Big Data Analytics

Lecture 4: Advanced R Programming

Prof. Dr. Ulrich Matter
(University of St. Gallen)

18/03/2021

1 GitHub

1.1 Initiate a new repository

1. Log in to your GitHub account and click on the plus-sign in the upper right corner. From the drop-down-menu select `New repository`.
2. Give your repository a name, for example `bigdatastat`. Then, click on the big green button `Create repository`. You have just created a new repository.
3. Open Rstudio and and navigate to a place on your hard-disk where you want to have the local copy of your repository.
4. Then create the local repository as suggested by GitHub (see the page shown right after you have clicked on `Create repository`: ". . . or create a new repository on the command line"). In order to do so, you have to switch to the Terminal window in RStudio and type (or copy paste) the commands as given by GitHub. This should look similar to

```
echo "# bigdatastat" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/umatter/bigdatastat.git
git push -u origin master
```

5. Refresh the page of your newly created GitHub repository. You should now see the result of your first commit.
6. Open `README.md` in RStudio and add a few words describing what this repository is all about.

1.2 Clone this course's repository

1. In RStudio, navigate to a folder on your hard-disk where you want to have a local copy of this course's GitHub repository.
2. Open a new browser window and go to `www.github.com/umatter/BigData`.
3. Click on `Clone or download` and copy the link.
4. In RStudio, switch to the Terminal, and type the following command (pasting the copied link).

```
git clone https://github.com/umatter/BigData.git
```

You have now a local copy of the repository which is linked to the one on GitHub. You can see this by changing to the newly created directory, containing the local copy of the repository:

```
cd BigData
```

Whenever there are some updates to the course's repository on GitHub, you can update your local copy with:

```
git pull
```

(Make sure you are in the `BigData` folder when running `git pull`.)

1.3   Fork this course's repository

1. Go to `https://github.com/umatter/BigData`, click on the 'Fork' button in the upper-right corner (follow the instructions).

2. Clone the forked repository (see the cloning of a repository above for details). Assuming you called your forked repository `BigData-forked`, you run the following command in the terminal (replacing `<yourgithubusername>`:

```
git clone https://github.com/`<yourgithubusername>`/BigData-forked.git
```

3. Switch into the newly created directory:

```
cd BigData-forked
```

4. Set a remote connection to the *original* repository

```
git remote add upstream https://github.com/umatter/BigData.git
```

You can verify the remotes of your local clone of your forked repository as follows

```
git remote -v
```

You should see something like

```
origin   https://github.com/<yourgithubusername>/BigData-forked.git (fetch)
origin   https://github.com/<yourgithubusername>/BigData-forked.git (push)
upstream     https://github.com/umatter/BigData.git (fetch)
upstream     https://github.com/umatter/BigData.git (push)
```

5. Fetch changes from the original repository. New material has been added to the original course repository and you want to merge it with your forked repository. In order to do so, you first fetch the changes from the original repository:

```
git fetch upstream
```

6. Make sure you are on the master branch of your local repository:

```
git checkout master
```

7. Merge the changes fetched from the original repo with the master of your (local clone of the) forked repo.

```
git merge upstream/master
```

8. Push the changes to your forked repository on GitHub.

```
git push
```

Now your forked repo on GitHub also contains the commits (changes) in the original repository. If you make changes to the files in your forked repo. you can add, commit, and push them as in any repository. Example: open `README.md` in a text editor (e.g. RStudio), add `# HELLO WORLD` to the last line of `README.md`, and save the changes. Then:

```
git add README.md
git commit -m "hello world"
git push
```

2 Advanced R Programming

2.1 R-tools to investigate performance/resource allocation

package	function	purpose
utils	object.size()	Provides an estimate of the memory that is being used to store an R object.
pryr	object_size()	Works similarly to object.size(), but counts more accurately and includes the size of environments.
pryr	compare_size()	Makes it easy to compare the output of object_size and object.size.
pryr	mem_used()	Returns the total amount of memory (in megabytes) currently used by R.
pryr	mem_change()	Shows the change in memory (in megabytes) before and after running code.
base	system.time()	Returns CPU (and other) times that an R expression used.
microbenchmark	microbenchmark()	Highly accurate timing of R expression evaluation.
bench	mark()	Benchmark a series of functions.
profvis	profvis()	Profiles an R expression and visualizes the profiling data (usage of memory, time elapsed, etc.).

2.2 R-tools to investigate structures and types

package	function	purpose
utils	str()	Compactly display the structure of an arbitrary R object.
base	class()	Prints the class(es) of an R object.
base	typeof()	Determines the (R-internal) type or storage mode of an object.

2.3 Data types and memory/storage

Data loaded into RAM can be interpreted differently by R depending on the data *type*. Some operators or functions in R only accept data of a specific type as arguments. For example, we can store the numeric values 1.5 and 3 in the variables a and b, respectively.

```r
a <- 1.5
b <- 3
a + b
```

```
## [1] 4.5
```

R interprets this data as type double (class 'numeric'):

```r
typeof(a)
```

```
## [1] "double"
```

```r
class(a)
```

```
## [1] "numeric"
```

```r
object.size(a)
```

```
## 56 bytes
```

If, however, we define a and b as follows, R will interpret the values stored in a and b as text (character).

```
a <- "1.5"
b <- "3"
a + b
```

```
typeof(a)
```

```
## [1] "double"
```

```
class(a)
```

```
## [1] "numeric"
```

```
object.size(a)
```

```
## 56 bytes
```

Note that the symbols `1.5` take up more or less memory depending on the data-type they are stored in. This directly links to how data/information is stored/represented in binary code, which in turn is reflected in how much memory is used to store these symbols in an object as well as what we can do with it.

2.4 Data structures

For now, we have only looked at individual bytes of data. An entire data set can consist of gigabytes of data and contain both text and numeric values. R provides several classes of objects providing different data structures. Both the choice of data types and data structures to store data in can affect how much memory is needed to contain a dataset in RAM.

2.4.1 Vectors vs Factors in R

Vectors are collections of values of the same type. They can contain either all numeric values or all character values.
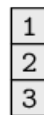

Figure 1: Illustration of a numeric vector (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).

For example, we can initiate a character vector containing information on the home towns of persons participating in a survey.

```
hometown <- c("St.Gallen", "Basel", "St.Gallen")
hometown
```

```
## [1] "St.Gallen" "Basel"     "St.Gallen"
```

```
object.size(hometown)
```

```
## 200 bytes
```

Unlike in the data types example above, it would likely be not that practical to store these values as type `numeric` to save memory. R would not know how to translate these strings into floating point numbers. Alternatively, we could think of a correspondence table that assigns a numeric (id) code to each unique town name in the data set. This way we would save memory but it would mean additional effort to work with the data. Fortunately, basic R already implements exactly this idea in a user-friendly way in a data-structure called `factor`.

Figure 2: Illustration of a factor (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).

Factors are sets of categories. Thus, the values come from a fixed set of possible values.

Considering the same example as above, we can store the same information in an object of type class `factor`.

```
hometown_f <- factor(c("St.Gallen", "Basel", "St.Gallen"))
hometown_f
```

```
## [1] St.Gallen Basel     St.Gallen
## Levels: Basel St.Gallen
```

```
object.size(hometown_f)
```

```
## 584 bytes
```

At first sight, the fact that `hometown_f` takes up more memory than its character vector sibling seems odd. But, we have encountered this kind of 'paradox' before. Again, the more sophisticated approach involves an 'overhead' (here not in terms of computing time but in terms of structure encoded in an object). `hometown_f` has more 'structure' (i.e., a mapping of numbers to 'factor levels'/category labels). This additional structure is also information that needs to be stored somewhere. As in previous examples of this 'overhead costs', this disadvantage is diminishing with larger data sets:

```
# create a large character vector
hometown_large <- rep(hometown, times = 1000)
# and the same content as factor
hometown_large_f <- factor(hometown_large)
# compare size
object.size(hometown_large)
```

```
## 24168 bytes
```

```
object.size(hometown_large_f)
```

```
## 12568 bytes
```

2.4.2 Matrices/Arrays

Matrices are two-dimensional collections of values, arrays higher-dimensional collections of values, of the same type.

Figure 3: Illustration of a numeric matrix (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).

For example, we can initiate a three-row/two-column numeric matrix as follows.

```
my_matrix <- matrix(c(1,2,3,4,5,6), nrow = 3)
my_matrix
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

And a three-dimensional numeric array as follows.

```
my_array <- array(c(1,2,3,4,5,6), dim = 3)
my_array
```

```
## [1] 1 2 3
```

2.4.3 Data frames, tibbles, and data tables

Recall that data frames are the typical representation of a (table-like) data set in R. Each column can contain a vector of a given data type (or a factor), but all columns need to be of identical length. Thus in the context of data analysis, we would say that each row of a data frame contains an observation, and each column contains a characteristic of this observation.
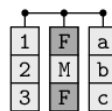

Figure 4: Illustration of a data frame (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).

The historical implementation of data frames in R is not very comfortable to work with large data sets.[1] Several newer implementations of the data-frame concept in R aim to make data processing faster. One is called `tibbles`, implemented and used in the `tidyverse` packages. The other is called `data table`, implemented in the `data.table`-package. In this course we will focus on the `data.table`-package.

Here is how we define a `data.table` in R:

```
# load package
library(data.table)
# initiate a data.table
dt <- data.table(person = c("Alice", "Ben"),
                 age = c(50, 30),
                 gender = c("f", "m"))
dt
```

```
##    person age gender
## 1:  Alice  50      f
## 2:    Ben  30      m
```

2.4.4 Lists

Similar to data frames and data tables, lists can contain different types of data in each element. For example, a list could contain different other lists, data frames, and vectors with differing numbers of elements.

This flexibility can easily be demonstrated by combining some of the data structures created in the examples above:

---

[1]In the early days of R this was not really an issue because data sets that are rather large by today's standards (in the Gigabytes) could not have been handled properly by normal computers anyhow (due to a lack of RAM).
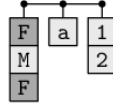
Figure 5: Illustration of a data frame (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).

```r
my_list <- list(my_array, my_matrix, dt)
my_list
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## [[3]]
##    person age gender
## 1:  Alice  50      f
## 2:    Ben  30      m
```

3 Programming with (Big) Data in R

3.1 Typical Programming Tasks

Programming tasks in the context of data analysis typically fall into one of the following broad categories.

- Procedures to import/export data.
- Procedures to clean and filter data.
- Implement functions for statistical analysis.

When writing a program to process large amounts of data in any of these areas, it is helpful to take into consideration the following design choices:

1. Which basic (already implemented) R functions are more or less suitable as building blocks for the program?
2. How can we exploit/avoid some of R's lower-level characteristics in order to implement efficient functions?
3. Is there a need to interface with a lower-level programming language in order to speed up the code? (advanced topic)

Finally, there is an additional important point to be made regarding the implementation of functions for *statistical analysis*: Independent of *how* we write a statistical procedure in R (or in any other language, for that matter), is there an *alternative statistical procedure/algorithm* that is faster but delivers approximately the same result (as long as we use a sufficiently large data sets). The following subsections elaborate briefly on each of these points and show some code examples to further illustrate these points.

3.2 Building blocks for programming with big data

When writing a program in R, we can rely on many already implemented functions on which we can build. Often, there are even several functions already implemented that take care of essentially the same task. When the amount of data to be processed by these functions is not large, it doesn't matter that much which ones we

choose to build our program on. However, when we are writing a program which likely has to process large amounts of data, we should think more closely about which building blocks we choose to base our program on. For example, when writing the data-import part of a program, we could use the traditional `read.csv()` or `fread()` from the `data.table`-package. The result is very similar (in many situations, the differences of the resulting objects would not matter at all).

```r
# read a CSV-file the 'traditional way'
flights <- read.csv("../data/flights.csv")
class(flights)
```

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

```r
# alternative (needs the data.table package)
library(data.table)
flights <- fread("../data/flights.csv")
class(flights)
```

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

However, the latter approach is usually much faster (see above for why this is the case in this example).

```r
system.time(flights <- read.csv("../data/flights.csv"))
```

```
##    user  system elapsed
##   1.382   0.012   1.401
```

```r
system.time(flights <- fread("../data/flights.csv"))
```

```
##    user  system elapsed
##   0.438   0.000   0.080
```

3.3 Writing Efficient Code

3.3.1 Memory allocation before looping

Recall the code example from the introductory lecture. When we write a `for`-loop that results in a vector or list of values, it is favorable to instruct R to pre-allocate the memory necessary to contain the final result. If we don't do that, each iteration of the loop causes R to re-allocate memory because the number of elements in the vector/list is changing. In simple terms, this means that R needs to execute more steps in each iteration.

In the following example, we compare the performance of two functions. One taking this principle into account, the other not. The function takes a numeric vector as input and returns the square root of each element of the numeric vector.

```r
# naïve implementation
sqrt_vector <-
    function(x) {
        output <- c()
        for (i in 1:length(x)) {
            output <- c(output, x[i]^(1/2))
        }

        return(output)
    }

# implementation with pre-allocation of memory
sqrt_vector_faster <-
    function(x) {
        output <- rep(NA, length(x))
```

```
    for (i in 1:length(x)) {
        output[i] <-  x[i]^(1/2)
    }

    return(output)
}
```

As a proof of concept we use `system.time()` to measure the difference in speed for various input sizes.[2]

```
# the different sizes of the vectors we will put into the two functions
input_sizes <- seq(from = 100, to = 10000, by = 100)
# create the input vectors
inputs <- sapply(input_sizes, rnorm)

# compute ouputs for each of the functions
output_slower <-
    sapply(inputs,
           function(x){ system.time(sqrt_vector(x))["elapsed"]
              }
          )
output_faster <-
    sapply(inputs,
           function(x){ system.time(sqrt_vector_faster(x))["elapsed"]
              }
          )
```

The following plot shows the difference in the performance of the two functions.

```
# load packages
library(ggplot2)

# initiate data frame for plot
plotdata <- data.frame(time_elapsed = c(output_slower, output_faster),
                       input_size = c(input_sizes, input_sizes),
                       Implementation= c(rep("sqrt_vector", length(output_slower)),
                              rep("sqrt_vector_faster", length(output_faster))))

# plot
ggplot(plotdata, aes(x=input_size, y= time_elapsed)) +
    geom_point(aes(colour=Implementation)) +
    theme_minimal(base_size = 18) +
    ylab("Time elapsed (in seconds)") +
    xlab("No. of elements processed")
```
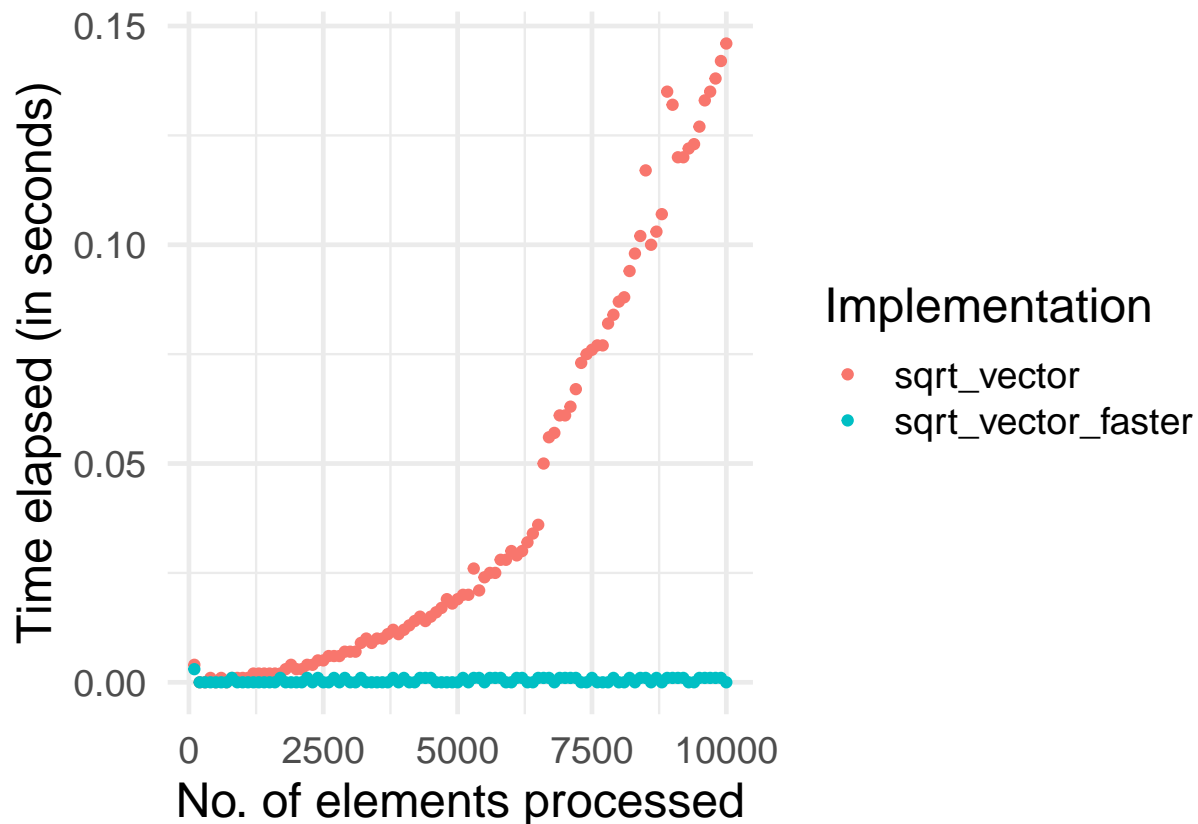
---

[2]We generate the numeric input by drawing vectors of (pseudo) random numbers via `rnorm()`.

3.3.2 Vectorization

We can further improve the performance of this function by exploiting the fact that in R 'everything is a vector' and that many of the basic R functions (such as math operators) are *vectorized*. In simple terms, this means that an operation is implemented to directly work on vectors in such a way that it can take advantage of the similarity of each of the vector's elements. That is, R only has to figure out once how to apply a given function to a vector element in order to apply it to all elements of the vector. In a simple loop, R has to go through the same 'preparatory' steps again and again in each iteration.

```r
# implementation with vectorization
sqrt_vector_fastest <-
    function(x) {
            output <-  x^(1/2)
        return(output)
    }


# speed test
output_fastest <-
    sapply(inputs,
        function(x){ system.time(sqrt_vector_fastest(x))["elapsed"]
            }
        )
```

Let's have a look at whether this improves the function's performance further.

```r
# load packages
library(ggplot2)

# initiate data frame for plot
```

```
plotdata <- data.frame(time_elapsed = c(output_faster, output_fastest),
                       input_size = c(input_sizes, input_sizes),
                       Implementation= c(rep("sqrt_vector_faster", length(output_faster)),
                           rep("sqrt_vector_fastest", length(output_fastest))))

# plot
ggplot(plotdata, aes(x=input_size, y= time_elapsed)) +
    geom_point(aes(colour=Implementation)) +
    theme_minimal(base_size = 18) +
    ylab("Time elapsed (in seconds)") +
    xlab("No. of elements processed")
```
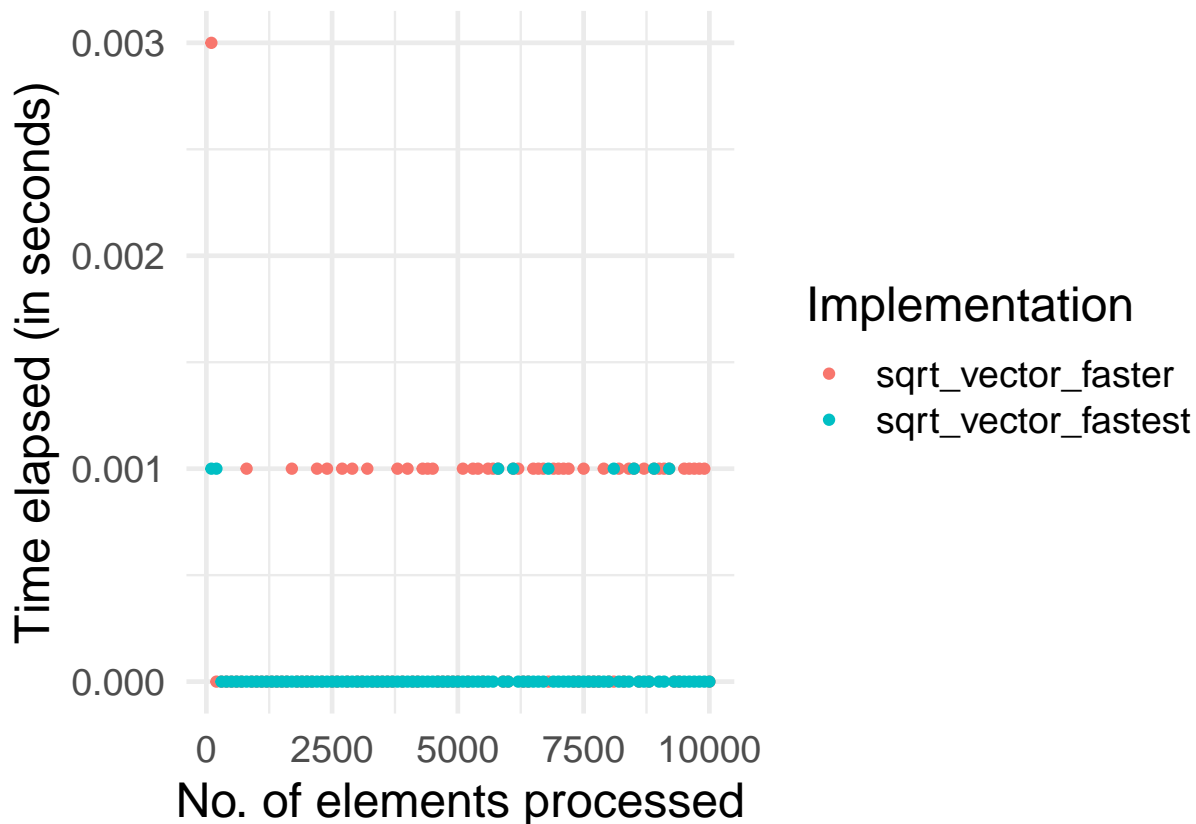

In the example above, we have simply exploited the fact that many of R's basic functions (such as math operators) are vectorized. If the program we want to implement cannot directly benefit from such a function, there are basically two ways to make use of vectorization (instead of loops written in R).

One approach is to use an `apply`-type function instead of loops. Probably most widely used is `lapply()`, a function that takes a vector (atomic or list) as input and applies a function `FUN` to each of its elements. It is a straightforward alternative to `for`-loops in many situations. The following example shows how we can get the same result by either writing a loop or using `lapply()`. The aim of the code example is to import the Health News in Twitter Data Set by Karami et al. (2017). The raw data consists of several text files that need to be imported to R consecutively.

The text-files are located in `data/twitter_texts/`. For either approach of importing all of these files, we first need a list of the paths to all of the files. We can get this with `list.files()`. Also, for either approach we will make use of the `fread`-function in the `data.table`-package.

```
# load packages
library(data.table)
```

```r
# get a list of all file-paths
textfiles <- list.files("../data/twitter_texts", full.names = TRUE)
```

Now we can read in all the text files with a `for`-loop as follows.

```r
# prepare loop
all_texts <- list()
n_files <- length(textfiles)
length(all_texts) <- n_files
# read all files listed in textfiles
for (i in 1:n_files) {
     all_texts[[i]] <- fread(textfiles[i])
}
```

The imported files are now stored as `data.table`-objects in the list `all_texts`. With the following line of code we combine all of them in one `data.table`.

```r
# combine all in one data.table
twitter_text <- rbindlist(all_texts)
# check result
str(twitter_text)
```

```
## Classes 'data.table' and 'data.frame':   42422 obs. of  3 variables:
##  $ V1:integer64 585978391360221184 585947808772960257 585947807816650752 585866060991078401 585794100
##  $ V2: chr  "Thu Apr 09 01:31:50 +0000 2015" "Wed Apr 08 23:30:18 +0000 2015" "Wed Apr 08 23:30:18 +0
##  $ V3: chr  "Breast cancer risk test devised http://bbc.in/1CimpJF" "GP workload harming care - BMA p
##  - attr(*, ".internal.selfref")=<externalptr>
```

Alternatively, we can make use of `lapply` as follows in order to achieve exactly the same.

```r
# prepare loop
all_texts <- lapply(textfiles, fread)
# combine all in one data.table
twitter_text <- rbindlist(all_texts)
# check result
str(twitter_text)
```

```
## Classes 'data.table' and 'data.frame':   42422 obs. of  3 variables:
##  $ V1:integer64 585978391360221184 585947808772960257 585947807816650752 585866060991078401 585794100
##  $ V2: chr  "Thu Apr 09 01:31:50 +0000 2015" "Wed Apr 08 23:30:18 +0000 2015" "Wed Apr 08 23:30:18 +0
##  $ V3: chr  "Breast cancer risk test devised http://bbc.in/1CimpJF" "GP workload harming care - BMA p
##  - attr(*, ".internal.selfref")=<externalptr>
```

Finally, we can make use of `Vectorization()` in order to 'vectorize' (as far as possible) our own import function (written for this example).

```r
# initiate the import function
import_file <-
     function(x) {
          parsed_x <- fread(x)
          return(parsed_x)
     }


# 'vectorize' it
import_files <- Vectorize(import_file, SIMPLIFY = FALSE)

# Apply the vectorized function
```

```
all_texts <- import_files(textfiles)
twitter_text <- rbindlist(all_texts)
# check the result
str(twitter_text)
```

```
## Classes 'data.table' and 'data.frame':   42422 obs. of  3 variables:
## $ V1:integer64 585978391360221184 585947808772960257 585947807816650752 585866060991078401 58579410(
## $ V2: chr  "Thu Apr 09 01:31:50 +0000 2015" "Wed Apr 08 23:30:18 +0000 2015" "Wed Apr 08 23:30:18 +0
## $ V3: chr  "Breast cancer risk test devised http://bbc.in/1CimpJF" "GP workload harming care - BMA |
## - attr(*, ".internal.selfref")=<externalptr>
```

3.4   R, beyond R

So far, we have explored idiosyncrasies of R we should be aware of when writing programs to handle and analyze large data sets. While this has shown that R has many advantages for working with data, it also revealed some aspects of R that might result in low performance compared other programming languages. A simple generic explanation for this is that R is an interpreted language, meaning that when we execute R code, it is processed (statement by statement) by an 'interpreter' that translates the code into machine code (without the user giving any specific instructions). In contrast, when writing code in a 'compiled language', we first have to explicitly compile the code and then run the compiled program. Running code that is already compiled is typically much faster than running R code that has to be interpreted before it can actually be processed by the CPU.

For advanced programmers, R offers various options to directly make use of compiled programs (for example, written in C, C++, or FORTRAN). In fact several of the core R functions installed with the basic R distribution are implemented in one of these lower-level programming languages and the R function we call simply interacts with these functions.

We can actually investigate this by looking at the source code of an R function. When simply typing the name of a function (such as our `import_file()`) to the console, R is printing the function's source code to the console.

```
import_file
```

```
## function(x) {
##          parsed_x <- fread(x)
##          return(parsed_x)
##      }
## <bytecode: 0x56325a5f6f68>
```

However, if we do the same for function `sum`, we don't see any actual source code.

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

Instead `.Primitive()` indicates that `sum()` is actually referring to an internal function (in this case implemented in C).

While the use of functions implemented in a lower-level language is a common technique to improve the speed of 'R' functions, it is particularly prominent in the context of functions/packages made to deal with large amounts of data (such as the `data.table` package).

References

Karami, Amir, Aryya Gangopadhyay, Bin Zhou, and Hadi Kharrazi. 2017. "Fuzzy Approach Topic Discovery in Health and Medical Corpora." *International Journal of Fuzzy Systems* 20 (4): 1334–45.

Murrell, Paul. 2009. *Introduction to Data Technologies.* London, UK: CRC Press.