Manipulating Data with Apply Functions

Brian Yoo

October 29, 2017

Introduction

To further my learning on simple loops taught during lecture, I looked for particular applications of loop functions and also for functions that may simplify the entire process of manipulating data with the use of loop functions. That is when I stumbled upon apply functions, and I delved deeper into studying the different functions within the apply() family. Apply Functions are typically referred to by many as being looping functions, where an input of elements, lists, or vectors are repeatedly manipulated by a particular condition or function. In general, looping allows you to write shorter codes and commands when manipulating data repeatedly with the same sets of instructions. Therefore, looping will allow you to manipulate data multiple times without the hassle of having to write the same repetitive codes however many times, which will also help in the reduction of mistakes that may be made during the coding process. With the use of loop functions in mind, the apply() family is made up of functions that will allow you to do exactly what a looping function will do to manipulate data, without having to use any of the explicit coding for loops.

Motivation

The motivation behind using the functions of the apply() family is quite similar to the motivation of using looping functions. As looping functions are used to shorten code, save time, and also reduce errors that may result when writing repetitive coding for similar manipulation, the functions within the apply() family were created with a similar purpose in mind. The functions of the apply() family allow you to write less codes that will manipulate parts of a data, that would have conventionally required many more loop codes. Therefore, the use of these pre-existing functions will allow the user to shorten their code, making the codes a lot more readable and simple, and will leave less room for errors.

Types of Apply Functions

apply()

The apply() function is used when you want to apply a given function to parts of a matrix. Below is the inputs required to run the function apply.

```
str(apply)
```

```
## function (X, MARGIN, FUN, ...)
```

X is the input vector or array that you wish to manipulate.

MARGIN takes on value 1 or 2, where 1 refers to the row of the matrix and 2 refers to the column of the matrix. Therefore, by setting MARGIN equal to 1, 2, or both will allow the apply() function to determine how to manipulate the data.

FUN refers to the function that is to be applied to the vector.

An example of the use of apply function is shown below, where we will manipulate the matrix of numbers from 1 to 20.

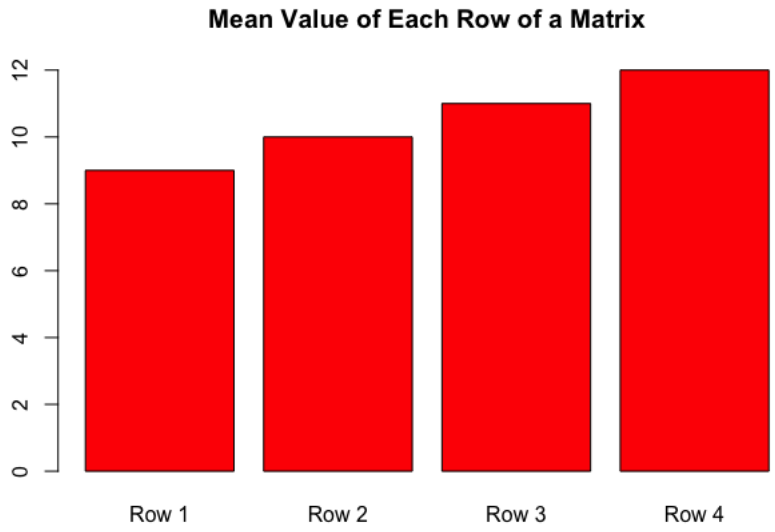
```
X = matrix(1:20, nrow = 4, ncol = 5)
apply(X, 1, mean)
```

```
## [1]  9 10 11 12
```

By setting the MARGIN = 1, the apply function found the average of all elements in the same row. Clearly, the apply function allows you to apply the mean function to each row without having to write seperate codes applying mean to each of the four rows.

You can use the apply function to graph the mean of each row, as shown below:

```
barplot(apply(X, 1, mean), main="Mean Value of Each Row of a Matrix", horiz=FALSE, col = 'red', names.arg=c("Row 1
", "Row 2", "Row 3", "Row 4"))
```



Additionally, another example is given below where the standard deviation of the columns of vector X is calculated using the apply function:

```
apply(X, 2, sd)
```

```
## [1] 1.290994 1.290994 1.290994 1.290994 1.290994
```

lapply()

The lapply() function is used to apply a function to all the elements of a list. As a result, the lapply() function will give your result in the form of a list. We will explore another function that allows you to get results in other forms as well.

```
str(lapply)
```

```
## function (X, FUN, ...)
```

The variables used in the lapply function is the same as the ones used in the apply function, with two differences. First, X does not have to be a matrix, but can be a list/set of numbers. Furthermore, as the function is applied to all elements, there is no need to specify the MARGIN value.

Let's explore an example applying lapply.

Set Y equal to the following array of numbers.

```
Y = list(a = 1:5, b = rnorm(11), c = sample(1:15, 5), d = seq(from = 15, to = 70, by = 5))
```

Now let's take the calculate the average of the four different array of numbers using only one command line.

```
lapply(Y, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 0.2504058
##
## $c
## [1] 7.8
##
## $d
## [1] 42.5
```

As seen above, the lapply function gives the results in a list, where the average of a, b, c, and d are calculated. Using the lapply function is clearly a lot more efficient than using multiple loop functions to calculate the mean of each array. Furthermore, given that the number of arrays increases to 100 or even to 1000, using the lapply function will prove to be much more concise, and result in less errors along the way.

Sapply()

Sapply is very similar to lapply, but different in that sapply will try to simplify the results given through lapply to a vector or a matrix. More specifically, if the list of results all contain elements with the length of 1, sapply will simplify that result into a vectore. Similarly, if the list of results are vectors with the same length greater than 1, the result will be simplified into a matrix. Therefore, additional computations and graphing with these results become more directly available, as the results are not in a list, but in the form of a vector or matrix.

In order to explore not only the effectiveness of the sapply function, but also its equivalence to using multiple loop functions, we will create two graphs comparing the average height of students in centimeters: one using loop functions and one using the sapply() function.

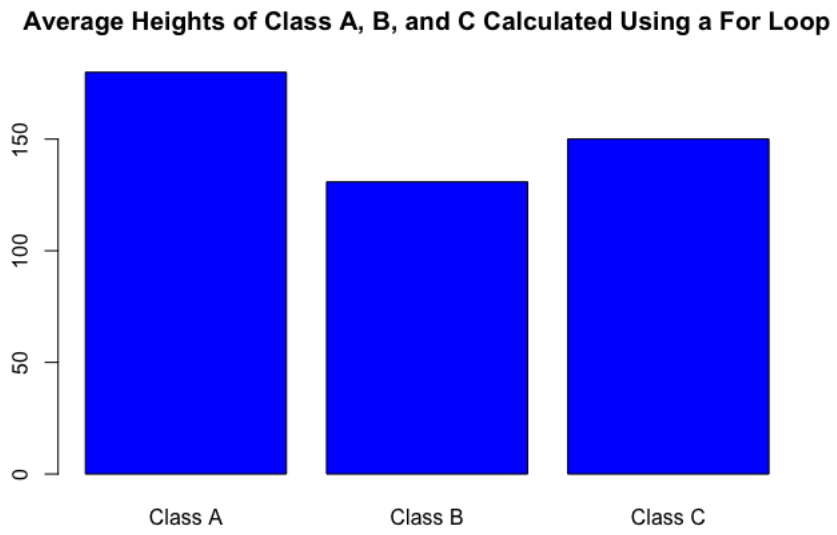
The heights of students in class A,B, and C are given below.

```
set.seed(74)
heightsA = runif(30, min = 170, max = 190)
set.seed(80)
heightsB = runif(30, min = 100, max = 170)
set.seed(84)
heightsC = runif(30, min = 140, max = 160)

Class_heights = data.frame(heightsA, heightsB, heightsC)
Class_heights
```

```
##    heightsA heightsB heightsC
## 1  184.0923 130.7729 155.2823
## 2  179.6001 139.1472 146.9196
## 3  173.9204 154.9930 157.4595
## 4  188.7394 105.1721 155.3580
## 5  187.3966 142.3152 144.5583
## 6  174.2806 100.5187 145.3464
## 7  170.4817 147.1379 142.5162
## 8  187.0959 144.2135 146.4096
## 9  184.0543 141.4639 152.0179
## 10 189.1596 152.1954 142.9317
## 11 186.3128 125.1036 154.6995
## 12 182.3750 114.2266 159.9915
## 13 188.8155 135.2777 157.4158
## 14 185.5100 158.7493 158.2548
## 15 171.0006 128.8614 140.5850
## 16 182.3460 113.4352 144.2083
## 17 173.9636 122.0058 141.8423
## 18 174.8502 110.7215 147.4080
## 19 171.2173 136.9995 141.6693
## 20 174.0005 164.7645 149.0378
## 21 178.7847 131.7279 141.4583
## 22 177.2621 116.6374 144.8329
## 23 184.6270 107.9948 147.0765
## 24 171.8470 134.4268 158.9602
## 25 176.0478 105.3041 153.8940
## 26 177.1547 108.6202 158.0406
## 27 177.0437 163.3667 154.6254
## 28 180.5539 130.7383 158.4407
## 29 183.2432 135.2609 154.7769
## 30 182.1367 122.3887 144.8133
```

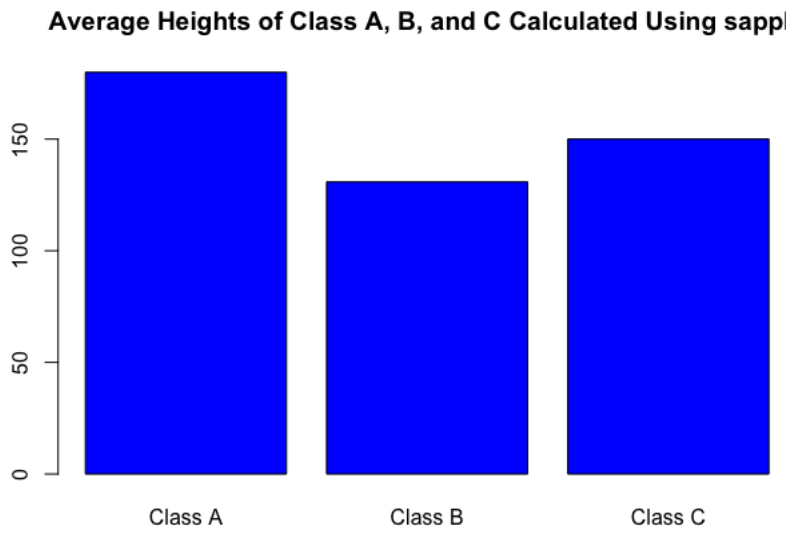
Calculate the average heights of three classes using a for loop

```
Class_mean_heights = rep(NA,3)
for (i in 1:length(Class_heights)) {
  Class_mean_heights[i] = mean(Class_heights[,i])
}

barplot(Class_mean_heights, main="Average Heights of Class A, B, and C Calculated Using a For Loop", horiz=FALSE,
col = 'blue', names.arg=c("Class A", "Class B", "Class C"))
```



Calculate the average heights of three classes using the sapply() function

```
Class_mean_heights2 = sapply(Class_heights, mean)

barplot(Class_mean_heights2, main="Average Heights of Class A, B, and C Calculated Using sapply", horiz=FALSE, col = 'blue', names.arg=c("Class A", "Class B", "Class C"))
```



Clearly, using the loop function and using sapply resulted in the same answers. But, the coding for using sapply was much simpler and concise, as sapply not only executed all the loop functions within the command, but also switched the results into a matrix, allowing the user to create a graph directly from the results of sapply.

mapply()

Unlike the past apply functions, mapply is able to apply the calculations over multiple arguments. Therefore, you can consider the mapply function to be the function that does multiple apply functions with one command. Let's take a look at the input variables for mapply.

```
str(mapply)
```

```
## function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

FUN refers to the function that is to be applied to arguments.

MoreArgs is the additional arguements that will run through the functions.

SIMPLIFY takes on TRUE or FALSE, where TRUE refers will make the mapply function to simplify the results (similar to what happens when the sapply function is used), and FALSE will negate the simplification of results.

USE.NAMES takes on a logical value as well, where it will use the names of the input arguments.

Let us now explore an example using the mapply function.

We will try applying the repeat function over multiple arguments for different amount of times. We will try to do the same operations without using and using the mapply function.

So repeat the number 5 six times, number 6 seven times, and number 7 eight times to create lists with the different number values.

Conventionally, the operations would have to require three seperate commands to do the above operations as shown below:

```
list(rep(5,6), rep(6,7), rep(7,8))
```

```
## [[1]]
## [1] 5 5 5 5 5 5
##
## [[2]]
## [1] 6 6 6 6 6 6 6
##
## [[3]]
## [1] 7 7 7 7 7 7 7 7
```

But, when using the mapply function, only one line of code is necessary, as shown below:

```
mapply(rep, 5:7, 6:8)
```

```
## [[1]]
## [1] 5 5 5 5 5 5
##
## [[2]]
## [1] 6 6 6 6 6 6 6
##
## [[3]]
## [1] 7 7 7 7 7 7 7 7
```

The solutions for both are the same, but the mapply function reduces having to write the rep() command three times by looping the rep() function three times within the mapply() function. Therefore, the mapply() function is clearly more concise and efficient than having to write out all the commands.

Conclusion

As shown in the examples given above, the various functions within the apply family allows you to manipulate data in a much more concise and efficient manner. As an example, we specifically explored the application of the apply() function, lapply() function, sapply() function, and the mapply() functions, where each function was able to manipulate an argument(s) according to the function given by the user. Therefore, I found these functions to be very useful, where knowing these functions and their proper applications could help you especially when dealing with a large data set. Furthermore, I find it very important and useful for users to always be exploring other options that would make their code more concise and efficient. This not only allows the code written by a user to be more accessible to others, but will also allow the user to be less error-prone and will save a lot of time for the user. Moreover, I believe that spending time to find more efficient methods of coding will in the long run, save more time for you. Therefore, I encourage you to be curious and look for various methods to make your code more concise and efficient; and, I hope that learning about these functions was able to help in your endeavors in creating more efficient codes.

References

Blog Posts

- http://adv-r.had.co.nz/Functional-programming.html#closures
- https://www.datacamp.com/community/tutorials/tutorial-on-loops-in-r
- https://www.r-bloggers.com/how-to-write-the-first-for-loop-in-r/

Videos

- https://www.coursera.org/learn/r-programming/lecture/t5iuo/loop-functions-lapply
- https://www.youtube.com/watch?v=f0U74ZvLfQo
- https://www.youtube.com/watch?v=ejVWRKidi9M
- https://www.youtube.com/watch?v=uL_LdYS-scQ