lab04

Yixuan Li, UPI:yil845

20 August 2021

##The purpose of this lab is to measure the time consumption of code processing and look at options to improve code efficiency (in R).

Tasks

Import

1, the following code measures time consumption for read.csv on test file.

```
system.time(countsDF<-read.csv("/course/NZTA/20130101_20130331_TMSTrafficQuarterHour.csv"))
```

```
##    user  system elapsed 
##  20.696   0.827  21.525
```

2, Comparing with read.csv code, read.table is 5 times faster because it takes less memory for intermediate objects.

```
system.time(countsDT<-data.table::fread("/course/NZTA/20130101_20130331_TMSTrafficQuarterHour.cs
v"))
```

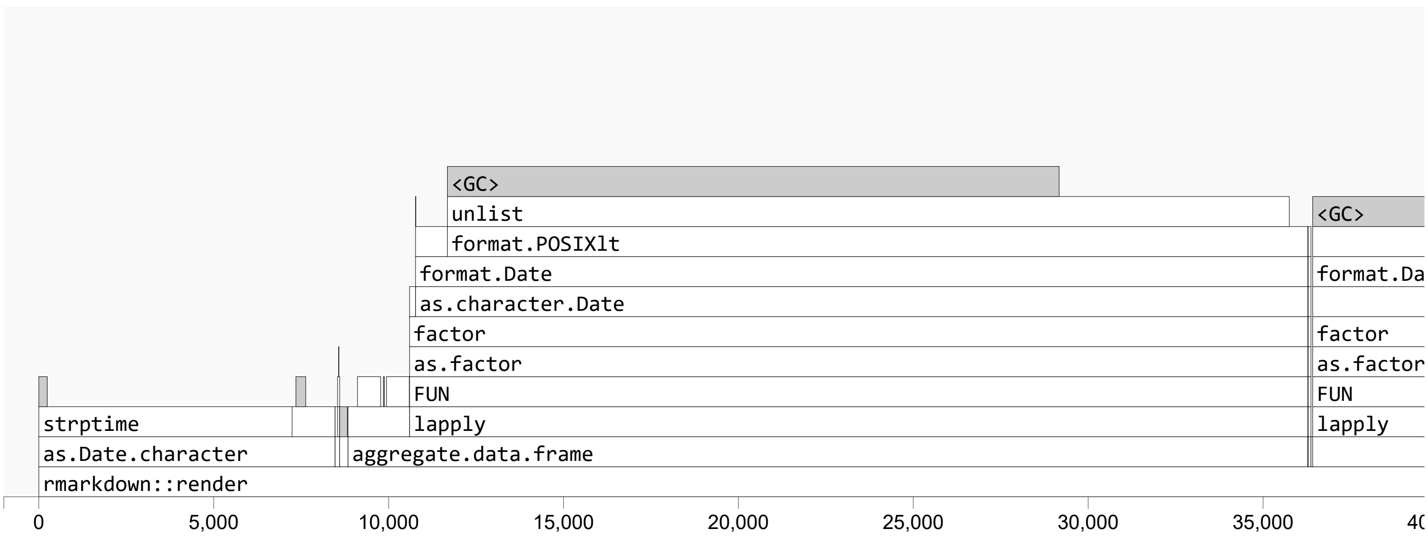
```
##    user  system elapsed 
##  27.438   0.213   4.256
```

Transform

3, profiling the following code and find out the maximum time span. The total time is ~ 5s. It shows aggregating takes most of time.

```
library(profvis)

## check total timespan of data.frame commands for adding a column of day in dataframe &suming u
p counts based on day, siteRef and class. Total time used is ~41s, in which aggregating takes th
e most of time,~32s. The result is stored in profile1.html.
a <- profvis({
  countsDF$day <- as.Date(countsDF$startDatetime, format="%d-%b-%Y")
  dailyCountsDF <- aggregate(countsDF["count"], countsDF[c("day", "siteRef", "class")],sum)
  })
htmlwidgets::saveWidget(a, "profile1.html")
a
```

(Sources not available)

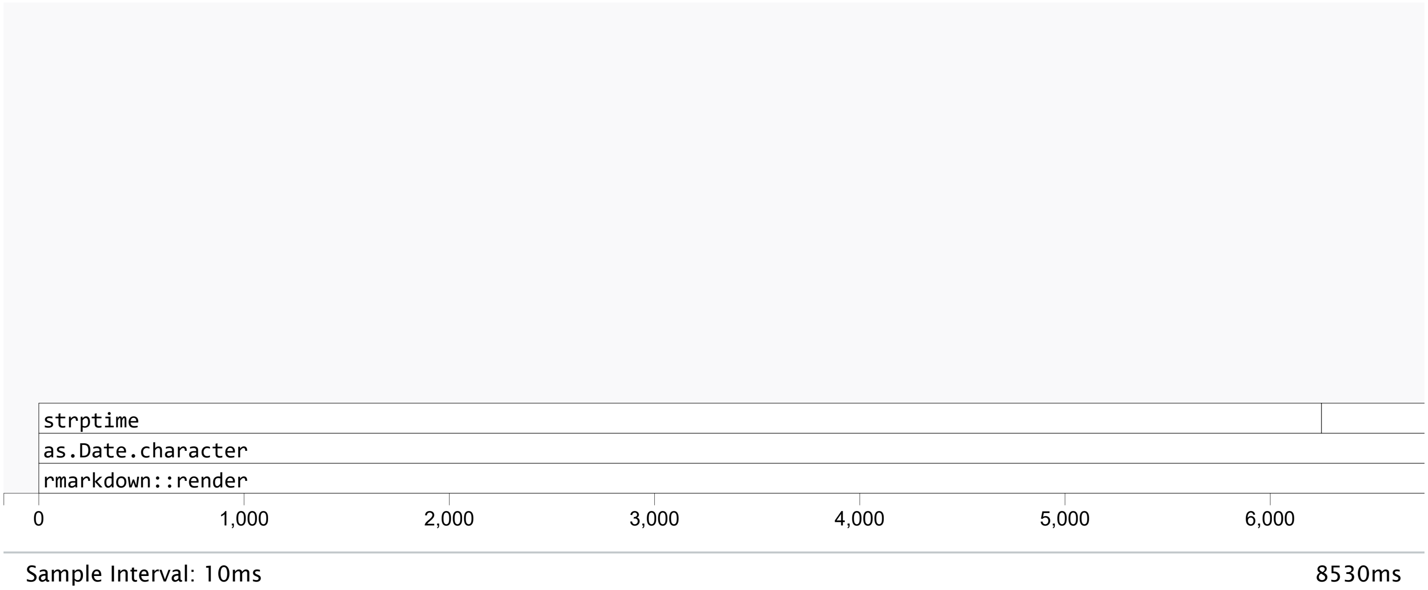


Sample Interval: 10ms 50050ms

```
## check total timespan of data.table functions to do the same thing. Total time used is ~8.8s,
 about 5 times faster than commands of data.frame. The result is stored in profile2.html.
b <- profvis(dailyCountsDT <- countsDT[, day := as.Date(startDatetime, format="%d-%b-%Y")][, sum
(count), .(day, siteRef, class)])
htmlwidgets::saveWidget(b, "profile2.html")
b
```

Flame Graph | Data | Options ▾

(Sources not available)

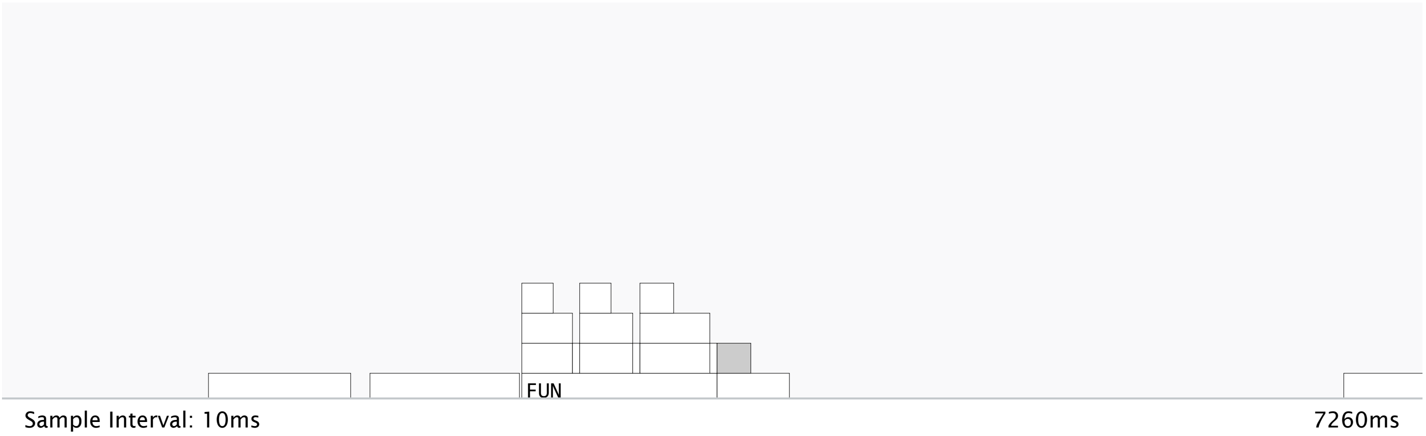


4, these codes are attempted to improve processing time. The total timespan is ~9ms, that means keeping date as character dramatically increased processing efficiency.The result is stored in profile3.html

```
countsDF1<-read.csv("/course/NZTA/20130101_20130331_TMSTrafficQuarterHour.csv")
c<- profvis({
countsDF1$day <- substr(as.character(countsDF1$startDatetime), 1, 11)
dailyCountsDF1 <- aggregate(countsDF1["count"], countsDF1[c("day", "siteRef", "class")],sum)
})
htmlwidgets::saveWidget(c, "profile3.html")
c
```

Flame Graph | Data | Options ▾

<expr>	Memory	Time
countsDF1<-read.csv("/course/NZTA/20130101_20130331_TMSTrafficQuarterHour.csv")		
c<- profvis({		
countsDF1$day <- substr(as.character(countsDF1$startDatetime), 1, 11)		400
dailyCountsDF1 <- aggregate(countsDF1["count"], countsDF1[c("day", "siteRef", "class")],sum)		
})		
htmlwidgets::saveWidget(c, "profile3.html")		
c		



5, The new code which produces dailyCountsDF1, shows the same result as the original code, althought the date formats are different. In dailyCountsDF, day is in date format, whereas in dailyCountsDF1, day is in character format.

```
dim(dailyCountsDF)
```

```
## [1] 72157     4
```

```
head(dailyCountsDF[order(dailyCountsDF$count, decreasing = TRUE),])
```

```
##              day  siteRef class count
## 45597 2013-03-28 01N10431     L 99129
## 45558 2013-02-14 01N10431     L 99035
## 45565 2013-02-22 01N10431     L 98618
## 50005 2013-03-28 01N29424     L 98540
## 45564 2013-02-21 01N10431     L 97987
## 45545 2013-02-01 01N10431     L 97696
```

```
dim(dailyCountsDF1)
```

```
## [1] 72157     4
```

```
head(dailyCountsDF1[order(dailyCountsDF1$count, decreasing = TRUE),])
```

```
##               day  siteRef class count
## 45594 28-MAR-2013 01N10431     L 99129
## 45552 14-FEB-2013 01N10431     L 99035
## 45575 22-FEB-2013 01N10431     L 98618
## 50002 28-MAR-2013 01N29424     L 98540
## 45572 21-FEB-2013 01N10431     L 97987
## 45514 01-FEB-2013 01N10431     L 97696
```

Model

6,Create a new variable scount that is the square-root of the count.

```
dailyCountsDF$scount<- sqrt(dailyCountsDF$count)
```

7,Split the data into 10 equal-sized chunks; 9 as training set and 1 as test set. the following code set a list called train, From train[[1]] to train[[9]], we use as train data, and train[[10]] as test data.

```
  index <- sample(rep(1:10, length.out=nrow(dailyCountsDF)))
  train <-list()
  for ( i in 1:10 ) {
  train[[i]] = dailyCountsDF[index == i,]
  }
 head(train[[10]])
```

```
##           day  siteRef class  count   scount
## 6  2013-01-06 00200002     H 1126.5 33.56337
## 12 2013-01-12 00200002     H 1496.5 38.68462
## 15 2013-01-15 00200002     H 2066.0 45.45327
## 27 2013-02-09 00200002     H 1279.5 35.77010
## 44 2013-02-26 00200002     H 2149.0 46.35731
## 46 2013-02-28 00200002     H 2226.0 47.18050
```

```
 dim(train[[10]])
```

```
## [1] 7215    5
```

```
# RMSE
RMSE <- function(obs, pred) {
  sqrt(mean((obs - pred)^2))
}
obs <- train[[10]]$scount

# function for lm(), predict and RMSE

lmFit = list()
all_mean = list()
predLM = list()
RMSE_class = list()
RMSE_mean = list()
  for (i in 1:9) {
    lmFit[[i]] = lm (scount ~ class, train[[i]])
    all_mean [[i]] = mean(train[[i]]$scount)
    predLM[[i]] = predict (lmFit[[i]], train[[10]])
    RMSE_class[[i]] = RMSE(obs,predLM[[i]])
    RMSE_mean[[i]] = RMSE(obs,all_mean[[i]])
  }
# Average RMSE for overal mean
RMSEMean = (RMSE_mean[[1]] + RMSE_mean[[2]] + RMSE_mean[[3]] + RMSE_mean[[4]] + RMSE_mean[[5]] +
RMSE_mean[[6]] + RMSE_mean[[7]] + RMSE_mean[[8]] + RMSE_mean[[9]]) / 9
RMSEMean
```

```
## [1] 56.38181
```

```
# Average RMSE for fitted curve
RMSEClass =(RMSE_class[[1]] + RMSE_class[[2]] + RMSE_class[[3]] + RMSE_class[[4]] + RMSE_class[[
5]] + RMSE_class[[6]] + RMSE_class[[7]] + RMSE_class[[8]] + RMSE_class[[9]]) / 9
RMSEClass
```

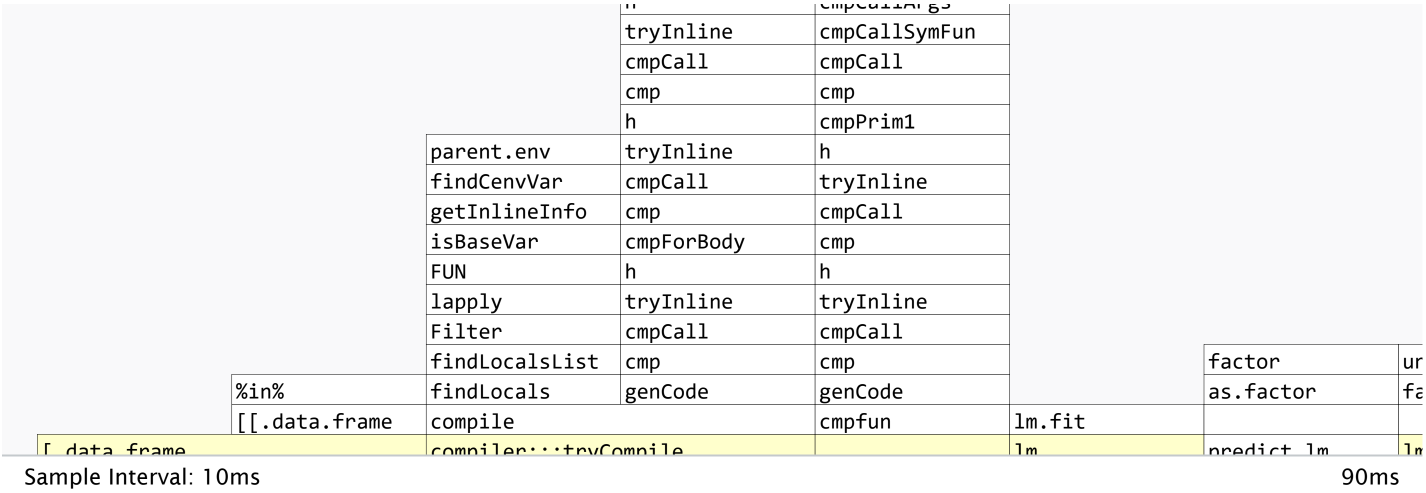
```
## [1] 44.7917
```

```
# Profile the code. Here we can see total time consumption is ~90ms, in which the linear fitting takes most of time, train data splitting is of the second-most-time-consumption. The profiling result is stored in profile4.html.

d <- profvis ({
    index <- sample(rep(1:10, length.out=nrow(dailyCountsDF)))
    train <-list()
    for ( i in 1:10 ) {
    train[[i]] = dailyCountsDF[index == i,]
    }
    head(train[[10]])
    dim(train[[10]])
    RMSE <- function(obs, pred) {
    sqrt(mean((obs - pred)^2))
    }
   obs <- train[[10]]$scount
   lmFit = list()
   all_mean = list()
   predLM = list()
   RMSE_class = list()
   RMSE_mean = list()
  for (i in 1:9) {
    lmFit[[i]] = lm (scount ~ class, train[[i]])
    all_mean [[i]] = mean(train[[i]]$scount)
    predLM[[i]] = predict (lmFit[[i]], train[[10]])
    RMSE_class[[i]] = RMSE(obs,predLM[[i]])
    RMSE_mean[[i]] = RMSE(obs,all_mean[[i]])
  }

  })
htmlwidgets::saveWidget(d, "profile4.html")
d
```

Flame Graph | Data | Options ▾

<expr> | **Memory** | **Time**

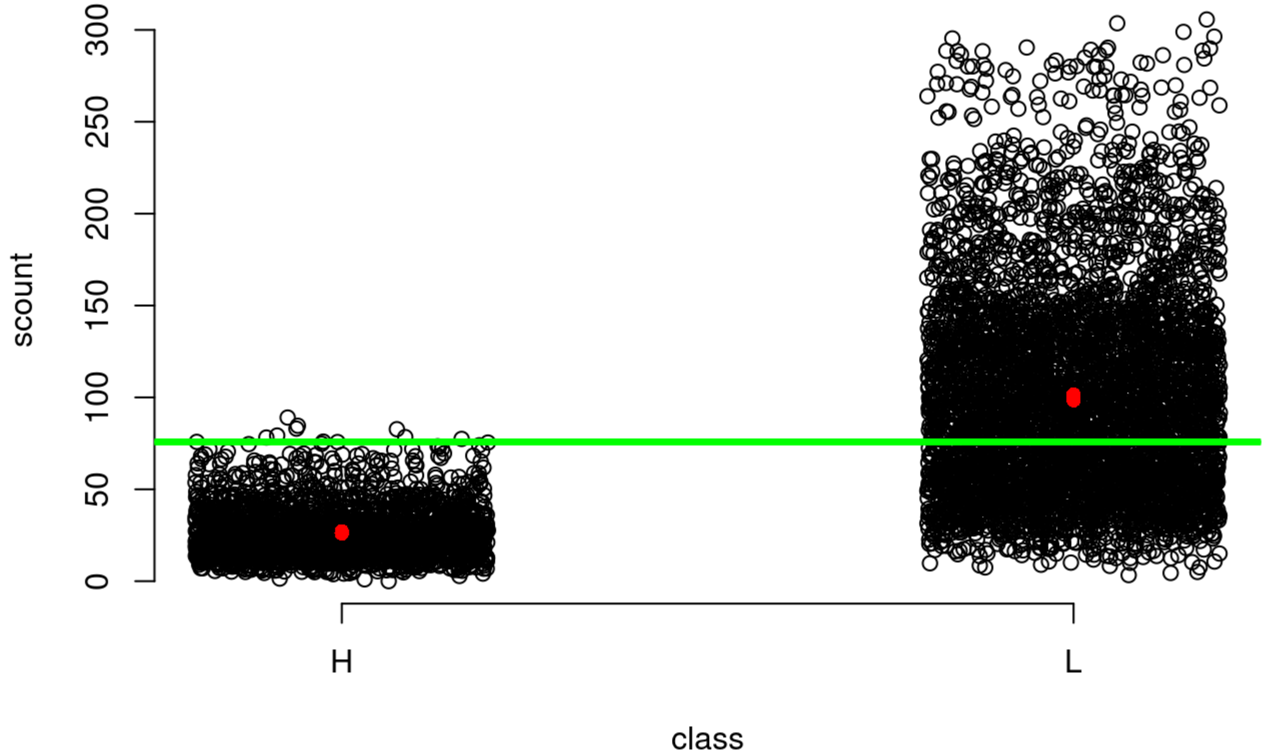
```
     index <- sample(rep(1:10, length.out=nrow(dailyCountsDF)))
     train <-list()
     for ( i in 1:10 ) {
     train[[i]] = dailyCountsDF[index == i,]
     }
    head(train[[10]])
    dim(train[[10]])

   # RMSE
   RMSE <- function(obs, pred) {
     sqrt(mean((obs - pred)^2))
   }
   obs <- train[[10]]$scount

   # function for lm(), predict and RMSE
```



visulization

8, test data are plotted as black circles in graph, nine lines of average mean curves are shown in green, and predicted linear fitted scount from nine training data sets are shown as read dots.

```
plot(scount ~ jitter(as.numeric(factor(class))), train[[10]],
     xlab="class", axes=FALSE)
axis(2)
axis(1, at=as.numeric(unique(factor(train[[10]]$class))),
     label=unique(factor(train[[10]]$class)))
for (i in 1:9) {
abline(h=all_mean[[i]], col="green")
points(as.numeric(unique(factor(train[[10]]$class))),
       predict(lmFit[[i]], data.frame(class=unique(factor(train[[10]]$class)))),
       pch=16, col="red")
}
```



Summary

In this lab, we tried all different way to check the time consumption of processing different R codes in order to improve our coding efficiency. We found out transforming data into date format consumed majority of time during data transformation. After modifying code and keeping date as simple character format, we saved almost 4 fold of time. During the data modeling, we found out that linear fitting lm() took most of the time in processing the code, after which the time splitting into train and test sets came in the second place.

The result of 9 training set modelling gave very similar results because they were generated from the same original data. The linear fitting is not ideal as usual. The computed RMSEs are relatively large showing the model is not working well.