Text data and data.table

Yufeng Huang

Associate Professor of Marketing, Simon Business School

August 4, 2022

Data get larger over time

- We've been dealing with datasets below 50MB
- In reality sample size, number of variables and diversity of data type is much higher
- My own example:
  - household purchase record for retail clothing, 2007-2015: **400MB**
  - flickr.com user behavior and camera purchase: **1.2G**
  - Yelp.com user reviews (Yelp academic dataset): **1.3G**
  - grocery shopper's purchase records from a select set of retailers (2001-2007): ~**4G** only in the yogurt category
  - New York City yellow cab travel record in Jan-Jun, 2015: **11.5G**
  - Airbnb data (still on-going): ~**900G**

What we do today

- Data format: tabular/non-tabular, text/non-text
- Reading and writing from/to text data
- The *data.table* package:
  - speed
  - notation
  - examples
- Quick preview of *feather*

Tabular data

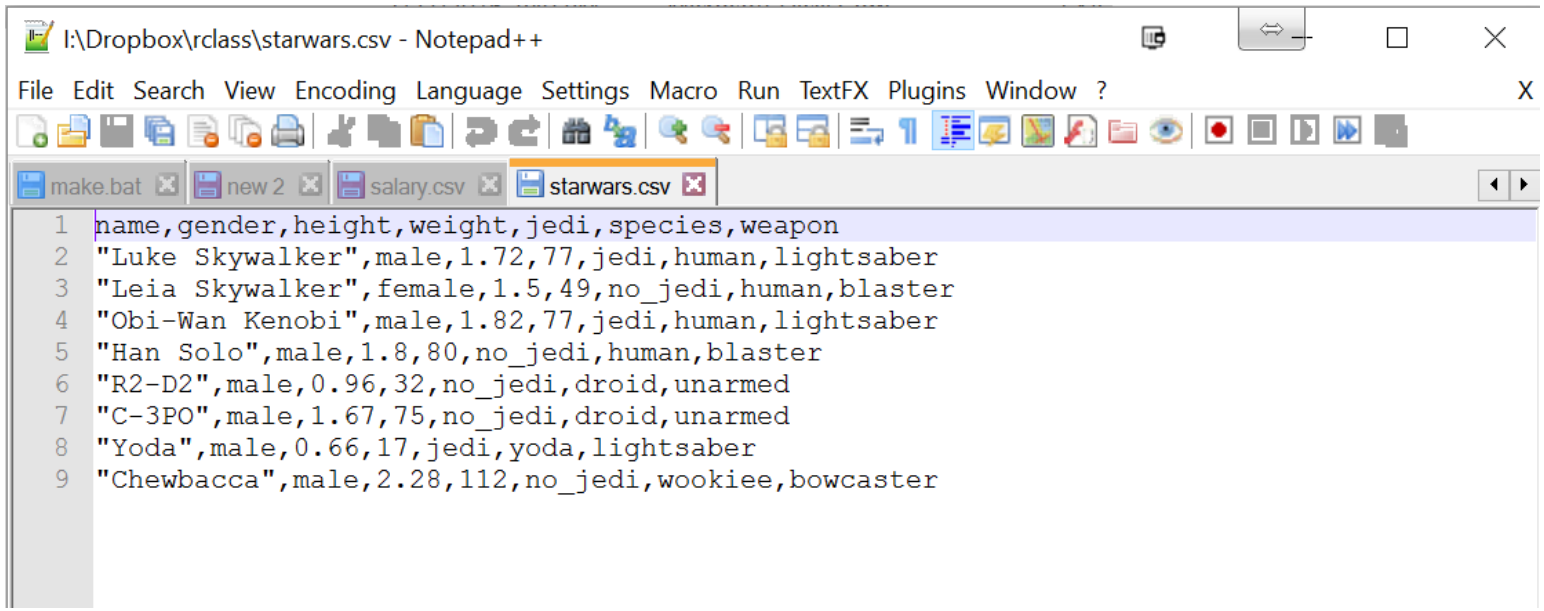
Tabular data are tables

```
# explicitly set work directory
setwd("L:/Dropbox/Teaching/Programming_for_analytics/2022/lecture_8/")
#    note: forward slash (/), backward slash will not (directly) work

# we've seen how a dataset can be imported
df <- read.csv("starwars.csv")
str(df)

## 'data.frame': 8 obs. of  7 variables:
##  $ name   : chr  "Luke Skywalker" "Leia Skywalker" "Obi-Wan Kenobi" "Han Solo" ...
##  $ gender : chr  "male" "female" "male" "male" ...
##  $ height : num  1.72 1.5 1.82 1.8 0.96 1.67 0.66 2.28
##  $ weight : int  77 49 77 80 32 75 17 112
##  $ jedi   : chr  "jedi" "no_jedi" "jedi" "no_jedi" ...
##  $ species: chr  "human" "human" "human" "human" ...
##  $ weapon : chr  "lightsaber" "blaster" "lightsaber" "blaster" ...

# stored in the form of data frame
df[1:6, 1:4]

##             name gender height weight
## 1 Luke Skywalker   male   1.72     77
## 2 Leia Skywalker female   1.50     49
## 3 Obi-Wan Kenobi   male   1.82     77
## 4       Han Solo   male   1.80     80
## 5          R2-D2   male   0.96     32
## 6          C-3PO   male   1.67     75
```

So we can find where things are

```
# info about luke?
df[df$name == "Luke Skywalker", ]

##              name gender height weight jedi species     weapon
## 1 Luke Skywalker   male   1.72     77 jedi   human lightsaber

# height of the first 3 obs
df$height[1:3]

## [1] 1.72 1.50 1.82

# who's taller than 1.80?
as.character(df$name[df$height > 1.80])

## [1] "Obi-Wan Kenobi" "Chewbacca"
```

Raw data can be stored in csv formats



We can save a data frame as a tabular dataset

```
# We can save data frame to file using write.table
#   here: write to tab-delimited text file
#   note: "\\t" means "tab"
#   note 2: we explicitly save the file at an *absolute* folder
write.table(df,
    "L:/Dropbox/Teaching/Programming_for_analytics/2022/lecture_8/starwars.txt",
    sep = "\t")
```

Text data: in this case tab-delimited

starwars - Notepad

File Edit Format View Help

```
"name"  "gender"        "height"        "weight"        "jedi"  "species"       "weapon"
"1"     "Luke Skywalker"        "male"  1.72    77      "jedi"  "human" "lightsaber"
"2"     "Leia Skywalker"        "female"        1.5     49      "no_jedi"       "human" "blaster"
"3"     "Obi-Wan Kenobi"        "male"  1.82    77      "jedi"  "human" "lightsaber"
"4"     "Han Solo"      "male"  1.8     80      "no_jedi"       "human" "blaster"
"5"     "R2-D2" "male"  0.96    32      "no_jedi"       "droid" "unarmed"
"6"     "C-3PO" "male"  1.67    75      "no_jedi"       "droid" "unarmed"
"7"     "Yoda"  "male"  0.66    17      "jedi"  "yoda"  "lightsaber"
"8"     "Chewbacca"     "male"  2.28    112     "no_jedi"       "wookiee"       "bowcaster"
```

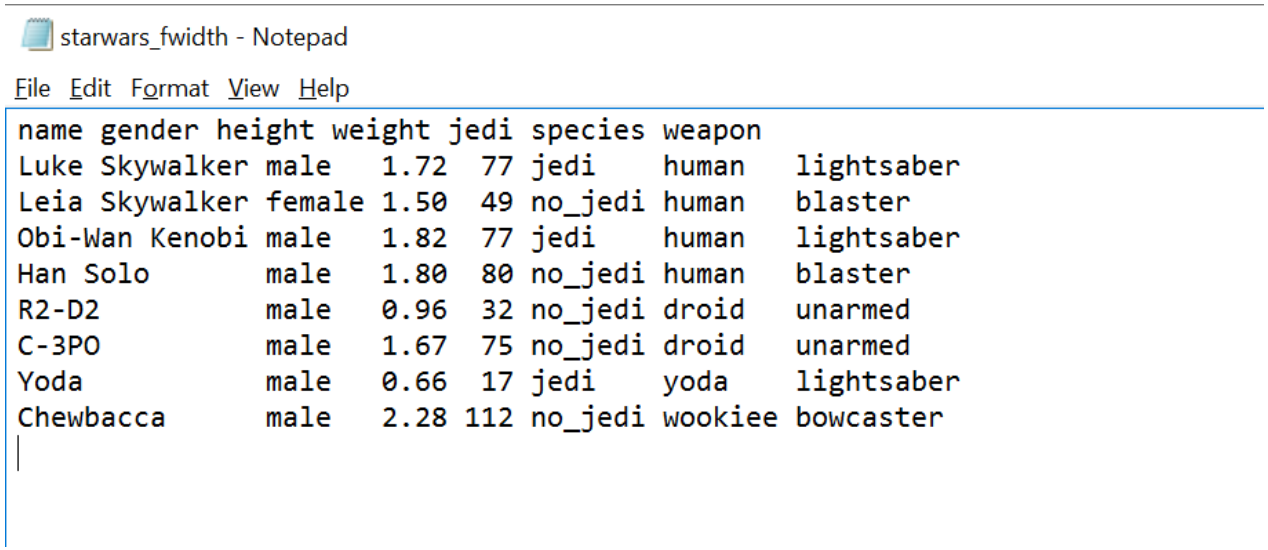
Text data do not have to be tab- or comma- delimited

```
# another common approach is space delim
write.table(df, "starwars_space.txt", sep = " ")

# or, can even use weird separators like "$"
write.table(df, "starwars_dollar.txt", sep = "$")

# read it back
df3 <- read.table("starwars_dollar.txt", sep = "$")

# still the same data
df3[1:3, 1:3]

##              name gender height
## 1 Luke Skywalker   male   1.72
## 2 Leia Skywalker female   1.50
## 3 Obi-Wan Kenobi   male   1.82
```

Tabular data can be stored in fixed-width

```
# define file location
loc <- "starwars_fwidth.txt"

# define width parameter
w <- c(14, 6, 4, 3, 7, 7, 10)

# write df into fixed width with package 'gdata'
library('gdata')
write.fwf(df, file = loc, width = w)
```

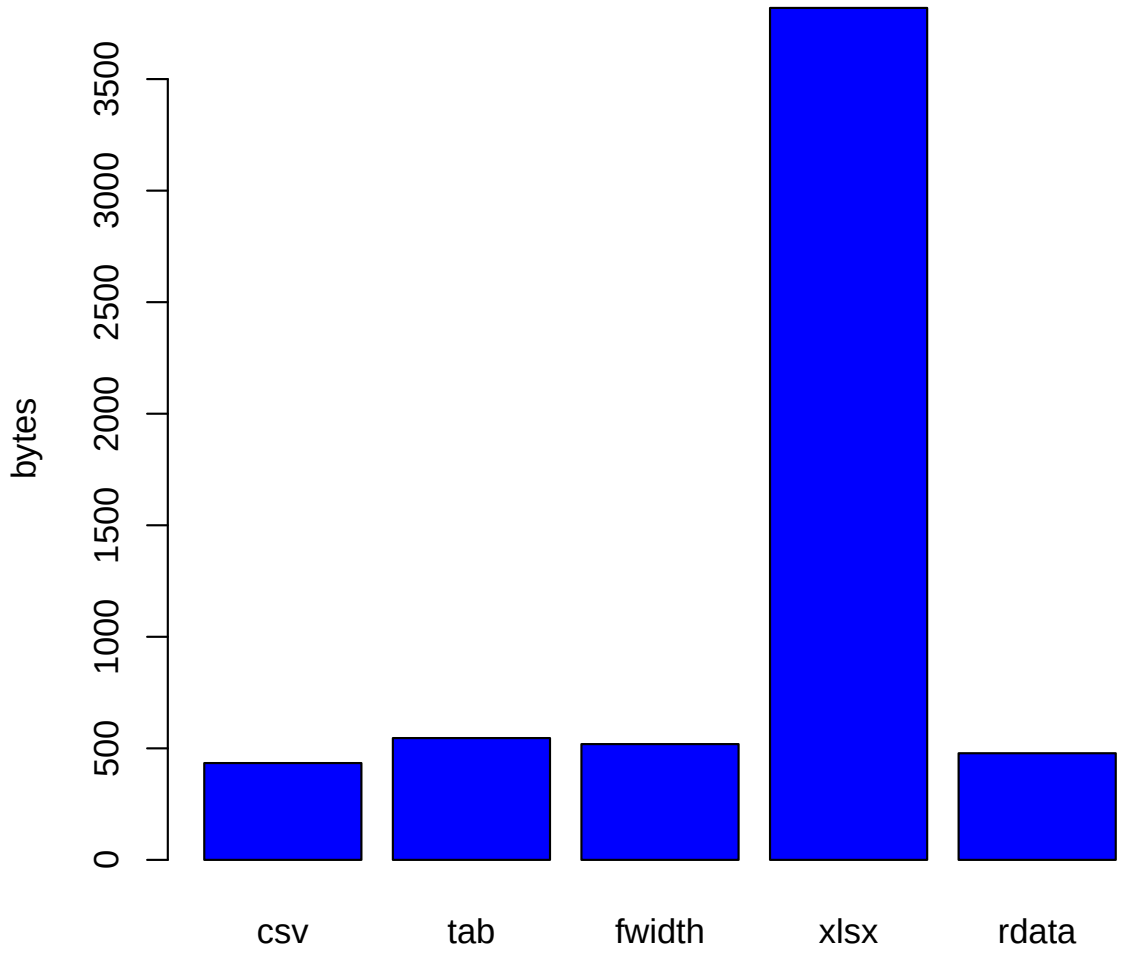
Fix width is easier on human eyes (but is more difficult to read or write)



Now compare size of different files

```
# read file info using file.info()
info.list <- list(
    file.info("starwars.csv"),
    file.info("starwars.txt"),
    file.info("starwars_fwidth.txt"),
    file.info("starwars.xlsx"),
    file.info("starwars.rdata")
)
# what are the sizes for the files?
fs <- data.frame(
    fileformat = c("csv", "tab", "fwidth", "xlsx", "rdata"),
    size = c(info.list[[1]]$size,
        info.list[[2]]$size,
        info.list[[3]]$size,
        info.list[[4]]$size,
        info.list[[5]]$size
    )
)
```

```r
# let's visualize the result
barplot(fs$size, names.arg = fs$fileformat,
    ylab = "bytes", col = "blue")
```



Size compared

- Everything similar size except for the Excel file
- Minimal file size for the information
  - the csv file has 418 characters including delimitor, so 418 bytes
  - adding 8 new lines "\n" which takes additional 8*2=16 bytes
  - final file size is 434 bytes, no waste at all
- Other files
  - tab data is 546 bytes because tab "\t" takes 2 bytes
  - Rdata is 478 bytes; quite efficient
  - Excel file has exactly the same information but is 8.8 times the size of csv

Non-tabular text data example: EPS and BMP

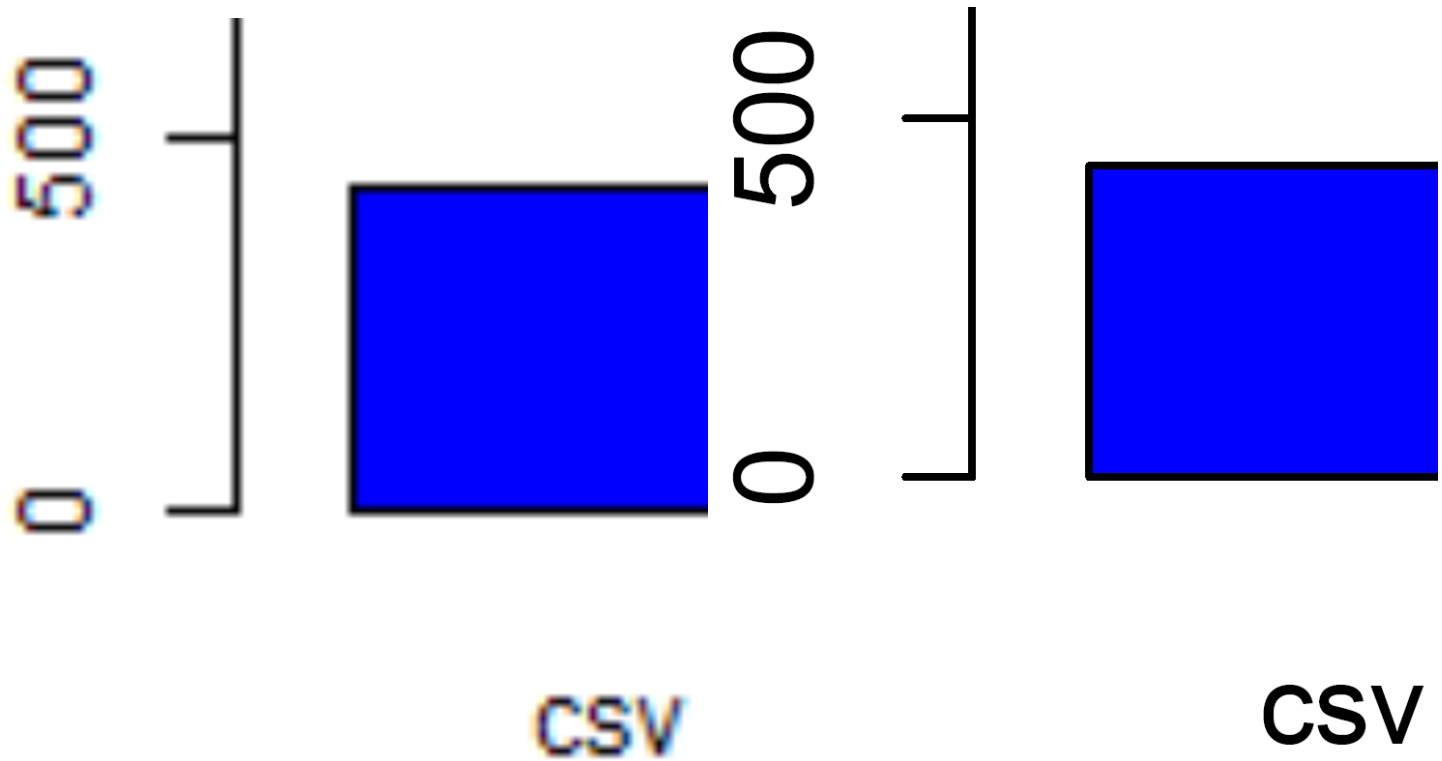
Export a figure to Encapsulated PostScript (EPS)

```
# load graphic devices library
library('grDevices')

# create an EPS file
postscript("example_bar.eps",,
    width = 480, height = 480, horizontal = F)
barplot(fs$size, names.arg = fs$fileformat,
    ylab = "bytes", col = "blue")
dev.off()

## pdf
##   2
```

Export the same figure to Bit MaP (BMP)

```
# create a BMP file
bmp("example_bar.bmp",
    width = 480, height = 480)
barplot(fs$size, names.arg = fs$fileformat,
    ylab = "bytes", col = "blue")
dev.off()

## pdf
##   2
```

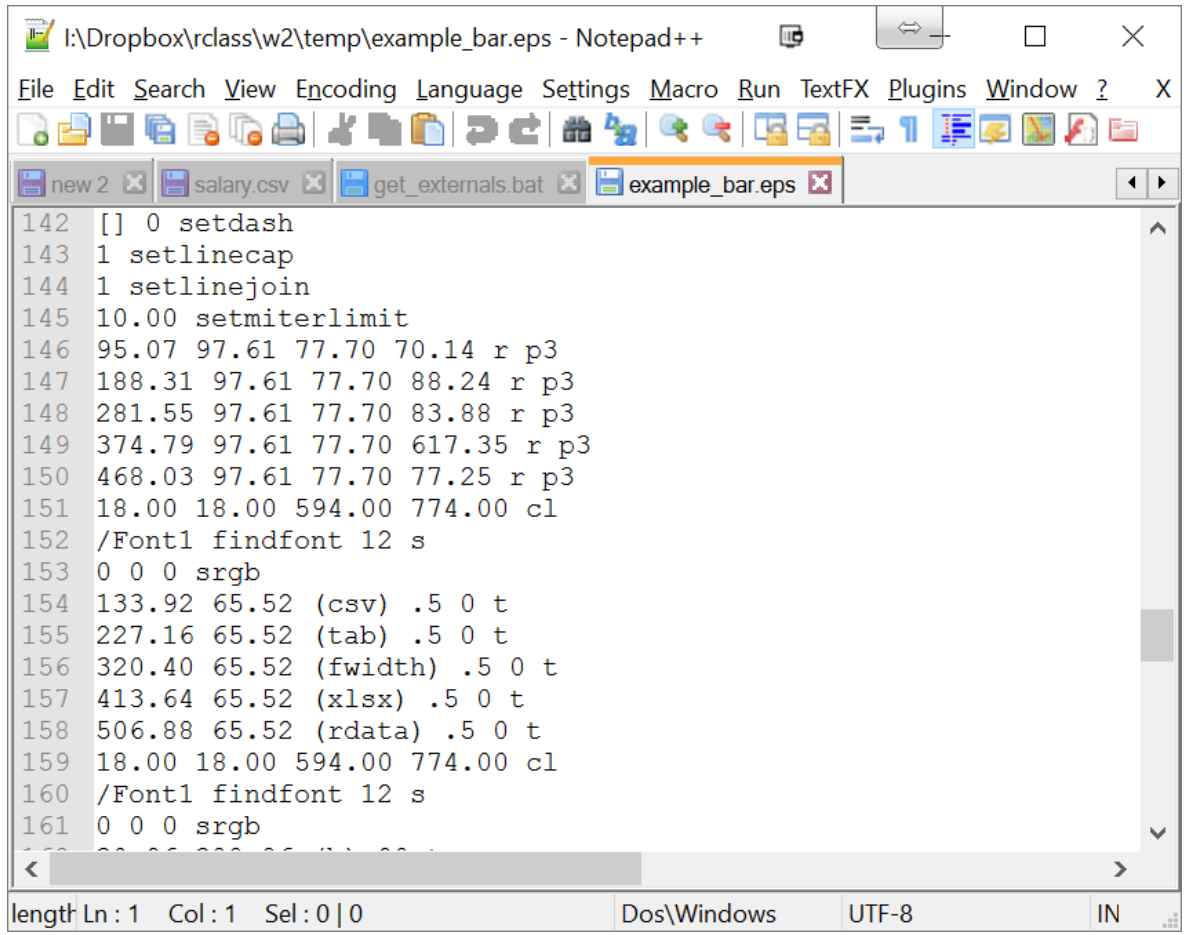
BMP = 227kb; EPS = 7kb[1]



which one is BMP and which one is EPS?

[1] To be exact, you're seeing a **rendered** EPS

Why? BMP stores RGB for each pixel; EPS stores instructions on how to draw the figure



So far

- Data is stored in files
  - takes space to store
  - takes time to read it
  - lots of data => file format matters A LOT!
- Text is a universal and efficient way to store data
  - can store tables (tabular data)
  - can store more general information (non-tabular data)
- Two examples where text data is vastly superior
  - For tabular: CSV (and other text) data is much smaller than Excel data
  - For graphics: EPS graphs is much smaller than BMP graphs

data.table()

data.table()

- Speed
- Notation
- Examples

Standard read.table

```
# library
library('data.table')

# data.frame way (bb for billboard, file size ~350MB)
t0 <- proc.time()
df_bb <- read.table("billboard_full.txt", sep = "\t",   # separator is tab
        quote = "",     # will read quote literally
        header = T,     # will read first line as header
        stringsAsFactors = F,
        fill = TRUE)    # fill: uneven columns will be filled

# check time
t1 <- proc.time()
t1 - t0

##    user  system elapsed
##   11.31    0.39   11.72

# check data
head(df_bb, n = 3)

##       metro   country countrycode  starttime    endtime        artist rank
## 1 Melbourne Australia          au 1233489600 1234094400     Radiohead    1
## 2 Melbourne Australia          au 1233489600 1234094400 Kings of Leon    2
## 3 Melbourne Australia          au 1233489600 1234094400      Coldplay    3
##   playcount                                  mbid
## 1        NA a74b1b7f-71a5-4011-9441-d0b5e4122711
## 2        NA 6ffb8ea9-2370-44d8-b678-e9237bbd347b
## 3        NA cc197bad-dc9c-440d-a5b5-d52ba2e14234

# remove df_bb because we won't use it later
rm(df_bb)
```

Compared: fread()

```
# data.table way
t0 <- proc.time()
dt_bb <- fread("billboard_full.txt", sep = "\t", header = T, stringsAsFactors = F)

 ## Warning in fread("billboard_full.txt", sep = "\t", header = T, stringsAsFactors = F): Found
   and resolved improper quoting out-of-sample.  First healed line 699661:  «Changsha China cn
    1269172800 1269777600 "Weird Al" Yankovic 58 ».  If the fields are not quoted (e.g.  field
         separator does not appear within any field), try quote="" to avoid this warning.

# check time
t1 <- proc.time()
t1 - t0 # FASTER!

##    user  system elapsed
##    1.45    0.08    0.96

# check data
head(dt_bb, n = 3)

##        metro   country countrycode  starttime    endtime        artist rank
## 1: Melbourne Australia          au 1233489600 1234094400     Radiohead    1
## 2: Melbourne Australia          au 1233489600 1234094400 Kings of Leon    2
## 3: Melbourne Australia          au 1233489600 1234094400      Coldplay    3
##    playcount                                  mbid
## 1:        NA a74b1b7f-71a5-4011-9441-d0b5e4122711
## 2:        NA 6ffb8ea9-2370-44d8-b678-e9237bbd347b
## 3:        NA cc197bad-dc9c-440d-a5b5-d52ba2e14234

# you'll notice the small colon at every row,
#    indicating that this is data table
```

Data table as data frames

Data tables are recognized as data frames

```
# let's do it on a smaller scale (your assignment 1 data)
url <- 'https://dl.dropboxusercontent.com/s/q6qzbfa1tdcqv6v/belgium_atm.csv'
dt_atm <- fread(url, stringsAsFactors = F)

head(dt_atm)

##    population numATMs   ATMwithdr withdrvalue unemprate numbranches
## 1:       3722       1   .25542593 79.13402557 0.0728676       0.500
## 2:       7006       2 1.837865114 102.6663437 0.0226948       0.500
## 3:       4234       0     missing     missing 0.0273973       0.125
## 4:       6229       0     missing     missing 0.0244020       0.750
## 5:      10303       1 .6062539816 98.93833923 0.0284383       0.375
## 6:       7424       0     missing     missing 0.0373114       0.875

#    note the colon ':' in each row, this indicates data table

# it is recognized as a data frame
is.data.frame(dt_atm)

## [1] TRUE
```

Some data frame notations work with data table

```
# subset by rows
dt_atm[1:3, ]

##    population numATMs   ATMwithdr withdrvalue unemprate numbranches
## 1:       3722       1   .25542593 79.13402557 0.0728676       0.500
## 2:       7006       2 1.837865114 102.6663437 0.0226948       0.500
## 3:       4234       0     missing     missing 0.0273973       0.125

# refer to columns
head(dt_atm$numATMs)

## [1] 1 2 0 0 1 0

# also this can refer to columns (as elements in a list)
head(dt_atm[["numATMs"]])

## [1] 1 2 0 0 1 0

# so basically row notations work, (some) list notations work
```

Difference 1: column notations

```
# same as in a data frame (updated data table supports this)
dt_atm[, c("numATMs", "population")]

##      numATMs population
##   1:       1       3722
##   2:       2       7006
##   3:       0       4234
##   4:       0       6229
##   5:       1      10303
##  ---
## 655:       0        601
## 656:       0       1028
## 657:       0       2033
## 658:       2      15521
## 659:       0       5941

# single-bracket list subsetting does NOT work
dt_atm[c(1, 2)]

##    population numATMs   ATMwithdr withdrvalue unemprate numbranches
## 1:       3722       1   .25542593 79.13402557 0.0728676         0.5
## 2:       7006       2 1.837865114 102.6663437 0.0226948         0.5

# and I'll explain why
```

Difference 2: lots of operations now work within the data table object

```
# delete the population column
dt_atm[, population := NULL]    # delete column

# construct a new column by another column
dt_atm[,
        avg_withdrawal := mean(as.numeric(ATMwithdr), na.rm = T),
        by = numATMs]

 ## Warning in `[.data.table`(dt_atm, , `:=`(avg_withdrawal, mean(as.numeric(ATMwithdr), :  NAs
                                    introduced by coercion

# print
dt_atm

##      numATMs   ATMwithdr withdrvalue unemprate numbranches avg_withdrawal
##   1:       1   .25542593 79.13402557 0.0728676       0.500      0.7576123
##   2:       2 1.837865114 102.6663437 0.0226948       0.500      1.0025904
##   3:       0     missing     missing 0.0273973       0.125            NaN
##   4:       0     missing     missing 0.0244020       0.750            NaN
##   5:       1 .6062539816 98.93833923 0.0284383       0.375      0.7576123
##  ---
## 655:       0     missing     missing 0.0217658       0.125            NaN
## 656:       0     missing     missing 0.0217658       0.250            NaN
## 657:       0     missing     missing 0.0217658       0.500            NaN
## 658:       2 .6984899044 110.1268387 0.0231947       0.875      1.0025904
## 659:       0     missing     missing 0.0231947       0.875            NaN

# and I'll explain these in more detail
```

Notation: DT[i, j, by]

DT[i, j, by]

- For a data table, say 'DT':
  - some standard data frame notations work
  - some do not because data.table notation overrides data frame notations
- Specifically, DT[i, j, by]
  - i: which rows do you want?
  - j: what do you want to do with the rows?
  - by: do this by which index?

DT[i]

```
# read ATM data again
url <- 'https://dl.dropboxusercontent.com/s/q6qzbfa1tdcqv6v/belgium_atm.csv'
dt_atm <- fread(url, stringsAsFactors = F)

# DT[i] gets the rows
dt_atm[1:3]     # row 1-3

##    population numATMs   ATMwithdr withdrvalue unemprate numbranches
## 1:       3722       1   .25542593 79.13402557 0.0728676       0.500
## 2:       7006       2 1.837865114 102.6663437 0.0226948       0.500
## 3:       4234       0     missing     missing 0.0273973       0.125

dt_atm[c(4, 1, 8)]      # row 4, 1, 8, in order (also note row num changed)

##    population numATMs   ATMwithdr withdrvalue unemprate numbranches
## 1:       6229       0     missing     missing 0.0244020       0.750
## 2:       3722       1   .25542593 79.13402557 0.0728676       0.500
## 3:       7129       1 .8399091363 89.14505768 0.0352083       0.625

# or gets rows by condition
dt_atm[numATMs <= 2 & population <= 5000]

##      population numATMs ATMwithdr withdrvalue unemprate numbranches
##   1:       3722       1 .25542593 79.13402557 0.0728676       0.500
##   2:       4234       0   missing     missing 0.0273973       0.125
##   3:       2868       0   missing     missing 0.0355649       0.375
##   4:       4105       0   missing     missing 0.0326871       0.250
##   5:       4992       0   missing     missing 0.0291726       0.375
##  ---
## 245:       3529       0   missing     missing 0.0208911       0.500
## 246:       2862       0   missing     missing 0.0217658       0.500
## 247:        601       0   missing     missing 0.0217658       0.125
## 248:       1028       0   missing     missing 0.0217658       0.250
## 249:       2033       0   missing     missing 0.0217658       0.500

#    note that we don't need to say dt_atm$numATM <= 2 as with data frames
```

DT[i, j]

```
# argument j: what do you want to do with those rows?

# keep all columns
dt_atm[1:3, ]

##    population numATMs   ATMwithdr withdrvalue unemprate numbranches
## 1:       3722       1   .25542593 79.13402557 0.0728676       0.500
## 2:       7006       2 1.837865114 102.6663437 0.0226948       0.500
## 3:       4234       0     missing     missing 0.0273973       0.125

# keep variable population (now a vector)
dt_atm[1:3, population]

## [1] 3722 7006 4234

# I can still use data frame notation (only in recent versions)
dt_atm[1:3, "population"]

##    population
## 1:       3722
## 2:       7006
## 3:       4234
```

DT[i, j]

```
# can run a function instead of just taking columns

# example: average number of ATMs in smaller towns
dt_atm[population <= 5000, mean(numATMs)]

## [1] 0.09638554

# data frame equivalent will be
mean(dt_atm[dt_atm$population <= 5000, ]$numATMs)

## [1] 0.09638554
```

Special operator := to create new columns

```
# create a variable (in the 'j' part)
#    ':=' means assignment by reference
dt_atm[, pop := population / 1000]      # population in thousands

# create multiple variable
#     turns population into thousands, turns two variables into numeric, and deletes pop
dt_atm[, `:=`(population = population / 1000,
                ATMwithdr = as.numeric(ATMwithdr),
                withdrvalue = as.numeric(withdrvalue),
                pop = NULL)]
head(dt_atm, n = 3)     #  and co (country code)

##    population numATMs ATMwithdr withdrvalue unemprate numbranches
## 1:      3.722       1 0.2554259    79.13403 0.0728676       0.500
## 2:      7.006       2 1.8378651   102.66634 0.0226948       0.500
## 3:      4.234       0        NA          NA 0.0273973       0.125
```

Special character .() to group function calls

```
# .() in DT[i]: keep multiple variable by grouping statements
dt_atm[1:3, .(numATMs, population, unemprate)]

##    numATMs population unemprate
## 1:       1      3.722 0.0728676
## 2:       2      7.006 0.0226948
## 3:       0      4.234 0.0273973

#    ".()" groups expressions in the j part
#    a bit like aggregate(x, by = list(var = mean(var)))

# .() in DT[i,j]: multiple summary statistics?
dt_atm[numATMs >= 1 & population <= 10000,
       .(avg_atm = mean(numATMs), var_atm = var(numATMs))]

##     avg_atm   var_atm
## 1: 1.567742 0.7122247
```

DT[i, j, by]

```
# at this point you can see the full notation DT[i, j, by]
#    example: what is the average withdraw value by number of ATMs?
#    added condition: for towns with population > 2000

dt_atm[population >= 2,
        .(ATMwithdr = mean(ATMwithdr), withdrvalue = mean(withdrvalue)),
        numATMs]

##    numATMs ATMwithdr withdrvalue
## 1:       1 0.7505181   101.05752
## 2:       2 1.0025904   100.37617
## 3:       0        NA          NA
## 4:       4 1.0827598    96.25565
## 5:       3 1.1169177    99.25264
## 6:       5 1.1931812    98.15437
## 7:       6 0.7416071    92.73267
```

Example 2: Billboard data

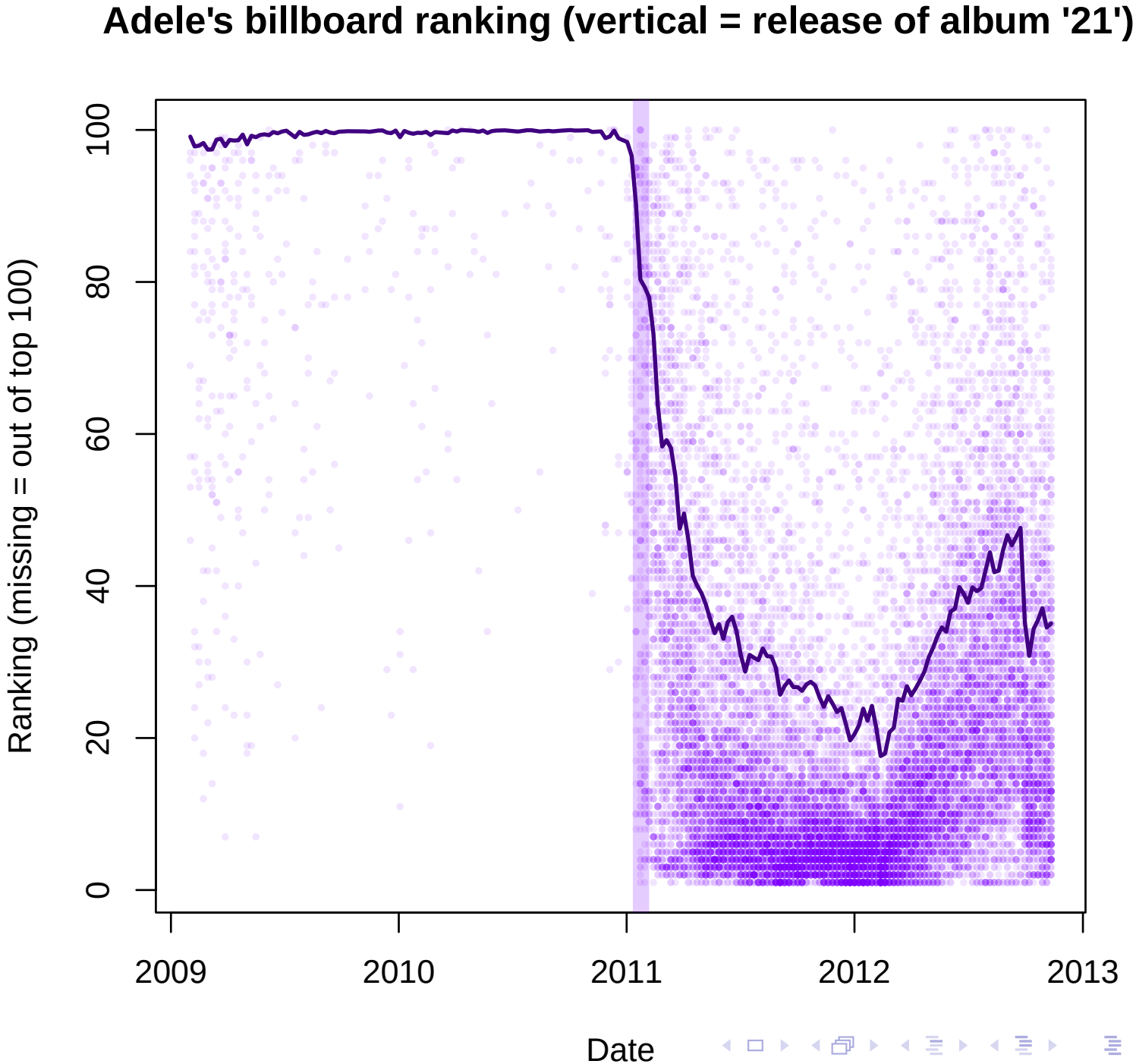
```
# now work with larger (still small) dataset on Billboard ranking
head(dt_bb, n = 2)      # mbid is universal albumn ID (standard in music industry)

##        metro   country countrycode  starttime    endtime        artist rank
## 1: Melbourne Australia          au 1233489600 1234094400     Radiohead    1
## 2: Melbourne Australia          au 1233489600 1234094400 Kings of Leon    2
##    playcount                                  mbid
## 1:        NA a74b1b7f-71a5-4011-9441-d0b5e4122711
## 2:        NA 6ffb8ea9-2370-44d8-b678-e9237bbd347b

t0 <- proc.time()

# convert starttime to date
#    numeric means seconds since 1970-1-1 07:00:00 (I will cover dates next week)
dt_bb$date <- as.Date(as.POSIXct(dt_bb$starttime, origin = "1970-1-1"))

# let's find subset artists who
#    1) ranked top 3
#    2) between September 2009 and August 2011
#    3) focus on the US market

# first data frame way (df_bb would take the same amt of time)
dt_sub <- subset(dt_bb, date >= "2009-09-01" & date <= "2011-08-31" &
        rank <= 3 & countrycode == "us")
freq.df <- aggregate(x = rank ~ artist, data = dt_sub, FUN = length)
names(freq.df)[2] <- "frequency"

# check time
t1 <- proc.time()
t1 - t0

##    user  system elapsed
##    0.15    0.03    0.18
```

Example 2: Billboard data

```
# now data table way
t0 <- proc.time()

# convert starttime to date (numeric means seconds)
#    we'll cover dates next week
dt_bb[, date := as.Date(as.POSIXct(dt_bb$starttime, origin = "1970-1-1"))]

freq_dt <- dt_bb[date >= "2009-09-01" & date <= "2011-08-31" & rank <= 3 & countrycode == "us", # i
        .(frequency = length(rank)),    # j
        artist] # by

t1 <- proc.time()
t1 - t0

##    user  system elapsed
##    0.14    0.06    0.21
```

Example 2 con'd: Let's find Adele's average ranking

```
# not every artist is on top 100 in every week
#    assume that any time Adele falls out of billboard she is ranked at 100
#    (probably not accurate...)

# total number of metropolitan areas (where billboard is announced) is 234
num.metro <- length(unique(dt_bb$metro))

# define a function to calculate average ranking, counting each missing at 100
find_average_rank <- function(rank) {
        freq.not100 <- num.metro - length(rank)     # count metros that do NOT have Adele on bb
        (sum(rank) + mean(freq.not100)*100) / num.metro     # assume that missing values are 100
}               # so total is sum rank + missing * 100

# use key to sort data
setkey(dt_bb, date)

# subset and aggregate at the same time
adele_ranking <- dt_bb[artist == "Adele", .(rank = find_average_rank(rank)), date]

# plot
plot(dt_bb[artist == "Adele", date], dt_bb[artist == "Adele", rank],
        col = rgb(0.5, 0, 1, 0.1), pch = 16, cex = 0.5, # cex is marker size
        main = "Adele's billboard ranking (vertical = release of album '21')",
        xlab = "Date", ylab = "Ranking (missing = out of top 100)"
)
# add release date for '21'
abline(v = as.Date("2011-01-24"), col = rgb(0.5, 0, 1, .2), lwd = 8)
# add average ranking we calculated
points(adele_ranking$date, adele_ranking$rank, type = 'l', col = rgb(0.25, 0, 0.5, 1), lwd = 2)
```


Adele's billboard ranking (vertical = release of album '21')
Ranking (missing = out of top 100)
0
20
40
60
80
100
2009
2010
2011
2012
2013
Date

Feather

Feather

- Data.table still runs slow in large data applications, especially when a lot of data are read and written to/from the hard drive
- Feather aims to provide a solution by using a different data format, *.feather*, which is aimed to facilitate short-term read/write tasks
  - but CAUTION: not for long-term data storage

```r
# load the package
library(feather)

## Warning:  replacing previous import 'lifecycle::last_warnings' by 'rlang::last_warnings' when
                                        loading 'hms'

# set work directory to an SSD
setwd("D:/")

# generate a data frame (courtesy to Hadley Wickham)
x <- runif(1e6) # 1 million rows
x[sample(1e6, 1e5)] <- NA       # 10% NAs
df <- as.data.frame(replicate(10, x))
system.time(write_feather(df, 'test.feather'))

##    user  system elapsed
##    0.01    0.04    0.04

system.time(write.csv(df, 'test.csv'))

##    user  system elapsed
##   28.32    0.26   28.62
```

```
# test read time
system.time(read_feather('D:/test.feather'))

##    user  system elapsed
##    0.03    0.02    0.04

system.time(fread('D:/test.csv'))

##    user  system elapsed
##    0.30    0.42    0.08

system.time(read.table('D:/test.csv'))

##    user  system elapsed
##    2.94    0.12    3.06

# HUGE speed gain in both write and read!
```

Conclusion

- File format
  - read.table and write.table
  - csv, tab-demited data and fixed format data
  - text files have huge space/compatibility advantages (beyond tabular data)
- DT[i, j, by]
  - i: which rows do you want?
  - j: want them for what (which function to apply)?[2]
  - by: do this along which index variable(s)?
- Feather seems to be the future for temporary read and write data

[2] Recall that anything that happens is a function call