

R language and data analysis: apply family

Qiang Shen

Oct.8,2016

vectorization

- ▶ R user vs. R programmer/developer
- ▶ Vectorization is the more limited process of converting a computer program from a scalar implementation, which processes a single pair of operands at a time, to a vector implementation which processes one operation on multiple pairs of operands at once.
- ▶ Vectorization is a particular form of how parallelism is achieved.

vectorization

- ▶ apply family
- ▶ plyr/reshape
- ▶ dplyr/data.table

Looping on the Command Line

- ▶ `apply`: Apply a function over the margins of an array
- ▶ `lapply`: Loop over a list and evaluate a function on each element
- ▶ `sapply`: Same as `lapply` but try to simplify the result
- ▶ `tapply`: Apply a function over subsets of a vector
- ▶ `mapply`: Multivariate version of `lapply`

`split` is also useful, especially in conjunction with `lapply`.

apply

- ▶ what does apply mean.
- ▶ apply to different dimension.
- ▶ compare with rowMeans etc.
- ▶ different functions.

apply

apply is used to evaluate a function (often an anonymous one) over the **margins** of an array.

- ▶ used to apply a function to the rows or columns of a matrix or an array.
- ▶ used with general arrays, e.g. taking the average, standard deviation.
- ▶ not really faster than writing a loop, but make it simpler.

apply: margin

Bivariate Probability Distribution

Example - Two discrete rv's X and Y

Bivariate pdf					
		Y			
		%	0	1	Pr(X)
X	0	1/8	0	1/8	
	1	2/8	1/8	3/8	
	2	1/8	2/8	3/8	
	3	0	1/8	1/8	
Pr(Y)		4/8	4/8	1	

Figure 1:

apply

```
str(apply)
```

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

- ▶ X is an array
- ▶ MARGIN is an integer vector indicating which margins should be “retained”.
- ▶ FUN is a function to be applied
- ▶ ... is for **other arguments** to be passed to FUN

apply a function to the rows or columns of a matrix.

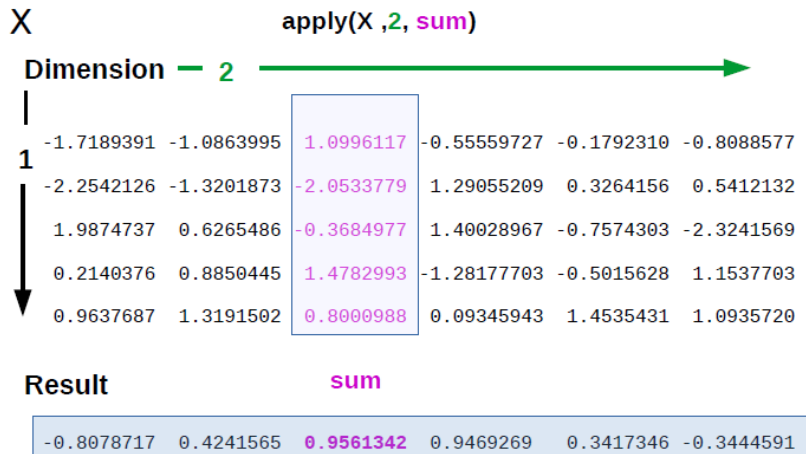


Figure 2: r

apply vs. standard method

```
y<-matrix(rnorm(6),2,3)
y
cbind(mean(y[1,]),mean(y[2,]))
apply(y,1,mean)
```

```
x <- matrix(rnorm(30), 5, 6)
sumx<-NULL
for (i in 1:6){
  temp<-sum(x[,i])
  sumx[i]<-temp
}
sumx
apply(x,2,sum)
```

col/row sums and means

- ▶ For sums and means of matrix dimensions, we have some shortcuts.
- ▶ `rowSums(x)`
- ▶ `rowMeans(x)`
- ▶ `colSums(x)`
- ▶ `colMeans(x)`

apply with ...

```
x <- matrix(rnorm(200), 20, 10)
dim(x)
apply(x, 2, quantile)
apply(x, 2, quantile, probs = c(0.25, 0.75))
```

lapply :start from an example

lapply Loop over a list and evaluate a function on each element

```
##user-defined function.
```

```
func<-function(x){
```

```
  if (x%%2 == 0) {
```

```
    ret<-'even'
```

```
  }else{
```

```
    ret<-'odd'}
```

```
  return(ret)
```

```
}
```

```
func(101)
```

```
vec<-round(runif(4)*100)
```

```
vec
```

```
vec;func(vec)
```

```
lapply(vec,func)
```

vectorization.

```
func<-Vectorize(func)
func(vec)
# ifelse(vec%%2, 'even', 'odd')
```

lapply

lapply takes three arguments:

```
str(lapply)
```

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

lapply: beyond apply.

lapply always returns a list, regardless of the input.

```
lapply(iris[,1:4],mean)
```


lapply

```
x <- list(a = 1:5, b = rnorm(10))  
x;lapply(x, mean)
```

lapply ...

```
x <- 1:4  
lapply(x, runif, min = 0, max = 10)
```

```
## [[1]]  
## [1] 4.5  
##  
## [[2]]  
## [1] 4.7 6.1  
##  
## [[3]]  
## [1] 0.41 6.56 8.46  
##  
## [[4]]  
## [1] 0.45 6.77 3.68 8.38
```

lapply

make use of *anonymous* functions for lapply. example: An anonymous function for extracting the 1st row of each matrix.

```
data <- list(a = matrix(1:6, 2, 3), b = matrix(1:6, 3, 2),  
data  
lapply(data, function(x) x[1,])
```

lapply

anonymous functions continued.

```
lapply(iris[,1:4],function(x) sd(x,na.rm=T)/mean(x,na.rm=T))
myfunc<-function(x){
  rec<-c(mean(x,na.rm=T),sd(x,na.rm=T))
  return(rec)
}
result<-lapply(iris[,1:4],myfunc)
result
```

methods to covert list into data.frame.

```
t(as.data.frame(result))  
# t(sapply(result, '['))  
do.call('rbind',result)
```

sapply

sapply will try to simplify the result of lapply if possible.

- ▶ If the result is a list where every element is length 1, then a vector is returned
- ▶ If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- ▶ If it can't figure things out, a list is returned

taply: split-apply-combine.

- ▶ Split up a big dataset
- ▶ Apply a function to each piece
- ▶ Combine all the pieces back together
- ▶ map-reduce in hadoop.

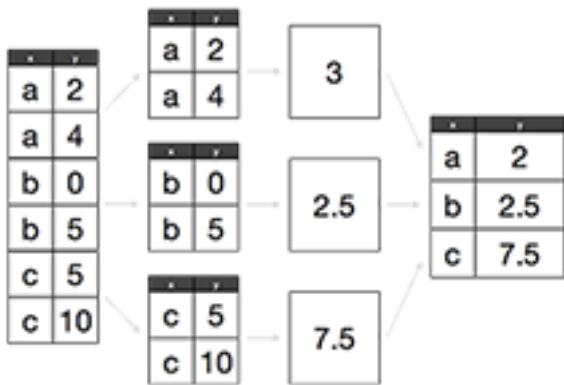


Figure 3:

tapply




tapply is used to apply a function over subsets of a vector. - apply, sapply/lapply vs. tapply

```
str(tapply)
```

```
## function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```


tapply

Take group means.

split			Apply	combine
x	f			
[1,]	-0.56047565	1		0.1935703
[2,]	-0.23017749	1		
[3,]	1.55870831	1		
[4,]	0.07050839	1		
[5,]	0.12928774	1		
[6,]	0.95683335	2		0.5526592
[7,]	0.45333416	2		
[8,]	0.67757064	2		
[9,]	0.57263340	2		
[10,]	0.10292468	2		
[11,]	2.28055488	3		1.8551235
[12,]	-0.72727063	3		
[13,]	2.69018435	3		
[14,]	1.50381245	3		
[15,]	3.52833655	3		

tapply

```
set.seed(123)
x <- c(rnorm(5), runif(5), rnorm(5, 1))
f <- gl(3, 5)
data<-cbind(x,f)
tapply(x, f, mean)
```

```
##      1      2      3
## 0.19 0.55 1.86
```

```
tapply(x, f, mean, simplify = FALSE)
```

```
## $`1`
## [1] 0.19
##
## $`2`
## [1] 0.55
##
## $`3`
```

tapply

Find group ranges.

```
tapply(x, f, range)
```

```
## $`1`
```

```
## [1] -0.56  1.56
```

```
##
```

```
## $`2`
```

```
## [1] 0.10 0.96
```

```
##
```

```
## $`3`
```

```
## [1] -0.73  3.53
```

tapply example

- ▶ iris flower dataset: Stata, python, R
- ▶ https://en.wikipedia.org/wiki/Iris_flower_data_set



Figure 4: iris

tapply example

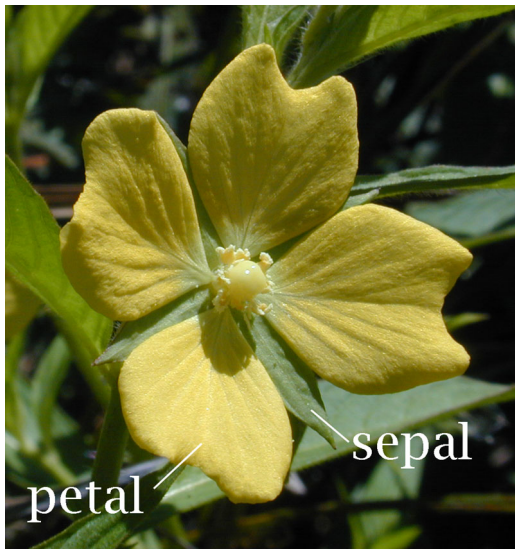


Figure 5: iris

tapply example

- ▶ vs. pivot table in excel

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
## $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4
## $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9
## $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1
## $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0
## $ Species      : Factor w/ 3 levels "setosa","versicolor"
```

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1          3.5          1.4          0.2   setosa
## 2          4.9          3.0          1.4          0.2   setosa
## 3          4.7          3.2          1.3          0.2   setosa
## 4          4.6          3.1          1.5          0.2   setosa
## 5          5.0          3.6          1.4          0.2   setosa
```

tapply example

- ▶ vs. pivot table in excel

```
tapply(iris[,1], iris$Species, mean)
```

```
##      setosa versicolor  virginica  
##      5.0      5.9      6.6
```

```
t(sapply(iris[,1:4],function(x) tapply(x, iris$Species, mean)))
```

```
##      setosa versicolor  virginica  
## Sepal.Length  5.01      5.9      6.6  
## Sepal.Width   3.43      2.8      3.0  
## Petal.Length  1.46      4.3      5.6  
## Petal.Width   0.25      1.3      2.0
```

```
# myfun<-function(x) {  
#   tapply(x, iris$Species, mean,na.rm=T)  
# }  
# sapply(iris[,1:3],myfun)
```

split

`split` takes a vector or other objects and splits it into groups determined by a factor or list of factors.

```
str(split)
```

```
## function (x, f, drop = FALSE, ...)
```

```
# function (x, f, drop = FALSE, ...)
```

- ▶ `x` is a vector (or list) or data frame
- ▶ `f` is a factor (or coerced to one) or a list of factors
- ▶ `drop` indicates whether empty factors levels should be dropped

split

```
x <- c(rnorm(10), runif(10), rnorm(10, 1))  
f <- gl(3, 10)  
split(x, f)
```

```
## $`1`
```

```
## [1] 0.549 0.238 -1.049 1.295 0.826 -0.056 -0.784 -0.117 0.005 0.005
```

```
##
```

```
## $`2`
```

```
## [1] 0.139 0.233 0.466 0.266 0.858 0.046 0.442 0.799 0.117 0.005
```

```
##
```

```
## $`3`
```

```
## [1] 0.18 1.68 0.68 -0.31 0.40 0.87 1.89 0.85 1.00 0.005
```

split

A common idiom is `split` followed by an `lapply`.

```
lapply(split(x, f), mean)
```

```
## $`1`  
## [1] -0.027  
##  
## $`2`  
## [1] 0.39  
##  
## $`3`  
## [1] 0.53
```

Splitting on More than One Level

```
x <- rnorm(10)
f1 <- gl(2, 5)
f2 <- gl(5, 2)
f1;f2
```

```
## [1] 1 1 1 1 1 2 2 2 2 2
```

```
## Levels: 1 2
```

```
## [1] 1 1 2 2 3 3 4 4 5 5
```

```
## Levels: 1 2 3 4 5
```

```
interaction(f1, f2)
```

```
## [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
```

```
## Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 2.4 1.5 2.5
```

```
data<-cbind(x,f1,f2)
```

```
data
```

split:Empty levels can be dropped.

```
str(split(x, list(f1, f2), drop = TRUE))
```

```
## List of 6
## $ 1.1: num [1:2] -0.772 0.287
## $ 1.2: num [1:2] -1.221 0.435
## $ 1.3: num 0.8
## $ 2.3: num -0.164
## $ 2.4: num [1:2] 1.243 -0.934
## $ 2.5: num [1:2] 0.394 0.404
```

Splitting a Data Frame

```
s <- split(iris, iris$Species)
# str(s)
sapply(s, function(x) colMeans(x[, 1:4], na.rm=T))
```

```
##                setosa versicolor virginica
## Sepal.Length    5.01          5.9         6.6
## Sepal.Width     3.43          2.8         3.0
## Petal.Length     1.46          4.3         5.6
## Petal.Width      0.25          1.3         2.0
```

```
# sapply(s, function(x) sapply((x[, 1:4]), mean, na.rm=T))
```

mapply

lapply and sapply only iterate over a single R object.

```
str(lapply)
```

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

```
str(mapply)
```

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

mapply

lapply and sapply only iterate over a single R object.

```
## list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))  
mapply(rep, 1:4, 4:1)
```

```
## [[1]]  
## [1] 1 1 1 1  
##  
## [[2]]  
## [1] 2 2 2  
##  
## [[3]]  
## [1] 3 3  
##  
## [[4]]  
## [1] 4
```

mapply

lapply and sapply only iterate over a single R object.

```
## list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))  
mapply(rep, 1:4, 4:1)
```

```
## [[1]]  
## [1] 1 1 1 1  
##  
## [[2]]  
## [1] 2 2 2  
##  
## [[3]]  
## [1] 3 3  
##  
## [[4]]  
## [1] 4
```


mapply

```
noise <- function(n, mean, sd) {  
  rnorm(n, mean, sd)  
}  
noise(5,1,2)  
noise(100, 1:20, 5:1)  
##revisit this when we talk about Sprintf function later.  
simulation<-mapply(noise,100000,1:20,5:1)  
dim(simulation)  
apply(simulation,2,mean)  
apply(simulation,2,sd)
```

mapply

```
##1. generate dataset called data
data<-data.frame(a=-c(1:5),b=-c(2:6),d=-c(3:7),
                 e=-c('1','2','a','x','y'),
                 f=-c('2','3','5','d','c'),
                 g=-c('3','k','5','6',NA),
                 stringsAsFactors = F)
data[,1:3]<-sapply(data[,1:3],as.character)
names(data)<-letters[1:6]
##2. do mapply
myfunc<-function(x, y) {
  ifelse(grepl('[a-z]',y), y, x)
}
data2<-data
data2[1:3] = mapply(myfunc, data[1:3], data[4:6])
data2
```

Looping on the Command Line

- ▶ `lapply`: Loop over a list and evaluate a function on each element
- ▶ `sapply`: Same as `lapply` but try to simplify the result
- ▶ `apply`: Apply a function over the margins of an array
- ▶ `tapply`: Apply a function over subsets of a vector
- ▶ `mapply`: Multivariate version of `lapply`