AFS 505 HW4

Aaron Appleby

2022-09-20

#HW4

1. **Functions**

a) **trim.CV**

```
# Create a function with the **function()** command, then use the function with the assigned name. A fu

## > function_name <- function(arg_1, arg_2, ...) {
##   Function body
##   }


# x = input dataframe
trim.CV <- function(x) {
  Q3 <- quantile(x,0.75)
  Q1 <- quantile(x,0.25)
  IQR <- IQR(x)
  trim <- !((x-Q3)>1.5*IQR | (Q1-x)>1.5*IQR)
   #after trimming the input x of outliers, compute coeficient of variation
  return(sd(trim)/mean(trim))
}



#trim <- iris[!((iris$'sepal.width.(cm)' - Q3) >  1.5*IQR |  (Q1 - iris$`sepal.width.(cm)`) > 1.5*IQR),
#trim
#iris$`sepal.widt=h.(cm)`
#head(iris)
#str(trim$`sepal.width.(cm)`)
#grep(4.2, trim$`sepal.width.(cm)`, ignore.case=T)



#boxplot(iris$`sepal.width.(cm)`)
```

1.

b) **apply TrimCV to sepal width**

```
# Call in Data Set iris
iris <- read.csv("https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data", header=F)
head(iris)
```

```
##    V1  V2  V3  V4          V5
## 1 5.1 3.5 1.4 0.2 Iris-setosa
## 2 4.9 3.0 1.4 0.2 Iris-setosa
## 3 4.7 3.2 1.3 0.2 Iris-setosa
## 4 4.6 3.1 1.5 0.2 Iris-setosa
## 5 5.0 3.6 1.4 0.2 Iris-setosa
## 6 5.4 3.9 1.7 0.4 Iris-setosa
```

```
#first row is not a header <- added header=F
##need to name columns
colnames(iris) <- c("sepal.length",
                    "sepal.width",
                    "petal.length",
                    "petal.width",
                    "class"
                    )
# Apply TrimCV function to sepal width from iris data set.
trim.CV(iris$sepal.width)
```

```
## [1] 0.1660757
```

2.

a. **tapply**

Use tapply() to compute group standard deviations for petal width, grouping by variety. Comment briefly on results.

The **tapply()** functions is in the **apply()** family of functionals. This function allows you to apply a function to a vector of data by one or more grouping variables.

So, for example, if you have petal width measurements as a variable in the iris dataframe and a second vector (or dataframe variable) with the variety (iris$class$) $for each measurement, then you could use * *tapply() * *to compute the mean, median or some other function by variety(iris$class$).

The grammar of **tapply()** is as follows:

> tapply(X, INDEX, FUN, . . .)
>
> X = vector or dataframe column INDEX = grouping variable (factor or coerced to factor) FUN = function to be applied

```
petal.width.sd <- with (iris, tapply(petal.width, INDEX=class, FUN=sd))
petal.width.sd
```

```
##     Iris-setosa Iris-versicolor  Iris-virginica
##       0.1072095       0.1977527       0.2746501
```

```
#petal.width.sd <-  tapply(iris$petal.width, INDEX=iris$class, FUN=sd)

## gives the same output as above using with()
## without with() you need to identify where the named column is from
```

Standard deviations of iris petal width follows the same pattern as mean with Iris-setosa having the smallest standard deviation, and Iris-virginica having the largest standard deviation.

2.

b) **aggregate**

Use aggregate() with standard specification to compute group medians for all numerical variables in the iris dataset, grouping by variety. Comment briefly on results.

The **aggregate()** function provides an extension of **tapply()** to dataframes. Whereas **tapply()** applies a function by a grouping variable or variables to a vector, **aggregate()** applies a function by a grouping variable or variables to multiple columns (variables) in a dataframe.

There are two ways to formulate an aggregate call as discussed below.

standard aggregate() call

aggregate(x, by, FUN, . . .)

x = dataframe by = grouping variable or list of grouping variables (factor or coerced to factor)
FUN = function to be applied

Keep in mind that you might need to select a subset of dataframe columns that are appropriate for the function being applied.

```
aggregate(iris[,-5], by=list(iris$class),FUN=mean)
```

```
##           Group.1 sepal.length sepal.width petal.length petal.width
## 1     Iris-setosa        5.006       3.418        1.464       0.244
## 2 Iris-versicolor        5.936       2.770        4.260       1.326
## 3  Iris-virginica        6.588       2.974        5.552       2.026
```

Without removing columns that do not have numeric values, aggregate returns NA. In this format, it is easy to see that petal width and length are correlated. Something else I did not realize until I saw this new table was sepal length follows the same pattern as petal measurements, with the smallest measurements for I-setosa.

3.

a) **apply**

Use the apply() function to compute column-wise sums of numerical data in the iris dataset.

For many operations, the **apply** function is the most efficient way to accomplish your data analysis task.

The **apply()** function is one of many *functionals* in R. These are functions that include other functions in parameterization.

The **apply()** function, which operates on matrices or arrays (3 or more dimensions with the same data type), can often a more efficient way to accomplish tasks that might otherwise be implemented with a control loop (e.g. for, while). The basic format of the apply command is as follows:

apply(X, MARGIN, FUN, . . .)

X = matrix or array (data) MARGIN = dimension over which to apply the function (1 = row, 2 = column) FUN = function to be applied Note that the '. . . ' designation provides a way to pass through parameters to the function.

column-wise apply operations

```
iris.sum <- apply(iris[,-5], MARGIN=2, FUN=sum)   # Margin=2 <- apply to column
iris.sum
```

```
## sepal.length  sepal.width petal.length  petal.width
##        876.5        458.1        563.8        179.8
```

```
#for style and to increase visual appeal, the matrix can be turned into a data frame and manipulated to
#data.frame(iris.sum)
#data.frame("iris measurement"=names(iris.sum),'sum of values'=iris.sum, row.names=NULL)
#data.frame(t(iris.sum))
```

3.

b) **Length/Width**

Use a for loop to compute the following ratio for each observation (row) in the iris dataset:

iris.ratio = (sepal.lenth+petal.length)/(sepal.width+petal.width)

What fraction of observations have a ratio > 2.5 ? (Note: you can write this as a for loop without embedding in a function.)

A **for()** statement is commonly used to iteratively loop through a code sequence. The format of this is as follows:

for (*variable.name* in *sequence expression*) *expression*

Or more commonly:

for (*variable.name* in *sequence expression*) { *expression 1 expression 2 expression 3* }

```
Ilxw <- rep(150) #create a repository to hold following for loop observations

for(i in 1:nrow(iris)){
Ilxw[i] <- (iris[i,1]+iris[i,3])/(iris[i,2]+iris[i,4])
}

Ilxw
```

```
##   [1] 1.756757 1.968750 1.764706 1.848485 1.684211 1.651163 1.621622 1.805556
##   [9] 1.870968 2.000000 1.769231 1.777778 2.000000 1.741935 1.666667 1.500000
##  [17] 1.558140 1.710526 1.804878 1.609756 1.972222 1.609756 1.473684 1.789474
##  [25] 1.861111 2.062500 1.736842 1.810811 1.833333 1.852941 1.939394 1.815789
```

```
##  [33] 1.595238 1.568182 2.000000 1.823529 1.837838 2.000000 1.781250 1.833333
##  [41] 1.657895 2.230769 1.676471 1.609756 1.666667 1.878788 1.675000 1.764706
##  [49] 1.743590 1.828571 2.543478 2.319149 2.565217 2.638889 2.581395 2.487805
##  [57] 2.244898 2.411765 2.666667 2.219512 2.833333 2.244444 3.125000 2.511628
##  [65] 2.190476 2.466667 2.244444 2.675676 2.891892 2.638889 2.140000 2.463415
##  [73] 2.800000 2.700000 2.547619 2.500000 2.761905 2.489362 2.386364 2.555556
##  [81] 2.657143 2.705882 2.487179 2.581395 2.200000 2.100000 2.478261 2.972222
##  [89] 2.255814 2.500000 2.605263 2.431818 2.578947 2.515152 2.450000 2.357143
##  [97] 2.357143 2.500000 2.250000 2.390244 2.120690 2.369565 2.549020 2.531915
## [105] 2.365385 2.784314 2.238095 2.893617 2.906977 2.180328 2.230769 2.543478
## [113] 2.411765 2.377778 2.096154 2.127273 2.500000 2.400000 2.979592 2.972973
## [121] 2.290909 2.187500 3.000000 2.488889 2.296296 2.640000 2.391304 2.291667
## [129] 2.448980 2.826087 2.872340 2.465517 2.400000 2.651163 2.925000 2.603774
## [137] 2.051724 2.428571 2.250000 2.365385 2.236364 2.222222 2.369565 2.309091
## [145] 2.137931 2.245283 2.568182 2.340000 2.035088 2.291667
```

```
Ilxw2.5<-Ilxw [Ilxw> 2.5]
str(Ilxw2.5)
```

```
##  num [1:39] 2.54 2.57 2.64 2.58 2.67 ...
```

39/150 is the fraction of observations with length/width ratios greater than 2.5

3.

c) **apply**

```
lxw <- function(x) {
  lw<-((x[1]+x[3])/(x[2]+x[4]))
       return(lw)
}
#iris.ratio = (iris$sepal.length+iris$petal.length)/(iris$sepal.width+iris$petal.width)
###above code does the same as the loop and apply...
#iris.ratio

iris.lxw <- apply(iris[,-5], MARGIN=1, FUN=lxw)
iris.lxw
```

```
##   [1] 1.756757 1.968750 1.764706 1.848485 1.684211 1.651163 1.621622 1.805556
##   [9] 1.870968 2.000000 1.769231 1.777778 2.000000 1.741935 1.666667 1.500000
##  [17] 1.558140 1.710526 1.804878 1.609756 1.972222 1.609756 1.473684 1.789474
##  [25] 1.861111 2.062500 1.736842 1.810811 1.833333 1.852941 1.939394 1.815789
##  [33] 1.595238 1.568182 2.000000 1.823529 1.837838 2.000000 1.781250 1.833333
##  [41] 1.657895 2.230769 1.676471 1.609756 1.666667 1.878788 1.675000 1.764706
##  [49] 1.743590 1.828571 2.543478 2.319149 2.565217 2.638889 2.581395 2.487805
##  [57] 2.244898 2.411765 2.666667 2.219512 2.833333 2.244444 3.125000 2.511628
##  [65] 2.190476 2.466667 2.244444 2.675676 2.891892 2.638889 2.140000 2.463415
##  [73] 2.800000 2.700000 2.547619 2.500000 2.761905 2.489362 2.386364 2.555556
##  [81] 2.657143 2.705882 2.487179 2.581395 2.200000 2.100000 2.478261 2.972222
##  [89] 2.255814 2.500000 2.605263 2.431818 2.578947 2.515152 2.450000 2.357143
##  [97] 2.357143 2.500000 2.250000 2.390244 2.120690 2.369565 2.549020 2.531915
## [105] 2.365385 2.784314 2.238095 2.893617 2.906977 2.180328 2.230769 2.543478
```

```
## [113] 2.411765 2.377778 2.096154 2.127273 2.500000 2.400000 2.979592 2.972973
## [121] 2.290909 2.187500 3.000000 2.488889 2.296296 2.640000 2.391304 2.291667
## [129] 2.448980 2.826087 2.872340 2.465517 2.400000 2.651163 2.925000 2.603774
## [137] 2.051724 2.428571 2.250000 2.365385 2.236364 2.222222 2.369565 2.309091
## [145] 2.137931 2.245283 2.568182 2.340000 2.035088 2.291667
```

4.

a) **read in data set**

```
# Call in Data Set ecoli
ecoli<- read.table("https://archive.ics.uci.edu/ml/machine-learning-databases/ecoli/ecoli.data", header=
colnames(ecoli) <- c("Sequence Name",
                     "mcg",
                     "gvh",
                     "lip",
                     "chg",
                     "aac",
                     "alm1",
                     "alm2",
                     "class")
#https://archive.ics.uci.edu/ml/machine-learning-databases/ecoli/ecoli.names states that there are 8 va
head(ecoli)
```

```
##   Sequence Name  mcg  gvh  lip chg  aac alm1 alm2 class
## 1     AAT_ECOLI 0.49 0.29 0.48 0.5 0.56 0.24 0.35    cp
## 2    ACEA_ECOLI 0.07 0.40 0.48 0.5 0.54 0.35 0.44    cp
## 3    ACEK_ECOLI 0.56 0.40 0.48 0.5 0.49 0.37 0.46    cp
## 4    ACKA_ECOLI 0.59 0.49 0.48 0.5 0.52 0.45 0.36    cp
## 5     ADI_ECOLI 0.23 0.32 0.48 0.5 0.55 0.25 0.35    cp
## 6    ALKH_ECOLI 0.67 0.39 0.48 0.5 0.36 0.38 0.46    cp
```

4.

b) **lapply() and sapply()**

The **lapply()** and **sapply()** functions are both in the **apply()** family of functionals. The basic function is **lapply()** with the **sapply()** being a *wrapper* function that generates results in a format more appropriate for some uses.

In a nutshell, **lapply()** uses apply for a list, data.frame (special kind of list) or vector. There is no MARGIN as the function FUN is applied to each element sequentially.

> lapply(X, FUN, . . .)
>
> X = list, vector or dataframe (data) FUN = function to be applied Note that the '. . . ' designation provides a way to pass through parameters to the function.

```
lapply(ecoli[,2:8],FUN=sum)
```

```
## $mcg
## [1] 168.02
```

```
##
## $gvh
## [1] 168
##
## $lip
## [1] 166.48
##
## $chg
## [1] 168.5
##
## $aac
## [1] 168.01
##
## $alm1
## [1] 168.06
##
## $alm2
## [1] 167.91
```

```
# automatically applies the function to columns first
##gives output in the form of a list

sapply(ecoli[,2:8],FUN=sum)
```

```
##    mcg    gvh    lip    chg    aac   alm1   alm2
## 168.02 168.00 166.48 168.50 168.01 168.06 167.91
```

```
# also automatically applies the function to columns first
##gives the output in the form of a vector

data.class(lapply(ecoli[,2:8],FUN=mean))## lapply gives output in the form of a list
```

```
## [1] "list"
```

```
data.class(sapply(ecoli[,2:8],FUN=mean))## sapply gives the output in the form of a vector (numeric in
```

```
## [1] "numeric"
```

Both lapply() and sapply() automatically apply the specified function to columns first. lapply() returns the output in the form of a list. sapply() returns the output in the form of a vector.

4.

c) **mean using sapply**

```
sapply(ecoli[,2:8],FUN=mean)
```

```
##       mcg       gvh       lip       chg       aac      alm1      alm2
## 0.5000595 0.5000000 0.4954762 0.5014881 0.5000298 0.5001786 0.4997321
```

```
apply(ecoli[,2:8], 2, mean)
```

```
##       mcg       gvh       lip       chg       aac      alm1      alm2
## 0.5000595 0.5000000 0.4954762 0.5014881 0.5000298 0.5001786 0.4997321
```

```
data.class(apply(ecoli[,2:8], 2, mean))
```

```
## [1] "numeric"
```

Both sapply and apply return the output as a numerical vector in this case. sapply automatically applies the function to columns first where apply you must specify either row or column to apply the function. The results are similar because both sapply and apply are being applied to a data frame, there would be some differences if ecoli was a matrix rather than a data frame, as sapply treats every value in the matrix as an element in the list (essentially coerces the matrix into a vector).