

Big Data Statistics for R and Python

Lecture 2: Programming with Data

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

25/02/2019

1 GitHub

1.1 Initiate a new repository

1. Log into 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 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`.)

2 Programming with Data: Data Structures and Data Types

2.1 R-tools to investigate structures and types

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

2.2 Data types

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 integer values 1.5 and 3 in the variables `a` and `b`, respectively.

```
a <- 1.5
b <- 3
```

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

```
typeof(a)
```

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

```
class(a)
```

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

Given that these bytes of data are interpreted as numeric, we can use operators (here: math operators) that can work with such functions:

```
a + b
```

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

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"
```

```
typeof(a)
```

```
## [1] "character"
```

```
class(a)
```

```
## [1] "character"
```

Now the same line of code as above will result in an error:

```
a + b
```

```
## Error in a + b: non-numeric argument to binary operator
```

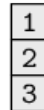
The reason is, that the operator `+` expects numeric or integer values as arguments.

2.3 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.

2.3.1 Vectors

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



1
2
3

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 the names of persons.

```
persons <- c("Andy", "Brian", "Claire")
persons
```

```
## [1] "Andy"  "Brian" "Claire"
```

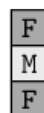
and we can initiate a numeric vector with the age of these persons.

```
ages <- c(24, 50, 30)
ages
```

```
## [1] 24 50 30
```

2.3.2 Factors

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



F
M
F

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

For example, we might want to initiate a factor that indicates the gender of a number of people.

```
gender <- factor(c("Male", "Male", "Female"))
gender
```

```
## [1] Male  Male  Female
## Levels: Female Male
```

2.3.3 Matrices/Arrays

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

1	4	7
2	5	8
3	6	9

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
```

2.3.4 Data frames, tibbles, and data tables

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.

1	F	a
2	M	b
3	F	c

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, based on the examples of vectors and factors shown above.

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

¹In 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).

```
# initiate a data.table
dt <- data.table(person = persons, age = ages, gender = gender)
dt

##    person age gender
## 1:   Andy  24   Male
## 2:  Brian  50   Male
## 3: Claire  30 Female
```

2.3.5 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.

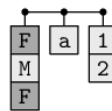


Figure 5: Illustration of a data frame (symbolic). Figure by Murrell (2009) (licensed under CC BY-NC-SA 3.0 NZ).

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

```
my_list <- list(my_array, my_matrix, df)
my_list

## [[1]]
## [1] 1 2 3
##
## [[2]]
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## [[3]]
## function (x, df1, df2, ncp, log = FALSE)
## {
##     if (missing(ncp))
##         .Call(C_df, x, df1, df2, log)
##     else .Call(C_dnf, x, df1, df2, ncp, log)
## }
## <bytecode: 0x44c8370>
## <environment: namespace:stats>
```

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).

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

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

```
# 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.

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

```
##      user  system elapsed
##    1.105    0.007    1.115
```

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

```
##      user  system elapsed
##    0.385    0.000    0.151
```

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.

```
# naive 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 outputs 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"]
    })
```

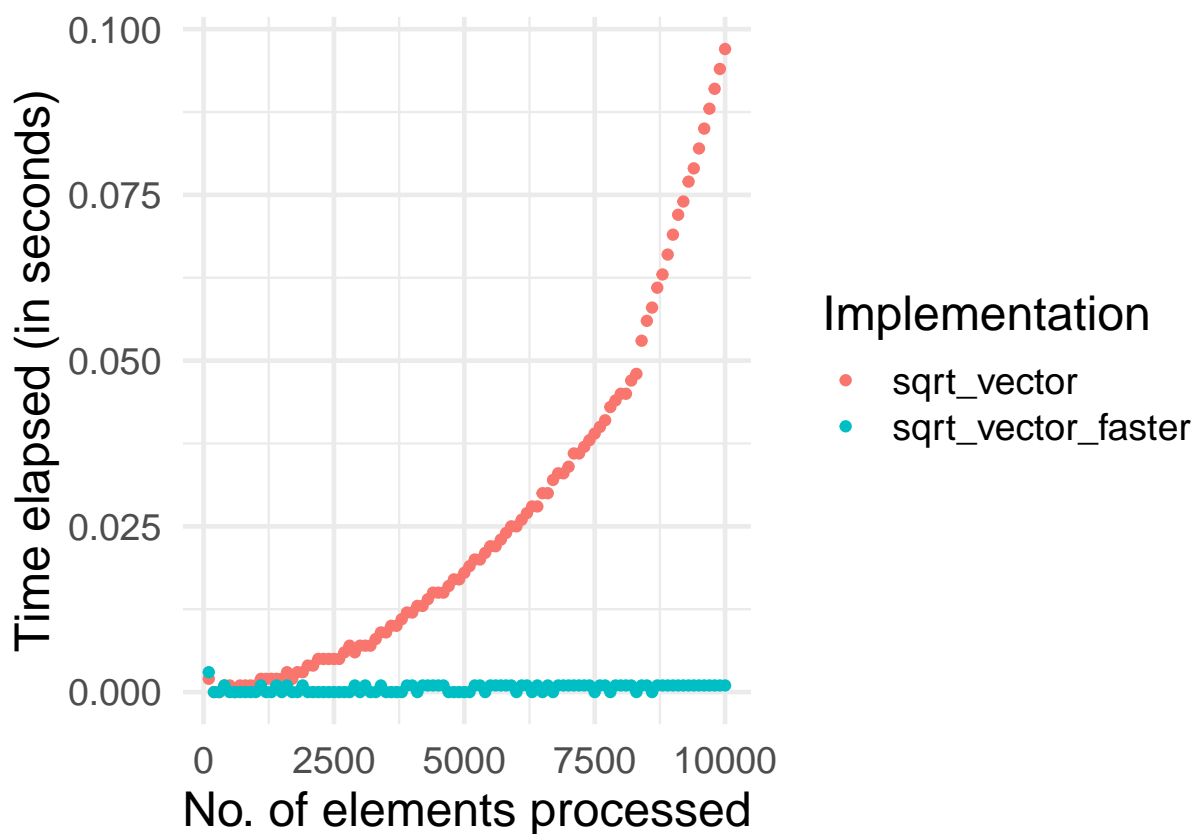
The following plot shows the difference in the performance of the two functions.

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

```
# 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")
```



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.


```

# 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 whether this improves the function's performance further.

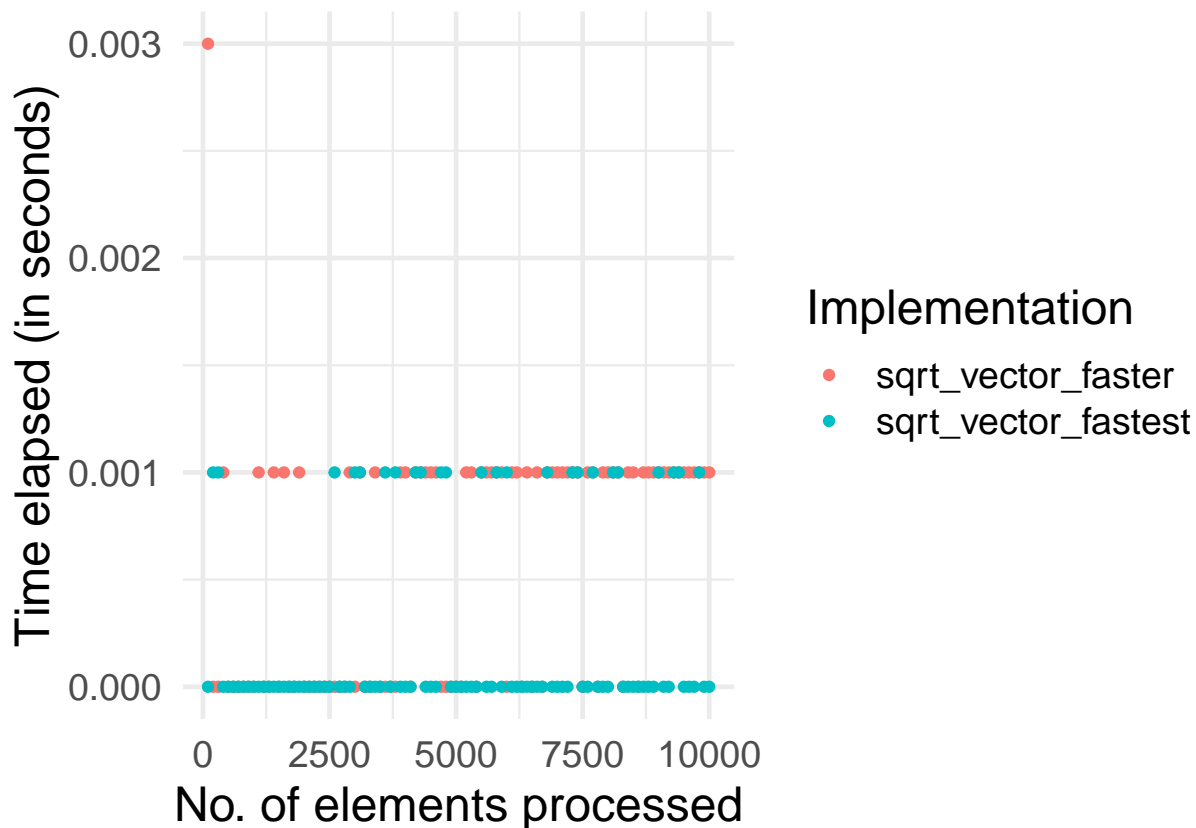
```

# 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)

# 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.

```
# 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`.

```
# 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 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 +
## $ 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.

```
# 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 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 +
## $ 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).

```
# 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 +
## $ V3: chr  "Breast cancer risk test devised http://bbc.in/1CimpJF" "GP workload harming care - BMA p
## - 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: 0x61169b0>
```

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).

3.5 Big Data statistics

Finally, even if all of the techniques outlined above are taken into account in order to optimize the implementation of a statistical analysis of large data sets in R (or any other language), there might be room for improvement. We might want to consider alternative statistical procedures that happened to be more efficient to compute a given statistic based on large amounts of data.

Here, we look at one such procedure that has recently been developed to estimate linear models when the classical OLS estimator is too computationally intensive for very large samples: The *Uluru* algorithm (Dhillon et al. 2013).

3.5.1 OLS as a point of reference

Recall the OLS estimator in matrix notation, given the linear model $\mathbf{y} = \mathbf{X}\beta + \epsilon$:

$$\hat{\beta}_{OLS} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}.$$

In order to compute $\hat{\beta}_{OLS}$, we have to compute $(\mathbf{X}^T\mathbf{X})^{-1}$, which implies a computationally expensive matrix inversion.³ If our data set is large, \mathbf{X} is large and the inversion can take up a lot of computation time. Moreover, the inversion and matrix multiplication to get $\hat{\beta}_{OLS}$ needs a lot of memory. In practice, it might well be that the estimation of a linear model via OLS with the standard approach in R (`lm()`) brings a computer to its knees, as there is not enough RAM available.

To further illustrate the point, we implement the OLS estimator in R.

```
beta_ols <-
  function(X, y) {

    # compute cross products and inverse
    XXi <- solve(crossprod(X,X))
    Xy <- crossprod(X, y)

    return( XXi  %*% Xy )
  }
```

Now, we will test our OLS estimator function with a few (pseudo) random numbers in a Monte Carlo study. First, we set the sample size parameters `n` (how many observations shall our pseudo sample have?) and `p` (how many variables shall describe these observations?) and initiate the data set `X`.

```
# set parameter values
n <- 10000000
p <- 4

# Generate sample based on Monte Carlo
# generate a design matrix (~ our 'dataset') with four variables and 10000 observations
X <- matrix(rnorm(n*p, mean = 10), ncol = p)
# add column for intercept
X <- cbind(rep(1, n), X)
```

Now we define how the real linear model looks like that we have in mind and compute the output `y` of this model, given the input `X`.⁴

```
# MC model
y <- 2 + 1.5*X[,2] + 4*X[,3] - 3.5*X[,4] + 0.5*X[,5] + rnorm(n)
```

Finally, we test our `beta_ols` function.

```
# apply the ols estimator
beta_ols(X, y)
```

```
##           [,1]
## [1,]  2.0013574
## [2,]  1.5002882
## [3,]  3.9996834
## [4,] -3.5002159
## [5,]  0.5001224
```

3.5.2 The Uluru algorithm as an alternative to OLS

Following Dhillon et al. (2013), we implement a procedure to compute $\hat{\beta}_{Uluru}$:

³The computational complexity of this is larger than $O(n^2)$. That is, for an input of size n , the time needed to compute (or the number of operations needed) is n^2 .

⁴In reality we would not know this, of course. Acting as if we knew the real model is exactly point of Monte Carlo studies to analyze the properties of estimators by simulation.

$$\hat{\beta}_{Uluru} = \hat{\beta}_{FS} + \hat{\beta}_{correct}$$

, where

$$\hat{\beta}_{FS} = (\mathbf{X}_{subs}^T \mathbf{X}_{subs})^{-1} \mathbf{X}_{subs}^T \mathbf{y}_{subs}$$

, and

$$\hat{\beta}_{correct} = \frac{n_{subs}}{n_{rem}} \cdot (\mathbf{X}_{subs}^T \mathbf{X}_{subs})^{-1} \mathbf{X}_{rem}^T \mathbf{R}_{rem}$$

, and

$$\mathbf{R}_{rem} = \mathbf{Y}_{rem} - \mathbf{X}_{rem} \cdot \hat{\beta}_{FS}$$

.

The key idea behind this is that the computational bottleneck of the OLS estimator, the cross product and matrix inversion, $(\mathbf{X}^T \mathbf{X})^{-1}$, is only computed on a sub-sample (\mathbf{X}_{subs} , etc.), not the entire data set. However, the remainder of the data set is also taken into consideration (in order to correct a bias arising from the sub-sampling). Again, we implement the estimator in R to further illustrate this point.

```
beta_uluru <-
  function(X_subs, y_subs, X_rem, y_rem) {

    # compute beta_fs (this is simply OLS applied to the subsample)
    XXi_subs <- solve(crossprod(X_subs, X_subs))
    Xy_subs <- crossprod(X_subs, y_subs)
    b_fs <- XXi_subs %*% Xy_subs

    # compute \mathbf{R}_{rem}
    R_rem <- y_rem - X_rem %*% b_fs

    # compute \hat{\beta}_{correct}
    b_correct <- (nrow(X_subs)/(nrow(X_rem))) * XXi_subs %*% crossprod(X_rem, R_rem)

    # beta uluru
    return(b_fs + b_correct)
  }
```

Test it with the same input as above:

```
# set size of subsample
n_subs <- 1000
# select subsample and remainder
n_obs <- nrow(X)
X_subs <- X[1L:n_subs,]
y_subs <- y[1L:n_subs]
X_rem <- X[(n_subs+1L):n_obs,]
y_rem <- y[(n_subs+1L):n_obs]

# apply the uluru estimator
beta_uluru(X_subs, y_subs, X_rem, y_rem)
```

```
##           [,1]
## [1,]  2.0782855
## [2,]  1.5013826
## [3,]  3.9991481
## [4,] -3.5034106
## [5,]  0.4955083
```

This looks quite good already. Let's have a closer look with a little Monte Carlo study. The aim of the simulation study is to visualize the difference between the classical OLS approach and the *Uluru* algorithm with regard to bias and time complexity if we increase the sub-sample size in *Uluru*. For simplicity, we only look at the first estimated coefficient β_1 .

```
# define subsamples
n_subs_sizes <- seq(from = 1000, to = 500000, by=10000)
n_runs <- length(n_subs_sizes)
# compute uluru result, stop time
mc_results <- rep(NA, n_runs)
mc_times <- rep(NA, n_runs)
for (i in 1:n_runs) {
  # set size of subsample
  n_subs <- n_subs_sizes[i]
  # select subsample and remainder
  n_obs <- nrow(X)
  X_subs <- X[1L:n_subs,]
  y_subs <- y[1L:n_subs]
  X_rem <- X[(n_subs+1L):n_obs,]
  y_rem <- y[(n_subs+1L):n_obs]

  mc_results[i] <- beta_uluru(X_subs, y_subs, X_rem, y_rem)[2] # the first element is the intercept
  mc_times[i] <- system.time(beta_uluru(X_subs, y_subs, X_rem, y_rem))[3]
}

# compute ols results and ols time
ols_time <- system.time(beta_ols(X, y))
ols_res <- beta_ols(X, y)[2]
```

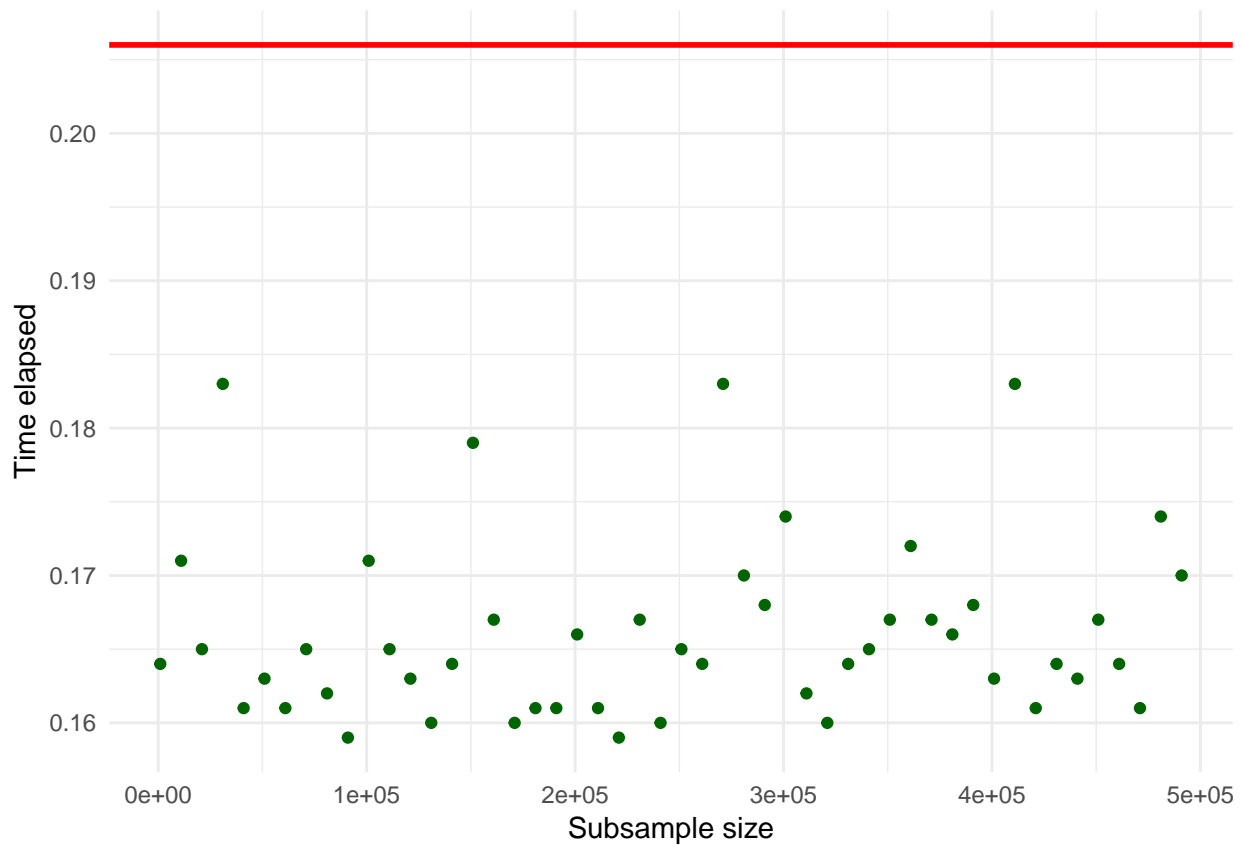
Let's visualize the comparison with OLS.

```
# load packages
library(ggplot2)

# prepare data to plot
plotdata <- data.frame(betal = mc_results,
                       time_elapsed = mc_times,
                       subs_size = n_subs_sizes)
```

First, let's look at the time used estimate the linear model.

```
ggplot(plotdata, aes(x = subs_size, y = time_elapsed)) +
  geom_point(color="darkgreen") +
  geom_hline(yintercept = ols_time[3],
            color = "red",
            size = 1) +
  theme_minimal() +
  ylab("Time elapsed") +
  xlab("Subsample size")
```

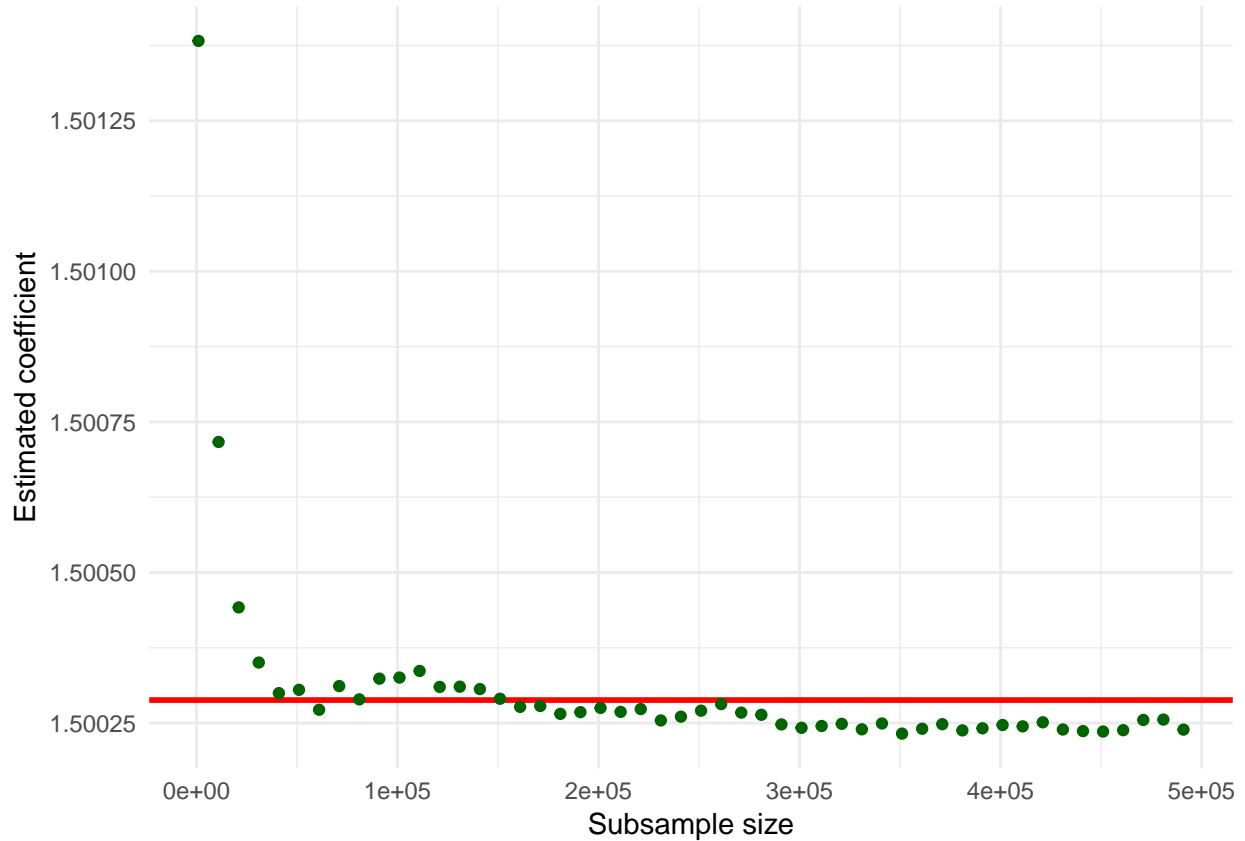


The horizontal red line indicates the computation time for estimation via OLS, the green points indicate the computation time for the estimation via the Uluru algorithm. Note that even for large sub-samples, the computation time is substantially lower than for OLS.

Finally, let's have a look at how close the results are to OLS.

```
ggplot(plotdata, aes(x = subs_size, y = beta1)) +
  geom_hline(yintercept = ols_res,
             color = "red",
             size = 1) +
  geom_point(color="darkgreen") +

  theme_minimal() +
  ylab("Estimated coefficient") +
  xlab("Subsample size")
```

The horizontal red line indicates the size of the estimated coefficient, when using OLS. The green points indicate the size of the same coefficient estimated by the Uluru algorithm for different sub-sample sizes. Note that even relatively small sub-samples already deliver estimates very close to the OLS estimates.

References

- Dhillon, Paramveer, Yichao Lu, Dean P. Foster, and Lyle Ungar. 2013. “New Subsampling Algorithms for Fast Least Squares Regression.” In *Advances in Neural Information Processing Systems 26*, 360–68.
- 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.