R language and data analysis: apply family

Qiang Shen

Sept.29,2016

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.

examples of vectorization

- vector extraction: V[1:10]
- vector assignment: V[1:10] <- seq(1,10)
- apply:sapply(V, mean)
- vector/matrix: A +/- B; A %*% B

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

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

apply

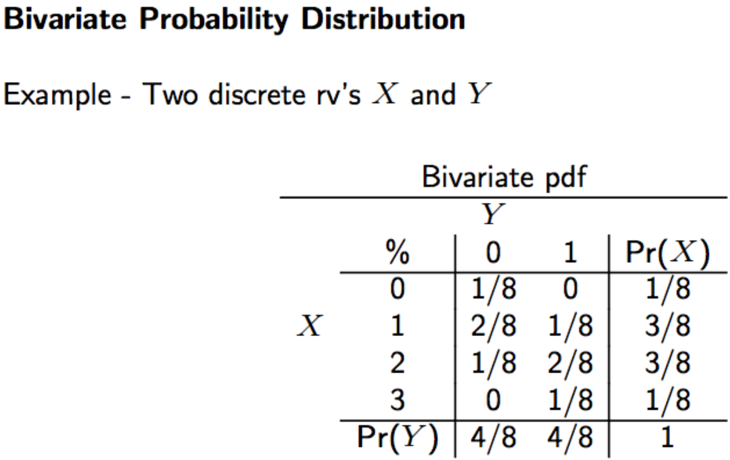
- what does apply mean.
- apply to different dimension.
- compare with rowMeans etc.
- different functions.

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)
	0	1/8	0	1/8
X	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:

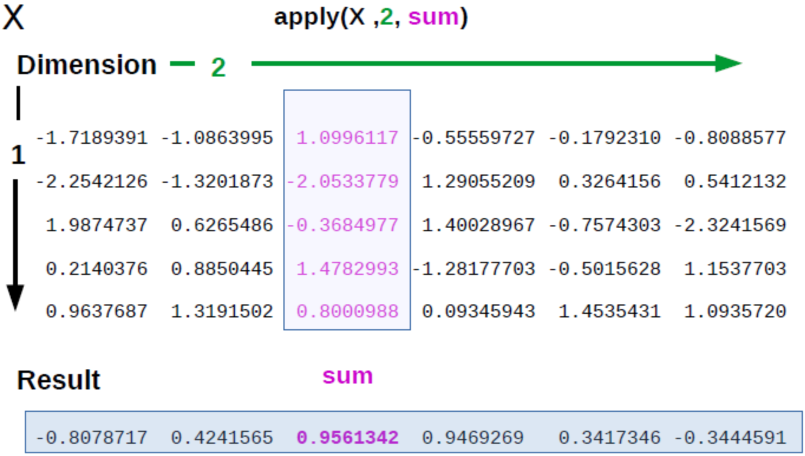
apply

```
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.



```
-1.7189391 -1.0863995  1.0996117 -0.55559727 -0.1792310 -0.8088577
-2.2542126 -1.3201873 -2.0533779  1.29055209  0.3264156  0.5412132
 1.9874737  0.6265486 -0.3684977  1.40028967 -0.7574303 -2.3241569
 0.2140376  0.8850445  1.4782993 -1.28177703 -0.5015628  1.1537703
 0.9637687  1.3191502  0.8000988  0.09345943  1.4535431  1.0935720
```

```
-0.8078717  0.4241565  0.9561342  0.9469269   0.3417346 -0.3444591
```

Figure 2: r

apply vs. standard method

```
y<-matrix(rnorm(6),2,3)
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 ...

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

lapply :start from an example

- a simple function

```
##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;func(vec)
lapply(vec,func)
```

vectorization.

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

lapply

`lapply` takes three arguments:

```r
str(lapply)
```

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

lapply: beyond apply.

`lapply` always returns a list, regardless of the input.

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

lapply

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

lapply . . .

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

```
## [[1]]
## [1] 7.945498
##
## [[2]]
## [1] 8.216793 2.590300
##
## [[3]]
## [1] 0.001757133 1.413812574 1.104137434
##
## [[4]]
## [1] 1.566145 2.911776 2.376669 8.381901
```

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.

```r
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

tapply: split-apply-combine.

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

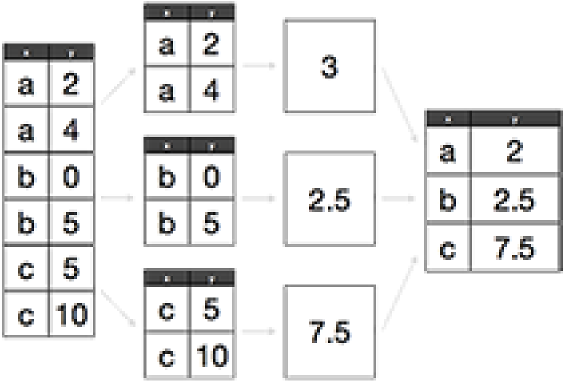


Figure 3:

tapply

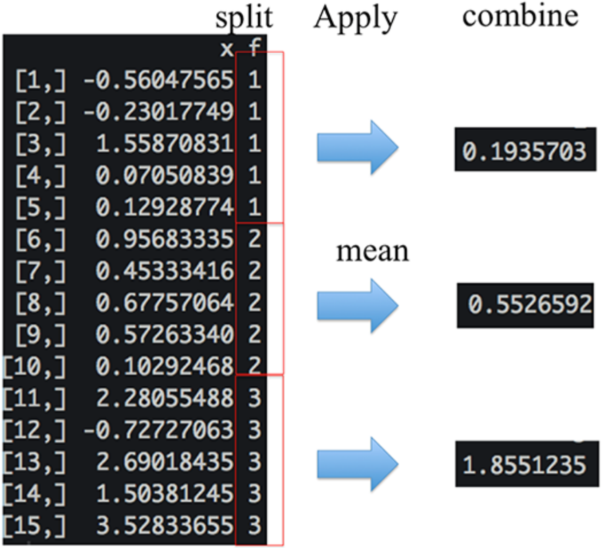
tapply is used to apply a function over subsets of a vector.

```r
str(tapply)
```

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

tapply

Take group means.



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

```
0.1935703
```

```
0.5526592
```

```
1.8551235
```

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
## 0.1935703 0.5526592 1.8551235
```

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

```
## $`1`
## [1] 0.1935703
##
## $`2`
## [1] 0.5526592
##
## $`3`
```

tapply

Find group ranges.

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

```
## $`1`
## [1] -0.5604756  1.5587083
##
## $`2`
## [1] 0.1029247 0.9568333
##
## $`3`
## [1] -0.7272706  3.5283366
```

tapply example

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

```
##     setosa versicolor  virginica
##      5.006      5.936      6.588
```

split

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

```r
str(split)
```

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

```r
# 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

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

```
## $`1`
##  [1]  0.54909674  0.23821292 -1.04889314  1.29476325  0.
##  [6] -0.05568601 -0.78438222 -0.73350322 -0.21586539 -0.
##
## $`2`
##  [1] 0.13880606 0.23303410 0.46596245 0.26597264 0.8578
##  [7] 0.44220007 0.79892485 0.12189926 0.56094798
##
## $`3`
##  [1]  0.1814843  1.6849361  0.6799436 -0.3115224  0.400
##  [7]  1.8867361  0.8486040  1.3297912 -2.2273228
```

split

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

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

```
## $`1`
## [1] -0.026563
##
## $`2`
## [1] 0.3931406
##
## $`3`
## [1] 0.5343631
```

Splitting a Data Frame

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

```
##              setosa versicolor virginica
## Sepal.Length  5.006      5.936     6.588
## Sepal.Width   3.428      2.770     2.974
## Petal.Length  1.462      4.260     5.552
## Petal.Width   0.246      1.326     2.026
```

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

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
```

```
##                    f1 f2
```

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
```

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