

Lecture 14: Split/Apply/Combine

STAT GR5206 *Statistical Computing & Introduction to Data Science*

Gabriel Young
Columbia University

June 27, 2017

- Gradient Descent
- Newton's Method
- Logistic Regression
- Maximum Likelihood Estimation in Logistic Regression

Topics for Today

- **The Split/Apply/Combine Model.** A general strategy for working with big data.
- **The `plyr` Package.** Improves on the `apply()` family.

The Split/Apply/Combine Model

Iterating in R without `for()` (a loop)

- Subsetting by indexing with conditionals
- `apply()`: apply a function to rows or columns of a matrix or data frame
- `lapply()`, `sapply()`: apply a function to elements of a list or vector
- `cbind()`, `rbind()`: concatenate these objects in a known pattern.

Split/Apply/Combine

Today we will learn a general strategy that can be summarized in three conceptual steps:

- **Split** whatever data object we have into meaningful chunks
- **Apply** the function of interest to each element in this division
- **Combine** the results into a new object of the desired structure

Split/Apply/Combine

Today we will learn a general strategy that can be summarized in three conceptual steps:

- **Split** whatever data object we have into meaningful chunks
- **Apply** the function of interest to each element in this division
- **Combine** the results into a new object of the desired structure

These are conceptual steps; often the apply and combine steps can be performed for us by a single call to the appropriate function from the `apply()` family

Simple but powerful

Does split-apply-combine sound simple? It is, but it's very powerful when combined with the right data structures.

- As usual, compared to explicit `for()` loops, often requires far less code.
- Makes you think: **What do I want to do?** vs **How do I want to do it?**
- Sets you in the right direction towards learning how to use MapReduce/Hadoop for really, really big data sets.

Example: Strikes dataset

Data set on 18 countries over 35 years (compiled by Bruce Western, in the Sociology Department at Harvard University). The measured variables:

- `country, year`: country and year of data collection
- `strike.volume`: days on strike per 1000 workers
- `unemployment`: unemployment rate
- `inflation`: inflation rate
- `left.parliament`: leftwing share of the government
- `centralization`: centralization of unions
- `density`: density of unions

Example: Strikes dataset

Since $18 \times 35 = 630$, some years missing from some countries

```
> strikes <- read.csv("strikes.csv", as.is = TRUE)
> dim(strikes)
```

```
[1] 625    8
```

```
> head(strikes, 3)
```

	country	year	strike.volume	unemployment	inflation
1	Australia	1951	296	1.3	19.8
2	Australia	1952	397	2.2	17.2
3	Australia	1953	360	2.5	4.3

	left.parliament	centralization	density
1	43	0.3748588	NA
2	43	0.3751829	NA
3	43	0.3745076	NA

Example: Strikes dataset

Our Research Question

Is there a relationship between a country's ruling party alignment (left versus right) and the volume of strikes?

How could we approach this?

- Worst way: by hand, write 18 separate code blocks
- Bad way: explicit `for()` loop, where we loop over countries
- Best way: split appropriately, then use `sapply()`

Example: Strikes dataset

Let's Study Just a Single Country

```
> italy.strikes <- subset(strikes, country == "Italy")  
> # Equivalently,  
> italy.strikes <- strikes[strikes$country == "Italy", ]  
> dim(italy.strikes)
```

```
[1] 35  8
```

Example: Strikes dataset

Let's Study Just a Single Country

```
> head(italy.strikes, 5)
```

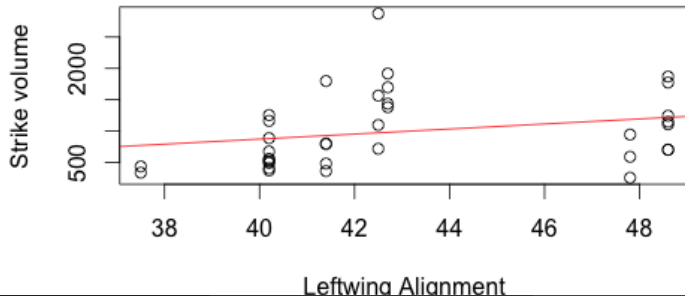
	country	year	strike.volume	unemployment	inflation
311	Italy	1951	437	8.8	14.3
312	Italy	1952	337	9.5	1.9
313	Italy	1953	545	10.0	1.4
314	Italy	1954	493	8.7	2.4
315	Italy	1955	511	7.5	2.3

	left.parliament	centralization	density
311	37.5	0.2513799	NA
312	37.5	0.2489860	NA
313	40.2	0.2482739	NA
314	40.2	0.2466577	NA
315	40.2	0.2540366	NA

Example: Strikes dataset

```
> italy.fit <- lm(strike.volume ~ left.parliament,  
+               data = italy.strikes)  
> plot(strike.volume~left.parliament, data = italy.strikes,  
+      main="Italy Strike Volume Versus Leftwing Alignment",  
+      ylab = "Strike volume", xlab = "Leftwing Alignment")  
> abline(italy.fit, col = 2)
```

Italy Strike Volume Versus Left-Wing Alignment



Example: Strikes dataset

One Down, Seventeen To Go

It's tedious and dangerous to do this repeatedly – typos! How can we do this an easier way?

Example: Strikes dataset

One Down, Seventeen To Go

It's tedious and dangerous to do this repeatedly – typos! How can we do this an easier way?

Now let's generalize our functions. We want the linear model coefficients:

```
> my.strike.lm <- function(country.df) {  
+   return(lm(strike.volume ~ left.parliament,  
+             data = country.df)$coeff)  
+ }  
  
> my.strike.lm(subset(strikes, country == "Italy"))
```

```
(Intercept) left.parliament  
-738.74531      40.29109
```


Example: Strikes dataset

We could for() loop it...

```
> strike.coef <- NULL
> my.countries <- c("France", "Italy", "USA")
> for (this.country in my.countries) {
+   country.dat <- subset(strikes, country == this.country)
+   new.coefs <- my.strike.lm(country.dat)
+   strike.coef <- cbind(strike.coef, new.coefs)
+ }
> colnames(strike.coef) <- my.countries
> strike.coef
```

	France	Italy	USA
(Intercept)	202.4261408	-738.74531	111.440651
left.parliament	-0.4255319	40.29109	5.918647

Example: Strikes dataset

The Best Way

Steps:

1. Split our data into appropriate chunks, each of which can be handled by our function. Here, the function `split()` is often helpful. Recall, `split(df, f = my.factor)` splits a data frame `df` into several data frames, defined by constant levels of the factor `my.factor`.

Example: Strikes dataset

The Best Way

Steps:

1. Split our data into appropriate chunks, each of which can be handled by our function. Here, the function `split()` is often helpful. Recall, `split(df, f = my.factor)` splits a data frame `df` into several data frames, defined by constant levels of the factor `my.factor`.
2. Apply our function to each chunk of data. Here, the functions `lapply()` or `sapply()` are often helpful.
3. Combine the results.

Example: Strikes dataset

One Down, Seventeen To Go

First we subset for every country using `split()`.

```
> strikes.split <- split(strikes, strikes$country)
> names(strikes.split)
```

```
[1] "Australia"  "Austria"    "Belgium"    "Canada"
[5] "Denmark"    "Finland"    "France"     "Germany"
[9] "Ireland"    "Italy"      "Japan"      "Netherlands"
[13] "New.Zealand" "Norway"     "Sweden"     "Switzerland"
[17] "UK"         "USA"
```

Example: Strikes dataset

The Best Way

So we want to apply `my.strikes.lm()` to each data frame in `strikes.split`. Think about what the output will be from each function call: vector of length 2 (intercept and slope), so we can use `sapply()`.

Example: Strikes dataset

The Best Way

So we want to apply `my.strikes.lm()` to each data frame in `strikes.split`. Think about what the output will be from each function call: vector of length 2 (intercept and slope), so we can use `sapply()`.

```
> strike.coef <- sapply(strikes.split[1:12], my.strike.lm)
> strike.coef
```

	Australia	Austria	Belgium	Canada
(Intercept)	414.7712254	423.077279	-56.926780	-227.8218
left.parliament	-0.8638052	-8.210886	8.447463	17.6766
	Denmark	Finland	France	Germany
(Intercept)	-1399.35735	108.2245	202.4261408	95.657134
left.parliament	34.34477	12.8422	-0.4255319	-1.312305
	Ireland	Italy	Japan	Netherlands
(Intercept)	-94.78661	-738.74531	964.73750	-32.627678
left.parliament	55.46721	40.29109	-24.07595	1.694387

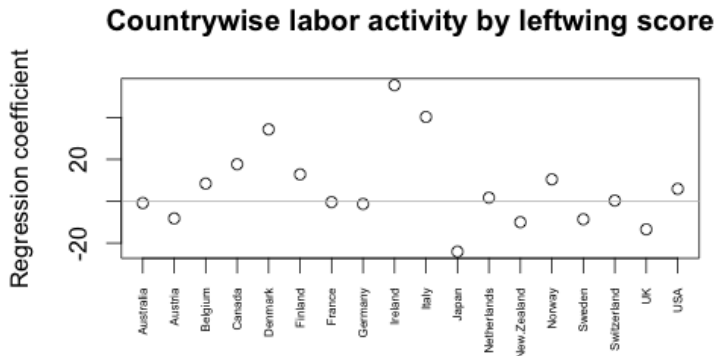
Example: Strikes dataset

The Best Way

We don't care about the intercepts, only the slopes (2nd row). Some are positive, some are negative! Let's plot them:

```
> plot(1:ncol(strike.coef), strike.coef[2, ], xaxt = "n",  
+      xlab = "", ylab = "Regression coefficient",  
+      main="Countrywise labor activity by leftwing score")  
> axis(side = 1, at = 1:ncol(strike.coef),  
+      labels = colnames(strike.coef), las = 2,  
+      cex.axis = 0.5)  
> abline(h = 0, col = "grey")
```

Example: Strikes dataset



Check Yourself

Tasks

- Using `split()` and `sapply()`, compute the average unemployment rate, inflation rates, and strike volume for each year in the `strikes` data set. The output should be a matrix of dimension 3×35 .
- Display the average unemployment rate by year and the average inflation rate by year, in the same plot. Label the axes and title the plot appropriately. Include an informative legend.

Check Yourself

Solution

```
> three.mean <- function(df) {  
+   return(apply(df[, c("unemployment", "inflation",  
+                       "strike.volume")], 2, mean))  
+ }  
  
> years.split <- split(strikes, strikes$year)  
> years.mat   <- sapply(years.split, three.mean)  
> dim(years.mat)
```

```
[1] 3 35
```

Check Yourself

Solution

```
> years.mat[, 1:8]
```

	1951	1952	1953	1954
unemployment	3.088889	3.683333	3.594444	3.505556
inflation	13.088889	5.794444	1.333333	1.833333
strike.volume	359.222222	588.666667	211.944444	139.333333
	1955	1956	1957	1958
unemployment	3.044444	3.033333	3.055556	3.422222
inflation	1.294444	3.705556	3.255556	3.472222
strike.volume	215.277778	561.944444	216.111111	145.611111

Check Yourself

Solution

```
> max.rate <- max(years.mat[1:2, ])  
> min.rate <- min(years.mat[1:2, ])  
> plot(colnames(years.mat), years.mat[1, ], xlab = "Year",  
+       ylab="Rate", type="l", ylim = c(min.rate, max.rate))  
> points(colnames(years.mat), years.mat[2, ], type = "l",  
+        col = "red")  
> legend("topleft", c("Unemployment", "Inflation"),  
+        fill = c("black", "red"), cex = .5)
```

Check Yourself

Solution



Using plyr

Reminder: iterating in R without `for()`

We've learned some tools in R for iteration without explicit `for()` loops:

- Indexing with conditionals + vectorization
- `apply()`: apply a function to rows or columns of a matrix or data frame
- `lapply()`: apply a function to elements of a list or vector
- `sapply()`: same as the above, but simplify the output (if possible)
- `tapply()`: apply a function to levels of a factor vector

Reminder: iterating in R without `for()`

We've learned some tools in R for iteration without explicit `for()` loops:

- Indexing with conditionals + vectorization
- `apply()`: apply a function to rows or columns of a matrix or data frame
- `lapply()`: apply a function to elements of a list or vector
- `sapply()`: same as the above, but simplify the output (if possible)
- `tapply()`: apply a function to levels of a factor vector

Clever indexing + vectorization is always useful, when possible.

The `apply()` family is often useful, but it has some issues: primarily, inconsistent output.

The `plyr` package

Most popular R package of all time (most downloads): `plyr`

Provides us with an extremely useful family of apply-like functions.

Advantage over the built-in `apply()` family is its consistency

The `plyr` package

Most popular R package of all time (most downloads): `plyr`

Provides us with an extremely useful family of apply-like functions.

Advantage over the built-in `apply()` family is its consistency

All `plyr` functions are of the form `**ply()`. Replace `**` with characters denoting types:

- First character: input type, one of `a`, `d`, `l`
- Second character: output type, one of `a`, `d`, `l`, or `_` (drop)

`a*ply()`: The Input is an Array

The signature for all `a*ply()` functions is:

```
a*ply(.data, .margins, .fun, ...)
```

- `.data` : an array
- `.margins` : index (or indices) to split the array by
- `.fun` : the function to be applied to each piece
- `...` : additional arguments to be passed to the function

`a*ply()`: The Input is an Array

The signature for all `a*ply()` functions is:

```
a*ply(.data, .margins, .fun, ...)
```

- `.data` : an array
- `.margins` : index (or indices) to split the array by
- `.fun` : the function to be applied to each piece
- `...` : additional arguments to be passed to the function

Note that this looks like:

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

Examples

```
> my.array          <- array(1:27, c(3,3,3))  
> rownames(my.array) <- c("R1", "R2", "R3")  
> colnames(my.array) <- c("C1", "C2", "C3")  
> dimnames(my.array)[[3]] <- c("Bart", "Lisa", "Maggie")
```

Examples

```
> my.array
```

```
, , Bart
```

	C1	C2	C3
R1	1	4	7
R2	2	5	8
R3	3	6	9

```
, , Lisa
```

	C1	C2	C3
R1	10	13	16
R2	11	14	17
R3	12	15	18

Examples

```
> my.array[, , 3]
```

	C1	C2	C3
R1	19	22	25
R2	20	23	26
R3	21	24	27

Examples

```
> library(plyr)
> aaply(my.array, 1, sum) # Get back an array
```

```
  R1  R2  R3
117 126 135
```

```
> adply(my.array, 1, sum) # Get back a data frame
```

```
  X1  V1
1 R1 117
2 R2 126
3 R3 135
```


Examples

```
> alply(my.array, 1, sum) # Get back a list
```

```
$`1`  
[1] 117
```

```
$`2`  
[1] 126
```

```
$`3`  
[1] 135
```

```
attr("split_type")  
[1] "array"  
attr("split_labels")  
  X1  
1 R1
```

Examples

```
> aapply(my.array, 2:3, sum) # Get back a 3 x 3 array
```

	X2		
X1	Bart	Lisa	Maggie
C1	6	33	60
C2	15	42	69
C3	24	51	78

Examples

```
> adply(my.array, 2:3, sum) # Get back a data frame
```

	X1	X2	V1
1	C1	Bart	6
2	C2	Bart	15
3	C3	Bart	24
4	C1	Lisa	33
5	C2	Lisa	42
6	C3	Lisa	51
7	C1	Maggie	60
8	C2	Maggie	69
9	C3	Maggie	78

Examples

```
> alply(my.array, 2:3, sum) # Get back a list
```

```
$`1`
```

```
[1] 6
```

```
$`2`
```

```
[1] 15
```

```
$`3`
```

```
[1] 24
```

```
$`4`
```

```
[1] 33
```

```
$`5`
```

```
[1] 42
```

`l*ply()` : The Input is a List

The signature for all `l*ply()` functions is:

```
l*ply(.data, .fun, ...)
```

- `.data` : a list
- `.fun` : the function to be applied to each element
- `...` : additional arguments to be passed to the function

`l*ply()` : The Input is a List

The signature for all `l*ply()` functions is:

```
l*ply(.data, .fun, ...)
```

- `.data` : a list
- `.fun` : the function to be applied to each element
- `...` : additional arguments to be passed to the function

Note that this looks like:

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

Examples

```
> my.list <- list(nums = rnorm(1000), lets = letters,  
+                pops = state.x77[ , "Population"])  
> head(my.list[[1]], 5)
```

```
[1] -0.1592102  1.2775853 -0.1612624 -1.0191692 -0.6084487
```

```
> head(my.list[[2]], 5)
```

```
[1] "a" "b" "c" "d" "e"
```

```
> head(my.list[[3]], 5)
```

Alabama	Alaska	Arizona	Arkansas	California
3615	365	2212	2110	21198

Examples

```
> laply(my.list, range) # Get back an array
```

	1	2
[1,]	"-3.41929284262538"	"3.49197028620251"
[2,]	"a"	"z"
[3,]	"365"	"21198"

```
> ldply(my.list, range) # Get back a data frame
```

	.id	V1	V2
1	nums	-3.41929284262538	3.49197028620251
2	lets	a	z
3	pops	365	21198

Examples

```
> llply(my.list, range) # Get back a list
```

```
$nums
```

```
[1] -3.419293  3.491970
```

```
$lets
```

```
[1] "a" "z"
```

```
$pops
```

```
[1] 365 21198
```

Examples

```
> # Doesn't work! Outputs have different types/lengths
> # laply(my.list, summary)
> # ldply(my.list, summary)
> llply(my.list, summary) # Works just fine
```

\$nums

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
-3.41900	-0.75450	-0.08982	-0.05617	0.60990	3.49200

\$lets

Length	Class	Mode
26	character	character

\$pops

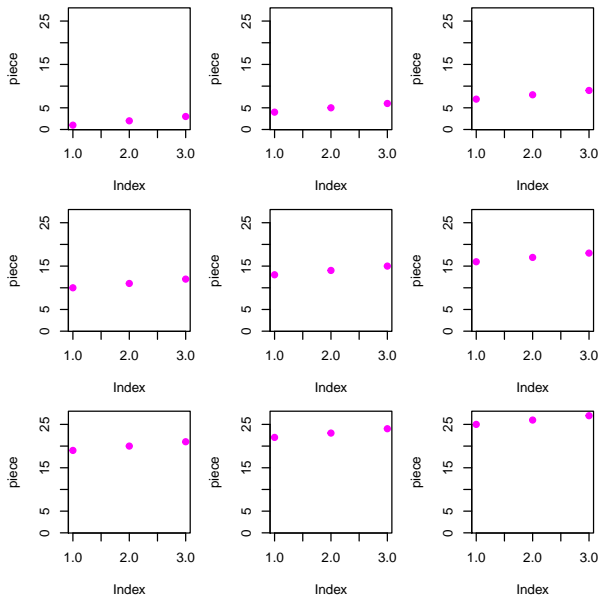
Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
365	1080	2838	4246	4968	21200

The Fourth Option for *

The fourth option for * is `_`: the function `a_ply()` (or `l_ply()`) has no explicit return object, but still runs the given function over the given array (or list), possibly producing side effects

```
> par(mfrow = c(3, 3), mar = c(4, 4, 1, 1))  
> a_ply(my.array, 2:3, plot, ylim = range(my.array),  
+       pch = 19, col = 6)
```

The Fourth Option for *



`d*ply()` : The Input is a Data Frame

The signature for all `d*ply()` functions is:

```
d*ply(.data, .variables, .fun, ...)
```

- `.data` : a data frame
- `.variables` : variable (or variables) to split the data frame by
- `.fun` : the function to be applied to each piece
- `...` : additional arguments to be passed to the function

`d*ply()` : The Input is a Data Frame

The signature for all `d*ply()` functions is:

```
d*ply(.data, .variables, .fun, ...)
```

- `.data` : a data frame
- `.variables` : variable (or variables) to split the data frame by
- `.fun` : the function to be applied to each piece
- `...` : additional arguments to be passed to the function

Note that this looks like:

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

Strikes Data Set, Revisited

Recall, data set on political economy of strikes:

```
> # Function to compute coefficients from regressing number
> # of strikes (per 1000 workers) on leftwing share of the
> # government
>
> my.strike.lm <- function(country.df) {
+   return(coef(lm(strike.volume ~ left.parliament,
+                   data=country.df)))
+ }
```

Strikes Data Set, Revisited

```
> # Getting regression coefficients separately
> # for each country, old way:
>
> strikes.list <- split(strikes, f = strikes$country)
> strikes.coefs <- sapply(strikes.list, my.strike.lm)
> strikes.coefs[, 1:12]
```

	Australia	Austria	Belgium	Canada
(Intercept)	414.7712254	423.077279	-56.926780	-227.8218
left.parliament	-0.8638052	-8.210886	8.447463	17.6766
	Denmark	Finland	France	Germany
(Intercept)	-1399.35735	108.2245	202.4261408	95.657134
left.parliament	34.34477	12.8422	-0.4255319	-1.312305
	Ireland	Italy	Japan	Netherlands
(Intercept)	-94.78661	-738.74531	964.73750	-32.627678
left.parliament	55.46721	40.29109	-24.07595	1.694387

Strikes Data Set, Revisited

```
> # Getting regression coefficient separately for each  
> # country, new way, in three formats:  
>  
> strike.coef.a <- dapply(strikes, .(country), my.strike.lm)  
> # Get back an array, note the difference to sapply()  
>  
> head(strike.coef.a)
```

country	(Intercept)	left.parliament
Australia	414.77123	-0.8638052
Austria	423.07728	-8.2108864
Belgium	-56.92678	8.4474627
Canada	-227.82177	17.6766029
Denmark	-1399.35735	34.3447662
Finland	108.22451	12.8422018

Strikes Data Set, Revisited

```
> strike.coef.d <- ddply(strikes, .(country), my.strike.lm)
> head(strike.coef.d) # Get back a data frame
```

	country	(Intercept)	left.parliament
1	Australia	414.77123	-0.8638052
2	Austria	423.07728	-8.2108864
3	Belgium	-56.92678	8.4474627
4	Canada	-227.82177	17.6766029
5	Denmark	-1399.35735	34.3447662
6	Finland	108.22451	12.8422018

Strikes Data Set, Revisited

```
> strike.coef.l <- dlply(strikes, .(country), my.strike.lm)
> head(strike.coef.l, 3) # Get back a list
```

```
$Australia
  (Intercept) left.parliament
414.7712254      -0.8638052
```

```
$Austria
  (Intercept) left.parliament
423.077279      -8.210886
```

```
$Belgium
  (Intercept) left.parliament
-56.926780      8.447463
```

Splitting on Two or More Variables

The function `d*ply()` makes it very easy to split on two (or more) variables: we just specify them, separated by a “,” in the `.variables` argument

```
> # First create a variable that indicates whether the year
> # is pre 1975, and add it to the data frame
>
> strikes$yearPre1975 <- strikes$year <= 1975
```

```
> # Then use (say) ddply() to compute regression
> # coefficients for each country pre & post 1975
>
> strike.coef.75 <- ddply(strikes, .(country, yearPre1975),
+                           my.strike.lm)
> dim(strike.coef.75) # Note there are 18 x 2 = 36 rows
```

```
[1] 36  4
```

Splitting on Two or More Variables

```
> head(strike.coef.75)
```

	country	yearPre1975	(Intercept)	left.parliament
1	Australia	FALSE	973.34088	-11.8094991
2	Australia	TRUE	-169.59900	12.0170866
3	Austria	FALSE	19.51823	-0.3470889
4	Austria	TRUE	400.83004	-7.7051918
5	Belgium	FALSE	-4182.06650	148.0049261
6	Belgium	TRUE	-103.67439	9.5802824

Splitting on Two or More Variables

```
> # Can also create factor variables on-the-fly with I()  
>  
> strike.coef.75 <- ddply(strikes,  
+                          .(country, I(year<=1975)),  
+                          my.strike.lm)  
> dim(strike.coef.75) # Again, 18 x 2 = 36 rows
```

```
[1] 36  4
```

Splitting on Two or More Variables

```
> head(strike.coef.75)
```

	country	I(year <= 1975)	(Intercept)	left.parliament
1	Australia	FALSE	973.34088	-11.8094991
2	Australia	TRUE	-169.59900	12.0170866
3	Austria	FALSE	19.51823	-0.3470889
4	Austria	TRUE	400.83004	-7.7051918
5	Belgium	FALSE	-4182.06650	148.0049261
6	Belgium	TRUE	-103.67439	9.5802824

Parallelization

- What happens if we have a really large data set and we want to use split-apply-combine?
- If the individual tasks are unrelated, then we should be speed up the computation by performing them **in parallel**.

Parallelization

- What happens if we have a really large data set and we want to use split-apply-combine?
- If the individual tasks are unrelated, then we should be speed up the computation by performing them **in parallel**.
- The `plyr` functions make this quite easy: let's take a look at the full signature for `dapply()`:

```
dapply(.data, .variables, .fun = NULL, ...,  
      .progress = "none", .inform = FALSE, .drop_i = TRUE,  
      .drop_o = TRUE, .parallel = FALSE, .paropts = NULL)
```

- The second to last argument `.parallel` (default `FALSE`) is for parallelization. If set to `TRUE`, then it performs the individual tasks in parallel, using the `foreach` package
- The last argument `.paropts` is for more advanced parallelization, these are additional arguments to be passed to `foreach`

Parallelization

- For more, read the `foreach` package first. May take some time to set up the parallel backend (this is often system specific)
- But once set up, parallelization is simple and beautiful with `**ply()`! The difference is just, e.g.,

```
daply(strikes.df, .(country), my.strike.lm)
```

versus

```
daply(strikes.df, .(country), my.strike.lm,  
      .parallel = TRUE)
```

Tasks

- Compute the average inflation rate for each country pre and post 1975, from `strikes`, using a single call to `daply()`, i.e., without using any auxiliary columns in `strikes`, like the ones created in `yearPre1975`, `countryPre1975`. (Hint: Recall the function `I()`. You'll also have to write a quick function to get the inflation mean.)
- Do the same thing with `split()` and `sapply()` to check your results.

Solutions

```
> inflation.mean <- function(country.df) {  
+   return(mean(country.df$inflation))  
+ }  
  
> inflation75 <- ddply(strikes, .(country, I(year<=1975)),  
+                       inflation.mean)  
  
> dim(inflation75)
```

```
[1] 36  3
```

Solutions

```
> head(inflation75)
```

	country	I(year <= 1975)	V1
1	Australia	FALSE	9.460
2	Australia	TRUE	5.448
3	Austria	FALSE	5.080
4	Austria	TRUE	5.112
5	Belgium	FALSE	6.400
6	Belgium	TRUE	3.700

Check Yourself

Solutions

```
> split.list <- list(strikes$country, I(strikes$year<=1975))
> data.split <- split(strikes, f = split.list)
> inflation75 <- sapply(data.split, inflation.mean)
> dim(inflation75)
```

NULL

```
> head(inflation75)
```

Australia.FALSE	Austria.FALSE	Belgium.FALSE
9.46	5.08	6.40
Canada.FALSE	Denmark.FALSE	Finland.FALSE
8.11	9.17	9.70

Reshaping Dataframes

Common to have data where some variables identify units and others are measurements corresponding to the unit.

- **Wide** form: columns for ID variables plus 1 column per measurement.
- **Narrow** form: columns for ID variables, plus 1 column identifying measurement, plus 1 column giving value.

Common to have data where some variables identify units and others are measurements corresponding to the unit.

- **Wide** form: columns for ID variables plus 1 column per measurement.
- **Narrow** form: columns for ID variables, plus 1 column identifying measurement, plus 1 column giving value.

Often want to convert from wide to narrow, or change what's ID and what's measure.

- `reshape` package introduced data-reshaping tools.
- `reshape2` package simplifies lots of common uses.
- `melt()` turns a wide dataframe into a narrow one.
- `dcast()` turns a narrow dataframe into a wide one.
- `acast()` turns a narrow dataframe into a wide array.

Example ¹

snoqualmie.csv has precipitation every day in Snoqualmie, WA for 36 years (1948–1983). One row per year, one column per day, units of 1/100 inch.

```
> snoq <- read.csv("snoqualmie.csv", header = FALSE,  
+                  as.is = TRUE)  
> colnames(snoq) <- 1:366  
> snoq$year      <- 1948:1983
```

^aFrom P. Guttorp, Stochastic Modeling of Scientific Data

Reshaping

```
> dim(snoq)
```

```
[1] 36 367
```

```
> snoq[1:3, 1:10]
```

	1	2	3	4	5	6	7	8	9	10
1	136	100	16	80	10	66	88	38	1	87
2	17	14	0	0	1	11	90	6	0	0
3	1	35	13	13	18	122	22	25	8	48

```
> snoq[1:3, 360:367]
```

	360	361	362	363	364	365	366	year
1	0	0	0	0	49	114	17	1948
2	47	245	121	72	27	41	NA	1949
3	4	40	10	5	93	23	NA	1950

Reshaping

Example

```
> #install.packages("reshape2")
> require(reshape2)
> snoq.melt <- melt(snoq, id.vars = "year",
+                  variable.name = "day",
+                  value.name = "precip")
> head(snoq.melt)
```

	year	day	precip
1	1948	1	136
2	1949	1	17
3	1950	1	1
4	1951	1	34
5	1952	1	0
6	1953	1	2

Reshaping

Example

```
> tail(snoq.melt)
```

	year	day	precip
13171	1978	366	NA
13172	1979	366	NA
13173	1980	366	80
13174	1981	366	NA
13175	1982	366	NA
13176	1983	366	NA

```
> dim(snoq.melt) # 36*366
```

```
[1] 13176      3
```

Reshaping

Example

Being sorted by day of the year and then by year is a bit odd

```
> snoq.melt.chron <- snoq.melt[order(snoq.melt$year,  
+                                   snoq.melt$day), ]  
> head(snoq.melt.chron)
```

	year	day	precip
1	1948	1	136
37	1948	2	100
73	1948	3	16
109	1948	4	80
145	1948	5	10
181	1948	6	66

Example

Most years have 365 days so some missing values:

```
> leap.days <- snoq.melt.chron$day == 366  
> sum(is.na(snoq.melt.chron$precip[leap.days]))
```

```
[1] 27
```


Example

Most years have 365 days so some missing values:

```
> leap.days <- snoq.melt.chron$day == 366  
> sum(is.na(snoq.melt.chron$precip[leap.days]))
```

```
| [1] 27
```

Tidy with `na.omit()`:

```
> snoq.melt.chron <- na.omit(snoq.melt.chron)
```

Reshaping

Example

`dcast()` turns back into wide form, with a formula of IDs ~ measures.

```
> snoq.recast <- dcast(snoq.melt, year ~ ...)
> dim(snoq.recast)
```

```
[1] 36 367
```

```
> snoq.recast[1:4, 1:15]
```

	year	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1948	136	100	16	80	10	66	88	38	1	87	8	4	0	0
2	1949	17	14	0	0	1	11	90	6	0	0	0	0	0	3
3	1950	1	35	13	13	18	122	22	25	8	48	3	2	8	0
4	1951	34	183	11	20	11	0	9	1	0	3	16	21	53	2

`acast()` casts into an array rather than a dataframe.

Example

- The formula could also specify multiple ID variables (including original measure variables), different measure variables (including original ID variables)...
- Also possible to apply functions to aggregates which all have the same IDs, select subsets of the data, etc.
- Strongly recommended reading:
 - Hadley Wickham, “Reshaping Data with the reshape Package”, *Journal of Statistical Software* 21 (2007): 12,
<http://www.jstatsoft.org/v21/i12>