R Data Wrangling

Lecture 12

Dirk Eddelbuettel

STAT 430: Data Science Programming Methods (Fall 2019)
Department of Statistics, University of Illinois

Data Wrangling

Overview

- In the last three lecture we covered
  - R data types
  - R control flow and functions
  - R data input/output
- So now let us learn how to slice + dice + aggregate some data
  - This lecture focuses on Base R and functions come with R
  - You can assume that these will always be at your disposal
- A solid grasp of basics will aid in using alternate approaches
  - data.table (next lecture)
  - dplyr (lecture after next)

Three key aspect we cover

- Alter / expand the data set by computing new columns
- Summarize / describe the data, also by taking simple subsets
- Conditional summaries based on some columns

Flights Data Set

```
> ## A large flights data set we can read in directly
> ## Also present in data-examples repo
> url <- paste0("https://github.com/Rdatatable/",
+               "data.table/blob/master/vignettes/",
+               "flights14.csv?raw=true")
> data <- read.csv(url)
> dim(data)
# [1] 253316     11
```

Define and assign

```
> ## using $ for columns
> data$tot_delay <- data$dep_delay + data$arr_delay
> ##
> ## equivalent
> data[, "tot_delay"] <- data[, "dep_delay"] +
+     data[, "arr_delay"]
```

Generally any *expression* can be assigned to a new column (which will be added), or equally to an existing column that will be overwritten.

Use `within`

Because the explicit enumeration as in `data[, ”dat_delay”]` etc is cumbersome, a more functional alternative exists:

```
> data <- within(data,
+                 tot_delay <- dep_delay + arr_delay)
```

ILLINOIS

Use `with`

Similar, `with` can give more "direct" access to the column identifiers:

```
> with(data, mean(distance))
# [1] 1099.445
> with(data, mean(distance)) == mean(data[,"distance"])
# [1] TRUE
```

Another with example

```
> ## nicer expression using with ...
> with(mtcars, mpg[cyl == 8  &  disp > 350])
# [1] 18.7 14.3 10.4 10.4 14.7 19.2 15.8
> ##
> ## ... than explictly referencing
> mtcars$mpg[mtcars$cyl == 8  &  mtcars$disp > 350]
# [1] 18.7 14.3 10.4 10.4 14.7 19.2 15.8
```

The `transform` function can be used as well:

```
> data <- transform(data,
+                   total_delay = arr_delay+dep_delay)
```

Note that we use `=` for assignment with in the scope of the `data` object here.

Useful to "sweep" a function across

- The `apply` function can operate on matrices and data.frames
- It takes three arguments:
    - the object
    - the direction: 1 for row-wise, 2 for column-wise
    - a function

Example

```r
> M <- matrix(1:9,3,3)
> M
#      [,1] [,2] [,3]
# [1,]    1    4    7
# [2,]    2    5    8
# [3,]    3    6    9
> apply(M, 1, sum)  # rows
# [1] 12 15 18
> apply(M, 2, sum)  # cols
# [1]  6 15 24
```

```r
> M <- matrix(1:9,3,3)
> D <- data.frame(M); D
#   X1 X2 X3
# 1  1  4  7
# 2  2  5  8
# 3  3  6  9
> apply(D, 1, sum)  # rows
# [1] 12 15 18
> apply(D, 2, sum)  # cols
# X1 X2 X3
#  6 15 24
```

Subsets

ILLINOIS

Index Expression

We can use any valid expression for the row indices

```
> mean(data[data$origin=="JFK" &
+            data$dest=="LAX", "tot_delay"])
# [1] 11.66105
```

This works because the expression `data$origin=="JFK" & data$dest=="LAX"` returns a logical vector of the same size as `data` so it can be used for indexing.

(But let's recall that purely numerical indexing also work.)

We can save one layer of $ subsetting via `with`:

```
> with(data, mean(data[origin=="JFK" &
+                      dest=="LAX", "tot_delay"]))
# [1] 11.66105
```

Summaries By Group

Using factor variables provides logical grouping, especially when the cardinality of the factor variable is low. *I.e.* here we would rather condition on `origin` with its three values than `dest` which has 109.

`aggregate`

In its simplest form, `aggregate` takes a vector argument, followed by a list of conditioning variable, and a function:

```
> aggregate(data$arr_delay,
+           list(origin=data$origin), mean)
#   origin         x
# 1    EWR 10.026121
# 2    JFK  7.731465
# 3    LGA  6.601968
```

aggregate

The `aggregate` function also has a more powerful formula interface

```
> ## average arrival delay by airport
> aggregate(arr_delay ~ origin, data, mean)
#   origin arr_delay
# 1    EWR 10.026121
# 2    JFK  7.731465
# 3    LGA  6.601968
```

Use `cbind` to select several columns for multiple summaries

```
> aggregate(cbind(arr_delay, dep_delay) ~ origin,
+           data, mean)
#   origin arr_delay dep_delay
# 1    EWR 10.026121  15.21248
# 2    JFK  7.731465  11.44617
# 3    LGA  6.601968  10.60500
```

The right-hand side of the formula can expands to provide multiple groups:

```
> head(aggregate(tot_delay ~ origin + carrier,
+                data, mean), 7)
#   origin carrier tot_delay
# 1    EWR      AA 24.597206
# 2    JFK      AA 15.938271
# 3    LGA      AA  9.578602
# 4    EWR      AS  4.942509
# 5    EWR      B6 18.480906
# 6    JFK      B6 22.785535
# 7    LGA      B6 22.019850
```

Summary of Formula notation

term	example	description
~	y ~ x	model y as a function of x
+	y ~ a + b	include columns a and b
-	y ~ a - b	include a but not b
:	y ~ a : B	estimate interaction of a and b
*	y ~ a * b	include a and b and their interaction
|	y ~ a | b	estimate y as function of a conditional on b

This will become more relevant when doing modeling.

Source: Table 13-3 in de Vries and Meys (2012)

tapply

Similar to `aggregate`

```
> tapply(data$tot_delay, data$origin, mean)
#      EWR      JFK      LGA
# 25.23860 19.17763 17.20697
```

tapply: Multiple indices

```
> head(tapply(data$tot_delay,
+              list(data$carrier, data$origin), mean), 8)
#          EWR      JFK       LGA
# AA 24.597206 15.93827  9.578602
# AS  4.942509       NA        NA
# B6 18.480906 22.78553 22.019850
# DL 25.699254 16.51813 16.366738
# EV 31.597748 36.90833 27.939167
# F9        NA       NA 51.321353
# FL        NA       NA 34.269384
# HA        NA 20.93846        NA
```

Note how `tapply` preserves all cells in the table.

by: aggregate by factor

```
> by(data$arr_delay, data$origin, summary)
# data$origin: EWR
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#  -71.00  -14.00   -3.00   10.03   17.00 1494.00
# --------------------------------------------------------
# data$origin: JFK
#     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
#  -79.000  -15.000   -3.000    7.731   14.000 1223.000
# --------------------------------------------------------
# data$origin: LGA
#     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
# -112.000  -16.000   -5.000    6.602   14.000  996.000
```

The lapply/sapply/mapply/... Functions

Loop over list

- `lapply` is very powerful and useful
    - it takes a list,
    - then applies then given function to each element
    - and returns a list of results
- *internally* a data.frame *is* a list so we can use it
- `lapply` is very powerful and useful
- `do.call()` can then be used to compact the results

Simple example

```
> set.seed(123) # make it reproducible
> alist <- list(A=matrix(1:9,3),B=seq(20,30),C=rnorm(5))
> lapply(alist, sum)
# $A
# [1] 45
#
# $B
# [1] 275
#
# $C
# [1] 0.9678513
```

A simplified version

- `sapply` is like `lapply`
- but returns a simpler object, where possible
- if the operation reduces to a number we get a vector back

Simple example

```
> set.seed(123) # make it reproducible
> alist <- list(A=matrix(1:9,3),B=seq(20,30),C=rnorm(5))
> sapply(alist, sum)
#           A           B           C
#  45.0000000 275.0000000   0.9678513
```

A multivariate `sapply`

- it sweeps the function over the *i*-th elements of *multiple* lists
- can often be used where a loop might be needed

```
> mapply(paste0, LETTERS[1:5], letters[1:5])
#    A    B    C    D    E
# "Aa" "Bb" "Cc" "Dd" "Ee"
```

Another data.frame operation

- The `merge()` function combines data.frame objects
- There is a fairly close connection to SQL's `JOIN` command
- Try the examples on the right
- See `help(merge)` for more

```r
A <- data.frame(x=c("A","C","E"),
                y=c(1,3,5))
B <- data.frame(x=c("B", "C", "D"),
                y=c(20,30,40))
merge(A, B, by="x")                # inner
merge(A, B, by="x", all=TRUE)      # outer
merge(A, B, by="x", all.x=TRUE)   # left
merge(A, B, by="x", all.y=TRUE)   # right
```

Map, Reduce, Filter

- R also has a set of function supporting a functional approach
- `Map(f, x)` applies binary `f()` to each element of `x`
  - this returns a list
  - try *.e.g.* `Map(sqrt, 1:5)`
- `Reduce(f, x)` applies `f(x,y)` over successive `x` elements
  - this (typically) returns a scalar
  - try *e.g.* `Reduce(sum, 1:5)`
- `Filter(f, x)` select elements where the predicate is true
- and more – we won't be using (or testing) these
- but it is useful to know about them

Data Wrangling

We have seen how

- to alter a `data.frame` by adding to it
- to operate on explicit subsets via indexing
- to operate on implicit groupings from factor
- to use some of the `*apply` functions
- to combine `data.frame` object via `merge`
- to use functional programming over lists and data.frame objects

These commands are versatile but at times a little cumbersome and idiosyncratic. They are however universal as part of R.

We will see alternatives for basic wrangling in the next two lectures.