Big Data IT tools

Tutorial 2 — How to optimize a statistical algorithm?

Arthur Katossky & Rémi Pépin

March 2020

Introduction

In this tutorial, we will explore several ways to accelerate a slow statistical procedure.

We start with the following code. The procedure uses a database of monthly declaration by air carriers operating in the US, the *Air Carrier Statistics* database, also known as the *T-100 data bank*. You do not need to bother dowloading the sources, but you will find additionnal information at this link: https://www.transtats.bts.gov/DatabaseInfo.asp?DB_ID=111.

Considering that each observation in this database is a flight (it is not *exactly* the case, but it does not matter here), the overall goal of the script is to compute :

- the overall number of flights per month
- the average number of passengers with a moving-window of 5, then 10 days
- a rolling-regression, explaining payload with seats, passengers, freight, mail, distance

Disclaimer: *the original data have been altered for the purpose of the tutorial ; specifically, a day variable has been created* ex nihilo*.*

1. The procedure to optimize

1.a Libraries

```
library(dplyr)
library(RPostgreSQL)
```

1.b Open a connection with the databse

```
pw <- "YojCqLr3Cnlw6onuzHU3" # Do not change the password !

drv <- dbDriver("PostgreSQL") # loads the PostgreSQL driver.
                              # (Needed to query the data)

# This is the proper connection.
# The same object will be used each time we want to connect to the database.
con <- dbConnect(drv, dbname = "postgres",
                 host = "tp-avion.c5bqsgv9tnea.eu-west-3.rds.amazonaws.com", port = 5432,
                 user = "postgres", password = pw)

rm(pw) # We don't need the password anymore.
```

```r
dbExistsTable(con, "flight") # Check whether the "flight" table exist.
```

```
## [1] TRUE
```

1.c Collect actual data set

```
## # A tibble: 2,000 x 46
##    departures_sche~ departures_perf~ payload seats passengers freight  mail
##    <chr>            <chr>              <dbl> <dbl>      <dbl>   <dbl> <dbl>
##  1 79.00            75.00            3681750 13499      12152   49162 66653
##  2 79.00            77.00             666050  2849       2118      18     0
##  3 79.00            79.00            2733400 11297       9058   62052     0
##  4 79.00            79.00            2543800 11261       8839   12813     0
##  5 79.00            79.00            3428600 13825      12105   40962     0
##  6 79.00            79.00            2733400 11297       7024   10869     0
##  7 79.00            79.00            2733400 11297       9130   21409     0
##  8 79.00            79.00            2733400 11297       9444   32704     0
##  9 79.00            79.00            2733400 11297       8821   13919     0
## 10 79.00            77.00            2664200 11011       9500   10275     0
## # ... with 1,990 more rows, and 39 more variables: distance <dbl>,
## #   ramp_to_ramp <dbl>, air_time <dbl>, unique_carrier <chr>, airline_id <chr>,
## #   unique_carrier_name <chr>, unique_carrier_entity <chr>, region <chr>,
## #   carrier <chr>, carrier_name <chr>, carrier_group <chr>,
## #   carrier_group_new <chr>, origin_airport_id <chr>,
## #   origin_airport_seq_id <chr>, origin_city_market_id <chr>, origin <chr>,
## #   origin_city_name <chr>, origin_state_abr <chr>, origin_state_fips <chr>,
## #   origin_state_nm <chr>, origin_wac <chr>, dest_airport_id <chr>,
## #   dest_airport_seq_id <chr>, dest_city_market_id <chr>, dest <chr>,
## #   dest_city_name <chr>, dest_state_abr <chr>, dest_state_fips <chr>,
## #   dest_state_nm <chr>, dest_wac <chr>, aircraft_group <chr>,
## #   aircraft_type <chr>, aircraft_config <chr>, year <dbl>, quarter <dbl>,
## #   month <dbl>, distance_group <chr>, class <chr>, day <int>
```

```r
# Query all the data of the flight database
FLIGHTS <- dbGetQuery(con, "SELECT * FROM flight")
FLIGHTS
```

1.d Flight number per month

```r
COUNTS <- data.frame(
  year  = integer(),
  month = integer(),
  count = integer()
)

n <- 0

for (i in 1:nrow(FLIGHTS)) {

  flight    <- FLIGHTS[i,]
  selection <- COUNTS$year == flight$year & COUNTS$month == flight$month

  if(nrow(COUNTS[selection,]) > 0) {
```

```
    COUNTS[selection, "count"] <- COUNTS[selection, "count"]+1

  }else{

    n <- n+1

    COUNTS[n, "year"]  <- flight$year
    COUNTS[n, "month"] <- flight$month
    COUNTS[n, "count"] <- 1
  }
}
```

```
COUNTS # just a sample is shown
```

```
##    year month count
## 1  2017     4   194
## 2  2017     5   171
## 3  2017     6   162
## 4  2017     7   125
## 5  2017     8   136
## 6  2017     9   233
## 7  2017    10   179
## 8  2017    11   209
## 9  2017    12   135
## 10 2017     3   156
## 11 2017     1   141
## 12 2017     2   156
## 13 2007     4     1
## 14 2007    10     1
## 15 2018     2     1
```

1.e Rolling average of the number of passengers, with a window of 5 and 10 days

```
PASSENGERS_adjusted <- tibble()

# Passengers per day
PASSENGERS <- FLIGHTS %>%
  group_by (year, month, day) %>%
  summarise (passengers = sum(passengers)) %>%
  arrange(year, month, day)

# 5-days rolling average. Begins at day 5.
for (i in 5:nrow(PASSENGERS)) {

  n <- 0
  for(j in (i-4):i) n <- n + PASSENGERS$passengers[j]

  new_row <- tibble(
    year          = PASSENGERS$year[i],
    month         = PASSENGERS$month[i],
    day           = PASSENGERS$day[i],
    passengers    = PASSENGERS$passengers[i],
    passengers_w5 = n/5
```

```
  )

  PASSENGERS_adjusted <- bind_rows(PASSENGERS_adjusted, new_row)
}

# 10-days rolling average. Begins at day 10.
for (i in 10:nrow(PASSENGERS)) {

  n <- 0
  for(j in (i-9):i) n <- n + PASSENGERS$passengers[j]

  PASSENGERS_adjusted[i-5, "passengers_w10"] <- n/10
}
```

```
PASSENGERS_adjusted
```

```
## # A tibble: 334 x 6
##     year month   day passengers passengers_w5 passengers_w10
##    <dbl> <dbl> <int>      <dbl>         <dbl>          <dbl>
##  1  2017     1     3      26036        20708.             NA
##  2  2017     1     4      34023        27072              NA
##  3  2017     1     5      32456        33003              NA
##  4  2017     1     6      24783        35206.             NA
##  5  2017     1     7      20606        27581.          23698.
##  6  2017     1     8      21574        26688.          27165.
##  7  2017     1     9      36867        27257.          32119.
##  8  2017     1    10      52343        31235.          34902.
##  9  2017     1    11      41605        34599           30625.
## 10  2017     1    12      15956        33669           29436.
## # ... with 324 more rows
```

1.f Rolling regression

A rolling regression is just a regression smoothed over time, exactly as the moving average is a smoothed version of the average. The regression is done on a moving windows of 1000 flights.

This regression explores the relationship between the payload on one hand, and on ther other hand number of seats, the mumber of transported passengers, the quantity of freight and mail transported and the flight distance.

Disclaimer: *do not try to make too much sense of this regression. We came short of a better exemple with the correct complexity balance.*

```
variables <- c("seats", "passengers", "freight", "mail", "distance")
betas     <- numeric()
FLIGHTS   <- FLIGHTS %>% arrange(year, month, day)

for (i in 1000:nrow(FLIGHTS)) {
  X <- data.matrix(FLIGHTS[(i-999):i, variables])
  X <- cbind(intercept=1, X)
  Y <- matrix(FLIGHTS$payload[(i-999):i], nrow = 1000, ncol = 1)
  betas <- cbind(betas, solve(t(X)%*% X) %*% t(X) %*% Y)
}
rownames(betas) <- c("intercept", "seats", "passengers", "freight", "mail", "distance")
betas[,1:10]
```

```
##                    [,1]         [,2]         [,3]         [,4]         [,5]
## intercept  50369.174695 50086.897288 50148.697122 50699.286852 50549.068573
## seats        229.602485   229.742146   229.768485   228.871223   229.054323
## passengers    10.380148    10.146950    10.049008    10.999461    10.693518
## freight        5.039484     5.036768     5.033220     5.029440     5.040137
## mail           1.743371     1.718814     1.724254     1.727622     1.720432
## distance      14.080319    15.039631    15.426716    15.698913    16.048520
##                    [,6]         [,7]         [,8]         [,9]        [,10]
## intercept  50765.945144 51064.260294 50984.069493 51011.753428 51447.506864
## seats        228.961969   229.101806   229.232251   229.292377   229.215462
## passengers    10.774219    10.497688    10.325990    10.190332    10.253251
## freight        5.040749     5.041169     5.041926     5.045085     5.045503
## mail           1.722753     1.727979     1.743635     1.741454     1.752515
## distance      16.004647    16.569165    16.770923    17.121209    16.543536
```

2. Execution

2.1. Run the code.

2.2. Does it take time ? Any idea why ?

2.3. Find a way to make it run quickly

3. Time-complexity and profiling

Now that the code is running, we will try to find how much time each part takes and prioritize our work.

3.1. Factorize the procedure into four functions, that each depends on the input size.

3.2. What is the empirical time complexity of each part ? Is it linear? Produce a time vs. input size plot to help you. *Just focus on a couple of values at the beginning of the input-size range. You may use the `microbenchmark` package or any similar packages.*

3.3. What is the memory complexity of each part ? Is it linear? Produce a max-memory used vs. input size plot. *You may use the `profmem()` function from the `profmem` package.*

3.4. You can perform all that in one step with R-Studio (Profile > Start profiling), the advantage beeing that you get the internal details of each execution.

3.5. What are the parts you would optimize in priority ?

You will find resources about profiling in R and R-Studio here:

- https://adv-r.hadley.nz/perf-measure.html
- https://rstudio.github.io/profvis/index.html
- https://adv-r.hadley.nz/names-values.html
- https://cran.r-project.org/web/packages/profmem/vignettes/profmem.html
- https://www.r-bloggers.com/5-ways-to-measure-running-time-of-r-code

4. Data-transfer optimisation

4.1. In the SQL querry, do we really need to do the following?

```sql
SELECT * FROM flight
```

4.2. Can we transfer some treatments done in R to the database ? If so, update the code. *Google is your friend!*

4.3. Let's focus on the total number of flights by month.

a. Can you optimmize the computation with (faster) R statements? Compare as many solutions as you can think of with `bench::mark()`.
b. Can you perform the computation with SQL statements?
c. Which is faster ? (including data fetch)

5. R-code optimisation

5.1 With `bench::mark()`, compare `v <- cumprod(1:n)` and `v<-1; for(i in 2:n) v[i] <- v[i-1]*i ; v`. Which is faster? Can you spot similar (un)vectorised patterns in our code?

5.2 With `bench::mark()`, compare `v <- rep(1,n)`, `v <- numeric() ; for(i in 1:n) v[i] <- 1 ; v` and `v <- numeric() ; for(i in 1:n) v <- c(1,v)`. Can you spot similar assignment patterns in our code?

6. Algorithmic optimisation

6.1. Think about how rolling averages work. Are we not performing just 5 times or 10 times the computations needed? How can you change the code to optimize it ?

6.2. Why is it a bad idea to recode the linear regression? Using `bench::mark()` compare our regression estimates with `lm()`. Why is it so much faster? *You may want to inspect the code, the real code is really short, most of if is just tests and warnings!*

You will find here the algorithmic complexity of many common mathematical operations:

- https://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations

7. Parallelisation

7.1. Do the regressions depend of each other ? Thus, suggest a way to accelerate this part of the procedure.

7.2 You may use the `doParallel` package to declare and use more than one core. A typical exemples runs like this:

```
library(foreach)              # Parallel for loops
library(parallel)             # Interface between R and multiple cores
library(doParallel)           # Interface between foreach and parallel
detectCores()                 # How many cores are available ?
registerDoParallel(cores=2)   # Number of cores you want to work with
foreach(i=1:10) %dopar% function(i) # Parallel for loop
```

Try to increase one by one the number of cores. Plot the number of cores vs. time. Is the speed-up proportionnal to the number of cores?

7.2 One regression is basically a bunch of matrix operations. How can we theoretically speed that up?

8. Sampling

We do not need to perform the actual complete computation if we are ready to accept some imprecision. But there is a trade-off between computation-time and precision. **For simplification of the problem, let's assume we are only interested with the average number of passenger per flight.**

Let's assume that a sample is taken from a (potentially infinite) population with a (known or knowable) data-generating process. An approach to the measure of uncertainty is **asymptotical inference**.

8.1. In this context, recall the asymptotical distribution of the mean estimator of a random sample of size n, as n tends to infinity. How can you estimate the distribution?

8.2. Modify the database request, so that it returns a *random* sample of size n. You may try first with $n = 40$ then wrap your code in a function for arbitrary n.

8.3 On the same graph, draw violin plots of the estimated distribution for sub-samples of size 40×2^k for $k = 1, ..., 10$ from the downloaded sample. You may want to complete the following code.

```
library(dplyr)
library(tictoc)

x         <- mean(FLIGHTS$passengers) + seq(-200, 200, by=10)
probs     <- tibble() # an empty data frame with dplyr package
estimates <- tibble()

for(k in 1:10){

  n <- 40*2^k

  tic()
  m_hat <- .............. # mean
  s_hat <- .............. # standard deviation
  t <- toc()


  estimates[k, "n"] <- n
  estimates[k, "m"] <- m_hat
  estimates[k, "s"] <- s_hat
  estimates[k, "t"] <- t$toc - t$tic

  probs[k,"x"] <- x
  probs[k,"f"] <- dnorm(x,m_hat,s_hat)
  probs[k,"k"] <- k

}

probs %>% ggplot(aes(x=x, y=f, group=k, color=k)) + geom_violin() + guides(color="none")
```

8.4. Make a plot of $\hat{\sigma}$ against computation time. What would be an acceptable level of precision here?

8.5. In case of more complex estimators (such as a rolling average, or rolling regression) the derivation of confidence bounds is not as straightforward. What else can be used? Could it be computed efficiently?

9 Scaling up: a bigger machine

Create a powerful EC2 instance to run you code (like a m5.2xlarge with 8 cores and 32 GB or Ram) to run your code and see if there is a big difference. To do this you have to

9.1. Create an EC2 instance with ubuntu on it

9.2. Authorize SSH connection to the instance (see : "Etablir une connexion SSH avec votre cluster" in the previous session)

9.3. Connect with SSH with a SSH tunnel from port 8157

9.4. Install and configure foxyproxy (see : "Installer FoxyProxy" in the previous session. It seems that foxyproxy doesn't work well with chrome, please use Firefox)

9.5. Install Rstudio server

```
# Install R
sudo apt-get install r-base
# To install local package
sudo apt-get install gdebi-core
# Donwload Rstudio server
wget https://download2.rstudio.org/server/trusty/amd64/rstudio-server-1.2.5033-amd64.deb
# Install it
sudo gdebi rstudio-server-1.2.5033-amd64.deb
```

9.6. Create a Rstudio user

```
# Make User
sudo useradd -m rstudio-user
sudo passwd rstudio-user
```

9.7. Connect to Rstudio server : https://**public-DNS**:8787 with **public-DNS** the public DNS of the instance

You will find all the steps and explanation in the previous practical session. The main differences are :

- you don't create an EMR cluster, just an EC2 instance
- you have to connect to the EC2 instance with it DNS public address

10 Scaling out: more machines

Create a spark cluster, install Rstudio-server on it (see TP 0) and update the code to use spark function. Run the code.

More documentation : https://spark.rstudio.com