Stat 652 Homework

Sagar Kalauni

2023-11-07

2. This problem involves the OJ data set which is part of the ISLR2 package. The data set contains sales information for Citrus Hill and Minute Maid orange juice. You may see the detail description of the data using ?OJ in R.

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

```
library(ISLR2)  #Loading the ISLR2 library in the R working environment

## Warning: package 'ISLR2' was built under R version 4.3.2

?OJ  # getting familier with the OJ (Orange Juice Data)

## starting httpd help server ... done

dim(OJ)

## [1] 1070   18
```

So there are 1070 observations and 18 variables

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

```
set.seed(12312)
train=sample(1:nrow(OJ), 800)  # we take 800 data for training set
test=OJ[-train,]
```

Checking for the column names in our data set

```
colnames(OJ)

##  [1] "Purchase"       "WeekofPurchase" "StoreID"        "PriceCH"
##  [5] "PriceMM"        "DiscCH"         "DiscMM"         "SpecialCH"
##  [9] "SpecialMM"      "LoyalCH"        "SalePriceMM"    "SalePriceCH"
## [13] "PriceDiff"      "Store7"         "PctDiscMM"      "PctDiscCH"
## [17] "ListPriceDiff"  "STORE"
```

(1) Fit a tree to the training data, with Purchase as the label and the other variables except as features. 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?

```
set.seed(12312)
library(tree)
```

```
## Warning: package 'tree' was built under R version 4.3.2
```

```
tree.d=tree(Purchase~., OJ, split = 'gini', subset =train ) # except Purchase all other variables in the data set are be considered as predictors.
```

Looking at the summary statistics of this tree.

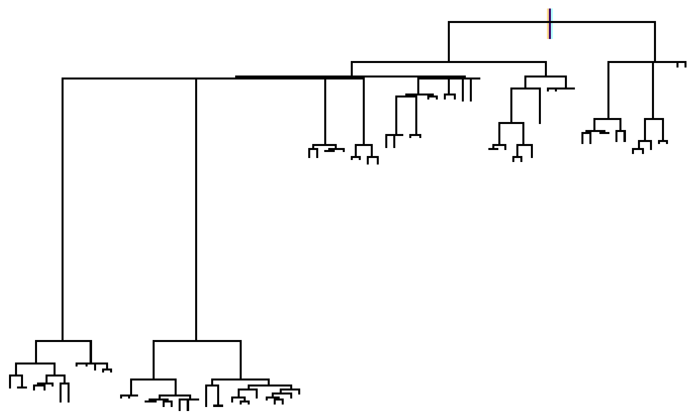
```
summary(tree.d)

##
## Classification tree:
## tree(formula = Purchase ~ ., data = OJ, subset = train, split = "gini")
## Variables actually used in tree construction:
##  [1] "SpecialMM"     "SpecialCH"     "DiscCH"        "DiscMM"
##  [5] "LoyalCH"       "STORE"         "PriceDiff"     "PriceCH"
##  [9] "StoreID"       "PriceMM"       "WeekofPurchase" "SalePriceMM"
## [13] "PctDiscMM"     "ListPriceDiff"
## Number of terminal nodes:  80
## Residual mean deviance:  0.629 = 452.9 / 720
## Misclassification error rate: 0.15 = 120 / 800
```

Interpretation: This is a classification tree, we have a total number of terminal node of 80, so it's a big tree. we have mean deviance: 0.629 , which is calculated deviance divided by total number of training observation minus the number of terminal nodes. We also have Misclassification error rate: 0.15, which is calculated as Number of Misclassification divided by total training set. (from video)

we see that the training error rate is 15%. The residual mean deviance reported is simply the deviance divided by $n - |T_0|$, which in this case is 800-80= 720.

(2) Create a plot of the tree. Pick one of the terminal nodes, and interpret the information displayed.

```
plot(tree.d)  # for Plotting the decision tree
```



```
#text(tree.d, pretty= 0) #if you want to see labels also
```

To interpert the tree, lets look tree in deitals again

```
set.seed(12312)
tree.d

## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 800 1061.000 CH ( 0.62250 0.37750 )
##     2) SpecialMM < 0.5 681  873.700 CH ( 0.65932 0.34068 )
##       4) SpecialCH < 0.5 566  742.200 CH ( 0.63604 0.36396 )
##         8) DiscCH < 0.05 473  624.400 CH ( 0.62791 0.37209 )
##          16) DiscMM < 0.03 381  503.200 CH ( 0.62730 0.37270 )
##            32) LoyalCH < 0.461965 137  141.400 MM ( 0.21168 0.78832 )
##              64) LoyalCH < 0.275811 92   67.350 MM ( 0.11957 0.88043 )
##               128) STORE < 1.5 18   22.910 MM ( 0.33333 0.66667 )
##                 256) LoyalCH < 0.134076 7    0.000 MM ( 0.00000 1.00000 )
*
##                 257) LoyalCH > 0.134076 11   15.160 CH ( 0.54545 0.45455
)
##                   514) PriceDiff < 0.255 5    6.730 CH ( 0.60000 0.40000
) *
##                   515) PriceDiff > 0.255 6    8.318 CH ( 0.50000 0.50000
) *
##               129) STORE > 1.5 74   36.600 MM ( 0.06757 0.93243 )
```

```
##                   258) PriceCH < 1.94 49    9.763 MM ( 0.02041 0.97959 )
##                     516) LoyalCH < 0.0657865 17    7.606 MM ( 0.05882
0.94118 )
##                      1032) LoyalCH < 0.0200955 12    0.000 MM ( 0.00000
1.00000 ) *
##                      1033) LoyalCH > 0.0200955 5    5.004 MM ( 0.20000
0.80000 ) *
##                     517) LoyalCH > 0.0657865 32    0.000 MM ( 0.00000
1.00000 ) *
##                   259) PriceCH > 1.94 25   21.980 MM ( 0.16000 0.84000 )
##                     518) LoyalCH < 0.0714805 18    0.000 MM ( 0.00000
1.00000 ) *
##                     519) LoyalCH > 0.0714805 7    9.561 CH ( 0.57143
0.42857 ) *
##                65) LoyalCH > 0.275811 45   60.570 MM ( 0.40000 0.60000 )
##                 130) StoreID < 1.5 16   21.170 MM ( 0.37500 0.62500 )
##                   260) PriceMM < 2.04 9    9.535 MM ( 0.22222 0.77778 ) *
##                   261) PriceMM > 2.04 7    9.561 CH ( 0.57143 0.42857 ) *
##                 131) StoreID > 1.5 29   39.340 MM ( 0.41379 0.58621 )
##                   262) PriceCH < 1.825 11   10.430 MM ( 0.18182 0.81818 ) *
##                   263) PriceCH > 1.825 18   24.730 CH ( 0.55556 0.44444 )
##                     526) PriceCH < 1.875 9   12.370 MM ( 0.44444 0.55556 )
*
##                     527) PriceCH > 1.875 9   11.460 CH ( 0.66667 0.33333 )
*
##              33) LoyalCH > 0.461965 244  197.000 CH ( 0.86066 0.13934 )
##                66) LoyalCH < 0.610074 74   91.720 CH ( 0.68919 0.31081 )
##                 132) PriceDiff < 0.235 22   29.770 MM ( 0.40909 0.59091 )
##                   264) StoreID < 2.5 14   19.120 MM ( 0.42857 0.57143 )
##                     528) PriceCH < 1.775 8   10.590 MM ( 0.37500 0.62500 )
*
##                     529) PriceCH > 1.775 6    8.318 CH ( 0.50000 0.50000 )
*
##                   265) StoreID > 2.5 8   10.590 MM ( 0.37500 0.62500 ) *
##                 133) PriceDiff > 0.235 52   50.910 CH ( 0.80769 0.19231 )
##                   266) WeekofPurchase < 249.5 25   29.650 CH ( 0.72000
0.28000 )
##                     532) PriceDiff < 0.27 14   18.250 CH ( 0.64286 0.35714
)
##                      1064) LoyalCH < 0.51 7    8.376 CH ( 0.71429 0.28571 )
*
##                      1065) LoyalCH > 0.51 7    9.561 CH ( 0.57143 0.42857 )
*
##                     533) PriceDiff > 0.27 11   10.430 CH ( 0.81818 0.18182
)
##                      1066) LoyalCH < 0.5136 6    7.638 CH ( 0.66667 0.33333
) *
##                      1067) LoyalCH > 0.5136 5    0.000 CH ( 1.00000 0.00000
) *
##                   267) WeekofPurchase > 249.5 27   18.840 CH ( 0.88889
```

```
0.11111 )
##                      534) PriceCH < 1.925 21   13.210 CH ( 0.90476 0.09524 )
##                       1068) STORE < 1.5 15    0.000 CH ( 1.00000 0.00000 ) *
##                       1069) STORE > 1.5 6    7.638 CH ( 0.66667 0.33333 ) *
##                      535) PriceCH > 1.925 6    5.407 CH ( 0.83333 0.16667 )
*
##                 67) LoyalCH > 0.610074 170   81.510 CH ( 0.93529 0.06471 )
##                  134) LoyalCH < 0.701955 32   27.740 CH ( 0.84375 0.15625 )
##                    268) LoyalCH < 0.67808 21    0.000 CH ( 1.00000 0.00000 )
*
##                    269) LoyalCH > 0.67808 11   15.160 CH ( 0.54545 0.45455 )
##                      538) StoreID < 2.5 6    8.318 MM ( 0.50000 0.50000 ) *
##                      539) StoreID > 2.5 5    6.730 CH ( 0.60000 0.40000 ) *
##                  135) LoyalCH > 0.701955 138   49.360 CH ( 0.95652 0.04348 )
##                    270) LoyalCH < 0.927095 89   19.140 CH ( 0.97753 0.02247
)
##                      540) LoyalCH < 0.799296 31   14.830 CH ( 0.93548
0.06452 )
##                       1080) PriceDiff < 0.285 15    0.000 CH ( 1.00000
0.00000 ) *
##                       1081) PriceDiff > 0.285 16   12.060 CH ( 0.87500
0.12500 )
##                         2162) LoyalCH < 0.735293 6    0.000 CH ( 1.00000
0.00000 ) *
##                         2163) LoyalCH > 0.735293 10   10.010 CH ( 0.80000
0.20000 ) *
##                      541) LoyalCH > 0.799296 58    0.000 CH ( 1.00000
0.00000 ) *
##                    271) LoyalCH > 0.927095 49   27.710 CH ( 0.91837 0.08163
)
##                      542) PriceMM < 2.205 41   15.980 CH ( 0.95122 0.04878 )
##                       1084) WeekofPurchase < 266 25   13.940 CH ( 0.92000
0.08000 )
##                         2168) LoyalCH < 0.950865 9    0.000 CH ( 1.00000
0.00000 ) *
##                         2169) LoyalCH > 0.950865 16   12.060 CH ( 0.87500
0.12500 )
##                           4338) STORE < 2.5 10   10.010 CH ( 0.80000 0.20000
) *
##                           4339) STORE > 2.5 6    0.000 CH ( 1.00000 0.00000
) *
##                       1085) WeekofPurchase > 266 16    0.000 CH ( 1.00000
0.00000 ) *
##                      543) PriceMM > 2.205 8    8.997 CH ( 0.75000 0.25000 )
*
##             17) DiscMM > 0.03 92  121.200 CH ( 0.63043 0.36957 )
##               34) LoyalCH < 0.528155 37   41.050 MM ( 0.24324 0.75676 )
##                 68) STORE < 0.5 20   16.910 MM ( 0.15000 0.85000 )
##                  136) WeekofPurchase < 237.5 9   11.460 MM ( 0.33333 0.66667
) *
```

```
##                 137) WeekofPurchase > 237.5 11    0.000 MM ( 0.00000 
1.00000 ) *
##                69) STORE > 0.5 17   22.070 MM ( 0.35294 0.64706 )  
##                 138) PriceMM < 2.135 12   13.500 MM ( 0.25000 0.75000 )  
##                   276) WeekofPurchase < 272.5 7    8.376 MM ( 0.28571 
0.71429 ) *
##                   277) WeekofPurchase > 272.5 5    5.004 MM ( 0.20000 
0.80000 ) *
##                 139) PriceMM > 2.135 5    6.730 CH ( 0.60000 0.40000 ) *
##              35) LoyalCH > 0.528155 55   37.910 CH ( 0.89091 0.10909 )  
##                70) DiscMM < 0.22 17   20.600 CH ( 0.70588 0.29412 )  
##                 140) SalePriceMM < 2.005 9    9.535 CH ( 0.77778 0.22222 ) 
*
##                 141) SalePriceMM > 2.005 8   10.590 CH ( 0.62500 0.37500 ) 
*
##                71) DiscMM > 0.22 38    9.249 CH ( 0.97368 0.02632 )  
##                 142) LoyalCH < 0.664147 6    5.407 CH ( 0.83333 0.16667 ) *
##                 143) LoyalCH > 0.664147 32    0.000 CH ( 1.00000 0.00000 ) 
*
##           9) DiscCH > 0.05 93  117.000 CH ( 0.67742 0.32258 )  
##            18) DiscMM < 0.2 84  106.900 CH ( 0.66667 0.33333 )  
##              36) PriceMM < 2.11 68   87.020 CH ( 0.66176 0.33824 )  
##                72) DiscCH < 0.115 50   68.590 CH ( 0.56000 0.44000 )  
##                 144) PriceDiff < 0.265 40   55.350 CH ( 0.52500 0.47500 )  
##                   288) LoyalCH < 0.727631 23   24.080 MM ( 0.21739 0.78261 
)  
##                     576) StoreID < 3.5 17   15.840 MM ( 0.17647 0.82353 )  
##                      1152) WeekofPurchase < 268.5 11    0.000 MM ( 0.00000 
1.00000 ) *
##                      1153) WeekofPurchase > 268.5 6    8.318 CH ( 0.50000 
0.50000 ) *
##                     577) StoreID > 3.5 6    7.638 MM ( 0.33333 0.66667 ) *
##                   289) LoyalCH > 0.727631 17    7.606 CH ( 0.94118 0.05882 
)  
##                     578) LoyalCH < 0.938594 9    0.000 CH ( 1.00000 0.00000 
) *
##                     579) LoyalCH > 0.938594 8    6.028 CH ( 0.87500 0.12500 
) *
##                 145) PriceDiff > 0.265 10   12.220 CH ( 0.70000 0.30000 )  
##                   290) WeekofPurchase < 252.5 5    6.730 CH ( 0.60000 
0.40000 ) *
##                   291) WeekofPurchase > 252.5 5    5.004 CH ( 0.80000 
0.20000 ) *
##                73) DiscCH > 0.115 18    7.724 CH ( 0.94444 0.05556 )  
##                 146) LoyalCH < 0.645047 6    5.407 CH ( 0.83333 0.16667 ) *
##                 147) LoyalCH > 0.645047 12    0.000 CH ( 1.00000 0.00000 ) 
*
##              37) PriceMM > 2.11 16   19.870 CH ( 0.68750 0.31250 )  
##                74) LoyalCH < 0.48323 6    5.407 MM ( 0.16667 0.83333 ) *
##                75) LoyalCH > 0.48323 10    0.000 CH ( 1.00000 0.00000 ) *
```

```
##           19) DiscMM > 0.2 9    9.535 CH ( 0.77778 0.22222 ) *
##        5) SpecialCH > 0.5 115  122.900 CH ( 0.77391 0.22609 )
##         10) STORE < 0.5 93   85.390 CH ( 0.82796 0.17204 )
##           20) WeekofPurchase < 274.5 85   57.430 CH ( 0.89412 0.10588 )
##             40) LoyalCH < 0.51 20   25.900 CH ( 0.65000 0.35000 )
##               80) SalePriceMM < 1.86 13   17.940 CH ( 0.53846 0.46154 )
##                160) PriceCH < 1.805 8   11.090 CH ( 0.50000 0.50000 ) *
##                161) PriceCH > 1.805 5    6.730 CH ( 0.60000 0.40000 ) *
##               81) SalePriceMM > 1.86 7    5.742 CH ( 0.85714 0.14286 ) *
##             41) LoyalCH > 0.51 65   17.860 CH ( 0.96923 0.03077 )
##               82) WeekofPurchase < 249 11   10.430 CH ( 0.81818 0.18182 )
##                164) LoyalCH < 0.705326 6    7.638 CH ( 0.66667 0.33333 ) *
##                165) LoyalCH > 0.705326 5    0.000 CH ( 1.00000 0.00000 ) *
##               83) WeekofPurchase > 249 54    0.000 CH ( 1.00000 0.00000 )
*
##           21) WeekofPurchase > 274.5 8    6.028 MM ( 0.12500 0.87500 ) *
##         11) STORE > 0.5 22   30.320 CH ( 0.54545 0.45455 )
##           22) SalePriceMM < 1.84 16   22.180 MM ( 0.50000 0.50000 )
##             44) DiscCH < 0.2 11   15.160 CH ( 0.54545 0.45455 )
##               88) LoyalCH < 0.4176 5    6.730 MM ( 0.40000 0.60000 ) *
##               89) LoyalCH > 0.4176 6    7.638 CH ( 0.66667 0.33333 ) *
##             45) DiscCH > 0.2 5    6.730 MM ( 0.40000 0.60000 ) *
##           23) SalePriceMM > 1.84 6    7.638 CH ( 0.66667 0.33333 ) *
##      3) SpecialMM > 0.5 119  161.200 MM ( 0.41176 0.58824 )
##        6) DiscCH < 0.08 108  146.000 MM ( 0.40741 0.59259 )
##         12) LoyalCH < 0.5324 63   58.350 MM ( 0.17460 0.82540 )
##           24) WeekofPurchase < 260.5 29   35.920 MM ( 0.31034 0.68966 )
##             48) StoreID < 1.5 14   14.550 MM ( 0.21429 0.78571 )
##               96) LoyalCH < 0.27904 6    8.318 MM ( 0.50000 0.50000 ) *
##               97) LoyalCH > 0.27904 8    0.000 MM ( 0.00000 1.00000 ) *
##             49) StoreID > 1.5 15   20.190 MM ( 0.40000 0.60000 )
##               98) PriceMM < 1.89 8   10.590 MM ( 0.37500 0.62500 ) *
##               99) PriceMM > 1.89 7    9.561 MM ( 0.42857 0.57143 ) *
##           25) WeekofPurchase > 260.5 34   15.210 MM ( 0.05882 0.94118 )
##             50) SalePriceMM < 2.155 26    0.000 MM ( 0.00000 1.00000 ) *
##             51) SalePriceMM > 2.155 8    8.997 MM ( 0.25000 0.75000 ) *
##         13) LoyalCH > 0.5324 45   52.190 CH ( 0.73333 0.26667 )
##           26) PctDiscMM < 0.192246 31   19.710 CH ( 0.90323 0.09677 )
##             52) SalePriceMM < 1.785 15   15.010 CH ( 0.80000 0.20000 )
##              104) WeekofPurchase < 240.5 10    6.502 CH ( 0.90000 0.10000
) *
##              105) WeekofPurchase > 240.5 5    6.730 CH ( 0.60000 0.40000 )
*
##             53) SalePriceMM > 1.785 16    0.000 CH ( 1.00000 0.00000 ) *
##           27) PctDiscMM > 0.192246 14   18.250 MM ( 0.35714 0.64286 )
##             54) ListPriceDiff < 0.195 8    8.997 MM ( 0.25000 0.75000 ) *
##             55) ListPriceDiff > 0.195 6    8.318 CH ( 0.50000 0.50000 ) *
##        7) DiscCH > 0.08 11   15.160 MM ( 0.45455 0.54545 )
##         14) WeekofPurchase < 259.5 5    5.004 MM ( 0.20000 0.80000 ) *
##         15) WeekofPurchase > 259.5 6    7.638 CH ( 0.66667 0.33333 ) *
```

Interpertation: For interpertaion purpose I took the terminal node at the 256 position in the tree(internal node), Clearly it is a terminal node because it has * sign with it and its information are as follows: for this split cretrion is LoyalCH < 0.134076, n value is 7 with no deviance (i.e 0.000), yvalue: MM and yprob in (0.00000 1.00000).

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

```
set.seed(12312)
pred.d=predict(tree.d, test, type="class")
pred.d  # Looking at the predicted lables

##   [1] MM CH CH MM CH CH MM CH CH CH CH MM CH CH CH CH CH CH CH CH CH CH CH
CH CH
##  [26] CH CH CH CH CH CH CH CH CH CH CH CH CH CH CH CH CH CH MM MM CH CH CH
CH CH
##  [51] CH CH CH CH CH CH CH CH CH CH CH CH CH MM CH CH CH MM CH CH MM MM MM
MM MM
##  [76] CH CH MM MM MM CH CH MM MM MM CH CH CH MM CH MM CH CH CH MM CH MM MM
CH MM
## [101] MM MM MM CH MM MM MM CH MM MM MM MM MM CH CH CH MM CH CH MM MM CH CH
MM CH
## [126] CH CH CH MM CH MM MM CH CH CH CH CH MM MM MM MM MM MM MM CH MM CH CH
CH CH
## [151] CH CH MM CH CH CH CH CH CH CH CH CH CH CH CH CH MM CH CH CH CH MM MM
MM MM
## [176] MM MM MM MM CH MM MM MM MM MM CH MM MM CH CH CH MM CH CH CH MM CH MM
MM CH
## [201] MM MM CH CH CH CH CH CH CH MM MM MM MM CH CH CH CH CH CH CH MM CH CH
MM CH
## [226] CH CH CH CH CH CH MM CH MM MM MM MM MM MM CH MM CH MM CH CH CH MM CH
MM CH
## [251] MM CH MM MM CH CH CH CH CH CH CH CH CH CH CH CH MM CH CH CH
## Levels: CH MM
```

Creating a confusion matrix for comparing the test labels to the predicted test labels

```
set.seed(12312)
table(pred.d, test$Purchase)

##
## pred.d  CH  MM
##     CH 133  42
##     MM  22  73
```

Interpertation: From the confusion matrix, we can see that the True-CH value is 133 and True-MM value is 73. False-CH value is 42 and False-MM value is 22. Misclassification rate= (42+22)/270. This is the misclassification rate in my test set so the test error rate is (42+22)/270 = 0.237037. so my test error rate is 23.37% and my training error rate was 15%, which makes sense also that my test error rate> training error rate.

Also accuracy in the test data: (133+73)/270 =0.762963 i.e 76.29%

(note:- if you re-run the predict() function then you might get slightly different results, due to 'ties', by book)

(4) Apply the cv.tree() function to the training set in order to determine the optimal tree size. Produce a plot with tree size on the x-axis and cross-validated classification error rate on the y-axis. Which tree size corresponds to the lowest cross-validated classification error rate?

```
#set.seed(12312)
#cv.d=cv.tree(tree.d)   # using deviance as a criteria for the cross-validation, right now not asked

#cv.d
#plot(cv.d$size, cv.d$dev, type = "b") # Since we have used deviance as our criteria for the cross-validation, we will use the same for plotting also, not asked
```

we are going to look at tree with lowest possible deviance with small size because we perfer a tree which is less complex and produce a minimum deviance.{not asked}

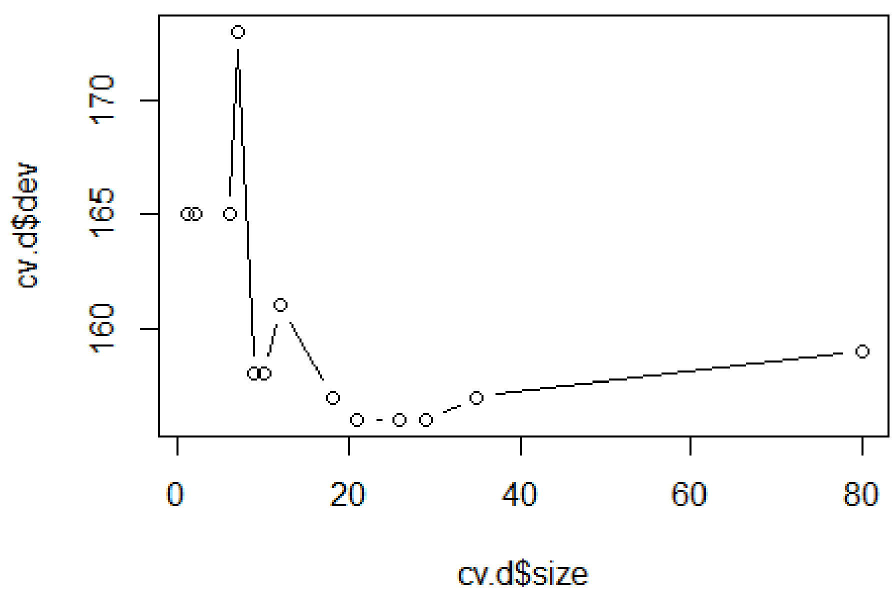
Asked one:-let's also look for the plot when cross-validation is done on the basis of misclassification

```
set.seed(12312)
cv.d=cv.tree(tree.d, FUN= prune.misclass)
names(cv.d)

## [1] "size"   "dev"    "k"      "method"

set.seed(12312)
cv.d

## $size
##  [1] 80 35 29 26 21 18 12 10  9  7  6  2  1
##
## $dev
##  [1] 159 157 156 156 156 157 161 158 158 173 165 165 165
##
## $k
##  [1]       -Inf  0.0000000  0.5000000  0.6666667  0.8000000  2.0000000
##  [7]  2.8333333  3.0000000  4.0000000 10.5000000 19.0000000 19.7500000
## [13] 21.0000000
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

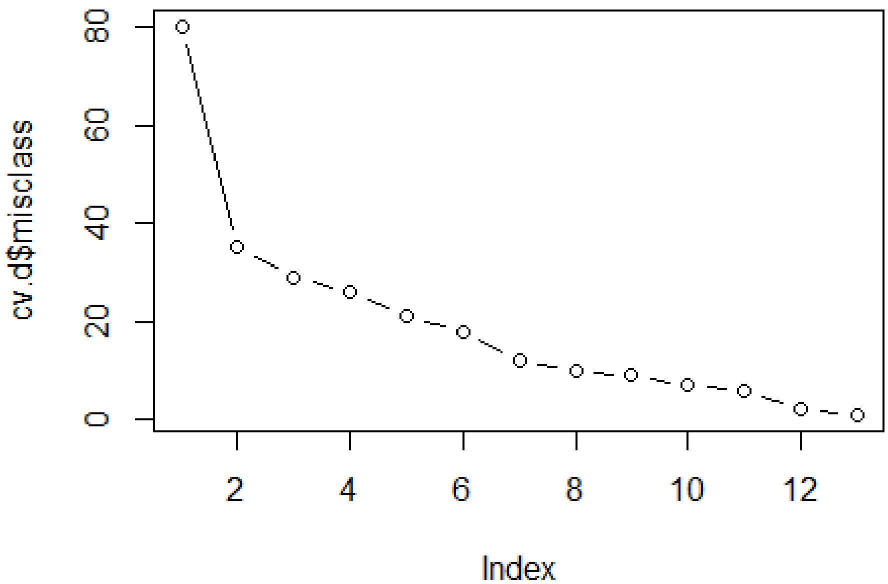
Clearly from above result we can see that the tree with either: 29,26 or 21 terminal nodes results in only 156 cross-validation error (which is minimum one) and same for all given three nodes.

let's visualize this

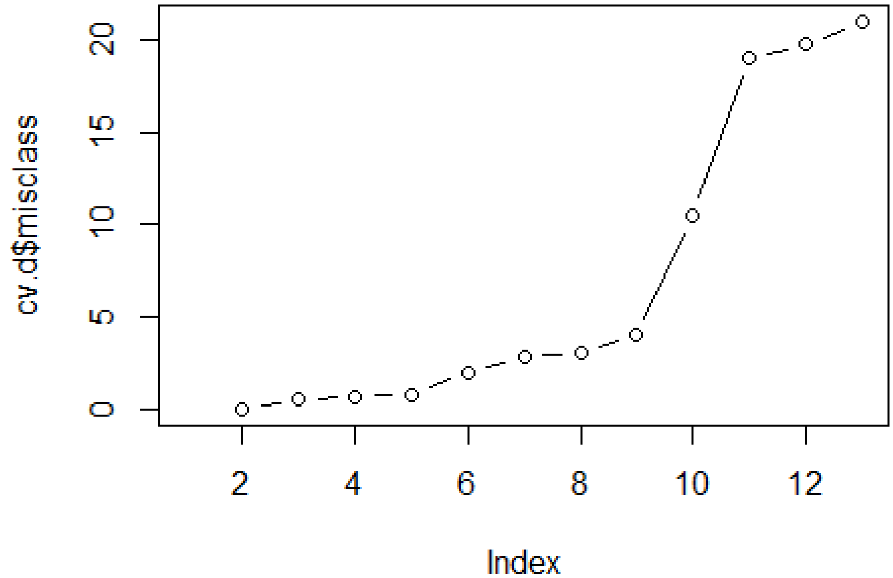
```
plot(cv.d$size, cv.d$dev, type = "b")
```




```
set.seed(12312)
plot(cv.d$size, cv.d$misclass, type = "b")
```




```
#Also not asked
plot(cv.d$k, cv.d$misclass, type = "b")
```



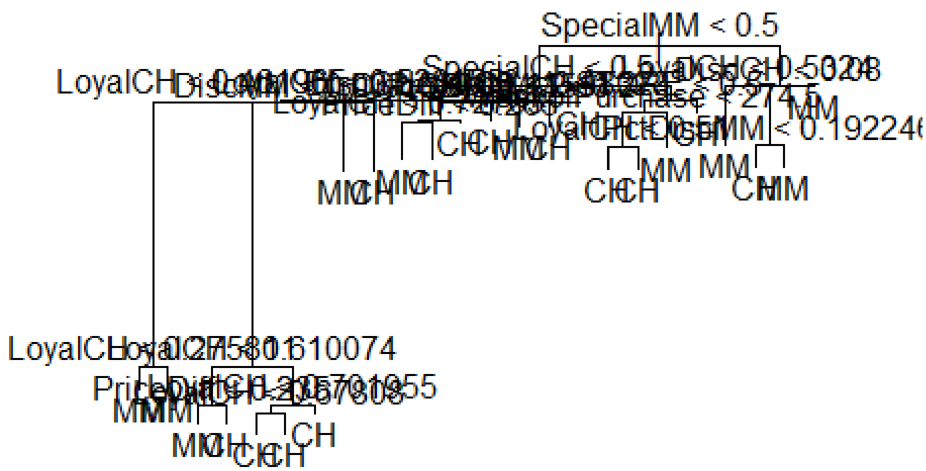
(5) Produce a pruned tree corresponding to the optimal tree size obtained using cross-validation. If cross-validation does not lead to selection of a pruned tree, then create a pruned tree with five terminal nodes.

ANSWER: choosing the smallest one

```
set.seed(12312)
prune.d=prune.tree(tree.d, best =21)
```

Now we can take a look at this smaller tree

```
#set.seed(12312)
#summary(prune.d)

plot(prune.d)
text(prune.d)
```



(5) Compare the training and test error rates between the pruned and unpruned trees. Which is higher?

ANSWER: For Training error:

```
set.seed(12312)
summary(prune.d)

##
## Classification tree:
## snip.tree(tree = tree.d, nodes = c(269L, 132L, 145L, 11L, 27L,
## 289L, 65L, 40L, 73L, 7L, 133L, 135L, 34L, 288L, 26L, 12L, 41L,
## 35L, 64L))
## Variables actually used in tree construction:
##  [1] "SpecialMM"      "SpecialCH"      "DiscCH"         "DiscMM"
##  [5] "LoyalCH"        "PriceDiff"      "PriceMM"        "STORE"
##  [9] "WeekofPurchase" "PctDiscMM"
## Number of terminal nodes:  24
## Residual mean deviance:  0.7864 = 610.2 / 776
## Misclassification error rate: 0.1638 = 131 / 800
```

From the summary statistics we can see that the Misclassification error rate(i.e Training error rate): 0.1638 (or 16.38%). After the pruneing the misclassification of our training data went up a litle, perviously it was 15% and now it is 16.38% (Increased)

```
set.seed(12312)
#Predict class on test data
```

```r
pred.d.prune=predict(prune.d,test, type = "class")
pred.d.prune
```

```
##   [1] MM MM CH MM CH CH MM CH CH CH CH MM CH CH CH CH CH CH CH CH CH CH CH
CH CH
##  [26] CH CH CH CH CH MM MM CH CH CH CH CH CH CH CH CH CH CH MM MM CH CH CH
CH CH
##  [51] CH CH CH CH CH CH CH CH CH CH CH CH MM MM CH CH CH MM CH CH MM MM MM
MM MM
##  [76] MM CH MM MM MM MM MM MM MM MM MM CH CH MM MM MM CH CH MM MM CH MM MM
CH CH
## [101] MM MM MM MM MM MM MM CH MM MM MM MM MM CH MM CH MM CH MM MM MM CH CH
MM CH
## [126] CH CH CH CH CH MM MM CH CH CH CH CH MM MM MM CH MM MM MM MM MM CH CH
CH CH
## [151] CH CH MM CH CH CH MM CH CH CH CH CH CH CH CH CH MM CH CH CH CH MM MM
MM MM
## [176] MM MM MM MM CH MM MM MM MM MM CH MM MM MM CH CH MM CH MM CH MM CH MM
MM CH
## [201] MM MM MM CH CH CH MM CH CH MM MM MM MM CH CH CH CH CH CH CH MM CH CH
MM CH
## [226] CH CH CH MM MM CH MM CH MM MM MM MM MM MM CH MM MM MM CH CH CH MM MM
MM CH
## [251] MM MM MM MM MM CH CH CH CH CH MM CH CH CH CH CH MM CH CH CH
## Levels: CH MM
```

```r
set.seed(12312)
table(pred.d.prune, test$Purchase)
```

```
##
## pred.d.prune  CH  MM
##           CH 123  29
##           MM  32  86
```

Interpretation: From the confusion matrix, we can see that the True-CH value is 123 and True-MM value is 86. False-CH value is 29 and False-MM value is 32. Misclassification rate= (29+32)/270. This is the misclassification rate in my test set so the test error rate is (29+32)/270 = 0.2259259. so my test error rate is 22.59% for the pruned tree. Also accuracy in the test data: (133+86)/270 = 0.8111111 i.e 81.11%

Talking about the compression, test error rate for the unpruned tree was 23.37% and test error rate for the pruned data is 22.59%, so kind a say It performs little well in the test data after pruning, which makes sense.

Taking about accuracy point of view: Unpruned tree has a accuracy of 76.29% in the test day but pruned tree has accuracy of 81.11%, so accuracy increases by some percentage in the test data after pruning.

(7) Perform random forest on the training set with 1,000 trees for a chosen values of the ”mtry”. You may experiment with a range of values of the parameter.

```r
set.seed(12312)
#install.packages("randomForest")
library(randomForest)

## Warning: package 'randomForest' was built under R version 4.3.2

## randomForest 4.7-1.1

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

```r
set.seed(12312)
# Let first choose the value of m to be sqrt(17) i.e nearly 4 for this
randomforest in classification problem
rf.OJ=randomForest(Purchase~., data=OJ, subset= train, mtry=4, ntree=1000,
importance=TRUE)
rf.OJ  # lets take a look at the output

##
## Call:
##  randomForest(formula = Purchase ~ ., data = OJ, mtry = 4, ntree = 1000,
importance = TRUE, subset = train)
##                Type of random forest: classification
##                      Number of trees: 1000
## No. of variables tried at each split: 4
##
##         OOB estimate of  error rate: 19.25%
## Confusion matrix:
##     CH  MM class.error
## CH 433  65   0.1305221
## MM  89 213   0.2947020
```

Its a classification problem and number of variable we tried at each split is 4. we have out-of-bag (OBB) error rate of 19.25%. We can also see the confusion matrix and class errors from the above output.

Now, I am just trying different values of m’s

```r
set.seed(12312)
# Trying m=6
rf.OJ=randomForest(Purchase~., data=OJ, subset= train, mtry=6, ntree=1000,
importance=TRUE)
rf.OJ  # lets take a look at the output

##
## Call:
##  randomForest(formula = Purchase ~ ., data = OJ, mtry = 6, ntree = 1000,
importance = TRUE, subset = train)
##                Type of random forest: classification
##                      Number of trees: 1000
## No. of variables tried at each split: 6
##
##         OOB estimate of  error rate: 20.5%
```

```
## Confusion matrix:
##     CH  MM class.error
## CH 428  70   0.1405622
## MM  94 208   0.3112583
```

Its a classification problem and number of variable we tried at each split is 6. we have out-of-bag (OBB) error rate of 20.5%. We can also see the confusion matrix and class errors from the above output.

```
set.seed(12312)
# Trying m=8
rf.OJ=randomForest(Purchase~., data=OJ, subset= train, mtry=8, ntree=1000,
importance=TRUE)
rf.OJ  # lets take a look at the output

##
## Call:
##  randomForest(formula = Purchase ~ ., data = OJ, mtry = 8, ntree = 1000,
importance = TRUE, subset = train)
##                Type of random forest: classification
##                      Number of trees: 1000
## No. of variables tried at each split: 8
##
##         OOB estimate of  error rate: 21.12%
## Confusion matrix:
##     CH  MM class.error
## CH 423  75   0.1506024
## MM  94 208   0.3112583
```

Its a classification problem and number of variable we tried at each split is 8. we have out-of-bag (OBB) error rate of 21.12%. We can also see the confusion matrix and class errors from the above output.

```
set.seed(12312)
# Trying m=10
rf.OJ=randomForest(Purchase~., data=OJ, subset= train, mtry=10, ntree=1000,
importance=TRUE)
rf.OJ  # lets take a look at the output

##
## Call:
##  randomForest(formula = Purchase ~ ., data = OJ, mtry = 10, ntree = 1000,
importance = TRUE, subset = train)
##                Type of random forest: classification
##                      Number of trees: 1000
## No. of variables tried at each split: 10
##
##         OOB estimate of  error rate: 21%
## Confusion matrix:
##     CH  MM class.error
```

```
## CH 423  75   0.1506024
## MM  93 209   0.3079470
```

Its a classification problem and number of variable we tried at each split is 10. we have out-of-bag (OBB) error rate of 21%. We can also see the confusion matrix and class errors from the above output.

```r
set.seed(12312)
# Trying m=12
rf.OJ=randomForest(Purchase~., data=OJ, subset= train, mtry=12, ntree=1000,
importance=TRUE)
rf.OJ  # lets take a look at the output

##
## Call:
##  randomForest(formula = Purchase ~ ., data = OJ, mtry = 12, ntree = 1000,
importance = TRUE, subset = train)
##                Type of random forest: classification
##                      Number of trees: 1000
## No. of variables tried at each split: 12
##
##         OOB estimate of  error rate: 21.38%
## Confusion matrix:
##     CH  MM class.error
## CH 420  78   0.1566265
## MM  93 209   0.3079470
```

Its a classification problem and number of variable we tried at each split is 12. we have out-of-bag (OBB) error rate of 21.38%. We can also see the confusion matrix and class errors from the above output.

```r
set.seed(12312)
# Trying m=14
rf.OJ=randomForest(Purchase~., data=OJ, subset= train, mtry=14, ntree=1000,
importance=TRUE)
rf.OJ  # lets take a look at the output

##
## Call:
##  randomForest(formula = Purchase ~ ., data = OJ, mtry = 14, ntree = 1000,
importance = TRUE, subset = train)
##                Type of random forest: classification
##                      Number of trees: 1000
## No. of variables tried at each split: 14
##
##         OOB estimate of  error rate: 21.62%
## Confusion matrix:
##     CH  MM class.error
## CH 415  83   0.1666667
## MM  90 212   0.2980132
```

Its a classification problem and number of variable we tried at each split is 14. we have out-of-bag (OBB) error rate of 21.62%. We can also see the confusion matrix and class errors from the above output.

RUN THIS CODE TOO

```
set.seed(12312)
# Trying m=16
rf.OJ=randomForest(Purchase~., data=OJ, subset= train, mtry=16, ntree=1000,
importance=TRUE)
rf.OJ  # lets take a look at the output

##
## Call:
##  randomForest(formula = Purchase ~ ., data = OJ, mtry = 16, ntree = 1000,
importance = TRUE, subset = train)
##                Type of random forest: classification
##                      Number of trees: 1000
## No. of variables tried at each split: 16
##
##         OOB estimate of  error rate: 21.12%
## Confusion matrix:
##     CH  MM class.error
## CH 417  81   0.1626506
## MM  88 214   0.2913907
```

Its a classification problem and number of variable we tried at each split is 16. we have out-of-bag (OBB) error rate of 21.12%. We can also see the confusion matrix and class errors from the above output.

In addition to these, I can also try 3,5,7…15 for my m value and check the output. I will not try m=17, because that will be Bagging not random Forest.

(8) Which variables appear to be the most important predictors in the RF model?

```
# before running this code please run the last code for m=16 one, I used that
one
set.seed(12312)
importance(rf.OJ)

##                        CH          MM MeanDecreaseAccuracy
MeanDecreaseGini
## WeekofPurchase  16.7265589   5.716393            18.182423
35.830714
## StoreID          8.8235077  14.387907            17.126076
11.949016
## PriceCH          8.3515041   7.000315            11.771776
4.813653
## PriceMM         10.2335085   1.683032            10.402566
4.363846
## DiscCH           0.4142774   4.964000             3.972429
2.211028
```

```
## DiscMM            6.3855097   8.519203            11.062247
2.846845
## SpecialCH         6.8743503   6.362092             9.567356
5.315484
## SpecialMM        -3.2757012  -1.131864            -3.057787
2.255528
## LoyalCH         112.0372951 140.746028           170.239545
224.484132
## SalePriceMM       7.6439509  11.235186            15.267883
11.203916
## SalePriceCH       8.4809831   3.893956             9.582246
5.535824
## PriceDiff        20.4486482  23.701773            32.436204
26.759881
## Store7           -0.4505884   4.593527             3.255883
1.193827
## PctDiscMM         8.2158469   8.806128            13.175335
3.583283
## PctDiscCH         0.2849907   4.721174             4.056611
2.663962
## ListPriceDiff    23.7850874   7.787215            25.745453
16.402801
## STORE             7.8631788  16.179724            18.839898
9.273292
```

From above output we can clearly see that the most important variable in predicting the Purchase is LoyalCH (i.e Customer brand Loyalty for CH)

(9) Use the RF model to predict the response on the test data. Form a confusion matrix. How does this compare with the result obtained using a single tree?

```
set.seed(12312)
yhat.rf= predict(rf.OJ, newdata = test)   # random forest with m=16 one, last
one
yhat.rf  # looking at them

##    4    7   11   13   14   16   19   20   24   25   27   33   34   36   42
45
##   MM   CH   CH   CH   CH   CH   MM   CH   CH   CH   CH   MM   MM   CH   CH
CH
##   47   50   54   62   67   70   73   76   77   80   86   88   90   94   96
97
##   CH   CH   CH   CH   CH   MM   CH   CH   CH   CH   CH   CH   CH   MM   CH
MM
##  100  102  106  108  118  119  126  127  131  132  134  137  148  157  159
160
##   CH   CH   CH   CH   CH   CH   CH   CH   CH   CH   CH   CH   MM   CH   CH
CH
##  162  172  173  178  182  184  187  199  203  214  216  217  218  220  228
231
##   CH   CH   CH   CH   CH   CH   CH   CH   CH   CH   CH   CH   CH   CH   CH
```

```
MM
##  242  245  247  262  272  273  274  275  280  281  283  294  296  300  301 
302 
##   CH   CH   CH   MM   MM   CH   MM   MM   MM   MM   MM   MM   MM   MM   MM 
MM 
##  304  305  307  308  310  313  314  320  322  327  330  341  346  351  357 
358 
##   MM   CH   MM   MM   MM   MM   CH   CH   MM   CH   CH   CH   CH   CH   MM 
CH 
##  360  362  364  366  375  384  386  402  406  410  411  413  418  419  420 
435 
##   MM   CH   MM   CH   MM   MM   MM   CH   MM   CH   MM   CH   MM   MM   MM 
CH 
##  437  441  450  452  453  455  459  472  473  478  480  501  504  509  513 
516 
##   MM   CH   CH   CH   MM   MM   MM   MM   MM   CH   CH   MM   CH   CH   CH 
CH 
##  519  521  523  526  529  532  536  537  540  549  556  558  559  566  571 
573 
##   MM   MM   MM   MM   CH   CH   CH   CH   CH   MM   MM   MM   CH   MM   MM 
MM 
##  575  578  579  583  587  596  597  609  613  621  624  627  630  631  632 
634 
##   MM   CH   CH   CH   CH   CH   CH   CH   CH   CH   CH   CH   MM   CH   CH 
CH 
##  635  636  643  654  656  657  667  670  671  673  674  677  678  688  690 
691 
##   CH   CH   CH   CH   CH   CH   MM   CH   MM   CH   MM   CH   MM   MM   MM 
MM 
##  699  700  702  705  708  711  712  717  726  727  732  735  739  744  745 
747 
##   MM   MM   MM   MM   MM   MM   MM   MM   MM   MM   MM   MM   MM   MM   CH 
MM 
##  751  757  758  775  777  785  787  789  794  797  801  807  808  815  821 
823 
##   MM   MM   CH   MM   MM   MM   MM   CH   MM   MM   CH   CH   CH   CH   CH 
CH 
##  825  832  841  847  848  849  851  858  859  865  866  870  872  875  878 
886 
##   CH   MM   MM   MM   MM   CH   CH   CH   CH   MM   CH   CH   MM   CH   CH 
MM 
##  887  891  892  894  905  916  922  929  934  938  952  954  955  956  959 
965 
##   CH   CH   CH   CH   CH   CH   CH   MM   MM   MM   MM   MM   MM   MM   MM 
MM 
##  969  976  979  984  986  990  992  995  996  999 1005 1006 1008 1009 1012 
1016 
##   MM   MM   MM   CH   CH   MM   MM   MM   MM   CH   MM   MM   MM   MM   CH 
CH 
## 1018 1023 1030 1033 1035 1042 1045 1050 1051 1053 1056 1061 1062 1067
```

```
##    CH    CH    CH    CH    CH    CH    CH    CH    CH    CH    MM    MM    MM    CH
## Levels: CH MM
```

```
set.seed(12312)
test.error=sum(yhat.rf!=test$Purchase)/270 # 270 is the total number of test data in my test set
test.error
```

```
## [1] 0.1851852
```

So the error rate for my test data is 0.1851852 (i.e 18.51%)

```
set.seed(12312)
# Creating a confusion matrix
table(yhat.rf, truth=test$Purchase)
```

```
##        truth
## yhat.rf  CH  MM
##      CH 129  24
##      MM  26  91
```

From the confusion matrix, we can see that the True-CH value is 129 and True-MM value is 91. False-CH value is 24 and False-MM value is 26. Misclassification rate= (24+26)/270. This is the misclassification rate in my test set so the test error rate is (24+26)/270 = 0.1851852. so my test error rate is 18.51%.

Comparison between single tree and random forest

1) First thing we can clearly see that our model does better in case of random forest as compared to single tree. The test error rate for single tree was 23.37% (for unpruned) and 22.59 for pruned, but for random forest test error rate reduce to 18.51% only.

2) talking about accuracy, single tree (unpruned) has the accuracy of 76.29% but the accuracy for the random forest become (129+91)/270= 81.48%

So as expected, random forest predict the variable more accuractly then single tree, which makes sense also.

---

1. Consider the Boston housing data set, from the ISLR2 library.

```
set.seed(12312)
library(ISLR2)
head(Boston)
```

```
##      crim zn indus chas   nox    rm  age    dis rad tax ptratio lstat medv
## 1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3  4.98 24.0
## 2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8  9.14 21.6
## 3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8  4.03 34.7
## 4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7  2.94 33.4
## 5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7  5.33 36.2
## 6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7  5.21 28.7
```

(a) Based on this data set, provide an estimate for the population mean of "medv". Call this estimate $\hat{\mu}$.

```
set.seed(12312)
mean50=vector(length=1000)
for(i in 1:1000){
  samp = sample(Boston$medv, size = 50)
  mean50[i] = mean(samp)
}
#mean50
mu_hat=mean(mean50)
mu_hat

## [1] 22.56684

# my output is 22.56684

# just checking how close it is
mean(Boston$medv)

## [1] 22.53281

# Actual value was 22.53281
```

(b) Provide an estimate of the standard error of $\hat{\mu}$. Recall, we can compute the standard error of the sample mean by dividing the sample standard deviation by the square root of the number of observations.

```
set.seed(12312)
#Estimation for the standard deviation
est_stand_error= sd(Boston$medv)/sqrt(nrow(Boston))
est_stand_error

## [1] 0.4088611
```

(c) Now estimate the standard error of $\hat{\mu}$ using the bootstrap. How does this compare to your answer from (b)? ANSWER:

```
set.seed(12312)
#I need to instal and load the boot in the working environment before start
using it
#install.packages("boot")
library(boot)

## Warning: package 'boot' was built under R version 4.3.2

# first let's create a function that I can use inside the boot() function
which calculate my desired statistics mean for the booted sample

mu_boot <- function(data, indices) {
 mean(data[indices])
}
# bootstrapping with 100 replications
boot_res_1000 <- boot(data=Boston$medv, statistic=mu_boot,
```

```
    R=1000)
boot_res_1000
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Boston$medv, statistic = mu_boot, R = 1000)
##
##
## Bootstrap Statistics :
##     original      bias    std. error
## t1* 22.53281 0.01785296    0.404425
```

Interpretation:-

Standard error in my part b was 0.4088611 but the standard error by bootstrap sampling statistics is 0.404425 for the replication length of 1000. So they are close to each other .

```
set.seed(12312)
# bootstrapping with 100 replications
boot_res_500 <- boot(data=Boston$medv, statistic=mu_boot,
    R=500)
boot_res_500
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Boston$medv, statistic = mu_boot, R = 500)
##
##
## Bootstrap Statistics :
##     original      bias    std. error
## t1* 22.53281 0.02741028   0.3973759
```

This is showing I need to increase the number of replication to match the standard error in part b.

(d) Based on your bootstrap estimate from (c), provide a 95 % normal confidence interval for the mean of ”medv”. Compare it to the results obtained using t.test(Boston$medv).

```
set.seed(12312)
# First let's check the given one
t.test(Boston$medv)
```

```
##
##  One Sample t-test
##
```

```
## data:  Boston$medv
## t = 55.111, df = 505, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  21.72953 23.33608
## sample estimates:
## mean of x
##  22.53281
```

So I found a 95% confidence interval (21.72953, 23.33608)

```
set.seed(12312)
# Now let's find bootstrap confidence interval
# Since my above boot() output has only one index, so it will be by default
the one of our interest
# as she say, I need to use normal by question
# since by default is always 95% so I will not write anything
# Point to be noted, I have calculated the confidence interval Based on 1000
bootstrap replicates
boot.ci(boot_res_1000, type = "norm")
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 1000 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = boot_res_1000, type = "norm")
##
## Intervals :
## Level      Normal
## 95%   (21.72, 23.31 )
## Calculations and Intervals on Original Scale
```

So I found a 95% normal confidence interval (21.72, 23.31)

Interpretation: They are almost close to each other, this may be because I have used high number of replication in bootstrap. with lower replication length you might get some difference but not big I guess.

(e) Use sample median to estimate $\hat{m}$ for the median value of medv in the population.

```
set.seed(12312)
# Question is little unclear for the direction
# our sample median is
median(Boston$medv)
```

```
## [1] 21.2
```

```
#
median50=vector(length=1000)
for(i in 1:1000){
  samp = sample(Boston$medv, size = 50)
  mean50[i] = median(samp)
```

```
}
estimated_median=median(mean50)
estimated_median

## [1] 21.2

# This is if you want this way, I think boot is best to do these stuffs

boot_med <- function(data, indices) {
 median(data[indices])
}

est_boot.med=boot(data = Boston$medv, statistic = boot_med, R=1000)
est_boot.med

##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Boston$medv, statistic = boot_med, R = 1000)
##
##
## Bootstrap Statistics :
##     original   bias    std. error
## t1*     21.2 -0.0082   0.3779426
```

(f) We now would like to estimate the standard error of $\hat{m}$. Unfortunately, there is no simple formula for computing the standard error of the median. Instead, estimate the standard error of the median using the bootstrap.

```
boot_med <- function(data, indices) {
 median(data[indices])
}

est_boot.med=boot(data = Boston$medv, statistic = boot_med, R=1000)
est_boot.med

##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Boston$medv, statistic = boot_med, R = 1000)
##
##
## Bootstrap Statistics :
##     original    bias    std. error
## t1*     21.2 -0.01505   0.3730149
```

So the required standard error of sample median is 0.3789714

————————-THE END——————————————————