Big Data Statistics for R and Python

Lecture 1: Introduction

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

25/02/2019

1 Example of Computation Time and Memory Allocation

1.1 Preparation

We first read the `economics` data set into R and extend it by duplicating its rows in order to get a slightly larger data set (this step can easily be changed to create a very large data set).

```r
# read dataset into R
economics <- read.csv("../data/economics.csv")
# have a look at the data
head(economics, 2)
```

```
##         date   pce    pop psavert uempmed unemploy
## 1 1967-07-01 507.4 198712    12.5     4.5     2944
## 2 1967-08-01 510.5 198911    12.5     4.7     2945
```

```r
# create a 'large' dataset out of this
for (i in 1:3) {
     economics <- rbind(economics, economics)
}
dim(economics)
```

```
## [1] 4592    6
```

1.2 Naïve Approach (ignorant of R)

The goal of this code example is to compute the real personal consumption expenditures, assuming that `pce` in the `economics` data set provides the nominal personal consumption expenditures. Thus, we have to use R to divide each value of `pce` by a deflator `1.05`.

The first approach we take is based on a simple `for`-loop, where in each iteration one element in `pce` is divided by the `deflator` and the resulting value is stored as a new element in the vector `pce_real`.

```r
# Naïve approach (ignorant of R)
deflator <- 1.05 # define deflator
# iterate through each observation
pce_real <- c()
n_obs <- length(economics$pce)
for (i in 1:n_obs) {
  pce_real <- c(pce_real, economics$pce[i]/deflator)
}

# look at the result
head(pce_real, 2)
```

```
## [1] 483.2381 486.1905
```

How long does it take?

```
# Naïve approach (ignorant of R)
deflator <- 1.05 # define deflator
# iterate through each observation
pce_real <- list()
n_obs <- length(economics$pce)
time_elapsed <-
     system.time(
         for (i in 1:n_obs) {
              pce_real <- c(pce_real, economics$pce[i]/deflator)
})

time_elapsed
```

```
##    user  system elapsed
##   0.080   0.000   0.081
```

Assuming a linear time algorithm ($O(n)$), we need that much time for one additional row of data:

```
time_per_row <- time_elapsed[3]/n_obs
time_per_row
```

```
##      elapsed
## 1.763937e-05
```

If we deal with big data, say 100 million rows, that is

```
# in seconds
(time_per_row*100^4)
```

```
##  elapsed
## 1763.937
```

```
# in minutes
(time_per_row*100^4)/60
```

```
##  elapsed
## 29.39895
```

```
# in hours
(time_per_row*100^4)/60^2
```

```
##   elapsed
## 0.4899826
```

Can we improve this?

1.2.1 Improvement 1: Pre-allocation of memory

In the naïve approach taken above, each iteration of the loop causes R to re-allocate memory because the number of elements in vector `pce_element` is changing. In simple terms, this means that R needs to execute more steps in each iteration. We can improve this with a simple trick by initiating the vector in the right size to begin with (filled with `NA` values).

```
# Improve memory allocation (still somewhat ignorant of R)
deflator <- 1.05 # define deflator
n_obs <- length(economics$pce)
```

```
# allocate memory beforehand
# Initiate the vector in the right size
pce_real <- rep(NA, n_obs)
# iterate through each observation
time_elapsed <-
     system.time(
         for (i in 1:n_obs) {
              pce_real[i] <- economics$pce[i]/deflator
})
```

Let's see if this helped to make the code faster.

```
time_per_row <- time_elapsed[3]/n_obs
time_per_row
```

```
##      elapsed
## 3.702091e-06
```

Again, we can extrapolate (approximately) the computation time, assuming the data set had millions of rows.

```
# in seconds
(time_per_row*100^4)
```

```
##  elapsed
## 370.2091
```

```
# in minutes
(time_per_row*100^4)/60
```

```
##  elapsed
## 6.170151
```

```
# in hours
(time_per_row*100^4)/60^2
```

```
##   elapsed
## 0.1028358
```

This looks much better, but we can do even better.

1.2.2 Improvement 2: Exploit vectorization.

In this approach, we exploit 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 a vectorized operation is implemented 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 through the same 'preparatory' steps again and again in each iteration, this is time-intensive.

In this example, we specifically exploit that the division operator `/` is actually a vectorized function. Thus, the division by our `deflator` is applied to each element of `economics$pce`.

```
# Do it 'the R wqy'
deflator <- 1.05 # define deflator
# Exploit R's vectorization!
time_elapsed <-
     system.time(
     pce_real <- economics$pce/deflator
```

```
          )
# same result
head(pce_real, 2)
```

```
## [1] 483.2381 486.1905
```

Now this is much faster. In fact, `system.time()` is not precise enough to capture the time elapsed…

```
time_per_row <- time_elapsed[3]/n_obs

# in seconds
(time_per_row*100^4)
```

```
## elapsed
##  21.777
```

```
# in minutes
(time_per_row*100^4)/60
```

```
##   elapsed
## 0.3629501
```

```
# in hours
(time_per_row*100^4)/60^2
```

```
##     elapsed
## 0.006049168
```

1.3 What do we learn from this?

1. How R allocates and deallocates memory can have a substantial effect on computation time.
    - (Particularly, if we deal with a large data set!)
2. In what way the computation is implemented can matter a lot for the time elapsed.
    - (For example, loops vs. vectorization/apply)