521 HW 4

William Tirone

The Honor Code

❗ Important

(a) Please state the names of people who you worked with for this homework. You can also provide your comments about the homework here.
(b) Please type/write the following sentences yourself and sign at the end. We want to make it extra clear that nobody cheats even unintentionally.
I hereby state that all of my solutions were entirely in my words and were written by me. I have not looked at another student's solutions and I have fairly credited all external sources in this write up.

1

1.1

TRUE. The l_2 is just a linear regularization term that can be added, also referencing ESL p. 125 eqn. 4.31.

1.2

TRUE. The logistic function takes numbers on the real line and maps them to [0,1].

1.3

FALSE. We do not need to scale (unless we are regularizing). Mentioned in Lecture 14 p. 21.

1.4

FALSE. Using maximum likelihood results in $X^T(y - p)$ which has no closed form solution but can be approximated with the Newton-Raphson method.

1.5

FALSE. It assumes they come from a Bernoulli distribution.

1.6

FALSE. The response variable y must be categorical for logistic regression.

1.7

FALSE. It becomes more computationally costly to calculate distances in higher dimensions for the algorithm and the points could become much farther away as we add dimensions.

1.8

TRUE. The Bayes' decision boundary is the unachievable best boundary, so LDA will more closely approximate this if it is linear.

1.9

FALSE. The Bayes' classifier is not achievable in practice.

1.10

TRUE. Adjusting K can increase or decrease the model complexity, which is the x-axis in a bias-variance plot.

2

2.1

$$a + v = O(d)$$
$$a^T v = O(d + d) = O(d)$$

2.2

$$A + B = O(2(n \times d)) = O(n \times d)$$

Space required: If A has all integer elements, it would require 4 bytes * (n * d), and if it contains float values, it would require 8 bytes * (n * d), since storing an integer requires 4 bytes and a float requires 8. I assume this varies based on hardware / programming language.

2.3

$$Av = O((d + d)n) = O(nd)$$

$$A^T B = O(n^2 d)$$

2.4

Doing this the smarter way, we change the order of multiplication and find $b = Bv = O(n^2)$ then using $A^T b = O(n^2 + n^2)$ so

$$A^T Bv = O(n^2)$$

However, going the other direction, we end up with:

$$A^T B = O(n^2 d)$$
$$A^T Bv = O(n^2 d)$$

3

4 (4.14 ISL)

```r
library(ISLR2)
```

```
Attaching package: 'ISLR2'

The following object is masked from 'package:MASS':

    Boston
```

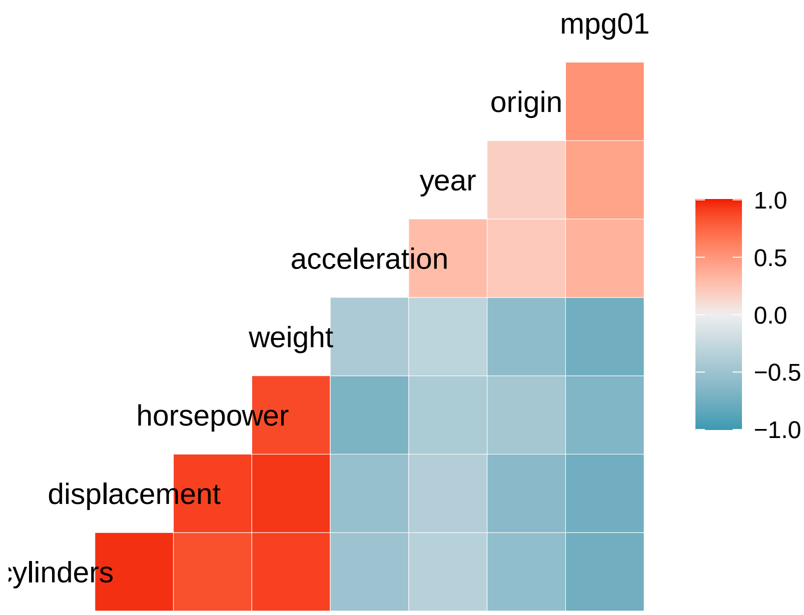
a)

```r
Auto$mpg01 = if_else(Auto$mpg > median(Auto$mpg),1,0)
```

b)

Below, looking at the correlation plot, it seems like all the variables have a strong relationship with mpg01. This makes sense, since every attribute of a car is going to affect its mpg. Year, origin, and acceleration improve MPG, while weight, horsepower, displacement, and cylinders decrease it and make it more likely to fall under the median. Newer cars have better MPG, and heavier cars have worse MPG. The same reasoning applies for the other factors, for physics reasons that I probably don't understand.

```r
# correlate, removing "name" column
ggcorr(Auto[,c(-1,-9)])
```



c)

```r
# code borrowed here: https://www.statology.org/train-test-split-r/
set.seed(123)
sample = sample(c(TRUE, FALSE), nrow(Auto), replace=TRUE, prob=c(0.8,0.2))
auto.train = Auto[sample, c(-1,-9)]
auto.test = Auto[!sample, c(-1,-9)]
```

d) LDA

Fitting the model on training data

```
# referencing ISL p. 187
lda.fit = lda(mpg01 ~ origin + year + acceleration + weight +
                horsepower + displacement + cylinders,
              data=auto.train)
lda.fit
```

```
Call:
lda(mpg01 ~ origin + year + acceleration + weight + horsepower +
    displacement + cylinders, data = auto.train)

Prior probabilities of groups:
        0         1
0.5062893 0.4937107

Group means:
    origin     year acceleration   weight horsepower displacement cylinders
0 1.167702 74.40373     14.78447 3606.665  128.65839     269.9565  6.689441
1 1.993631 77.57962     16.47898 2321.166   78.89172     113.8376  4.152866

Coefficients of linear discriminants:
                      LD1
origin        0.159011755
year          0.124446573
acceleration -0.032096125
weight       -0.001172640
horsepower    0.010471621
displacement -0.001242684
cylinders    -0.428862636
```

Finding test error:

```
lda.pred = predict(lda.fit, auto.test)
table(lda.pred$class, auto.test$mpg01)
```

```
     0  1
  0 31  4
  1  4 35
```

```r
cat("LDA test error", 1-mean(lda.pred$class == auto.test$mpg01))
```

```
LDA test error 0.1081081
```

e) QDA

Fitting the model

```r
# Smarket.2005 is their test set
# p. 189 ISL
qda.fit = qda(mpg01 ~ origin + year + acceleration + weight +
                horsepower + displacement + cylinders,
              data=auto.train)

qda.fit
```

```
Call:
qda(mpg01 ~ origin + year + acceleration + weight + horsepower +
    displacement + cylinders, data = auto.train)

Prior probabilities of groups:
        0         1
0.5062893 0.4937107

Group means:
    origin     year acceleration   weight horsepower displacement cylinders
0 1.167702 74.40373     14.78447 3606.665  128.65839     269.9565  6.689441
1 1.993631 77.57962     16.47898 2321.166   78.89172     113.8376  4.152866
```

Finding QDA test error

```r
qda.pred = predict(qda.fit, auto.test)
table(qda.pred$class, auto.test$mpg01)
```

	0	1
0	33	6
1	2	33

```r
cat("QDA test error", 1-mean(qda.pred$class == auto.test$mpg01))
```

```
QDA test error 0.1081081
```

f) Logistic Regression

```r
#p. 184
log.fit = glm(mpg01 ~ origin + year + acceleration + weight +
                horsepower + displacement + cylinders,
              data=auto.train,
              family=binomial)
summary(log.fit)
```

```
Call:
glm(formula = mpg01 ~ origin + year + acceleration + weight +
    horsepower + displacement + cylinders, family = binomial,
    data = auto.train)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-2.30655  -0.06257  -0.00002   0.15488   2.30956

Coefficients:
               Estimate Std. Error z value    Pr(>|z|)
(Intercept)  -16.917071   6.806768  -2.485     0.01294 *
origin         0.278639   0.403100   0.691     0.48941
year           0.497690   0.095529   5.210 0.000000189 ***
acceleration  -0.160559   0.160642  -0.999     0.31756
weight        -0.003855   0.001375  -2.802     0.00507 **
horsepower    -0.074588   0.030784  -2.423     0.01539 *
displacement  -0.021912   0.015813  -1.386     0.16584
cylinders      0.485349   0.539218   0.900     0.36807
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 440.79  on 317  degrees of freedom
```

```
Residual deviance: 111.32  on 310  degrees of freedom
AIC: 127.32

Number of Fisher Scoring iterations: 8
```

Test Error:

```
# first find probabilities using predict and our model
log.probs = predict(log.fit, auto.test, type = 'response')

# create vector with zeros for below median MPG and 1's for above median.
log.pred <- rep(0, 74)
log.pred[log.probs > .5] = 1

# table output
table(log.pred,auto.test$mpg01)
```

```
log.pred  0  1
       0 33  7
       1  2 32
```

```
cat("logistic test error : ", 1-mean(log.pred == auto.test$mpg01))
```

```
logistic test error :  0.1216216
```

g) naive Bayes

```
nb.fit = naiveBayes(mpg01 ~ origin + year + acceleration + weight +
                horsepower + displacement + cylinders,
              data=auto.train)
nb.fit
```

```
Naive Bayes Classifier for Discrete Predictors

Call:
naiveBayes.default(x = X, y = Y, laplace = laplace)
```

```
A-priori probabilities:
Y
        0         1 
0.5062893 0.4937107 

Conditional probabilities:
   origin
Y       [,1]      [,2]
  0 1.167702 0.4774414
  1 1.993631 0.8733726

   year
Y       [,1]     [,2]
  0 74.40373 3.038048
  1 77.57962 3.561056

   acceleration
Y       [,1]     [,2]
  0 14.78447 2.783715
  1 16.47898 2.456598

   weight
Y       [,1]     [,2]
  0 3606.665 686.1514
  1 2321.166 362.3148

   horsepower
Y        [,1]     [,2]
  0 128.65839 37.99854
  1  78.89172 15.78850

   displacement
Y       [,1]     [,2]
  0 269.9565 90.58424
  1 113.8376 31.97907

   cylinders
Y       [,1]     [,2]
  0 6.689441 1.437167
  1 4.152866 0.590150
```

Test error:

```r
nb.pred = predict(nb.fit, auto.test)
table(nb.pred, auto.test$mpg01)
```

```
nb.pred  0  1
      0 32  3
      1  3 36
```

```r
cat("Naive Bayes test error", 1-mean(nb.pred == auto.test$mpg01))
```

```
Naive Bayes test error 0.08108108
```

h) KNN

First, without standardization:

```r
knn.pred.noscale = knn(auto.train, auto.test, auto.train$mpg01, k=3)
cat("KNN Error Rate : ", 1-mean(knn.pred.noscale == auto.test$mpg01))
```

```
KNN Error Rate :  0.08108108
```

Now with standardization, since the variables have very different scales, we get a lower error rate.

```r
standardized.train = scale(auto.train)
standardized.test = scale(auto.test)

knn.pred = knn(standardized.train, standardized.test, auto.train$mpg01, k=3)
table(knn.pred, auto.test$mpg01)
```

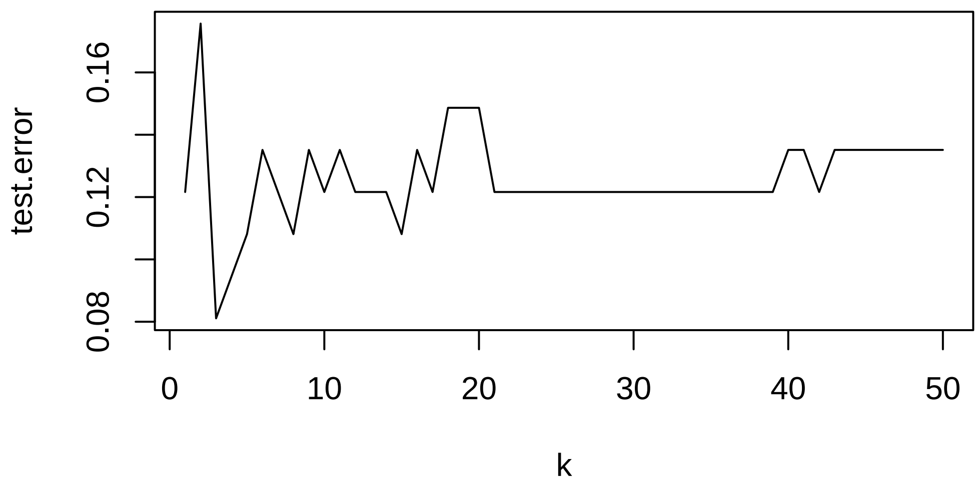
```
knn.pred  0  1
       0 35  0
       1  0 39
```

```r
cat("KNN Error Rate : ", 1-mean(knn.pred == auto.test$mpg01))
```

```
KNN Error Rate :  0
```

Choosing the best K by iterating through different choices. It looks like around k=3 gives us the lowest test error, without standardization. With standardization, we can see above, it correctly predicted all the responses.

```
k = 1:50
test.error = c()
for (i in k) {
  knn = knn(auto.train, auto.test, auto.train$mpg01, k=i)
  test.error = c(test.error,1-mean(knn == auto.test$mpg01))
}

plot(k, test.error,type='l')
```



4

hand written, see attached.

5

a)

Though we could standardize the data here, like mentioned in problem 4, I don't think it's necessary since the variables are on fairly similar scales.

```r
knn_predict = function(X_train,X_test,y_train,k){

  y_test = c()

  for (i in 1:dim(X_test)[1]){

    # use this to make matrix of correct dim
    one_point = matrix(as.numeric(X_test[i,]),
                       nrow=dim(X_train)[1],
                       ncol=4,
                       byrow=TRUE)

    # find the row norms and build a new dataframe with labels
    residuals = data.frame(y_train,
                       rowNorms(as.matrix(X_train - one_point)))

    # sorting by distance and pulling out predicted label by maj vote
    label = majorityVote(arrange(residuals, residuals[,2])[1:k,1])$majority
    y_test = c(y_test, label)
  }

  return(y_test)
}
```

b)

```r
# code from homework
library(datasets)
data(iris)
training <- c(1:47, 51:97, 101:146)
testing <- c(48:50, 98:100, 147:150)
train_set <- iris[training, ]
test_set <- iris[testing, ]

pred_knn <- knn_predict(train_set[, -5], test_set[, -5], train_set$Species, k=1)
pred_knn
```

```
 [1] "setosa"     "setosa"     "setosa"     "versicolor" "versicolor"
 [6] "versicolor" "virginica"  "virginica"  "virginica"  "virginica"
```

Comparing to R version for accuracy's sake:

```
a = knn_predict(train_set[, -5], test_set[, -5], train_set$Species, k=10)
b = knn(train_set[,-5], test_set[,-5], train_set$Species, k=10)
a==b
```

```
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

c)

I deviated from the instructions slightly - my `find_kcv` function returns both the errors and the optimal value because I wanted to plot the errors and see what was going on. I was surprised to see that the errors did not increase linearly but were all over the place.

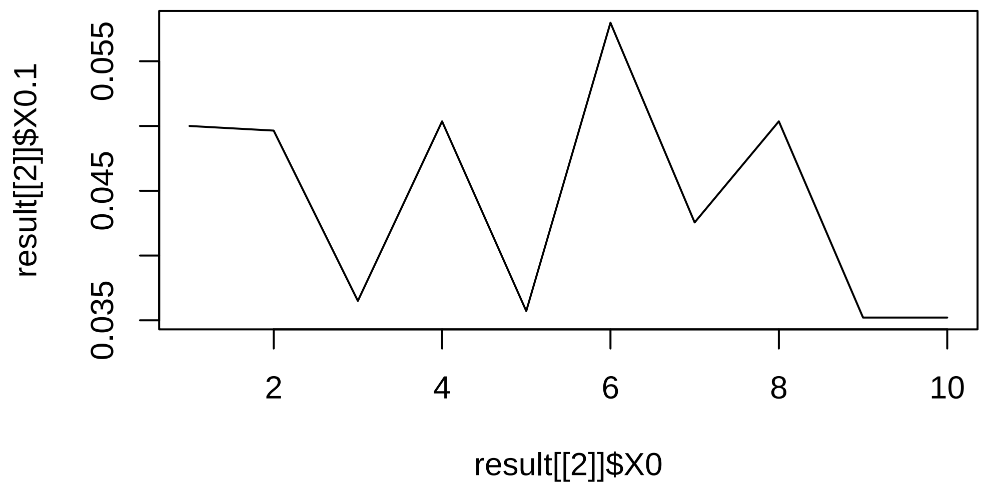
```
find_kcv = function(X_train,y_train,ks=1:10,nfold=5){

  # empty frame to hold errors
  errors = data.frame(0,0)

  for (k in ks)  {

    results = c()

    for (i in 1:nfold) {

      # create folds
      folds = createFolds(y_train,nfold)

      # test
      test_fold = X_train[folds[[i]],]
      test_labels = y_train[folds[[i]]]

      # train
      train_fold = X_train[-folds[[i]],]
      train_labels = y_train[-folds[[i]]]

      # run knn
      pred_knn = knn_predict(train_fold, # no indexes here
                             test_fold,
                             train_labels,
                             k=k)
```

```
      result = test_labels == pred_knn
      results = c(results, result)
    }

    val = 1 - mean(results)
    errors[k,] = c(k,val)
  }

  # finding optimal k
  optimal_k = errors[errors[,2] == min(errors[,2]),][1]
  return_values = list(optimal_k, errors)
  return(return_values)
}

result = find_kcv(train_set[, -5], train_set[, 5])
cat("Optimal K Value :", as.integer(result[[1]][1,1]))
```

```
Optimal K Value : 9
```

Plotting the result of validating over different K values:

```
plot(result[[2]]$X0, result[[2]]$X0.1, type='l')
```



Out of curiosity, trying this again with a very large k, choosing optimal k, and plotting. Contrary to the small K, we can see below that the errors increase erratically as K grows and jump very high from 70-90.

```r
large_k = find_kcv(train_set[, -5], train_set[, 5], ks=1:100)
as.integer(large_k[[1]][1,1])
```

```
[1] 3
```

```r
plot(large_k[[2]]$X0, large_k[[2]]$X0.1, type='l',col='purple')
```