Lecture 2: Advanced R

Zhentao Shi

March 18, 2020

Advanced R

Introduction

In this lecture, we will talk about efficient computation in R.

- R is a vector-oriented language. In most cases, vectorization speeds up computation.
- We turn to more CPUs for parallel execution to save time if there is no more room to optimize the code to improve the speed.
- Clusters are accessed remotely. Communicating with a remote cluster is different from operating a local machine.

Vectorization

Despite mathematical equivalence, various ways of calculation can perform distinctively in terms of computational speed.

Does computational speed matter? For a job that takes less than a minutes, the time difference is not a big deal. But sometimes economic problems can be clumsy. For structural estimation commonly seen in industrial organization, a single estimation can take up to a week. In econometrics, other computational intensive procedures include bootstrap, simulated maximum likelihood and simulated method of moments. Even if a single execution does not take much time, repeating such a procedure for thousands of replications will consume a non-trivial duration. Moreover, machine learning methods that crunch big data usually involve tuning parameters, so the same procedure must be carried out at each point of a grid of tuning paramters. For those problems, code optimization is essential.

Of course, optimizing code takes human time. It is a balance of human time and machine time.

Example

The heteroskedastic-robust variance for the OLS regression is

$$(X'X)^{-1}X'\hat{e}\hat{e}'X(X'X)^{-1}$$

The difficult part is $X'\hat{e}\hat{e}'X = \sum_{i=1}^{n} \hat{e}_i^2 x_i x_i'$. There are at least 4 mathematically equivalent ways to compute this term.

1. literally sum $\hat{e}_i^2 x_i x_i'$ over $i = 1, \ldots, n$ one by one.
2. $X'\mathrm{diag}(\hat{e}^2)X$, with a dense central matrix.
3. $X'\mathrm{diag}(\hat{e}^2)X$, with a sparse central matrix.
4. Do cross product to `X*e_hat`. It takes advantage of the element-by-element operation in R.

We first generate the data of binary response and regressors. Due to the discrete nature of the dependent variable, the error term in the linear probability model is heteroskedastic. It is necessary to use the heteroskedastic-robust variance to consistently estimate the asymptotic variance of the OLS estimator. The code chunk below estimates the coefficients and obtains the residual.

```r
lpm <- function(n) {
  # estimation in a linear probability model

  # set the parameters
  b0 <- matrix(c(-1, 1), nrow = 2)

  # generate the data
  e <- rnorm(n)
  X <- cbind(1, rnorm(n)) # you can try this line. See what is the difference.
  Y <- (X %*% b0 + e >= 0)
  # note that in this regression b0 is not converge to b0 because the model is changed.

  # OLS estimation
  bhat <- solve(t(X) %*% X, t(X) %*% Y)

  # calculate the t-value
  bhat2 <- bhat[2] # parameter we want to test
  e_hat <- Y - X %*% bhat
  return(list(X = X, e_hat = as.vector(e_hat)))
}
```

We run the 4 estimators for the same data, and compare the time.

```r
# an example of robust variance matrix.
# compare the implementation via matrix, Matrix (package) and vecteroization.
library(Matrix)

# n = 5000; Rep = 10; # Matrix is quick, matrix is slow, adding is OK
n <- 50
Rep <- 1000
# Matrix is slow,  matrix is quick, adding is OK

for (opt in 1:4) {
  pts0 <- Sys.time()

  for (iter in 1:Rep) {
    set.seed(iter) # to make sure that the data used
    # different estimation methods are the same

    data.Xe <- lpm(n)
    X <- data.Xe$X
    e_hat <- data.Xe$e_hat

    XXe2 <- matrix(0, nrow = 2, ncol = 2)

    if (opt == 1) {
      for (i in 1:n) {
        XXe2 <- XXe2 + e_hat[i]^2 * X[i, ] %*% t(X[i, ])
      }
    } else if (opt == 2) { # the vectorized version with dense matrix
      e_hat2_M <- matrix(0, nrow = n, ncol = n)
      diag(e_hat2_M) <- e_hat^2
      XXe2 <- t(X) %*% e_hat2_M %*% X
    } else if (opt == 3) { # the vectorized version with sparse matrix
      e_hat2_M <- Matrix(0, ncol = n, nrow = n)
      diag(e_hat2_M) <- e_hat^2
      XXe2 <- t(X) %*% e_hat2_M %*% X
    } else if (opt == 4) { # the best vectorization method. No waste
      Xe <- X * e_hat
      XXe2 <- t(Xe) %*% Xe
    }

    XX_inv <- solve(t(X) %*% X)
    sig_B <- XX_inv %*% XXe2 %*% XX_inv
  }
  cat("n = ", n, ", Rep = ", Rep, ", opt = ", opt, ", time = ", Sys.time() - pts0, "\n")
}
```

```
## n =  50 , Rep =  1000 , opt =  1 , time =  0.6472809
## n =  50 , Rep =  1000 , opt =  2 , time =  0.158607
## n =  50 , Rep =  1000 , opt =  3 , time =  1.273492
## n =  50 , Rep =  1000 , opt =  4 , time =  0.09774303
```

We clearly see the difference in running time, though the 4 methods are mathematically the same. When n is small, `matrix` is fast and `Matrix` is slow; the vectorized version is the fastest. When n is big, `matrix` is slow and `Matrix` is fast; the vectorized version is still the fastest.

Efficient Loop

R was the heir of S, an old language. R evolves with packages that are designed to adapt to new big data environement. Many examples can be found in Wickham and Grolemund (2016). Here we introduce plyr.

In standard `for` loops, we have to do a lot of housekeeping work. Hadley Wickham's `plyr` simplifies the job and facilitates parallelization.

Example

Here we calculate the empirical coverage probability of a Poisson distribution of degrees of freedom 2. We first write a user-defined function `CI` for confidence interval, which was used in the last lecture.

```r
CI <- function(x) { # construct confidence interval
  # x is a vector of random variables

  n <- length(x)
  mu <- mean(x)
  sig <- sd(x)
  upper <- mu + 1.96 / sqrt(n) * sig
  lower <- mu - 1.96 / sqrt(n) * sig
  return(list(lower = lower, upper = upper))
}
```

This is a standard `for` loop.

```r
Rep <- 100000
sample_size <- 1000
mu <- 2

# append a new outcome after each loop
pts0 <- Sys.time() # check time
for (i in 1:Rep) {
  x <- rpois(sample_size, mu)
  bounds <- CI(x)
  out_i <- ((bounds$lower <= mu) & (mu <= bounds$upper))
  if (i == 1) {
    out <- out_i
  } else {
    out <- c(out, out_i)
  }
}
cat("empirical coverage probability = ", mean(out), "\n") # empirical size
```

```
## empirical coverage probability =  0.94896
```

```r
pts1 <- Sys.time() - pts0 # check time elapse
print(pts1)
```

```
## Time difference of 21.2908 secs
```

```r
# pre-define a container
out <- rep(0, Rep)
pts0 <- Sys.time() # check time
for (i in 1:Rep) {
  x <- rpois(sample_size, mu)
  bounds <- CI(x)
  out[i] <- ((bounds$lower <= mu) & (mu <= bounds$upper))
}
cat("empirical coverage probability = ", mean(out), "\n") # empirical size
```

```
## empirical coverage probability =  0.94989
```

```r
pts1 <- Sys.time() - pts0 # check time elapse
print(pts1)
```

```
## Time difference of 13.58924 secs
```

Pay attention to the line `out = rep(0, Rep)`. It *pre-defines* a vector `out` to be filled by `out[i]`

`= ( ( bounds$lower <= mu  ) & (mu <= bounds$upper) )`. The computer opens a continuous patch of memory for the vector `out`. When new result comes in, the old element is replaced. If we do not pre-define `out` but append one more element in each loop, the length of `out` will change in each replication and every time a new patch of memory will be assigned to store it. The latter approach will spend much more time just to locate the vector in the memory.

`out` is the result container. In a `for` loop, we pre-define a container, and replace the elements of the container in each loop by explicitly calling the index.

In contrast, a `plyr` loop saves the house keeping chores, and makes it easier to parallelize. In the example below, we encapsulate the chunk in the `for` loop as a new function `capture`, and run the replication via `__ply`. `__ply` is a family of functions. `ldply` here means that the input is a list (`l`) and the output is a data frame (`d`) .

```r
library(plyr)

capture <- function(i) {
  x <- rpois(sample_size, mu)
  bounds <- CI(x)
  return((bounds$lower <= mu) & (mu <= bounds$upper))
}

pts0 <- Sys.time() # check time
out <- ldply(.data = 1:Rep, .fun = capture)
cat("empirical coverage probability = ", mean(out$V1), "\n") # empirical size
```

```
## empirical coverage probability =  0.94893
```

```r
pts1 <- Sys.time() - pts0 # check time elapse
print(pts1)
```

```
## Time difference of 12.01922 secs
```

This example is so simple that the advantage of `plyr` is not dramatic. The difference in coding will be noticeable in complex problems with big data frames. In terms of speed, `plyr` does not run much faster than a `for` loop. They are of similar performance. Parallel computing will be our next topic. It is quite easy to implement parallel execution with `plyr`—we just need to change one argument in the function.

Parallel Computing

Parallel computing becomes essential when the data size is beyond the storage of a single computer, for example Li et al. (2018). Here we explore the speed gain of parallel computing on a multicore machine.

The packages `foreach` and `doParallel` are useful for parallel computing. Below is the basic structure. `registerDoParallel(number)` prepares a few CPU cores to accept incoming jobs.

```r
library(plyr)
library(foreach)
library(doParallel)
```

```r
registerDoParallel(a_number) # opens specified number of CPUs

out <- foreach(icount(Rep), .combine = option) %dopar% {
  my_expressions
}
```

If we have two CPUs running simultaneously, in theory we can cut the time to a half of that on a single CPU. Is that what happening in practice?

Example

Compare the speed of a parallel loop and a single-core sequential loop.

```r
library(foreach)
library(doParallel)

registerDoParallel(2) # open 2 CPUs

pts0 <- Sys.time() # check time

out <- foreach(1:Rep, .combine = c) %dopar% {
  capture(i)
}

# out = ldply(.data = 1:Rep, .fun = capture, .parallel = T,
#            .paropts = list(.export = ls(envir=globalenv() )) )
cat("empirical coverage probability = ", mean(out), "\n") # empirical size
```

```
## empirical coverage probability =  0.94937
```

```r
pts1 <- Sys.time() - pts0 # check time elapse
print(pts1)
```

```
## Time difference of 37.25178 secs
```

```r
print(pts1)
```

```
## Time difference of 37.25178 secs
```

Surprisingly, the above code block of parallel computing runs even more slowly. It is because the task in each loop can be done in very short time. In contrast, the code chunk below will tell a different story. There the time in each loop is non-trivial, and then parallelism dominates the overhead of the CPU communication. The only difference between the two implementations below is that the first uses `%dopar%` and the latter uses `%do%`.

```r
Rep <- 200
sample_size <- 2000000

registerDoParallel(8) # change the number of open CPUs according to
# the specification of your computer

pts0 <- Sys.time() # check time
```

```r
out <- foreach(icount(Rep), .combine = c) %dopar% {
  capture()
}

cat("empirical coverage probability = ", mean(out), "\n") # empirical size
```

```
## empirical coverage probability =  0.94
```

```r
pts1 <- Sys.time() - pts0 # check time elapse
print(pts1)
```

```
## Time difference of 5.660641 secs
```

```r
pts0 <- Sys.time()
out <- foreach(icount(Rep), .combine = c) %do% {
  capture()
}
cat("empirical coverage probability = ", mean(out), "\n") # empirical size
```

```
## empirical coverage probability =  0.965
```

```r
pts1 <- Sys.time() - pts0 # check time elapse
print(pts1)
```

```
## Time difference of 20.59729 secs
```

Remote Computing

Investing money from our own pocket to an extremely powerful laptop to conduct heavy-lifting computational work is unnecessary. (i) We do not run these long jobs every day, it is more cost efficient to share a workhorse. (ii) We cannot keep our laptop always on when we move it around. The right solution is remote computing on a server.

No fundamental difference lies between local and remote computing. We prepare the data and code, open a shell for communication, run the code, and collect the results. One potential obstacle is dealing with a command-line-based operation system. Such command line tools is the norm of life two or three decades ago, but today we mostly work in a graphic operating system like Windows or OSX. For Windows users (I am one of them), I recommend PuTTY, a shell, and WinSCP, a graphic interface for input and output.

Most servers in the world are running Unix/Linux operation system. Here are a few commands for basic operations.

- mkdir
- cd
- copy
- top
- screen
- ssh user@address
- start a program

Our department's computation infrastructure has been improving. A server dedicated to professors is a 16-core machine. Students also have access to a powerful multi-core computer.

1. Log in `econsuper.econ.cuhk.edu.hk`;
2. Save the code block below as `loop_server.R`, and upload it to the server;
3. In a shell, run `R --vanilla <loop_server.R> result_your_name.out`;
4. To run a command in the background, add `&` at the end of the above command. To keep it running after closing the console, add `nohup` at the beginning of the command.

```
library(plyr)
library(foreach)
library(doParallel)

# prepare the functions
mu <- 2
CI <- function(x) {
  # x is a vector of random variables

  n <- length(x)
  mu <- mean(x)
  sig <- sd(x)
  upper <- mu + 1.96 / sqrt(n) * sig
  lower <- mu - 1.96 / sqrt(n) * sig
  return(list(lower = lower, upper = upper))
}

capture <- function() {
  x <- rpois(sample_size, mu)
  bounds <- CI(x)
  return((bounds$lower <= mu) & (mu <= bounds$upper))
}

############ implementation ###############
Rep <- 400
sample_size <- 5000000

# compare to the parallel version
registerDoParallel(16) # opens other CPUs

pts0 <- Sys.time() # check time
out <- foreach(icount(Rep), .combine = c) %dopar% {
  capture()
}

cat("empirical coverage probability = ", mean(out), "\n") # empirical size
pts1 <- Sys.time() - pts0 # check time elapse
print(pts1)
```

```r
pts0 <- Sys.time()
out <- foreach(icount(Rep), .combine = c) %do% {
  capture()
}
cat("empirical coverage probability = ", mean(out), "\n") # empirical size
pts1 <- Sys.time() - pts0 # check time elapse
print(pts1)
```

Reading

Wickham and Grolemund: Ch 10, 11, and 21

References

Li, Quefeng, Guang Cheng, Jianqing Fan, and Yuyan Wang. 2018. "Embracing the Blessing of Dimensionality in Factor Models." *Journal of the American Statistical Association* 113 (521): 380–89.

Wickham, Hadley, and Garrett Grolemund. 2016. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media, Inc.