Basic Course on **R**: The apply family of functions

David Nieuwenhuijse

May 18th - 24th, 2017

Contents

1 Using apply functions to run functions on data

Once you have written a function, you would like to apply it to some piece of data. As described in the previous chapter you can simply enter some values as arguments of the function and run it. However, usually you would like to run the function on all of your data. To do that you could write a for loop that loops through you data and applies the function to the whole dataset. However, there is a special family of functions in R that make it easier to apply your function to a range of different data classes in different ways. This family of functions are called apply functions.

The apply functions make it easier to run functions over vectors, matrixes, and data.frames. We will discuss four functions of the apply family that are regularly used apply(), lapply(), sapply() and tapply().

1.1 apply()

The apply function works by “applying” a specified function to an data object. It requires 3 arguments: the data, a so-called “MARGIN”, and a function. The data can be a vector, data.frame or a matrix. The MARGIN indicates whether you want to apply the function to the rows or the columns of your data, or both. To apply the function to the rows the MARGIN should be 1, to apply it to the columns it should be 2 and to apply it to both it should be c(1,2). The function can be an existing function, such as “sum” or “mean”, or your own custom function.

As an example we will apply the function “max” to some data, in this case a matrix.

First we create a matrix of 10 by 10.

```r
mat <- matrix(1:100,nrow=10)

mat
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1   11   21   31   41   51   61   71   81    91
##  [2,]    2   12   22   32   42   52   62   72   82    92
##  [3,]    3   13   23   33   43   53   63   73   83    93
##  [4,]    4   14   24   34   44   54   64   74   84    94
##  [5,]    5   15   25   35   45   55   65   75   85    95
##  [6,]    6   16   26   36   46   56   66   76   86    96
##  [7,]    7   17   27   37   47   57   67   77   87    97
##  [8,]    8   18   28   38   48   58   68   78   88    98
##  [9,]    9   19   29   39   49   59   69   79   89    99
## [10,]   10   20   30   40   50   60   70   80   90   100
```

Then we apply our function “max” to the matrix rows, indicated with a 1 (notice that we do not run the function by writing max(), but we just give the name of the function that should be run: max).

```r
apply(mat, 1, max)
```

```
##  [1]  91  92  93  94  95  96  97  98  99 100
```

The result of applying the function max to the rows of the matrix is a vector containing the maximal values for each row.

We can also determine the maximal value in each column by using 2 as the MARGIN value.

```r
apply(mat, 2, max)
```

```
##  [1]  10  20  30  40  50  60  70  80  90 100
```

As mentionned before, it is also possible to apply the functions to each element in the matrix by using c(1,2). In that case it doesn't make sense to determine the maximum value, so lets take the square root.

```
apply(mat, c(1,2), sqrt)
```

```
##           [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]
##  [1,] 1.000000 3.316625 4.582576 5.567764 6.403124 7.141428 7.810250
##  [2,] 1.414214 3.464102 4.690416 5.656854 6.480741 7.211103 7.874008
##  [3,] 1.732051 3.605551 4.795832 5.744563 6.557439 7.280110 7.937254
##  [4,] 2.000000 3.741657 4.898979 5.830952 6.633250 7.348469 8.000000
##  [5,] 2.236068 3.872983 5.000000 5.916080 6.708204 7.416198 8.062258
##  [6,] 2.449490 4.000000 5.099020 6.000000 6.782330 7.483315 8.124038
##  [7,] 2.645751 4.123106 5.196152 6.082763 6.855655 7.549834 8.185353
##  [8,] 2.828427 4.242641 5.291503 6.164414 6.928203 7.615773 8.246211
##  [9,] 3.000000 4.358899 5.385165 6.244998 7.000000 7.681146 8.306624
## [10,] 3.162278 4.472136 5.477226 6.324555 7.071068 7.745967 8.366600
##           [,8]     [,9]     [,10]
##  [1,] 8.426150 9.000000  9.539392
##  [2,] 8.485281 9.055385  9.591663
##  [3,] 8.544004 9.110434  9.643651
##  [4,] 8.602325 9.165151  9.695360
##  [5,] 8.660254 9.219544  9.746794
##  [6,] 8.717798 9.273618  9.797959
##  [7,] 8.774964 9.327379  9.848858
##  [8,] 8.831761 9.380832  9.899495
##  [9,] 8.888194 9.433981  9.949874
## [10,] 8.944272 9.486833 10.000000
```

Because sqrt also works on matrices, it is actually unnecessary to use apply to run it for each element in the matrix. In cases where functions cannot directly be run on a matrix, apply offers a short and readible alternative to writing a nested for loop.

1.2 lapply()

The lapply function is used to run a function on list objects. Let's assume we have a list of different sized matrices and we would like to know the dimensions of these matrices. We can then run the function "dim" on the list using lapply. lapply only requires a list object and a function as arguments and always returns a list of results.

```
mylist <- list(matrix(1:16,nrow=4), matrix(1:9,nrow=3),matrix(1:4,nrow=2))

lapply(mylist, dim)
```

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

Because dataframes are lists of lists, it is also possible to run lapply on dataframes. In that case lapply will apply the function to the columns of the data.frame object and it returns a list of values.

```
df <- data.frame("col1"=c(1,1,1,1), "col2"=c(2,2,2,2), "col3"=c(3,3,3,3))

lapply(df, sum)
```

```
## $col1
## [1] 4
##
## $col2
## [1] 8
##
## $col3
## [1] 12
```

1.3 sapply()

sapply is a user-friendly version of lapply. The difference with lapply is that sapply tries to turn the list of results into a more user-friendly format, such as a vector or a matrix.

For the first example the results are turned into a matrix.

```
sapply(mylist, dim)
```

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

For the second example, the results are turned into a vector.

```
sapply(df, sum)
```

```
## col1 col2 col3
##    4    8   12
```

There is no difference between lapply and sapply in how the data is used, but it gives you more flexibility in how the results are created.

1.4 tapply()

tapply lets you apply a function on groupings of your data. Imagine that you have a dataset in which a grouping factor separates your data into two groups of patients. With tapply you can apply a function to those two groups separately. The only thing tapply requires is the column you would like to apply the function to, the grouping factor and the function you would like to apply.

```
patients <- data.frame("group"=paste('grp',c(1,1,1,1,1,1,2,2,2,2,2,2),sep='-'), "outcome"=rnorm(12))
patients
```

```
##    group     outcome
## 1  grp-1  0.22115820
## 2  grp-1  1.03751935
## 3  grp-1 -0.06887431
## 4  grp-1  0.48706043
## 5  grp-1  1.10705296
## 6  grp-1  0.09463406
## 7  grp-2  0.28877781
## 8  grp-2  0.03450461
## 9  grp-2 -0.64676566
## 10 grp-2  1.02515888
## 11 grp-2 -0.28689227
## 12 grp-2  1.11382687
```

```
tapply(patients$outcome, patients$group, mean)
```

```
##     grp-1     grp-2
## 0.4797585 0.2547684
```

It is also possible to use multiple factors in a list to create groups, which returns a matrix.

```
patients <- data.frame("group"=paste('grp',c(1,1,1,1,1,1,2,2,2,2,2,2),sep='-'),
                       "serotype"=c("A","B","A","B","A","B","A","B","A","B","A","B"),
                       "outcome"=rnorm(12))

tapply(patients$outcome, list(patients$group, patients$serotype), mean)
```

```
##                A          B
## grp-1  0.5983194  0.5041594
## grp-2 -0.6070275 -0.8957907
```

These are some (trivial) examples of how you can use the apply family of functions to quickly apply a function to your data. It is possible to do the same thing by using for loops, but apply functions are generally faster to write and read. In some cases using apply to run your function can also increase the speed of your code. More on increasing the speed of your code will follow in later lectures.