HW 3 More on R

HW: Wylan Gao

STAT 5400

Due: Sep 20, 2024 9:30 AM

Submit your solutions as an .Rmd file and accompanying .pdf file. Include all the **relevant** R code and output. With the `echo` option in R markdown, you opt to hide some R code that is not very relevant (but still run it to generate the document).

Always comment on your result whenever it is necessary.

Problems

1. The map function in `tidyverse`

Redo Question 4 in HW 1, say, write your own factorial function. At this time, explore `map_dbl` function in the R package `tidyverse`.

```
library(tidyverse)
```

```
## Warning: package 'tidyverse' was built under R version 4.2.3
```

```
## Warning: package 'ggplot2' was built under R version 4.2.3
```

```
## Warning: package 'tibble' was built under R version 4.2.3
```

```
## Warning: package 'tidyr' was built under R version 4.2.3
```

```
## Warning: package 'readr' was built under R version 4.2.3
```

```
## Warning: package 'purrr' was built under R version 4.2.3
```

```
## Warning: package 'dplyr' was built under R version 4.2.3
```

```
## Warning: package 'stringr' was built under R version 4.2.3
```

```
## Warning: package 'forcats' was built under R version 4.2.3
```

```
## Warning: package 'lubridate' was built under R version 4.2.3
```

```
## ── Attaching core tidyverse packages ──────────────────────── tidyverse
2.0.0 ──
## ✓ dplyr     1.1.4     ✓ readr     2.1.5
## ✓ forcats   1.0.0     ✓ stringr   1.5.1
## ✓ ggplot2   3.5.1     ✓ tibble    3.2.1
## ✓ lubridate 1.9.3     ✓ tidyr     1.3.1
## ✓ purrr     1.0.2
```

```
## ── 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
```

```
library(purrr)
library(microbenchmark)

# Custom factorial function using recursion/iteration
factorial_func <- function(n) {
  if (n < 0) { #this is the base case
    stop("Factorial is not defined for negatives.")
  } else if (n == 0) {
    return(1)  # 0! is 1
  } else {
    return(n * factorial_func(n - 1))  # This is the recursive step.
  }
}

# Create a vector of numbers to calculate factorial
numbers <- 1:10  # Factorials from 1! to 10!

# This is using the map_dbl function.
custom_factorial_results <- map_dbl(numbers, factorial_func)

# Compare with the built-in factorial function
builtin_factorial_results <- map_dbl(numbers, factorial)

# Print results for comparison
print(custom_factorial_results)
```

```
##  [1]       1       2       6      24     120     720    5040   40320
362880
## [10] 3628800
```

```
print(builtin_factorial_results)
```

```
##  [1]       1       2       6      24     120     720    5040   40320
362880
## [10] 3628800
```

```
# Benchmarking
microbenchmark(
  custom = map_dbl(numbers, factorial_func),
  builtin = map_dbl(numbers, factorial),
  times = 10000
)
```

```
## Unit: microseconds
##     expr  min   lq     mean median   uq    max neval
##   custom 48.3 51.1 60.99837   55.2 63.9 5764.2 10000
##  builtin 27.1 29.2 34.94427   31.6 36.2 2568.7 10000
```

The custom function run time is slower.

2. The skewness function.

Download and install the R package `moments`. Find the following `skewness` function.

```
library(moments)
"skewness" <-
function (x, na.rm = FALSE)
{
    if (is.matrix(x))
        apply(x, 2, skewness, na.rm = na.rm)
    else if (is.vector(x)) {
    if (na.rm) x <- x[!is.na(x)]
    n <- length(x)
     (sum((x-mean(x))^3)/n)/(sum((x-mean(x))^2)/n)^(3/2)
    }
    else if (is.data.frame(x))
        sapply(x, skewness, na.rm = na.rm)
    else skewness(as.vector(x), na.rm = na.rm)
}

calc_skew <- function(data) {
  # This handles NA values.
  data <- na.omit(data)

  # This calculates the skew for vectors.
  if (is.vector(data)) {
    length_n <- length(data)
    mean_data <- mean(data)
    sd_data <- sd(data)
    skewness <- sum((data - mean_data)^3) / length_n / (sd_data^3)
    return(skewness)
  }

  # This is for matrices and dataframes.
  if (is.matrix(data) || is.data.frame(data)) {
    skewness <- apply(data, 2, function(col) {
      col <- na.omit(col)
      length_n <- length(col)
      mean_data <- mean(col)
      sd_data <- sd(col)
      sum((col - mean_data)^3) / length_n / (sd_data^3)
    })
    return(skewness)
  }
```

```r
  stop("Input must be a vector, matrix, or dataframe.")
}

# Test the function with symmetric and positively skewed data
symmetric_data <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
positive_skew_data <- c(1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6,
6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9,
9, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10)

calc_skew(symmetric_data)

## [1] 0

calc_skew(positive_skew_data)

## [1] -0.556031

install.packages("moments")

## Warning: package 'moments' is in use and will not be installed

install.packages("microbenchmark")

## Warning: package 'microbenchmark' is in use and will not be installed

library(moments)
library(microbenchmark)

# Define the skewness function from the moments package
moments_skew <- function(data) {
  skewness(data, na.rm = TRUE)
}

# Compare the performance
benchmark_results <- microbenchmark(
  calc_skew(symmetric_data),
  moments_skew(symmetric_data),
  calc_skew(positive_skew_data),
  moments_skew(positive_skew_data),
  times = 1000
)

print(benchmark_results)

## Unit: microseconds
##                               expr  min   lq    mean median    uq     max
neval
##         calc_skew(symmetric_data) 17.4 18.7 23.6060   19.7 20.90  2681.3
1000
##      moments_skew(symmetric_data) 10.7 11.8 13.3747   12.4 13.30    78.3
```

```
1000
##     calc_skew(positive_skew_data) 19.1 20.5 22.9751   21.5 22.60   115.1
1000
##  moments_skew(positive_skew_data) 12.4 13.6 31.6094   14.3 15.25 15861.1
1000
```

The moments skew is more efficient. I cant figure out a way to be faster. THe packege is more effieicent. It is because it is written in C.

Write your own skewness function and use `microbenchmark` library to compare the speed with the `skewness` function in `moment`. Can your function be faster than that? Also make sure your function handles both vectors, matrices, and dataframes, as well as NA values in as the `moment` package.

3. Monty Hall problem

Back in 1970, a game show called "Let's Make a Deal" was hosted by Monty Hall. Suppose there are three doors on the stage. Behind one door there is a nice prize, while behind the other two there are worthless prizes. A contestant selects one door at random, and then Monty Hall opens one of the other two door to reveal a worthless prize. Monty Hall then expresses the willingness to trade the curtain that the contestant has chosen for the other door that has not been opened. Should the contestant switch curtains or stick with the one that she has?

- Determine the probability that she wins the prize if she switches.
- Use a simulation to verify your results.
- What if there are two prizes and four doors? What about m prizes and n doors, where $m < n$?

```r
n <- 10000

# simulation
monty_hall <- function(switch = TRUE) {
  doors <- 1:3
  prize_door <- sample(doors, 1)
  initial_choice <- sample(doors, 1)

  # Monty opens a door that is not the initial choice and not the prize door
  if (initial_choice == prize_door) {
    opened_door <- sample(setdiff(doors, initial_choice), 1)
  } else {
    opened_door <- setdiff(doors, c(initial_choice, prize_door))
  }

  if (switch) {
    final_choice <- setdiff(doors, c(initial_choice, opened_door))
  } else {
    final_choice <- initial_choice
  }
```

```
  return(final_choice == prize_door)
}

# Simulate the game with switching
results_switch <- replicate(n, monty_hall(switch = TRUE))
probability_switch <- mean(results_switch)

# Simulate the game without switching
results_stick <- replicate(n, monty_hall(switch = FALSE))
probability_stick <- mean(results_stick)

# Print the probabilities
cat("Probability of winning if switching:", probability_switch, "\n")

## Probability of winning if switching: 0.6633

cat("Probability of winning if staying:", probability_stick, "\n")

## Probability of winning if staying: 0.3387
```

The probaility the person switched is prize is not behind your first choice is 2/3. The probablity the prize is behind the other door given (not a first choice) is 1. The probabilty if the person is sticking is 1/3. Therefore the person should switch.

A = Event prize behind 1st choice. B = Prize is in the door excluding A. The probablity the prize is is not behind the 1st choice is 3/4. The probability that a prize is behind a different door given it is not in your first choice is 1. Therefore by multipling each other the probablty of switching is 3/4.

What if there are two prizes and four doors? What about m prizes and n doors, where m <n? We can conclude the algorithmn that the prize is not behind is your first choice is P() = n-1/n (is doors) Probablity is behind the door given it is not in your first choice is m / n - 2

By multipling these we we would get swithcing (n-1/n)* (m / n - 2)

4. Coding practice I

Check if a positive integer is power of 4.

```
foo <- function(n) {
  if (n <= 0) {
    return(FALSE)
  }
  while (n %% 4 == 0) {
    n <- n / 4
  }
  return(n == 1)
}

# Examples
foo(16)  # TRUE
```

```
## [1] TRUE

foo(31)

## [1] FALSE

foo(1)

## [1] TRUE

foo(81)# FALSE

## [1] FALSE
```

5. Coding practice II

Write your own function to merge and sort two sorted vectors without using any sorting function such as `sort` or `order`.

```
set.seed(5400)
foo <- function (a1, a2) {
  #this merges a list
  merged <- numeric(length(a1) + length(a2))
  i <- j <- k <- 1


  #Merge the two arrays by comparing elements from a1 and a2
  # This while loop continues until we exhaust one of the arrays (a1 or a2)
  while (i <= length(a1) && j <= length(a2)) {
    if (a1[i] <= a2[j]) {
      merged[k] <- a1[i]
      i <- i + 1
    } else {
      merged[k] <- a2[j]
      j <- j + 1
    }
    k <- k + 1
  }
  # If there are remaining elements in a1 (after a2 is exhausted), add them
to 'merged'
  while (i <= length(a1)) {
    merged[k] <- a1[i]
    i <- i + 1
    k <- k + 1
  }
  # If there are remaining elements in a2 merge them.
  while (j <= length(a2)) {
    merged[k] <- a2[j]
    j <- j + 1
    k <- k + 1
  }
```

```r
  return(merged)
  # input: foo(a1, a2)
  # output: sort(c(a1, a2)) # but you cannot use sort in your function
}
a1 <- sort(sample(8, 8, replace=TRUE))
a2 <- sort(sample(10, 8, replace=TRUE))
foo(a1, a2)
```

```
##  [1]  1  2  2  3  3  3  4  5  5  5  6  8  8  9 10 10
```