Applying Vectorized Functions

Chapter 8

Stats 20 Lec 2

Fall 2017

Contents

Acknowledgements: Vivian Lew and Juana Sanchez

Learning Objectives

After studying this chapter, you should be able to:

- Summarize an R object with `str()` and `summary()`.
- Understand how to use the `apply` family of functions: `apply()`, `lapply()`, `sapply()`, `tapply()`.

1 Vectorized Summary Functions

Recall that a function in R is vectorized if applying the function to an object will automatically apply the function to individual components of the object. For (atomic) vectors, vector arithmetic is implements operations element-by-element.

```
c(1,2,3) + c(2,3,4)
```

```
## [1] 3 5 7
```

For more complex data structures (like matrices, data frames, and lists), we may be interested in applying a function to each row, column, or component. We will start with some generic vectorized functions that provide useful summaries for columns or components of R objects.

1.1 The `str()` Function

For a quick overview of any object in R, the **`str()`** function returns a compact display of the internal **structure** of the input object. As an example, we will apply this function to the `trees` data in the `datasets` package.

```
data(trees) # Load the trees data
str(trees) # Display the structure of the trees object
```

```
## 'data.frame':    31 obs. of  3 variables:
##  $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
##  $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
##  $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
```

The output of the `str(trees)` command shows that `trees` is a data frame with 31 observations and 3 variables. A brief summary of each component (column) in `trees` is given: Each component of `trees` is numeric (`num`), and the first few values from each component are printed.

The `str()` function is well suited for displaying the contents of nested lists (lists inside lists).

```
parks.df <- data.frame("Name"=c("Leslie","Ron","April"),"Height"=c(62,71,66),
                       "Weight"=c(115,201,66),"Income"=c(4000,NA,2000))
L <- list(1:10,matrix(1:6,nrow=2,ncol=3),parks.df,list(1:5,matrix(1:9,nrow=3,ncol=3)))
str(L)
```

```
## List of 4
##  $ : int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ : int [1:2, 1:3] 1 2 3 4 5 6
##  $ :'data.frame':    3 obs. of  4 variables:
##   ..$ Name  : Factor w/ 3 levels "April","Leslie",..: 2 3 1
##   ..$ Height: num [1:3] 62 71 66
##   ..$ Weight: num [1:3] 115 201 66
##   ..$ Income: num [1:3] 4000 NA 2000
##  $ :List of 2
```

```
##   ..$ : int [1:5] 1 2 3 4 5
##   ..$ : int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
```

1.2 The summary() Function

We previously used the **summary()** function to compute a few standard summary statistics on numeric vectors.

```r
summary(trees$Volume)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   10.20   19.40   24.20   30.17   37.30   77.00
```

The summary() function is an example of a **polymorphic** function in that it changes its output based on the type of input. Specifically, the output of summary() will depend on the class of the input object.

For data frames, the summary() function will compute summary statistics for each column in the data frame. If the column is a character or factor vector, the summary() output will adapt and return frequencies. For lists, the summary() function will return the length, class attribute, and mode of each component.

```r
summary(trees)
```

```
##      Girth           Height       Volume
##  Min.   : 8.30   Min.   :63   Min.   :10.20
##  1st Qu.:11.05   1st Qu.:72   1st Qu.:19.40
##  Median :12.90   Median :76   Median :24.20
##  Mean   :13.25   Mean   :76   Mean   :30.17
##  3rd Qu.:15.25   3rd Qu.:80   3rd Qu.:37.30
##  Max.   :20.60   Max.   :87   Max.   :77.00
```

```r
summary(parks.df)
```

```
##      Name       Height          Weight          Income
##  April :1   Min.   :62.00   Min.   : 66.0   Min.   :2000
##  Leslie:1   1st Qu.:64.00   1st Qu.: 90.5   1st Qu.:2500
##  Ron   :1   Median :66.00   Median :115.0   Median :3000
##             Mean   :66.33   Mean   :127.3   Mean   :3000
##             3rd Qu.:68.50   3rd Qu.:158.0   3rd Qu.:3500
##             Max.   :71.00   Max.   :201.0   Max.   :4000
##                                             NA's   :1
```

```r
summary(L)
```

```
##      Length Class      Mode
## [1,] 10     -none-     numeric
## [2,]  6     -none-     numeric
## [3,]  4     data.frame list
## [4,]  2     -none-     list
```

2 The apply Family of Functions

One of the most widely used features of R is the **apply** family of functions. The apply family consists of vectorized functions that minimize the need to use loops or repetitive code. We will cover the most common apply functions used in statistics, namely apply(), lapply(), sapply(), and tapply(). There are other functions in the same family (mapply(), rapply(), and vapply()), but these will not be covered.

2.1 The apply() Function

Suppose we want to compute the mean of each variable in the `trees` data frame. We could compute these individually, but it would require repetitive code (or a for loop).

```
mean(trees[,1]) # Or mean(trees$Girth)
```

```
## [1] 13.24839
```

```
mean(trees[,2]) # Or mean(trees$Height)
```

```
## [1] 76
```

```
mean(trees[,3]) # Or mean(trees$Volume)
```

```
## [1] 30.17097
```

For data sets with more than a few variables, using repetitive code is inefficient and cumbersome.

The **apply()** function is used to apply a function to the rows or columns (the **margins**) of matrices or data frames.

The syntax of `apply()` is `apply(X,MARGIN,FUN,...)`, where the arguments are:

- `X`: A matrix or data frame
- `MARGIN`: A vector giving the subscript(s) over which the function will be applied over. A `1` indicates rows, `2` indicates columns, and `c(1,2)` indicates rows and columns.
- `FUN`: The function to be applied.
- `...`: Any optional arguments to be passed to the `FUN` function (for example, `na.rm=TRUE`).

Using `apply()`, we can apply the `mean()` function to each column in `trees` simultaneously with a single command.

```
# Compute the mean of every column of the trees data frame
apply(trees,2,mean)
```

```
##    Girth   Height   Volume
## 13.24839 76.00000 30.17097
```

To compute the mean of each row, we can change the margin argument `MARGIN` from `2` (columns) to `1` (rows).

```
# Compute the mean of every row of the trees data frame
apply(trees,1,mean)
```

```
##  [1] 29.53333 27.96667 27.33333 32.96667 36.83333 37.83333 30.86667
##  [8] 34.73333 37.90000 35.36667 38.16667 36.13333 36.26667 34.00000
## [15] 35.36667 36.36667 43.90000 42.23333 36.80000 34.23333 42.16667
## [22] 41.96667 41.60000 42.10000 45.30000 51.23333 51.73333 52.06667
## [29] 49.83333 49.66667 61.53333
```

Side Note: Technically, the mean of every row or column in a matrix or data frame can also be computed using the `rowMeans()` and `colMeans()` functions, but `apply()` works more generally, since `apply()` allows us to apply any function, not just `mean()`.

Caution: Use caution when using `apply()` to a data frame. Ideally, the columns of the data frame should all be of the same type. The `apply()` function is intended for matrices (and arrays, which are higher dimensional versions of matrices). Using `apply()` on a data frame will first coerce the data frame into a matrix with `as.matrix()` before applying the function in the `FUN` argument.

Question: What does `apply(parks.df,2,mean)` output? Why does this command not give the results we intended? How can we find the mean of each of the numeric columns in `parks.df` using `apply()`?

Note: If the applied function in the `FUN` argument of `apply()` returns a single value, the output of the `apply()` function will be a vector. If the applied function returns a vector (with more than one element), then the output of `apply()` will be a matrix.

```
# Compute the range (min and max) of every column of the trees data frame
apply(trees,2,range)
```

```
##      Girth Height Volume
## [1,]   8.3     63   10.2
## [2,]  20.6     87   77.0
```

Question: How is `summary(trees)` different from `apply(trees,2,summary)`?

Note: The `FUN` argument does not have to be a built-in function. We can also write our own function and apply it to each row or column.

As an example, suppose we want to compute the squared deviations from the mean for each variable in the `trees` data frame.

```
squared.devs <- function(x){
  # This function inputs a vector and computes the squared deviations away from the mean.
  (x - mean(x))^2
}
# Apply the squared.devs() function to every column of the trees data frame
trees.sdevs <- apply(trees,2,squared.devs)
# Print the first few rows of the output
head(trees.sdevs)
```

```
##          Girth Height   Volume
## [1,] 24.486535     36 394.8554
## [2,] 21.607503    121 394.8554
## [3,] 19.788148    169 398.8396
## [4,]  7.553632     16 189.6396
## [5,]  6.494277     25 129.2989
## [6,]  5.994599     49 109.6412
```

The function can also be written directly into the `FUN` argument without having to save it as a separate object.

```
# Creates the same object as apply(trees,2,squared.devs)
trees.sdevs <- apply(trees,2,function(x){(x - mean(x))^2})
# Print the first few rows of the output
head(trees.sdevs)
```

```
##          Girth Height   Volume
## [1,] 24.486535     36 394.8554
## [2,] 21.607503    121 394.8554
## [3,] 19.788148    169 398.8396
## [4,]  7.553632     16 189.6396
## [5,]  6.494277     25 129.2989
## [6,]  5.994599     49 109.6412
```

2.2 The `lapply()` Function

The **`lapply()`** function is used to apply a function to each component of a list (`lapply` is short for "list apply"). The output of `lapply()` will be a list.

The syntax of `lapply()` is `lapply(X,FUN,...)`, where the arguments are:

- `X`: A list
- `FUN`: The function to be applied.
- `...`: Any optional arguments to be passed to the `FUN` function.

Note that there is no margin argument like in `apply()`, as lists have a single index.

```
# Return the class of each component in the L list
lapply(L,class)
```

```
## [[1]]
## [1] "integer"
##
## [[2]]
## [1] "matrix"
##
## [[3]]
## [1] "data.frame"
##
## [[4]]
## [1] "list"
```

Note: Since data frames are (stored as) lists, `lapply()` also works for data frames.

```
# Compute the range (min and max) of every column of the trees data frame
lapply(trees,range)
```

```
## $Girth
## [1]  8.3 20.6
##
## $Height
## [1] 63 87
##
## $Volume
## [1] 10.2 77.0
```

Question: How is `apply(trees,2,range)` different from `lapply(trees,range)`?

The list output from `lapply()` is particularly useful when the result from each component may have a different length (or even a different dimension or class).

```
which.median <- function(x){
  # This function inputs a vector and
  # returns the indices for values that attain the median.
  which(x == median(x))
}
lapply(trees,which.median)
```

```
## $Girth
## [1] 16 17
##
## $Height
## [1] 12 13
##
## $Volume
## [1] 11
```

2.3 The sapply() Function

The output that is returned from `lapply()` is always a list, with the same number of components as the input list. In many cases, the output could be simplified to a vector or matrix.

The **`sapply()`** function is a wrapper for `lapply()`, so `sapply()` also applies a function to each component of a list. The only difference is that `sapply()` will try to simplify the output from `lapply()` whenever possible (`sapply` is short for "simplified [l]apply"). In particular:

- If the result is a list where every component is a vector of length 1 (i.e., a scalar), then `sapply()` will return a vector.
- If the result is a list where every component is a vector of the same length (greater than 1), then `sapply()` will return a matrix.
- If the result is a list where every component is not a vector of the same length, then `sapply()` will return a list (i.e., the same output as from `lapply()`.)

By using `lapply()`, we found the class of each component of list `L`. Notice the difference when using `sapply()`.

```
sapply(L,class)
```

```
## [1] "integer"    "matrix"     "data.frame" "list"
```

The output of each application of the `class()` function is a single character value, so `sapply()` returns a vector.

Just like `lapply()`, `sapply()` also works for data frames.

```
sapply(trees,range)
```

```
##      Girth Height Volume
## [1,]   8.3     63   10.2
## [2,]  20.6     87   77.0
```

Note: Notice that `sapply(trees,range)` gives the same output as `apply(trees,2,range)`. Since a data frame is stored as a list with the column vectors as its components, `sapply()` applies functions to the components of `trees` as a list, and `apply()` with `MARGIN=2` applies functions to the columns of the coerced matrix version of `trees` (`as.matrix(trees)`). The output is the same in this case. However, since `lapply()` and `sapply()` do not coerce data frames into having columns of the same type, certain functions may produce different results.

Question: How is `apply(parks.df,2,mean)` different from `sapply(parks.df,mean)`?

2.4 The tapply() Function

The **`tapply()`** function is used to apply a function to subsets of a vector.

The syntax of `tapply()` is `tapply(X,INDEX,FUN,...,simplify=TRUE)`, where the arguments are:

- `X`: A numeric or logical vector
- `INDEX`: A factor or list of factors that identifies the subsets. Non-factors will be coerced into factors.
- `FUN`: The function to be applied.
- `...`: Any optional arguments to be passed to the `FUN` function.
- `simplify`: Logical value that specifies whether to simplify the output to a vector or matrix.

The `tapply()` function splits the values of the vector `X` into groups, each group corresponding to a level of the `INDEX` factor, then applies the function in `FUN` to each group.

As an example, we will consider the `diamonds` data in the `ggplot2` package.

```
library(ggplot2) # Load the ggplot2 package
data(diamonds) # Load the diamonds data into the workspace
diamonds
```

```
## # A tibble: 53,940 × 10
##    carat       cut color clarity depth table price     x     y     z
##    <dbl>     <ord> <ord>   <ord> <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1   0.23     Ideal     E     SI2  61.5    55   326  3.95  3.98  2.43
## 2   0.21   Premium     E     SI1  59.8    61   326  3.89  3.84  2.31
## 3   0.23      Good     E     VS1  56.9    65   327  4.05  4.07  2.31
## 4   0.29   Premium     I     VS2  62.4    58   334  4.20  4.23  2.63
## 5   0.31      Good     J     SI2  63.3    58   335  4.34  4.35  2.75
## 6   0.24 Very Good     J    VVS2  62.8    57   336  3.94  3.96  2.48
## 7   0.24 Very Good     I    VVS1  62.3    57   336  3.95  3.98  2.47
## 8   0.26 Very Good     H     SI1  61.9    55   337  4.07  4.11  2.53
## 9   0.22      Fair     E     VS2  65.1    61   337  3.87  3.78  2.49
## 10  0.23 Very Good     H     VS1  59.4    61   338  4.00  4.05  2.39
## # ... with 53,930 more rows
```

Side Note: The `diamonds` data is an example of a **tibble** object, which is a "trimmed down" version of a data frame. Tibbles are data frames with the added class `tbl_df` that makes the way R prints the object more readable (e.g., typing `diamonds` did not print the entire data frame with 53940 rows). Since tibbles are still data frames, all syntax and functions for data frames work for tibbles, so we will treat them the same way. Tibbles are one of the main objects used in the `tidyverse` packages. Documentation for tibbles can be found in the `tibble` package.

Suppose we are interested in whether the mean price of a diamond differs by the color of the diamond. The `tapply()` function can split the prices based on the color rating and compute the mean of each subset.

```
# Compute the mean of price of diamonds, grouped by color
tapply(diamonds$price,diamonds$color,mean)
```

```
##        D        E        F        G        H        I        J
## 3169.954 3076.752 3724.886 3999.136 4486.669 5091.875 5323.818
```

From the output, we see that the mean price of diamonds is higher for higher color ratings.

Suppose we want to know the mean price for each color/cut combination. The `tapply()` function can also group values based on combinations of levels from multiple factors. When using multiple factors in the `INDEX` argument, the factors need to be put into a list.

```
# Compute the mean price of diamonds for each color/cut combination
with(diamonds,tapply(price,list(color,cut),mean))
```

```
##       Fair     Good Very Good  Premium    Ideal
## D 4291.061 3405.382  3470.467 3631.293 2629.095
## E 3682.312 3423.644  3214.652 3538.914 2597.550
## F 3827.003 3495.750  3778.820 4324.890 3374.939
## G 4239.255 4123.482  3872.754 4500.742 3720.706
## H 5135.683 4276.255  4535.390 5216.707 3889.335
## I 4685.446 5078.533  5255.880 5946.181 4451.970
## J 4975.655 4574.173  5103.513 6294.592 4918.186
```

Question: How would you find out how many observations are in each category (or combination of categories)?