Big Data Analytics

Lecture 2: Programming with (Big) Data

Prof. Dr. Lyudmila Grigoryeva (based on course lecture materials of Prof. Dr. Ulrich Matter)

Updates

Group Examinations

- Make sure to build teams of 3 (2).
- Register your team (and all members):
- *Team name*: important for handing out team assignments via GitHub Classroom.

Goals for today

- Know basic tools to assess performance of an R-program/script.
- Know the most common tricks to write more efficient R code.
- Know the very basics of SQL.
- (Be able to set up an SQLite database on your own machine and import data to it.)

Software: Programming with (Big) Data

Programming with (Big) data: common tasks

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

Programming with (Big) data: considerations

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 characteristics?
3. Is there a need to interface with a lower-level programming language? (advanced topic)

Measuring R performance

Measuring performance

Usual angle: measure how long a script/code chunk needs to run through, find ways to speed it up.

A simple example:

```
# how much time does it take to run this loop?
system.time(for (i in 1:100) {i + 5})
```

```
##    user  system elapsed
##   0.015   0.002   0.016
```

Measuring performance: speed

```
# load package
library(microbenchmark)
# how much time does it take to run this loop (exactly)?
microbenchmark(for (i in 1:100) {i + 5})
```

```
## Unit: milliseconds
##                          expr      min       lq     mean   median       uq     max neval
##  for (i in 1:100) {     i + 5 } 1.228758 1.310188 1.598654 1.471728 1.819067 3.93013   100
```

Measuring performance: memory

What part of a program (or which object) takes up how much memory?

A simple example:

```
hello <- "Hello, World!"
object.size(hello)
```

```
## 120 bytes
```

Measuring performance: changes in memory

```
# load package
library(pryr)

# initiate a vector with 1000 (pseudo)-random numbers
mem_change(
        thousand_numbers <- runif(1000)
        )
```

```
## 12.8 kB
```

```
# initiate a vector with 1M (pseudo)-random numbers
mem_change(
        a_million_numbers <- runif(1000^2)
        )
```

```
## 8 MB
```

Profiling

Assess several lines of code/an entire script. Get overview of timing and memory usage. Find bottlenecks.

- `bench` package: overview in console, compare different implementations.
- `profvis` package: graphical overview.

`bench::mark()` example

example in R (see `lecture2.R`)

`profvis` example

example in R (see `lecture2.R`)

Overview/summary

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`	`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.).

Writing efficient R code

Memory allocation and growing objects

- R tends to *"grow"* already initiated *objects in memory* when they are modified.
- Initially, a small amount of memory is occupied by the object at some location in memory.
- *Problem*: growing objects mean re-allocating memory, which needs time!

Example to illustrate the problem: for-loops

Context: implement a function that computes the square-root of each element value in a numeric vector. Two approaches: a) naïve implementation (ignorant of growing objects), b) pre-allocation of memory.

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

          return(output)
     }
```

Example to illustrate the problem: for-loops

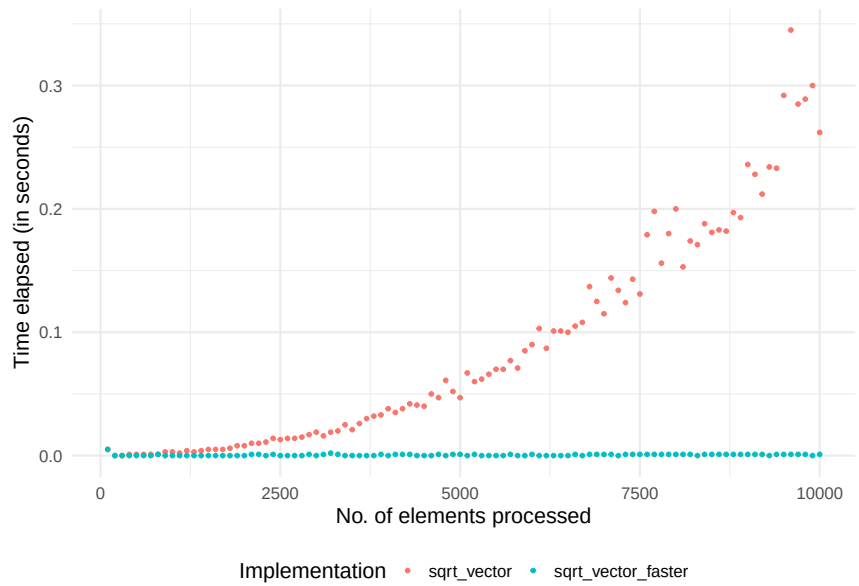
```
# b) implementation with pre-allocation of memory
sqrt_vector_faster <-
     function(x) {
          output <- rep(NA, length(x))
          for (i in 1:length(x)) {
               output[i] <-  x[i]^(1/2)
          }

          return(output)
     }
```

Example to illustrate the problem: for-loops

```
# 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"]
                 }
            )
```

Example to illustrate the problem: for-loops

```
# 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) +
     theme(legend.position = "bottom") +
     ylab("Time elapsed (in seconds)") +
     xlab("No. of elements processed")
```



Memory allocation and growing objects: take-away message

- *Avoid growing objects, pre-allocate memory!*
- Simple (but important) case: properly initiate container for *for-loop* results.

Vectorization in basic R functions

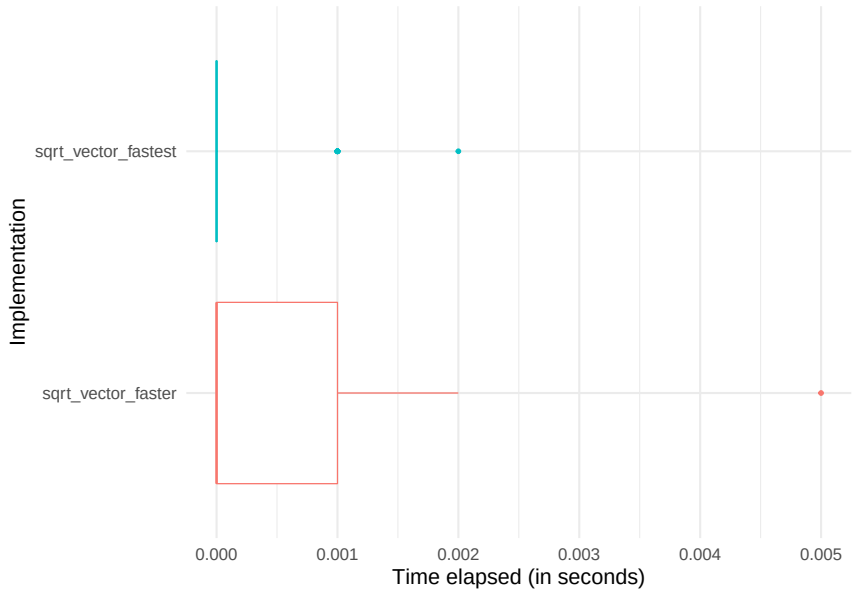
- In R 'everything is a vector' and many of the most basic R functions (such as math operators) are *vectorized*.
- Directly work on vectors, take advantage of the similarity of each of the vector's elements.
- Contrast: In a simple loop, R has to go through the same 'preparatory' steps again and again in each iteration.

Vectorization in basic R functions: illustration

```
# 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"]
                 }
            )
```

Vectorization in basic R functions: illustration

```
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=time_elapsed, y=Implementation)) +
     geom_boxplot(aes(colour=Implementation), show.legend = FALSE) +
     theme_minimal(base_size = 18) + xlab("Time elapsed (in seconds)")
```



Vectorization: take-away message

- *Make use of vectorization (in basic R functions)*
- Simple (but important) case: math operators.

apply-type functions in R

Looks like vectorization, but actually runs a loop under the hood.

Example: use `sapply` to compute column-wise averages.

```
# load and inspect example data (see ?midwest for details)
data("midwest")
head(midwest)
```

```
## # A tibble: 6 x 28
##     PID county    state  area poptotal popdensity popwhite popblack popamerindian popasian popother
##   <int> <chr>     <chr> <dbl>    <int>      <dbl>    <int>    <int>          <int>    <int>    <int>
## 1   561 ADAMS     IL    0.052    66090      1271.    63917     1702             98      249      124
## 2   562 ALEXANDER IL    0.014    10626       759      7054     3496             19       48        9
## 3   563 BOND      IL    0.022    14991       681.    14477      429             35       16       34
## 4   564 BOONE     IL    0.017    30806      1812.    29344      127             46      150     1139
## 5   565 BROWN     IL    0.018     5836       324.     5264      547             14        5        6
## 6   566 BUREAU    IL    0.05     35688       714.    35157       50             65      195      221
## # ... with 17 more variables: percwhite <dbl>, percblack <dbl>, percamerindan <dbl>,
## #   percasian <dbl>, percother <dbl>, popadults <int>, perchsd <dbl>, percollege <dbl>,
## #   percprof <dbl>, poppovertyknown <int>, percpovertyknown <dbl>, percbelowpoverty <dbl>,
## #   percchildbelowpovert <dbl>, percadultpoverty <dbl>, percelderlypoverty <dbl>, inmetro <int>,
## #   category <chr>
```

```
# compute column averages for columns 5 to 11
sapply(midwest[, 5:11], mean)
```

```
##      poptotal    popdensity      popwhite      popblack popamerindian      popasian      popother
##    96130.3021     3097.7430    81839.9153    11023.8810      343.1098     1310.4645     1612.9314
```

`apply`-type functions in R: take-away message

- Instead of writing simple loops, use `apply`-type functions to save time writing code (and make the code easier to read).
- This automatically avoids the memory-allocation problems with growing objects.
- `apply`-type functions look like vectorization, but are actually running loops.

Avoid unnecessary copying

- Objects/values do not have names but *names have values*!
- Objects have a 'memory address'/identifiers.
- We can 'bind' several different names to values.

Illustration: binding

```
# initiate object
a <- runif(10000)
# link other name to values
b <- a

# proof that this is not copying
object_size(a)
```

```
## 80.05 kB
```

```
mem_change(c <- a)
```

```
## -4.41 kB
```

Illustration: memory address

```
# load packages
library(lobstr)

# check memory addresses of objects
obj_addr(a)
```

```
## [1] "0x7fcb580c8000"
```

```
obj_addr(b)
```

```
## [1] "0x7fcb580c8000"
```

Illustration: copy-on-modify

```
# check the first element's value
a[1]
```

```
## [1] 0.04556701
```

```
b[1]
```

```
## [1] 0.04556701
```

```
# modify a, check memory change
mem_change(a[1] <- 0)
```

```
## 79 kB
```

```
# check memory addresses
obj_addr(a)
```

```
## [1] "0x7fcb68750000"
```

```
obj_addr(b)
```

```
## [1] "0x7fcb580c8000"
```

The entire vector was copied!

Illustration: modify in place

With a single binding, no need to copy!

```
mem_change(d <- runif(10000))
```

```
## 80.2 kB
```

```
mem_change(d[1] <- 0)
```

```
## 584 B
```

Avoid unnecessary copying: take-away message

- In practice (more complex code) it is often hard to predict whether or not a copy will occur.
  - E.g., usual R functions vs. 'primitive' C functions.
- Use `tracemem()` to check your code for potential improvements (avoid unnecessary copying).

Releasing memory

- If your program uses up a lot of memory, all processes on your computer might substantially slow down.
- Remove/delete an object once you do not need it anymore with the `rm()` function.

Example:

```
mem_change(large_vector <- runif(10^8))
```

```
## 800 MB
```

```
mem_change(rm(large_vector))
```

```
## -800 MB
```

Releasing memory: take-away message

- `rm()` removes objects that are currently accessible in the global R environment.
- However, some objects/values might technically not be visible/accessible anymore.
- Solution: call `gc()` (the garbage collector).
- R will automatically run the garbage collector once it is close to running out of memory, but, explicitly calling `gc` can still improve the performance of your script when working with large data sets

Beyond R

- For advanced programmers, R offers various options to directly make use of compiled programs (for example, written in C, C++, or FORTRAN).
- Several of the core R functions are implemented in one of these lower-level programming languages.

Beyond R

```
# inspect source code
sum
```

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

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

SQL basics

SQL basics

> *Are tomorrow's bigger computers going to solve the problem? For some people, yes—their data will stay the same size and computers will get big enough to hold it comfortably. For other people it will only get worse—more powerful computers means extraordinarily larger datasets. If you are likely to be in this latter group, you might want to get used to working with databases now.*

(Burns 2011)

Structured Query Language

- Traditionally only encountered in the context of *relational database management systems*.
- Now also used to query data from *data warehouse* systems (e.g. Amazon Redshift) and even to query massive amounts (terabytes or even petabytes) of data stored in *data lakes* (e.g., Amazon Athena).
- Bread-and-butter tool to query structured data.

What is it for? Illustration

Query/prepare/join data for analysis. Let's create a point of reference in R.

What is it for? Illustration

```
## Warning in fun(libname, pkgname): couldn't connect to display ":0"
```

In SQL, we can get exactly the same, with the following command.

```
SELECT
strftime('%Y', `date`)  AS year,
AVG(unemploy) AS average_unemploy
FROM econ
WHERE "1968-01-01"<=`date`
GROUP BY year LIMIT 6;
```

Table 2: 6 records

year	average_unemploy
1968	2797.417
1969	2830.167
1970	4127.333
1971	5021.667
1972	4875.833
1973	4359.333

Key take-away: in R, we instruct the computer what to do in order to get the table we want. In SQL we instruct the computer what table we want.

The Structured Query Language (SQL) has become a bread and butter tool for data analysts and data scientists due to its

Getting started with SQLite

- SQLite
  - Free, full-featured SQL database engine.
  - Widely used across platforms.
  - Typically pre-installed on Windows/MacOSX.

Set up an SQLite database

In this first code example, we set up an SQLite database using the command line. Open the terminal and switch to the `data` directory.

```
cd data
```

Set up an SQLite database: initiate database

Start up SQLite, create a new (empty) database called `mydb.sqlite` and connect to the newly created database.

```
sqlite3 mydb.sqlite
```

Set up an SQLite database: create table

We create a new table called `econ` based on the same data used in the R example above.

```
CREATE TABLE econ(
"date" DATE,
"pce" REAL,
"pop" REAL,
"psavert" REAL,
"uempmed" REAL,
"unemploy" INTEGER
);
```

Set up an SQLite database: import data to table

```
-- prepare import
.mode csv
-- import data from csv
.import --skip 1 economics.csv econ
```

Set up an SQLite database: check db tables

Now we can have a look at the new database table in SQLite. `.tables` shows that we now have one table called `econ` in our database and `.schema` displays the structure of the new `econ` table.

```
.tables

# econ

.schema econ

# CREATE TABLE econ(
# "date" DATE,
# "pce" REAL,
# "pop" REAL,
# "psavert" REAL,
# "uempmed" REAL,
# "unemploy" INTEGER
# );
```

Simple queries

Set output mode

```
.header on
```

```
.mode columns
```

Select all (*) variable values of the observation of January 1968.

```
select * from econ where date = '1968-01-01';
```

Table 3: 1 records

date	pce	pop	psavert	uempmed	unemploy
1968-01-01	531.5	199808	11.7	5.1	2878

Simple queries

Select all dates and unemployment values of observations with more than 15 million unemployed, ordered by date.

```
select date,
unemploy from econ
where unemploy > 15000
order by date;
```

Table 4: 9 records

date	unemploy
2009-09-01	15009
2009-10-01	15352
2009-11-01	15219
2009-12-01	15098
2010-01-01	15046
2010-02-01	15113
2010-03-01	15202
2010-04-01	15325
2010-11-01	15081

Joins

Let's extend the previous example by importing an additional table to our `mydb.sqlite`. The additional data is stored in the file `inflation.csv` and contains information on the US yearly inflation rate measured in percent.

```
-- Create the new table
CREATE TABLE inflation(
"date" DATE,
"inflation_percent" REAL
);

-- prepare import
.mode csv
-- import data from csv
.import --skip 1 inflation.csv inflation
-- switch back to column mode
.mode columns
```

Joins: example

- Aim: get a table that serves as basis for a Phillips curve plot, with yearly observations and the variables `year`, `average_unemp_percent`, and `inflation_percent`.
- `econ` contains monthly observations, while `inflation` contains yearly observations.
- We can combine the two data sets at the level of years.

Joins: R as a reference point

```r
# import data
econ <- read.csv("data/economics.csv")
inflation <- read.csv("data/inflation.csv")

# prepare variable to match observations
econ$year <- lubridate::year(econ$date)
inflation$year <- lubridate::year(inflation$date)

# create final output
years <- unique(econ$year)
averages <- sapply(years, FUN = function(x) {
        mean(econ[econ$year==x,"unemploy"]/econ[econ$year==x,"pop"])*100

} )
unemp <- data.frame(year=years,
                    average_unemp_percent=averages)

# combine via the year column
# keep all rows of econ
output<- merge(unemp, inflation[, c("year", "inflation_percent")], by="year")


# inspect output
head(output)
```

```
##   year average_unemp_percent inflation_percent
## 1 1967              1.512179          2.772786
## 2 1968              1.394202          4.271796
## 3 1969              1.396464          5.462386
## 4 1970              2.012547          5.838255
## 5 1971              2.418970          4.292767
## 6 1972              2.323808          3.272278
```

The same table can be created in SQLite.

```
SELECT
strftime('%Y', econ.date)  AS year,
AVG(unemploy/pop)*100 AS average_unemp_percent,
inflation_percent
FROM econ INNER JOIN inflation ON year = strftime('%Y', inflation.date)
GROUP BY year
```

Table 5: Displaying records 1 - 6

year	average_unemp_percent	inflation_percent
1967	1.512179	2.772786
1968	1.394202	4.271796
1969	1.396464	5.462386
1970	2.012547	5.838255
1971	2.418970	4.292767
1972	2.323808	3.272278

Exit SQLite

When done working with the database, we can exit SQLite with the `.quit` command.

References

Burns, Patrick. 2011. *The r Inferno*. Lulu Press, Inc. https://www.burns-stat.com/pages/Tutor/R_inferno.pdf.