HW3_IS457_85

Mon Oct 1, 2018

Part 1. Start with "apply" function (9pts)

(1) Create a new matrix by the following codes, briefly but completely explain what each line is doing. (3pts)

Ans :

set.seed(457)

This statement sets the seed value for out random generator. This help in makes the results reproducible.

one_num <- sample(1:9,25,replace=TRUE)

The above code creates an integer vector called **one_num** and samples **25** values from the range **1:9**.

one_num

```
[1] 2 2 1 5 9 2 3 9 2 7 8 7 9 2 2 3 9 7 4 3 1 1 9 4 3
```

one_matrix <- matrix(one_num,ncol=5)

The above statement creates a matrix **one_matrix** from the vector **one_num** having 5 columns.

one_matrix

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    2    8    3    1
[2,]    2    3    7    9    1
[3,]    1    9    9    7    9
[4,]    5    2    2    4    4
[5,]    9    7    2    3    3
```

The above code gives us the desired matrix of elements.

(2) Use the "apply" function on one_matrix to answer questions(2)(3)(4).

Calculate the mean of each row. (1pt) Calculate the sum of each column. (1pt)

Ans :

apply(one_matrix, 1, mean)

```
[1] 3.2 4.4 7.0 3.4 4.8
```

The above code gives us a vector containing the mean of values for each row.

apply(one_matrix, 2, sum)

```
[1] 19 23 28 26 18
```

The above code gives us a vector containing the sum of values of each column.

(3) Find the difference between the biggest and the smallest number for each row. (2pts)

Ans :

apply(one_matrix, 1, max)-apply(one_matrix, 1, min)

```
[1] 7 8 8 3 7
```

As can be seen from the output above the code gives us the difference between the smallest and biggest value for each row.

(4) Calculate the sum of all numbers smaller than 5 for each column. (2pts)

Ans :

apply(apply(one_matrix, c(1, 2), function(x){ if(x<5) return (x) else return(0) }), 2, sum)

```
[1]  5  7  4 10  9
```

The output above shows that the code gives us the sum of all the numbers smaller than 5 for each column.

Part 2. Get familiar with "sapply" and "lapply" functions (6pts)

(5) Let's play with the "iris" dataset. Here's the command to load it:

data(iris)

Tell me the data types of each column(variable) by using "sapply" function (1pt)

Ans :

sapply(iris[,1], class)

```
  [1] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
  [8] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [15] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [22] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [29] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [36] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [43] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [50] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [57] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [64] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [71] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [78] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [85] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [92] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [99] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[106] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[113] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[120] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[127] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
```

```
[134] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[141] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[148] "numeric" "numeric" "numeric"
```

sapply(iris[,2], class)

```
  [1] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
  [8] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [15] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [22] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [29] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [36] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [43] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [50] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [57] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [64] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [71] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [78] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [85] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [92] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [99] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[106] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[113] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[120] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[127] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[134] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[141] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[148] "numeric" "numeric" "numeric"
```

sapply(iris[,3], class)

```
  [1] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
  [8] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [15] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [22] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [29] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [36] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [43] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [50] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [57] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [64] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [71] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [78] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [85] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [92] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [99] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[106] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[113] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[120] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[127] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[134] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[141] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[148] "numeric" "numeric" "numeric"
```

sapply(iris[,4], class)

```
  [1] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
  [8] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [15] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [22] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [29] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [36] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
```

```
 [43] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [50] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [57] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [64] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [71] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [78] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [85] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [92] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
 [99] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[106] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[113] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[120] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[127] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[134] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[141] "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"
[148] "numeric" "numeric" "numeric"
```

sapply(iris[,5], class)

```
  [1] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
  [9] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
 [17] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
 [25] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
 [33] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
 [41] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
 [49] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
 [57] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
 [65] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
 [73] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
 [81] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
 [89] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
 [97] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
[105] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
[113] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
[121] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
[129] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
[137] "factor" "factor" "factor" "factor" "factor" "factor" "factor" "factor"
[145] "factor" "factor" "factor" "factor" "factor" "factor"
```

As can be seen from the outputs above the datatype of columns 1-4 is **numeric** and the datatype for column 5 is **factor**.

(6) Do question(5) again and get the same result (check the data type) but use "lapply" function. (1pt) Based on the output format, briefly explain the difference between "sapply" and "lapply" functions. (2pts)

Ans :

lapply(iris[,1], class)

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

[[2]]
[1] "numeric"

[[3]]
[1] "numeric"

[[4]]
[1] "numeric"
```

```
[[5]]
[1] "numeric"

[[6]]
[1] "numeric"
```

lapply(iris[,2], class)

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

[[2]]
[1] "numeric"

[[3]]
[1] "numeric"

[[4]]
[1] "numeric"

[[5]]
[1] "numeric"

[[6]]
[1] "numeric"
```

lapply(iris[,3], class)

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

[[2]]
[1] "numeric"

[[3]]
[1] "numeric"

[[4]]
[1] "numeric"

[[5]]
[1] "numeric"

[[6]]
[1] "numeric"
```

lapply(iris[,4], class)

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

[[2]]
[1] "numeric"

[[3]]
[1] "numeric"

[[4]]
[1] "numeric"

[[5]]
[1] "numeric"
```

```
[[6]]
[1] "numeric"
```

lapply(iris[,5], class)

```
[[1]]
[1] "factor"

[[2]]
[1] "factor"

[[3]]
[1] "factor"

[[4]]
[1] "factor"

[[5]]
[1] "factor"

[[6]]
[1] "factor"
```

We can see from the output of question 5 and question 6 that the **sapply** function gives us a vector output whereas **lapply** function gives us list as an output. Both the functions apply the function in their parameter to each element in the object provided. **lapply** gives us the list of the same length provided to it in the input. **sapply** is a wrapper around **lapply** which provides a user-friendly vectorized output for the function applied to the list.

(7) Now create a new list from one_matrix by using the following codes.

list_1 <- list(a=one_matrix[,1],b=c(one_matrix[,2],one_matrix[,3]))

Take natural log of this list using the following code:

log(list_1)

Please briefly explain why the code above doesn't work. (1pt)
Now write code that takes the natural log of list_1 (using the apply family of functions). (1pt)

Ans :

list_1

```
$`a`
[1] 2 2 1 5 9

$b
 [1] 2 3 9 2 7 8 7 9 2 2
```

sapply(list_1, log)

```
$`a`
[1] 0.6931472 0.6931472 0.0000000 1.6094379 2.1972246

$b
 [1] 0.6931472 1.0986123 2.1972246 0.6931472 1.9459101 2.0794415 1.9459101
 [8] 2.1972246 0.6931472 0.6931472
```

The reason why **log(list_1)** statement gave an error because **log()** is a mathematical function which expects a **vector** input. The input we provided was a list of integer vectors. To get the desired output we need to make use of **sapply()** function to run the function **log()** on each of the integer elements**.**

Part 3. Try "tapply" function and its equivalents. (12pts)

We will use the data set "**mtcars**"; familiarize yourself with it first. Here is the code to load the data:
data(mtcars)

(8) Subset 4 columns "mpg", "hp", "wt" and "am" to a new data frame, named df_car. (1pt) In df_car, convert "am" to a factor variable with two levels: "automatic" and "manual" (2pts) (Hint: read help documentation of mtcars)

Ans :

df_car = mtcars[, c("mpg","hp", "wt", "am")]

df_car$am = factor(df_car$am, labels = c("automatic", "manual"))

str(df_car)

```
'data.frame':	32 obs. of  4 variables:
 $ mpg: num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ hp : num  110 110 93 110 175 105 245 62 95 123 ...
 $ wt : num  2.62 2.88 2.32 3.21 3.44 ...
 $ am : Factor w/ 2 levels "automatic","manual": 2 2 2 1 1 1 1 1 1 1 ...
```

The above code shows **am** variable having two factors “**automatic**” and “**manual**”.

(9) Use "tapply" function on df_car to find the mean of mpg by different am levels. (2pts)

Ans :

tapply(df_carmpg, df_caram, mean)

```
automatic    manual 
 17.14737  24.39231 
```

The above output shows the mean of mpg by different am levels.

(10) Look up the function "by". Obtain the mean of mpg, hp and wt, by different am levels, using only one function call. (2pts)

Ans :

by(df_car[, 1:3], df_car[,"am"], apply, 2, mean)

```
df_car[, "am"]: automatic
       mpg         hp         wt 
 17.147368 160.263158   3.768895 
------------------------------------------------------------ 
df_car[, "am"]: manual
```

```
      mpg        hp        wt
 24.39231 126.84615   2.41100
```

The above code obtains the mean of **mpg**, **hp**, **wt** by different **am** levels.

(11) Do question(10) again by using the "aggregate" function. You may need to look this up as well. (2pts)

Ans :

aggregate(df_car[,1:3], df_car["am"], mean)

```
am      mpg       hp       wt
1 automatic 17.14737 160.2632 3.768895
2    manual 24.39231 126.8462 2.411000
```

The aggregate function gives us the same output results.

(13) Same as question(10), but this time use the combination of "apply" and "tapply" functions to get the same results. (3pts)

Ans :

apply(df_car[1:3],2,tapply,df_car$am,mean)

```
               mpg       hp       wt
automatic 17.14737 160.2632 3.768895
manual    24.39231 126.8462 2.411000
```

We get the same output as before but using only the **apply()** and **tapply()** functions.

Part 4. More functions in "apply" family (4pts)

(14) "mapply" function is a multivariate version of "sapply". Create list_2 by following code:

list_2 <- list(a=one_matrix[,2],b=c(one_matrix[,3],one_matrix[,4]))

Add up corresponding elements in list_1 and list_2, then take natural log of it. (2pts)

Ans :

sapply(mapply("+", list_1, list_2),log)

```
$`a`
[1] 1.386294 1.609438 2.302585 1.945910 2.772589

$b
 [1] 2.302585 2.302585 2.890372 1.386294 2.197225 2.397895 2.772589 2.772589
 [9] 1.791759 1.609438
```

We make use of **mapply()** function to add the two lists and then **sapply()** function to get the log values.

(15) "rapply" function is used to apply a function to all elements of a list recursively. Create list_3 by following code:

list_3 <- list(aa=one_matrix[,1],b=c("this","is","character"))

Calculate the natural log of all integer in list_3. (2pts) Do not remove characters in the list or subsetting.

Ans :

rapply(list_3, log, classes = "integer", how = "replace")

```
$`aa`
[1] 0.6931472 0.6931472 0.0000000 1.6094379 2.1972246

$b
[1] "this"      "is"        "character"
```

As can be seen from the output above **rapply()** gives us the log values for only the sublist "**aa**".

Part 5. Linear regression (9pts)

(16) Look back in the "iris" dataset.

data(iris)

Fit a simple linear regression model using lm() to predict Petal.Length from Petal.Width. (2pts) How do you interpret the result of regression? Hint: interpret the two coefficients from the output of lm(). (2pts)

Ans :

lm(Petal.Length ~ Petal.Width, data = iris)

```
Call:
lm(formula = Petal.Length ~ Petal.Width, data = iris)

Coefficients:
(Intercept)  Petal.Width
      1.084        2.230
```

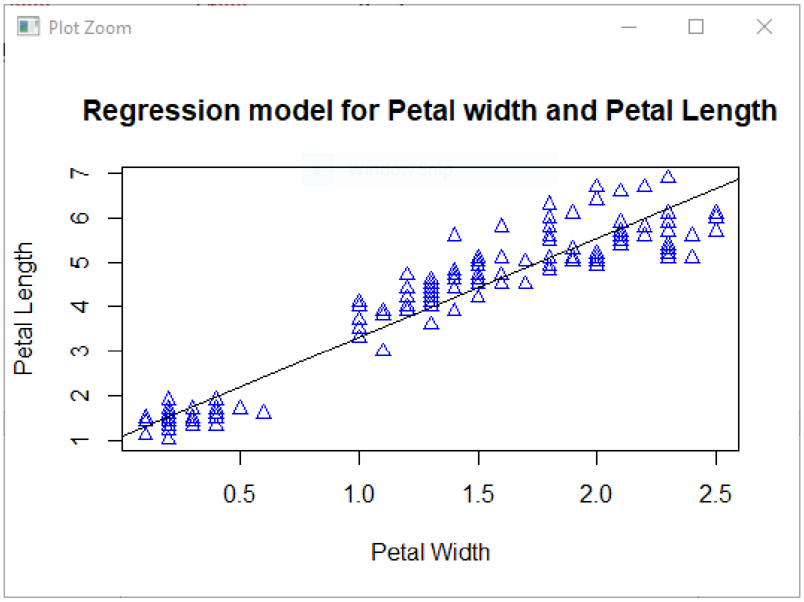
The intercept which in this case is the petal length gives us the expected value for the petal length by taking the petal width into consideration. In this case the expected value for the petal length is **1.084** cms. The second coefficient is the slope which give us the effect of the change in width has on the length of the petal. Here for every unit change in the width changes the length by **2.230** cms**.**

(17) Create a scatterplot with x-axis of Petal.Width and y-axis of Petal.Length. (2pt) Add the linear regression line you found above to the scatterplot. (1pt) Provide an interpretation for your plot. (2pts)

Ans :

plot(iris$Petal.Width,iris$Petal.Length, pch = 2, col = "red", main = "Regression model for Petal width and Pet al Length", xlab = "Petal Width", ylab = "Petal Length")

abline(lm(Petal.Length ~ Petal.Width, data = iris))



From the above plot it can be observed that as the size of the petal length increases as the width of the petal increases. We can see a positive correlation between these two variables. We can also observe that the values are clustered into two groups. The positive correlation between the width and length makes sense as any growth in a plant is usually evenly distributed and not concentrated to any particular characteristics.