STAT 6340 Mini Project 6

Matthew Lynn

April 24, 2019

Section 1

Section 2 is coded in Section 1

Question 1

> ** Consider the data stored in cereal.csv file posted on eLearning. It contains measurements on 8 variables for 43 breakfast cereals.**

```
cereal = read.csv("cereal.csv", header = T)
```

(a)

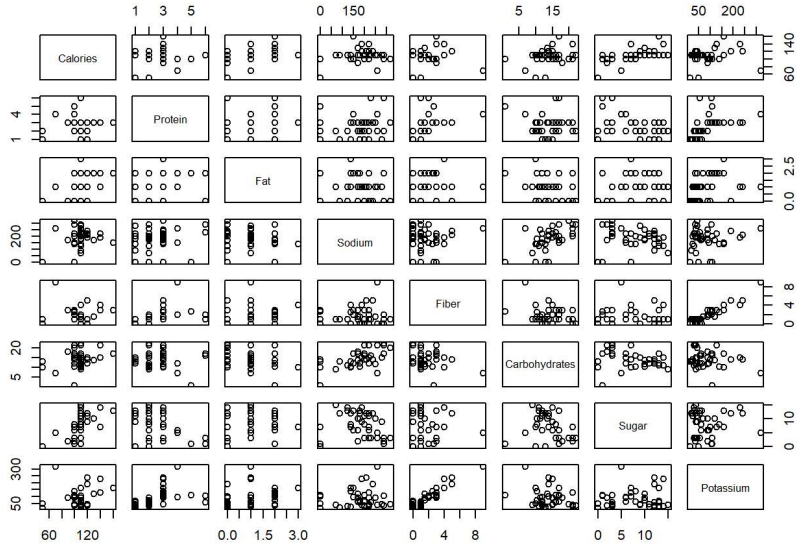
> **Perform an exploratory analysis of the data.**

We perform simple exploratory data analysis on the dataset cereal

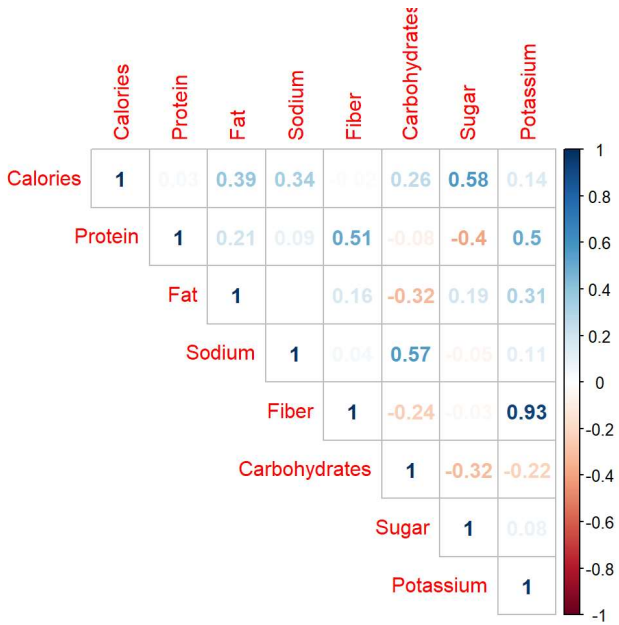
```
str(cereal) # To see what data types we are messing with
```

```
## 'data.frame':    43 obs. of  11 variables:
##  $ Brand        : Factor w/ 43 levels "ACCheerios","AllBran",..: 1 6 7 10 17 19 21 23 25 29
...
##  $ Manufacturer : Factor w/ 3 levels "G","K","Q": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Group        : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Calories     : int  110 110 110 110 110 110 110 110 100 130 ...
##  $ Protein      : int  2 6 1 1 1 3 2 2 2 3 ...
##  $ Fat          : int  2 2 1 1 1 1 1 1 1 2 ...
##  $ Sodium       : int  180 290 180 180 280 250 260 180 220 170 ...
##  $ Fiber        : num  1.5 2 0 0 0 1.5 0 0 2 1.5 ...
##  $ Carbohydrates: num  10.5 17 12 12 15 11.5 21 12 15 13.5 ...
##  $ Sugar        : int  10 1 13 13 9 10 3 12 6 10 ...
##  $ Potassium    : int  70 105 55 65 45 90 40 55 90 120 ...
```

```
# Let us drop Brand, Manufacturer, and Group. They do not seem to benefit our
# analysis and the question mentions 8 variables so dropping these makes sense.
data = subset(cereal, select = -c(Brand, Manufacturer, Group))
pairs(subset(data)) # are there any obvious relations
```




```
# a correlation matrix to measure the relations, need to drop the string data types
corrplot::corrplot(cor(data), method = "number", type = "upper")
```



(b)

Do you think standardizing the variables before clustering would be a good idea?

Almost always this is a good idea. We see from above that our data consists of many different types of macro and micro nutrients which are all measured differently.

(c)

Would you use metric-based or correlation-based distance to cluster the cereals?

Both methods have merit here. We may seet to use metric-based for pairing nutrients that have similar amounts. On the other hand, we may want to pair nutrients that have some relation. We could proceed with both methods and compare.

(d)

Regardless of your answers in (b) and (c), standardize the variables and hierarchically cluster the cereals using complete linkage and Euclidean distance. Display the results using a dendogram. How many clusters would you choose?

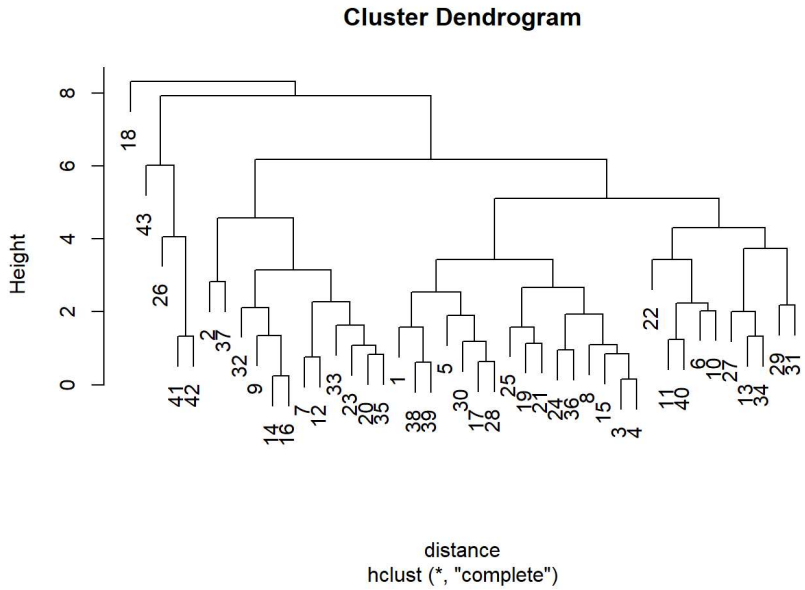
Below, cut at k = 3 seems to give us the best matching. This makes sense since there are 3 manufacturers.

```
set.seed(1)
data_scaled <- scale(data)

distance <- dist(data_scaled, method = 'euclidean')

hc.complete <- hclust(distance, method = 'complete')

plot(hc.complete)
```




```
# We cut at as it gave us the best group matching
cut3 = cutree(hc.complete, 3)
rbind(cut3, cereal$Group)
```

```
##      1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
## cut3 1 1 1 1 1 1 1 1 1  1  1  1  1  1  1  1  2  1  1  1  1  1  1  1  3
##      1 1 1 1 1 1 1 1 1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2
##      27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
## cut3  1  1  1  1  1  1  1  1  1  1  1  1  1  1  3  3  3
##       2  2  2  2  2  2  2  2  2  2  2  3  3  3  3  3  3
```

```
mean(cut3 == cereal$Group)
```

```
## [1] 0.4883721
```

(e)

Repeat (d) using K-means clustering with K = 2, 3, and 4. Which value of K would you choose?

From below, we would choose k = 2 due to its higher accuracy

```
#K-Means Clustering
set.seed(1)

#2 clusters
k2 = kmeans(data_scaled,2,nstart=50)$cluster
rbind(k2, cereal$Group)
```

```
##    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
## k2 1 2 1 1 1 1 1 1 1  2  2  1  2  2  1  2  1  2  1  1  1  2  1  1  1  1  2
##    1 1 1 1 1 1 1 1 1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2  2  2
##    28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
## k2  1  2  1  2  1  1  2  1  1  2  1  1  2  1  1  2
##     2  2  2  2  2  2  2  2  2  2  2  3  3  3  3  3  3
```

```
mean(k2 == cereal$Group)
```

```
## [1] 0.4186047
```

```
#3 clusters
k3 = kmeans(data_scaled,3,nstart=50)$cluster
rbind(k3, cereal$Group)
```

```
##    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
## k3 2 1 2 2 2 2 3 2 3  1  1  3  1  3  2  3  2  1  2  3  2  1  3  2  2  3  1
##    1 1 1 1 1 1 1 1 1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2  2
##    28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
## k3  3  1  2  1  3  3  1  3  2  3  2  2  1  3  3  1
##     2  2  2  2  2  2  2  2  2  2  3  3  3  3  3  3
```

```
mean(k3 == cereal$Group)
```

```
## [1] 0.2790698
```

```
#4 clusters
k4 = kmeans(data_scaled,4,nstart=50)$cluster
rbind(k4, cereal$Group)
```

```
##    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
## k4 2 4 2 2 2 2 4 2 4  3  3  4  3  4  2  4  2  3  2  4  2  3  4  2  2  1  3
##    1 1 1 1 1 1 1 1 1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2  2  2
##    28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43
## k4  4  3  2  3  4  4  3  4  2  4  2  2  3  1  1  1
##     2  2  2  2  2  2  2  2  2  2  3  3  3  3  3  3
```

```
mean(k4 == cereal$Group)
```

```
## [1] 0.1627907
```

(f)

Compare the results from (d) and (e). Which method would you recommend?

It appears that hierarchical cluster with cut k = 3 gives us the best group matching.

```
c(mean(cut3 == cereal$Group), "Hierarchical Clustering with K=3")
```

```
## [1] "0.488372093023256"               "Hierarchical Clustering with K=3"
```

```
c(mean(k2 == cereal$Group), "Kmeans Clustering with k = 2")
```

```
## [1] "0.418604651162791"            "Kmeans Clustering with k = 2"
```

```
c(mean(k3 == cereal$Group), "Kmeans Clustering with k = 3")
```

```
## [1] "0.27906976744186"            "Kmeans Clustering with k = 3"
```

```
c(mean(k4 == cereal$Group), "Kmeans Clustering with k = 4")
```

```
## [1] "0.162790697674419"           "Kmeans Clustering with k = 4"
```

Question 2

> ** Consider the Caravan dataset from the previous project. Standardize the predictors and split the data into training and test sets just as in the previous project.**

```
library(ISLR)
caravan = Caravan
s.caravan = scale(caravan[, -86]) #86 column is purchase
c(var(caravan[,1]),var(caravan[,2]),var(s.caravan[,1]),var(s.caravan[,2]))
```

```
## [1] 165.0378474   0.1647078   1.0000000   1.0000000
```

```
test = 1:1000
train = (-test)
xtrain = caravan[train,]
xtest = caravan[test,]
ytrain = caravan$Purchase[train]
ytest = caravan$Purchase[test]
```

(a)

> **Fit a tree to the training data. Summarize the results. Unless the number of terminal nodes is large, display the tree graphically and explicitly describe the regions corresponding to the terminal nodes that provide a partition of the predictor space (i.e., provide expressions for the regions R1, . . . , RJ). Report the confusion matrix and error rate for the unpruned tree based on the test data.**

```
fit = tree::tree(Purchase ~ ., data = caravan[train,])
fit
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##  1) root 4822 2187.0 No ( 0.94007 0.05993 )
##    2) PPERSAUT < 5.5 2853  664.7 No ( 0.97511 0.02489 ) *
##    3) PPERSAUT > 5.5 1969 1370.0 No ( 0.88928 0.11072 )
##      6) PBRAND < 2.5 973  466.7 No ( 0.93525 0.06475 ) *
##      7) PBRAND > 2.5 996  861.2 No ( 0.84438 0.15562 )
##       14) MOPLLAAG < 4.5 510  536.9 No ( 0.78039 0.21961 ) *
##       15) MOPLLAAG > 4.5 486  290.6 No ( 0.91152 0.08848 ) *
```

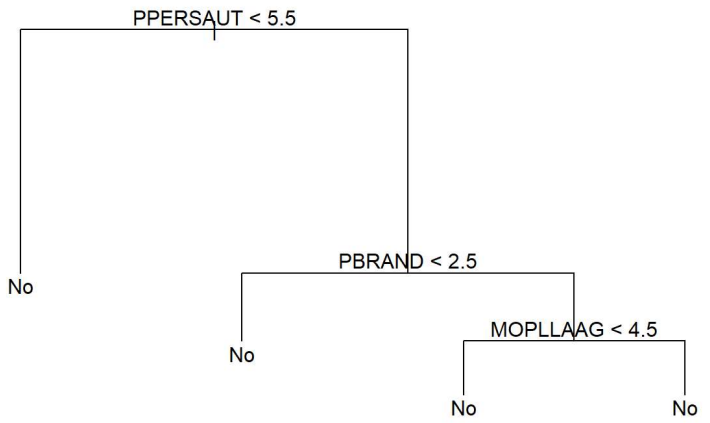
```
summary(fit)
```

```
##
## Classification tree:
## tree::tree(formula = Purchase ~ ., data = caravan[train, ])
## Variables actually used in tree construction:
## [1] "PPERSAUT" "PBRAND"   "MOPLLAAG"
## Number of terminal nodes:  4
## Residual mean deviance:  0.4066 = 1959 / 4818
## Misclassification error rate: 0.05993 = 289 / 4822
```

```
tree.pred = predict(fit,caravan[test,], type='class')
caret::confusionMatrix(tree.pred,ytest, positive = 'Yes')
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  No Yes
##        No  941  59
##        Yes   0   0
##
##               Accuracy : 0.941
##                 95% CI : (0.9246, 0.9548)
##     No Information Rate : 0.941
##     P-Value [Acc > NIR] : 0.5346
##
##                  Kappa : 0
##  Mcnemar's Test P-Value : 4.321e-14
##
##             Sensitivity : 0.000
##             Specificity : 1.000
##          Pos Pred Value :   NaN
##          Neg Pred Value : 0.941
##              Prevalence : 0.059
##          Detection Rate : 0.000
##    Detection Prevalence : 0.000
##       Balanced Accuracy : 0.500
##
##        'Positive' Class : Yes
##
```

```
plot(fit)
text(fit, pretty = 0)
```




```
unpruned.err = mean(tree.pred != ytest)
```

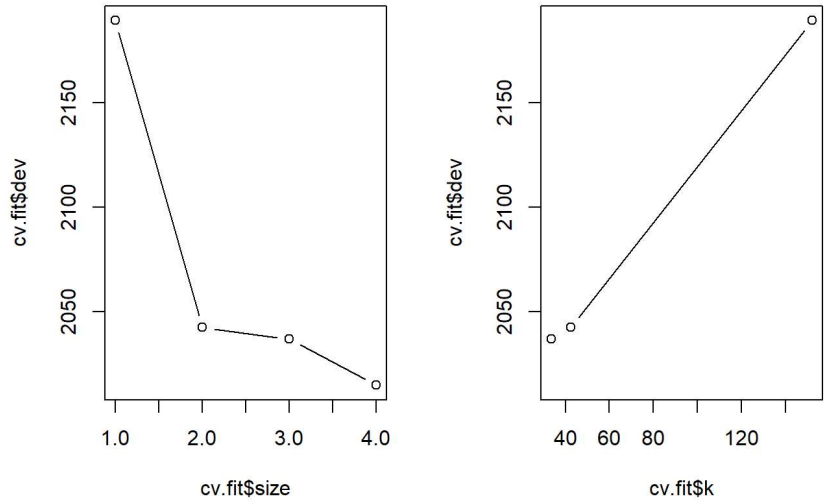
(b)

Use cross-validation to determine whether pruning is helpful and determine the optimal size for the pruned tree. Compare the pruned and un-pruned trees. Report the confusion matrix and error rate for the pruned tree based on the test data. Which predictors seem to be the most important?

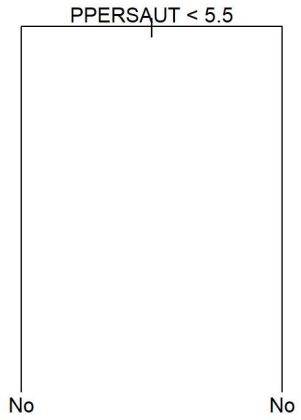
```
set.seed(1)
library(tree)
cv.fit = cv.tree(fit)
cv.fit
```

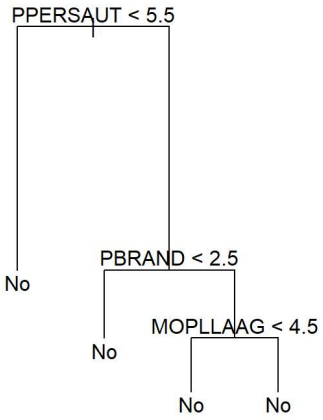
```
## $size
## [1] 4 3 2 1
##
## $dev
## [1] 2014.637 2036.755 2042.274 2189.460
##
## $k
## [1]      -Inf  33.65143  42.52014 151.96279
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

```
par(mfrow = c(1,2))
plot(cv.fit$size, cv.fit$dev, type = "b")
plot(cv.fit$k, cv.fit$dev, type = "b")
```




```
prune.fit = prune.tree(fit, best = 2)
plot(prune.fit)
text(prune.fit, pretty = 0)
plot(fit)
text(fit, pretty = 0)
```






```
fit.pred = predict(fit, newdata = caravan[test,], type = 'class')
# plot(fit.pred, y)
1-mean(fit.pred==ytest)
```

```
## [1] 0.059
```

```
caret::confusionMatrix(fit.pred,ytest, positive = 'Yes')
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  No Yes
##        No  941  59
##        Yes   0   0
##
##               Accuracy : 0.941
##                 95% CI : (0.9246, 0.9548)
##     No Information Rate : 0.941
##     P-Value [Acc > NIR] : 0.5346
##
##                  Kappa : 0
##  Mcnemar's Test P-Value : 4.321e-14
##
##             Sensitivity : 0.000
##             Specificity : 1.000
##          Pos Pred Value :   NaN
##          Neg Pred Value : 0.941
##              Prevalence : 0.059
##          Detection Rate : 0.000
##    Detection Prevalence : 0.000
##       Balanced Accuracy : 0.500
##
##        'Positive' Class : Yes
##
```

```
prune.err = mean(fit.pred != ytest)
```

(c)

Use a bagging approach to analyze the data with B = 1000. Compute the confusion matrix and error rate based on the test data. Which predictors seem to be the most important?

```
library(randomForest)
set.seed(1)
baggy = randomForest(Purchase ~ ., data = xtrain, mtry = 85, ntree=1000, importance = T)
baggy
```

```
##
## Call:
##  randomForest(formula = Purchase ~ ., data = xtrain, mtry = 85,      ntree = 1000, importance
= T)
##                Type of random forest: classification
##                      Number of trees: 1000
## No. of variables tried at each split: 85
##
##         OOB estimate of  error rate: 7.78%
## Confusion matrix:
##       No Yes class.error
## No  4434  99  0.02183984
## Yes  276  13  0.95501730
```

```
baggy.pred = predict(baggy, xtest, type = 'class')
caret::confusionMatrix(baggy.pred,ytest, positive = 'Yes')
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  No Yes
##        No  912  51
##        Yes  29   8
##
##               Accuracy : 0.92
##                 95% CI : (0.9014, 0.9361)
##     No Information Rate : 0.941
##     P-Value [Acc > NIR] : 0.99712
##
##                  Kappa : 0.127
##  Mcnemar's Test P-Value : 0.01888
##
##             Sensitivity : 0.1356
##             Specificity : 0.9692
##          Pos Pred Value : 0.2162
##          Neg Pred Value : 0.9470
##              Prevalence : 0.0590
##          Detection Rate : 0.0080
##    Detection Prevalence : 0.0370
##       Balanced Accuracy : 0.5524
##
##        'Positive' Class : Yes
##
```

```
bag.err = mean(baggy.pred != ytest)
# importance(baggy) # some of the variables seem important
```

(d)

Repeat (c) with a random forest approach with B = 1000 and m ≈ √p.

```
library(randomForest)
set.seed(1)
# mtry defaults to √p when its a classification problem
baggySqrt = randomForest(Purchase ~ ., data = xtrain, ntree=1000, importance = T)
baggySqrt
```

```
##
## Call:
##  randomForest(formula = Purchase ~ ., data = xtrain, ntree = 1000,      importance = T)
##                Type of random forest: classification
##                      Number of trees: 1000
## No. of variables tried at each split: 9
##
##         OOB estimate of  error rate: 6.64%
## Confusion matrix:
##       No Yes class.error
## No  4493  40 0.008824178
## Yes  280   9 0.968858131
```

```
baggySqrt.pred = predict(baggySqrt, xtest, type = 'class')
caret::confusionMatrix(baggySqrt.pred,ytest, positive = 'Yes')
```

```
## Confusion Matrix and Statistics
##
##          Reference
## Prediction  No Yes
##        No  927  55
##        Yes  14   4
##
##               Accuracy : 0.931
##                 95% CI : (0.9135, 0.9459)
##    No Information Rate : 0.941
##    P-Value [Acc > NIR] : 0.918
##
##                  Kappa : 0.0785
##  Mcnemar's Test P-Value : 1.469e-06
##
##             Sensitivity : 0.0678
##             Specificity : 0.9851
##          Pos Pred Value : 0.2222
##          Neg Pred Value : 0.9440
##              Prevalence : 0.0590
##          Detection Rate : 0.0040
##    Detection Prevalence : 0.0180
##       Balanced Accuracy : 0.5265
##
##        'Positive' Class : Yes
##
```

```
# This would be the most important variables but this function is garbage for RF
# importance(baggySqrt)
bagSqrt.err = mean(baggySqrt.pred != ytest)
```

(e)

Repeat (c) with a boosting approach with B = 1000, d = 1, and λ = 0.01.

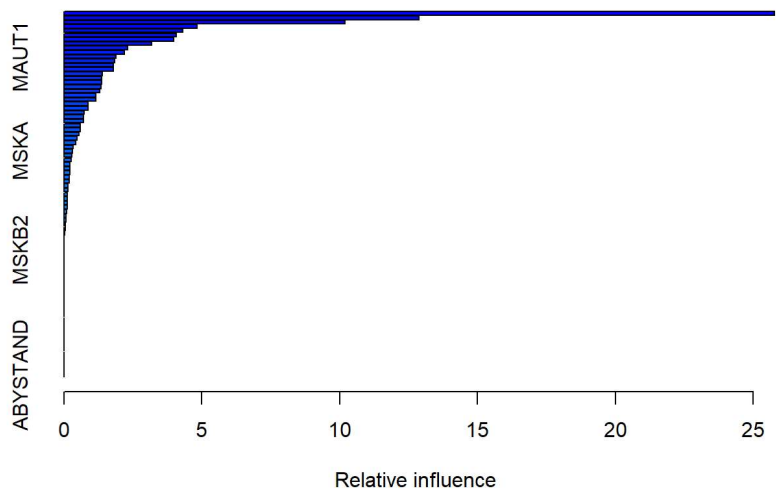
```
library(gbm)
set.seed(1)
booster = gbm((unclass(Purchase)-1) ~ ., data = xtrain, distribution = "bernoulli",
              n.trees = 1000, interaction.depth = 1, shrinkage = 0.01)
booster
```

```
## gbm(formula = (unclass(Purchase) - 1) ~ ., distribution = "bernoulli",
##     data = xtrain, n.trees = 1000, interaction.depth = 1, shrinkage = 0.01)
## A gradient boosted model with bernoulli loss function.
## 1000 iterations were performed.
## There were 85 predictors of which 52 had non-zero influence.
```

```
booster.pred = predict(booster, xtest, booster$n.trees, type = 'response', importance = T)
booster.pred = factor(ifelse(booster.pred >= 0.2, "Yes", "No"))
caret::confusionMatrix(booster.pred,ytest, positive = 'Yes')
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  No Yes
##        No  917  46
##        Yes  24  13
##
##                Accuracy : 0.93
##                  95% CI : (0.9124, 0.945)
##     No Information Rate : 0.941
##     P-Value [Acc > NIR] : 0.93562
##
##                   Kappa : 0.2361
##  Mcnemar's Test P-Value : 0.01207
##
##             Sensitivity : 0.2203
##             Specificity : 0.9745
##          Pos Pred Value : 0.3514
##          Neg Pred Value : 0.9522
##              Prevalence : 0.0590
##          Detection Rate : 0.0130
##    Detection Prevalence : 0.0370
##       Balanced Accuracy : 0.5974
##
##        'Positive' Class : Yes
##
```

```
boost.err = mean(booster.pred != ytest)
summary(booster)[1:5,] # top 5 variables
```



	var <fctr>	rel.inf <dbl>
PPERSAUT	PPERSAUT	25.774629
PPLEZIER	PPLEZIER	12.874527
PBRAND	PBRAND	10.192927
MOPLLAAG	MOPLLAAG	4.835471
MINKGEM	MINKGEM	4.321904

5 rows

(f)

Repeat (c) with a KNN approach with K chosen optimally using the test data.

```
library(class)
ks = c(seq(1, 15, by = 1)) # k = 15 cas my computer is slow
nks = length(ks)
err.rate.train = numeric(length = nks)
err.rate.test = numeric(length = nks)
names(err.rate.train) = names(err.rate.test) = ks
for (i in seq(along = ks)) {
  mod.train = knn(s.caravan[train,], s.caravan[train,], ytrain, k = ks[i])
  mod.test = knn(s.caravan[train,], s.caravan[test,], ytrain, k = ks[i])
  err.rate.train[i] = 1 - sum(mod.train == ytrain)/length(ytrain)
  err.rate.test[i] = 1 - sum(mod.test == ytest)/length(ytest)
}
# Now we want to find the optimal k using a min function
result <- data.frame(ks, err.rate.train, err.rate.test)
result[err.rate.test == min(result$err.rate.test), ]
```

	ks <dbl>	err.rate.train <dbl>	err.rate.test <dbl>
9	9	0.05931149	0.058
10	10	0.05910411	0.058
12	12	0.05931149	0.058

3 rows

```
# It appears we can get away with k = 9, our optimal k
knn = knn(s.caravan[train,], s.caravan[test,], ytrain, k = 9)
caret::confusionMatrix(knn,ytest, positive = 'Yes')
```

```
## Confusion Matrix and Statistics
##
##          Reference
## Prediction  No Yes
##        No  941  58
##        Yes   0   1
##
##               Accuracy : 0.942
##                 95% CI : (0.9257, 0.9557)
##    No Information Rate : 0.941
##    P-Value [Acc > NIR] : 0.4811
##
##                  Kappa : 0.0314
##  Mcnemar's Test P-Value : 7.184e-14
##
##             Sensitivity : 0.01695
##             Specificity : 1.00000
##          Pos Pred Value : 1.00000
##          Neg Pred Value : 0.94194
##              Prevalence : 0.05900
##          Detection Rate : 0.00100
##    Detection Prevalence : 0.00100
##       Balanced Accuracy : 0.50847
##
##        'Positive' Class : Yes
##
```

```
knn.err = mean(knn != ytest)
```

(g)

Repeat (c) with the logistic regression method you recommended in the previous project.

```
library(caret)
set.seed(1)
tc <- trainControl()
model <- train(xtrain, ytrain,
               method="glm", family="binomial", trControl = tc)
model
```

```
## Generalized Linear Model
##
## 4822 samples
##   86 predictor
##    2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 4822, 4822, 4822, 4822, 4822, 4822, ...
## Resampling results:
##
##   Accuracy  Kappa
##   1         1
```

```
model = glm(Purchase~., data = caravan, family = binomial, subset = train)
prob = predict(model, xtest, type = "response")
pred <- ifelse(prob >= 0.2, "Yes", "No")
confusionMatrix(table(pred, ytest))
```

```
## Confusion Matrix and Statistics
##
##      ytest
## pred   No Yes
##   No  897  44
##   Yes  44  15
##
##                Accuracy : 0.912
##                  95% CI : (0.8927, 0.9288)
##     No Information Rate : 0.941
##     P-Value [Acc > NIR] : 0.9999
##
##                   Kappa : 0.2075
##  Mcnemar's Test P-Value : 1.0000
##
##             Sensitivity : 0.9532
##             Specificity : 0.2542
##          Pos Pred Value : 0.9532
##          Neg Pred Value : 0.2542
##              Prevalence : 0.9410
##          Detection Rate : 0.8970
##    Detection Prevalence : 0.9410
##       Balanced Accuracy : 0.6037
##
##        'Positive' Class : No
##
```

```
LR.err = mean(pred != ytest) # Overall misclassification rate
```

(h)

Compare the results from the various methods. Which method would you recommend?

From below, it would appear that KNN with `K = 9` wins here. Just barely though.

```
print(paste("Misclassification rates are"))
```

```
## [1] "Misclassification rates are"
```

```
print(paste("Unpruned Tree", unpruned.err))
```

```
## [1] "Unpruned Tree 0.059"
```

```
print(paste("CV pruned Tree", prune.err))
```

```
## [1] "CV pruned Tree 0.059"
```

```
print(paste("Bagging", bag.err))
```

```
## [1] "Bagging 0.08"
```

```
print(paste("Random Forest", bagSqrt.err))
```

```
## [1] "Random Forest 0.069"
```

```
print(paste("Boosting", boost.err))
```

```
## [1] "Boosting 0.07"
```

```
print(paste("KNN", knn.err))
```

```
## [1] "KNN 0.058"
```

```
print(paste("Logistic Regression", LR.err))
```

```
## [1] "Logistic Regression 0.088"
```

Question 3

Consider the Hitters data discussed in Chapter 8 of the book. Use log Salary as a response and Years and Hits as predictors.

```
library(ISLR)
Hitters <- na.omit(Hitters)
```

(a)

** Fit a tree to these data by "hand," i.e., implement all the steps and perform all the necessary calculations by yourself. You may use the "hand calculation" handout as a model.**

Function below to determine best place to split

```
ss = function(x, y) { # calculates sum-square
  sum((y-x)^2)
}

splitter = function(x, y) { # 'Hand' Calculation
  splits = sort(unique(x))
  RSS = c()
  for (i in seq_along(splits)) {
    sp = splits[i] # split point considering variable x[i]
    R1 = y[x <= sp] # first half pair region
    R2 = y[x > sp] # second
    c1 = mean(R1) # prediction of R1
    c2 = mean(R2)
    RSS[i] = ss(c1,R1)+ss(c2,R2)
  }
  bestSp = splits[which.min(RSS)]
  bestRSS = min(RSS)
  leftx = splits[splits < bestSp]
  left = length(leftx)
  rightx = splits[splits > bestSp]
  right = length(rightx)
  print(paste("Best split point with lowest RSS & remaining left and right obs"))
  return(c(bestRSS, bestSp, left, right))
}
y = log(Hitters$Salary)
years = Hitters$Years
hits = Hitters$Hits
splitter(years,y) # we can stop since 3 < 5 obs on one node
```

```
## [1] "Best split point with lowest RSS & remaining left and right obs"
```

```
## [1] 115.0585   4.0000   3.0000  17.0000
```

```
splitter(hits,y)
```

```
## [1] "Best split point with lowest RSS & remaining left and right obs"
```

```
## [1] 160.9715 117.0000  69.0000  60.0000
```

```
  splitter(hits[hits<117],y) # *remaining left
```

```
## [1] "Best split point with lowest RSS & remaining left and right obs"
```

```
## [1] 196.8808  99.0000  55.0000  13.0000
```

```
    splitter(hits[hits<99],y) # **remaining left
```

```
## [1] "Best split point with lowest RSS & remaining left and right obs"
```

```
## [1] 203.8382  63.0000  24.0000  30.0000
```

```
    splitter(hits[hits<117&hits>99],y) # **remaining right
```

```
## [1] "Best split point with lowest RSS & remaining left and right obs"
```

```
## [1] 198.385 110.000   7.000   5.000
```

```
  splitter(hits[hits>117],y) # *remaining right
```

```
## [1] "Best split point with lowest RSS & remaining left and right obs"
```

```
## [1] 203.707 139.000  19.000  40.000
```

```
    splitter(hits[hits>117&hits<139],y) # **remaining left
```

```
## [1] "Best split point with lowest RSS & remaining left and right obs"
```

```
## [1] 202.5576 126.0000   7.0000  11.0000
```

```
    splitter(hits[hits>139],y) # **emaining right
```

```
## [1] "Best split point with lowest RSS & remaining left and right obs"
```

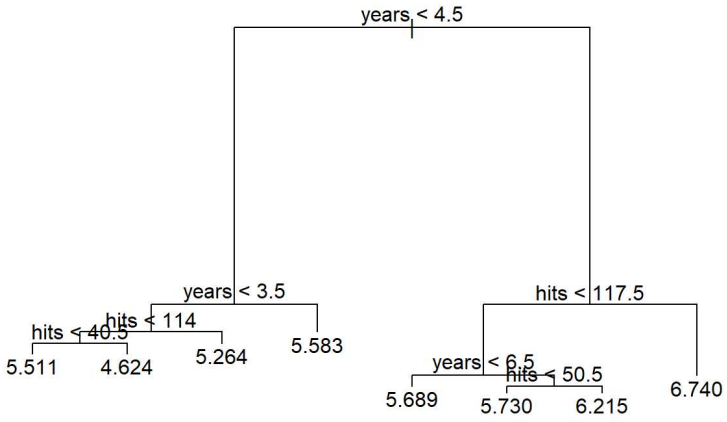
```
## [1] 206.2173 213.0000  37.0000   2.0000
```

```
# you get the idea, we would keep splitting til obs on either node dip below 5
```

(b)

> **Fit a tree using tree function in R and compare your results.**

```
fit = tree::tree(formula = y ~ years + hits, data = Hitters)
fit
plot(fit)
text(fit, pretty = 0)
```



Question 4

> **Consider the Heart discussed in Chapter 8 of the book. Use HD as a binary response, Thal and Ca as predictors, and deviance as the impurity measure. Repeat the previous exercise for these data.**

```
heart = read.csv("Heart.csv", header = T)
```

(a)

> ** Fit a tree to these data by "hand," i.e., implement all the steps and perform all the necessary calculations by yourself. You may use the "hand calculation" handout as a model.**

Function below to determine best place to split

```
d = function(y) { # calculates deviance, how deviant!
  n <- length(y)
  n.yes <- sum(y == "Yes")
  p.yes <- n.yes/n
  n.no <- sum(y == "No")
  p.no <- n.no/n
  -2*(n.yes * log(p.yes) + n.no * log(p.no))

}

splooter = function(x, y) { # 'Hand' Calculation
  splits = sort(unique(x))
  dev = c()
  for (i in seq_along(splits)) {
    # print(x[i]) # current Level
    sp = splits[i] # split point considering variable x[i]
    yi = subset(y, x == x[i])
    dev[i] = d(yi)
  }
  print(dev)
  bestSp = x[which.min(dev)]
  bestdev = min(dev)
  print(paste("Best split for min Deviance"))
  return(c(bestdev, bestSp))
}
y = heart$AHD
thal = heart$Thal
ca = heart$Ca
"Deviance set for Thal"
```

```
## [1] "Deviance set for Thal"
```

```
splooter(thal, y)
```

```
## [1]  22.91451 176.14042 128.76797
## [1] "Best split for min Deviance"
```

```
## [1] 22.91451  1.00000
```

```
"Deviance set for Ca"
```

```
## [1] "Deviance set for Ca"
```

```
splooter(ca, y)
```

```
## [1] 202.21640  16.90836  36.30660 202.21640
## [1] "Best split for min Deviance"
```

```
## [1] 16.90836  3.00000
```

(b)

> **Fit a tree using tree function in R and compare your results.**

```
fit = tree::tree(formula = y ~ thal + ca, data = heart)
fit
plot(fit)
text(fit, pretty = 0)
```