Statistical Computing (36-350)
Lecture 22: Split/Apply/Combine, encore

36-350
Massive thanks to Vince Vu

10 November 2014
Agenda

- High-level overview of split/apply/combine
- Understanding how we split
- Tailoring the applied function to the split
Large data sets are usually highly structured. Structure lets us group data in many different ways. Sometimes we focus on individual pieces of data. Often we aggregate information within groups, and compare across them.
Row (column) means of a matrix

- Divide the matrix into rows (columns)
- Compute the mean of each row (column)
- Combine the results into a vector
Row Means

matrix
(array, 2 dimensional)
Row Means

(vector, 1 dimensional)

(vector, 1 dimensional)

(vector, 1 dimensional)
Row Means

\[
\text{mean}() \\
\text{mean}() \\
\text{mean}()
\]
Row Means
Row Means

vector (1 dimensional)
Another Example

Data organized into 48 continental states
Fit a different model for each of 4 different geographic regions
Splitting by Region
Splitting by Region

data.frames
Splitting by Region

lm( )

lm( )

lm( )

lm( )

lm( )
Splitting by Region
Combine into a list

list of lm objects

Lecture 22
The Basic Pattern

\[ \text{split} \quad \text{apply} \quad \text{combine} \]

\[
\begin{array}{c}
\text{split} \\
\begin{array}{c}
\text{apply} \\
\begin{array}{c}
\text{combine} \\
\end{array}
\end{array}
\end{array}
\]

\[ f \left( \begin{array}{c}
\end{array} \right) \]

An Easy Warm-Up
A Slightly Less Easy Warm-Up
The Abstract Pattern
The Basic Pattern (cont’d.)

**Split**  divide the problem into smaller pieces

**Apply**  Work on each piece independently

**Combine**  Recombine the pieces

A common pattern for both programming and data analysis, many implementations

Python: `map()`, `filter()`, `reduce()`

Google `mapReduce`

R: `split`, `*apply`, `aggregate`, ...

R: `plyr` package
Input Data Structure

Each type (array, list, data frame) has its own ways of being split
Will mostly go over howplyr does it
Inputs: $d$-dimensional Arrays

$d$ dimensions that can be subscripted independently

∴ can be split $2^d - 1$ different ways

2D arrays can be split 3 ways: rows, columns, cells
Splitting 3D Arrays

$$2^3 - 1 = 7 \text{ ways to split}$$

from Wickham (2011)
a*ply() 

y <- a*ply(.data, .margins, .fun, ...) 

.data an array  
.margins subscripts which the function gets applied over  
.fun the function to be applied  
... additional arguments to function  

Returns a * (a = array, d = data frame, l = list, _ = nothing)
Why the Funny Argument Names?

Why `.data` or `.margins` instead of `data` or `margins`?
To avoid collisions with the extra arguments to the function `.fun`
apply vs. aaply

Base R: `apply(X, 1, FUN, ...)` (rows) or `apply(X, 2, FUN, ...)` (columns)
plyr: `aaply(.data, 1, .fun, ...), aaply(.data, 2, .fun, ...)``
Pretty much equivalent, usually little point to plyr if that’s all you’re doing
Input: Lists — `l*ply()`

Lists can only be split one way

`y <- l*ply(.data, .fun, ...)`
Input: Data Frames

Can be split into groups according to the values of variables in the columns
Groups need not be of equal size
- e.g., split census tracts by state
- e.g., split census tracts by urban/suburban/rural
- e.g., split census tracts by state and type
d*ply()

```
y <- d*ply(.data, .variables, .fun, ...)
```

- `.data` a data frame
- `.variables` variables used to define groups
- `.fun` the function to be applied
- `...` additional arguments to the function

Returns array, data frame, list, nothing
The Splitting Variables

:variables can be of two forms
.(var1, var2) or
\(c(\text{var1}', \text{var2}')\)
searches .data for those variables first, then the parent environment
Looking in the parent environment can lead to some odd type-conversion issues
Advice: make the variables you want to split on part of the data frame
The Splitting Variables

.variables=.(var1) splits off a new dataframe for each unique value of var1.
.variables=.(var1, var2) splits on each unique combination of values of var2.

What if e.g. you want to compare cases where var1 >= var2 with those where var1 < var2?
The splitting variables are *still* columns of the smaller dataframes that the function gets applied to. e.g., if you split on Country in the data from lab, each resulting dataframe still has a Country column.
Data Frames Have Two Natures

- Data frame is a list of vectors
  - Can be split into separate columns
  - Can be used with `lapply()`
- Data frame responds to array-like indexing
  - Can be split like a 2D array
  - Can be used with `aapply()`
Function that is applied to each piece

Should:

- Take a piece as its first argument
- Return same type as eventual output (but there are exceptions)
- Sometimes cause side effects (plot, save, ...)

Things to Remember About the Processing Function

- Its input should be a *whole* piece of the original data
  - Row/column/slab of an array
  - A smaller dataframe from the original dataframe
- Not all of that piece may be relevant; do any selection inside the function
- You can write and debug that function by manually splitting off an example piece, and doing your processing on it first
Output Data Structure

Defines how results are combined and labeled

- Array (a)
- List (l)
- Data frame (d)
- Discarded (_) — for side effects, e.g., plotting
Output Arrays

Output organized in the expected way. Processing function should return an object of same type each time it is called.

If processing function returns a list, then output will be a list-array (list with dimensions)

Avoid this
Output Data Frames

Output will contain results with additional label columns indicating which group the result corresponds to.
Applying the pattern to your problem

- check data type of
  - input data structure
  - output data structure
- Use a built-in function, or write a processing function and test it on one piece
- Call appropriate `**ply()`
Could always do the same thing with for loops, but those are
- verbose — lots of “how”, obscures “what”
- painful/error-prone book-keeping (indices, placeholders, …)
- clumsy — hard to parallelize
Regularly sampled spatial data

\[
\text{measures} \leftarrow \text{array(STUFF, dim = c(10, 10, 100))}
\]

10 × 10 grid of locations
100 measurements at each location
Problem: Standardize measurements at each location
Standardize one location:

\[
z \leftarrow \text{scale(measures}[1, 1, \])}
\]
Iteration
Iteration

```r
y <- array(dim = dim(measures))
for(i in 1:dim(measures)[1]) {
    for(j in 1:dim(measures)[2]) {
        y[i, j, ] <- scale(measures[i, j, ])
    }
}
```
Iteration

```r
y <- array(dim = dim(measures))
for(i in 1:dim(measures)[1]) {
    for(j in 1:dim(measures)[2]) {
        y[i, j, ] <- scale(measures[i, j, ])
    }
}
```

Base R:

```r
y <- apply(measures, 1:2, scale)
```
Iteration

```r
y <- array(dim = dim(measures))
for(i in 1:dim(measures)[1]) {
    for(j in 1:dim(measures)[2]) {
        y[i, j, ] <- scale(measures[i, j, ])
    }
}
```

Base R:

```r
y <- apply(measures, 1:2, scale)
```
Iteration

```r
y <- array(dim = dim(measures))
for(i in 1:dim(measures)[1]) {
    for(j in 1:dim(measures)[2]) {
        y[i, j, ] <- scale(measures[i, j, ])
    }
}
```

Base R:

```r
y <- apply(measures, 1:2, scale)
```

plyr

```r
y <- aaply(measures, 1:2, scale)
```
Ragged spatial data

measures <- data.frame(loc.x = FOO,
                        loc.y = BAR,
                        value = BAZ)

Irregularly sampled (x,y) locations
Different number of measurements at each location
Standardize measurements at each location
Handle one location:

```r
df <- subset(measures, loc.x = 1 & loc.y = 1)
z <- scale(df$value)
```
Handle one location:

```r
df <- subset(measures, loc.x = 1 & loc.y = 1)
z <- scale(df$value)
```

Iteration
Handle one location:

```r
df <- subset(measures, loc.x = 1 & loc.y = 1)
z <- scale(df$value)
```

Iteration

Left as an exercise for the student
Handle one location:

df <- subset(measures, loc.x = 1 & loc.y = 1)
z <- scale(df$value)

Iteration

Left as an exercise for the student

Base R
Handle one location:

\[ df <- \text{subset}(measures, \text{loc.x} = 1 \& \text{loc.y} = 1) \]
\[ z <- \text{scale}(df$value) \]

Iteration

Left as an exercise for the student

Base R

Left as an exercise
Handle one location:

```r
df <- subset(measures, loc.x = 1 & loc.y = 1)
z <- scale(df$value)
```

Iteration

Left as an exercise for the student

Base R

Left as an exercise

plyr
Handle one location:

```r
df <- subset(measures, loc.x = 1 & loc.y = 1)
z <- scale(df$value)
```

Iteration

Left as an exercise for the student

Base R

Left as an exercise

```r
plyr

y <- ddply(measures, .(loc.x, loc.y), function(df) { return(scale(df$value)) } )
```

Only want to scale one column of the split-off data frame

Used an anonymous function; could also define a function previously
Don’t Force It

Don’t use split/apply/combine as a fancy way of writing for

```r
lply(1:708, function(i) {
  # several hundred lines of code follow
})
```

Use the pattern (and the tools) when:

- The problem naturally breaks the data into smaller pieces
- You can solve the problem on each piece in the same way, and independently of the other pieces
- You need to re-integrate the piecemeal solutions