6 Control Structures Homework 1,2,3,4

Andrew Marshall

7/6/2018

# Data

As with Homework 5, all the numeric values you need, other than 0.05, 0, 1, 2 and 3 are defined below:

Year=c(1936, 1946, 1951, 1963, 1975, 1997, 2006)  
CaloriesPerRecipeMean <- c(2123.8, 2122.3, 2089.9, 2250.0, 2234.2, 2249.6, 3051.9)  
CaloriesPerRecipeSD <- c(1050.0, 1002.3, 1009.6, 1078.6, 1089.2, 1094.8, 1496.2)  
CaloriesPerServingMean <- c(268.1, 271.1, 280.9, 294.7, 285.6, 288.6, 384.4)  
CaloriesPerServingSD <- c(124.8, 124.2, 116.2, 117.7, 118.3, 122.0, 168.3)  
ServingsPerRecipeMean <- c(12.9, 12.9, 13.0, 12.7, 12.4, 12.4, 12.7)  
ServingsPerRecipeSD <- c(13.3, 13.3, 14.5, 14.6, 14.3, 14.3, 13.0)  
sample.size <- 18  
tenth.increment <- 0.10  
hundredth.increment <- 0.100  
idx.1936 <- 1  
idx.2006 <- length(CaloriesPerRecipeMean)  
idxs36\_07 <- c(idx.1936,idx.2006)  
alpha=0.05  
  
  
CookingTooMuch.dat <- data.frame(  
 Year=Year,  
 CaloriesPerRecipeMean = CaloriesPerRecipeMean,  
 CaloriesPerRecipeSD = CaloriesPerRecipeSD,  
 CaloriesPerServingMean = CaloriesPerServingMean,  
 CaloriesPerServingSD = CaloriesPerServingSD,  
 ServingsPerRecipeMean = ServingsPerRecipeMean,  
 ServingsPerRecipeSD = ServingsPerRecipeSD  
)

Similar restrictions from Homework 4 and 5 apply. In this homework you wil be expected to show some proficiency coding decision and iteration constructs. Use if statements where you might otherwise have used binary array indices, and use for/do loops where you otherwise have used vector operations.

# Exercise 1

Recreate the table from Homework 5, Exercise 1. However, for this table, include only the unique and non-trivial (difference not equal 0) pairs. The table you create here will have the same columns as the previous exercise (Mean1, Mean2, SD1, SD2, CohenD). It should look something like:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Year1 | Year2 | Mean1 | Mean2 | SD1 | SD2 | CohenD |
| 1946 | 1936 | 2122.3 | 2123.8 | 1002.3 | 1050.0 |  |
| 1951 | 1936 | 2089.9 | 2123.8 | 1009.6 | 1050.0 |  |
| 1963 | 1936 | 2250.0 | 2123.8 | 1078.6 | 1050.0 |  |
| … | … | … | … | … | … | … |
| 1951 | 1946 | 2089.9 | 2122.3 | 1009.6 | 1002.3 |  |
| 1963 | 1946 | 2250.0 | 2122.3 | 1078.6 | 1002.3 |  |
| … | … | … | … | … | … | … |
| 2006 | 1997 | 3051.9 | 2249.6 | 1496.2 | 1094.8 |  |

Write two nested loops, the outer loop iterating over and the inner loop over . Set individual table elements by indexing CaloriesPerRecipeMean,CaloriesPerRecipeSD and Year.

Some suggestions: You can start with table filled with 0 or NA values, with the number of rows equal to the number of unique combinations among the treatments. You could also start with empty vectors (Mean1 <- c()), concatenate values (Mean1 <- c(Mean1, current.M1)) as you iterate over and , then create a table with these vectors. A third option might be to create two index vectors as you iterate over and and use these vectors to index means, standard deviations and year. In any case, this exercise requires the use of two nested loops.

Show your code, and print your table.

#Assigning values for use in nested for loop  
  
  
Year <- CookingTooMuch.dat$Year  
Mean <- CookingTooMuch.dat$CaloriesPerRecipeMean  
SD <- CookingTooMuch.dat$CaloriesPerRecipeSD  
  
Year <- Year[!is.na(Year)]  
Mean <- Mean[!is.na(Mean)]  
SD <- SD[!is.na(SD)]  
  
  
  
  
Year1 <- matrix (nrow = 21, ncol = 0)  
Year2 <- matrix (nrow = 21, ncol = 0)  
Mean1 <- matrix (nrow = 21, ncol = 0)  
Mean2 <- matrix (nrow = 21, ncol = 0)  
SD1 <- matrix (nrow = 21, ncol = 0)  
SD2 <- matrix (nrow = 21, ncol = 0)  
CohenD <- matrix (nrow = 21, ncol = 0)  
  
CPR.df <- data.frame (Year1,Year2,Mean1,Mean2,SD1,CohenD)  
CPR.df

## data frame with 0 columns and 21 rows

#Initializing values for outer and inner for loops  
i <- idx.1936  
j <- i+1  
k <- idx.2006  
  
for (i in 1:k-1) {  
 for (j in i+1:k) {  
 {   
 c(Year1,Year2[j])  
 c(Year2,Year1[i])  
 c(Mean1,Mean2[j])  
 c(Mean2,Mean1[i])  
 c(SD1,SD2[j])  
 c(SD2,SD1[i])  
 }   
 }  
}  
  
Year1 <- Year1[!is.na(Year1)]  
Year2 <- Year2[!is.na(Year2)]  
Mean1 <- Mean1[!is.na(Mean1)]  
Mean2 <- Mean2[!is.na(Mean2)]  
SD1 <- SD1[!is.na(SD1)]  
SD2 <- SD2[!is.na(SD2)]  
  
  
#Cohen's d function  
d\_12 <- function(m\_1,m\_2,s\_1,s\_2) {  
d\_12\_var <- (abs(m\_1-m\_2))/(sqrt((s\_1^2 + s\_2^2)/2))  
return(d\_12\_var)  
}  
  
#Cohen wrapper function from previous assignment  
cohen.wrapper <- function(table.row) {  
return(d\_12(table.row[3],table.row[4],table.row[5],table.row[6]))  
}  
  
# using Apply function to compute CohenD column  
CPR.df$CohenD <- apply(CPR.df,1,cohen.wrapper)  
  
#Adding new column to data frame  
cbind(CPR.df,CohenD)

## CohenD  
## 1 NA  
## 2 NA  
## 3 NA  
## 4 NA  
## 5 NA  
## 6 NA  
## 7 NA  
## 8 NA  
## 9 NA  
## 10 NA  
## 11 NA  
## 12 NA  
## 13 NA  
## 14 NA  
## 15 NA  
## 16 NA  
## 17 NA  
## 18 NA  
## 19 NA  
## 20 NA  
## 21 NA

# Exercise 2.

In Homework 4 and 5, we calculated sums of likelihood over series of . This provides an approximate integral, using the Newton-Cotes formula

with , is the number of in the series, step size and weight .

We will improve this approximation by iterating over successive approximations of over series of with increasingly smaller step sizes, using your likelihood function from the previous homework.

## Part a.

Let . Calculate by summing over $L(\bf{X\_0})$, where is a series from incremented by . Multiply this sum by for an approximate . Use and for this exercise.

#Initializing values for function L\_0  
mu <- 0  
sigma <- 1  
i <- 0  
h\_0 <- tenth.increment  
  
#Generating sequence x for use with function L\_0  
X\_0 <- seq(from=(mu-sigma),to=(mu+sigma),by=h\_0)  
X\_0 <- X\_0[!is.na(X\_0)]  
k <- length(X\_0)  
sum.x <- 0  
  
# Creating function L\_0  
L\_0 <- function(X\_0) {  
 for (i in 1:k) {  
 sum.x <- sum.x + (1/(sigma \* sqrt(2 \* pi))) \* (exp(-((X\_0[i]-mu)^2)/(2\*sigma^2)))  
 }  
 mul.x <- sum.x \* tenth.increment   
 return(mul.x)  
}  
  
#Executing function L\_0 on sequence x  
L\_0(X\_0)

## [1] 0.7064831

## Part b.

Let . Create a second series by setting . Compute from this series as in part a.

#Initializing values for function L\_1  
mu <- 0  
sigma <- 1  
h\_1 <- h\_0/2  
i<-1  
  
#Generating sequence x for use with function L\_1  
X\_1 <- seq(from=(mu-sigma),to=(mu+sigma),by=h\_1)  
X\_1 <- X\_1[!is.na(X\_1)]  
k <- length(X\_1)  
sum.x <- 0  
  
# Creating function L\_1  
L\_1 <- function(X\_1) {  
 for (i in 1:k) {  
 sum.x <- sum.x + (1/(sigma \* sqrt(2 \* pi))) \* (exp(-((X\_1[i]-mu)^2)/(2\*sigma^2)))  
 }  
 mul.x <- sum.x \* h\_1  
 return(mul.x)  
}  
#Executing function L\_0 on sequence x  
L\_1(X\_1)

## [1] 0.6946872

## Part c.

Compute . If , your sequence of iterations has converged on a solution for . Finish with Part d. Otherwise, increment , let . Create the next series and compute the next .

Hint: code this first as a for loop of a small number of until you know your code will converge toward a solution. I’ve found is sufficient.

#Initializing values for function L\_2  
mu <- 0  
sigma <- 1  
h\_0 <- tenth.increment  
h\_1 <- h\_0/2  
h\_2 <- h\_1/2  
h\_3 <- h\_2/2  
h\_4 <- h\_3/2  
h\_5 <- h\_4/2  
h\_6 <- h\_5/2  
h\_7 <- h\_6/2  
h\_8 <- h\_7/2  
h\_9 <- h\_8/2  
h\_10 <- h\_9/2  
h\_11<- h\_10/2  
h\_12<- h\_11/2  
h\_13<- h\_12/2  
i.L1<-1  
i.L2<-2  
i.L3<-3  
i.L4<-4  
i.L5<-5  
i.L6<-6  
i.L7<-7  
i.L8<-8  
i.L9 <-9  
i.L10 <- 10  
i.L11 <- 11  
i.L12 <- 12  
i.L13 <- 13  
  
#Generating sequence x for use with function L\_13  
X\_13 <- seq(from=(mu-sigma),to=(mu+sigma),by=h\_13)  
X\_13 <- X\_13[!is.na(X\_13)]  
k <- length(X\_13)  
sum.x <- 0  
  
# Creating function L\_13  
L\_13 <- function(X\_13) {  
 for (i in 1:k) {  
 sum.x <- sum.x + (1/(sigma \* sqrt(2 \* pi))) \* (exp(-((X\_13[i.L13]-mu)^2)/(2\*sigma^2)))  
 }  
 mul.x <- sum.x \* h\_13  
 return(mul.x)  
}  
#Executing function L\_0 on sequence x  
L\_13(X\_13)

## [1] 0.4840153

#Generating sequence x for use with function L\_12  
X\_12 <- seq(from=(mu-sigma),to=(mu+sigma),by=h\_12)  
X\_12 <- X\_12[!is.na(X\_12)]  
k <- length(X\_12)  
sum.x <- 0  
  
# Creating function L\_12  
L\_12 <- function(X\_12) {  
 for (i in 1:k) {  
 sum.x <- sum.x + (1/(sigma \* sqrt(2 \* pi))) \* (exp(-((X\_12[i.L12]-mu)^2)/(2\*sigma^2)))  
 }  
 mul.x <- sum.x \* h\_12  
 return(mul.x)  
}  
#Executing function L\_2 on sequence x  
L\_12(X\_12)

## [1] 0.4840773

0.4840153-0.4840773

## [1] -6.2e-05

abs(0.4840153-0.4840773) < 0.0001

## [1] TRUE

# Part d

Report , , and . Compare your final to

Results for final L\_i were as follows:

L\_i = 0.4840153 i = 13 n = 163841 h = 1.220703e-05

pnorm(1, lower.tail = TRUE)-pnorm(-1, lower.tail = TRUE)

## [1] 0.6826895

Is your within of this value?

Based on the following calcuation of |0.4840153 - 0.6826895| = 0.1986742, it would appear that the final L was not within 0.0001 of 0.6826895.

You might find it interesting to produce staircase plots for the first 2-4 iterations (plot vs on one graph). You might also find it interesting to plot or versus or . You can create vectors to hold the intermediate steps. How many iterations might it take to get within 0.000001 of the expected value from R?

# Exercise 3

## Part a.

Write a function to compute mean, standard deviation, skewness and kurtosis from a single vector of numeric values. You can use the built-in mean function, but must use one (and only one) for loop to compute the rest. Be sure to include a check for missing values.

See <https://www.itl.nist.gov/div898/handbook/eda/section3/eda35b.htm> for formula for skewness and kurtosis. This reference gives several definitions for both skewness and kurtosis, you only need to implement one formula for each. Note that for computing skewness and kurtosis, standard deviation is computed using as a divisor, not .

Your function should return a list with Mean, SD, Skewness and Kurtosis. If you use IML, you will need to implement this as a subroutie and use call by reference; include these variables in parameter list.

#generate sequence for vector.x  
vector.x <- seq(from=0,to=9,by=1)  
  
k <- length(vector.x)  
  
# mean function for vector  
vector.mean <- function(vector.input){  
return (mean(vector.input))  
   
}  
vector.mean(vector.x)

## [1] 4.5

#standard deviation function for vector  
vector.sd <- function (vector.input) {  
 for (i in 1:k) {  
 vector.input <- vector.input + i  
 }  
sd <- sqrt(sum((vector.input-vector.mean(vector.input))^2/(k-1)))  
return(sd)  
}  
vector.sd(vector.x)

## [1] 3.02765

#skewness function for vector  
vector.skewness <- function(vector.input) {  
 vector.median.length <- 0  
 for (i in 1:k) {  
 vector.median.length <- vector.median.length + i  
 }  
 vector.median <- (vector.median.length + 1)/2  
 skewness <- 3 \* ((vector.mean(vector.input) - vector.median)/vector.sd(vector.input))  
 return(skewness)  
}  
vector.skewness(vector.x)

## [1] -23.28538

#kurtosis function for vector  
vector.kurtosis <- function(vector.input) {  
 kurtosis.sum <-0  
 for (i in 1:k) {  
 kurtosis.sum <- kurtosis.sum +(vector.input[i]-vector.mean(vector.input))^4  
 }  
 kurtosis <- ((kurtosis.sum/k)/(vector.sd(vector.input)^4)-3)  
 return(kurtosis)  
}  
vector.kurtosis(vector.x)

## [1] -1.561636

## Part b.

Use this to compute skewness and kurtosis for Calories per Serving and Servings per Recipe, 1938 or 2006 from the Joy of Cooking data set. You will need to read in the file provided for lecture for this exercise, but you won’t need to upload this file to D2L.

PathToJoy = "C:/Users/drewm/Documents/GitHub/code-stat700/JoyofCooking.csv"  
JoyofCooking.dat <- read.csv(PathToJoy,header=TRUE)  
  
#Assigning vectors to variables and removing NA values  
CPS2006 <- JoyofCooking.dat$CaloriesperServing2006  
CPS2006 <- CPS2006[!is.na(CPS2006)]  
SPR2006 <- JoyofCooking.dat$ServingsperRecipe2006  
SPR2006 <- SPR2006[!is.na(SPR2006)]  
  
#Skewness function  
vector.skewness(CPS2006)

## [1] 1.168241

vector.skewness(SPR2006)

## [1] -4.780096

#Kurtosis functions  
vector.kurtosis(CPS2006)

## [1] -2.996289

vector.kurtosis(SPR2006)

## [1] -2.998903

## Part c.

If you wish, compare your function results with the skewness and kurtosis in the moments package.

# Exercise 4

Write a function to compute Euclids algorithm for GCD.

Given positive integers and , the greatest common denominator can be found by

## Part a

Implement this using a while loop.

#While loop GCD function  
GCD <- function(a, b) {  
while(b != 0) {  
gcd <- b  
b <- a %% b  
a <- gcd  
}  
return(gcd)  
}  
  
#While Loop GCD function test  
GCD(18,48)

## [1] 6

GCD(19,48)

## [1] 1

GCD(20,48)

## [1] 4

## Part b

Implement this using recursion.

#Recursive GCD function  
GCD <- function(a,b) {  
 gcd <- a%%b;  
 return(ifelse(gcd, GCD(b,gcd), b))  
}  
  
#Recursive GCD function test  
GCD(18,48)

## [1] 6

GCD(19,48)

## [1] 1

GCD(20,48)

## [1] 4

Test both implementations with , and

If you choose SAS for this exercise, you may need to demonstrate if IML or the macro languages supports this kind of recursion.

# Exercise 5

Repeat the analysis from Exercise 5, Homework 4, but this time implement calculating and using iteration, not array functions.

## Part a

Starting with CaloriesPerServingMean and CaloriesPerServingSD, compute the summations below using loops:

and

## Part b

Calculate F-ratio and a for this , using the distribution with and degrees of freedom. Use .

## Part c.

Print , , and and compare these values to the values from Homework 4.

# Exercise 6

Modify your Tukey HSD function from previous Homework to accept three parameters, s, n and optional k and alpha. Check for the following conditions:

* If s and n are length 1, assume s is pooled standard deviation; use k as the number of comparisons and compute degrees of freedom as .
* if s is length 1 and length n > 1,assume s is pooled standard deviation, compute a harmonic mean for n, and use as d.f. Use length n as the number of means, ignore the k parameter.
* if s is length > 1 and length n = 1, compute a pooled standard deviation and use n as a common sample size. Use for d.f., ignore the k parameter.
* if length s = length n, compute a pooled standard deviation and harmonic mean, ignore the k parameter.
* if both length s > 1 and length n > 1 but length s is not equal to length n, then return NA.

Test your function with the following sets of parameters:

tukey.hsd(128.5,18,7)  
tukey.hsd(128.5,c(rep(18,7)))  
tukey.hsd(128.5,c(17,18,17,18,17,18,17))  
tukey.hsd(CaloriesPerServingSD,18)  
tukey.hsd(CaloriesPerServingSD,c(rep(18,7)))  
tukey.hsd(CaloriesPerServingSD,c(rep(18,6)))

The results should be nearly identical (to round error) for four of the six. Another should be slightly larger, and one should return NA.