STA242 Assignment 5

Yilun Yang
912425744

GIT REPO ADDRESS: git@bitbucket.org:ArsMing276/stat-242.git

In this assignment we have two primary task—Compute the deciles of the total amount less the tolls AND Fit a linear regression predicting total amount less the tolls using trip time as the predictor.

At first, I am not sure which approach is faster, thus I tested to read in data with shell command in fread function, with shell in pipe and with shell in system. Out of my expectation, it took me about 8 minutes to read a column for the latter two approach but only 1.5 minute for fread function! I then saved total minus tolls and triptime from 24 files into rda files with fread function. The average time consumption to extract one column with fread is:

user	system	elapsed
9.46	0.17	92.24

1. Compute the deciles of the total amount less the tolls

I am still not sure which is faster-- Reading data first then use the rda files to compute the deciles or read and process both in parallel. I applied both approaches to test, for the parallel process, I loaded data with fread function and stored the result in frequency table. This took me 29 minutes totally (My laptop is kind of out of date). For the approach with fread function directly, it took me about 40 minutes to finish all work. The result for deciles is:

0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
-1430	5.99	7.49	8.49	9.74	10.99	12.99	14.99	18.49	26.11	685908.1

We can see most of the data are between 6 and 26, but the range is from -1430 to 685980.1 thus we have a really light tail in this distribution.

Two results are totally the same.

For computational time, Parallel computation took a little less time than applying fread function directly. This is not surprising. However, we can see although parallel time is less, it's not so significant to be four times. I assume this is due to the assignment of data and variables to each worker and the communication of threads.

For programming time, they are similar. Both of approaches are very simple to implement and they didn't take me much time. Making frequency tables in this problems is a little tricky. I really don't know if there exists any command which can combine all frequency tables together with different keys. I wrote it myself in which I create a table with the names as union of keys from all tables then combine those tables together. Nothing fancy.

2. Fit a linear regression predicting total amount less the tolls using trip time as the predictor.

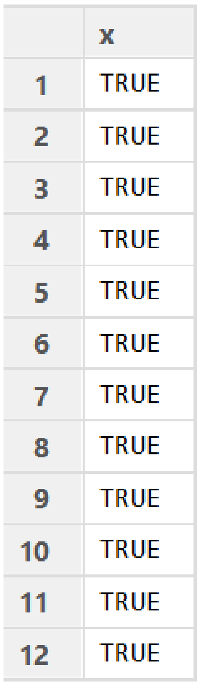
(1) Test if two sample list match each other

To finish this task, I firstly compare if trip_fare and trip_data match each other for different months by sampling 50000 observations in each file and compare the first, second and sixth columns(they are medallion, license hack and pick up time). I stored the sampled data for trip_fare and trip_data into two list, each list has length 12 as 12 files and each contains a matrix with three rows as medallion license hack and pick up time and 50000 columns as observations. The first two observations in the matrix are like the following:

	Obs1	Obs2
Medallion	82420BD6311D2BB88E2F213A8CA15BECE	430D7E81CB45708C7F9E7AC8554C5049
License Hacking	213779F7AA40E58009B3457E3FA7089D	2F0A0422967CD8AE9DA3455025B8F38F
Pickup Time	2013-01-08 10:21:01	2013-01-13 10:47:00

First two observations in each matrix

By comparing the two lists for trip_fare and trip_data, we found they are totally the same. Result is below:



	x
1	TRUE
2	TRUE
3	TRUE
4	TRUE
5	TRUE
6	TRUE
7	TRUE
8	TRUE
9	TRUE
10	TRUE
11	TRUE
12	TRUE

We can see 12 lists are all the same.

In this problem, I only used parallel with shell command because it's speed is already acceptable, it took me about 47 minutes totally to sample from 12 trip_fare files and 12 trip_data files, stored them in list and compare them.

(2) Perform regression

Consider beta1 and beta 0 can be rewrite as:

$$\beta_1 = \frac{\sum_{i=1}^{n} X_i Y_i - \sum_{i=1}^{n} X_i \sum_{i=1}^{n} Y_i \frac{1}{n}}{\sum_{i=1}^{n} X_i^2 - (\sum_{i=1}^{n} X_i)^2 \frac{1}{n}} \qquad \beta_0 = \frac{1}{n}\sum_{i=1}^{n} Y_i - \frac{1}{n}\sum_{i=1}^{n} X_i \beta_1$$

Our intuition if to calculate $\sum_{i=1}^{n} X_i Y_i$ $\sum_{i=1}^{n} Y_i$ $\sum_{i=1}^{n} X_i$ and $\sum_{i=1}^{n} X_i^2$ at first then estimation can be got. I used parallel approach with pipe command in order to process the data in blocks and hopes this will save my memory. Also, I use a serialized approach to use the rda data sets I saved before in a loop to calculate the estimators. Result is as following:

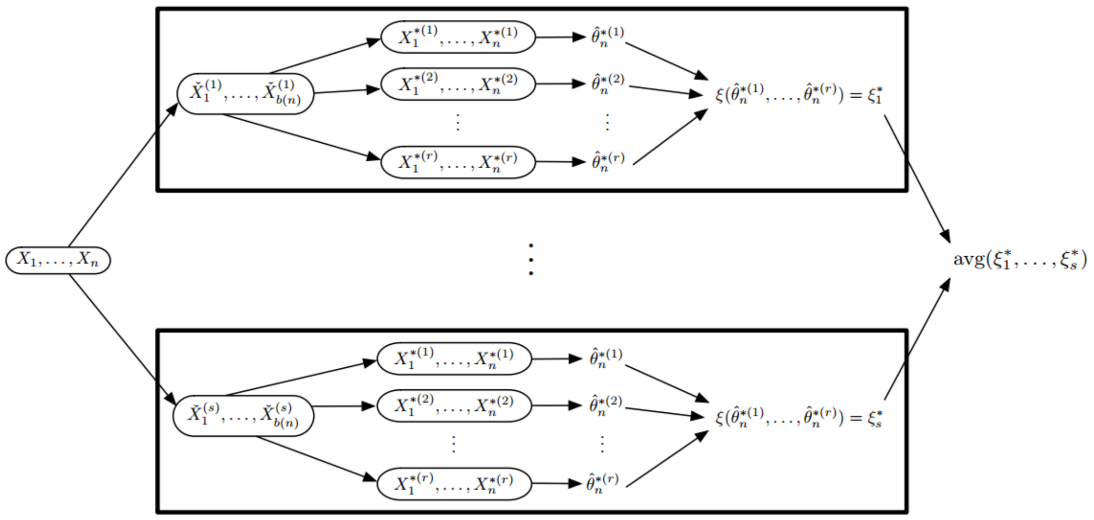
β_1 = 2.06e-5, β_0 = 14.523

Results for the two approaches are the same. However, for computation time and programming time, that is a different story.

The computation time is out of my expectation, it took me nearly 3 hours to get the result for parallel computation but only 5 minutes in serialized process, even adding the rda data read in and process time, it is still not longer than 45 minutes. I believe there are two reasons here. Firstly, I want to process in blocks thus I have to choose a slower method to read in data, which is pipe compared with fread. Secondly, sending data and function to each worker also took lots of time. Finally, function for each node is much more complex than the former problem, I actually used three connections to read data and many variables to cumulatively update the data, I believe it also took lots of time.

For programming time, Serialized process is also easier, however, because we need not to consider how to read in blocks in nodes, we can just use a loop to deal with each file. Making up a parallel process with three connections to read data in blocks and compute the estimators at the same time is what I spent the most time in. I think I will pick up the serialized process for this problem.

(3) Use BLB to compute standard error for beta1 and beta0


$X_1, \ldots, X_n$
$\check{X}_1^{(1)}, \ldots, \check{X}_{b(n)}^{(1)}$
$X_1^{*(1)}, \ldots, X_n^{*(1)}$
$X_1^{*(2)}, \ldots, X_n^{*(2)}$
$X_1^{*(r)}, \ldots, X_n^{*(r)}$
$\hat{\theta}_n^{*(1)}$
$\hat{\theta}_n^{*(2)}$
$\hat{\theta}_n^{*(r)}$
$\xi(\hat{\theta}_n^{*(1)}, \ldots, \hat{\theta}_n^{*(r)}) = \xi_1^*$
$\check{X}_1^{(s)}, \ldots, \check{X}_{b(n)}^{(s)}$
$\xi(\hat{\theta}_n^{*(1)}, \ldots, \hat{\theta}_n^{*(r)}) = \xi_s^*$
$\text{avg}(\xi_1^*, \ldots, \xi_s^*)$

BLB is not a hard method to state but kind of hard to code in my opinion. I firstly sample 10 subsample sets with each has m = n^0.6 = 87720 samples then I resampling each subsample n times and compute each beta0 and beta1 for each resampling branch. Finally I estimate the standard error for each subsample based on the beta0 and beta1 we estimated for each resampling branch and averaged on them to get the final result.

I also applied parallel and serialized approach again. Not out of my expectation, since the function is even more complicated than the former problem, serialized approach is out-performed again. Actually parallel approach took me nearly 4 hours and 17 minutes to get the result while serialized approach took me 54 minutes including reading data and the whole BLB process. The result is as following:

$se(\beta_1)$ = 6.2e-8, se(β_0)= 0.0001

In this problem, the hardest part if how to resample 87720 samples into more than 170 million for many times. Frequency table is a good idea, but there are also other tough questions like how to spot the corresponding x and y in the original files since we sampled indices not data itself and how to determine how many times that each of the 87720 observations to be sampled in the resampling process? We can’t sample 170 million times in serial because of the memory and time concerns. Here I used multinomial random generation method to pick out the times for each observation to be sampled. Generally, this part is what challenged me the most.

Computation time and programming time is quite like the conclusion of former question – function is complex, data is big and we have to use block read in method due to sampling for parallel approach, of course, serialized approach is preferred.

Conclusion:

Parallel computation approach is not as useful as I was considered, there are too many conditions to restrict it from being well performed. Besides, it also breaks easier due to memory issue and the process will be terminated even if only one node has a mistake. We have to check many times before we perform the parallel process. However, I believe Operation System is also a reason to cause this slowness, I heard some friends with windows OS like me also complain about the slowness in parallel while seldom in MAC OS system. Nevertheless, as we can see in the first problem, when data is not that big and function for each node is not that complex, parallel can outperform serialized approach even in windows.

APPENDIX： Code

```
library('snow')
library('data.table')
file = list.files("D:/Rwd", pattern = "fare.*\\.csv$",
full.names = TRUE)
cl = makeCluster(4)
clusterExport(cl, "fread")
els = clusterSplit(cl, file)

f = function(x){
  ##read in files without title
  total = as.data.frame(fread(sprintf('tail -n +2 -q %s|cut -f
11 -d, ', paste(x, collapse = ' '))))[,1]
  toll = as.data.frame(fread(sprintf('tail -n +2 -q %s|cut -f 10
-d ,', paste(x, collapse = ' '))))[,1]
  diff = as.numeric(total) - as.numeric(toll)
  freq = table(diff)
  freq
}

cldiff = clusterApplyLB(cl, els, f)
save(cldiff, file = 'cldiff.rda')

nm = unique(names(do.call(c, cldiff)))
nm = nm[order(as.numeric(nm))] ##find out what numbers are in
the freq table
allfreq = rep(0, length(nm))
names(allfreq) = nm  ##make a big table with all numbers
for(i in 1:4){
  freq = rep(0, length(nm))
  freq[match(names(cldiff[[i]]), nm)] = cldiff[[i]]
  allfreq = allfreq + freq  ##extend small tables into a big one
and combine all tables up.
}
allfreq

##finding out cumulative frequncy for numbers an deciles
##Then determine which group each decile should belong to.
cumfreq = cumsum(allfreq)
deciles = seq(1, sum(allfreq), length.out = 11)
groups = sapply(deciles, function(x) names(cumfreq[cumfreq <=
x][sum(cumfreq <= x)]))
groups


##sampling to check if two data sets match
library('snow')
bigsample = function(filename){
  n = system(sprintf('wc -l %s', filename), intern = TRUE)
  n = as.numeric(strsplit(n, ' ')[[1]][1]) ##check row number
for each file
  set.seed(242)
  ids = sample(2:n, 50000) ##skip title
  jumps = diff(c(0, sort(ids)))
  con = file(filename, "r")  # connection
  isOpen(con)  # TRUE  - just checking

  ans = character(50000) ##sample 50000 obs for each file
  for(i in seq_along(jumps))
    ans[i] = readLines(con, jumps[i])[jumps[i]]
    j = match(ids, sort(ids))
    ans[j]

    ans
}

f = function(filename){
  samp = bigsample(filename)  ##get the 50000 samples
  if(grepl("fare", filename)) colind = c(1,2,4)
  else colind = c(1,2,6)
  matsamp    =    sapply(strsplit(samp,    ','),    function(x)
x[colind])##extract different columns accrording to file name
and put into a matrix
  matsamp
}

tripfare  =  list.files("D:/Rwd",  pattern  =  "fare.*\\.csv$",
full.names = TRUE)
tripdata = list.files("D:/Rwd", pattern = "trip_data.*\\.csv$",
full.names = TRUE)

cl1 = makeCluster(4)
clusterExport(cl1, "bigsample")
faresamp = parLapply(cl1, tripfare, f)
save(faresamp, file = 'faresamp.rda')

cl2 = makeCluster(4)
clusterExport(cl2, "bigsample")
datasamp = parLapply(cl2, tripdata, f)
save(datasamp, file = 'datasamp.rda')

##check if faresamp and datasamp are totally the same in the
three columns
checkequal  =  sapply(1:12,  function(x)  all(faresamp[[x]]  ==
datasamp[[x]]))

#############

##calculate b0 and b1 parallel version
library('snow')
library('stringr')
tripfare  =  list.files("D:/Rwd",  pattern  =  "fare.*\\.csv$",
full.names = TRUE)
tripdata = list.files("D:/Rwd", pattern = "trip_data.*\\.csv$",
full.names = TRUE)
file = c(tripdata, tripfare)
pos = do.call(c, lapply(1:12, function(x) grep(sprintf('_%d.', 
x), file, fixed = TRUE)))
file = file[pos]  ##rearrange file order according to month, 
this is because we want each node have
                 ##trip_fare and trip_data of same months
cl = makeCluster(4)
clusterExport(cl, "str_trim")
els = clusterSplit(cl, file)

f = function(x){
  n  =  system(sprintf('wc  -l  %s',  paste(c(x[2],x[4],x[6]), 
collapse = ' ')), intern = TRUE)
  n = as.numeric(strsplit(str_trim(n[4]), ' ')[[1]][1]) - 3 #-3 
because of title!!!
  iter = n%/%5000000 + 1  ##check how many iteration read in 
blocks to the end of file

  xysum = 0
  xsum = 0
  ysum = 0
  xsqrsum = 0

  ##Three connection of 9,10,11 columns
  con1  =  pipe(sprintf('tail  -n  +2  -q  %s|cut  -f  11  -d,', 
paste(c(x[2],x[4],x[6]), collapse = ' ')))
  con2  =  pipe(sprintf('tail  -n  +2  -q  %s|cut  -f  10  -d,', 
paste(c(x[2],x[4],x[6]), collapse = ' ')))
  con3  =  pipe(sprintf('tail  -n  +2  -q  %s|cut  -f  9  -d,', 
paste(c(x[1],x[3],x[5]), collapse = ' ')))
  for(i in 1:iter){
    fare11 = readLines(con1, 5000000)
    fare10 = readLines(con2, 5000000)
    data9 = readLines(con3, 5000000) ##get the data

    rsp = as.numeric(fare11) - as.numeric(fare10)
    rsp = rsp[!is.na(rsp)]
    pdt = as.numeric(data9)
    pdt = pdt[!is.na(pdt)] ##get predictor and response variables

    xysum = xysum + sum(pdt * rsp)
    xsum = xsum + sum(pdt)
    ysum = ysum + sum(rsp)
    xsqrsum = xsqrsum + sum(pdt^2) ## for each interation, adding
these statistc up
  }

  stat = c(xysum, xsum, ysum, xsqrsum, n)
  names(stat) = c('xysum', 'xsum', 'ysum', 'xsqrsum', 'n')

  stat
}

clest = clusterApply(cl, els, f) ##This is a list of our
statistics in each node
save(clest, file = 'clest.rda')
xystat = apply(do.call(rbind, clest), 2, sum) ##calculate the
statistics in all nodes
beta1 = (xystat[1] - xystat[2] * xystat[3] /
xystat[5])/(xystat[4] - xystat[2]^2 / xystat[5])
beta0 = xystat[3]/n - beta1 * xystat[2]/xystat[5]


####calculate b0 and b1 serialized version
library('stringr')
tripfare = list.files("D:/Rwd", pattern = "diff_.*\\.rda$",
full.names = TRUE)
tripdata = list.files("D:/Rwd", pattern = "time_.*\\.rda$",
full.names = TRUE)
file = c(tripdata, tripfare)
pos = do.call(c, lapply(1:12, function(x) grep(sprintf('_%d.',
x), file, fixed = TRUE)))
file = file[pos]
xysum = 0
xsum = 0
ysum = 0
xsqrsum = 0

for(i in seq(1, 23, 2)){
  load(file[i])
  load(file[i+1])

  xysum = xysum + sum(triptime * diff)
  xsum = xsum + sum(triptime)
  ysum = ysum + sum(diff)
  xsqrsum = xsqrsum + sum(triptime^2)
```

```
}

n = 173179759
beta1 = (xysum - xsum * ysum / n)/(xsqrsum - xsum^2 / n)
beta0 = ysum/n - beta1 * xsum/n

####beta1 = 2.06e-5, beta0 = 14.523####

####BLB to calculate se parallel version
library('snow')
library('stringr')
library('multinomRob')

##calculate row number for all files and the totle number
tripfare  =  list.files("D:/Rwd",  pattern  =  "fare.*\\.csv$",
full.names = TRUE)
pos = do.call(c, lapply(1:12, function(x) grep(sprintf('_%d.',
x), tripfare, fixed = TRUE)))
tripfare = tripfare[pos]
wc = system(sprintf('wc -l %s', paste(tripfare, collapse = ' ')),
intern = TRUE)
wc   =   sapply(strsplit(str_trim(wc),   '   '),   function(x)
as.numeric(x[1]))
flen = wc[1:12] - 1

n = wc[13] - 12  ##total number minus title lines
m = ceiling(n^0.6) ##subsample number
subsample = list()
for(i in 1:10) {
  set.seed(i)
  subsample[[i]] = sample(n, m)
}

##For  resample,  we  can't  sample  directly,  but  we  can  use
multinomial random generating function
##to get how many times each number in subsample observations be
sampled
##we sample 1000 times because we have 10 subsample sets and in
each set we have 100 resample data
resample          =          sapply(1:1000,          function(x)
{set.seed(x);rmultinomial(n, pr = rep(1/m, m))})
save(resample, file = 'resample.rda')

##check  what  rows(all  files  combined)  are  in  the  resampling
```

```
result.
sampind = sort(unique(do.call(c, subsample)))

##To determine which file each each row number belongs to and
calculate the row number in
##the corresponding file instead of the row number of all files
combined
cumlen = cumsum(flen)
group = cut(sampind, c(0, cumlen), labels = FALSE)
grpsamp = list()
for(i in 1:12){
  grpsamp[[i]]  =  subset(sampind,  group  ==  i)  -  c(0,
cumlen[1:11])[i]
}

f = function(filename){
  ##check which month the file is
  which = which(sapply(1:12, function(x) grepl(sprintf('_%s.',
x), filename, fixed = TRUE)))
  ids = grpsamp[[which]]
  jumps = diff(c(0, ids))
  len = length(ids)
  ans = character(len)


  ##get x or y based on which file is
  if(grepl("fare", filename)){
    con1 = pipe(sprintf('cut -f 11 -d, %s', filename))
    con2 = pipe(sprintf('cut -f 10 -d, %s', filename))
    readLines(con1, 1)
    readLines(con2, 1)
    for(i in seq_along(jumps)){
      line11 = readLines(con1, jumps[i])[jumps[i]]
      line10 = readLines(con2, jumps[i])[jumps[i]]
      ans[i] = as.numeric(line11) - as.numeric(line10)
    }


    ans
  }
  else{
    con = pipe(sprintf('cut -f 9 -d, %s', filename))
    readLines(con, 1)
    for(i in seq_along(jumps))
```

```
      ans[i] = readLines(con, jumps[i])[jumps[i]]

    ans
  }

}

tripfare  =  list.files("D:/Rwd",  pattern  =  "fare.*\\.csv$",
full.names = TRUE)
tripdata = list.files("D:/Rwd", pattern = "trip_data.*\\.csv$",
full.names = TRUE)
file = c(tripdata, tripfare)
pos = do.call(c, lapply(1:12, function(x) grep(sprintf('_%d.',
x), file, fixed = TRUE)))
file = file[pos]
cl = makeCluster(6)
clusterExport(cl, "grpsamp")
sampdat = parLapply(cl, file, f)
save(sampdat, file = 'sampdat.rda')
######This is extremely slow thus not be considered.

library('stringr')
library('multinomRob')
library('plotrix')
tripfare  =  list.files("D:/Rwd",  pattern  =  "fare.*\\.csv$",
full.names = TRUE)
pos = do.call(c, lapply(1:12, function(x) grep(sprintf('_%d.',
x), tripfare, fixed = TRUE)))
tripfare = tripfare[pos]
wc = system(sprintf('wc -l %s', paste(tripfare, collapse = ' ')),
intern = TRUE)
wc   =   sapply(strsplit(str_trim(wc),   '   '),   function(x)
as.numeric(x[1]))
flen = wc[1:12] - 1

n = wc[13] - 12
m = ceiling(n^0.6)
subsample = list()
for(i in 1:10) {
  set.seed(i)
  subsample[[i]] = sample(n, m)
}
```

```
resample           =           sapply(1:1000,          function(x)
{set.seed(x);rmultinomial(n, pr = rep(1/m, m))})
save(resample, file = 'resample.rda')

sampind = sort(unique(do.call(c, subsample)))
cumlen = cumsum(flen)
group = cut(sampind, c(0, cumlen), labels = FALSE)
grpsamp = list()
for(i in 1:12){
  grpsamp[[i]]   =   subset(sampind,   group   ==   i)   -   c(0,
cumlen[1:11])[i]
}


tripfare  =  list.files("D:/Rwd",  pattern  =  "diff_.*\\.rda$",
full.names = TRUE)
tripdata  =  list.files("D:/Rwd",  pattern  =  "time_.*\\.rda$",
full.names = TRUE)
file = c(tripdata, tripfare)
pos = do.call(c, lapply(1:12, function(x) grep(sprintf('_%d.',
x), file, fixed = TRUE)))
file = file[pos]

xyvalue = list()
for(i in 1:12){
  ids = grpsamp[[i]]
  load(file[2 * i - 1])
  load(file[2 * i])
  x = triptime[ids]
  y = diff[ids]
  xyvalue[[i]] = data.frame(x = x, y = y)
}

##put the x and y value we got into a data frame as an table to
match index, x and y
xyvalues = do.call(rbind, xyvalue)
indtbl  =  data.frame(sampind  =  sampind,  x  =  xyvalues$x,  y  =
xyvalues$y)

f = function(x){
  mch = match(x, indtbl$sampind)
  df = data.frame(xval = indtbl$x[mch], yval = indtbl$y[mch])

  df
}
subsampxy = lapply(subsample, f) ##get the x and y value from
the former table for each subsample observation


findbeta = function(subsampxy, resample){
  beta1 = numeric(1000)
  beta0 = numeric(1000)
  for(i in 1:1000){
    ##for each resample dataset, the repetition times are counted
as weight for the regression
    ##We  calculate  beta0  and  beta1  for  each  resample  set
according to their subsample group
    weight = resample[,i]
    group = cut(1:1000, breaks = 100 * (0:10), labels = FALSE)
    x = subsampxy[[group[i]]]$x
    y = subsampxy[[group[i]]]$y
    xysum = sum(x * y *weight)
    xsum = sum(x * weight)
    ysum = sum(y * weight)
    xsqrsum = sum(x^2 * weight)
    n = sum(weight)
    b1 = (xysum - xsum * ysum / n)/(xsqrsum - xsum^2 / n)
    b0 = ysum/n - b1 * xsum/n
    beta1[i] = b1
    beta0[i] = b0
  }
  betadf = data.frame(beta1 = beta1, beta0 = beta0)

  betadf
}

betadf = findbeta(subsampxy, resample)

##Finally, we calculate se for each subsample group and average
them to get the final se for beta
beta1se = mean(tapply(betadf$beta1, group, std.error))
beta0se = mean(tapply(betadf$beta0, group, std.error))

######se for b1 and b0 are 6.2e-8, 0.0001)
```