Statistical Computing (36-350)
Lecture 12: Split/Apply/Combine with plyr

Cosma Shalizi
Massive thanks to Vince Vu

10 October 2012
Agenda

Abstracting split, apply, combine plyr usage Examples
Recommended reading: http://plyr.had.co.nz/
Many problems can be solved this way:

- Divide the big problem into smaller ones (split)
- Solve each piece independently and in the same way (apply)
- Put the piecemeal solutions together (combine)
split

\[ f(\quad) \]

\[ f(\quad) \]

\[ f(\quad) \]

apply

combine
Example from last time

\begin{verbatim}
x <- split(strikes, strikes$country)
y <- lapply(x, strikes_vs_left, coefficients.only=TRUE)
coefs <- do.call(rbind, y)
\end{verbatim}

split the data by country
fit the same linear model for each country
combine the results into an array
apply() arrays
lapply() list or vector, output list
sapply() list or vector, simply to vector
vapply() list or vector, safer simplify to vector
tapply() data frames (tables)
mapply() multiple vectors (special case of 2d array)
Grew up without any particular plan
Functions are useful, but
- Output is inconsistent (lists or array)
- Too much to remember
- Too much to write
The `plyr` model

Abstract the pattern:
- Input data structure (split)
- Processing function (apply)
- Output data structure (combine)

Functions named and designed consistently
*ply() — replace * with 2 characters:
- first character: input type array, data frame or list (a, d, l)
- second character: output type array, data frame, list, or discard (a, d, l, _)

Lecture 10
Each type (array, list, data frame) has its own ways of being split
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
$2^3 - 1 = 7$ ways to split

from Wickham (2011)
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)
Lists can only be split one way

\[ y \leftarrow \text{listy}(\text{.data, .fun, ...}) \]
x <- list(alice="Wonderland", babur="Samarkand")

What’s the difference between
x[[1]] and x[1]?
$x[[1]]$ is the 1st component of $x$
It is a string, namely "Wonderland"
It has no components
$x[[1]]$ is the 1st component of $x$
It is a string, namely "Wonderland"
It has no components
$x[1]$ is a subset of $x$
It is a list, which happens to be of length 1
It has components, what is $x[1][[1]]$?
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()

```r
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.

**.variables** can be of two forms:

- `(var1, var2)`
- `c('var1', 'var2')`

searches `.data` for those variables first, then the parent environment.
Data frame is a list of vectors
∴ Can be split into separate columns
∴ Can be used with `lply()`
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, ...)
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 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)
Output will contain results with additional label columns indicating which group the result corresponds to.
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()`
Install plyr
install.packages("plyr", dependencies = T)

Load plyr
library(plyr)

(use require in code, returns TRUE or FALSE as appropriate)
Regularly sampled spatial data

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

10 \times 10 \text{ grid of locations}
100 \text{ measurements at each location}

Problem: Standardize measurements at each location

Standardize one location:

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

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

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

Base R:

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

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

Base R:

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

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

Base R:

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

plyr

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

\[ x <- \text{data.frame}(\text{loc.x} = \text{FOO}, \]
\[ \text{loc.y} = \text{BAR}, \]
\[ \text{value} = \text{BAZ}) \]

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

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

```r
df <- subset(x, 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

```r
y <- ddply(x, .(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
Handle one location:

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

Iteration

Left as an exercise for the student
Handle one location:

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

Iteration

Left as an exercise for the student

Base R
Handle one location:

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

Iteration

Left as an exercise for the student

Base R

Left as an exercise
Handle one location:

df <- subset(x, 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:

df <- subset(x, 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

y <- ddply(x, .(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
Continuing the parliamentary politics/strikes connection

```r
strikes <- read.csv("http://www.stat.cmu.edu/~cshalizi/uADA/12/hw/06/strikes.csv")
strikes <- strikes[,,-7] # drop "centralization" variable

<table>
<thead>
<tr>
<th>country</th>
<th>year</th>
<th>strike.volume</th>
<th>unemployment</th>
<th>inflation</th>
<th>left.parliament</th>
<th>density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Australia</td>
<td>1983</td>
<td>313</td>
<td>9.8</td>
<td>10.1</td>
<td>60</td>
<td>48.5</td>
</tr>
<tr>
<td>Australia</td>
<td>1984</td>
<td>241</td>
<td>8.9</td>
<td>4</td>
<td>55.4</td>
<td>47.6</td>
</tr>
<tr>
<td>Australia</td>
<td>1985</td>
<td>226</td>
<td>8.2</td>
<td>6.7</td>
<td>55.4</td>
<td>45.9</td>
</tr>
<tr>
<td>Austria</td>
<td>1951</td>
<td>43</td>
<td>3.5</td>
<td>27.5</td>
<td>43.6</td>
<td>NA</td>
</tr>
<tr>
<td>Austria</td>
<td>1952</td>
<td>39</td>
<td>4.7</td>
<td>13.6</td>
<td>43.6</td>
<td>NA</td>
</tr>
<tr>
<td>Austria</td>
<td>1953</td>
<td>20</td>
<td>5.8</td>
<td>-1.6</td>
<td>46.7</td>
<td>NA</td>
</tr>
</tbody>
</table>
```
If left parties gain an extra 1% of the gov’t., how much more strike activity?
Try this as a linear model
Analysis in a function

strikes_vs_left <- function(df, coefficients.only = FALSE) {
  fit <- lm(strike.volume ~ left.parliament, data = df)
  if (coefficients.only) {
    return(coefficients(fit))
  } else {
    return(fit)
  }
}
Input: a data frame, \texttt{strikes}
Split by: \texttt{country}
Output desired: an array of regression coefficients $\Rightarrow$ an array
$\therefore$ use \texttt{daply}
Processing function: \texttt{strikes\_vs\_left}
Iteration

```
coefs <- matrix(nrow=nlevels(strikes$country),ncol=2)
for (i in 1:nlevels(strikes$country)) {
  x <- subset(strikes, country==levels(strikes$country)[i])
  coefs[i,] <- strikes_vs_left(x, coefficients.only=TRUE)
}
rownames(coefs) <- levels(strikes$country)

Base R
x <- split(strikes, strikes$country)
y <- lapply(x, strikes_vs_left, coefficients.only=TRUE)
coefs <- do.call(rbind, y)

plyr
coefs <- daply(strikes, .(country), strikes_vs_left, coefficients.only=TRUE)
```
Iteration

coeffs <- matrix(nrow=nlevels(strikes$country),ncol=2)
for (i in 1:nlevels(strikes$country)) {
  x <- subset(strikes, country==levels(strikes$country)[i])
  coefs[i,] <- strikes_vs_left(x,coefficients.only=TRUE)
}
rownames(coefs) <- levels(strikes$country)
Iteration

coeys <- matrix(nrow=nlevels(strikes$country),ncol=2)
for (i in 1:nlevels(strikes$country)) {
    x <- subset(strikes, country==levels(strikes$country)[i])
    coefs[i,] <- strikes_vs_left(x,coefficients.only=TRUE)
}
rownames(coefs) <- levels(strikes$country)

Base R
Iteration

coops <- matrix(nrow=nlevels(strikes$country),ncol=2)
for (i in 1:nlevels(strikes$country)) {
  x <- subset(strikes, country==levels(strikes$country)[i])
  coefs[i,] <- strikes_vs_left(x,coefficients.only=TRUE)
}
rownames(coefs) <- levels(strikes$country)

Base R

x <- split(strikes, strikes$country)
y <- lapply(x, strikes_vs_left, coefficients.only=TRUE)
coefs <- do.call(rbind, y)
Iteration

coefs <- matrix(nrow=nlevels(strikes$country),ncol=2)
for (i in 1:nlevels(strikes$country)) {
  x <- subset(strikes, country==levels(strikes$country)[i])
  coefs[i,] <- strikes_vs_left(x,coefficients.only=TRUE)
}
rownames(coefs) <- levels(strikes$country)

Base R

x <- split(strikes, strikes$country)
y <- lapply(x, strikes_vs_left, coefficients.only=TRUE)
coefs <- do.call(rbind, y)

plyr
Iteration

coefs <- matrix(nrow=nlevels(strikes$country),ncol=2)
for (i in 1:nlevels(strikes$country)) {
  x <- subset(strikes, country==levels(strikes$country)[i])
  coefs[i,] <- strikes_vs_left(x,coefficients.only=TRUE)
}
rownames(coefs) <- levels(strikes$country)

Base R

x <- split(strikes, strikes$country)
y <- lapply(x, strikes_vs_left, coefficients.only=TRUE)
coefs <- do.call(rbind, y)

plyr

coops <- daply(strikes, .(country), strikes_vs_left, coefficients.only=TRUE)
How many complete observations per country?

```
nrow.omitting.nas <- function(df) { nrow(na.omit(df)) }
daply(strikes, .(country), nrow.omitting.nas)
```

How many complete observations per year?

```
daply(strikes, .(year), nrow.omitting.nas)
```
For each country, take median of each variable
Omit a year with NAs, but only for that variable
e.g., omit 1951 for Austria for density but not inflation
In: data frame
Out: data frame (country by variables)
Processing:

\[
\text{colMedians} <- \text{function(df)} \{ \text{apply(df,2,median,na.rm=TRUE)} \}\]

\[
\text{medians} <- \text{ddply(strikes, .(country), colMedians)}
\]

Doesn’t work!

\[
> \text{head(medians)}
\]

<table>
<thead>
<tr>
<th>country</th>
<th>year</th>
<th>strike.volume</th>
<th>unemployment</th>
<th>inflation</th>
<th>left.parliament</th>
<th>density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Australia</td>
<td>1968</td>
<td>326</td>
<td>2.5</td>
<td>5.9</td>
<td>41.0</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>Austria</td>
<td>1968</td>
<td>11</td>
<td>2.0</td>
<td>4.0</td>
<td>47.9</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
<td>&lt;NA&gt;</td>
<td>42.1</td>
</tr>
<tr>
<td>Canada</td>
<td>1968</td>
<td>470</td>
<td>5.6</td>
<td>4.0</td>
<td>59.0</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>Denmark</td>
<td>1968</td>
<td>38</td>
<td>6.0</td>
<td>6.5</td>
<td>46.9</td>
<td>&lt;NA&gt;</td>
</tr>
<tr>
<td>Finland</td>
<td>1968</td>
<td>155</td>
<td>2.2</td>
<td>7.1</td>
<td>27.0</td>
<td>&lt;NA&gt;</td>
</tr>
</tbody>
</table>
Problem: country, a factor variable, is still part of each split data frame; medians don't make sense
Slightly inelegant solution:

\[
\text{colMedians} \leftarrow \text{function}(\text{df}) \{ \text{apply}(\text{df}[,-1], 2, \text{median}, \text{na.rm}=\text{TRUE}) \} \\
\text{medians} \leftarrow \text{ddply}(\text{strikes}, .(\text{country}), \text{colMedians})
\]

<table>
<thead>
<tr>
<th>country</th>
<th>year</th>
<th>strike.volume</th>
<th>unemployment</th>
<th>inflation</th>
<th>left.parliament</th>
<th>density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Australia</td>
<td>1968.0</td>
<td>326.0</td>
<td>2.5</td>
<td>4.30</td>
<td>41.0</td>
<td>48.10</td>
</tr>
<tr>
<td>Austria</td>
<td>1968.0</td>
<td>11.0</td>
<td>2.0</td>
<td>4.00</td>
<td>47.9</td>
<td>60.90</td>
</tr>
<tr>
<td>Belgium</td>
<td>1965.5</td>
<td>186.5</td>
<td>2.9</td>
<td>3.85</td>
<td>33.0</td>
<td>42.10</td>
</tr>
<tr>
<td>Canada</td>
<td>1968.0</td>
<td>470.0</td>
<td>5.6</td>
<td>3.70</td>
<td>59.0</td>
<td>32.45</td>
</tr>
<tr>
<td>Denmark</td>
<td>1968.0</td>
<td>38.0</td>
<td>6.0</td>
<td>6.50</td>
<td>46.9</td>
<td>62.20</td>
</tr>
<tr>
<td>Finland</td>
<td>1968.0</td>
<td>155.0</td>
<td>2.2</td>
<td>7.10</td>
<td>27.0</td>
<td>61.10</td>
</tr>
</tbody>
</table>

More elegant: have \text{colMedians} figure out which columns are numeric (or logical), drop the rest.
Similar in effect to base R

\[
\text{aggregate}((\text{strikes}[,-1]), \text{by}=\text{list}(\text{strikes}$country), \text{FUN=median}, \text{na.rm}=\text{TRUE})
\]
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
Summary

1. `plyr` simplifies using split/apply/combine, abstracting away from implementation details
2. You focus on figuring out the input type, the output type, and the processing function
3. Try writing a processing function for one piece, then generalize