Data Wrangling with R

Stefan Zohren

19 May 2016

Content of this lecture

- Getting started with R
  - Installation and Packages
  - Basic commands
  - Vectors, matrices and lists
  - Data frames
  - Basic programming
- Importing, exporting and manipulating data with R
- Packages for advanced data wrangling
  - Manipulating data with `reshape2`
  - Manipulating data with `dplyr`
  - From data frames to `data.table`
  - Interacting with databases using `SQLite` and `dplyr`
- Summary

Calculating with R

You can use R interactively as a calculator, i.e.

```
3+sqrt(2)
```

```
## [1] 4.414214
```

returns the value of $3 + \sqrt{2}$. If you are not sure what a function does type `?` for help, i.e. `?sqrt`.

We can assign values to variables using the assignment operator `<-`

```
x <- 3
y <- 4.5
x+y
```

```
## [1] 7.5
```

Objects and types

A useful command to examine objects is the `str` (structure) command

```
str(x)
```

```
##  num 3
```

We see that R by default made x a numeric (double) variable. We could change it to be of type integer by invoking the command

```
x<-as.integer(x)
str(x)
```

```
##  int 3
```

Vectors

Create a vector:

```
vec<-c(25,28,1.85,1.70,90,75)
```

Let us again look at the structure

```
str(vec)
```

```
##  num [1:6] 25 28 1.85 1.7 90 75
```

It is a numeric (double) vector with entries indexed by 1,..,6.
Observe that, as opposed to C/C++ or Python, **indices start at 1.**

Vectors: Indexing and slicing

To get for example the first entry, use `vec[1]`. We can also slice vectors:

```
vec[3:5]
```

```
## [1]  1.85  1.70 90.00
```

To return all but a given entry:

```
vec[-1]
```

```
## [1] 28.00  1.85  1.70 90.00 75.00
```

Matrices

We can create a matrix from our vector

```
M<-matrix(vec,ncol=3)
```

```
##      [,1] [,2] [,3]
## [1,]   25 1.85   90
## [2,]   28 1.70   75
```

Observe the indecing: `M[row(s),column(s)]`. Matrices can also be sliced, similar to vectors.

Lists

A `list` is similar to a vector, but allows to store arbitrary objects in it:

```
l <- list(v=vec,m=M)
l
```

```
## $v
## [1] 25.00 28.00  1.85  1.70 90.00 75.00
##
## $m
##      [,1] [,2] [,3]
## [1,]   25 1.85   90
## [2,]   28 1.70   75
```

Here each element of the list has a name, in our case `v` and `m` and we can use those names to access the elements, e.g. `l$v`.

Data frame: basics

We can take our matrix from the previous slides, add column names and transform it into a data frame:

```
colnames(M)=c("age","height","weight")
data<-as.data.frame(M)
data
```

```
##   age height weight
## 1  25   1.85     90
## 2  28   1.70     75
```

Data frame: adding and manipulating

We can add a new column:

```
data$weightOheight <- data$weight/data$height
data
```

```
##   age height weight weightOheight
## 1  25   1.85     90       48.64865
## 2  28   1.70     75       44.11765
```

A very useful function is apply:

```
apply(data,2,mean)
```

```
##           age        height        weight weightOheight
##      26.50000       1.77500      82.50000      46.38315
```

Basic programming

You can write your own functions and loops etc. Here one example:

```
my.mean <- function(vector){
  n=length(vector)
  sum=0
  for(i in 1:n){
    sum = sum+vector[i]
  }
  sum/n  # note: no need for 'return'
}
```

For example `my.mean(1:6)` gives `3.5`. A major application is to use your function in `apply` similar to what we discussed above, i.e. `apply(data,2,my.mean)`.

Importing and exporting data

Probably the easiest way of importing and exporting data in R is using the `read.csv` and `write.csv` commands. This is done for example

```
flights <- read.csv("flights14.csv")
```

There are many options, type `?read.csv` for help.

To write files use `write.csv`.

You can also import and export Excel files directly using the `xlsx` package, but it is generally better to use the `.csv` files as an universal exchange medium.

Inspecting data

To inspect newly imported data use

```
head(flights)
tail(flights)
```

to see that first and last 6 rows,

```
dim(flights)
```

for the dimensions, as well as `nrow` and `ncol`, and

```
str(flights)
```

for the structure.

Subsetting of data

We can easily subset and query the data. For example, the following command gives the total number of flights by American Airlines:

```
sum(flights$carrier == 'AA')
```

```
## [1] 26302
```

We can select only those rows from the data frame and save it into a new data frame called `flights.AA`:

```
flights.AA <- flights[flights$carrier == 'AA',]
```

Computing on rows

Using R we can effectively compute on rows and columns. For example the mean departure delay of a all flights (rows) is given by

```
mean(flights$dep_delay)
```

```
## [1] 12.46526
```

Computing on columns

We can also compute on columns and furthermore create new columns which are dynamically obtained from computations on other columns. For example, we can calculate the `gain = arr_delay - dep_delay` as a difference on arrival delay minus departure delay and add this information as a new column to the data frame

```
flights$gain <- flights$arr_delay - flights$dep_delay
head(flights)
```

Packages for advanced data wrangling

We now discuss packages for advanced data wrangling:

- Manipulating data with `reshape2`
- Manipulating data with `dplyr`
- From data frames to `data.table`
- Interacting with databases using `SQLite` and `dplyr`

Make sure you install all packages used.

Manipulating data with `reshape2`

One of the main functionalities of the package `reshape2` is the capability to transform data frames from a wide format to a long format and vice-versa.

The main functions of the package are:

- `melt`
- `cast` (became `dcast` and `acast` in `reshape2`)

See:

- Reshaping data with the reshape package, by H. Wickham, in J. Stat. Software (2007).

Loading the package and example data

We start by loading the package and a funny data set on chips which comes with it

```
library(reshape2)
data(french_fries)
head(french_fries)
```

```
##    time treatment subject rep potato buttery grassy ranc
## 61    1         1       3   1    2.9     0.0    0.0    0
## 25    1         1       3   2   14.0     0.0    0.0    1
## 62    1         1      10   1   11.0     6.4    0.0    0
## 26    1         1      10   2    9.9     5.9    2.9    2
## 63    1         1      15   1    1.2     0.1    0.0    1
## 27    1         1      15   2    8.8     3.0    3.6    1
```

Melting: Transforming data from wide to long

The first step in transforming data using the `reshape2` package is to 'melt' the data from a wide to a long format:

```
chips.m <- melt(french_fries, id = 1:4)
head(chips.m)
```

```
##    time treatment subject rep variable value
## 1     1         1       3   1   potato   2.9
## 2     1         1       3   2   potato  14.0
## 3     1         1      10   1   potato  11.0
## 4     1         1      10   2   potato   9.9
## 5     1         1      15   1   potato   1.2
## 6     1         1      15   2   potato   8.8
```

We see that we kept the first four columns, as we specified by `id = 1:4`. However the remaining variables are now specified in two columns `variable` and `values`.

Casting: Transforming data from long to wide

Having molten the data, we can now cast it back into wide format using different prescriptions. This is done using the `dcast` function. We can use this to calculate for example the mean for each variable given a treatment:

```
chips.c <- dcast(chips.m, treatment ~ variable,
                 mean, na.rm=TRUE)
chips.c[,1:5]
```

```
##   treatment   potato  buttery    grassy   rancid
## 1         1 6.887931 1.780087 0.6491379 4.065517
## 2         2 7.001724 1.973913 0.6629310 3.624569
## 3         3 6.967965 1.717749 0.6805195 3.866667
```

Here `treatment ~ variable` means `treatment` as a function of `variable`.

Manipulating data with `dplyr`

We now look at more modern packages for data manipulation starting with the package `dplyr`.

The main functions of the package are:

- `filter`
- `arrange`
- `select`
- `mutate`
- `summarise`
- Pipes from `magrittr`

Loading the package and sample data

We load the package:

```
library(dplyr)
```

as well as the flights data used earlier:

```
flights <- read.csv("flights14.csv")
dim(flights)
```

```
## [1] 253316     17
```

We see it has 17 columns and 253316 observations.

Tabular data frames

`dplyr` has a wrapper to the conventional `data.frame` which allows for easy printing

Let us convert the conventional data frame into a tabular data frame:

```
flights <- tbl_df(flights)
```

It is now of class

```
class(flights)
```

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

You can easily print it by typing

```
flights
```

Using the `filter` functionality

The function `filter` the function `filter` provides a fast and compact way to filter the data. For example, we can quickly filter for all flights with tail number N619AA:

```
flights.N327AA <- filter(flights,tailnum == 'N327AA')
dim(flights.N327AA)
```

```
## [1]  7 17
```

The are a total of 7 such flights.

Using the arrange functionality

The function arrange can be used to arrange the data frame with respect to certain columns. We can for example arrange it with respect to departure delay, firstly in ascending order (default)

```
arrange(flights,dep_delay)
```

as well as descending order, using desc,

```
arrange(flights,desc(dep_delay))
```

Using the `select` functionality

Another useful functionality is the possibility to select certain columns of the data frame. This is done using the select function. For example, for all flights with number N327AA let us only keep a data frame with year, month, date and departure time

```
select(flights.N327AA, year, month, day, dep_time)
```

```
## Source: local data frame [7 x 4]
##
##    year month   day dep_time
##   (int) (int) (int)    (int)
## 1  2014     1     1     1902
## 2  2014     1     2     1649
## 3  2014     1     4     1005
## 4  2014     1     5     2006
## 5  2014     1     5      738
## 6  2014     1     6     1859
## 7  2014     1     7     1344
```

Using the mutate functionality

We can create new columns and add them to the data frame using mutate. This is similar to using apply on a data frame. For example, we can add the speed as another column to the data

```
flights.speed <- mutate(flights,
        speed = distance / air_time * 60)
head(select(flights.speed, speed))
```

```
## Source: local data frame [6 x 1]
##
##      speed
##      (dbl)
## 1 413.6490
## 2 409.0909
## 3 423.0769
## 4 395.5414
## 5 424.2857
## 6 434.3363
```

Using the summarise functionality

The function summarise can be used to obtain summaries of the data, as for example the mean of certain expressions.

```
summarise(flights.speed ,
  avspeed = mean(speed, na.rm = TRUE))
```

```
## Source: local data frame [1 x 1]
##
##    avspeed
##      (dbl)
## 1 399.3063
```

Using pipes from magrittr

If you are a unix/linux user, you will most probably be familiar with the **pipe** operator >. The pipe operator can be used on the shell to pass the output of one command over as the input of another command. The package magrittr introduces a similar functionality in R using the operator %>%.

For example, instead of

```
flights.1 <- select(flights, distance, air_time)
flights.2 <- mutate(flights.1,
                    speed = distance / air_time * 60)
flights.3 <- arrange(flights.2 , desc(speed))
```

...

Using pipes from magrittr

. . . we can write compactly

```
library(magrittr)
flights.3 <- select(flights, distance, air_time) %>%
             mutate(speed = distance / air_time * 60) %>%
             arrange(desc(speed))
```

using the pipe operator %>% in magrittr.

From data frames to `data.table`

The advantage of a `data.table` is that it is implemented entirely in C++ offering highly optimised querying functionalities.

We discuss the following functionalities of the `data.table` package:

- Subsetting on rows
- Ordering
- Subsetting on columns
- Computing on columns and aggregating results
- Data table with dplyr and magrittr

See Introduction to data.table in the package vignette.

Loading the package and example data

The package is loaded as follows:

```
library(data.table)
```

To illustrate the packages functionalities we use the same data set as we used in the previous section. We can easily convert a `data.frame` into a `data.table` using the command `data.table()`

```
flights <- data.table(flights)
```

Alternatively, we can import directly into a `data.table` using

```
flights <- fread("flights14.csv")
```

Subsetting on rows

Data tables have similar subsetting functionalities as `dplyr` offers using the filer command. However, the data table follows a syntax which is closer to the subsetting of rows in data frames, but is a more advanced SQL-style syntax. For example, to filter for flights with number N327AA we simple write:

```
flights.N327AA <- flights[tailnum == 'N327AA']
```

Type `head(flights.N327AA)` or `View(flights.N327AA)` to insect it.

Ordering

Similar to the `arrange` function in `dplyr` we can use the order function when subsetting:

```
flights[order(dep_delay)]
```

or in descending order

```
flights[order(-dep_delay)]
```

Subsetting on columns

If we want to subset on columns as we did in `dplyr` using the `select` command, we can do this on data tables using another subsetting syntax, namely the `.()` command:

```
flights.N327AA[ ,.(year, month, day, dep_time)]
```

```
##     year month day dep_time
## 1: 2014     1   1     1902
## 2: 2014     1   2     1649
## 3: 2014     1   4     1005
## 4: 2014     1   5     2006
## 5: 2014     1   5      738
## 6: 2014     1   6     1859
## 7: 2014     1   7     1344
```

Here `.(year, month, day, dep_time)` selects the columns `year`, `month`, `day` and `dep_time`.

Computing on columns and aggregating results

You can do computation on the data and also summarise those computations using the `.()` syntax of the `data.table` package. For example, similar to `mutate` in `dplyr`, new columns can be introduced as follows

```
flights[ ,.(speed = distance / air_time * 60)]
```

With nearly the same syntax we can also summarise:

```
flights[ ,.(mean(distance / air_time * 60,na.rm=TRUE))]
```

```
##           V1
## 1: 399.3063
```

Data table with dplyr and magrittr

Note that you can also use `dplyr` function as well as pipes to work on data tables. For example, we could run the exact command we used previously on a data frame with `dplyr`, but now on a data table:

```
select(flights, distance, air_time) %>%
    mutate(speed = distance / air_time * 60) %>%
    arrange(desc(speed))
```

This provides both the great functionalities and simple syntax from `dplyr` as well as the speed advantage of `data.table`.

Interacting with databases using `SQLite` and `dplyr`

We have already seen how to use `dplyr` to quickly query and manipulate data frames. Furthermore, we have seen that the same functionalities can also be used on data tables. For those working with databases it will probably be nice to hear that the same is also true for databases. Combining `dplyr` with the `SQLite` package you can operate on SQL databases with the same commands as you used before on data frames and data tables (see Databases)

Comment on web-based formats using `XML` and `jsonlite`

Apart from databases we can also use `dplyr` and `data.table` in combination with web-based formats such as `xml` or `json`. Have a look at the following packages:

- `XML`
- `jsonlite`

From what you have learned in this course it is straightforward to extend our workflow to include `xml` and `json` files.

- For example, you can download an entire image of `wikipedia` in `xml` and query it for joint occurrences of certain words.
- Many other online services also offer APIs where you can download data in `json` format, such as google map data, twitter data, weather data etc.

Summary

- This course is about how to efficiently handle portentially large and messy data sets in `R`
- We revised basic objects in `R`, like vectors, matrices, lists and data frames
- Basic functionalities in `R` for importing, exporting and manipulation data were discussed
- We reviewed more advanced packages for data wrangling in `R` including:
  - `reshape2` for melting and casting data
  - `dplyr` for advanced subsetting and querying of data
  - `data.table` for fast and efficient data containers
  - `magrittr` for using pipes in workflows
- We also briefly commented on how to go from in-memory data to databases and other web-based formats