1. Hand code the KNN algorithm and apply your algorithm to the iris data in R. Submit your outputs from R along with your R code.

a. Write a function "distance(u,v)" that will compute the Euclidean distance of two numeric vectors $\mathbf{u} = (u_1, ..., u_n)$ and $\mathbf{v} = (v_1, ..., v_n)$, given by $d(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{i=1}^{n}(u_i - v_i)^2}$. Try this function on the vectors $(0, 0, 0, 1)$ and $(2, 5, 2, 4)$ to check that your function result is correct.

b. Write a function "neighbors(data, t, k)" that will return the k closest neighbors of a vector t from a set of vectors. Use the Euclidean distance function in question 1 to determine the distance of vector t and each vector in the data set.

c. Write a function "KNN(train, t, k)" that returns the predicted class of a vector t base on the training data. You will call the neighbors() function to get the set of k nearest neighbors. You will then predict the class of t as the majority class of these neighbors.

d. Apply your function to the iris data, which can be attached using:
library(datasets); data(iris).
Randomly pick 120 observations as your training data and the rest 30 observations as your test set.
You can use the following code in R to split the data:
Take $k = 3$, run your KNN algorithm on each of the test data and compute the test error rate.

2. Perform a classification analysis on the heart disease data (provided on Blackboard). The ultimate goal is to find a predictive model to predict the cardiovascular disease status. The data contains the following variables:

Age: age of the patient [years]
Sex: sex of the patient [M: Male, F: Female]
ChestPainType: chest pain type [TA: Typical Angina, ATA: Atypical Angina, NAP: Non-Anginal Pain, ASY: Asymptomatic]
RestingBP: resting blood pressure [mm Hg]
Cholesterol: serum cholesterol [mm/dl]
FastingBS: fasting blood sugar [1: if FastingBS > 120 mg/dl, 0: otherwise]
RestingECG: resting electrocardiogram results [Normal: Normal, ST: having ST-T wave abnormality, LVH: showing left ventricular hypertrophy]
MaxHR: maximum heart rate achieved [Numeric value between 60 and 202]
ExerciseAngina: exercise-induced angina [Y: Yes, N: No]
Oldpeak: oldpeak = ST [Numeric value measured in depression]
ST_Slope: the slope of the peak exercise ST segment [Up: upsloping, Flat: flat, Down: downsloping]
HeartDisease: output class [1: heart disease, 0: Normal]

a. Randomly split the data into 70% training and 30% testing. And do the following exploration. Fit a logistic regression mode using the training data to model P(HeartDisease= 1| All predictors) and answer the following questions.

(i) Report the test summary of each individual coefficients. Do any of the predictors appear to be statistically significant?
(ii) Interpret the coefficients of "Cholesterol" and "Sex" in context of the problem.
(iii) Use the logistic regression model to predict for the testing data. Compute the confusion matrix and overall fraction of correct predictions. Comment on what the confusion matrix is telling you about the types of mistakes made.

b. Now use LDA, QDA and KNN to fit a model using the training data.

(i) Fit a LDA model using the training set. Compute the confusion matrix and the overall fraction of correct predictions for the test data.
(ii) Fit a QDA model using the training set. Compute the confusion matrix and the overall fraction of correct predictions for the test data.
(iii) Fit KNN model with only numerical predictors. Experiment with values for K. With your choice of K, compute the confusion matrix and the overall fraction of correct predictions for the test data.

c. Which of these methods appears to provide the best results on this data? Discuss.

d. Experiment with different combinations of predictors for each of the methods. Report the variables, method, and associated confusion matrix that appears to provide the best results on the test data.

STAT 562 HW 1

2023-10-21

1a

```r
distance <- function(u,v)
{
  u <- as.numeric(u)
  v <- as.numeric(v)
  return(sqrt(sum((u-v)^2)))
}

distance(c(0,0,0,1), c(2,5,2,4))
```

```
## [1] 6.480741
```

distance(c(0,0,0,1), c(2,5,2,4)) gives $\sqrt{42}$ by hand and by using the distance function.

1b

```r
neighbors <- function(data, t, k) {
  n <- nrow(data)

  distances <- c()
  for (i in 1:n) {
    distances[i] <- distance(as.numeric(data[i, 1:4]), t[1:4])
  }
  smallest_k_distances <- sort(distances,index.return = TRUE)$ix[1:k]

  return(data[smallest_k_distances,])
}
```

1c

```r
KNN <- function(train, t, k)
{
  NNs <- neighbors(train, t, k)

  species_vector <- c("setosa", "versicolor", "virginica")

  n = length(species_vector)

  num_per_species = c(0,0,0)

  for(i in 1:n)
```

```
  {
    for(j in 1:k)
    {
      if(NNs[j, 5] == species_vector[i])
      {
        num_per_species[i] = num_per_species[i] + 1
      }
    }
  }

  return(species_vector[which.max(num_per_species)])
}
```

1d

```
set.seed(123)
library(datasets); data(iris)
smp_size <- floor(0.8 * nrow(iris))
train_ind <- sample(seq_len(nrow(iris)), size = smp_size)

train <- iris[train_ind, ]
test <- iris[-train_ind, ]

result_vec <- c()
for(i in 1:30)
{
  result_vec <- append(result_vec, KNN(train, test[i,], 3))
}

error_vec <- result_vec == test[, 5]
mean(error_vec == 1)
```

```
## [1] 0.9666667
```

We find an error rate of $(1 - 96.666\%) = 3.333\%$ when $k = 3$. This of course depends on the training/testing split. Without setting the seed, the percentage of correct identifications seems to range from .9 to 1. Not significantly different from one another.

2a

```
set.seed(1234)
heart_data = read.csv("/Users/jacksonhuff/Desktop/heart.csv")
attach(heart_data)
smp_size <- floor(0.7 * nrow(heart_data))
train_ind <- sample(seq_len(nrow(heart_data)), size = smp_size)

train <- heart_data[train_ind, ]
test <- heart_data[-train_ind, ]

model <- glm(formula = HeartDisease ~ ., family = binomial, data = heart_data)

summary(model)
```

```
##
## Call:
## glm(formula = HeartDisease ~ ., family = binomial, data = heart_data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.6531  -0.3747   0.1745   0.4457   2.5778
##
## Coefficients:
##                    Estimate Std. Error z value Pr(>|z|)
## (Intercept)       -1.163656   1.416003  -0.822 0.411197
## Age                0.016550   0.013197   1.254 0.209803
## SexM               1.466477   0.279834   5.241 1.60e-07 ***
## ChestPainTypeATA -1.830289   0.326293  -5.609 2.03e-08 ***
## ChestPainTypeNAP -1.685682   0.266001  -6.337 2.34e-10 ***
## ChestPainTypeTA  -1.488392   0.432572  -3.441 0.000580 ***
## RestingBP          0.004194   0.006010   0.698 0.485296
## Cholesterol       -0.004115   0.001087  -3.785 0.000154 ***
## FastingBS          1.136482   0.274999   4.133 3.59e-05 ***
## RestingECGNormal -0.177033   0.271925  -0.651 0.515022
## RestingECGST     -0.268546   0.350020  -0.767 0.442945
## MaxHR             -0.004288   0.005023  -0.854 0.393249
## ExerciseAnginaY   0.900292   0.244513   3.682 0.000231 ***
## Oldpeak            0.380643   0.118466   3.213 0.001313 **
## ST_SlopeFlat       1.453902   0.429086   3.388 0.000703 ***
## ST_SlopeUp        -0.994101   0.450196  -2.208 0.027234 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1262.14  on 917  degrees of freedom
## Residual deviance:  594.19  on 902  degrees of freedom
## AIC: 626.19
##
## Number of Fisher Scoring iterations: 6
```

```r
predictions <- predict(model, newdata = test, type = "response")

true_false <- predictions > .5
true_false[true_false == TRUE] = 1
true_false[true_false == FALSE] = 0

conf_matrix <- table(reference = test[,length(test)], prediction = true_false)
conf_matrix
```

```
##          prediction
## reference   0   1
##         0 104  19
##         1  14 139
```

2a i

The above output provides the estimate, standard error, and the test statistic for significance (in this case, a t-statistic) and its associated p-value. Testing each coefficient at significance $\alpha = .05$ yields that Sex, ChestPainTypeATA, ChestPainTypeNAP, ChestPainTypeTA, Cholesterol, FastingBS, ExerciseAnginaY, Oldpeak, ST_SlopeFlat, ST_SlopeUp are all statistically significant.

2a ii

First for cholesterol. For a one unit increase in cholesterol level, the odds of having heart disease decrease by $100 * (1 - e^{-0.004115})\% \approx .4\%$ with all other input levels held fixed.

Now for SexM. Since SexM is a binary variable, we can say that the odds of having heart disease are about $e^{1.466477} \approx 4.33$ times higher for males compared to females, with all other input levels held fixed.

2a iii

We find that the overall fraction of correct predictions (accuracy) is $\frac{104+139}{104+139+19+14} = 0.880$. That is, our model correctly predicted the presence of heart disease in a patient about 88% of the time. We find that our model had 19 false positives and 14 false negatives. Our model does not seem to prefer to predict false positives or false negatives. This is good, our model does not seem to be exhibiting bias in either direction. We find the precision to be $\frac{139}{139+19} = .880$ and the recall to be $\frac{139}{139+14} = .908$. Both are recall and our precision are quite high.

#2b

```r
lda.out = lda(HeartDisease ~ ., data = train)
lda.out
```

```
## Call:
## lda(HeartDisease ~ ., data = train)
##
## Prior probabilities of groups:
##         0         1
## 0.4470405 0.5529595
##
## Group means:
##        Age      SexM ChestPainTypeATA ChestPainTypeNAP ChestPainTypeTA
## 0 50.78049 0.6376307       0.36585366        0.3310105      0.05574913
## 1 55.99437 0.8957746       0.04788732        0.1436620      0.03943662
##   RestingBP Cholesterol FastingBS RestingECGNormal RestingECGST    MaxHR
## 0  130.6516    227.6725 0.1289199        0.6236934    0.1672474 147.1847
## 1  134.8592    178.7042 0.3521127        0.5436620    0.2591549 126.1690
##   ExerciseAnginaY   Oldpeak ST_SlopeFlat ST_SlopeUp
## 0       0.1289199 0.4191638    0.1951220  0.7665505
## 1       0.6197183 1.2509859    0.7661972  0.1380282
##
## Coefficients of linear discriminants:
##                            LD1
## Age               0.0073025832
## SexM              0.6082210140
## ChestPainTypeATA -0.9424150241
## ChestPainTypeNAP -0.9625298833
## ChestPainTypeTA  -0.6385046288
## RestingBP         0.0007992514
```

```
## Cholesterol      -0.0020805125
## FastingBS         0.4553073760
## RestingECGNormal -0.0155599596
## RestingECGST     -0.0372528210
## MaxHR            -0.0027605962
## ExerciseAnginaY   0.5837843123
## Oldpeak           0.1869260366
## ST_SlopeFlat      0.5989022889
## ST_SlopeUp       -1.0220419698
```

```
lda.class = predict(lda.out, test)$class
table(lda.class, test$HeartDisease)
```

```
##
## lda.class   0   1
##         0 103  19
##         1  20 134
```

```
model_QDA = qda(HeartDisease ~ ., data = train)
model_QDA
```

```
## Call:
## qda(HeartDisease ~ ., data = train)
##
## Prior probabilities of groups:
##         0         1
## 0.4470405 0.5529595
##
## Group means:
##        Age      SexM ChestPainTypeATA ChestPainTypeNAP ChestPainTypeTA
## 0 50.78049 0.6376307       0.36585366        0.3310105      0.05574913
## 1 55.99437 0.8957746       0.04788732        0.1436620      0.03943662
##   RestingBP Cholesterol FastingBS RestingECGNormal RestingECGST    MaxHR
## 0  130.6516    227.6725 0.1289199        0.6236934    0.1672474 147.1847
## 1  134.8592    178.7042 0.3521127        0.5436620    0.2591549 126.1690
##   ExerciseAnginaY   Oldpeak ST_SlopeFlat ST_SlopeUp
## 0       0.1289199 0.4191638    0.1951220  0.7665505
## 1       0.6197183 1.2509859    0.7661972  0.1380282
```

```
QDA.class =  predict(model_QDA, test)$class
table(QDA.class, test$HeartDisease)
```

```
##
## QDA.class   0   1
##         0 109  21
##         1  14 132
```

```
train_numerical_only = train[-c(2, 3, 7, 9, 11)]
test_numerical_only = test[-c(2, 3, 7, 9, 11)]

test_pred <- knn(train = scale(train_numerical_only), test = scale(test_numerical_only),
                 cl = train$HeartDisease, k = 3)

table(true_state = test$HeartDisease, prediction = test_pred)
```

```
##           prediction
## true_state   0   1
```

```
##           0 123   0
##           1   0 153
```

2b i

We find that the overall fraction of correct predictions for the test data is $\frac{103+134}{103+134+19+20} = .859$

2b ii

We find that the overall fraction of correct predictions for the test data is $\frac{109+132}{109+132+21+14} = .873$

2b iii

Upon using many different values for K, as long as K is suitably small, the overall fraction of correct predictions for the test data is $\frac{123+153}{123+153+0+0} = 1$. If we make K inappropriately large, say K = 100, then the overall fraction of correct predictions for the test data becomes slightly lower, but not dramatically.

2c

All of the methods above seem to produce high quality estimates, but on this particular training/testing set, the KNN classifier has the highest accuracy. It appears that the boundary region is highly non-linear leading to KNN producing quality results. The KNN may provide better results due to a large sample size, giving KNN an advantage over the QDA classifier. It may be that in the LDA approach that we have violated our assumption that there is a common covariance matrix (tests of hypothesis can be used to verify if this assumption has been violated).

```r
significant_predictors <- heart_data[c(1, 2, 3, 5, 6, 9, 10, 11, 12)]

train_ind <- sample(seq_len(nrow(heart_data)), size = smp_size)

train <- significant_predictors[train_ind, ]
test <- significant_predictors[-train_ind, ]

model <- glm(formula = HeartDisease ~ ., family = binomial, data = significant_predictors)

summary(model)
```

```
##
## Call:
## glm(formula = HeartDisease ~ ., family = binomial, data = significant_predictors)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.6504  -0.3732   0.1735   0.4444   2.6215
##
## Coefficients:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)      -1.718801   0.851936  -2.018 0.043641 *
## Age               0.023059   0.011855   1.945 0.051770 .
## SexM              1.465131   0.277999   5.270 1.36e-07 ***
## ChestPainTypeATA -1.857269   0.322969  -5.751 8.89e-09 ***
## ChestPainTypeNAP -1.718681   0.262777  -6.540 6.13e-11 ***
```

```
## ChestPainTypeTA  -1.491457   0.428065  -3.484 0.000494 ***
## Cholesterol      -0.003981   0.001027  -3.877 0.000106 ***
## FastingBS         1.133219   0.273450   4.144 3.41e-05 ***
## ExerciseAnginaY   0.935998   0.237673   3.938 8.21e-05 ***
## Oldpeak           0.377384   0.116562   3.238 0.001205 **
## ST_SlopeFlat      1.458074   0.427810   3.408 0.000654 ***
## ST_SlopeUp       -1.027949   0.445824  -2.306 0.021126 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1262.14  on 917  degrees of freedom
## Residual deviance:  595.81  on 906  degrees of freedom
## AIC: 619.81
##
## Number of Fisher Scoring iterations: 5
```

```
predictions <- predict(model, newdata = test, type = "response")

true_false <- predictions > .5
true_false[true_false == TRUE] = 1
true_false[true_false == FALSE] = 0

conf_matrix <- table(reference = test[,length(test)], prediction = true_false)
conf_matrix
```

```
##          prediction
## reference   0   1
##         0 100  20
##         1  21 135
```

```
lda.out = lda(HeartDisease ~ ., data = train)
lda.out
```

```
## Call:
## lda(HeartDisease ~ ., data = train)
##
## Prior probabilities of groups:
##         0         1
## 0.4517134 0.5482866
##
## Group means:
##        Age      SexM ChestPainTypeATA ChestPainTypeNAP ChestPainTypeTA
## 0 50.07586 0.6482759       0.36551724        0.3241379      0.07241379
## 1 55.92330 0.9090909       0.03977273        0.1363636      0.04261364
##   Cholesterol FastingBS ExerciseAnginaY   Oldpeak ST_SlopeFlat ST_SlopeUp
## 0    225.3207 0.1068966       0.1310345 0.4151724    0.1896552  0.7758621
## 1    175.2557 0.3579545       0.6306818 1.3150568    0.7443182  0.1363636
##
## Coefficients of linear discriminants:
##                           LD1
## Age                0.011090529
## SexM               0.712683134
## ChestPainTypeATA -1.107653998
## ChestPainTypeNAP -0.958329402
```

```
## ChestPainTypeTA  -0.795854039
## Cholesterol      -0.001784774
## FastingBS         0.535889912
## ExerciseAnginaY   0.548591377
## Oldpeak           0.216705347
## ST_SlopeFlat      0.555987681
## ST_SlopeUp       -0.962894768
```

```
lda.class = predict(lda.out, test)$class
table(lda.class, test$HeartDisease)
```

```
##
## lda.class   0   1
##         0 101  18
##         1  19 138
```

```
model_QDA = qda(HeartDisease ~ ., data = train)
model_QDA
```

```
## Call:
## qda(HeartDisease ~ ., data = train)
##
## Prior probabilities of groups:
##         0         1
## 0.4517134 0.5482866
##
## Group means:
##        Age      SexM ChestPainTypeATA ChestPainTypeNAP ChestPainTypeTA
## 0 50.07586 0.6482759       0.36551724        0.3241379      0.07241379
## 1 55.92330 0.9090909       0.03977273        0.1363636      0.04261364
##   Cholesterol FastingBS ExerciseAnginaY   Oldpeak ST_SlopeFlat ST_SlopeUp
## 0    225.3207 0.1068966       0.1310345 0.4151724    0.1896552  0.7758621
## 1    175.2557 0.3579545       0.6306818 1.3150568    0.7443182  0.1363636
```

```
QDA.class =  predict(model_QDA, test)$class
table(QDA.class, test$HeartDisease)
```

```
##
## QDA.class   0   1
##         0 102  30
##         1  18 126
```

2d

We attempt to use only the predictors that we identified as being statistically significant in our logistic regression model. Since KNN does not have the appropriate structure for categorical variables we will not discuss using KNN for this analysis. We also have a KNN model which does not seem to be producing errors. We see from above that our KNN with a relatively small k, using only numerical predictors has better accuracy than any model calculated above.

It appears as though the aforementioned KNN model fit with k = 3 and using only numerical categories still remains as the most accurate model tested so far.

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.

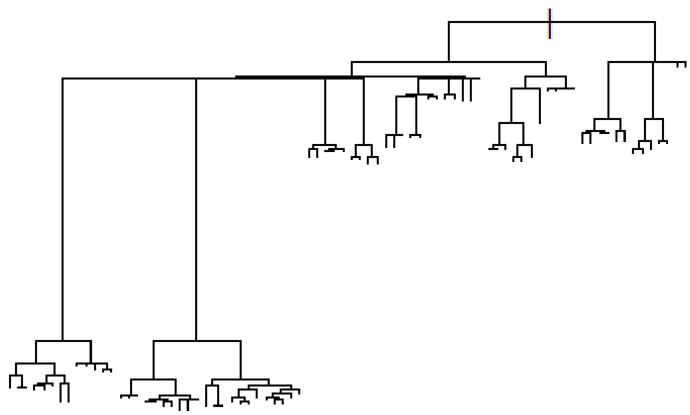
```
summary(tree.d)

##
## 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}

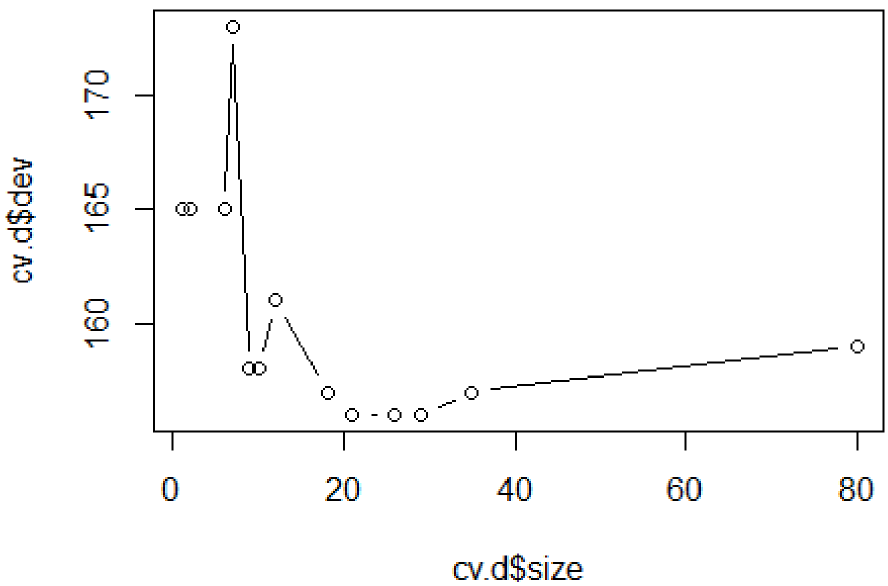
Asked one:-let's also look for the plot when cross-validation is done on the basis of misclassification

```
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"
```

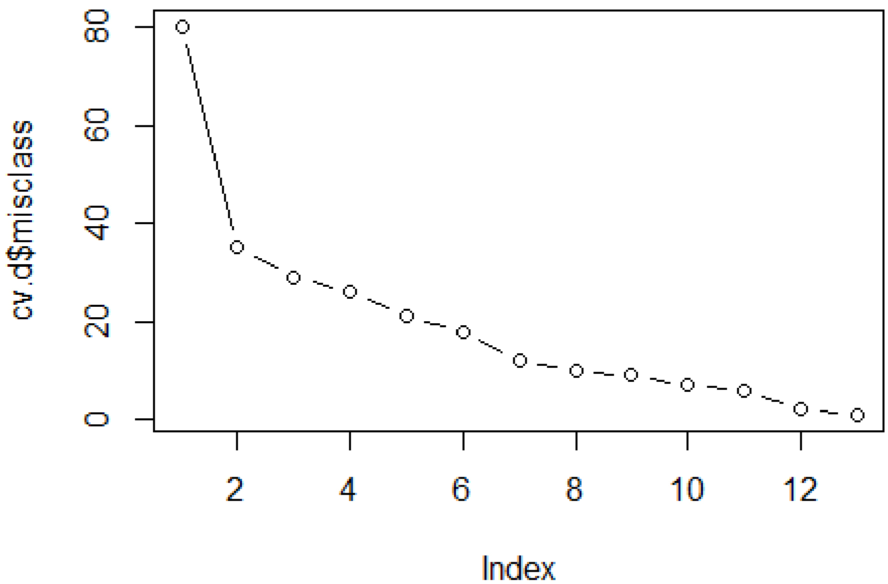
Clearly from above result we can see that the tree with either: 29,26 or 21 terminal nodes results in only 156 cross-validation error (which is minimum one) and same for all given three nodes.

let's visualize this

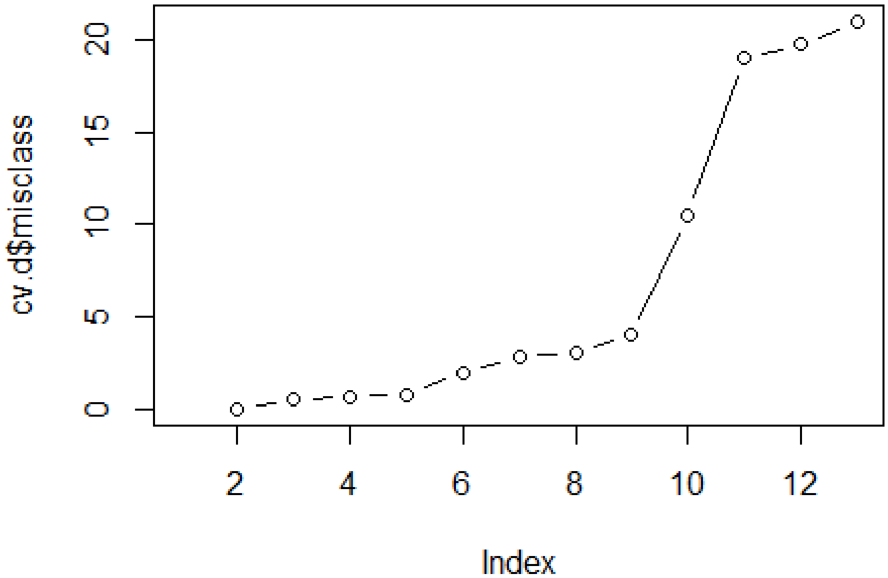
```
plot(cv.d$size, cv.d$dev, type = "b")
```




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




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



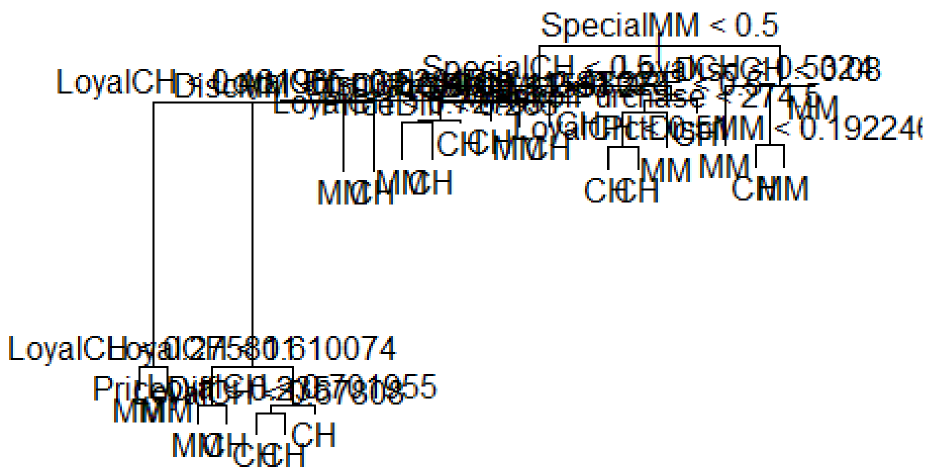
(5) Produce a pruned tree corresponding to the optimal tree size obtained using cross-validation. If cross-validation does not lead to selection of a pruned tree, then create a pruned tree with five terminal nodes.

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
```

```
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

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.

```
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.

```
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 ˆm. 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

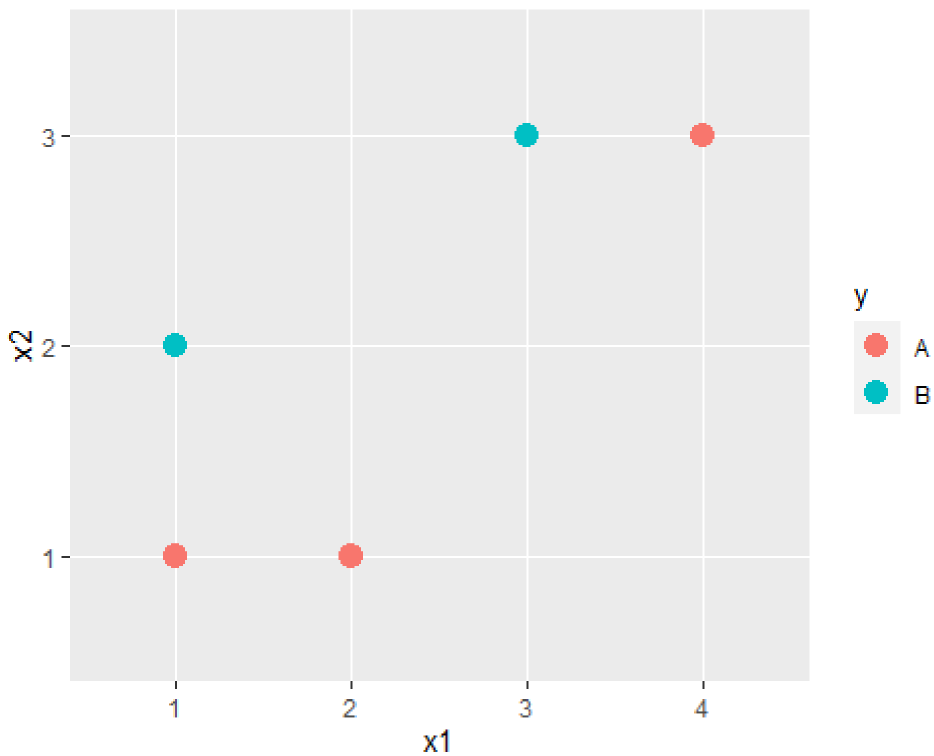
————————-THE END——————————————————

Example stat 562

Sagar Kalauni

2023-11-07

```
library(tidyverse)

## Warning: package 'tidyverse' was built under R version 4.3.1

## Warning: package 'ggplot2' was built under R version 4.3.1

## Warning: package 'lubridate' was built under R version 4.3.1

## ── Attaching core tidyverse packages ──────────────────────── tidyverse
2.0.0 ──
## ✔ dplyr     1.1.2     ✔ readr     2.1.4
## ✔ forcats   1.0.0     ✔ stringr   1.5.0
## ✔ ggplot2   3.4.2     ✔ tibble    3.2.1
## ✔ lubridate 1.9.2     ✔ tidyr     1.3.0
## ✔ purrr     1.0.1
## ── Conflicts ──────────────────────────────────────────
tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all
conflicts to become errors

y=c("A","B","A","B","A")
x1=c(1,1,4,3,2)
x2=c(1,2,3,3,1)
data=as.data.frame(cbind(y,x1,x2))
ggplot(data,aes(x=x1,y=x2,col=y))+geom_point(size=4)
```



#originally: # – Originally we are checking how much impurity does the data set have in the very starting. #– Since this is the very small data set so we are doing it manually, but it is not possible to do like this #– manually in a huge data set.

```
g=2/5*3/5*2
```

#0.48

#choosing 1st node split: #– Start spliting through x-axis # – Now here we are trying different place to split the data set and according to each place we are tying to #– to calculate the impurity, and we get minimum impurity in g1.3 this split (means x1 less then3.5 and greater then 3.5)

```
g1.1=1/2*1/2*2*(0.4)+1/3*2/3*2*(0.6) #0.4667
g1.2=1/3*2/3*2*(0.6)+1/2*1/2*2*(0.4) #0.4667
g1.3=1/2*1/2*2*(0.8)+0*0.2 #0.4
```

Now we will similarly split through y-axis

check the impurity in the each split and choose the one having the minimum impurity, and mimimum impurity

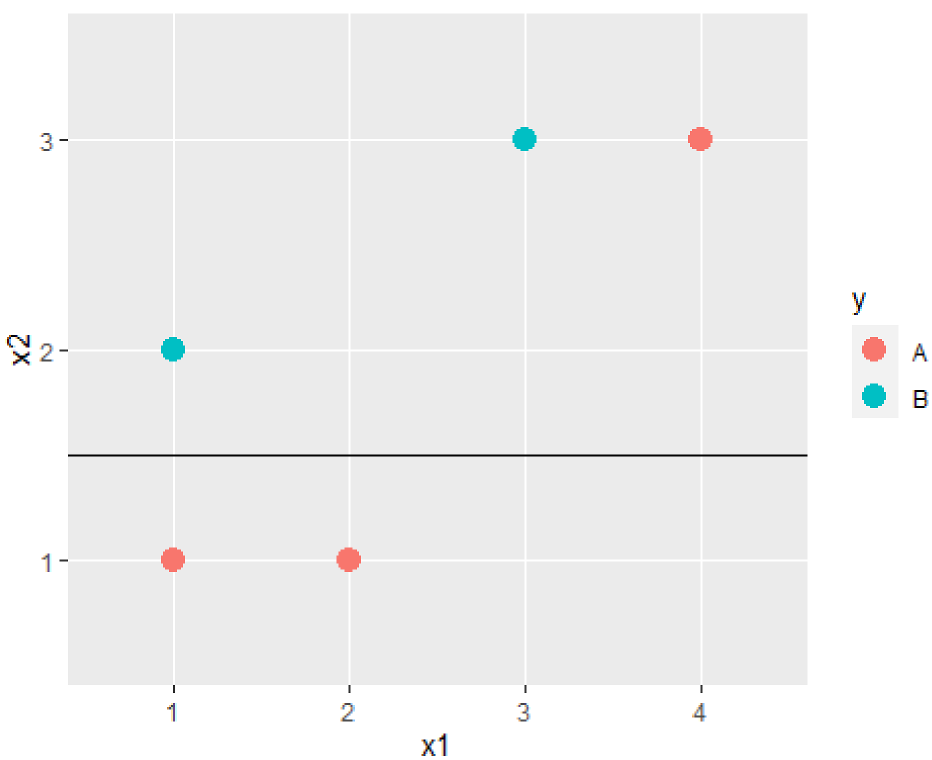
will be for the one having the pure node.

The minimum impurity is along g2.2 so we will split along that, means x2<1.5 or x2>1.5

```
g2.1=0*(0.4)+1/3*2/3*2*(0.6) #0.2667
g2.2=1/3*2/3*2*(0.6)+1/2*1/2*2*(0.4) #0.4667
```

#pick x2< 1.5 v.s x2 > 1.5 as 1st split

```
ggplot(data,aes(x=x1,y=x2,col=y))+geom_point(size=4)+geom_hline(yintercept=1.
5)
```

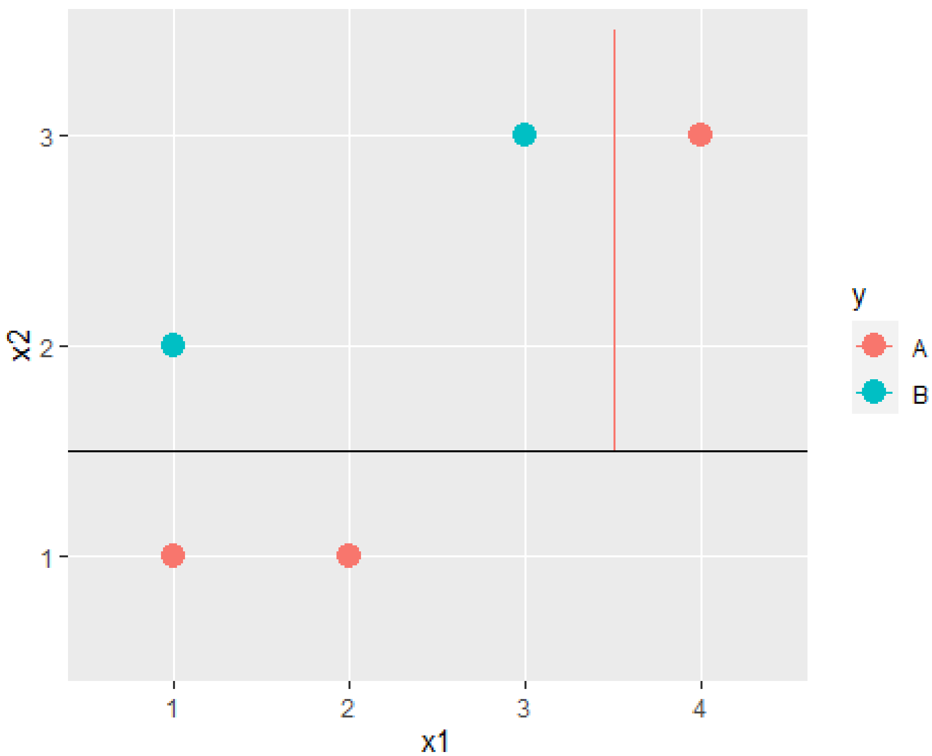


#choosing 2nd split

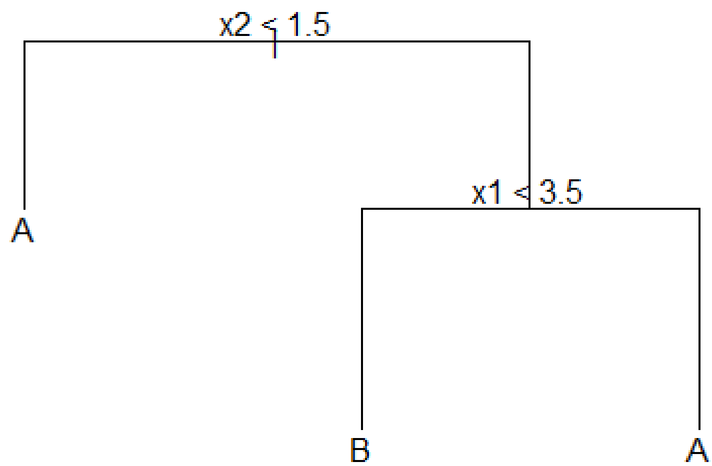
```
g1.1=0*1/3+1/2*1/2*2*2/3 #0.333
g1.3=0
g2.2=0*1/3+1/2*1/2*2*2/3 #0.333
```

#pick x1< 3.5 v.s x1 > 3.5 as 2nd split

```
ggplot(data,aes(x=x1,y=x2,col=y))+geom_point(size=4)+
  geom_hline(yintercept=1.5)+geom_segment(aes(x = 3.5, y = 1.5, xend = 3.5,
yend = 3.5))
```




```
library(tree)

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

out=tree(as.factor(y)~.,data,control=tree.control(nobs=5,mincut = 0,
minsize=0, mindev = 0))
plot(out)
text(out)
```



##Classification Tree Example, Default data

```r
library(tree)
library(ISLR2)

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

train=sample(1:10000,7000) # We take 7000 for training and 3000 for test
test=Default[-train,]
tree.d=tree(default~.,Default,split="gini",subset=train)
```

– default is only one column of the table which need to be predicted

– the name of the data set is Default which is in the ISLR2 library

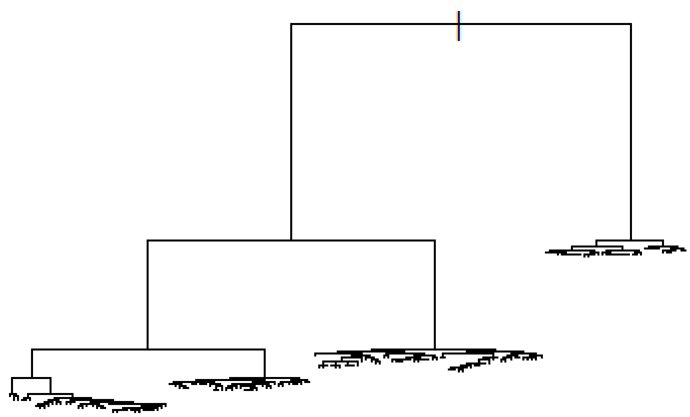
–code: default is the variable I need to predict and I want to predict it crossponidng to all prediction (~.)

– my data set name is Default and spliting criteria is gini and I will only make tree using tarining data set.

```r
summary(tree.d)
```

```
##
## Classification tree:
## tree(formula = default ~ ., data = Default, subset = train, split =
"gini")
## Number of terminal nodes:  156
## Residual mean deviance:  0.0955 = 653.6 / 6844
## Misclassification error rate: 0.024 = 168 / 7000
```

```
plot(tree.d)
```



#predict class on test data

```
pred.d=predict(tree.d,test,type="class")
table(pred.d,test$default)
```

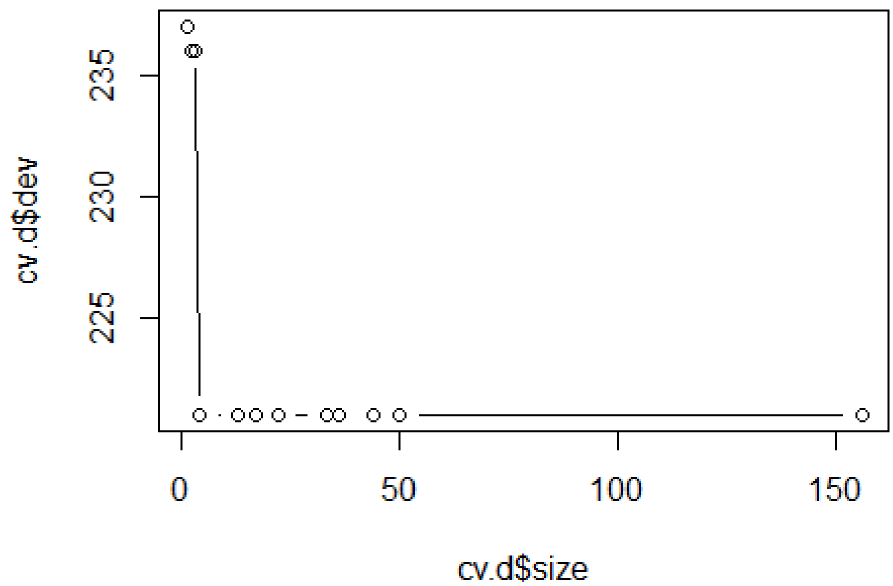
```
##
## pred.d   No  Yes
##     No  2854  43
##     Yes   50  53
```

#pruning

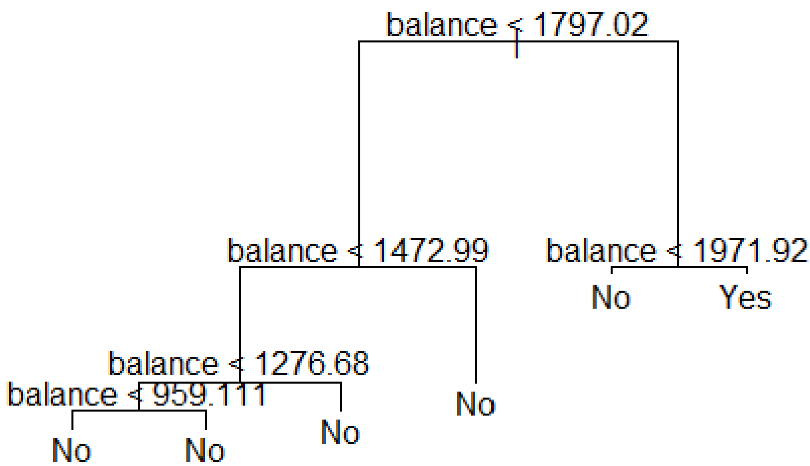
```
cv.d=cv.tree(tree.d)
```

#or if you want to use misclassification rate for the CV instead of the default deviance,

```
cv.d=cv.tree(tree.d,FUN = prune.misclass)
plot(cv.d$size, cv.d$dev, type="b")
```




```
prune.d=prune.tree(tree.d,best=6)
summary(prune.d)

##
## Classification tree:
## snip.tree(tree = tree.d, nodes = c(9L, 7L, 16L, 5L, 17L, 6L))
## Variables actually used in tree construction:
## [1] "balance"
## Number of terminal nodes:  6
## Residual mean deviance:  0.161 = 1126 / 6994
## Misclassification error rate: 0.02886 = 202 / 7000

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



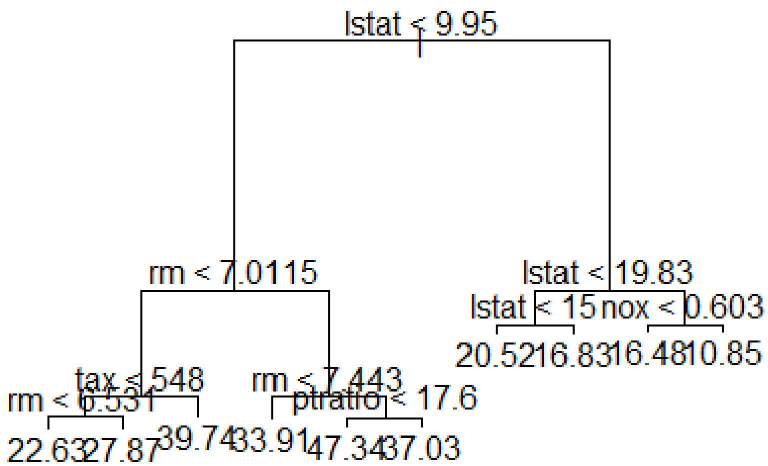
#predict class on test data

```
pred.d.prune=predict(prune.d,test,type="class")
table(pred.d.prune,test$default)

##
## pred.d.prune   No  Yes
##           No 2893   58
##          Yes   11   38
```

#Regression Tree Example: Boston House Price

```
library(ISLR2)
train=sample(1:506,350) # We take 350 for training and  for test
test=Boston[-train,]
tree.b=tree(medv~.,Boston,subset=train)
summary(tree.b)

##
## Regression tree:
## tree(formula = medv ~ ., data = Boston, subset = train)
## Variables actually used in tree construction:
## [1] "lstat"   "rm"      "tax"     "ptratio" "nox"
## Number of terminal nodes:  10
## Residual mean deviance:  13.28 = 4515 / 340
## Distribution of residuals:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -16.04000  -2.04600   0.06297   0.00000   2.17300  16.09000
```

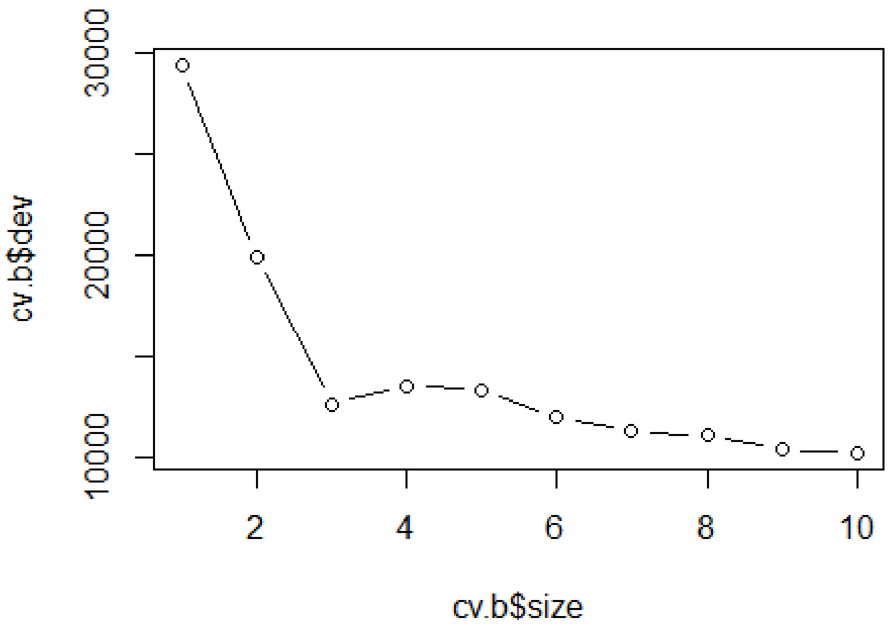
```r
plot(tree.b)
text(tree.b,pretty = 0)
```



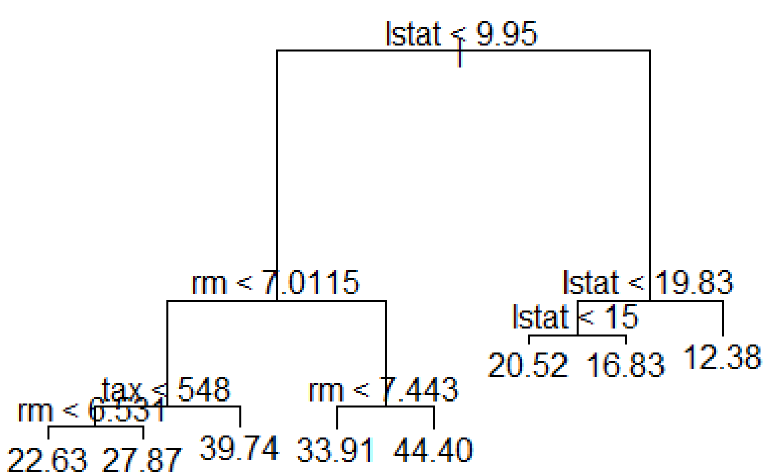

```r
test.mse=mean((test$medv-predict(tree.b,test))^2)
test.mse

## [1] 17.33287
```

#pruning if needed

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




```
prune.b=prune.tree(tree.b,best=8)
plot(prune.b)
text(prune.b)
```



Homework-3 ML 562

Sagar Kalauni

2023-11-28

1. Let's explore the maximal margin classifier on a toy data set. We are given n = 7 observations in p = 2 dimensions. For each observation, there is an associated class label Y .

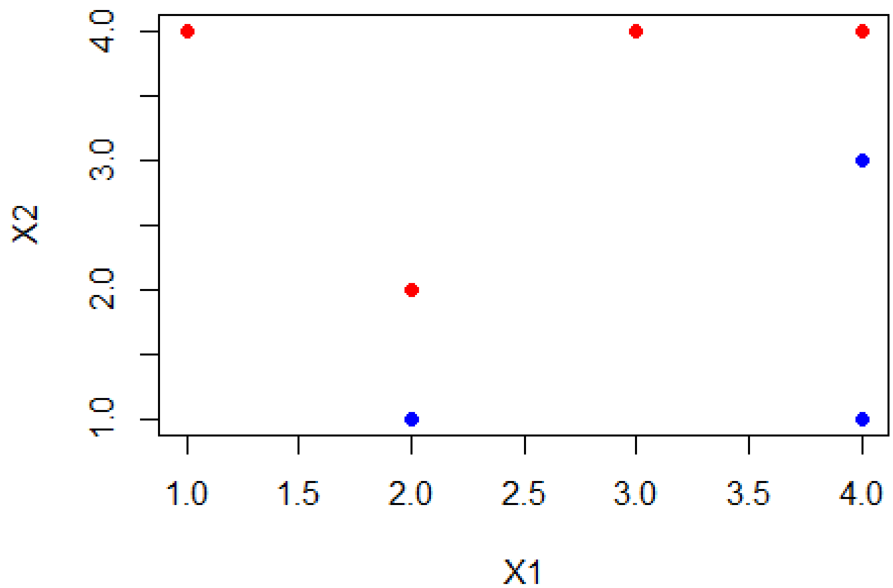
```
set.seed(12321)
X1 <- c(3,2,4,1,2,4,4)
X2 <- c(4,2,4,4,1,3,1)
Y  <- c(rep("Red", 4),rep("Blue", 3))
mydf <- data.frame(X1, X2, Y)

mydf
```

```
##   X1 X2    Y
## 1  3  4  Red
## 2  2  2  Red
## 3  4  4  Red
## 4  1  4  Red
## 5  2  1 Blue
## 6  4  3 Blue
## 7  4  1 Blue
```

```
set.seed(12321)
#install.packages("tidyverse")
library(ggplot2)
#install.packages("e1071")
library(e1071)
```

(a) Sketch the observations.

```
set.seed(12321)
# Creating the scatter plot
plot(mydf$X1, mydf$X2, col=Y, pch=19, xlab = "X1", ylab = "X2")
```



Observation: From the above plot we can say that they are linearly seperable.

(b) Sketch the optimal separating hyperplane.

```
set.seed(12321)
library(e1071)
mydf$Y=as.factor(mydf$Y)

# Fitting our model with some random cost
fit.svm = svm(Y ~ ., data = mydf, kernel = "linear", cost = 10, scale =
FALSE)
fit.svm$index

## [1] 2 3 6
```

-I believe to sketch the optimal separating hyperplane, my model should also have to be the one with the best fit(among the tested cost values), So I will first find the best fitting model and then draw Optimal separating hyperplane with the help of that.

- The optimal separating hyperplane refers to the decision boundary that maximally separates different classes in the feature space.

-Here I want to do cross-validation to find the best model but tune function by default take 10-fold cross validation and my sample size was not enough to do that so I need to do the tunecontrol and preform cross-validation.

Cross-validation to find the best fit model

```
set.seed(12321)
# perform cross-validation
tune.out <- tune(
  svm,                        # SVM function
  Y ~ .,                      # Formula for the model
  data = mydf,                 # my data frame
  kernel = "linear",          # Linear kernel
  ranges = list(cost = c(0.001, 0.01, 0.1, 1, 5, 10, 100)),  # Range of cost
values(she did not mention in particular which value to take so I am taking
of my wish)
  tunecontrol = tune.control(sampling = "cross", cross = 2)  # 2-fold cross-
validation
)

summary(tune.out)

##
## Parameter tuning of 'svm':
##
## - sampling method: 2-fold cross validation
##
## - best parameters:
##  cost
##     5
##
## - best performance: 0.25
##
## - Detailed performance results:
##    cost     error dispersion
## 1 1e-03 0.5833333  0.1178511
## 2 1e-02 0.5833333  0.1178511
## 3 1e-01 0.5833333  0.1178511
## 4 1e+00 0.5833333  0.1178511
## 5 5e+00 0.2500000  0.3535534
## 6 1e+01 0.2500000  0.3535534
## 7 1e+02 0.2500000  0.3535534
```

Observation: -Here 5e+00 means $5 \times 10^0 = 5$, so we can see the error is minimum when cost is 5. So our model will be best when cost=5 (amoung the given cost values)

Now we have clear idea which cost will give us our best model, so let's find our best model

```
best.mod=svm(Y ~ ., data = mydf, kernel = "linear", cost = 5, scale = FALSE,
)
best.mod

##
## Call:
## svm(formula = Y ~ ., data = mydf, kernel = "linear", cost = 5, ,
```

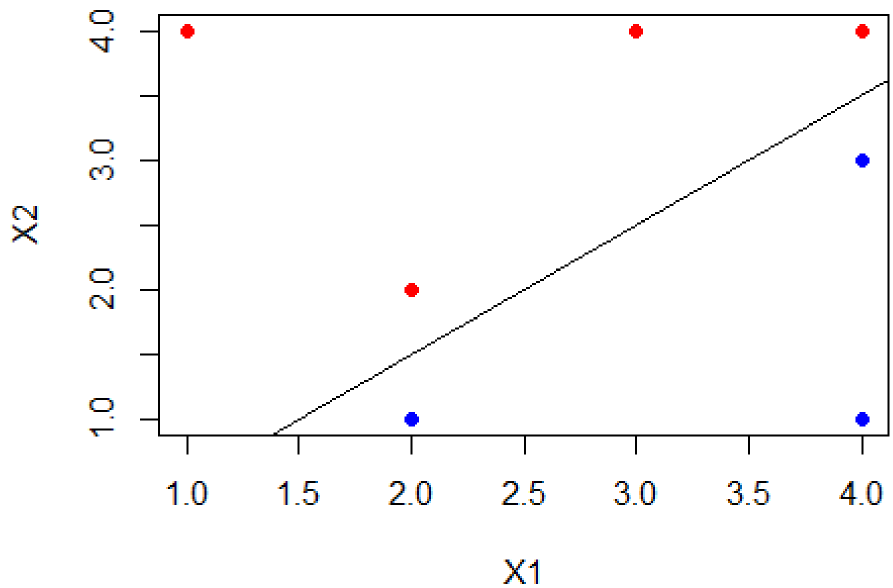
```
##     scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  5
##
## Number of Support Vectors:  3
```

```
set.seed(12321)
summary(tune.out$best.model)
```

```
##
## Call:
## best.tune(METHOD = svm, train.x = Y ~ ., data = mydf, ranges = list(cost =
c(0.001,
##     0.01, 0.1, 1, 5, 10, 100)), tunecontrol = tune.control(sampling =
"cross",
##     cross = 2), kernel = "linear")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  5
##
## Number of Support Vectors:  4
##
##  ( 2 2 )
##
##
## Number of Classes:  2
##
## Levels:
##  Blue Red
```

Now using this best model to sketch the optimal separating hyperplane.

```
set.seed(12321)
# Extract beta_0 and beta_1
beta0 = best.mod$rho
beta = drop(t(best.mod$coefs) %*% as.matrix(mydf[best.mod$index,1:2]))

# Replot, this time with the solid line representing the optimal(maximal)
margin plane.
plot(X1, X2, col=Y, pch=19, data=mydf)
abline(beta0/beta[2], -beta[1]/beta[2])
```



Here we got our optimal seperating hyperplane In code the a, b arguments above in abline() represent the intercept and slope, single values in the plot functions.

(c) Provide the equation for this hyperplane. Describe the classification rule. It should be something along the lines of ?Classify to Red if $\beta_0 + \beta_1 X_1 + \beta_2 X_2 > 0$ ANSWER: The equation of the given hyperplane is: $\beta_0 + \beta_1 X_1 + \beta_2 X_2 = 0$, where $_0 = -1.00041, _1=-1.999846,_2=1.999693 $

Hence the exect equation of the hyperplane is: $-1.00041 + -1.999846X_1 + 1.999693X_2 = 0$ which on simplification became: $X_2 = -1.000077X_1 + (-0.500281)$

```
set.seed(12321)
paste("Intercept: ", round(beta0/beta[2],1), ", Slope: ", round(-
beta[1]/beta[2],1), sep="")

## [1] "Intercept: -0.5, Slope: 1"
```

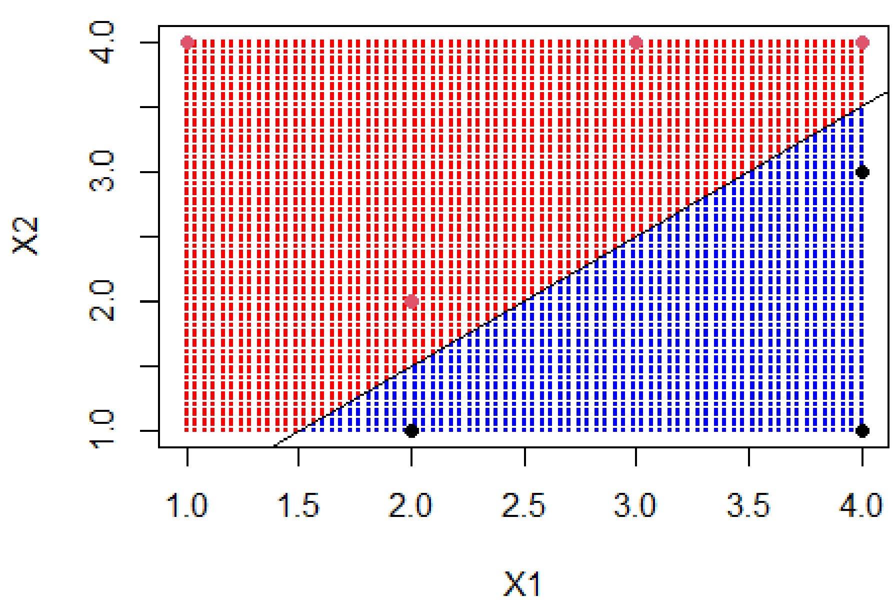
If the Values were rounded then the equation becomes: $X_2 = 1X_1 + (-0.5)$

-The Classification Rule is any point that lies below hyperplane(lower half space) will be classified as blue and any point that lies above the hyperplane(upper half space) will be classified as Red.

Mathematically, any point lies in $\beta_0 + \beta_1 X_1 + \beta_2 X_2 > 0$ classified as RED otherwise BLUE

```
set.seed(12321)
# Making better plot
make.grid = function(x, n = 75) {
```

```r
    grange = apply(x, 2, range)
    x1 = seq(from = grange[1,1], to = grange[2,1], length = n)
    x2 = seq(from = grange[1,2], to = grange[2,2], length = n)
    expand.grid(X1 = x1, X2 = x2)
 }
xgrid = make.grid(mydf)
ygrid = predict(best.mod, xgrid)

plot(xgrid, col = c("blue","Red")[as.numeric(ygrid)], pch = 20, cex = .2)
points(mydf, col =mydf$Y, pch = 19)
#points(mydf[best.mod$index,1:2], pch = 5, cex = 2)



### Add the margins
## you have to do some work to get back the linear coefficients
beta = t(best.mod$coefs)%*%as.matrix(mydf[best.mod$index,1:2])
beta0 = best.mod$rho

abline(beta0 / beta[2], -beta[1] / beta[2])
```

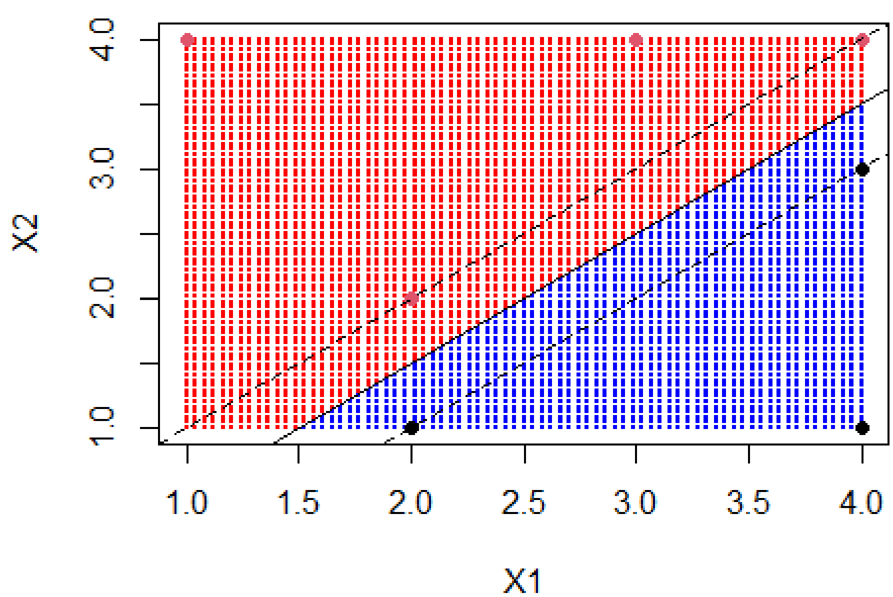



```r
#abline((beta0 - 1) / beta[2], -beta[1] / beta[2], lty = 2)
#abline((beta0 + 1) / beta[2], -beta[1] / beta[2], lty = 2)
```

(d) On your sketch, indicate the margin for the maximal margin hyperplane.

```r
set.seed(12321)
# Making better plot
```

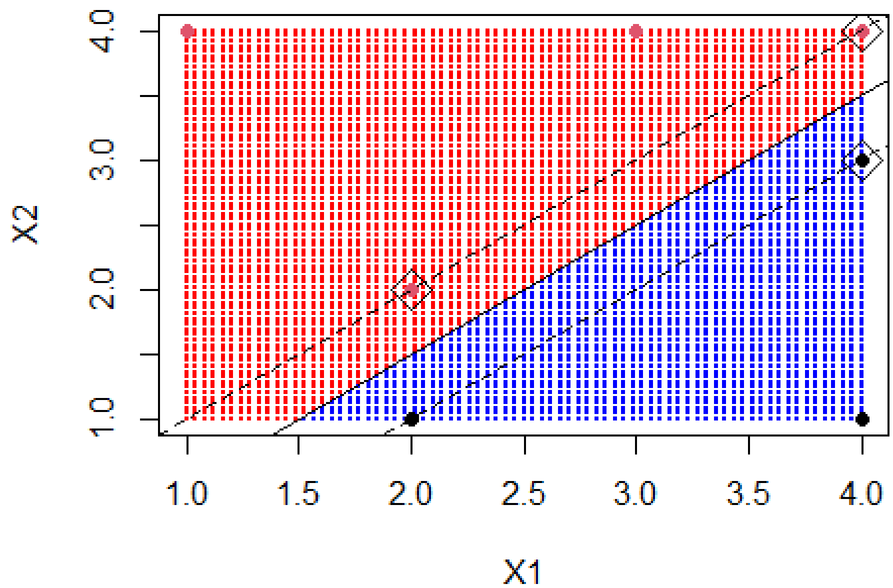
```r
make.grid = function(x, n = 75) {
    grange = apply(x, 2, range)
    x1 = seq(from = grange[1,1], to = grange[2,1], length = n)
    x2 = seq(from = grange[1,2], to = grange[2,2], length = n)
    expand.grid(X1 = x1, X2 = x2)
 }
xgrid = make.grid(mydf)
ygrid = predict(best.mod, xgrid)

plot(xgrid, col = c("blue","Red")[as.numeric(ygrid)], pch = 20, cex = .2)
points(mydf, col =mydf$Y, pch = 19)
#points(mydf[best.mod$index,1:2], pch = 5, cex = 2)


### Add the margins
## you have to do some work to get back the linear coefficients
beta = t(best.mod$coefs)%*%as.matrix(mydf[best.mod$index,1:2])
beta0 = best.mod$rho

abline(beta0 / beta[2], -beta[1] / beta[2])
abline((beta0 - 1) / beta[2], -beta[1] / beta[2], lty = 2)
abline((beta0 + 1) / beta[2], -beta[1] / beta[2], lty = 2)
```



Observation: -To find the margin length we compute the smallest distance from any training observation to the given separating hyperplane. This is the same as computing the distance from the dashed margin line to the solid hyperplane.The margin width is from the solid line to either of the dashed lines

(e) Indicate the support vectors for the maximal margin classifier.

```
set.seed(12321)
# Making better plot
make.grid = function(x, n = 75) {
    grange = apply(x, 2, range)
    x1 = seq(from = grange[1,1], to = grange[2,1], length = n)
    x2 = seq(from = grange[1,2], to = grange[2,2], length = n)
    expand.grid(X1 = x1, X2 = x2)
 }
xgrid = make.grid(mydf)
ygrid = predict(best.mod, xgrid)

plot(xgrid, col = c("blue","Red")[as.numeric(ygrid)], pch = 20, cex = .2)
points(mydf, col =mydf$Y, pch = 19)
points(mydf[best.mod$index,1:2], pch = 5, cex = 2)


### Add the margins
## you have to do some work to get back the linear coefficients
beta = t(best.mod$coefs)%*%as.matrix(mydf[best.mod$index,1:2])
beta0 = best.mod$rho

abline(beta0 / beta[2], -beta[1] / beta[2])
abline((beta0 - 1) / beta[2], -beta[1] / beta[2], lty = 2)
abline((beta0 + 1) / beta[2], -beta[1] / beta[2], lty = 2)
```

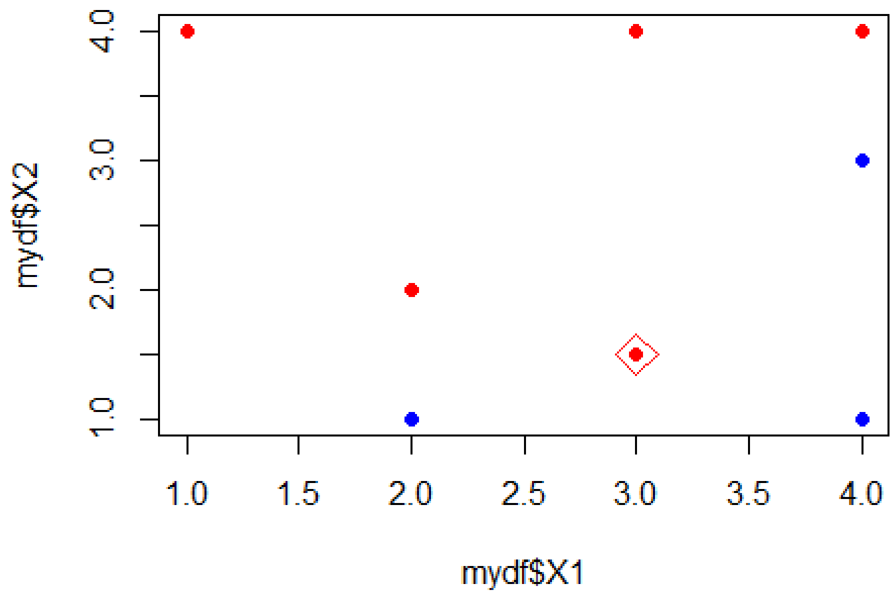


Observation: -

Support vectors are indicated by the square box around them.

(f) Draw an additional observation on the plot so that the two classes are no longer separable by a hyperplane. Answer: So In order to make data no longer separable by a hyperplane, I just need to add one point (at least, I can add more also) in the opposite side of the halfspace determined by our hyperplane. If our hyperplane classify all point to be blue in the lower halfspace $\beta_0 + \beta_1 X_1 + \beta_2 X_2 < 0$, I will add one Red point over there, then hyperplane can not seperate them. I need to keep in mind that, newly added point should be outside of the margin also

```
set.seed(12321)
plot(mydf$X1, mydf$X2, col=Y, pch=19)
points(3, 1.5, col="Red", pch=19)
points(3, 1.5, col="red", pch=5, cex=2)
```



Observation: This newly added data point which is red point with red squre around make the data point linearly inseperable that means we can not seperate our two classes using linear classifier like hyperplane.

---

2. In this problem, you will use support vector approaches in order to predict Purchase based on the OJ data set.

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

```
set.seed(100)
library(ISLR2) #Loading the ISLR2 library in the R working environment

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

set.seed(100)
# Load the OJ dataset
data(OJ)
dim(OJ)

## [1] 1070   18

# Spliting the data into training and testing set
set.seed(100)
Index=sample(1:nrow(OJ), 800) # we take 800 data for training set
train=OJ[Index,]
test=OJ[-Index,]
```

(b) Fit a linear SVM to the training data using cost=0.01, with Purchase as the response and the other variables as predictors. Describe the results obtained.

```
set.seed(100)
library(e1071)

# Fitting a linear model with cost=0.01
OJ.fit.svm = svm(Purchase ~ ., data =train, kernel = "linear", cost = 0.01,
scale = FALSE)
OJ.fit.svm

##
## Call:
## svm(formula = Purchase ~ ., data = train, kernel = "linear", cost = 0.01,
##     scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.01
##
## Number of Support Vectors:  623

set.seed(100)
summary(OJ.fit.svm)

##
## Call:
## svm(formula = Purchase ~ ., data = train, kernel = "linear", cost = 0.01,
##     scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.01
##
## Number of Support Vectors:  623
##
##  ( 312 311 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

Observation: Summary tells us that, the linear kernel was used with cost=0.01 and that there were 623 support vectors, out of which 312 belongs to one class and 311 belongs to the other class. Number of classes are two with levels CH and MM

(c) What are the training and test error rates?

```
set.seed(100)
# Prediciting the class for our training dataset
pred_train=predict(OJ.fit.svm, train)
pred_train[1:10] # Looking at the first 10 prediction made by our model in the training dataset

##  503  985 1004  919  470  823  838  903 1031  183
##   CH   CH   CH   CH   CH   CH   MM   CH   CH   CH
## Levels: CH MM

set.seed(100)
# Confusion matrix
table(pred_train, train$Purchase)

##
## pred_train  CH  MM
##         CH 466 177
##         MM  22 135
```

Observation: -Looking at the confusion matrix we see that the training error rate is: (177+22)/800= 0.24875 i.e 24.875%

Now predicting the class for our test data set using our model

```
set.seed(100)
pred_test=predict(OJ.fit.svm, test)
pred_test[1:10]   # Looking at the first 10 prediction made by our model in the Test dataset

##  3  5  7  8 20 25 27 29 33 36
## CH CH CH CH CH CH CH CH MM CH
## Levels: CH MM

set.seed(100)
# Confusion Matrix
table(pred_test, test$Purchase)

##
## pred_test  CH  MM
##         CH 157  63
##         MM   8  42
```

Observation: -Looking at the confusion matrix we see that the test error rate is: (63+8)/270= 0.262963 i.e 26.2963%

(d) Tune the linear SVM with various values of cost. Report the cross-validation errors associated with different values of this parameter. Select an optimal cost. Compute the training and test error rates using this new cost value. Comment on your findings.

```r
set.seed(100)
# perform cross-validation
OJ.tune.out <- tune(
  svm,                    # SVM function
  Purchase~.,                  # Formula for the model
  data = train,              # my training data frame
  kernel = "linear",      # Linear kernel
  ranges = list(cost = seq(0.01, 10, length.out = 20))  # Range of cost
values(she did not mention in particular which value to take so I am taking
of my wish)
)
OJ.tune.out
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##      cost
##  6.845263
##
## - best performance: 0.17
```

```r
summary(OJ.tune.out)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##      cost
##  6.845263
##
## - best performance: 0.17
##
## - Detailed performance results:
##          cost   error dispersion
## 1   0.0100000 0.17500 0.04639804
## 2   0.5357895 0.17500 0.03908680
## 3   1.0615789 0.17500 0.03908680
## 4   1.5873684 0.17250 0.03525699
## 5   2.1131579 0.17125 0.03230175
## 6   2.6389474 0.17125 0.03438447
## 7   3.1647368 0.17375 0.03251602
## 8   3.6905263 0.17250 0.03476109
## 9   4.2163158 0.17250 0.03476109
## 10  4.7421053 0.17125 0.03729108
## 11  5.2678947 0.17125 0.03729108
```

```
## 12  5.7936842 0.17125 0.03729108
## 13  6.3194737 0.17125 0.03729108
## 14  6.8452632 0.17000 0.03782269
## 15  7.3710526 0.17000 0.03782269
## 16  7.8968421 0.17000 0.03782269
## 17  8.4226316 0.17000 0.03782269
## 18  8.9484211 0.17000 0.03782269
## 19  9.4742105 0.17000 0.03782269
## 20 10.0000000 0.17000 0.03782269
```

Observation: -Here we can see the error is minimum when cost is 6.845263. So our model will be best when cost=6.845263 So the best performance model can be obtained using cost=0.01(depending upon the cost which we have tried on , can not say in general)

```
OJ.best.mod=svm(Purchase~ ., data = train, kernel = "linear", cost =
6.845263, scale = FALSE, )
OJ.best.mod

##
## Call:
## svm(formula = Purchase ~ ., data = train, kernel = "linear", cost =
6.845263,
##     , scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  6.845263
##
## Number of Support Vectors:  323

set.seed(100)
# Prediciting the class for our training dataset using this new best model
after cross-validation
B_pred_train=predict(OJ.best.mod, train)
B_pred_train[1:10] # Looking at the first 10 prediction made by our new best
model in the training dataset

##  503  985 1004  919  470  823  838  903 1031  183
##   CH   CH   MM   CH   CH   CH   MM   CH   CH   CH
## Levels: CH MM

set.seed(100)
# Confusion matrix
table(B_pred_train, train$Purchase)

##
## B_pred_train  CH  MM
##           CH 429  65
##           MM  59 247
```

Observation: -Looking at the confusion matrix we see that the training error rate is: (65+59)/800= 0.155 i.e 15.5% for this new best fit model.

Now predicting the class for our test data set using this new best model

```
set.seed(100)
B_pred_test=predict(OJ.best.mod, test)
B_pred_test[1:10]   # Looking at the first 10 prediction made by new best
model in the Test dataset

##  3  5  7  8 20 25 27 29 33 36
## CH CH CH CH CH CH CH CH MM CH
## Levels: CH MM

# Confusion matrix
table(B_pred_test, test$Purchase)

##
## B_pred_test  CH  MM
##          CH 144  28
##          MM  21  77
```

Observation: -Looking at the confusion matrix we see that the test error rate is: (28+21)/270= 0.1814815 i.e 18.14815% for this new best fit model.

Conclusion: This is kind of interesting observation, the training error rate goes down from 24.875% (i) to 15.5% (ii) when using the best model and test error rate goes down from 26.2963% (i) to 18.14815%. So we can say that by doing model tuning we make our model really nice compared the original one.

(e) Now repeat (d), with radial basis kernels, with different values of gamma and cost. Comment on your results. Which approach seems to give the better results on this data?

Radial

```
set.seed(100)
library(e1071)

# Fitting a linear model with cost=0.01
Radial.OJ.svm = svm(Purchase ~ ., data =train, kernel = "radial", gamma=0.5,
cost = 5, scale = FALSE)
summary(Radial.OJ.svm)

##
## Call:
## svm(formula = Purchase ~ ., data = train, kernel = "radial", gamma = 0.5,
##     cost = 5, scale = FALSE)
##
##
## Parameters:
```

```
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  5
##
## Number of Support Vectors:  451
##
##  ( 245 206 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

Summary tells us that, the radial kernel was used with cost=5, gamma=0.5 and that there were 451 support vectors, out of which 245 belongs to one class and 206 belongs to the other class. Number of classes are two with levels CH and MM

```r
set.seed(100)
# Prediciting the class for our training dataset with radial kernel
R_pred_train=predict(Radial.OJ.svm, train)
R_pred_train[1:10] # Looking at the first 10 prediction made by our model in the training dataset with radial kernel
```

```
##  503  985 1004  919  470  823  838  903 1031  183
##   CH   CH   MM   MM   CH   CH   MM   CH   CH   CH
## Levels: CH MM
```

```r
set.seed(100)
# Confusion matrix
table(R_pred_train, train$Purchase)
```

```
##
## R_pred_train  CH  MM
##           CH 459  46
##           MM  29 266
```

Observation: -Looking at the confusion matrix we see that the training error rate is: (46+29)/800= 0.09375 i.e 9.375%

now predicting for the test data

```r
set.seed(100)
R_pred_test=predict(Radial.OJ.svm, test)
R_pred_test[1:10]   # Looking at the first 10 prediction made by our model in the Test dataset
```

```
##  3  5  7  8 20 25 27 29 33 36
## CH CH CH MM CH CH CH CH MM MM
## Levels: CH MM
```

```r
set.seed(100)
# Confusion Matrix
table(R_pred_test, test$Purchase)
```

```
##
## R_pred_test  CH  MM
##          CH 133  38
##          MM  32  67
```

Observation: -Looking at the confusion matrix we see that the test error rate is: (38+32)/270= 0.2592593 i.e 25.92593% with radial kernel.

Now lets try to find the best model with radial kernel by trying different values of cost and gamma

```r
set.seed(100)
# perform cross-validation
R.OJ.tune <- tune(
  svm,                   # SVM function
  Purchase~.,                 # Formula for the model
  data = train,            # my training data frame
  kernel = "radial",     # radial kernel is used
  ranges=list(cost=c(0.001, 0.01, 0.1, 1,5,10,100),gamma=c(0.5,1,2,3,4))  # 
Range of cost values and gamma values
)
summary(R.OJ.tune)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost gamma
##     1   0.5
##
## - best performance: 0.1825
##
## - Detailed performance results:
##     cost gamma   error dispersion
## 1  1e-03   0.5 0.39000 0.03809710
## 2  1e-02   0.5 0.39000 0.03809710
## 3  1e-01   0.5 0.30000 0.03864008
## 4  1e+00   0.5 0.18250 0.04005205
## 5  5e+00   0.5 0.20375 0.03682259
## 6  1e+01   0.5 0.20875 0.03775377
## 7  1e+02   0.5 0.21750 0.03395258
## 8  1e-03   1.0 0.39000 0.03809710
## 9  1e-02   1.0 0.39000 0.03809710
## 10 1e-01   1.0 0.34250 0.04090979
## 11 1e+00   1.0 0.19375 0.03784563
```

```
## 12 5e+00   1.0 0.21375 0.03606033
## 13 1e+01   1.0 0.21125 0.03747684
## 14 1e+02   1.0 0.22750 0.03425801
## 15 1e-03   2.0 0.39000 0.03809710
## 16 1e-02   2.0 0.39000 0.03809710
## 17 1e-01   2.0 0.37375 0.04267529
## 18 1e+00   2.0 0.21125 0.04059026
## 19 5e+00   2.0 0.22625 0.03458584
## 20 1e+01   2.0 0.22625 0.03356689
## 21 1e+02   2.0 0.23375 0.03335936
## 22 1e-03   3.0 0.39000 0.03809710
## 23 1e-02   3.0 0.39000 0.03809710
## 24 1e-01   3.0 0.38375 0.03729108
## 25 1e+00   3.0 0.22375 0.03972562
## 26 5e+00   3.0 0.23125 0.01692508
## 27 1e+01   3.0 0.23625 0.02389938
## 28 1e+02   3.0 0.24000 0.03425801
## 29 1e-03   4.0 0.39000 0.03809710
## 30 1e-02   4.0 0.39000 0.03809710
## 31 1e-01   4.0 0.38625 0.03653860
## 32 1e+00   4.0 0.22625 0.03304563
## 33 5e+00   4.0 0.23250 0.02220485
## 34 1e+01   4.0 0.23250 0.03016160
## 35 1e+02   4.0 0.24625 0.03634805
```

Observation: -Here we can see the error is minimum when cost is 1 and gamma=0.5. So our model will be best when cost=1 and gamma=0.5(depending upon the cost which we have tried on , can not say in general)

```
R.OJ.best.mod=svm(Purchase~ ., data = train, kernel = "radial", cost = 1,
gamma=0.5, scale = FALSE, )
summary(R.OJ.best.mod)

##
## Call:
## svm(formula = Purchase ~ ., data = train, kernel = "radial", cost = 1,
##     gamma = 0.5, , scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##
## Number of Support Vectors:  544
##
##  ( 287 257 )
##
##
## Number of Classes:  2
```

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

```r
set.seed(100)
# Prediciting the class for our training dataset with radial kernel and best model
tune_R_pred_train=predict(R.OJ.best.mod, train)
tune_R_pred_train[1:10] # Looking at the first 10 prediction made by our model in the training dataset with radial kernel and best model
```

```
##  503  985 1004  919  470  823  838  903 1031  183
##   MM   CH   MM   MM   CH   CH   MM   CH   CH   CH
## Levels: CH MM
```

```r
set.seed(100)
# Confusion matrix
table(tune_R_pred_train, train$Purchase)
```

```
##
## tune_R_pred_train  CH  MM
##                CH 449  88
##                MM  39 224
```

Observation: -Looking at the confusion matrix we see that the training error rate is: (88+39)/800= 0.15875 i.e 15.875% with radial kernel and best model.

Now predicting the test data set using this new best model with radial kernel

```r
set.seed(100)
tune_R_pred_test=predict(R.OJ.best.mod, test)
tune_R_pred_test[1:10]   # Looking at the first 10 prediction made by our model in the Test dataset
```

```
##  3  5  7  8 20 25 27 29 33 36
## CH CH CH MM CH CH CH CH MM MM
## Levels: CH MM
```

```r
set.seed(100)
# Confusion matrix
table(tune_R_pred_test, test$Purchase)
```

```
##
## tune_R_pred_test  CH  MM
##               CH 137  47
##               MM  28  58
```

Observation: -Looking at the confusion matrix we see that the test error rate is: (47+28)/270= 0.2777778 i.e 27.77778% with radial kernel and best model.

Conclusion: From above we see that the training error rate went up from 9.375% (i)to 15.875% (ii) when using the best model and test error rate went up from 25.92593% (i)to

27.77778%. So we can say that by doing model tuning we did not get our new model as good model for predicting the test set compared to original.

Comprasion

comparing the best linear and best radial model, we conclude that best linear model was more nicer then best radial for predicting this test dataset because best linear model has test error rate: 18.14815% only but the best radial model has the test error rate of 27.77778%

(f) Now repeat again, with polynomial basis kernels, with different values of degree and cost. Comment on your results. Which approach (kernel) seems to give the best results on this data?

Polynomial

```
set.seed(100)
library(e1071)

# Fitting a polynomial model with cost=5 and degree=3
Poly.OJ.svm = svm(Purchase ~ ., data =train, kernel = "polynomial", degree=3,
cost = 5, scale = FALSE)
summary(Poly.OJ.svm)

##
## Call:
## svm(formula = Purchase ~ ., data = train, kernel = "polynomial",
##     degree = 3, cost = 5, scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  polynomial
##        cost:  5
##      degree:  3
##      coef.0:  0
##
## Number of Support Vectors:  226
##
##  ( 115 111 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

Summary tells us that, the polynomial kernel was used with cost=5, degree=3 and that there were 226 support vectors, out of which 115 belongs to one class and 111 belongs to the other class. Number of classes are two with levels CH and MM

```
set.seed(100)
# Prediciting the class for our training dataset with polynomial kernel
poly_pred_train=predict(Poly.OJ.svm, train)
poly_pred_train[1:10] # Looking at the first 10 prediction made by our model
in the training dataset with polynomial kernel
```

```
##  503  985 1004  919  470  823  838  903 1031  183
##   CH   CH   MM   CH   CH   CH   MM   CH   CH   CH
## Levels: CH MM
```

```
set.seed(100)
# Confusion matrix
table(poly_pred_train, train$Purchase)
```

```
##
## poly_pred_train  CH  MM
##              CH 429  71
##              MM  59 241
```

Observation: -Looking at the confusion matrix we see that the training error rate is: (71+59)/800= 0.1625 i.e 16.25%

now predicting for the test data

```
set.seed(100)
poly_pred_test=predict(Poly.OJ.svm, test)
poly_pred_test[1:10]   # Looking at the first 10 prediction made by our model
in the Test dataset
```

```
##  3  5  7  8 20 25 27 29 33 36
## CH CH CH CH CH CH CH CH MM CH
## Levels: CH MM
```

```
set.seed(100)
# Confusion Matrix
table(poly_pred_test, test$Purchase)
```

```
##
## poly_pred_test  CH  MM
##             CH 143  26
##             MM  22  79
```

Observation: -Looking at the confusion matrix we see that the test error rate is: (26+22)/270= 0.1777778 i.e 17.77778% with radial kernel.

Now lets try to find the best model with polynomial kernel by trying different values of cost and degree

```r
set.seed(100)
# perform cross-validation
poly.OJ.tune <- tune(
  svm,                    # SVM function
  Purchase~.,                  # Formula for the model
  data = train,             # my training data frame
  kernel = "polynomial",      # polynomial kernel is used
  ranges=list(cost=c(0.001, 0.01, 0.1,
1,5,10,50,100),degree=c(0.25,0.33,0.5,1,2,3,4))  # Range of cost values and
degree values
)
summary(poly.OJ.tune)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost degree
##     1      1
##
## - best performance: 0.16625
##
## - Detailed performance results:
##     cost degree   error dispersion
## 1  1e-03   0.25 0.39000 0.03809710
## 2  1e-02   0.25 0.39000 0.03809710
## 3  1e-01   0.25 0.39000 0.03809710
## 4  1e+00   0.25 0.39000 0.03809710
## 5  5e+00   0.25 0.39000 0.03809710
## 6  1e+01   0.25 0.39000 0.03809710
## 7  5e+01   0.25 0.39000 0.03809710
## 8  1e+02   0.25 0.39000 0.03809710
## 9  1e-03   0.33 0.39000 0.03809710
## 10 1e-02   0.33 0.39000 0.03809710
## 11 1e-01   0.33 0.39000 0.03809710
## 12 1e+00   0.33 0.39000 0.03809710
## 13 5e+00   0.33 0.39000 0.03809710
## 14 1e+01   0.33 0.39000 0.03809710
## 15 5e+01   0.33 0.39000 0.03809710
## 16 1e+02   0.33 0.39000 0.03809710
## 17 1e-03   0.50 0.39000 0.03809710
## 18 1e-02   0.50 0.39000 0.03809710
## 19 1e-01   0.50 0.39000 0.03809710
## 20 1e+00   0.50 0.39000 0.03809710
## 21 5e+00   0.50 0.39000 0.03809710
## 22 1e+01   0.50 0.39000 0.03809710
## 23 5e+01   0.50 0.39000 0.03809710
## 24 1e+02   0.50 0.39000 0.03809710
```

```
## 25 1e-03   1.00 0.39000 0.03809710
## 26 1e-02   1.00 0.38750 0.03908680
## 27 1e-01   1.00 0.17000 0.03827895
## 28 1e+00   1.00 0.16625 0.04411554
## 29 5e+00   1.00 0.17375 0.03928617
## 30 1e+01   1.00 0.17375 0.03928617
## 31 5e+01   1.00 0.17125 0.03438447
## 32 1e+02   1.00 0.17125 0.03729108
## 33 1e-03   2.00 0.39000 0.03809710
## 34 1e-02   2.00 0.38875 0.03972562
## 35 1e-01   2.00 0.31875 0.04686342
## 36 1e+00   2.00 0.19375 0.03019037
## 37 5e+00   2.00 0.17875 0.03866254
## 38 1e+01   2.00 0.17750 0.04031129
## 39 5e+01   2.00 0.17250 0.03574602
## 40 1e+02   2.00 0.17875 0.03910900
## 41 1e-03   3.00 0.39000 0.03809710
## 42 1e-02   3.00 0.37375 0.04387878
## 43 1e-01   3.00 0.28625 0.04226652
## 44 1e+00   3.00 0.18375 0.04210189
## 45 5e+00   3.00 0.17250 0.03525699
## 46 1e+01   3.00 0.17500 0.03333333
## 47 5e+01   3.00 0.19125 0.02503470
## 48 1e+02   3.00 0.20250 0.02874698
## 49 1e-03   4.00 0.39000 0.03809710
## 50 1e-02   4.00 0.37375 0.04387878
## 51 1e-01   4.00 0.31500 0.04479893
## 52 1e+00   4.00 0.22625 0.04803428
## 53 5e+00   4.00 0.20250 0.04158325
## 54 1e+01   4.00 0.19875 0.03508422
## 55 5e+01   4.00 0.19875 0.03087272
## 56 1e+02   4.00 0.19375 0.02841288
```

(Funny event, I have to wait approx 3 min to run this code) Observation: -Here we can see the error is minimum when cost is 1 and degree=1. So our model will be best when cost=1 and gamma=0.5(depending upon the cost which we have tried on , can not say in general)

```
poly.OJ.best.mod=svm(Purchase~ ., data = train, kernel = "polynomial", cost =
1, degree=1, scale = FALSE, )
summary(poly.OJ.best.mod)

##
## Call:
## svm(formula = Purchase ~ ., data = train, kernel = "polynomial",
##     cost = 1, degree = 1, , scale = FALSE)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  polynomial
```

```
##        cost:  1
##      degree:  1
##      coef.0:  0
##
## Number of Support Vectors:  484
##
##  ( 242 242 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

```r
set.seed(100)
# Prediciting the class for our training dataset with polynomial kernel and best model
tune_poly_pred_train=predict(poly.OJ.best.mod, train)
tune_poly_pred_train[1:10] # Looking at the first 10 prediction made by our model in the training dataset with polynomial kernel and best model
```

```
##  503  985 1004  919  470  823  838  903 1031  183
##   CH   CH   MM   CH   CH   CH   MM   CH   CH   CH
## Levels: CH MM
```

```r
set.seed(100)
# Confusion matrix
table(tune_poly_pred_train, train$Purchase)
```

```
##
## tune_poly_pred_train  CH  MM
##                   CH 429  87
##                   MM  59 225
```

Observation: -Looking at the confusion matrix we see that the training error rate is: (87+59)/800= 0.1825 i.e 18.25% with polynomial kernel and best model.

Now predicting the test data set using this new best model with polynomial kernel

```r
set.seed(100)
tune_poly_pred_test=predict(poly.OJ.best.mod, test)
tune_poly_pred_test[1:10]   # Looking at the first 10 prediction made by our model in the Test dataset
```

```
##  3  5  7  8 20 25 27 29 33 36
## CH CH CH CH CH CH CH CH MM CH
## Levels: CH MM
```

```r
set.seed(100)
# Confusion matrix
table(tune_poly_pred_test, test$Purchase)
```

```
##
## tune_poly_pred_test  CH  MM
##                   CH 147  23
##                   MM  18  82
```

Observation: -Looking at the confusion matrix we see that the test error rate is: (23+18)/270= 0.1518519 i.e 15.18519% with polynomial kernel and best model.

Conclusion: From above we see that the training error rate went up from 16.25% (i)to 18.25% (ii) when using the best model and test error rate went down from 17.77778% (i)to 15.18519%. So we can say that by doing model tuning we did get our new model as good model for predicting the test set.

Comprasion

comparing the best linear and best radial model and best polynomial mode, we conclude that best linear model was more nicer then best radial for predicting this test dataset because only but the

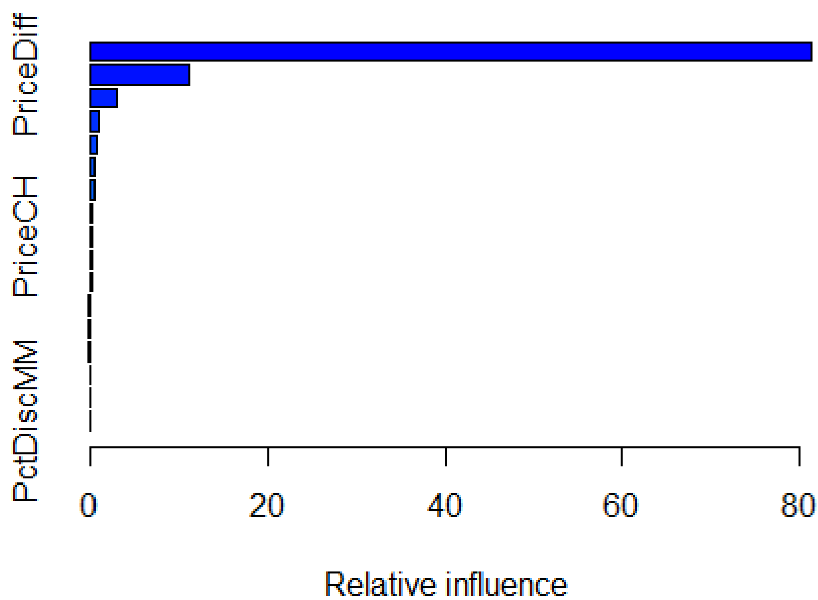
-best linear model has test error rate: 18.14815% -best radial model has the test error rate of 27.77778% -best polynomial model has the test error rate of 15.18519%

(among my given values of cost, gamma, degree)We can say polonomial kernel is best, linear is second best and radial goes last for predicting our test data.

(g) Perform gradient boost (using gbm function in R) on the training set with 1,000 trees for a chosen values of the shrinkage parameter. You may experiment with a range of values of the shrinkage parameter. Answer: Before applying gradient boost, we will first convert data type of our purchase variable[The reference for this is book page no. 174, chapter 4(for exam)]

```
contrasts(OJ$Purchase)

##    MM
## CH  0
## MM  1

library(gbm)

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

## Loaded gbm 2.1.8.1

# Converted all my training data set to binary response
OJ.train = train
OJ.train$Purchase = factor(OJ.train$Purchase, levels=c("CH","MM"),
labels=c(0,1))
OJ.train$Purchase = as.integer(OJ.train$Purchase)-1

# Converted all my Testing data set to binary response
```

```
OJ.test = test
OJ.test$Purchase = factor(OJ.test$Purchase, levels=c("CH","MM"),
labels=c(0,1))
OJ.test$Purchase = as.integer(OJ.test$Purchase)-1

# What I did here is that I first convert the Purchase variable to factor 0,1
from factor CH, MM.
# After that I changed Purchase to numeric so it become 1,2 but I need 0,1 so
subtracted 1 from both

set.seed(100)
#Trying learning rate of 0.001(shrinkage paremeter)
boost.OJ_1 = gbm(Purchase ~ ., data=OJ.train, distribution="bernoulli",
               n.trees=1000, interaction.depth=4, shrinkage=0.001)

#boost.OJ.pred = predict(boost.OJ, newdata=OJ.boost[test.id, ], n.trees=5000,
type="response")
summary(boost.OJ_1)
```



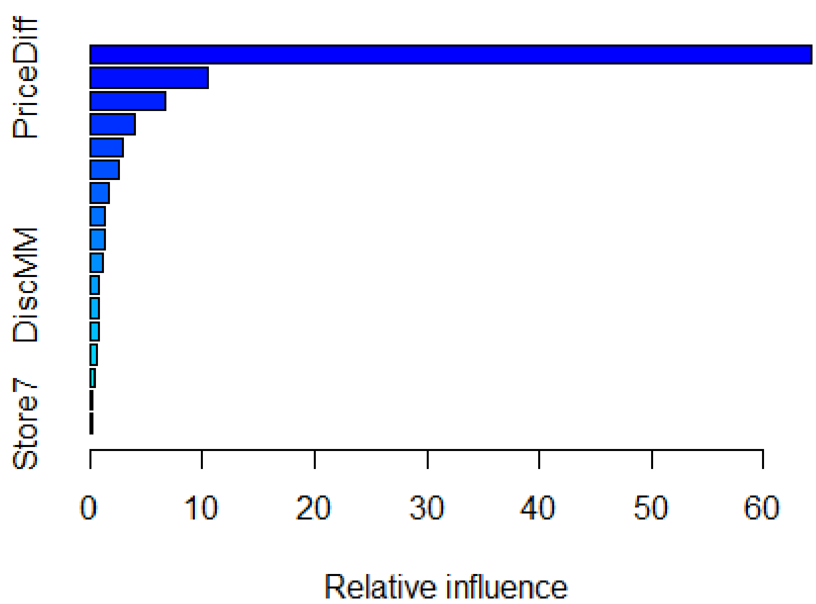

```
##                           var     rel.inf
## LoyalCH               LoyalCH 81.31581802
## PriceDiff           PriceDiff 11.13510966
## ListPriceDiff   ListPriceDiff  3.09025031
## StoreID               StoreID  1.01439548
## SalePriceMM       SalePriceMM  0.76137059
## WeekofPurchase WeekofPurchase  0.54789484
## STORE                   STORE  0.49470665
```

```
## SpecialCH             SpecialCH  0.32338489
## PriceCH                 PriceCH  0.27845690
## SalePriceCH         SalePriceCH  0.26143073
## PriceMM                 PriceMM  0.21336105
## DiscCH                   DiscCH  0.13650070
## Store7                   Store7  0.13593927
## DiscMM                   DiscMM  0.12808395
## SpecialMM             SpecialMM  0.09607851
## PctDiscCH             PctDiscCH  0.04290508
## PctDiscMM             PctDiscMM  0.02431337
```

Observation: We see that LoyalCH and PriceDiff are the most important variables.

Trying different values of the learning rate(Shrinkage parameter)

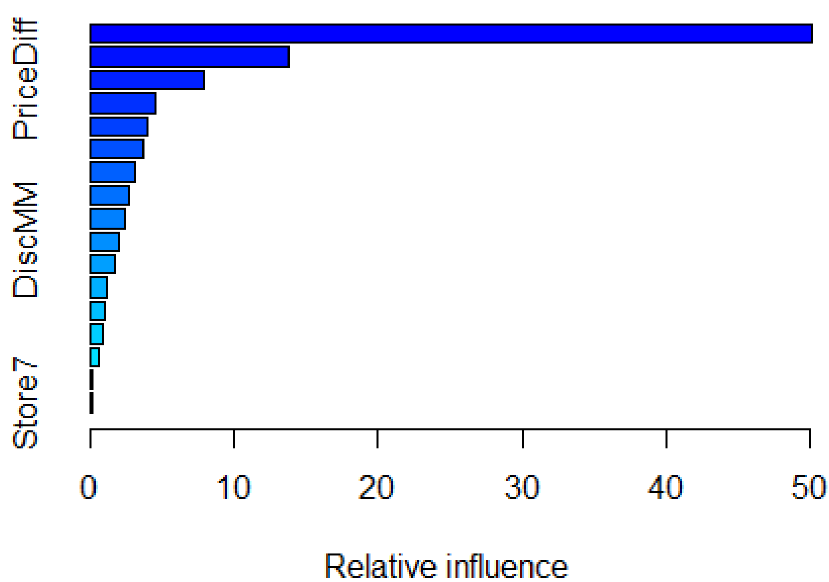
```
#Trying learning rate of 0.01(shrinkage paremeter)
boost.OJ_2=gbm(Purchase~., data = OJ.train,  n.trees = 1000, distribution
="bernoulli",  interaction.depth =4, shrinkage = 0.01)

summary(boost.OJ_2)
```




```
##                                var    rel.inf
## LoyalCH                    LoyalCH 64.2716329
## PriceDiff                PriceDiff 10.5343962
## WeekofPurchase WeekofPurchase  6.6289727
## ListPriceDiff     ListPriceDiff  4.0015017
## StoreID                    StoreID  2.9709493
```

```
## SalePriceMM       SalePriceMM  2.6095022
## STORE                   STORE  1.7452825
## SalePriceCH       SalePriceCH  1.3418927
## PriceCH               PriceCH  1.2258143
## PriceMM               PriceMM  1.0903073
## DiscMM                 DiscMM  0.8312014
## SpecialCH           SpecialCH  0.7394305
## SpecialMM           SpecialMM  0.6875707
## DiscCH                 DiscCH  0.4919267
## PctDiscMM           PctDiscMM  0.3511397
## PctDiscCH           PctDiscCH  0.2580347
## Store7                 Store7  0.2204444
```

```
#Trying learning rate of 0.01(shrinkage paremeter)
boost.OJ_3=gbm(Purchase~., data = OJ.train,  n.trees = 1000, distribution
="bernoulli",  interaction.depth =4, shrinkage = 0.1)

summary(boost.OJ_3)
```




```
##                                var     rel.inf
## LoyalCH                    LoyalCH 50.1068807
## WeekofPurchase WeekofPurchase 13.7551499
## PriceDiff               PriceDiff  7.8611662
## ListPriceDiff       ListPriceDiff  4.5610340
## StoreID                   StoreID  4.0045121
## SalePriceMM           SalePriceMM  3.6932731
## STORE                       STORE  3.0898383
```

```
## PriceCH                PriceCH  2.7594595
## PriceMM                PriceMM  2.4742854
## DiscMM                  DiscMM  2.0401289
## SalePriceCH        SalePriceCH  1.7385279
## SpecialMM            SpecialMM  1.1263269
## SpecialCH            SpecialCH  1.0208753
## DiscCH                  DiscCH  0.8781028
## PctDiscMM            PctDiscMM  0.5774586
## PctDiscCH            PctDiscCH  0.1658227
## Store7                  Store7  0.1471577
```

[Ask her: Can I say as the learning rate increase other variable then LoyalCH are also becoming more and more important each time?]

(h) Which variables appear to be the most important predictors in the boost model? Answer:- LoyalCH variable appears to be the most important predictor from the boost model.

(i) Use the boosting model to predict the response on the test data. Form a confusion matrix. How does this compare with the result SVM obtained? Answer: for predicting we use the modle with learning rate 0.1

```
set.seed(100)
glm.probs=predict(boost.OJ_3 , OJ.test, type = "response")

## Using 1000 trees...

glm.probs[1:10]

##  [1] 0.007514068 0.023404826 0.008723721 0.099361581 0.141581807
0.002850720
##  [7] 0.003545146 0.002283526 0.368595711 0.021847529

glm.pred <- rep("CH", 270)
glm.pred[glm.probs > .5] = "MM"

table(glm.pred, OJ.test$Purchase)

##
## glm.pred   0   1
##       CH 137  27
##       MM  28  78
```

Observation: -Looking at the confusion matrix we see that the test error rate is: (27+28)/270= 0.2037037 i.e 20.37037%

Comparing this boosting model to SVM model

-The bosting model with learning rate 0.1 has thas the test error rate= 20.37037% -best linear model has test error rate: 18.14815% -best radial model has the test error rate of 27.77778% -best polynomial model has the test error rate of 15.18519%

So this boosting model only did better as compared to svm model with radial kernal in our test dataset. With other kernal smv model did much better then 20.370.7%

——————————————————–THE END———————————————————————-

Machine Learning HW-4

Sagar Kalauni

2023-12-10

1) Consider a neural network with two hidden layers: p = 4 input units, 2 units in the first hidden layer, 3 units in the second hidden layer, and a single output.

(a) Draw a picture of the network.

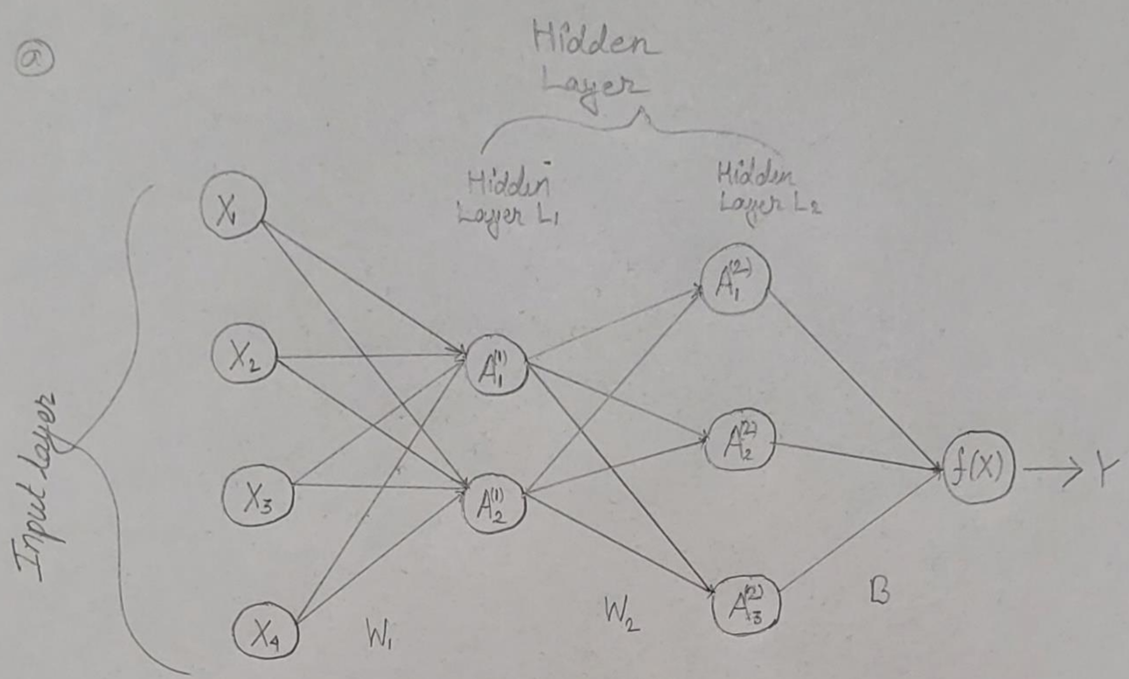
(b) Write out an expression for f(X), assuming ReLU activation functions. Be as explicit as you can!

(c) How many parameters are there? ANSWER:- Please look for the attached figure below:

HW#4

Solution for Q. No. - 1

ⓐ



ⓑ The expression for $f(X)$ is given as

$$f(X) = \beta_0 + \sum \beta_K A_K^{(L)}$$

If our activation function is ReLU (Retified linear unit). Then

$$f(X) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

ⓒ The above figure network has $2+3+1 = 6$ neurons, $[4*2]+[2*3]+[3*1$
$= 8+6+3 = 17$ weights and $2+3+1 = 6$ biases, for a total of
23 learnable parameter

$\therefore$ # of parameters $= 23$. $\square$

2) Consider the Default data. Split the data into 70% training and 30% test.

```r
set.seed(100)
#install.packages("ISLR2")
library(ISLR2)

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

library(nnet)

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

standardize=function(x) {(x-min(x))/(max(x)-min(x))}
Default$income=standardize(Default$income)
Default$balance=standardize(Default$balance)

index=sample(1:nrow(Default), 0.7*nrow(Default))
train=Default[index,]
test=Default[-index,]
```

(a) Fit a neural network using a single hidden layer with 10 units.

```r
set.seed(100)
#install.packages("nnet")
#library(nnet)

NN.fit=nnet(default~., data=train, size=10 )  # For linout:-Default logistic
output units

## # weights:  51
## initial  value 6634.591690
## iter  10 value 772.690204
## iter  20 value 564.802404
## iter  30 value 559.909439
## iter  40 value 558.935767
## iter  50 value 558.251088
## iter  60 value 557.674311
## iter  70 value 557.266359
## iter  80 value 556.877075
## iter  90 value 556.663528
## iter 100 value 556.331984
## final  value 556.331984
## stopped after 100 iterations

set.seed(100)
test_prob=predict(NN.fit, test)

test_pred=rep("No", nrow(test))
test_pred[test_prob>0.5]="Yes"

table(test_pred, test$default)
```

```
##
## test_pred   No  Yes
##       No  2900   59
##       Yes   15   26
```

Observation: The test Accuracy of the Neural network with single hidden layer having 10 units in the test data set is:$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} = \frac{2900+26}{2900+59+15+26} = 0.9753333$

```
set.seed(100)
# Fit a linear logistic regression model
logistic_model=glm(
  formula = default ~ income + balance + student,
  data = train,
  family = binomial
)
glm_test_prob=predict(logistic_model, newdata = test)

glm_test_pred=rep("No", nrow(test))
glm_test_pred[glm_test_prob>0.5]="Yes"

table(glm_test_pred, test$default)

##
## glm_test_pred   No  Yes
##           No  2909   70
##           Yes    6   15
```

Observation: The test Accuracy of the Logistic regression model in the test data set is:$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} = \frac{2909+15}{2909+70+6+15} = 0.9746667$

(b) Compare the classification performance of your model with that of linear logistic regression. ANSWER: Both are doing comparatively same but neural network with single hidden layer has slight more accuracy then logistic regression model in our test data.

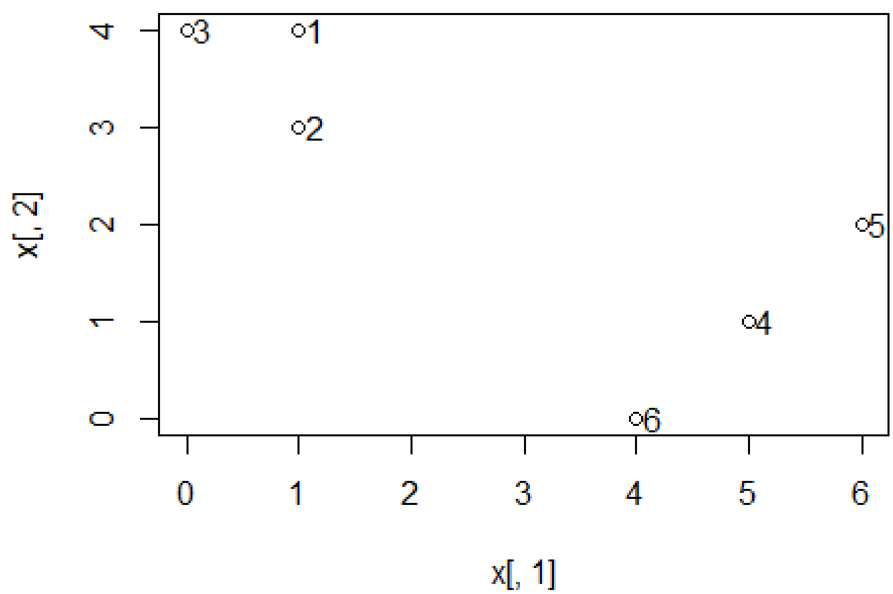
3) In this problem, you will perform K-means clustering manually, with $K = 2 $, on a small example. The observations are as follows.

```
mydata <- data.frame(
  Obs = c(1, 2, 3, 4, 5, 6),
  X1 = c(1, 1, 0, 5, 6, 4),
  X2 = c(4, 3, 4, 1, 2, 0)
)
print(mydata)

##   Obs X1 X2
## 1   1  1  4
## 2   2  1  3
## 3   3  0  4
## 4   4  5  1
```

```
## 5   5  6  2
## 6   6  4  0
```

(a) Sketch the observations.

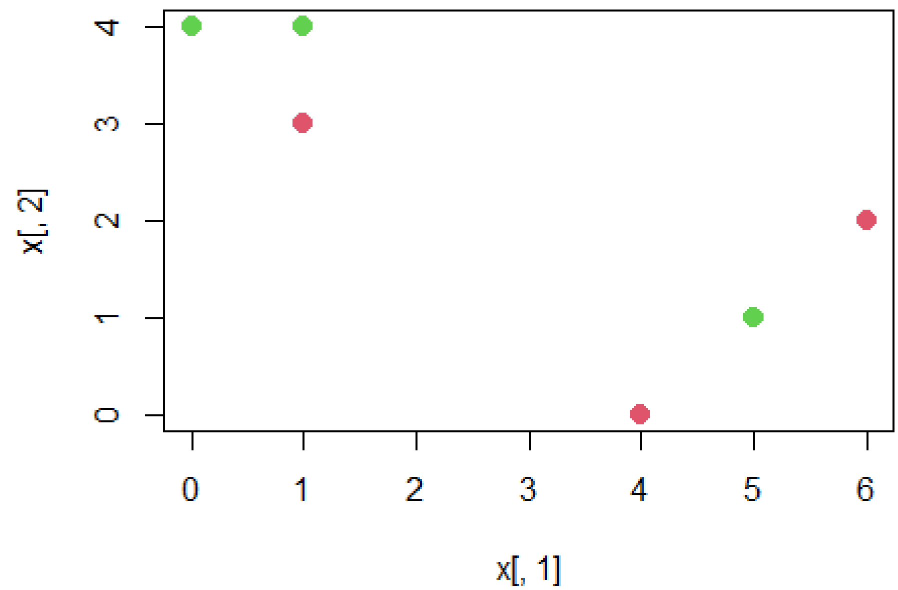
```
x <- cbind(c(1, 1, 0, 5, 6, 4), c(4, 3, 4, 1, 2, 0))
plot(x[,1], x[,2])
text(mydata$X1+0.15, mydata$X2, 1:6)
```




```
# Here I am showing each data point with its observation number in the plot,
+0.15 is added not to concide the label and data in one, so at the same y-
distance and little more x-distance, I am showing my label of the data point.
```

(b) Randomly assign a cluster label to each observation.

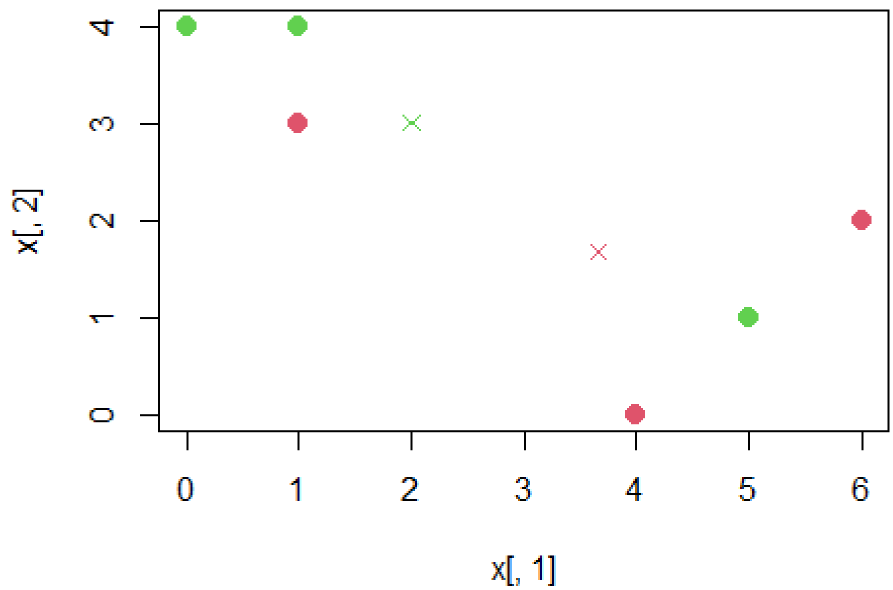
```
set.seed(100)
labels=sample(2, nrow(x), replace = T)
labels

## [1] 2 1 2 2 1 1

plot(x[, 1], x[, 2], col = (labels + 1), pch = 20, cex = 2)
```



(c) Compute the centroid for each cluster. ANSWER:- We compute the centroid of red cluster as $ {x}*{11} = (1 + 4 + 6) = $ and $ {x}*{12} = (3 + 0 + 2) = $

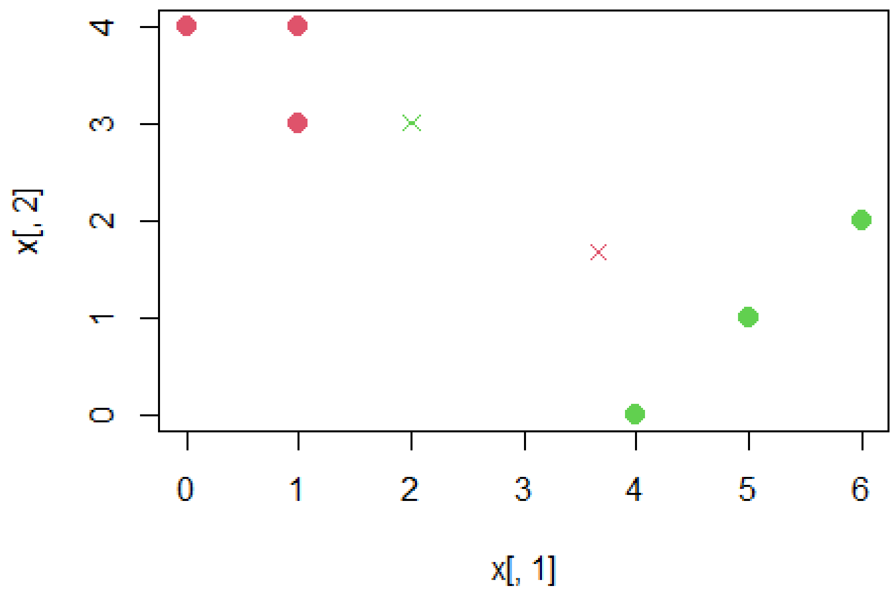
and the centroid of the green cluster as: $ {x}*{21} = (0 + 1 + 5) = 2 $ and $ {x}*{22} = (4 + 4 + 1) = 3 $

```
set.seed(100)
centroid1 <- c(mean(x[labels == 1, 1]), mean(x[labels == 1, 2]))
centroid2 <- c(mean(x[labels == 2, 1]), mean(x[labels == 2, 2]))
centroid1

## [1] 3.666667 1.666667

centroid2

## [1] 2 3

plot(x[,1], x[,2], col=(labels + 1), pch = 20, cex = 2)
points(centroid1[1], centroid1[2], col = 2, pch = 4)
points(centroid2[1], centroid2[2], col = 3, pch = 4)
```



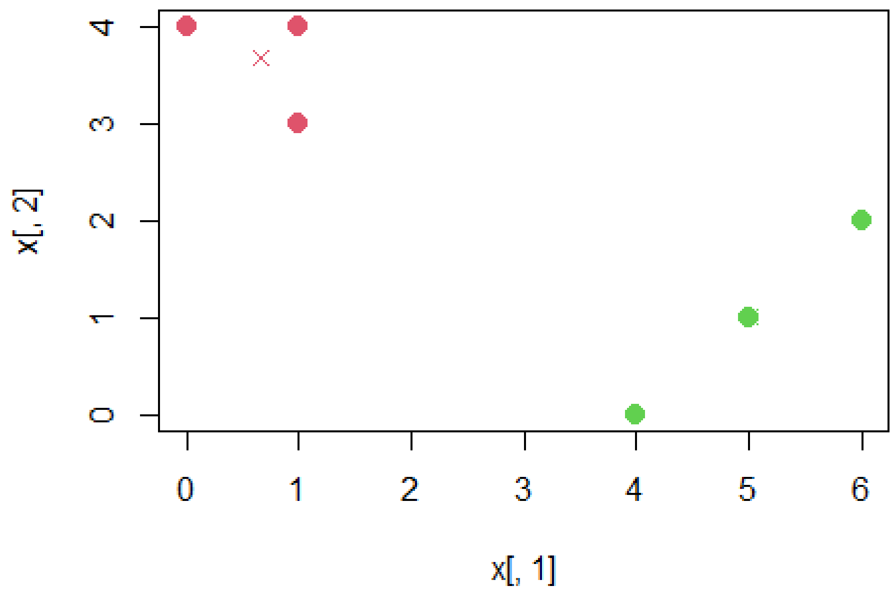
(d) Assign each observation to the centroid to which it is closest, in terms of Euclidean distance. Report the cluster labels for each observation.

```
labels <- c(1, 1, 1, 2, 2, 2)
plot(x[, 1], x[, 2], col = (labels + 1), pch = 20, cex = 2)
points(centroid1[1], centroid1[2], col = 2, pch = 4)
points(centroid2[1], centroid2[2], col = 3, pch = 4)
```



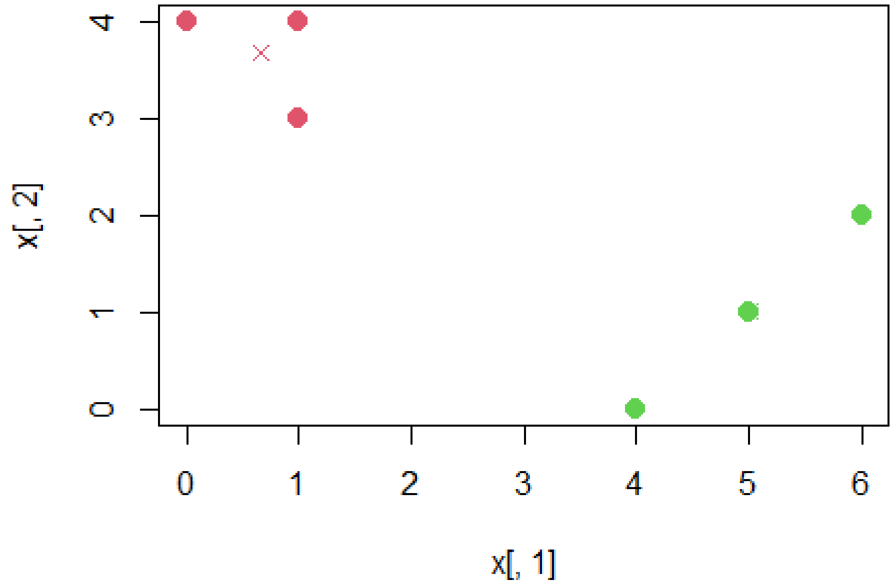
(e) Repeat (c) and (d) until the answers obtained stop changing. Answer:-We compute the centroid of red cluster as $ {x}*{11} = (0 + 1 + 1) = $ and $ {x}*{12} = (3 + 4 + 4) = $ and the centroid of the green cluster as: $ {x}*{21} = (4 + 5 + 6) = 5 $ and $ {x}*{22} = (0 + 1 + 2) = 1 $

```
set.seed(100)
centroid1 <- c(mean(x[labels == 1, 1]), mean(x[labels == 1, 2]))
centroid2 <- c(mean(x[labels == 2, 1]), mean(x[labels == 2, 2]))
centroid1

## [1] 0.6666667 3.6666667

centroid2

## [1] 5 1

plot(x[,1], x[,2], col=(labels + 1), pch = 20, cex = 2)
points(centroid1[1], centroid1[2], col = 2, pch = 4)
points(centroid2[1], centroid2[2], col = 3, pch = 4)
```



Re- asagining

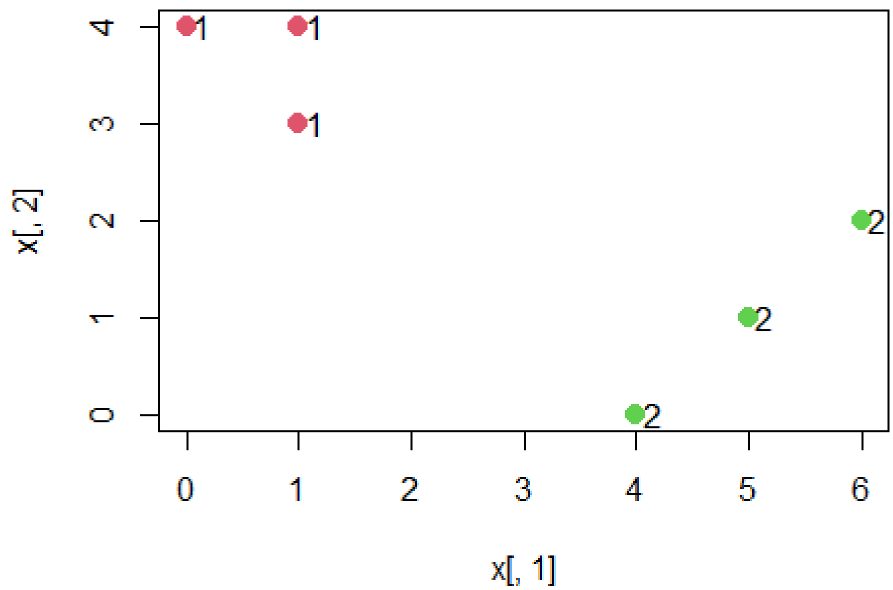
```
labels <- c(1, 1, 1, 2, 2, 2)
plot(x[, 1], x[, 2], col = (labels + 1), pch = 20, cex = 2)
points(centroid1[1], centroid1[2], col = 2, pch = 4)
points(centroid2[1], centroid2[2], col = 3, pch = 4)
```



If we assign each observation to the centroid to which it is closest, nothing changes, so the algorithm is terminated at this step.

(f) In your plot from (a), label the observations according to the final cluster labels obtained.

```
plot(x[, 1], x[, 2], col=(labels + 1), pch = 20, cex = 2)
text(x[, 1]+0.15, x[, 2], labels)
```



4. In this problem, you consider the gene expression data (Khan, in ISLR), and then perform clustering on the data.

```
# Application to Gene Expression Data
set.seed(500)
library(ISLR)

##
## Attaching package: 'ISLR'

## The following object is masked _by_ '.GlobalEnv':
##
##     Default

## The following objects are masked from 'package:ISLR2':
##
##     Auto, Credit

names(Khan)

## [1] "xtrain" "xtest"  "ytrain" "ytest"

dim(Khan$xtrain)

## [1]   63 2308

dim(Khan$xtest)

## [1]   20 2308
```

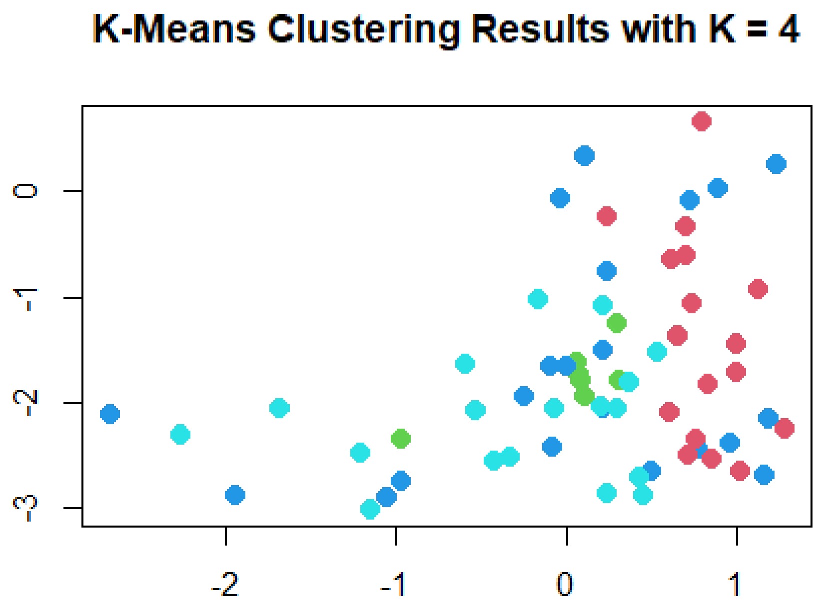
```r
length(Khan$ytrain)

## [1] 63

length(Khan$ytest)

## [1] 20

table(Khan$ytrain)

##
##  1  2  3  4
##  8 23 12 20

table(Khan$ytest)

##
## 1 2 3 4
## 3 6 6 5

dat=data.frame(x=Khan$xtrain, y=as.factor(Khan$ytrain))
```

(a) Perform K-means clustering of the "xtrain" with K = 4. How well do the clusters that you obtained in K-means clustering compare to the true class labels ("ytrain")?

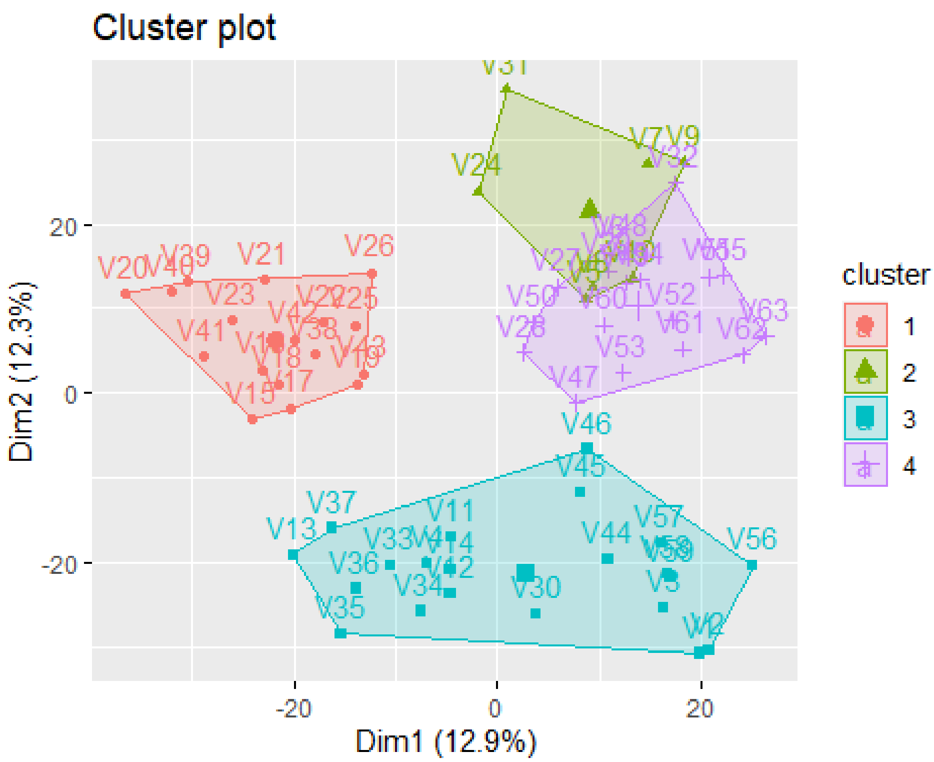
```r
set.seed(500)
khan_clust=kmeans(Khan$xtrain,centers=4)
khan_clust$cluster

##  V1  V2  V3  V4  V5  V6  V7  V8  V9 V10 V11 V12 V13 V14 V15 V16 V17 V18
V19 V20
##   3   3   3   3   2   4   2   2   2   2   3   3   3   3   1   1   1   1
1   1
## V21 V22 V23 V24 V25 V26 V27 V28 V29 V30 V31 V32 V33 V34 V35 V36 V37 V38
V39 V40
##   1   1   1   2   1   1   4   4   4   3   2   4   3   3   3   3   3   1
1   1
## V41 V42 V43 V44 V45 V46 V47 V48 V49 V50 V51 V52 V53 V54 V55 V56 V57 V58
V59 V60
##   1   1   1   3   3   3   4   4   4   4   4   4   4   4   4   3   3   3
3   4
## V61 V62 V63
##   4   4   4

set.seed(500)
library(factoextra)

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

## Loading required package: ggplot2

## Warning: package 'ggplot2' was built under R version 4.3.1
```

```
## Welcome! Want to learn more? See two factoextra-related books at
https://goo.gl/ve3WBa
```

```r
plot(Khan$xtrain, col = (khan_clust$cluster + 1),
main = "K-Means Clustering Results with K = 4",
xlab = "", ylab = "", pch = 20, cex = 2)
```



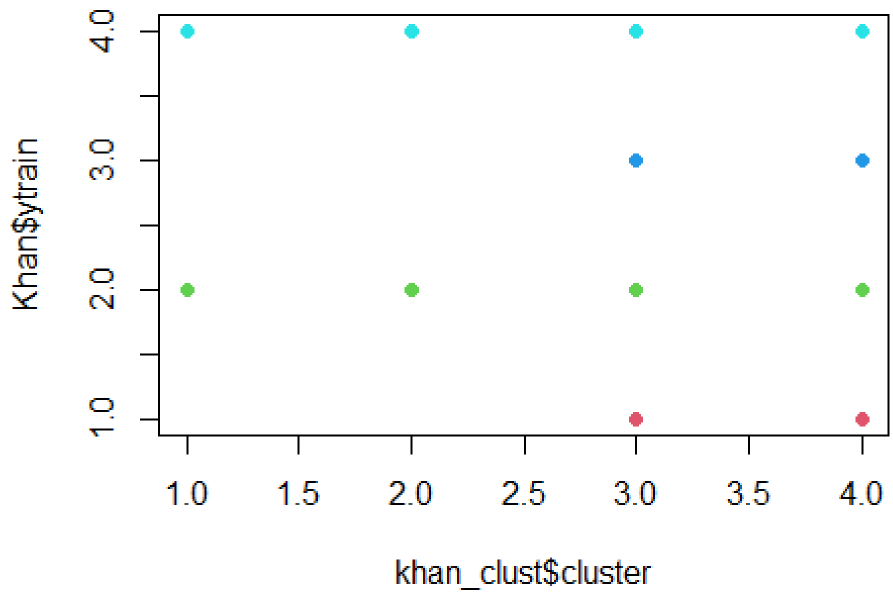

```r
fviz_cluster(list(data=Khan$xtrain,cluster=khan_clust$cluster))
```




```r
set.seed(500)
table(khan_clust$cluster,Khan$ytrain)

##
##     1 2 3 4
##   1 0 9 0 8
##   2 0 5 0 2
##   3 4 8 3 6
##   4 4 1 9 4
```

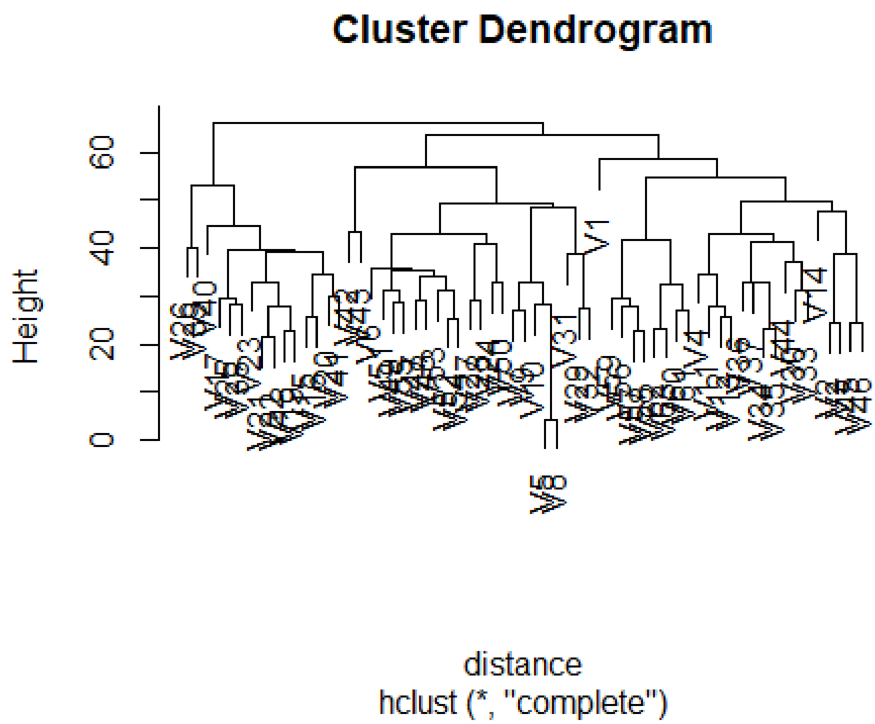
Observation: the clustering done by kmean clustering algorithm has accuracy of:
(0+5+3+4)/63 = 0.1904762 i.e 12.69%. so we can say it performs really poor in clustering.

```r
set.seed(500)
plot(khan_clust$cluster,Khan$ytrain, col=Khan$ytrain+1, pch=19)
```



(b) Using hierarchical clustering with complete linkage and Euclidean distance, cluster the states.

```
set.seed(500)
distance=dist(dat,method="euclidean")
cc=hclust(distance,method="complete")
plot(cc)
```

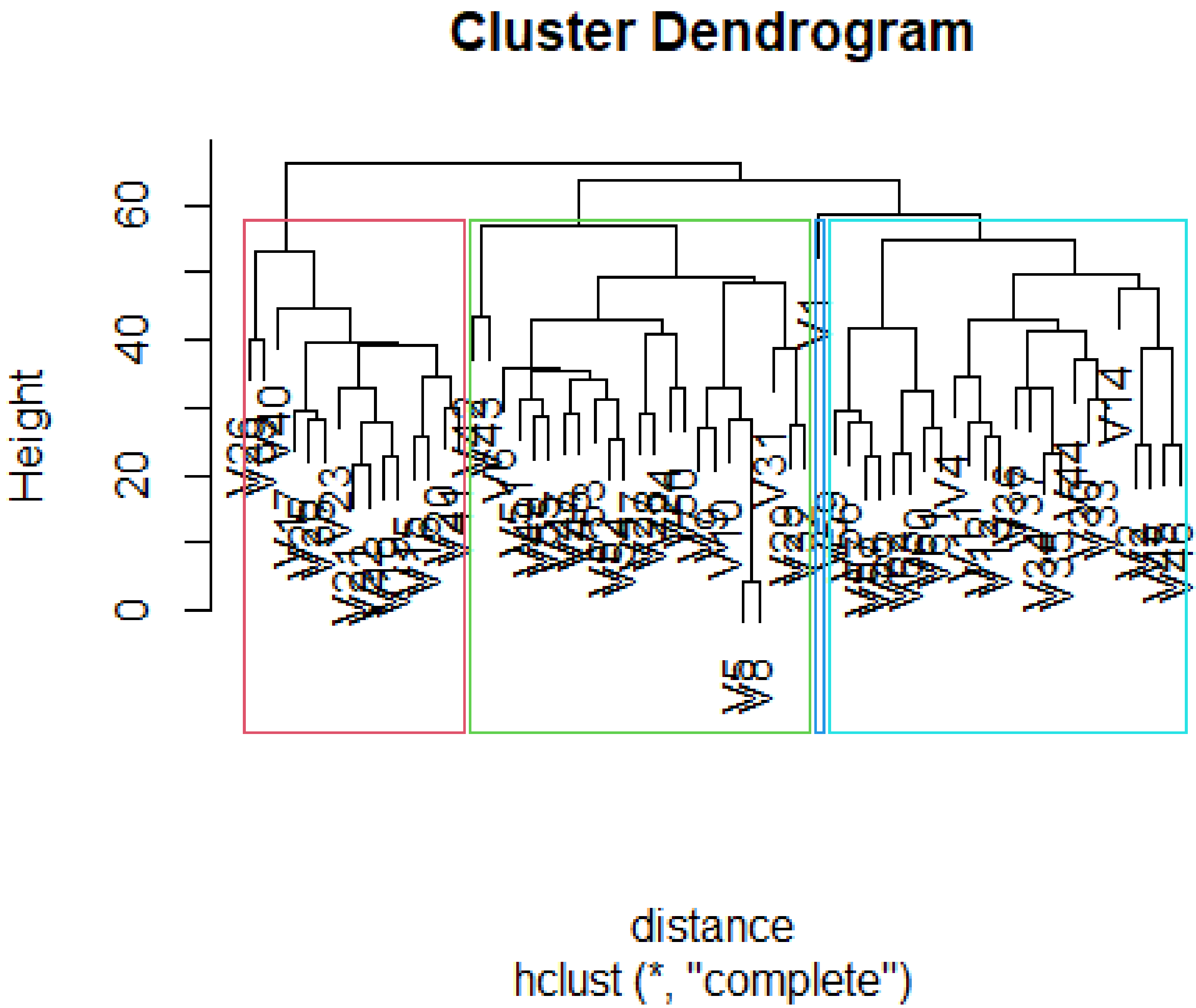


(c) Cut the dendrogram at a height that results in 4 distinct clusters.

```
set.seed(100)
cutree(cc, 4)

##  V1  V2  V3  V4  V5  V6  V7  V8  V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 
V19 V20 
##   1   2   2   2   3   3   3   3   3   3   2   2   2   2   4   4   4   4 
4   4 
## V21 V22 V23 V24 V25 V26 V27 V28 V29 V30 V31 V32 V33 V34 V35 V36 V37 V38 
V39 V40 
##   4   4   4   3   4   4   3   3   3   2   3   3   2   2   2   2   2   4 
4   4 
## V41 V42 V43 V44 V45 V46 V47 V48 V49 V50 V51 V52 V53 V54 V55 V56 V57 V58 
V59 V60 
##   4   3   3   2   2   2   3   3   3   3   3   3   3   3   3   2   2   2 
2   2 
## V61 V62 V63 
##   2   2   2
```

If you want to vissually look the cut point and look at the cluster formed

```
plot(cc)
rect.hclust(cc, k = 4, border = 2:5)
```



I zoomed this picture for you, clearly you can see now you are remaining with only 4 cluster and one of the cluster has V1 as the only entity on it.