Big Data Analytics

Lecture 3: Computation and Memory II

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

05/03/2020

1 Resource allocation

When optimizing the performance of an analytics program processing large amounts of data, it is useful to differentiate between the efficient allocation of computational (CPU) power, and the allocation of RAM (and mass storage).^I[n many data analysis tasks the two are, of course, intertwined. However, keeping both aspects in mind when optimizing an analytics program helps to choose the right tools.] Below, we will look at both aspects in turn.

1.1 Case study: Parallel processing

In this example, we estimate a simple regression model that aims to assess racial discrimination in the context of police stops.[1] The example is based on the 'Minneapolis Police Department 2017 Stop Dataset', containing data on nearly all stops made by the Minneapolis Police Department for the year 2017.

We start with importing the data into R.

```
stopdata <- read.csv("https://vincentarelbundock.github.io/Rdatasets/csv/carData/MplsStops.csv")
```

We specify a simple linear probability model that aims to test whether a stopped person identified as 'white' is less likely to have her vehicle searched when stopped by the police. In order to take into account level-differences between different police precincts, we add precinct-indicators to the regression specification

First, let's remove observations with missing entries (`NA`) and code our main explanatory variable and the dependent variable.

```
# remove incomplete obs
stopdata <- na.omit(stopdata)
# code dependent var
stopdata$vsearch <- 0
stopdata$vsearch[stopdata$vehicleSearch=="YES"] <- 1
# code explanatory var
stopdata$white <- 0
stopdata$white[stopdata$race=="White"] <- 1
```

We specify our baseline model as follows.

```
model <- vsearch ~ white + factor(policePrecinct)
```

And estimate the linear probability model via OLS (the `lm` function).

```
fit <- lm(model, stopdata)
summary(fit)
```

[1] Note that this example aims to illustrate a point about computation in an applied econometrics context. It does not make any argument about identification or the broader research question whatsoever.

```
##
## Call:
## lm(formula = model, data = stopdata)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.13937 -0.06329 -0.05473 -0.04227  0.97729
##
## Coefficients:
##                           Estimate Std. Error t value Pr(>|t|)
## (Intercept)               0.054733   0.005154  10.619  < 2e-16 ***
## white                    -0.019553   0.004465  -4.380 1.19e-05 ***
## factor(policePrecinct)2  0.008556   0.006757   1.266   0.2054
## factor(policePrecinct)3  0.003409   0.006483   0.526   0.5990
## factor(policePrecinct)4  0.084639   0.006232  13.582  < 2e-16 ***
## factor(policePrecinct)5 -0.012465   0.006371  -1.956   0.0504 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.254 on 19078 degrees of freedom
## Multiple R-squared:  0.02502,    Adjusted R-squared:  0.02476
## F-statistic: 97.92 on 5 and 19078 DF,  p-value: < 2.2e-16
```

A potential problem with this approach (and there might be many more in this simple example) is that observations stemming from different police precincts might be correlated over time. If that is the case, we likely underestimate the coefficient's standard errors. There is a standard approach to compute estimates for so-called *cluster-robust* standard errors which would take the problem of correlation over time within clusters into consideration (and deliver a more conservative estimate of the SEs). However this approach only works well if the number of clusters in the data is roughly 50 or more. Here we only have 5.

The alternative approach is to compute bootstrapped standard errors. That is, we apply the bootstrap resampling procedure at the cluster level. Specifically, we draw B samples (with replacement), estimate and record for each bootstrap-sample the coefficient vector, and then estimate SE_{boot} based on the standard deviation of all respective estimated coefficient values.

```r
# load packages
library(data.table)
# set the 'seed' for random numbers (makes the example reproducible)
set.seed(2)

# set number of bootstrap iterations
B <- 10
# get selection of precincts
precincts <- unique(stopdata$policePrecinct)
# container for coefficients
boot_coefs <- matrix(NA, nrow = B, ncol = 2)
# draw bootstrap samples, estimate model for each sample
for (i in 1:B) {

     # draw sample of precincts (cluster level)
     precincts_i <- sample(precincts, size = 5, replace = TRUE)
     # get observations
     bs_i <- lapply(precincts_i, function(x) stopdata[stopdata$policePrecinct==x,])
     bs_i <- rbindlist(bs_i)
```

```r
    # estimate model and record coefficients
    boot_coefs[i,] <- coef(lm(model, bs_i))[1:2] # ignore FE-coefficients
}
```

Finally, let's compute SE_{boot}.

```r
se_boot <- apply(boot_coefs,
                 MARGIN = 2,
                 FUN = sd)
se_boot
```

```
## [1] 0.004042725 0.004689611
```

Note that even with a very small B, computing SE_{boot} takes up some time to compute. When setting B to over 500, computation time will be substantial. Also note that running this code does hardly use up more memory then the very simple approach without bootstrapping (after all, in each bootstrap iteration the data set used to estimate the model is approximately the same size as the original data set). There is little we can do to improve the script's performance regarding memory. However we can tell R how to allocate CPU resources more efficiently to handle that many regression estimates.

Particularly, we can make use of the fact that most modern computing environments (such as a laptop) have CPUs with several *cores*. We can exploit this fact by instructing the computer to run the computations *in parallel* (simultaneously computing on several cores). The following code is a parallel implementation of our bootstrap procedure which does exactly that.

```r
# install.packages("doSNOW", "parallel")
# load packages for parallel processing
library(doSNOW)

# get the number of cores available
ncores <- parallel::detectCores()
# set cores for parallel processing
ctemp <- makeCluster(ncores) #
registerDoSNOW(ctemp)


# set number of bootstrap iterations
B <- 10
# get selection of precincts
precincts <- unique(stopdata$policePrecinct)
# container for coefficients
boot_coefs <- matrix(NA, nrow = B, ncol = 2)

# bootstrapping in parallel
boot_coefs <-
     foreach(i = 1:B, .combine = rbind, .packages="data.table") %dopar% {

          # draw sample of precincts (cluster level)
          precincts_i <- sample(precincts, size = 5, replace = TRUE)
          # get observations
          bs_i <- lapply(precincts_i, function(x) stopdata[stopdata$policePrecinct==x,])
          bs_i <- rbindlist(bs_i)

          # estimate model and record coefficients
          coef(lm(model, bs_i))[1:2] # ignore FE-coefficients

```

```r
    }


# be a good citizen and stop the snow clusters
stopCluster(cl = ctemp)
```

As a last step, we compute again SE_{boot}.

```r
se_boot <- apply(boot_coefs,
                 MARGIN = 2,
                 FUN = sd)
se_boot
```

```
## (Intercept)       white
## 0.003572022 0.004213160
```

1.2 Case study: Memory allocation

Consider the first steps of a data pipeline in R. The first part of our script to import and clean the data looks as follows.

```r
###########################################################
# Big Data Statistics: Flights data import and preparation
#
# U. Matter, January 2019
###########################################################

# SET UP -----------------

# fix variables
DATA_PATH <- "materials/data/flights.csv"

# DATA IMPORT ----------------
flights <- read.csv(DATA_PATH)

# DATA PREPARATION --------
flights <- flights[,-1:-3]
```

When running this script, we notice that some of the steps take a while. Moreover, while none of these steps obviously involves a lot of computation (such as a matrix inversion or numerical optimization), it quite likely involves memory allocation. We first read data into RAM (allocated to R by our operating system). It turns out that there are different ways to allocate RAM when reading data from a CSV file. Depending on the amount of data to be read in, one or the other approach might be faster. We first investigate the RAM allocation in R with `mem_change()` and `mem_used()`.

```r
# SET UP -----------------

# fix variables
DATA_PATH <- "../data/flights.csv"
# load packages
library(pryr)


# check how much memory is used by R (overall)
mem_used()
```

```
## 1.08 GB
```

```r
# check the change in memory due to each step

# DATA IMPORT ----------------
mem_change(flights <- read.csv(DATA_PATH))
```

```
## -6.64 MB
```

```r
# DATA PREPARATION --------
flights <- flights[,-1:-3]

# check how much memory is used by R now
mem_used()
```

```
## 1.07 GB
```

The last result is kind of interesting. The object `flights` must have been larger right after importing it than at the end of the script. We have thrown out several variables, after all. Why does R still use that much memory? R does by default not 'clean up' memory unless it is really necessary (meaning no more memory is available). In this case, R has still way more memory available from the operating system, thus there is no need to 'collect the garbage' yet. However, we can force R to collect the garbage on the spot with `gc()`. This can be helpful to better keep track of the memory needed by an analytics script.

```r
gc()
```

```
##             used  (Mb) gc trigger   (Mb) limit (Mb)  max used   (Mb)
## Ncells   1024589  54.8    2085852  111.4         NA   2085852  111.4
## Vcells 126556308 965.6  256657211 1958.2      16384 256657211 1958.2
```

Now, let's see how we can improve the performance of this script with regard to memory allocation. Most memory is allocated when importing the file. Obviously, any improvement of the script must still result in importing all the data. However, there are different ways to read data into RAM. `read.csv()` reads all lines of a csv file consecutively. In contrast, `data.table::fread()` first 'maps' the data file into memory and only then actually reads it in line by line. This involves an additional initial step, but the larger the file, the less relevant is this first step with regard to the total time needed to read all the data into memory. By switching on the `verbose` option, we can actually see what `fread` is doing.

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

# DATA IMPORT ----------------
flights <- fread(DATA_PATH, verbose = TRUE)
```

```
##   omp_get_num_procs()            4
##   R_DATATABLE_NUM_PROCS_PERCENT  unset (default 50)
##   R_DATATABLE_NUM_THREADS        unset
##   omp_get_thread_limit()         2147483647
##   omp_get_max_threads()          4
##   OMP_THREAD_LIMIT               unset
##   OMP_NUM_THREADS                unset
##   RestoreAfterFork               true
##   data.table is using 2 threads. See ?setDTthreads.
## Input contains no \n. Taking this to be a filename to open
## [01] Check arguments
##   Using 2 threads (omp_get_max_threads()=4, nth=2)
##   NAstrings = [<<NA>>]
##   None of the NAstrings look like numbers.
```

```
##   show progress = 0
##   0/1 column will be read as integer
## [02] Opening the file
##   Opening file ../data/flights.csv
##   File opened, size = 29.53MB (30960660 bytes).
##   Memory mapped ok
## [03] Detect and skip BOM
## [04] Arrange mmap to be \0 terminated
##   \n has been found in the input and different lines can end with different line endings (e.g. mixed
## [05] Skipping initial rows if needed
##   Positioned on line 1 starting: <<year,month,day,dep_time,sched_>>
## [06] Detect separator, quoting rule, and ncolumns
##   Detecting sep automatically ...
##   sep=','  with 100 lines of 19 fields using quote rule 0
##   Detected 19 columns on line 1. This line is either column names or first data row. Line starts as:
##   Quote rule picked = 0
##   fill=false and the most number of columns found is 19
## [07] Detect column types, good nrow estimate and whether first row is column names
##   Number of sampling jump points = 100 because (30960659 bytes from row 1 to eof) / (2 * 8882 jump0s
##   Type codes (jump 000)    : 555555555A5AAA5555A  Quote rule 0
##   Type codes (jump 100)    : 555555555A5AAA5555A  Quote rule 0
##   'header' determined to be true due to column 1 containing a string on row 1 and a lower type (int3
##   =====
##   Sampled 10048 rows (handled \n inside quoted fields) at 101 jump points
##   Bytes from first data row on line 2 to the end of last row: 30960501
##   Line length: mean=92.03 sd=3.56 min=68 max=98
##   Estimated number of rows: 30960501 / 92.03 = 336403
##   Initial alloc = 370043 rows (336403 + 9%) using bytes/max(mean-2*sd,min) clamped between [1.1*estn
##   =====
## [08] Assign column names
## [09] Apply user overrides on column types
##   After 0 type and 0 drop user overrides : 555555555A5AAA5555A
## [10] Allocate memory for the datatable
##   Allocating 19 column slots (19 - 0 dropped) with 370043 rows
## [11] Read the data
##   jumps=[0..30), chunk_size=1032016, total_size=30960501
## Read 336776 rows x 19 columns from 29.53MB (30960660 bytes) file in 00:00.128 wall clock time
## [12] Finalizing the datatable
##   Type counts:
##         14 : int32     '5'
##          5 : string    'A'
## =============================
##    0.001s (  1%) Memory map 0.029GB file
##    0.004s (  3%) sep=',' ncol=19 and header detection
##    0.000s (  0%) Column type detection using 10048 sample rows
##    0.006s (  5%) Allocation of 370043 rows x 19 cols (0.033GB) of which 336776 ( 91%) rows used
##    0.117s ( 91%) Reading 30 chunks (0 swept) of 0.984MB (each chunk 11225 rows) using 2 threads
##    +    0.056s ( 44%) Parse to row-major thread buffers (grown 0 times)
##    +    0.057s ( 45%) Transpose
##    +    0.003s (  3%) Waiting
##    0.000s (  0%) Rereading 0 columns due to out-of-sample type exceptions
##    0.128s        Total
```

Let's put it all together and look at the memory changes and usage. For a fair comparison, we first have to delete `flights` and collect the garbage with `gc()`.

```
# SET UP -----------------

# fix variables
DATA_PATH <- "../data/flights.csv"
# load packages
library(pryr)
library(data.table)

# housekeeping
flights <- NULL
gc()
```

```
##              used  (Mb) gc trigger   (Mb) limit (Mb)  max used   (Mb)
## Ncells   1013604  54.2    2085852  111.4         NA   2085852  111.4
## Vcells 124221376 947.8  256657211 1958.2      16384 256657211 1958.2
```

```
# check the change in memory due to each step

# DATA IMPORT ----------------
mem_change(flights <- fread(DATA_PATH))
```

```
## 36.4 MB
```

Note that `fread()` uses up more memory. By default, `fread` does not parse strings as `factors` (and `read.csv()` does). Storing strings in factors is more memory efficient.

2 Beyond memory

In the previous example we have inspected how RAM is allocated to store objects in the R computing environment. But what if all RAM of our computer is not enough to store all the data we want to analyze?

Modern operating systems have a way to dealing with such a situation. Once all RAM is used up by the currently running programs, the OS allocates parts of the memory back to the hard-disk which then works as *virtual memory*. The following figure illustrates this point.

For example, when we implement an R-script that imports one file after the other into the R environment, ignoring the RAM capacity of our computer, the OS will start *paging* data to the virtual memory. This happens 'under the hood' without explicit instructions by the user. We quite likely notice that the computer slows down a lot when this happens.

While this default usage of virtual memory by the OS is helpful to run several applications at the same time, each taking up a moderate amount of memory, it is not a really useful tool for processing large amounts of data in one application (R). However, the underlying idea of using both RAM and Mass storage simultaneously in order to cope with a lack of memory, is very useful in the context of big data statistics.

Several R packages have been developed that exploit the idea behind virtual memory explicitly for big data analytics. The basic idea behind these packages is to map a data set to the hard disk when loading it into R. The actual data values are stored in chunks on the hard-disk, while the structure/metadata of the data set is loaded into R. See this week's slide set as well as Walkowiak (2016), Chapter 3 for more details and example code.

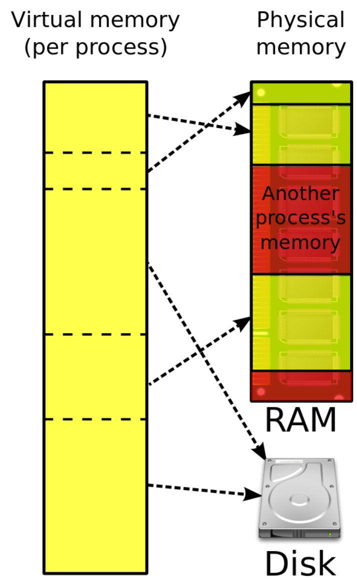

Figure 1: Virtual memory. Figure by Ehamberg (CC BY-SA 3.0).

References

Walkowiak, Simkon. 2016. *Big Data Analytics with R*. Birmingham, UK: PACKT Publishing.