Data Summaries: `apply` Family of Functions

Michael Malick

Updated: 15 May 2013

`apply` Functions

- The `apply` family of functions are used to 'apply' (i.e., perform) some function iteratively on a dataset
- The `apply` functions alleviate the need to construct loops in order to execute some operation on each row, column, or element of a data structure (e.g., matrix, array, dataframe, or list)

Why `apply`?

- Summarize columns or rows of a dataset
- Compute summary statistics by factor or group
- Execute group-wise transformations
- Fit the same model to subsets of a dataframe

Some Terminology

- The term **array** is used generically when talking about the apply family of functions
    - 1-dimensional array = vector
    - 2-dimensional array = matrix
    - 3-dimensional array = array
- **Ragged array** = an array with irregular group sizes

`apply` Functions

Function	Summary	Input	Output
`apply`	apply a function to the rows/columns of a matrix	array	array or list
`tapply`	apply a function to subsets of a vector	vector	array
`by`	apply a function to subsets of a dataframe	dataframe	list
`lapply`	apply a function to each element of a list	list	list
`sapply`	apply a function to each element of a list	list	array
`mapply`	apply a function to multiple data structures	array	array or list

`apply` Functions

Function	Summary	Input	Output
`apply`	apply a function to the rows/columns of a matrix	array	array or list
`tapply`	apply a function to subsets of a vector	vector	array
`by`	apply a function to subsets of a dataframe	dataframe	list
`lapply`	apply a function to each element of a list	list	list
`sapply`	apply a function to each element of a list	list	array
`mapply`	apply a function to multiple data structures	array	array or list

Resources

- Stackoverflow discussion

apply()

When you want to apply a function to the rows or columns of a matrix (and higher-dimensional analogues)

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

- `X` = input matrix or array (if you give it a dataframe it will be coerced into an array)
- `MARGIN` = either the rows (1), the columns (2), or both (1:2)
- `FUN` = some function (e.g., `mean()`)

Example: `apply()`

```
mat <- matrix(1:25, nrow = 5, ncol = 5)

apply(mat, 1, sum)    # by rows
apply(mat, 2, sum)    # by columns
apply(mat, 1:2, sum) # by rows and columns

head(iris)
apply(iris[, 1:4], 2, mean)

# user defined function
apply(mat, 1:2, function(x) x^2)

mat*mat
```

For Loop vs. apply()

```
# Apply
apply(iris[, 1:4], 2, mean)


# For Loop
dat <- rep(NA, 4)
names(dat) <- names(iris[1:4])

for(i in 1:4) {
    dat[i] <- mean(iris[, i])
}
```

You Try...

1. Using the `apply()` function, what are the column means of the `mtcars` dataset?

tapply()

When you want to apply a function to subsets of a vector and the subsets are defined by some other vector, usually a factor

```
tapply(X, INDEX, FUN = NULL, ...)
```

- `X` = atomic object (usually a vector)
- `INDEX` = list of factors (same length as `X`)
- `FUN` = some function (e.g., `mean()`)

Example: `tapply()`

```
tapply(iris$Petal.Length, iris$Species, mean)
```

```
dat <- list(c(1, 2, 3, 4, 5),
    c("Red", "Red", "Red", "Blue", "Blue"))
tapply(dat[[1]], dat[[2]], sum)
```

```
dat <- list(c(1, 2, 3, 4, NA),
    c("Red", "Red", "Red", "Blue", "Blue"))
tapply(dat[[1]], dat[[2]], sum, na.rm = TRUE)
```

You Try...

1. Using the `tapply()` function, what is the median yield for each variety in the `barley` dataset (need to make sure the lattice package is loaded)

by()

When you want to apply a function to subsets of a data frame and the subsets are defined by some vector, usually a factor

```
by(data, INDICES, FUN, ...)
```

- `data` = a dataframe or matrix
- `INDICES` = a factor
- `FUN` = some function (e.g., `mean()`)

Example: by()

```
summary(iris[ ,1:4])
by(iris[ ,1:4], iris$Species, summary)
```

```
attach(iris)

by(iris, Species,
    function(x) lm(Sepal.Length ~
    Sepal.Width, data = x))

detach(iris)
```

You Try...

1. Using the `by()` function, produce summary statistics for yield for each variety in the `barley` dataset

`lapply()`

When you want to apply a function to each element of a list in turn and get a list back

```
lapply(X, FUN, ...)
```

- `X` = a vector or list of vectors
- `FUN` = some function (e.g., `mean()`)

Example: `lapply()`

```
dat <- list(A = c(1, 2, 3, 4, 5),
     B = c(11, 12, 13, 14, 15))

lapply(dat, mean)

lapply(iris[, 1:4], mean)# similar to apply()
```

sapply()

When you want to apply a function to each element of a list in turn, but you want a **vector** back, rather than a list

```
sapply(X, FUN, ...)
```

- `X` = a vector or list of vectors
- `FUN` = some function (e.g., `mean()`)

Example: `sapply()`

```
dat <- list(A = c(1, 2, 3, 4, 5),
     B = c(11, 12, 13, 14, 15))

sapply(dat, mean)

sapply(iris[, 1:4], mean)# same as apply()
```

mapply()

When you have several data structures (e.g., vectors, lists) and you want to apply a function to the 1st elements of each, and then the 2nd elements of each, etc., coercing the result to a vector/array as in `sapply`

```
mapply(FUN, ..., MoreArgs = NULL)
```

- `FUN` = some function (e.g., `mean()`)
- `...` = arguments to compute over
- `MoreArgs` = list of other arguments to `FUN` (e.g., `na.rm = TRUE`)

Example: `mapply()`

```
lst1 <- list(W = c(1:10), X = c(11:20))
lst2 <- list(Y = c(21:30), Z = c(31:40))

mapply(max, lst1$W, lst1$X, lst2$Y, lst2$Z)
```