Lecture 12: Split/Apply/Combine

STAT GR5206

Statistical Computing & Introduction to Data Science

Cynthia Rush
Columbia University

December 1, 2017

Course Notes

- Homework due December 11.
- Statistics Department Holiday Party next Friday. RSVP!
- Final Friday, December 15, 1:10pm - 4:00pm. (Location TBD.)

The Split/Apply/Combine Model

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.

General Strategy

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)
head(strikes, 3)
```

Check Yourself

Sometimes we want to split up the rows of a data frame or entries of a vector by levels of a factor.

`split()` by Levels of a Factor

`split(x, f = my.index)` splits a data frame or vector `x` according to levels of `my.index`

Tasks

- First, split the data by country using `split()` and call the output `strikes.split`.
- Using `strikes.split` and `sapply()`, compute the average unemployment rate for each country. What country has the highest average unemployment rate? The lowest?

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?

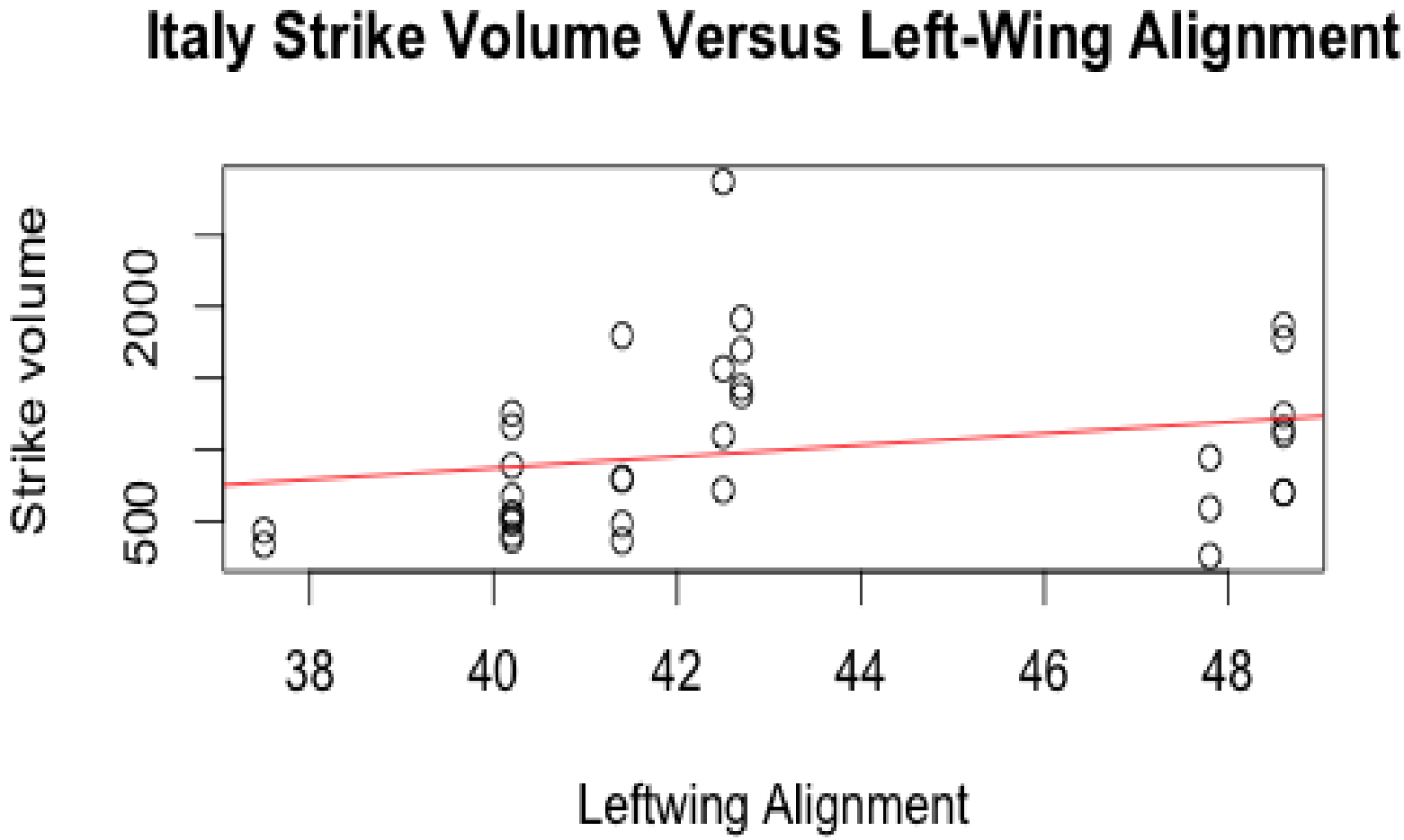
- Worst way: by hand, write 18 separate code blocks
- Bad way: explicit `for()` loop, where we loop over countries
- Best way: split appropriately, then use `sapply()`

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)
head(italy.strikes, 5)

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)
```

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"))
```

Example: Strikes dataset

We could `for()` loop it...

```
strike.coef  <- NULL
countries <- unique(strikes$country)

for (this.country in 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) <- countries
strike.coef
```

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)
```

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()`.

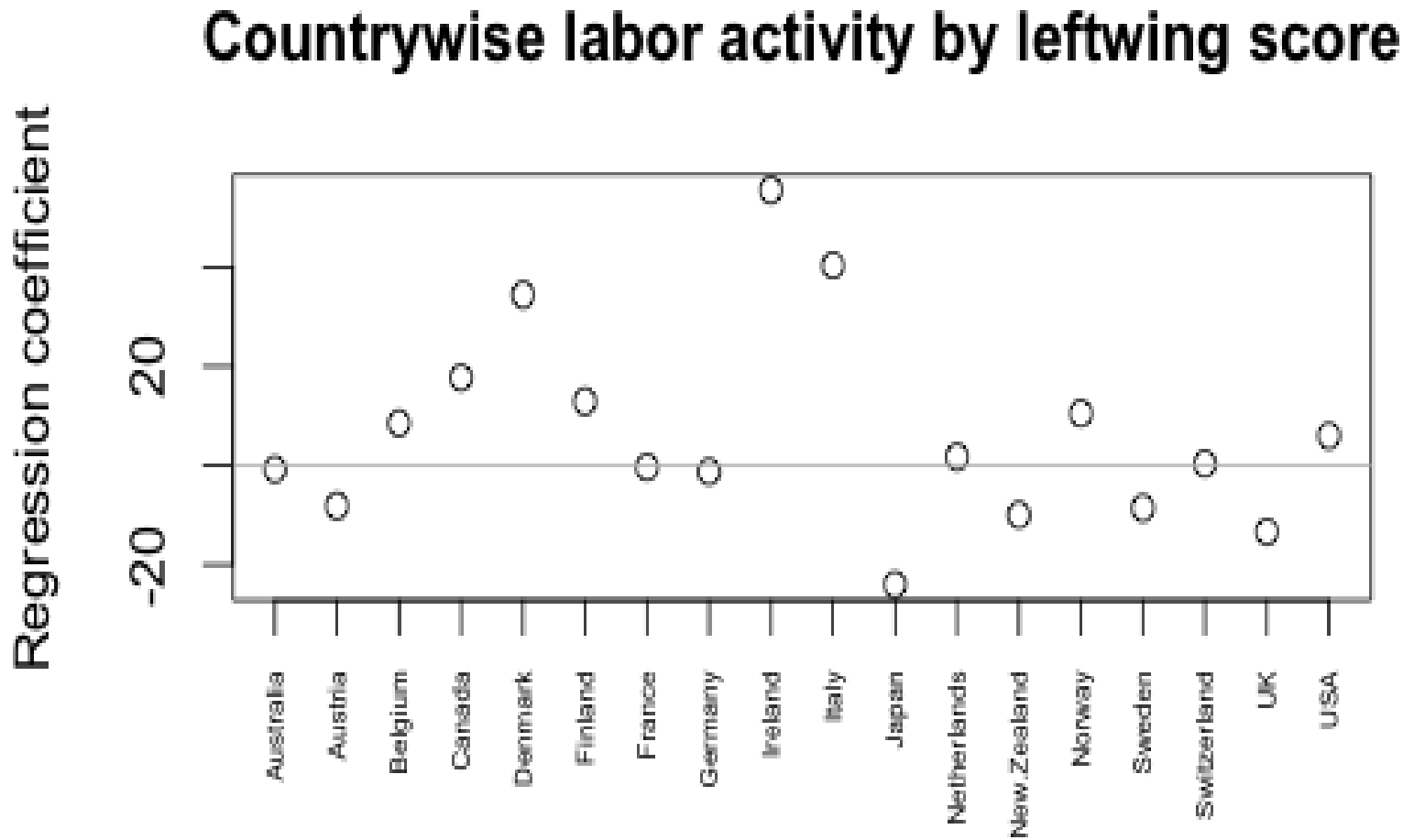
```
strike.coef <- sapply(strikes.split[1:12], my.strike.lm)
strike.coef
```

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 x 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.

Solution


Unemployment
Inflation
Rate
2
6
10
1950
1955
1960
1965
1970
1975
1980
1985
Year

Using plyr

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

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

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")

my.array
my.array[, , 3]

library(plyr)
aaply(my.array, 1, sum) # Get back an array
adply(my.array, 1, sum) # Get back a data frame
alply(my.array, 1, sum) # Get back a list

aaply(my.array, 2:3, sum) # Get back a 3 x 3 array
adply(my.array, 2:3, sum) # Get back a data frame
alply(my.array, 2:3, sum) # Get back a list
```

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

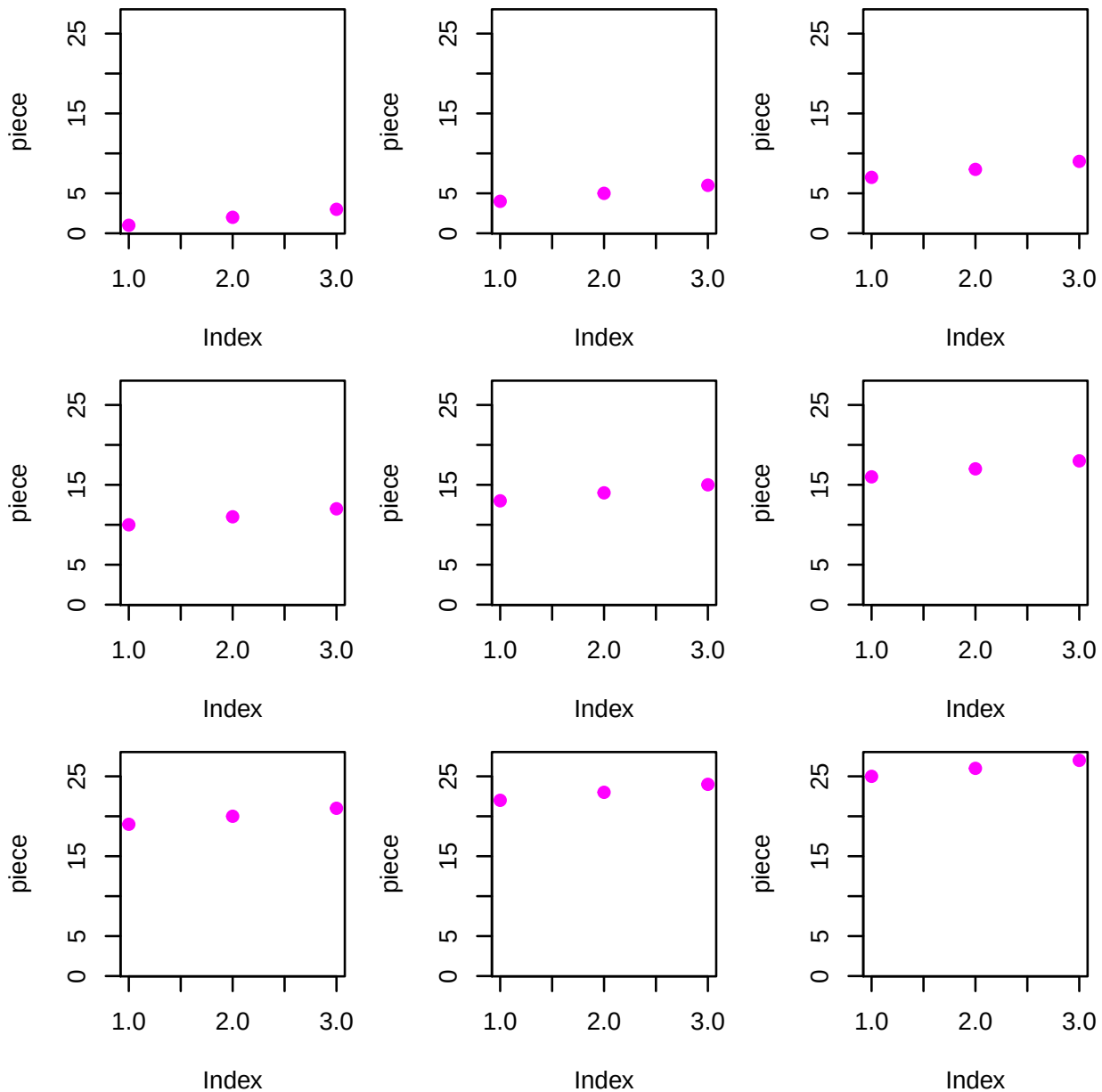
```
my.list <- list(nums = rnorm(1000), lets = letters,
                pops = state.x77[ ,"Population"])
head(my.list[[1]], 5)
head(my.list[[2]], 5)
head(my.list[[3]], 5)

laply(my.list, range) # Get back an array
ldply(my.list, range) # Get back a data frame
llply(my.list, range) # Get back a list

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

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

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)))
}

# 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]
```

Strikes Data Set, Revisited

```
# Getting regression coefficient separately for each
# country, new way, in three formats:

strike.coef.a <- daply(strikes, .(country), my.strike.lm)

# Get back an array, note the difference to sapply()

head(strike.coef.a)

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

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

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
head(strike.coef.75)
```

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
head(strike.coef.75)
```

Check Yourself

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.

Now to dplyr

- `dplyr` provides a set of tools for efficiently manipulating datasets in `R`.
- It is the next iteration of `plyr`, focusing only on dataframes.
- `dplyr` aims to provide a function for each basic data manipulation task.

1. `select()`: select variables based on their names.
2. `filter()`: select cases based on their values.
3. `arrange()`: reorder the cases.
4. `summarize()`: condense multiple values to a single value.
5. `mutate()`: add new variables that are functions of existing variables.
6. `sample_n()`: take random samples.

dplyr Example

Imagine we were only interested in some of the columns of the `strikes` dataset.

```
library(dplyr)

ycs <- select(strikes, year, country, strike.volume)
head(ycs)

# Same as
ycs <- strikes[, c("year", "country", "strike.volume")]
```

Pipe operator

- Power of `dplyr` is the pipe operator `%>%` that makes code more efficient and readable.
- Pipes allow the user to combine several functions.
- Pipes take the input on the left side of the `%>%` symbol and pass it in as the first argument to the function on the right side.

Pipe Example

```
ycs <- strikes %>% select(year, country, strike.volume)

# Maybe we're only interested in strikes where the unemployment
# is greater than 4 percent

ycs_unemploy <- strikes %>%
                filter(unemployment > 4) %>%
                select(year, country, strike.volume)
head(ycs_unemploy)
dim(ycs)
dim(ycs_unemploy)

# Note that the order matters!
```

Pipe Example

```
# group_by acts like split
# Most useful when combined with summarize

strikes.split   <- strikes %>% group_by(country)
country.strikes <- strikes %>%
                   group_by(country) %>%
                   summarize(avg.strikes = mean(strike.volume))
```

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 `daply()`:

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

versus

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

Reshaping Dataframes

Reshaping

Common to have data where some variables identify units, and others are measurements.

- **Wide** form: columns for ID variables plus 1 column per measurement.
  - Good for things like correlating measurements, or running regressions.
- **Narrow** form: columns for ID variables, plus 1 column identifying measurement, plus 1 column giving value.
  - Good for summarizing, subsetting.

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

Reshaping

- `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.

Reshaping: 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
snoq[1:3, 360:367]

#install.packages("reshape2")
require(reshape2)
snoq.melt <- melt(snoq, id.vars = "year", variable.name = "day",
                  value.name = "precip")
head(snoq.melt)
tail(snoq.melt)
dim(snoq.melt) # 36*366
```

[1] From P. Guttorp, Stochastic Modeling of Scientific Data

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)
```

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]))
```

Tidy with `na.omit()`:

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

Reshaping: Example

Today's precipitation vs. next day's:

```
short.chron <- snoq.melt.chron[-nrow(snoq.melt.chron), ]
precip.next <- snoq.melt.chron$precip[-1]
snoq.pairs  <- data.frame(short.chron, precip.next)
head(snoq.pairs)
```

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

```
snoq.recast <- dcast(snoq.melt, year ~ ...)
dim(snoq.recast)
snoq.recast[1:4, 1:15]
```

`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.
- Recommended reading if you want to use `reshape` package:
  - Hadley Wickham, "Reshaping Data with the reshape Package", *Journal of Statistical* Software 21 (2007): 12, `http://www.jstatsoft.org/v21/i12`