Lecture 16: More on Split-Apply-Combine

Statistical Computing, 36-350

Monday November 16, 2015

Outline

- Quick review: split-apply-combine and 'plyr'
- Parallelization
- High-level tips and themes
- Practice problem

Split-apply-combine: general strategy

When performing a repetitive task over a large data set, we can think about the following three general steps:

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

This can be especially helpful for **ragged data**, i.e., data where the dimensions are irregular

Split-apply-combine: how?

Two possible frameworks:

- `*apply()` functions in base R
- `**ply()` functions in `plyr` package
- For the latter functions, can 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)

Example: global unemployment rate over the years

Remember the strikes data set from the last couple of lectures:

```
strikes = read.csv("http://www.stat.cmu.edu/~ryantibs/statcomp/lectures/strikes.csv")
class(strikes)
```

```
## [1] "data.frame"
```

```
head(strikes)
```

```
##     country year strike.volume unemployment inflation left.parliament
## 1 Australia 1951           296          1.3      19.8            43.0
## 2 Australia 1952           397          2.2      17.2            43.0
## 3 Australia 1953           360          2.5       4.3            43.0
## 4 Australia 1954             3          1.7       0.7            47.0
## 5 Australia 1955           326          1.4       2.0            38.5
## 6 Australia 1956           352          1.8       6.3            38.5
##   centralization density
## 1      0.3748588      NA
## 2      0.3751829      NA
## 3      0.3745076      NA
## 4      0.3710170      NA
## 5      0.3752675      NA
## 6      0.3716072      NA
```

Question: what is the average global employment rate over the years?

With a for loop:

```
years = unique(strikes$year)
avg.rate.1 = numeric(length(years))
for (i in 1:length(years)) {
  strikes.subset = subset(strikes, year==years[i])
  avg.rate.1[i] = mean(strikes.subset$unemployment)
}
```

With base R:

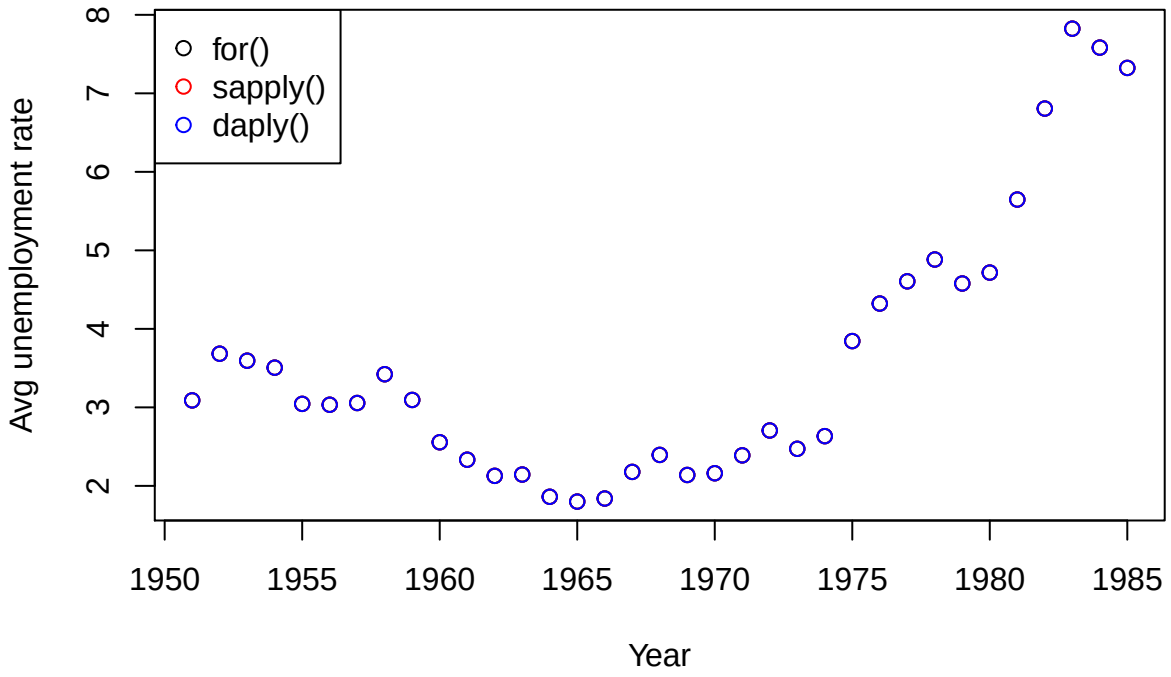
```
strikes.split = split(strikes, strikes$year)
avg.rate.2 = sapply(strikes.split, function(df) { mean(df$unemployment) })
```

With `plyr`:

```
library(plyr)
avg.rate.3 = daply(strikes, .(year), function(df) { mean(df$unemployment) })
```

They should all be the same

```
plot(years, avg.rate.1, xlab="Year", ylab="Avg unemployment rate")
points(years, avg.rate.2, col="red")
points(years, avg.rate.3, col="blue")
legend("topleft", legend=c("for()", "sapply()", "daply()"),
       col=c("black", "red", "blue"), pch=21)
```



Why do we like `plyr` functions?

Because they make it completely transparent what data type goes in, and what data type comes out

- `a*ply()` : input arrays, as in

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

- `d*ply()` : input data frames, as in

  ```
  processed = d*ply(.data, .(splitvariable), .fun, ...)
  ```

- `l*ply()` : input lists, as in

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

  (note: only one way to split a list!)

- for the `*` character, use `a,d,l` to output an array, data frame, or list, respectively; or use `_` to output nothing

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 specification 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`

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.,

```
avg.rate.3 = daply(strikes, .(year),
  function(df) {mean(df$unemployment)})
```

versus

```
avg.rate.4 = daply(strikes, .(year),
  function(df) {mean(df$unemployment)}, .parallel=TRUE)
```

High-level tips and themes

- Often the main work in split-apply-combine lies in the processing function, the one you write before you call `**ply()`
- Remember: its input should be a piece of the original data, and you should make it so that it produces output of a consistent format (if appropriate)
- You can try writing it and debugging it by working with a manually split-off piece of the original data set (before you call `*dply()`)

- Split-apply-combine and `plyr` are great ... but should we always use them?
- Don't use as simply a fancy way of writing `for()` loops, as in

  ```
  aaply(1:n, 1, function(i) {
    # hundreds of lines of code here
  })
  ```

- Rather, use when:
  - The problem naturally breaks up into smaller chunks
  - You can more or less easily write a function to solve the subproblem given a chunk
  - You want to reintegrate all the answers at the end

Practice problems

Question: what are the trends in the various countries' unemployment rates over time?

Specifically, our goal is to:

1. Compute a scaled version of unemployment rate for each country in the `strikes` data frame
2. Produce 35 plots (since there are 35 years total) displaying these rates

---

Setup: first, let's just reload the data and the plyr package and the maps package (for your convenience, so you can run this code block, and then work in the console)

```
strikes = read.csv("http://www.stat.cmu.edu/~ryantibs/statcomp/lectures/strikes.csv")
library(plyr)
library(maps)
```

Now here's are your tasks (go ahead and fill in the code below):

```
# 0. Enter your unique ID here:

# 1. Fill in the following function my.scale(), which, given some subset of the
# rows of strikes, returns a vector of scaled values of the unemployment rates,
# so that the minimum becomes 0 and the maximum becomes 1

my.scale = function(df) {
  min.val = min(df$unemployment)
  max.val =  max(df$unemployment)
  # TODO, fill in the right return values
}

# Check it by uncommenting the code below; the first 4 return values should be
# 0.00000000, 0.10588235, 0.14117647, 0.04705882

#my.scale(subset(strikes, country=="Australia"))
```

```
# 2. Now, use dlply(), to run my.scale() on each country in strikes, and get back a
# list of scaled unemployment rates

unemployment.scaled.list = 0 # TODO, use dlply()

# 3. Here, unravel the list, and append it as a column to strikes

strikes$unemployment.scaled = 0 # TODO, unlist unemployment.scaled.list
head(strikes)
```

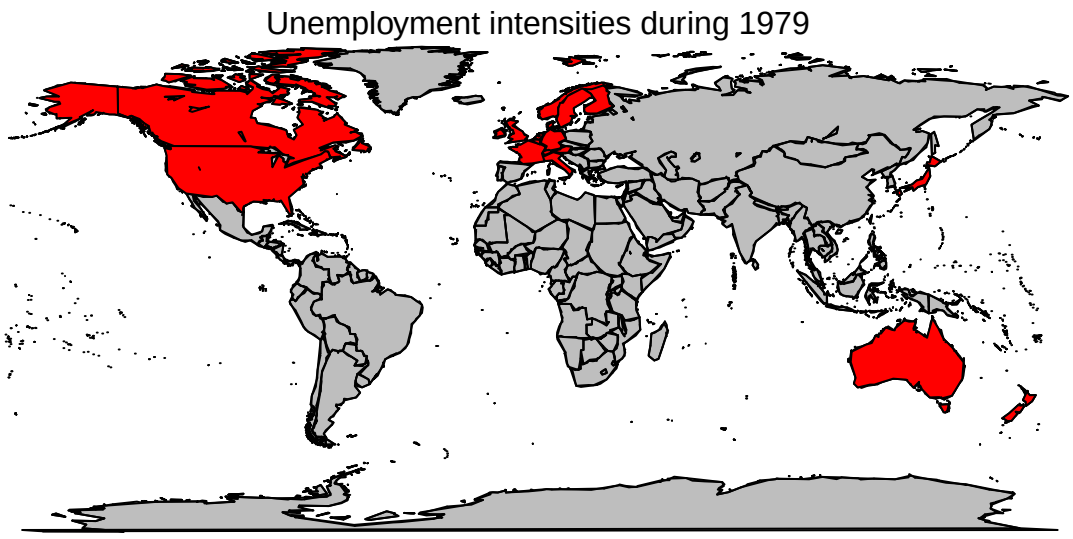
```
##     country year strike.volume unemployment inflation left.parliament
## 1 Australia 1951           296          1.3      19.8            43.0
## 2 Australia 1952           397          2.2      17.2            43.0
## 3 Australia 1953           360          2.5       4.3            43.0
## 4 Australia 1954             3          1.7       0.7            47.0
## 5 Australia 1955           326          1.4       2.0            38.5
## 6 Australia 1956           352          1.8       6.3            38.5
##   centralization density unemployment.scaled
## 1      0.3748588      NA                   0
## 2      0.3751829      NA                   0
## 3      0.3745076      NA                   0
## 4      0.3710170      NA                   0
## 5      0.3752675      NA                   0
## 6      0.3716072      NA                   0
```

```
# Here is a function that plots a map of the world, which, given some subset of
# the rows of strikes, plots a map of the world and colors in the scaled unemployment
# rates for the various countries

my.map = function(df) {
  my.colors = colorRamp(heat.colors(50))
  map("world", fill=TRUE, col="gray")
  map("world", add=TRUE)
  map("world", regions=df$country, add=TRUE, fill=TRUE,
      col=rgb(my.colors(df$unemployment.scaled),max=255))
  mtext(paste("Unemployment intensities during",unique(df$year)[1]))
}

# Test it out by first looking the map for the strikes data during the year 1979

my.map(subset(strikes, year==1979))
```



```r
# 4. Now, use d*ply() to create 35 plots, and save them in a PDF

pdf(file="scaled.unemployment.by.year.pdf", height=6, width=6)
# TODO, use one of the d*ply() functions to make 35 plots
graphics.off()

# Take a look at your PDF (with 35 pages to it!). Do you notice something about
# the global trend over time? Note that red is low, white is high ...
# rates ...
```

```r
# Bonus. Perhaps 35 plots is too much. Repeat these previous steps, but instead
# of looking at scaled unemployment rates each year, look at averages over 5 years.
# This will give you 7 final plots, instead of 35
```