Programming for Data Science
Apply family in R language

Marco Beccuti

Università degli Studi di Torino

Dipartimento di Informatica

Apply family in R

- How to efficiently apply a function to each element of array, data frame and list.

  For instance: to apply a function to the rows/columns of a matrix

- Functions apply, lapply, sapply, tapply can be used:

  apply : only used for arrays/matrices;

  lapply : takes any data structure and gives a list of results;

  sapply : like lapply, but it tries to simplify the result to a vector or matrix if possible;

  tapply :allows us to apply a function on a subset of values grouped according to one or more factors.

Function apply()

- the apply function returns a vector or array of values obtained by applying a function to margins of an array or matrix.

apply(X, MARGIN, FUN, ...)

X : array;
MARGIN : 1 for rows, 2 for columns;
FUN : one function to be applied;
... : optional arguments to FUN;

$> m$

	[,1]	[,2]
[1,]	−0.1767643	−0.1950407
[2,]	1.5306045	0.3307676
[3,]	−0.3806768	0.8992097

$> apply(m, 1, sum)$ **by rows**
[1] − 0.3718050 1.8613721 0.5185329

$> apply(m, 2, sum)$ **by columns**
[1]0.9731634 1.0349366

Function apply()

- the apply function returns a vector or array of values obtained by applying a function to margins of an array or matrix.

apply(X, MARGIN, FUN, ...)

X	:	array;
MARGIN	:	1 for rows, 2 for columns;
FUN	:	one function to be applied;
...	:	optional arguments to FUN;

$> m$

	[, 1]	[, 2]
[1,]	−0.1767643	−0.1950407
[2,]	1.5306045	0.3307676
[3,]	−0.3806768	0.8992097

$> apply(m, 1, max)$ **by rows**

[1] − 0.1767643 1.5306045 0.8992097

$> apply(m, 2, min)$ **by columns**

[1] − 0.3806768 − 0.1950407

Function apply()

- the apply function returns a vector or array of values obtained by applying a function to margins of an array or matrix.

apply(X, MARGIN, FUN, ...)

X : array;
MARGIN : 1 for rows, 2 for columns;
FUN : one function to be applied;
... : optional arguments to FUN;

$> m = matrix(rnorm(6), nrow = 2)$

	[, 1]	[, 2]
[1,]	0.34963183	1.0360705
[2,]	−1.04686386	0.6846824
[3,]	−0.06385193	0.6219289

$> apply(m, 1, quantile, prob = c(0.25, 0.75))$ **by columns**

	[, 1]	[, 2]	[, 3]
25%	0.5212415	− 0.6139773	0.1075933
75%	0.8644608	0.2517958	0.4504837

Function lapply()

- the lapply function returns a list where each element is the result of applying a function to the corresponding element of input data structure.

  lapply(X, FUN, ...)

  X : any data that can be compatible with a list;
  FUN : one function to be applied;
  ... : optional arguments to FUN;

> *data*() **to list in-built data set**
> *lapply*(*trees*, *mean*) **trees is a in-built data set**
$*Girth*
[1]13.24839
$*Height*
[1]76
$*Volume*
[1]30.17097

```
> trees
   Girth Height Volume
1    8.3     70   10.3
2    8.6     65   10.3
3    8.8     63   10.2
4   10.5     72   16.4
5   10.7     81   18.8
6   10.8     83   19.7
7   11.0     66   15.6
8   11.0     75   18.2
9   11.1     80   22.6
```

Function sapply()

- the sapply function is a user-friendly version and wrapper of "lapply" by default returning a vector, matrix .

sapply(X, FUN, ...)

X : any data that can be compatible with a list/vector/matrix;
FUN : one function to be applied;
... : optional arguments to FUN;

> *data*() **to list in-built data set**
> *sapply*(*trees*, *mean*) **trees is a in-built data set**

Girth	***Height***	***Volume***
13.24839	76.00000	30.17097

```
> trees
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
4  10.5     72   16.4
5  10.7     81   18.8
6  10.8     83   19.7
7  11.0     66   15.6
8  11.0     75   18.2
9  11.1     80   22.6
```

Function tapply()

- the tapply function allows us to apply a function on a subset of values grouped according to one or more factors .

tapply(X, INDEX, FUN, ...)

X : any data that can be compatible with a list;
INDEX : list of one or more factors used to cluster X;
FUN : one function to be applied;
... : optional arguments to FUN;

> *library(MASS)* **to load MASS data set**
> *Cars93* **Car93 is a MASS data set**

	Manufacturer	Model	Type	Min.Price	Price	Max.Price	MPG.city
1	Acura	Integra	Small	12.9	15.9	18.8	25
2	Acura	Legend	Midsize	29.2	33.9	38.7	18
3	Audi	90	Compact	25.9	29.1	32.3	20
4	Audi	100	Midsize	30.8	37.7	44.6	19
5	BMW	535i	Midsize	23.7	30.0	36.2	22
6	Buick	Century	Midsize	14.2	15.7	17.3	22
7	Buick	LeSabre	Large	19.9	20.8	21.7	19
8	Buick	Roadmaster	Large	22.6	23.7	24.9	16
9	Buick	Riviera	Midsize	26.3	26.3	26.3	19

> *tapply(Cars93$Price, Cars93$Manufacturer, mean)* **Compute the average price for each brand**

How to apply a function in parallel

- In R different possibilities exist (e.g. parlapply, mclapply, clusterApply, . . .), but not all are portable (i.e.they can not be used indifferently on Window, Linux and macOS);

- **"snow"** package provides parallelization functionality on Window, Linux and macOs;

  > *install.packages*("*snow*")

- It supports: Socket and Message Passing Interface (MPI) protocols;

  Socket
  - Portable, but low level protocol;
  - Can be used interactively;
  - Good for running on a multicore machine;

  MPI
  - Needs the **"Rmpi"** package;
  - Cannot be used interactively;
  - Good for running on several nodes;
  - Works everywhere where **Rmpi** is installed.

How to apply a function in parallel

BASE R	SNOW
lapply	parLapply
sapply	parSapply
apply(rowwise)	parRapply, parApply(,1)
apply(columnwise)	parCapply, parApply(,2)

- To use these function you have to initialize the cluster with command makeCluster()
- When computation is terminated you have to close the cluster with command stopCluster()

How to apply a function in parallel

- An example using 6 cores and Sys.sleep()

  $> z = list(1, 1, 1, 1, 1, 1)$

  $> cl < -makeCluster(6)$

  $> system.time(parSapply(cl, z, Sys.sleep))$ **it is ~ 6 times faster than sapply**

  $> stopCluster(cl)$

How to apply a function in parallel

- An example using 8 cores and an own function

  $> z = rexp(6000000, 6)$

  $> z = list(z, z, z, z, z, z, z, z)$

  $> cl < -makeCluster(8)$

  $> system.time(parSapply(cl, z, function(val)\{2\hat{}val + 2\hat{}val + 2\hat{}val\}))$
  it is ~ 2 times faster than sapply

  $> stopCluster(cl)$

How to apply a function in parallel

- An example using 8 cores and an own function

  $> z = rexp(6000000, 6)$

  $> z = list(z, z, z, z, z, z, z, z)$

  $> cl <- makeCluster(8)$

  $> system.time(parSapply(cl, z, function(val)\{val + val\}))$
  it is ~ 3 times slower than sapply

  $> stopCluster(cl)$

Parallelization Efficiency

- The time spent in each invocation of the worker function should not be too short;
- If the time spent in each invocation of the worker function vary very much, try the load balancing versions of the functions (e.g. clusterApplyLB);
- Avoid copying large things back and forth:
  - Export large datasets up front with clusterExport();
  - Write the worker function to return as little as possible.

Parallelization Efficiency

- In general, it is recommend using forking instead of sockets if you are not on Windows;
- Forking with parSapply implemented in library parallel

  $> library(parallel)$

  $> z = rexp(6000000, 6)$

  $> z = list(z, z, z, z, z, z, z, z)$

  $> cl < -makeForkCluster(nnodes = 8)$

  $> system.time(parSapply(cl, z, function(val)\{2\hat{}val + 2\hat{}val + 2\hat{}val\}))$
  it is ~ 3 times faster than sapply

  $> stopCluster(cl)$

Function do.call()

- it constructs and executes a function call from a name or a function and a list of arguments to be passed to it..

do.call(what, args)

what : character string naming the function to be called;

args : a list of arguments to the function call. The names attribute of args gives the argument names;

$> a = c(2, 2, 2, 2)$

$> f = c("mean", "sum")$

$> do.call(f[1], list(a))$

[1] 2

$> do.call(f[1], list(a))$

[1] 8

Exercises on apply

- Compute sums of the columns of the hills data set (in library MASS);
- Compute row and column sums of a matrix 10x10 whose values are generated according to uniform distribution between 4 and 10;
- Use apply to calculate the standard deviation of the columns of a matrix;
- Create a list of vectors of varying length (using sample() function);
- Consider in-built data set "airquality" compute the average wind speed and ozone percentage with respect to "month" column.

Exercises on apply

- Compute sums of the columns of the hills data set(in library MASS);

  $> lapply(hills, sum)$

  $> sapply(hills, sum)$

Exercises on apply

- Compute row and column sums of a matrix 10x10 whose values are generated according to uniform distribution between 4 and 10

  $> m = matrix(runif(100, min = 4, max = 10), ncol = 10)$
  $> apply(m, 1, sum)$
  $> apply(m, 2, sum)$

Exercises on apply

- Use apply to calculate the standard deviation of the columns of a matrix.

  $> m = matrix(runif(100, min = 4, max = 10), ncol = 10)$
  $> apply(m, 2, sd)$

Exercises on apply

- Create a list of vectors of varying length (using sample() function)

  $> veclen = sample(11:40)$
  $> mylist = lapply(veclen, runif)$

  or

  $> mylist = lapply(sample(11:40, 10), runif)$

Exercises on apply

- Consider in-built data set "airquality" compute the average wind speed and ozone percentage with respect to "month" column.

  $> tapply(airquality\$Wind, airquality\$Month, mean)$

  $> tapply(airquality\$Ozone, airquality\$Month, mean, na.rm = TRUE)$
  na.rm=TRUE removes NA from mean computation

Exercises on apply

- Compute a hundred times the mean of 1000000 observations distributed as $\mathcal{N}(0, 1)$ using lapply and parLapply. Compare the execution times using rbenchmark package.

Exercises on apply

- Compute a hundred times the mean of 1000000 observations distributed as $\mathcal{N}(0,1)$ using lapply and parLapply. Compare the execution times using rbenchmark package.

  $> install.packages("rbenchmark")$

  $> library(rbenchmark)$

  $> cl <- makeCluster(4)$

  $> benchmark($
  $parLapply(cl, 1:100, function(x)\{mean(rnorm(1000000))\}),$
  $lapply(1:100, function(x)\{mean(rnorm(1000000))\}), replications = 5)$

  $> stopCluster(cl)$

```
                                                               test replications
2        lapply(1:100, function(x) {\n    mean(rnorm(1e+06))\n})            5
1 parLapply(cl, 1:100, function(x) {\n    mean(rnorm(1e+06))\n})            5
  elapsed relative user.self sys.self user.child sys.child
2  32.650    2.933    32.607    0.000          0         0
1  11.132    1.000     0.013    0.009          0         0
```