Lecture 2: Advanced R

Zhentao Shi

Feb 11, 2017

Git

References

- Online Tutorial
- Pro Git

basic commands

Local

- git clone https://github.com/frankshi/econ5170
- git config –global user.name
- git config –global user.email
- git init
- git status
- git add
- git commit
- git log
- git tag -a v1.0 -m 'message'
- git rm –cached filename
- git branch [brach_name]
- git checkout [commit_id or branch name]
- .gitignore

Remote

- git remote add origin
- git push origin master

Packages

The following packages are useful for parallel computing.

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

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

Speed

Mathematically equivalent though, different calculation methods perform very differently in terms of computational speed.

example: calculate the heteroskedastic-robust variance for the OLS regression of a linear probability model. The difficult part is $\sum_{i=1}^{n} e_i^2 x_i x_i'$. We propose four ways.

1. literally add through $i = 1, \ldots, n$ one by one.
2. $X'\text{diag}(e^2)X$, with a dense central matrix.
3. $X'\text{diag}(e^2)X$, with a sparse central matrix.
4. Xe cross product.

```
# an example of robust matrix, sparse matrix. Vecteroization.
rm(list = ls( ) )
library(Matrix)

set.seed(111)

# n = 6000; 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 the parameters
    b0 = matrix( c(-1,1), nrow = 2 )

    # generate the data
    e = rnorm(n)
    X = cbind( 1, rnorm(n) )
    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

    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) {
      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)  {
      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)  {
      Xe = X * e
```

```
      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 =  0.789505
## n =  50 , Rep =  1000 , opt =  2 ,
##   time =  0.1535141
## n =  50 , Rep =  1000 , opt =  3 ,
##   time =  1.528266
## n =  50 , Rep =  1000 , opt =  4 ,
##   time =  0.10021
```

Efficient loop

`plyr` is an R package developed by Hadley Wickham.

example: calculate the empirical coverage probability of a poisson distribution of degree of freedom 2.

```
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) )
}
```

The standard loop

```
Rep = 10000
sample_size = 1000

# a standard loop
out = rep(0, Rep)
pts0 = Sys.time() # check time
mu = 2
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 =  0.9504
```

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

```
## Time difference of 0.8527648 secs
```

A `plyr` loop. It saves book keeping chores, and easier to parallelize.

```
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 =  0.9533
```

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

```
## Time difference of 0.9502132 secs
```

Parallel computing

The basic structure for parallel computing.

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

registerDoParallel() # opens other CPUs

l_ply(.data = 1:10,
      .fun = myfunction,
      .parallel = TRUE,
      .paropts = list( .packages = package.list,
        .export = ls(envir=globalenv() ) )
)
```

In this comparative example, we try

```
registerDoParallel(2) # open 2 CPUs



pts0 = Sys.time() # check time
out = ldply(.data = 1:Rep, .fun = capture, .parallel = T,
            .paropts = list(.export = ls(envir=globalenv() )) )
cat( "empirical coverage probability = ", mean(out$V1), "\n") # empirical size
```

```
## empirical coverage probability =  0.9519
```

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

```
## Time difference of 4.352745 secs
```

The above block indeed takes more time, because each loop runs very fast.

The code below shows a different story. Each loop takes more time, which dominates the overhead of the CPU communication.

```
Rep = 200
sample_size = 2000000
pts0 = Sys.time() # check time
out = ldply(.data = 1:Rep, .fun = capture, .parallel = FALSE)
cat( "empirical coverage probability = ", mean(out$V1), "\n") # empirical size
pts1 = Sys.time() - pts0 # check time elapse
print(pts1)

# compare to the parallel version
pts0 = Sys.time() # check time
out = ldply(.data = 1:Rep, .fun = capture, .parallel = TRUE,
            .paropts = list(.export = ls(envir=globalenv() )) )
cat( "empirical coverage probability = ", mean(out$V1), "\n") # empirical size
pts1 = Sys.time() - pts0 # check time elapse
print(pts1)
```

Econ Super

Try out this script on our econ super computer.

1. Log in `econsuper`;
2. Save the code block below as `loop_server.R`, and upload it to the server;
3. In a terminal, 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.