Exercise 3: Efficient data managment

Philipp Kling

17.06.2020

Contents


```
library(knitr)

## Global options
options(max.print="75")
opts_chunk$set(echo=FALSE,
                 cache=FALSE,
               prompt=FALSE,
               tidy=TRUE,
               comment=NA,
               message=FALSE,
               warning=FALSE)
opts_knit$set(width=75)
rm(list = ls())
```

Preparation

Installation of required packages

```
# install.packages('dplyr')
library(dplyr)
```

General overview

Packages to consider:

- **apply** function family to apply functios to vectors in a more efficient way than loops.

- **dplyr** package for human readable and efficient data managment.
- **vroom** package for fast reading of delimited datasets (e.g. large csv files). See e.g. vignette("vroom") for a description.
- **purrr** package for functional programming (similar to apply family). See https://purrr.tidyverse.org/ for more information.
- **data.table** package to work with large datasets and have a similar syntax as base R.

apply

Generally: apply a function to an object. This is generally faster than looping over data.

- **apply**: the base function apply(x, MARGIN, FUN) applies functions to matrices
  - x is the object
  - MARGIN is the dimension to which is should be applied (1 represents rows, 2 represents columns)
  - FUN is the function that should be applied to the object
- **lapply**: applies a function to a list (or vector and data.frames) and *returns a list*
  - "[" is a selector operator. lapply(listOfMatrices, "[", 1) selects the first row in each matrix of listOfMatrices, lapply(listOfMatrices, "[", , 1) selects the first column of each matrix in listOfMatrices.
- **sapply**: works like lapply, but tries to *simplify* the results (can also be achieved most of the time with "unlist(lapply(. . .))").
- Others:
  - **mapply**: from *multivariate* apply
  - **tapply**: apply a function to categories defined by a factor variable.
  - **rapply**: recursively applies a function to a list.
  - **vapply**: allows the specification of the return format.

Application: calculate means for wordcount and the page number of articles by section

```
load("../data/ex3/guardianapi_uknews_combined.Rda")
```

```
garticles_split <- split(garticles, garticles$sectionId)
class(garticles_split)  # a list of data.frames. For each section a separate data.frame
```

```
[1] "list"
```

```
head(names(garticles_split))  # The names of the list items. Each is the name of the sections
```

```
[1] "uk-news"     "travel"      "business"    "environment" "politics"
[6] "culture"
```

Do it in a for loop with R

```
system.time({
    res <- list()  # Placeholder for our results
    for (n in names(garticles_split)) {
        section_df <- garticles_split[[n]]  # Extract the dataframe representing the section
        res[[n]] <- data.frame(mean_wordcount = mean(section_df$wordcount, na.rm = TRUE),
            mean_pagenumber = mean(section_df$newspaperPageNumber, na.rm = TRUE),
            n = nrow(section_df))
```

```
    }
    res <- do.call(rbind, res)
    print(head(res))
})
```

```
            mean_wordcount mean_pagenumber    n
uk-news            825.419        13.16170 1661
travel            1221.165        13.24823  272
business          1067.757        29.98241 2106
environment        715.019        16.18851  843
politics          1370.844        10.58297 2404
culture           1309.587        14.86331  184
```

```
   user  system elapsed
  0.026   0.004   0.030
```

Do it with the apply family

```
system.time({
    # Define function beforehand
    get_means <- function(data_frame) {
        r <- data.frame(mean_wordcount = mean(data_frame$wordcount, na.rm = TRUE),
            mean_pagenumber = mean(data_frame$newspaperPageNumber, na.rm = TRUE),
            n = nrow(data_frame))
        return(r)
    }
    # Apply the function to the splitted data (list of data frames from above)
    res <- lapply(garticles_split, get_means)
    # Append the results together
    res <- do.call(rbind, res)
    print(head(res))
})
```

```
            mean_wordcount mean_pagenumber    n
uk-news            825.419        13.16170 1661
travel            1221.165        13.24823  272
business          1067.757        29.98241 2106
environment        715.019        16.18851  843
politics          1370.844        10.58297 2404
culture           1309.587        14.86331  184
```

```
   user  system elapsed
  0.016   0.000   0.016
```

dplyr

Data manipulation grammar for R. Its very user friendly and connects to many innovative developments in R.

- very fast in comparison to base R
- uses verbose language that makes code human readable (contrast to e.g. data.table)

Base functions

- **filter()** to select cases based on their values. Extracts rows that meet logical criteria.

- **arrange()** to reorder the cases. Orders rows by values of a column.
- **select()** to select variables based on their names. **rename()** to rename the columns of a data frame.
- **mutate()** and **transmute()** to add new variables that are functions of existing variables. Mutate keeps old variables, transmute removes the original rows.
- **summarise()** to summarise data into single row of values. This is a new data frame then and not an appended column.
- **sample_n()** and **sample_frac()** to take random samples.

More information and an overview: https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

```
rm(list = ls())
load("../data/ex3/guardianapi_uknews_combined.Rda")
```

Example: filter()

Extract rows that meet logical criteria.

```
# Base R
system.time({
    extracted <- garticles[garticles$sectionId == "business" & garticles$wordcount >
        700, ]
})
```

```
   user  system elapsed
  0.002   0.000   0.002
```

```
# dplyr
system.time({
    extracted <- dplyr::filter(garticles, sectionId == "business", wordcount >
        700)
})
```

```
   user  system elapsed
  0.004   0.000   0.004
```

Example: mutate()

Add new variables that are functions of existing variables.

```
# Base R
system.time({
    garticles$about_economics <- garticles$sectionId %in% c("money", "business")
})
```

```
   user  system elapsed
  0.000   0.000   0.001
```

```
# dplyr
system.time({
    garticles <- dplyr::mutate(garticles, about_economics = sectionId %in% c("money",
        "business"))
})
```

```
   user  system elapsed
  0.001   0.000   0.002
```

Example: group_by()

groupy_by() allows the splitting of a dataset into subgroups and which can then be used to be processed further.

Here we calculate the same summary statistics as before with base R and the apply function: we calculate the average wordcount and page number for each section.

```
# dplyr
system.time({
    by_section <- dplyr::group_by(garticles, sectionId)
    res <- dplyr::summarise(by_section, wordcount = mean(wordcount, na.rm = TRUE),
        pageNumber = mean(newspaperPageNumber, na.rm = TRUE), count = n())

})
```

```
   user  system elapsed
  0.003   0.000   0.003
```

```
res
```

```
# A tibble: 67 x 4
   sectionId   wordcount pageNumber count
   <fct>           <dbl>      <dbl> <int>
 1 uk-news          825.       13.2  1661
 2 travel          1221.       13.2   272
 3 business        1068.       30.0  2106
 4 environment      715.       16.2   843
 5 politics        1371.       10.6  2404
 6 culture         1310.       14.9   184
 7 us-news         1021.       13.1   265
 8 money            741.       35.9   393
 9 film             990.       17.0   529
10 world            957.       16.1  1672
# ... with 57 more rows
```

The pipe operator %>%

Dplyr (and other packages of the tidyverse) make use of a pipe operator %>%. Instead of saving results in intermediate data.frames or replacing the current data.frame in every line, we can transfer the output of one operation directly to the next. This saves space and makes the code more readable (because we need to use less parentheses).

```
by_section <- group_by(garticles, sectionId)
res <- summarise(by_section, wordcount = mean(wordcount, na.rm = TRUE), pageNumber = mean(newspaperPageI
    na.rm = TRUE), count = n())
head(res)
```

```
# A tibble: 6 x 4
  sectionId   wordcount pageNumber count
  <fct>           <dbl>      <dbl> <int>
1 uk-news          825.       13.2  1661
2 travel          1221.       13.2   272
3 business        1068.       30.0  2106
4 environment      715.       16.2   843
5 politics        1371.       10.6  2404
```

```
6 culture         1310.       14.9   184
```

Can be rewritten to:

```r
res <- garticles %>% group_by(sectionId) %>% summarise(wordcount = mean(wordcount,
    na.rm = T), pageNumber = mean(newspaperPageNumber, na.rm = TRUE), count = n())
head(res)
```

```
# A tibble: 6 x 4
  sectionId   wordcount pageNumber count
  <fct>           <dbl>      <dbl> <int>
1 uk-news          825.       13.2  1661
2 travel          1221.       13.2   272
3 business        1068.       30.0  2106
4 environment      715.       16.2   843
5 politics        1371.       10.6  2404
6 culture         1310.       14.9   184
```

data.table

data.table is a format that allows the storage and handling of large datasets. It is comparable to the data processing with dplyr. data.table seems to perform better for really large datasets and is more similar to the syntax of data.frames of base R. So if you are already familiar with the data.frame notation you might prefer this syntax.

Repetion data.frames:

- You can access the variables (columns) of a data.frame in two ways:
  - df$variable selects the column 'variable'
  - df[,c("variable")] selects the column 'variable' as well. This is more useful if you want to select multiple columns
- You subset data.frames by refering to conditions on their rows and columns. Everything that is before the "," refers to rows. Everything that's after the "," refers to the columns. Above we introduced the condition df$column == "variable".
  - df[df$variable == 1,] conditions on the rows. This would lead to a subsetting of the data.frame in a sense that we only get rows where a particular 'variable' has the value 1.

data.table

Now, data table allows for the same syntax, but follows a logic that is similar to SQL (language which is used in data bases). The additional processes are appended after the regular syntax leading to the general formulation:

- **data.table[subsetting, operation, grouping]**
  - **subsetting**: This is what happens before the first ','. This is similar to the data.frame.
    - data.table[variable1 == 'A'] selects only rows where the column 'variable1' is equal to 'A'.
  - **operations**: This is what happens after the first ',' and before the second ','. This is an enhanced conditioning on the columns as they not only allow for the subsetting of the columns, but also for their modification.
    - data.table[, .(variable1, variable2)] selects every row but only the two columns 'variable1' and 'variable2'. The .(...) represents an operation: select only two variables.
    - data.table[variable1 == "A", .(variable1, variable2)] selects only rows where the column 'variable1' is queal to 'A' (subsetting) and then displays only the two columns

'variable1' and 'variable2' (operation). If you like to, you could also use the old syntax: data.table[data.table$variable1 == "A", c("variable1","variable")]
    * data.table[variable1 == "A", .(mean_var2 = mean(variable2))] selects rows where the column 'variable1' is equal to 1 (subsetting) and then calculates a mean for those observations for the column 'variable2' (operation) and reports it as 'mean_var2'.
    * Note: .N can be used as an operation to calculate the number of observations.
  - **grouping**: Aggregations can be done in the third part of the syntax. E.g. apply an operation to each subgroup of the dataset.
    * data.table[variable1 == 'A', .N, by = variable2]: Calculates the number of observations (operation: .N) for each group existent in column 'variable2' and but only for observations where the column 'variable1' has the value 'A'.

```
rm(list = ls())
library(data.table)
load("../data/ex3/guardianapi_uknews_combined.Rda")
gadt <- data.table::as.data.table(garticles)
```

Suppose we would like to receive the same summary statistics (mean wordcount, mean page number) as in the above exercises. Using data.table commands, we would write it as follows:

```
# data.table
system.time({
  res <- gadt[,.(mean_wordcount = mean(wordcount, na.rm=TRUE),

                 mean_pagenumber = mean(newspaperPageNumber, na.rm = TRUE),

                 count = .N),
              by=sectionId]
  print(head(res))
})
```

```
     sectionId mean_wordcount mean_pagenumber count
1:     uk-news        825.419        13.16170  1661
2:      travel       1221.165        13.24823   272
3:    business       1067.757        29.98241  2106
4: environment        715.019        16.18851   843
5:    politics       1370.844        10.58297  2404
6:     culture       1309.587        14.86331   184
```

```
   user  system elapsed
  0.011   0.000   0.007
```