Practical: Efficient Code

BaselRBootcamp 2017

Slides

Here a link to the lecture slides for this session: **LINK** (https://therbootcamp.github.io/_sessions/D4S1_EfficientCode/EfficientCode.html)

Overview

In this practical you'll learn how to write efficient code. By the end of this practical you will know how to:

1. Profile your code to identify critical parts.
2. Make code more efficient.
3. How to do parallel computing.

Benchmarking and profiling functions

Functions to profile your code are:

Function	Package	Description
`proc.time()`	`base`	Returns the time.
`system.time()`	`base`	Runs one expression once and returns elapsed CPU time
`microbenchmark()`	`microbenchmark`	Runs one or many expressions multiple times and returns statistics on elapsed time.
`lineprof(), shine()`	`lineprof`	Evaluates entire scripts. (From Hadley's Github)

Microbenchmark: Example

Small (minimal) chunks of code can conveniently be tested using `microbenchmark()`.

```
# load packages
library(microbenchmark)
library(tibble)

# get data
df <- data.frame('var_1' = rnorm(1000,1),
                 'var_2' = rnorm(1000,1))
tbl <- as.tibble(df)

# microbenchmark pt. 1
microbenchmark(df[['var_1']], df$var_1, tbl$var_1)
```

Profiling: Example

Larger code chunks or even scripts can conveniently be tested using `system.time()` and `lineprof()` from the `lineprof` package.

```
# ---- install and load packages
install.packages('devtools', repos = "https://stat.ethz.ch/CRAN/")
devtools::install_github("hadley/lineprof")
library(lineprof)
library(readr)
library(dplyr)

# ---- define code chunk as function

my_chunkfun <- function(){

  # load data
  data <- read_csv('http://tinyurl.com/titanic-dataset-web')

  # remove first column
  data <- data[,-1]

  # mutate
  data <- data %>%
    mutate(months = Age * 12)

  # select
  test_data <- data %>%
    select(Sex, Age, Survived)

  # multiple regression
  # Survival predicted by Sex, Age, and their interaction
  model <- glm(Survived ~ Sex * Age,
               data = test_data,
               family = 'binomial')

  # evaluate model
  summary(model)

  }

# ---- profiling

# profile using system.time
system.time(my_chunkfun())

# profile using lineprof
profile <- lineprof(my_chunkfun())
shine(profile)
```

Tasks

Microbenchmark

1. Run the microbenchmark example from above. What do you find? Are `tibble` 's fast or slow?

2. Repeat the comparison of `tibble`s and basic `data.frame`s of the first exercise and include now for both data frame types also the `.subset2()` function (don't forget the dot). The function takes two arguments: The first argument is the data frame, the second argument is the column identifyier (index or name). What do you find?

3. Compare the the function `mean()` to the operation composed of its basic ingredients `sum()` and `length()`, i.e., `sum(my_vec) / length(my_vec)`. To do this first create a vector consisting of random numbers using `runif()` (see `?runif`). Then test both ways with `microbenchmark()` What do you find?

4. Test the type of each of `mean()`, `sum()`, `length()`, and `.subset2()` using `typeof()`. What's the fast type?

Profiling

5. Copy the profiling example into a new script file. After installing and loading the `devtools` and `lineprof` packages, run the code under *'define code chunk as function'* and then test the function by runnning it, i.e., execute `my_chunkfun()`. You just defined and executed your first self-created function. Continue by profiling the function using `system.time()` and `lineprof()`. What do you find? What parts of the code are most computationally expensive? Repeat the analysis. Remember R compiles functions after first use.

Speeding up code

6. When speeding up code, the first question should always be whether faster solutions are already out there. In this case there are. Check out the `data.table` package (means: install and load) and use the `fread()` function. Try defining a new function using this function rather than `read_csv()`. Then compare the performance of the two. How much faster is the new relative to the old function (use `system.time()`)?

7. The next step of optimising code is to identify bits that are not necessary. Try to identify a bit that is not entirely necessary, remove it, and evaluate the function's performance again.

8. Next think about whether vectorization may make sense. Find a code chunk that may be written using a vector and vector multiplication and try to implement it. Is there any improvement?

Speeding up code pt 2 (advanced)

9. In 95% of all cases the above steps will produce efficient-enough code. Sometimes, however, one is interested in even faster code execution. This is particularly the case when dealing with large data sets. One reason for this is that run time can be a super-linear function of data size, i.e., a twice as large data sets requires more than twice as much computation time. To see this, program a simple function that identifies the smallest value of a vector (passed on as the argument) using `sort()` and selecting the first element of the sorted vector, i.e., `sort(my_vector)[1]`. Then feed it random vectors (using `runif()`) of length either 1e5+, 1e+6, 1e+7, 1e+8, and 1e+9 and evalute the computation time (using `system.time()`). Does it increase by more or less than 10 times each step? You want to repeat this a couple of times.

Excursion: How to program functions? Functions are always defined as this `my_fun <- function(){}`. Within the parentheses you define the names of the arguments, e.g., `function(variable_1, variable_2)`. Within the curly brackets you define the function's expression, i.e., what it's supposed to do. This could be for instance `variable_1 + variable_2`, when the goal is to compute the element-wise sum of two vectors. By calling (executing) the function, the argument names inside the functions expression, i.e., `variable_1` and `variable_2` will then be replaced by the objects that were provided (passed on) as arguments. That is, if the function is provided with two vectors `my_vec_1` and `my_vec_2`, i.e., `my_fun(my_vec_1, my_vec_2)`, then

the function will compute the sum of these two vectors. This requires, of course, that the provided arguments fit to whatever is done with them inside the function. In this case, the objects are thus required to be `numerical` and cannot be, e.g., of type `character`. The full function definition in this case is `my_fun <- function(variable_1, variable_2) {variable_1 + variable_2}`. After its been defined you would could call it using `my_fun(object_1, object_2)`.

9. When large datasets need to be processed and speed is of the essence, it can be extremely useful to rely on multi-threaded, parallel computation. That is to run a task on multiple processors in parallel. To do this, R has relatively convenient packages, in particular, the `paralell` package, which has recently been included in the standard R library. To use parallel execution four things need to be done. (1) The data need to be split into separate jobs. For instance, a vector may be split into a list containing 100 separate pieces. (2) A function needs to be defined that performs the desired operations on a single job (one piece of the vector). (3) A cluster of workers needs to be created using, e.g., `makeCluster`. (3) The jobs and the function need to be combined in one of `parallel` 's functions. Those functions manage the passing on of jobs to individual workers in the cluster and the retrieval of the results of their computations. This particular style of programming is also known as functional programming. Now try to run the code below, which implements the function from above in this 'divide and conquer'-manner. If it runs and you understood what it does, compare its execution time to its non-parallel twin above.

```
library(parallel)

# define data
my_vec <- runif(1e+8)

# create jobs
# matrix splits data in 100 columns
# as.data.frame and as.list transform it to a list with 100 vectors
jobs <- as.list(as.data.frame(matrix(my_vec, ncol = 100)))

# define function
my_fun <- function(x) sort(x)[1]

# create a cluster with as many workers as cores
cl <- makeCluster(detectCores())

# apply function to jobs using a load balanced (LB) handler
result <- clusterApplyLB(cl, jobs, my_fun)

# flatten result and apply my_fun one last time
my_fun(unlist(result))
```

Additional reading

- For more details check out check out Hadley Wickham's Advanced R (http://adv-r.had.co.nz/).
- For more on parallel computing see Parallel R (https://www.amazon.com/dp/B005Z29QT4/ref=cm_sw_su_dp) by McCallum and Weston.