数理解析研究所講究録
第2067巻 2018年 125-131

# Introduction to R and Time Series

PREINING Norbert
アクセリア株式会社、東京都
Vienna University of Technology, Vienna, Austria
北陸先端科学技術大学院大学、能美市

**Abstract**

We give a short introduction to R, and discuss how to deal with time series based data using R by providing examples from network data analysis.

## 1 Introduction

In the plethora of programming languages available, R[1] takes somehow a strange place. Written as replacement for the commercial S-Plus implementation of the S programming language, it has superseeded its origin and nowadays can be considered the most widely used statistical programming language.

Similar to well know languages like Python, R exhibits its functionality in a REPL, a Read-Eval-Print-Loop, but can also be programmed in scripts, a scripting language.

The R environment consists of the main interpreter, and is enriched with a huge (currently 11985) number of additional packages for practically every (computational) task. Although packages related to statistics are strong in numbers, there are others for different areas, e.g., machine learning. Statistics is often tied to proper presentation, and thus support for advanced graphics generation is provide both by core R as well as many addon packages.

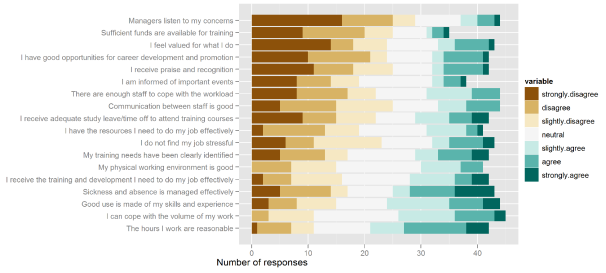


Figure 1: Example of visualizing questionnaires
`https://4dpiecharts.com/2010/09/25/visualising-questionnaires/`

R was conceived as free and open source replacement of S-Plus, and has kept this status, with a very active user and developer community improving and advancing the state of R. The recent boom in machine learning and in particular neural networks has

[1]`http://www.r-project.org/`

brought R with its highly parallelized architecture and optimized computations somehow into a bigger spotlight and makes R one of the top requested language skills in data science.

To download R, head over to the web site of the R Project where precompiled binaries for Windows, Mac OS and several Unix/Linux variants are available. For additional packages the Comprehensive R Archive Network (CRAN) contains several thousands which can be installed directly from the interpreter.

## 2 First steps with R

### 2.1 Starting and quitting

Starting and quitting the interpreter is usually done by calling the command `R`:

```
$ R
R version 3.4.1 (2017-06-30) -- "Single Candle"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)
...
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> q()
Save workspace image? [y/n/c]: n
$
```

and the builtin help system can be accessed using `?` and `??`, the former giving documentation for a function or item (usage: `?<item>`). The later one *searches* for a keyword in the list of all documentation files.

### 2.2 Numbers and Vectors

In R there are basically no simple numbers, everything is automatically converted to vectors of length 1. In practice there is no change in usage but it can be easily be seen by assigning a value and then printing and computing the length of it:

```
> a = 42; a
[1] 42
> length(a)
[1] 1
> a = c(4, 2) ; a
[1] 4 2
> length(a)
[1] 2
```

Having everything in vectors means that all operators work on vectors. Basic mathematical operators work component-wise, that is element per element:

```
> a = c(9, 3)
> b = c(-2, 4)
> a + b
[1] 7 7
> a * b
[1] -18 12
```

But what happens if the vectors do not agree in their length? In the case the lengths of the vectors are multiples of each other, the shorter one is reused (recycled):

```
> a = c(9, 3, 7, 6)
> b = c(-2, 4)
> a + b
[1] 7 7 5 10
> a * b
[1] -18 12 -14 24
```

On the other hand, if the lengths of the vectors are not multiples of each other, warnings are issued and computation interrupted:

```
> a = c(9, 3, 7)
> b = c(-2, 4)
> a + b
Warning message:
In a + b : longer object length is not a multiple of shorter object length
```

## 2.3 User defined functions

Like most programming language, R also provides facilities to write one's own functions. The notion to define a function might look funny to users of more traditional programming languages, but exhibits another nice feature – functions are first class objects and can be assigned, reassigned, passed around, simply treated like every other variable:

```
> myfunc = function(a, b) {
+   a*b/2
+ }
> myfunc(4,8)
[1] 16
```

Users of functional programming languages will feel immediately at home seeing this lambda-abstraction.

Let us now turn our attention to arguments of functions: Without going into details we only want to mention that functions can have normal arguments as well as optional, named, and default arguments. This helps immensely as many of the functions provided by add-on packages have into the tens and more of possible arguments, but in most cases passing only one or two does suffice completely.

Just do give you an example of what first-class objects means, here is some sample code using the function from above:

```
> doit = function(f, c) {
+   f(c,c)
+ }
> doit(myfunc, 9)
```

which will result in the answer `[1] 40.5`.

## 2.4 Other types of objects

Coming from the background of statistical data analysis, R provides a variety of different data types, *matrices* and multi-dimensional *arrays* (sometimes referred to as *tensors*) are generalizations of vectors to multiple indices. *Factors* used for treating of categorical data, that is data with a fixed set of values (like continent names). *Lists* are often used as the work-horse of R as they can contain elements of different types, in contrast to arrays and matrices which only contain the same data type. Another very useful, in particular for data science, data type is *data frame*. These data frames are like matrices but can contain different types of objects.

There are many more types available, we will see a few later on.

# 3 Time series with R

Time series are quite common – repeatedly taken measurements make up time series. Typical examples are the temperature in a room, rain amount per day, average grade of students per year, birth rate by year, just to name a few.

At our company we are using netflow data captured from routers with about 1200 features per time slot and about > 100000 data points (steadily increasing). Dealing with this amount of data in a not-vectorized environment would mean lots of waiting time during computations.

## 3.1 Objects for time series

R provides several ways to represent time series as objects:

- builtin types: `data.frame`, `matrix`, `vector`, `ts`
- `zoo`: provides a representation for indexed totally ordered observations which includes irregular time series (load with `library(zoo)`).
- `xts`: an extension of zoo (different time-based indexing, user-added properties, ...) (load with `library(xts)`).

Each of these objects has their own advantage and disadvantage: I used plain matrices for faster computation and index look-up, but converted it at the end to `xts` for better indexing, searching, slicing, and graphic displaying.

## 3.2 Reading data

Input for statistical computations often comes as CSV – comma separated values. The following example provides a slice of the data we are actually processing:

```
timeslot,ibytes
1472656200,113096548352
1472656800,97321410560
1472657400,88503681024
...
```

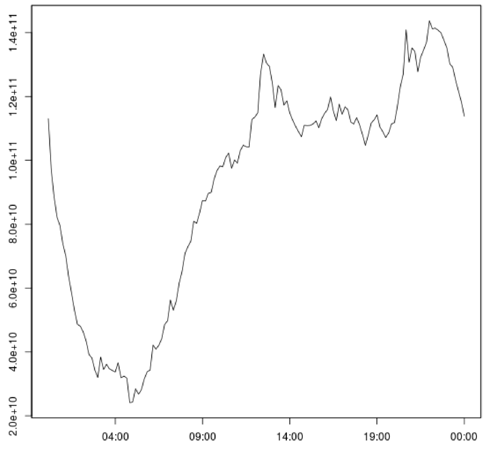
The columns are *timestamp*, the number of seconds from epoch (1970-1-1), a very common way to identify a time, *ibytes* the number of incoming bytes in the last time slot. To keep things simple we only exhibit two columns here, in reality there are more than 1200 columns with a variety of data aggregated.

Reading this kind of data into R is straight-forward, in particular when using `read.zoo` to read directly into a `zoo` object:

```
> flowdata = read.zoo('data1.csv', sep=",", FUN=function(i) strptime(i,'%s↩
  ↪'), header = TRUE)
> flowdata
2016-09-01 00:10:00 2016-09-01 00:20:00 2016-09-01 00:30:00
       113096548352         97321410560         88503681024
2016-09-01 00:50:00 2016-09-01 01:00:00 2016-09-01 01:10:00
        79659446272         73839611904         70095294464
...
```

Some explanation to the function call:

- first argument `'data1.csv'` gives the file to be read
- `sep=","` defines the field separator
- `header=TRUE` says that the first line should be ignored, because it contains the header
- `FUN=...` this part parses the number (timestamp) in the first field into a proper time object



After having imported the data in this way, a simple call to `plot(flowdata)` will give you a nice plot as shown on the right.

Of course plotting axes, sizes, color, output format and many more options can be adjusted to achieve the preferred output.

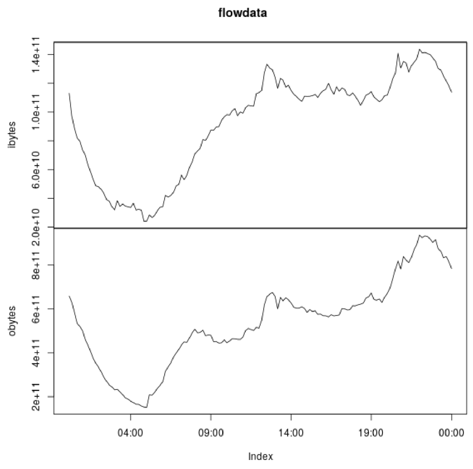
## 3.3 Multiple time series

As I mentioned above, the actual data we are processing contains many different columns (features). Let us assume for now only two more columns after the timeslot:

```
timeslot,ibytes
1472656200,113096548352,658813333504
1472656800,97321410560,630560137216
1472657400,88503681024,582952984576
...
```

Here the columns are as before, with the addition of the obytes column counting the number of outgoing bytes in the last time slot.

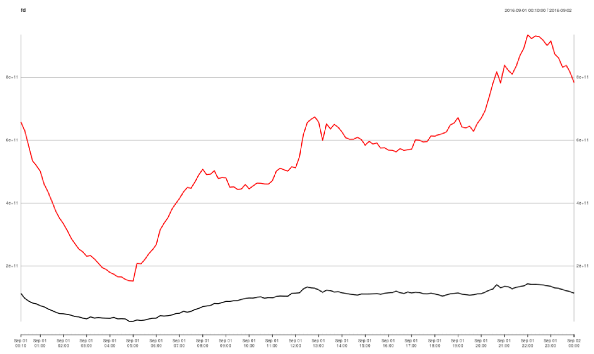
We can read the file in and plot the resulting date in the very same way as before, obtaining the following plot:



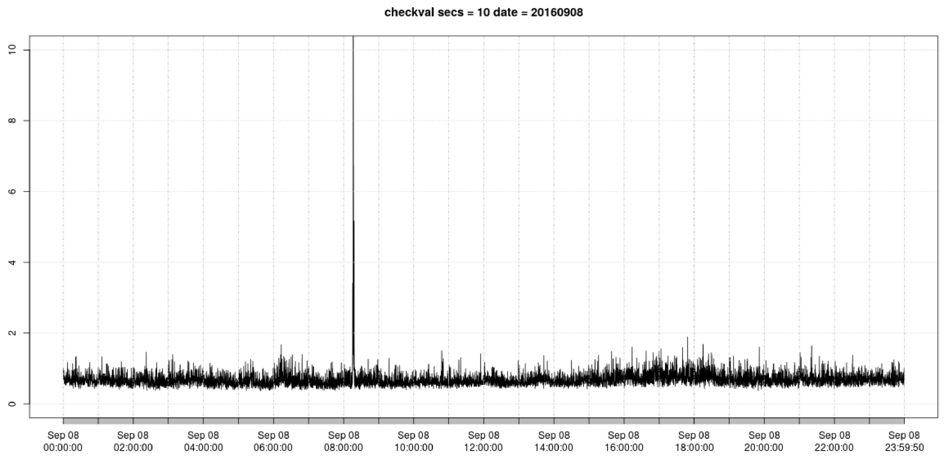
To exhibit the advantages of xts in (besides many others) plotting, here we convert the resulted flow data into an xts object, followed by a plot statement:

```
> fd = read.zoo('data1.csv', sep=",", FUN=function(i) strptime(i,'%s'), ⤸
    ⤷header = TRUE)
> flowdata = as.xts(fd)
> plot(flowdata, major.ticks=600)
```

which results in the following graph:



Of course, this is only the start of our work as data engineer and security researcher: From here on we are working to detect anomalities in the data, due to DDOS or other attacks. Here is a typical example of a time series during a DDOS attack:



## 4 R in education

School and university education often focuses on different programming languages, in particular Java is popular. For statistical and mathematical applications, Mathematica (commercial) and Octave (OSS) are often used, but I consider R the better choice: notation feels very natural, very much like writing mathematics by hand; the large and very supportive community; no requirement for license fees, sources are readily available; and last but not least it is actually used in real-world applications, providing a preparation for life after school!

## 5 Conclusion

While hardly taught in schools or universities, R provides an excellent programming environment for statistical computations, big data, and even school level math. With highly vectorized operations R can compete with large computing frameworks without offloading computations to the GPU (but add-on packages to do this are available, too). Saving and loading R data is very efficient in storage, allowing to pick up work where you left it the other day. Various plotting facilities, and various additional plotting packages, makes generating graphics for reports easy and fun.

For everyone with only a slight interest in a computational approach to mathematics and statistics, R should not be left out the list of considered languages.