Assignment 3 : Predict 454

Rahul Sangole

2018-11-05

```
library('tidyverse')
library('knitr')
library('kableExtra')
library('foreach')
library("doMC")
library("doParallel")
library('ggthemr')
ggthemr("fresh")
options(knitr.table.format = "latex")
knitr::opts_chunk$set(cache = TRUE)
```

Loading the data

Loading the data into the system for the first time is attempted using two techniques. In the first, `purrr::map_df` (tidy equivalent of the lapply/sapply core functions) is used to iterate over the unzipped csv files in the folder, apply column types, merge into a tibble, and store in `df`. We can see that `df` is a large tibble, of 11.5 million observations by 30 columns. Though, this method results in the entire tibble being in memory.

```
df <- list.files(pattern = "(csv)$") %>%
        map_df(~read_csv(.x,
                         col_types = cols(
                                 .default = col_integer(),
                                 UniqueCarrier = col_character(),
                                 TailNum = col_character(),
                                 AirTime = col_character(),
                                 Origin = col_character(),
                                 Dest = col_character(),
                                 TaxiIn = col_character(),
                                 TaxiOut = col_character(),
                                 CancellationCode = col_character(),
                                 CarrierDelay = col_character(),
                                 WeatherDelay = col_character(),
                                 NASDelay = col_character(),
                                 SecurityDelay = col_character(),
                                 LateAircraftDelay = col_character()
                         ),
                         col_names = T,
                         progress = T))
```

```
##
|                                                                     |   0%
|                                                                     |   0%
|                                                                     |   0%
|                                                                     |   0%
|                                                                     |   0%
```

```
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|===========================================================|  99%  463 MB
|============================================================| 100%  463 MB
```

```r
dim(df)
```

```
## [1] 11555122       29
```

In the second method, a locally hosted Postgresql database is used to store the data. The command structure to create the table and load the data is shown below.

```r
drv <- dbDriver("PostgreSQL")
con <- dbConnect(
    drv,
    dbname = "postgres",
    host = "localhost",
    port = 5433,
    user = "postgres",
    password = "password"
)

if (!dbExistsTable(con, "airlines")) {
    # specifies the details of the table
    sql_command <-
        "CREATE TABLE
    airlines(
    Year integer,  integer, DayofMonth integer, DayOfWeek integer, DepTime integer, CRSDepTime integer,
    CRSElapsedTime integer, AirTime integer, ArrDelay integer, DepDelay integer, Origin integer, Dest in
    CancellationCode integer, Diverted integer, CarrierDelay integer, WeatherDelay integer, NASDelay int
    )
    WITH (OIDS=FALSE)
    TABLESPACE pg_default;
    ALTER TABLE aviation_safety_training OWNER TO postgres;"

    dbGetQuery(con, sql_command)
```

```
    dbWriteTable(
        conn = con,
        name = 'airlines',
        value = df,
        append = TRUE,
        row.names = FALSE
    )
}
# close the connection
dbDisconnect(con)
```

The third method - usage of `bigmemory` was attempted. Though, due to some error on my system, I could not get the read function to work correctly. Thus, I could not investigate it's usage.

Comparison of `%do` and `%dopar`

Since I only have a four core machine, there's a chance that a plain `%do` might fare better than a `%dopar` given that the latter has some overhead associated with creation of the parallel executors. I'm going to test this first using `microbenchmark`. The calculations are so quick, that we must run 10ˆ6 iterations to see some differences in execution time.

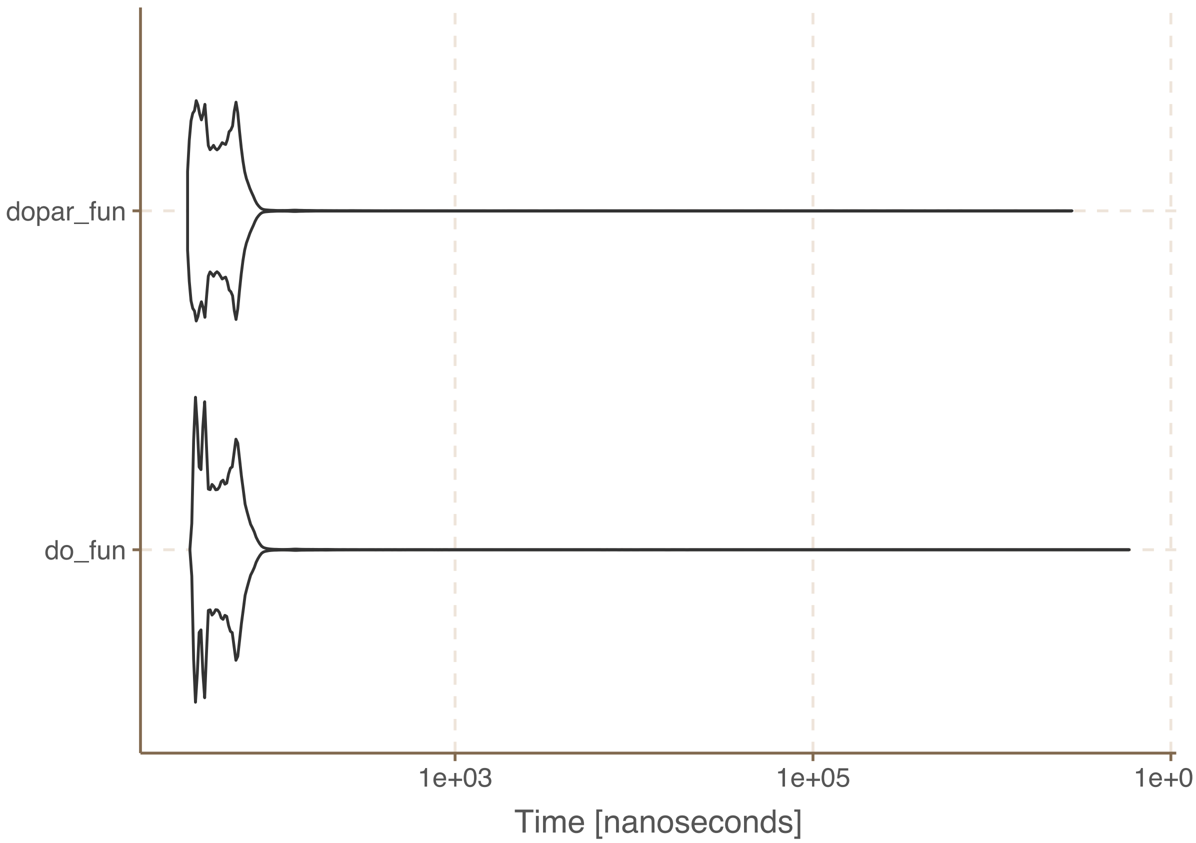
```
df$depHours <- floor(df$CRSDepTime/100)
df$depHours[df$depHours==24] <- 0
splits <- split(1:nrow(df),df$depHours)

myProbs <- c(0.50,0.75,0.90)

doMC::registerDoMC(4)

do_fun <- function(){
        foreach(hour = splits, .combine = cbind) %do% {
                        quantile(df[hour,"DepDelay"][[1]], myProbs, na.rm=T)}
}
dopar_fun <- function(){
        foreach(hour = splits, .combine = cbind) %dopar% {
                        quantile(df[hour,"DepDelay"][[1]], myProbs, na.rm=T)}
}
benchmark <- microbenchmark::microbenchmark(
        do_fun,
        dopar_fun,
        times= 1000000L
        )
```

We can see that, on average, a straight `%do%` runs 38 ns, while `%dopar%` runs 40 ns. The difference is too less to worry about for this dataset size, to be honest. It might be different for a much larger dataset though.

```
benchmark
```

```
## Unit: nanoseconds
##       expr min lq     mean median uq     max neval cld
##     do_fun  33 39 87.37443     48 59 5846084 1e+06   a
##  dopar_fun  32 38 72.79741     46 58 2802263 1e+06   a
```

```
microbenchmark:::autoplot.microbenchmark(benchmark)
```



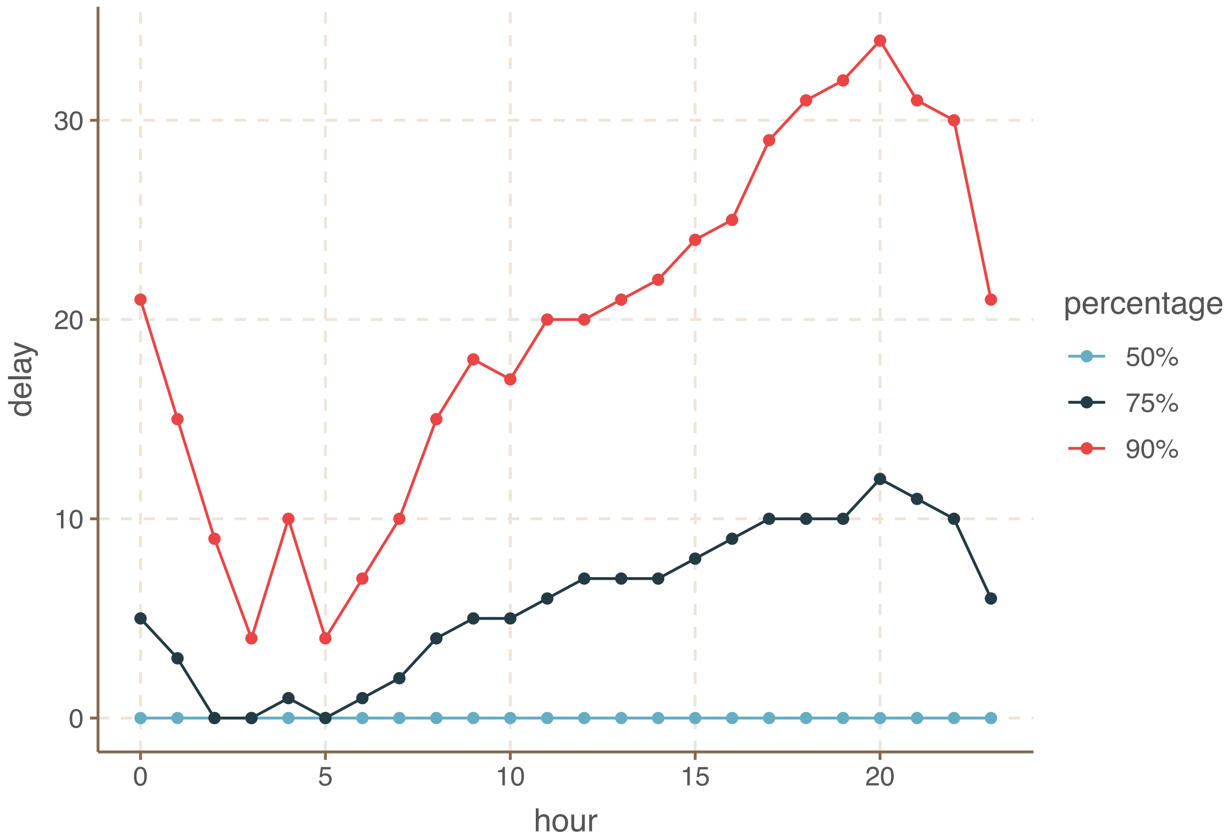
Plots

Plot #1: run the analysis as in the book by hour of the day but with myProbs <- c(0.50,0.75,0.90).

Here, the hours are calculated using the `CRSDepTime` variable as shown in the book. Thereafter, for each hour, a foreach-do is executed. The plot shows that the delays are the highest for the evening hours, and lowest in the early hours of the morning.

```
df$depHours <- floor(df$CRSDepTime/100)
df$depHours[df$depHours==24] <- 0
splits <- split(1:nrow(df),df$depHours)

myProbs <- c(0.50,0.75,0.90)

delayQuantiles <- foreach(hour = splits, .combine = cbind) %do% {
        quantile(df[hour,"DepDelay"][[1]], myProbs, na.rm=T)
}

colnames(delayQuantiles) <- names(splits)
melted <- reshape2::melt(delayQuantiles)
names(melted) <- c("percentage","hour","delay")
melted %>%
```

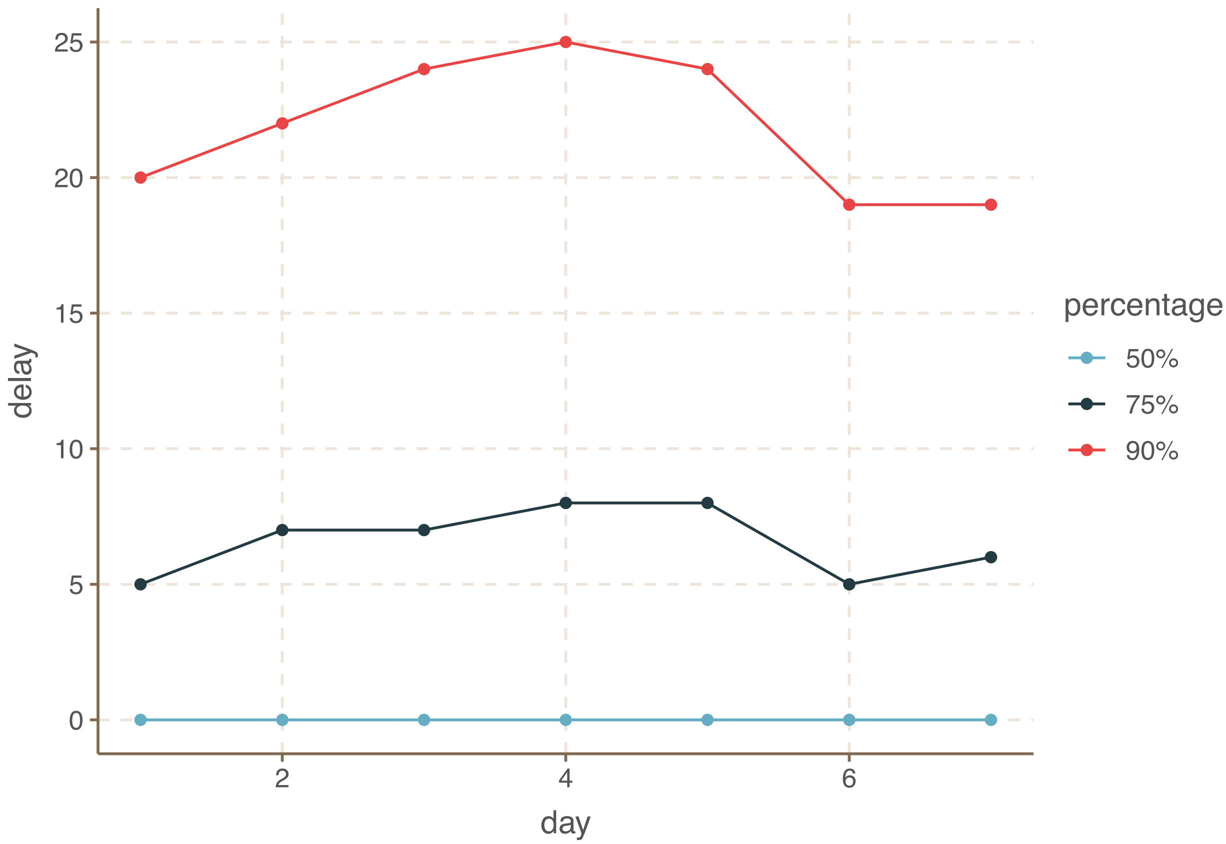
```
        ggplot(aes(y=delay,x=hour,color=percentage))+
        geom_line()+
        geom_point()
```



Plot #2: run the analysis but change it to day of the week and use myProbs <- c(0.50,0.75,0.90).

To get the similar graph for day of the week, we simply need to change the variable used for the splits to `DayOfWeek` as shown below. Here, we can see that the highest delays are 3, 4 and 5 which fall mid-week when the travel might be the highest.

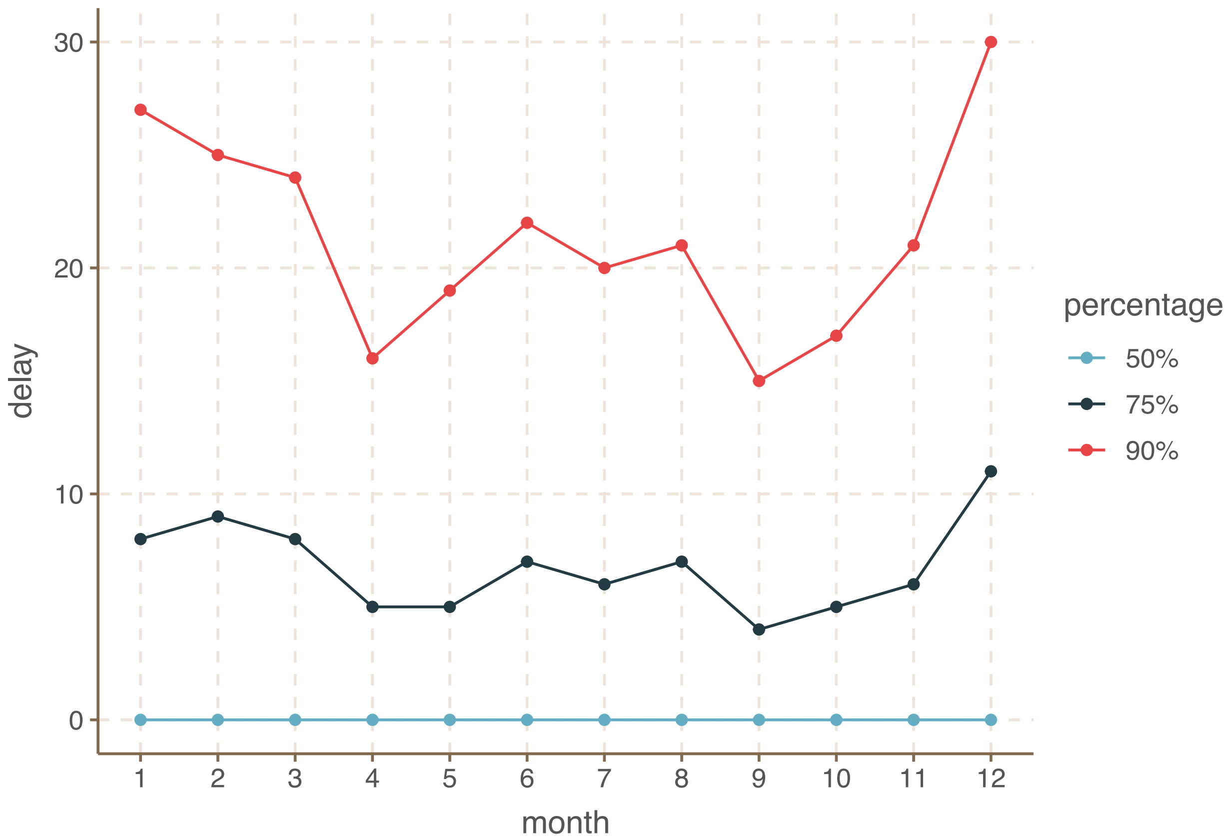
```
splits <- split(1:nrow(df),df$DayOfWeek)
myProbs <- c(0.50,0.75,0.90)
delayQuantiles <- foreach(day = splits, .combine = cbind) %do% {
        quantile(df[day,"DepDelay"][[1]], myProbs, na.rm=T)
}
colnames(delayQuantiles) <- names(splits)
melted <- reshape2::melt(delayQuantiles)
names(melted) <- c("percentage","day","delay")
melted %>%
        ggplot(aes(y=delay,x=day,color=percentage))+
        geom_line()+
        geom_point()
```



Plot #3: run the analysis but change it to month of the year and use myProbs <- c(0.50,0.75,0.90).

Now, we can use the `Month` variable for the splits. The same code as before is used. Now, we see that the delays are highest in Dec and Jan, during the holiday season, when travel is the highest.

```
splits <- split(1:nrow(df),df$Month)
myProbs <- c(0.50,0.75,0.90)
delayQuantiles <- foreach(month = splits, .combine = cbind) %do% {
        quantile(df[month,"DepDelay"][[1]], myProbs, na.rm=T)
}
colnames(delayQuantiles) <- names(splits)
melted <- reshape2::melt(delayQuantiles)
names(melted) <- c("percentage","month","delay")
melted %>%
        ggplot(aes(y=delay,x=month,color=percentage))+
        geom_line()+
        geom_point()+
        scale_x_continuous(breaks = seq(1,12,1))
```


30
20
10
0
delay
1
2
3
4
5
6
7
8
9
10
11
12
month
percentage
50%
75%
90%