R programming

Artür Manukyan

December 3, 2020

Matrices

- Matrices are used for many purposes in R

```
> m <- rnorm(12)
> m

 [1]  0.9494222  0.8166734  0.9671824 -1.7864953 -0.5218239 -0.8029565
 [7] -1.8051463 -0.8884292 -0.4819043 -1.2933338 -0.2161673 -1.4497220

> dim(m) <- c(3,4)
> m

          [,1]       [,2]       [,3]       [,4]
[1,] 0.9494222 -1.7864953 -1.8051463 -1.2933338
[2,] 0.8166734 -0.5218239 -0.8884292 -0.2161673
[3,] 0.9671824 -0.8029565 -0.4819043 -1.4497220
```

- or you can specify the dimensions with the matrix() function

```
> m <- rnorm(12)
> m

 [1] -1.83961512  0.04244126 -0.03166508 -0.47782611 -1.32690127 -1.16243158
 [7]  0.13809153  0.10379338  0.26860503  0.09375672  0.66246754  0.06701164

> m <- matrix(m,nrow=3,ncol=4,byrow=F)
> m

            [,1]       [,2]      [,3]       [,4]
[1,] -1.83961512 -0.4778261 0.1380915 0.09375672
[2,]  0.04244126 -1.3269013 0.1037934 0.66246754
[3,] -0.03166508 -1.1624316 0.2686050 0.06701164
```

Matrices

- Basic functions on matrices
  - nrow() and ncol() calls numbers of rows and columns
  - t() calls the transpose of the matrix
  - rownames() and colnames() are the names of columns and rows

```
> m <- rnorm(12)
> m <- matrix(m,nrow=3,ncol=4,byrow=F)
> nrow(m)
```

```
[1] 3
```

```
> ncol(m)
```

```
[1] 4
```

```
> colnames(m) <- c("A","B","C","D")
> m
```

```
               A          B         C          D
[1,]  0.12394796  1.3154234 0.9968355  0.4413296
[2,]  0.02655774 -0.8579061 0.8457881  0.9018971
[3,] -0.47550780  0.3490124 0.2132153 -0.5657494
```

Merging Vectors

- rbind() and cbind() functions merges vectors or matrices into matrices

```
> X1 <- rnorm(12)
> X2 <- 1:12
> m <- cbind(X1,X2)
> m
```

```
              X1 X2
 [1,]  0.7179971  1
 [2,]  1.9253669  2
 [3,] -0.1676504  3
 [4,] -0.7827515  4
 [5,]  1.0876736  5
 [6,] -1.5190781  6
 [7,] -0.9142900  7
 [8,] -1.6296740  8
 [9,]  0.2870266  9
[10,]  1.0465450 10
[11,]  1.0162229 11
[12,] -1.8545128 12
```

Merging Vectors

- Number of columns should be equal for rbind
- Likewise, number of rows should be equal for cbind

```
> data_1 <- matrix(rnorm(12),nrow=3,ncol=4,byrow=T)
> data_2 <- matrix(rnorm(16),nrow=4,ncol=4,byrow=F)
> data_new <- rbind(data_1,data_2)
> data_new
```

```
           [,1]       [,2]         [,3]       [,4]
[1,] -1.6178251 -0.3233475 -0.843480201 -1.2717424
[2,] -0.7878121 -0.9117782 -0.191396380 -0.9507478
[3,] -1.2227275 -1.1741398 -1.108056384 -0.8678325
[4,]  0.4108297 -0.7407136 -0.002151329 -1.0023822
[5,]  2.4149035  0.2027108 -1.504017600 -0.3793893
[6,]  1.0244861  0.5286680  1.392723137  1.2701646
[7,]  0.2284962 -0.4862941  1.283637517 -0.7255216
```

Indexing Matrices

```
> m <- matrix(rnorm(12),nrow=3,ncol=4,byrow=F)
> m
```

```
           [,1]       [,2]       [,3]      [,4]
[1,] -0.5267084 -0.4637852 1.29983795 -1.232867
[2,]  0.0389733 -2.6034900 0.00954156 -1.170770
[3,]  0.5537107  0.2021856 0.43645742 -1.361930
```

```
> index_row <- 1:3
> index_col <- c(1,3,4)
> m[index_row,index_col]
```

```
           [,1]       [,2]      [,3]
[1,] -0.5267084 1.29983795 -1.232867
[2,]  0.0389733 0.00954156 -1.170770
[3,]  0.5537107 0.43645742 -1.361930
```

Indexing Matrices: Examples

- Let m be a matrix with 7 rows where
  3 rows are from the "training set"
  4 rows are from the "test set"

```
> set_m <- rep(c(1,2),c(3,4))
> set_m <- factor(set_m,levels = 1:2
+                  ,labels = c("training","test"))
> m <- matrix(rnorm(21),nrow=7,ncol=3,byrow=T)
> m[set_m=="training",]
```

```
           [,1]       [,2]       [,3]
[1,] -1.1759010  0.1579945  0.8915471
[2,]  0.6968994 -0.4536412  0.6820949
[3,] -0.5826836 -0.6340029 -0.6915423
```

Indexing Matrices: Examples

```
> m <- cbind(m,set_m)
> m
                                           set_m
[1,] -1.1759010  0.15799454  0.8915471     1
[2,]  0.6968994 -0.45364116  0.6820949     1
[3,] -0.5826836 -0.63400293 -0.6915423     1
[4,]  0.8240483  0.53835903  0.6815398     2
[5,]  0.8571331 -0.01819739  1.4327240     2
[6,] -0.7337359 -0.15168591 -0.4017226     2
[7,] -1.4878936 -0.72207070 -0.7874091     2
> colnames(m) <- c("X1","X2","X3","set")
> m[set_m=="training",]
             X1         X2         X3 set
[1,] -1.1759010  0.1579945  0.8915471   1
[2,]  0.6968994 -0.4536412  0.6820949   1
[3,] -0.5826836 -0.6340029 -0.6915423   1
```

Lists

- A list is an ordered collection of components
- Components may be arbitrary R objects (data frames, vectors, lists, etc.)
- Single bracket notation for sublists
- Double bracket notation for individual components
- Construct using the function list()

```
> L1 <- list(name="Fred", wife="Mary",
+             no.children=3, child.ages=c(4,7,9))
> L1[[3]]

[1] 3
```

Lists

```
> L1 <- list(name="Fred", wife="Mary",
+            no.children=3, child.ages=c(4,7,9))
> L1[[3]]

[1] 3

> L1[c(1,3)]

$name
[1] "Fred"

$no.children
[1] 3

> L1$no.children

[1] 3
```

Lists: Example

```
> data_new <- matrix(rnorm(21),nrow=7,ncol=3,byrow=T)
> set_data <- rep(c(1,2),c(3,4))
> set_data <- factor(set_data,levels = 1:2,
+                      labels = c("training","test"))
> table_set <- table(set_data)
> dim_data <- dim(data_new)
> data_info <- list(data = data_new, set = set_data,
+                    set_info = table_set, dim = dim_data)
```

Lists: Example

```
> names(data_info)
[1] "data"      "set"       "set_info" "dim"
> data_info$set_info
set_data
training     test
       3        4
> data_info$dim
[1] 7 3
> data_info$set
[1] training training training test     test     test     t
Levels: training test
```

Lists: Example

```
> data_info$data
            [,1]        [,2]       [,3]
[1,]  0.8714042 -0.97127687  1.2170574
[2,]  0.4545998  0.27865005  0.9688815
[3,]  1.3821152  0.13976678 -1.4253669
[4,] -1.8148906  0.86140204  0.5326529
[5,] -1.0845851  1.41654947 -0.4814033
[6,]  0.3269725 -0.81030249  1.2556171
[7,] -0.5696174  0.06839643  2.2507647
```

If Statements

```
> x <- 3
> if(x > 4){
+   cat("x is bigger than 4")
+ } else {
+   cat("x is smaller or equal than 4")
+ }

x is smaller or equal than 4
```

If Statements

```
> x <- 3
> if(x > 4){
+   cat("x is bigger than 4")
+ } else if(x==4){
+   cat("x is equal to 4")
+ } else {
+   cat("x is smaller than 4")
+ }
x is smaller than 4
```

Loops

```
> x <- rnorm(12)
> x
 [1]  0.52143281 -1.93454715 -0.02574563 -0.84363691  0.48223439 -0.365
 [7]  0.08245287  0.17916581  0.52421701  0.89163523  1.72646791  1.473
> for(i in 1:length(x))
+ {
+   if(x[i] > 0){
+     x[i] <- 1
+   } else {
+     x[i] <- 0
+   }
+ }
> x
 [1] 1 0 0 0 1 0 1 1 1 1 1 1
```

Vectorization

- A vectorised version of the if statement is ifelse() function
- This is useful if you want to perform some action on every element of a vector that satisfies some condition.
- If it is possible to write code with less loops and ifs, go for it!
- It should be much faster

```
> x <- rnorm(12)
> x

 [1]  1.151379534 -0.546015765 -1.583285747 -0.002819026  0.566389678
 [6] -0.379654760  0.595551404 -1.278362654 -0.637809101 -0.056317505
[11] -1.247872021  0.689254507

> ifelse(x > 0, 1, 0)

 [1] 1 0 0 0 1 0 1 0 0 0 0 1
```

Functions

```
> ifelse_new <- function(x,break_point,
+                         true_arg,false_arg){
+   y <- x
+   for(i in 1:length(y)) {
+     if(y[i] > break_point){
+       y[i] <- 1
+     } else {
+       y[i] <- 0
+     }
+   }
+   return(y)
+ }
> x <- rnorm(12)
> ifelse(x > 0, 1, 0)
```

```
 [1] 0 1 0 1 1 1 0 1 0 1 1 0
```

```
> ifelse_new(x,0,1,0)
```

```
 [1] 0 1 0 1 1 1 0 1 0 1 1 0
```

The apply family

- "apply" ing functions to each element of a vector/data frame/list/array
  - apply, lapply, sapply, tapply
  - apply: only used for arrays/matrices
  - lapply(): takes any structure and gives a list of results
  - tapply(): allows you to create tables of values from subgroups defined by one or more factors

```
> apply(iris[,-5],2,mean)

Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

The apply family

- "apply" ing functions to each element of a vector/data frame/list/array
  - lapply(): takes any structure and gives a list of results

```
> L1 <- list(sample(1:10,50,replace = TRUE),
+            rep(c(1,2),30),
+            rep(c("training","test"),each=20))
> lapply(L1,table)
[[1]]

 1  2  3  4  5  6  7  8  9 10
 8  4  5  6  6  3  3  7  6  2

[[2]]

 1  2
30 30

[[3]]

    test training
      20       20
```

Writing Efficient R codes

- Bad code

```
> bad_code <- function(n){
+   x <- NULL
+   for(i in 1:n){
+     x[i] <- sqrt(i)
+   }
+   return(x)
+ }
```

- Better code

```
> good_code <- function(n){
+   x <- sqrt(1:n)
+   return(x)
+ }
```

- the runtime analysis

```
> if(!"microbenchmark" %in% rownames(installed.packages()))
+   install.packages("microbenchmark")
> library(microbenchmark)
> n <- 100000
> analysis <- microbenchmark(times= 100,good_code(n),bad_code(n))
```

Writing Efficient R codes: Some examples

- Bad code: Standardizing the vector

```
> bad_code <- function(n,lower_bound,upper_bound){
+   x <- runif(n,lower_bound,upper_bound)
+   maxx <- max(x)
+   minx <- min(x)
+   for(i in 1:n){
+     x[i] <- (x[i] - minx)/(maxx-minx)
+   }
+   return(x)
+ }
```

  - R has to call "-" and "/" several times
  - R has to call "[]" several times

Writing Efficient R codes: Some examples

- Better code

```
> good_code <- function(n,lower_bound,upper_bound){
+   x <- runif(n,lower_bound,upper_bound)
+   x <- (x - min(x))/(max(x)-min(x))
+   return(x)
+ }
```

  - "-","/" are only called once
  - "[]" has not been called.
- The runtime analysis

```
> n <- 100000
> analysis <- microbenchmark(times= 100,
+                            good_code(n,0,3),
+                            bad_code(n,0,3))
```

Writing Efficient R codes: "apply" family

- Lets simulate a list with arbitrarily different sizes of sampes

```
> x <- list()
> for(i in 1:100){
+   rand_x <- sample(1:100,1)
+   x[[i]] <- sample(1:100,rand_x,replace = T)
+ }
```

- Create a table for each element, put that in a list

```
> bad_code <- function(x){
+   len_x <- length(x)
+   table_x <- list()
+   for(i in 1:len_x){
+     table_x[[i]] <- table(x[[i]])
+   }
+   return(table_x)
+ }
```

Writing Efficient R codes: "apply" family

- good code, using lapply

```
> good_code <- function(x){
+   return(lapply(x,table))
+ }
```

- The runtime analysis

```
> analysis <- microbenchmark(times= 100,
+                             good_code(x),bad_code(x))
```

- It is not always the case that a code is faster than the other
- But it is definitely cleaner !!!

Writing Efficient R codes: "apply" family

- It is not always the case that an "apply" function is faster
- Always use an alternative, if requires less functions
- Method 1

```
> x = 1:100
> cs_for = function(x){
+   for(i in x){
+     if(i == 1){
+       xc = x[i]
+     } else {
+       xc = c(xc, sum(x[1:i]))
+     }
+   }
+   xc
+ }
```

Writing Efficient R codes: "apply" family

- Method 2

```
> cs_apply = function(x){
+   sapply(x, function(x) sum(1:x))
+ }
```

- Method 3

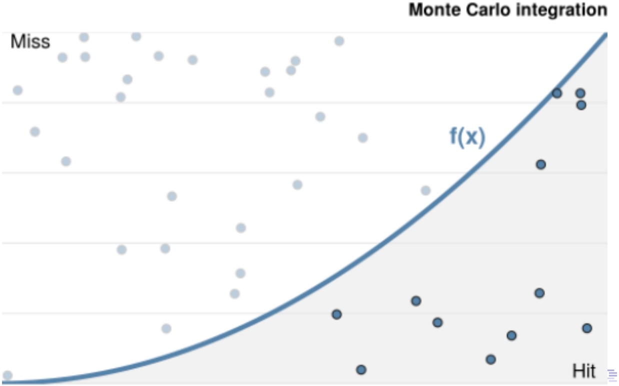
```
> analysis <- microbenchmark(cs_for(x),
+                            cs_apply(x),
+                            cumsum(x))
```

Writing Efficient R codes: Monte Carlo Example

- Monte Carlo Integration: Finding the Area with a collection of randomly drawn points

$$\int_0^1 x^2 dx \tag{1}$$

- Sampling random points inside a box and check if it is under the curve



Writing Efficient R codes: Monte Carlo Example

- bad code

```
> monte_carlo = function(n) {
+   hits = 0
+   for (i in 1:n) {
+     u1 = runif(1)
+     u2 = runif(1)
+     if (u1 ^ 2 > u2)
+       hits = hits + 1
+   }
+   return(hits / n)
+ }
```

- good code

```
> monte_carlo_vec = function(n) sum(runif(n)^2 > runif(n))/n
```

- The run time analysis

```
> n <- 1000
> analysis <- microbenchmark(monte_carlo(n), monte_carlo_vec(n))
```