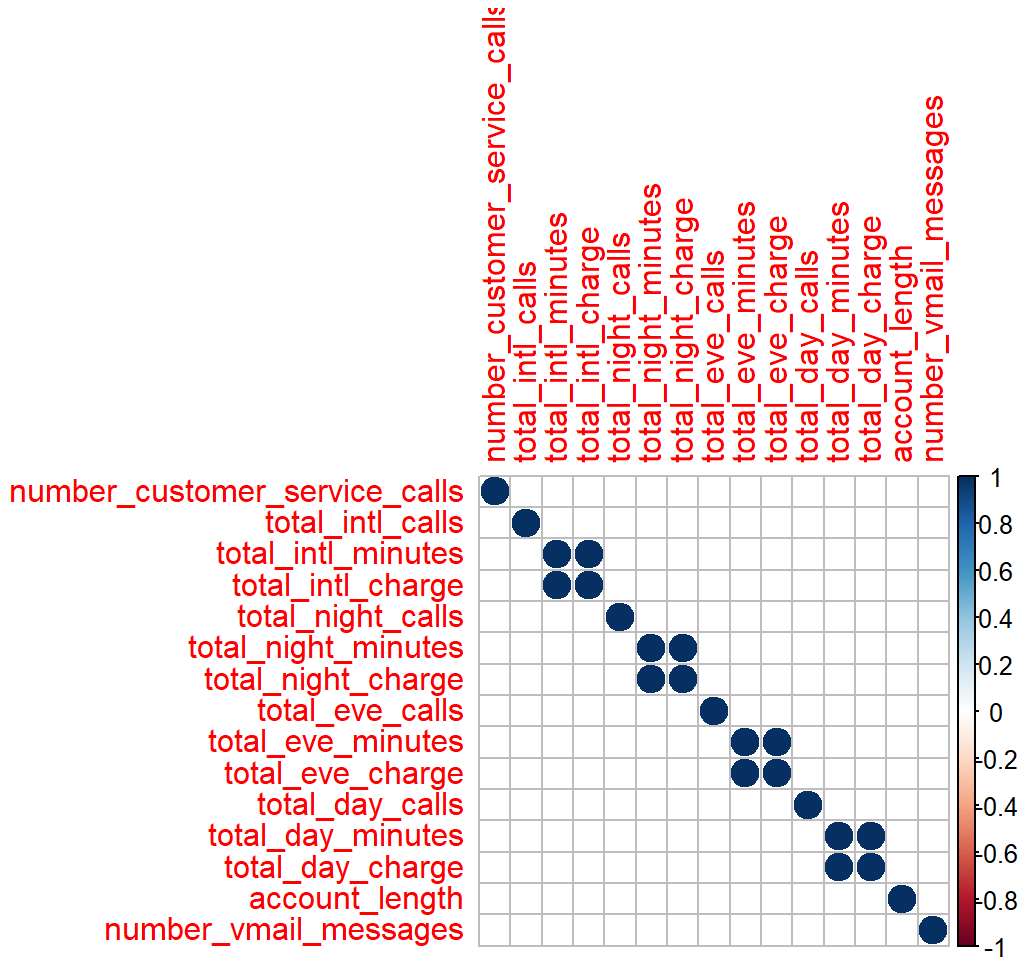
```
set.seed(1)
ind = sample(2, nrow(mlc_churn), replace=TRUE,prob = c(0.8,0.2))
churnTrain = mlc_churn[ind==1,]
churnTest  = mlc_churn[ind==2,]
```

Q 2 a

Explore the data by visualizing the relationship between the predictors and the outcome.Are there important features of the predictor data themselves, such as between-predictor correlations or degenerate distributions? Can functions of more than one predictor be used to model the data more effectively?

Answer 2 (a)

```
# Drop all factor predictors:
#
churnTrain1 = churnTrain[,-c(1,3,4,5)]
churnTest1 = churnTest[,-c(1,3,4,5)]

# Build various linear models:
#
x1 = churnTrain1[,-16] # drop the churn response
y1 = churnTrain1[,16]



#Correlation plot with other predictors
#pairs(x[,1:15])
Cor = round(cor(x1[,1:15]))
#ggcorrplot(Cor)

library(corrplot)
corrplot(Cor, order = "hclust")
```




```
highCorr <- findCorrelation(Cor, cutoff = .5)
head(highCorr)
```

```
## [1]  3  6  9 12
```

```
# Look for (and drop) zero variance columns:
zv_cols = nearZeroVar(x1)
w = x1[,-zv_cols]

# There are no nearzeroVar columns

print("There are not nearZeroVar predictors")
```

```
## [1] "There are not nearZeroVar predictors"
```

```
# Get a high level View of which predictors might be most predictive:
#
```

There are no nearzeroVar columns which means all predictors are important. There are few predictors which havestrong correlation.

Q 2 b

Apply boosting, bagging, random forests, and $BART$ to this data set. Be sure to fit the models on a training set and to evaluate their performance on a test set. How accurate are the results compared to simple methods like linear or logistic regression? What criteria should be used to evaluate the effectiveness and performance of the models.

Answer 2 (b)

```
# Modelling - bagging boosting, random forests and BART
# Random forest , bagging

# y is the churnTrain response


churnTrain1$y = ifelse(churnTrain1$churn == "yes",1,0)
churnTest1$y = ifelse(churnTest1$churn == "yes",1,0)

churnTrainreal = churnTrain1[,-16]
churnTestreal = churnTest1[,-16]



#Bagging

sim.bag = bagging(y ~ ., data = churnTrainreal, ntree = 3000, mtry = 2, na.action=na.omit)
sim.bag.pred.tr = predict(sim.bag, newdata = churnTrainreal, type = "class")
sim.bag.pred.te = predict(sim.bag, newdata = churnTestreal, type = "class")
print("Testing error using bagging")
```

```
## [1] "Testing error using bagging"
```

```
errbag = sqrt(mean((sim.bag.pred.te - churnTestreal$y)^2))



#Random Forest

sim.rf = randomForest(y ~ ., data = churnTrainreal, ntree = 3000, mtry = 2, na.action=na.omit)
sim.rf.pred.tr = predict(sim.rf, newdata = churnTrainreal, type = "class")
sim.rf.pred.te = predict(sim.rf, newdata = churnTestreal, type = "class")
print("Testing error using Random Forest")
```

```
## [1] "Testing error using Random Forest"
```

```
errrf = sqrt(mean((sim.rf.pred.te - churnTestreal$y)^2))


#boosting

sim.boost = gbm(y ~ ., data = churnTrainreal, shrinkage=0.01,
  distribution = 'bernoulli', n.trees = 3000, verbose=F)
sim.b.pred.tr = predict(sim.boost, newdata = churnTrainreal, n.trees = 1000, type = "response")
sim.b.pred.tr = ifelse(sim.b.pred.tr>0.5,1,0)
sim.b.pred.te = predict(sim.boost, newdata = churnTestreal, n.trees = 1000, type = "response")
sim.b.pred.te = ifelse(sim.b.pred.te>0.5,1,0)
print("Testing error using boost")
```

```
## [1] "Testing error using boost"
```

```
errboost = sqrt(mean((sim.b.pred.te - churnTestreal$y)^2))


#BART

#library(rJava)
#library(bartMachine)

#sim.bart = bart(churnTrainreal, y, ndpost = 200)

#sim.bart = bart(y ~ ., data = churnTrainreal, ntree = 3000, mtry = 2, na.action=na.omit)
#sim.bart.pred.tr = predict(sim.bart, newdata = churnTrainreal, type = "class")
#sim.bart.pred.te = predict(sim.bart, newdata = churnTestreal, type = "class")
#print("Testing error using BART")
#sqrt(mean((sim.bart.pred.te - churnTestreal$y)^2))



# Linear Regression

sim.lm = lm(y ~ ., data = churnTrainreal)
sim.lm.pred.tr = predict(sim.lm, newdata = churnTrainreal)
sim.lm.pred.te = predict(sim.lm, newdata = churnTestreal)
print("Testing error using Random Forest")
```

```
## [1] "Testing error using Random Forest"
```

```
errlm = sqrt(mean((sim.lm.pred.te - churnTestreal$y)^2))


#Logistic Regression
sim.glm <- glm(as.factor(y) ~ ., data = churnTrainreal, family = "binomial")
sim.glm.pred.tr = predict(sim.glm, newdata = churnTrainreal)
sim.glm.pred.te = predict(sim.glm, newdata = churnTestreal)
print("Testing error using Logistic Regression")
```

```
## [1] "Testing error using Logistic Regression"
```

```
errlogit = sqrt(mean((sim.glm.pred.te - churnTestreal$y)^2))
```

```
results.table <- data.frame(errbag, errrf, errboost, errlm, errlogit)

colnames(results.table) = c("Error Bagging", "Error Random Forest", "Error Boosting", "Error Linear Regress
ion", "Error Logistic Regression")

results.table
```

```
##   Error Bagging Error Random Forest Error Boosting Error Linear Regression
## 1     0.2688968           0.2704374       0.361387               0.3324065
##   Error Logistic Regression
## 1                  2.550328
```

Based on the data, we can say that Bagging has the least RMSE and so it is the most effective model.

Chapter 8 - Book - An Introduction to Statistical Learning - 8.4 - Exercises page 334:

Q8.4.9

This problem involves the **OJ** data set which is part of the **ISLR2** package.

```
glimpse(OJ)
```

```
## Rows: 1,070
## Columns: 18
## $ Purchase       <fct> CH, CH, CH, MM, CH, CH, CH, CH, CH, CH, CH, CH, CH, CH, CH,~
## $ WeekofPurchase <dbl> 237, 239, 245, 227, 228, 230, 232, 234, 235, 238, 240, ~
## $ StoreID        <dbl> 1, 1, 1, 1, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 1, 2, 2~
## $ PriceCH        <dbl> 1.75, 1.75, 1.86, 1.69, 1.69, 1.69, 1.69, 1.75, 1.75, 1~
## $ PriceMM        <dbl> 1.99, 1.99, 2.09, 1.69, 1.69, 1.99, 1.99, 1.99, 1.99, 1~
## $ DiscCH         <dbl> 0.00, 0.00, 0.17, 0.00, 0.00, 0.00, 0.00, 0.00, 0.00, 0~
## $ DiscMM         <dbl> 0.00, 0.30, 0.00, 0.00, 0.00, 0.00, 0.40, 0.40, 0.40, 0~
## $ SpecialCH      <dbl> 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
## $ SpecialMM      <dbl> 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0~
## $ LoyalCH        <dbl> 0.500000, 0.600000, 0.680000, 0.400000, 0.956535, 0.965~
## $ SalePriceMM    <dbl> 1.99, 1.69, 2.09, 1.69, 1.69, 1.99, 1.59, 1.59, 1.59, 1~
## $ SalePriceCH    <dbl> 1.75, 1.75, 1.69, 1.69, 1.69, 1.69, 1.69, 1.75, 1.75, 1~
## $ PriceDiff      <dbl> 0.24, -0.06, 0.40, 0.00, 0.00, 0.30, -0.10, -0.16, -0.1~
## $ Store7         <fct> No, No, No, No, Yes, Yes, Yes, Yes, Yes, Yes, Yes, Yes,~
## $ PctDiscMM      <dbl> 0.000000, 0.150754, 0.000000, 0.000000, 0.000000, 0.000~
## $ PctDiscCH      <dbl> 0.000000, 0.000000, 0.091398, 0.000000, 0.000000, 0.000~
## $ ListPriceDiff  <dbl> 0.24, 0.24, 0.23, 0.00, 0.00, 0.30, 0.30, 0.24, 0.24, 0~
## $ STORE          <dbl> 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 2~
```

Q 8.4.9 a

Create a training set containing a random sample of 800 observations, and a test set containing the remaining observations.

Answer 8.4.9 (a)

```
require(ISLR)

set.seed(5)

train=sample(1:nrow(OJ),800)

train.OJ=OJ[train,]
test.OJ=OJ[-train,]
```

Q 8.4.9 b

Fit a tree to the training data, with **Purchase** as the response and the other variables as predictors. Use the **summary()** function to produce summary statistics about the tree, and describe the results obtained. What is the training error rate? How many terminal nodes does the tree have?

Answer 8.4.9 (b)

```
tree.OJ <- tree(Purchase ~ .,data=train.OJ)

summary(tree.OJ)
```

```
##
## Classification tree:
## tree(formula = Purchase ~ ., data = train.OJ)
## Variables actually used in tree construction:
## [1] "LoyalCH"       "PriceDiff"     "ListPriceDiff"
## Number of terminal nodes:  9
## Residual mean deviance:  0.7347 = 581.1 / 791
## Misclassification error rate: 0.1662 = 133 / 800
```

The classification tree has 9 terminal nodes and a training error rate of 16.62%.

Although there are 17 predictors in the dataset, only 3 were used in splits. These were:

LoyalCH , PriceDiff , ListPriceDiff

Q 8.4.9 c

Type in the name of the tree object in order to get a detailed text output. Pick one of the terminal nodes, and interpret the information displayed.

Answer 8.4.9 (c)

```
tree.OJ
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##  1) root 800 1068.00 CH ( 0.61250 0.38750 )
##    2) LoyalCH < 0.5036 346  412.40 MM ( 0.28324 0.71676 )
##      4) LoyalCH < 0.280875 164  125.50 MM ( 0.12805 0.87195 )
##        8) LoyalCH < 0.0356415 56   10.03 MM ( 0.01786 0.98214 ) *
##        9) LoyalCH > 0.0356415 108  103.50 MM ( 0.18519 0.81481 ) *
##      5) LoyalCH > 0.280875 182  248.00 MM ( 0.42308 0.57692 )
##       10) PriceDiff < 0.05 71   67.60 MM ( 0.18310 0.81690 ) *
##       11) PriceDiff > 0.05 111  151.30 CH ( 0.57658 0.42342 ) *
##    3) LoyalCH > 0.5036 454  362.00 CH ( 0.86344 0.13656 )
##      6) PriceDiff < -0.39 31   40.32 MM ( 0.35484 0.64516 )
##       12) LoyalCH < 0.638841 10    0.00 MM ( 0.00000 1.00000 ) *
##       13) LoyalCH > 0.638841 21   29.06 CH ( 0.52381 0.47619 ) *
##      7) PriceDiff > -0.39 423  273.70 CH ( 0.90071 0.09929 )
##       14) LoyalCH < 0.705326 135  143.00 CH ( 0.77778 0.22222 )
##         28) ListPriceDiff < 0.255 67   89.49 CH ( 0.61194 0.38806 ) *
##         29) ListPriceDiff > 0.255 68   30.43 CH ( 0.94118 0.05882 ) *
##       15) LoyalCH > 0.705326 288   99.77 CH ( 0.95833 0.04167 ) *
```

Choosing node 11), which is a terminal node as it is marked by a *:

First the root node: 1) root 800 1068.00 CH (0.61250 0.38750)

This means that, at the root node, there are 800 observations, the deviance is 1068.00, the overall prediction is CH and the split is 61.25% CH vs 38.75% MM.

We can see that, from the root node, three splits take place to produce the terminal node labelled by 11):

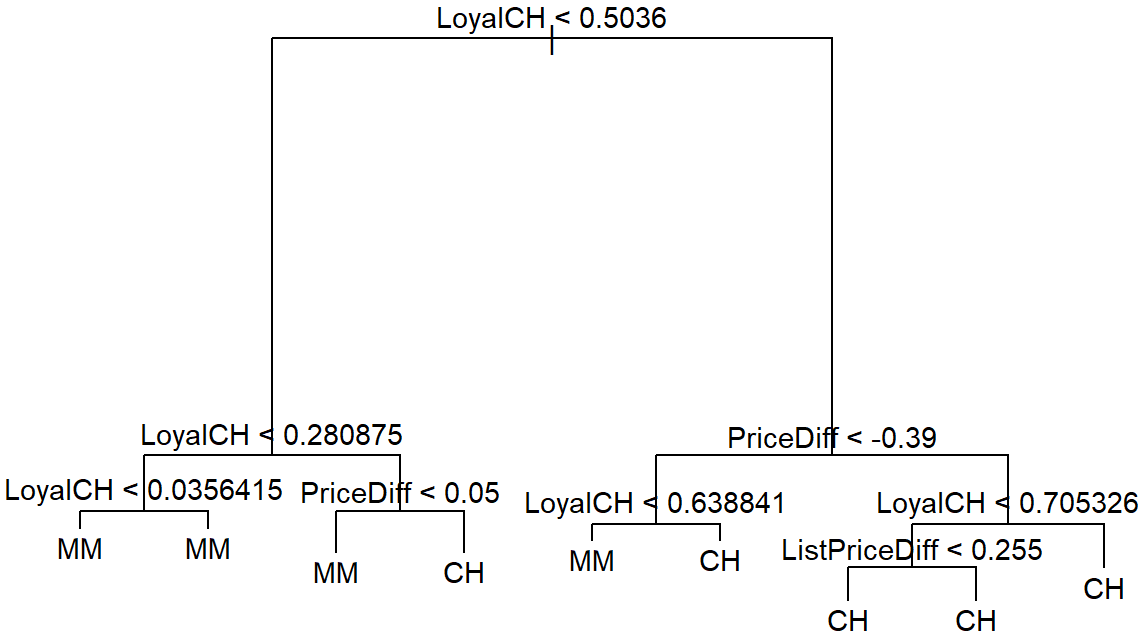
A split at LoyalCH = 0.5036 A split at LoyalCH = 0.280875 164 A split at LoyalCH = 0.280875 182

Q 8.4.9 d

Create a plot of the tree, and interpret the results.

Answer 8.4.9 (d)

```
plot(tree.OJ)
text(tree.OJ, cex = 0.9)
```



So, looking at the diagram , we can make some quick observations.

1. The left branch shows that when there is a relative low customer loyalty for Citrus Hill then the Minute Maid is chosen. Otherwise, the cheaper drink will be the one purchased.
2. Similarly, in the right sub tree this model predicts that when there is a strong preference for Citrus Hill it will always be the drink chosen.
3. When the List Price Diff is lower, then the customer will purchase Citrus Hill

Q 8.4.9 e

Predict the response on the test data, and produce a confusion matrix comparing the test labels to the predicted test labels. What is the test error rate?

Answer 8.4.9 (e)

The confusion matrix with test data

```
test.pred.OJ <- predict(tree.OJ, test.OJ, type = "class")
table(test.pred.OJ, test_actual = test.OJ$Purchase)
```

```
##             test_actual
## test.pred.OJ  CH  MM
##           CH 148  32
##           MM  15  75
```

The test error rate is given below.

```
1 - mean(test.pred.OJ == test.OJ$Purchase)
```

```
## [1] 0.1740741
```

Q 8.4.9 f

Apply the **cv.tree()** function to the training set in order to determine the optimal tree size.

Answer 8.4.9 (f)

Now, we want to get low test error, so, I specify FUN = prune.misclass. So, this indicates that we want the classification error rate to help us with the cross-validation and pruning process, instead of using the default for the cv.tree() function, meaning deviance.

```
set.seed(2)

cv.tree.OJ <- cv.tree(tree.OJ, K = 10, FUN = prune.misclass)
cv.tree.OJ
```

```
## $size
## [1] 9 6 5 3 2 1
##
## $dev
## [1] 149 149 149 173 172 310
##
## $k
## [1]  -Inf   0.0   1.0   8.5   9.0 150.0
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

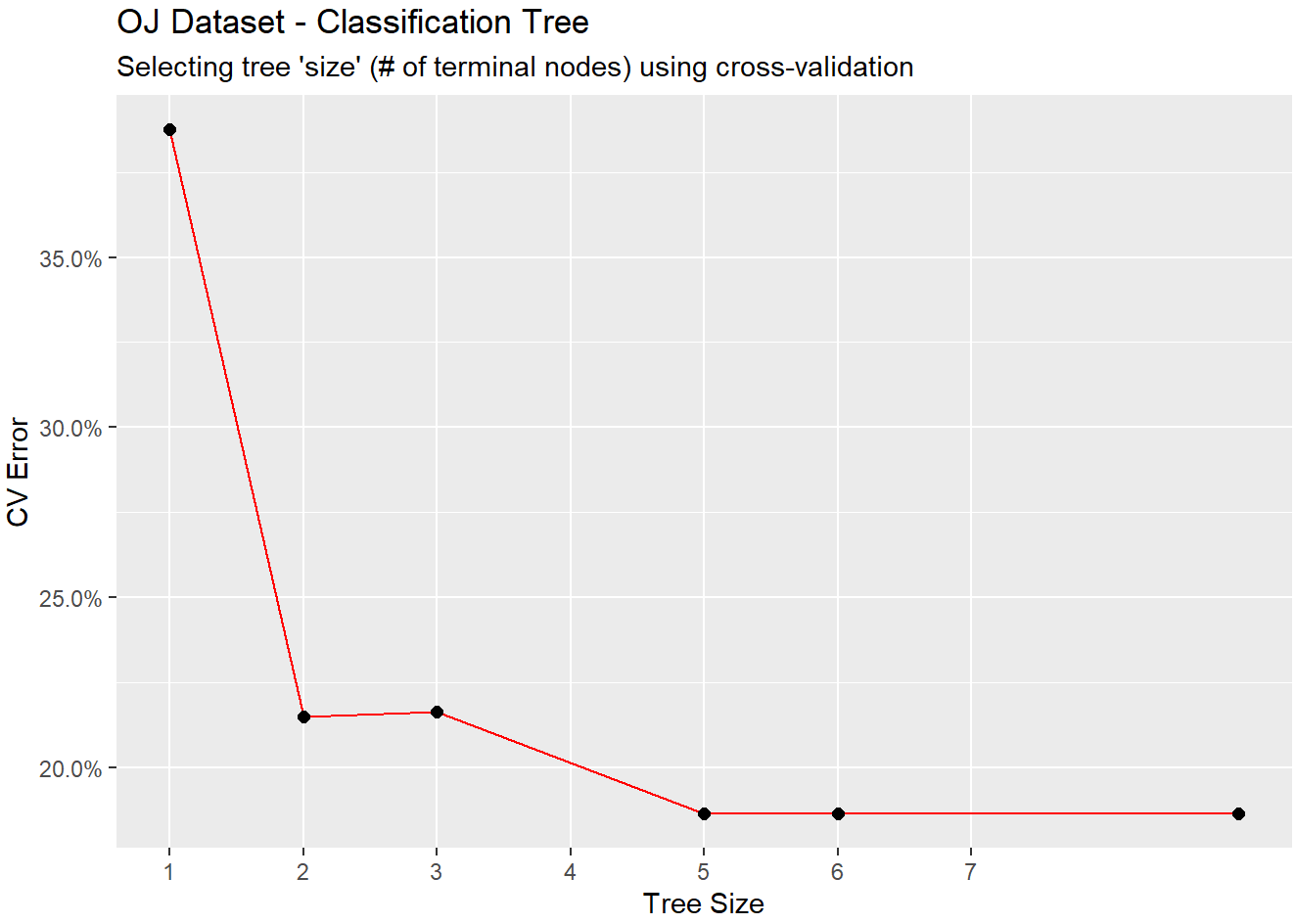
Q 8.4.9 g

Produce a plot with tree size on the x-axis and cross-validated classification error rate on the y-axis.

Answer 8.4.9 (g)

The plot is below. Note that size is on the x-axis and CV_Error is on the Y-axis. Size (X-axis) is the number of terminal nodes (so 1 means it is just the root node with no splits), and CV_Error (Y-Axis) gives the total number of errors made from the out-of-fold predictions during cross-validation (only because we specified FUN = prune.misclass - omitting this would mean this reports the deviance). From this we can obtain the cross-validation error rate.

```
#plot(cv.tree.OJ)

data.frame(size = cv.tree.OJ$size, CV_Error = cv.tree.OJ$dev / nrow(train.OJ)) %>%
ggplot(aes(x = size, y = CV_Error)) +
  geom_line(col = "red") +
  geom_point(size = 2) +
  scale_x_continuous(breaks = seq(1, 7), minor_breaks = NULL) +
  scale_y_continuous(labels = scales::percent_format()) +
  scale_color_manual(values = c("deepskyblue3", "green")) +
  theme(legend.position = "top") +
  labs(title = "OJ Dataset - Classification Tree",
       subtitle = "Selecting tree 'size' (# of terminal nodes) using cross-validation",
       x = "Tree Size",
       y = "CV Error")
```



Q 8.4.9 h

Which tree size corresponds to the lowest cross-validated classification error rate?

Answer 8.4.9 (h)

Of the sequence of trees generated, trees of sizes 5 and 6 have the same cross-validation error. It makes sense to select the more parsimonious model here with 5 terminal nodes.

Q 8.4.9 i

Produce a pruned tree corresponding to the optimal tree size obtained using cross-validation. If cross-validation does not lead to selection of a pruned tree, then create a pruned tree with five terminal nodes.

Answer 8.4.9 (i)

So, I found out earlier that the tree with 5 and 6 terminal nodes have the same cross-validation error. Now, here below I produce the pruned tree with 5 terminal nodes.

```
pruned.OJ <- prune.tree(tree.OJ, best = 5)
pruned.OJ
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##  1) root 800 1068.00 CH ( 0.61250 0.38750 )
##    2) LoyalCH < 0.5036 346  412.40 MM ( 0.28324 0.71676 )
##      4) LoyalCH < 0.280875 164  125.50 MM ( 0.12805 0.87195 ) *
##      5) LoyalCH > 0.280875 182  248.00 MM ( 0.42308 0.57692 ) *
##    3) LoyalCH > 0.5036 454  362.00 CH ( 0.86344 0.13656 )
##      6) PriceDiff < -0.39 31   40.32 MM ( 0.35484 0.64516 ) *
##      7) PriceDiff > -0.39 423  273.70 CH ( 0.90071 0.09929 )
##       14) LoyalCH < 0.705326 135  143.00 CH ( 0.77778 0.22222 ) *
##       15) LoyalCH > 0.705326 288   99.77 CH ( 0.95833 0.04167 ) *
```

Q 8.4.9 j

Compare the training error rates between the pruned and unpruned trees. Which is higher?

Answer 8.4.9 (j)

Below is the training error for the unpruned tree

```
mean(predict(tree.OJ, type = "class") != train.OJ$Purchase)
```

```
## [1] 0.16625
```

Below is the training error for the pruned tree

```
mean(predict(pruned.OJ, type = "class") != train.OJ$Purchase)
```

```
## [1] 0.18875
```

What we see here is that the training error for the pruned tree is higher. This is what is expected as the training error of a tree monotonically decreases as its flexibility (number of splits) increases.

Q 8.4.9 k

Compare the test error rates between the pruned and unpruned trees. Which is higher?

Answer 8.4.9 (k)

Below is the test error for the unpruned tree

```
mean(predict(tree.OJ, type = "class", data = test.OJ) != test.OJ$Purchase)
```

```
## [1] 0.48
```

Below is the test error for the pruned tree

```
mean(predict(pruned.OJ, type = "class", data = test.OJ) != test.OJ$Purchase)
```

```
## [1] 0.5025
```

So, here also the test error is higher for the pruned tree.