R programming, a gentle introduction

M1 IREF, M1 ROAD

Laurent Bergé

B_xSE, University of Bordeaux

Fall 2022

Data management:

Example

Problem: Herfindahl

Question

Does high tech employment becomes more geographically concentrated over time?

The file "High_tech_Employment_Eurostat.txt"★ contains information on the spatial distribution of high technology employment in Europe for several years.

The aim of this exercise is to compute the Herfindahl index for each country and year.

For a country c and a year y, the Herfindahl measures the level of spatial concentration of an activity and is defined as:

$$h_c, y = \sum_{i=1}^{n_c} s_{c,y,i}^2,$$

with $s_{c,y,i}$ the share of national employment of country c in year y located in region i.

An Herfindahl of 1 would mean that all employment is concentrated in only one region.

1. Import the data, and keep only observations at the NUTS2 level (roughly the regional level). NUTS2 level is a 4-characters geographical code of the form: XXYY with XX the country code and YY the NUTS2 code (e.g. FR17). Figure out and deal with data issues.
2. Compute the Herfindahl index for each country and year.
3. Plot the evolution of the Herfindahl for the EU5 (DE, BE, IT, FR, NL).

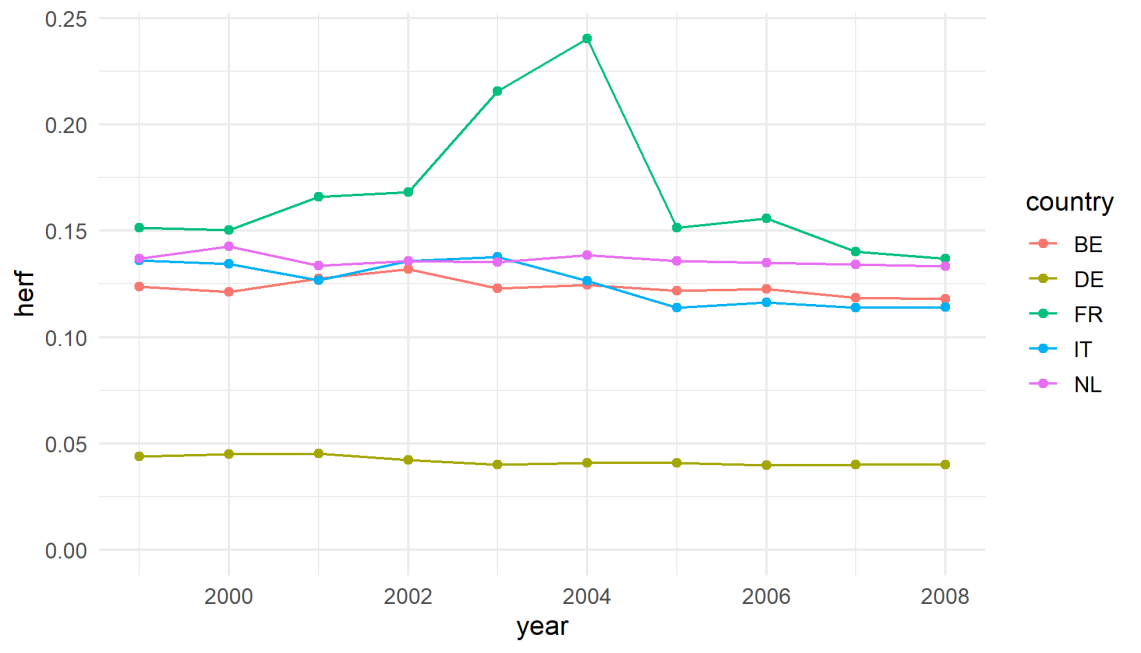
★: Can be found on my webpage -> teachhing - > R intro

Data cleaning

```
library(data.table)
base_raw = fread("_DATA/_RAW/High_tech_Employment_Eurostat.txt")

# Selection & renaming
base_emp = base_raw[nchar(geo) == 4,
                    .(nuts2 = geo,
                      year,
                      emp = Employment_highTech)]

# we zeroe the NAs (ad hoc choice, maybe wrong)
base_emp[is.na(emp), emp := 0]

# we create the country
base_emp[, country := substr(nuts2, 1, 2)]

# total emp per country-year
base_emp[, emp_total_cy := sum(emp), by = .(country, year)]

# share
base_emp[, share := emp / emp_total_cy]

# the final Herfindahl DB
base_herf = base_emp[, .(herf = sum(share ** 2)),
                     by = .(country, year, emp_total = emp_total_cy
```

Graph

```
library(ggplot2)

ctry = c("FR", "BE", "DE", "IT", "NL")

ggplot(base_herf[country %in% ctry],
       aes(x = year, y = herf, col = country)) +
  geom_line() +
  geom_point() +
  theme_minimal()
```



Data management with R

Data management

All you need is....

data.table

Outline: `data.table`

- import
- subset
- create simple variables
- create complex variables
- combine data sets

Importation: `data.table`

You can import rectangular data sets with `fread`:

```
library(data.table)
base_ht_emp = fread("data/High_tech_Employment_Eurostat.txt")
head(base_ht_emp)
#>      geo year Employment_highTech
#> 1:    AT 2008                161.0
#> 2:   AT1 2008                 80.6
#> 3:  AT11 2008                  3.5
#> 4:  AT12 2008                 34.1
#> 5:  AT13 2008                 43.0
#> 6:   AT2 2008                 32.8
```

The function `fread` is nice because:

- it guesses column types (based on a large random sample of rows)
- it guesses the delimiter
- it is fast

Data management with `data.table`

Data management with `data.table`

Pros:

- memory efficient
- very fast
- compact syntax

Cons:

- annoying startup message
- non-explicit intricate syntax
- non-standard R syntax

Data management with `data.table`

Pros:

- memory efficient
- very fast
- **compact syntax**

Cons:

- annoying startup message
- **non-explicit intricate syntax**
- **non-standard R syntax**

You can't have one without the other!

Creating a data.table

```
library(data.table)
dt = data.table(id = c("Mum", "Dad"), value = 1:2)
dt
#>     id value
#> 1: Mum     1
#> 2: Dad     2
```

From existing data.frames:

```
df = iris[, 1:3] # new DF
dt = df
setDT(dt) # change the type to DT without recreating it in memory
dt = as.data.table(iris) # creates a copy
```

data.table syntax

DT[i, j, by]

i: row index j: column index/values by: aggregation

You can do most of what you need in data management with just that!

Difference with DF: Row selection

In a `data.table`, you don't need a comma after the index to select rows (differently from `data.frame`s).

```
set.seed(1)
df = iris[sample(150, 5), c(1,5)]
dt = as.data.table(df)

# dt[i] is +/- equivalent to df[i, ]
dt[1]
#>    Sepal.Length    Species
#> 1:          5.8 versicolor

# look at the difference
head(df[1])
#>     Sepal.Length
#> 68           5.8
#> 129          6.4
#> 43           4.4
#> 14           4.3
#> 51           7.0
```

Difference with DF: Column selection

Inside the brackets of a `data.table`, you need not use character vectors to reference column names:★

```
# works => NON STANDARD R
dt[, Species]
#> [1] versicolor virginica  setosa     setosa     versicolor
#> Levels: setosa versicolor virginica

# does not work => error message
df[, Species]
#> Error in `[.data.frame`(df, , Species): object 'Species' not fou
```

★: To understand what's going on, see here.

Difference with DF: Ordering

Ordering the DT: use the names of the variables directly:

- `order(var_name_1, var_name_2, etc)`: use directly variable names
- `order(var_name_1, -var_name_2)`: put a minus in front of a variable to have a decreasing order

Example

```
dt = as.data.table(iris)
head(dt[order(Sepal.Length)], 3)
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1:          4.3         3.0          1.1         0.1  setosa
#> 2:          4.4         2.9          1.4         0.2  setosa
#> 3:          4.4         3.0          1.3         0.2  setosa

# decreasing order
head(dt[order(Species, -Sepal.Length)], 3)
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1:          5.8         4.0          1.2         0.2  setosa
#> 2:          5.7         4.4          1.5         0.4  setosa
#> 3:          5.7         3.8          1.7         0.3  setosa
```

Difference with DF: Subsetting

Subsetting: use the variables names directly!

```
dt = as.data.table(iris)
head(dt[Species == "setosa" & Sepal.Length == 5.1])
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1:          5.1         3.5          1.4         0.2  setosa
#> 2:          5.1         3.5          1.4         0.3  setosa
#> 3:          5.1         3.8          1.5         0.3  setosa
#> 4:          5.1         3.7          1.5         0.4  setosa
#> 5:          5.1         3.3          1.7         0.5  setosa
#> 6:          5.1         3.4          1.5         0.2  setosa
```

It is equivalent to:

```
df = iris
head(df[df$Species == "setosa" & df$Sepal.Length == 5.1, ])
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1           5.1         3.5          1.4         0.2  setosa
#> 18          5.1         3.5          1.4         0.3  setosa
#> 20          5.1         3.8          1.5         0.3  setosa
#> 22          5.1         3.7          1.5         0.4  setosa
#> 24          5.1         3.3          1.7         0.5  setosa
#> 40          5.1         3.4          1.5         0.2  setosa
```

Creating new variables

Creating new variables: Canonical way

To create new variable**s**, the syntax is:

```
dt[, c("vector", "of", "names") := list(stuff1, stuff2, stuff3)]
```

```
dt = as.data.table(iris)
dt[, c("id", "x") := list(1:nrow(dt), Sepal.Length**2)]
head(dt, 3)
#>    Sepal.Length Sepal.Width Petal.Length Petal.Width Species id
#> 1:          5.1         3.5          1.4         0.2  setosa  1
#> 2:          4.9         3.0          1.4         0.2  setosa  2
#> 3:          4.7         3.2          1.3         0.2  setosa  3
```

Keep in mind that this syntax is **data.table specific**.

Creating new variables: Too long syntax?

The canonical syntax to create new variables may seem too long:

```
dt[, c("id", "x") := list(1:nrow(dt), Sepal.Length**2)]
```

Good news

data.table allows for shorthands!

Creating new variables: Shorthands

Starters:

- you can replace `list(stuff)` by `.(stuff)`: yes, a point is a function
- there is an internal variable, which can be summoned with `.N`, reporting the number of observations★

Canonical call:

```
dt[, c("id", "x") := list(1:nrow(dt),
                          Sepal.Length*
```

Use `.` for `list`:

```
dt[, c("id", "x") := .(1:nrow(dt),
                       Sepal.Length**2)
```

Use `.N`:

```
dt[, c("id", "x") := .(1:.N,
                       Sepal.Length**2)
```

★: The *number of observations* is context specific, you'll see that later.

Creating new variables: Shorthands

When there is only one variable to be created:

- you can avoid the quotes in the left side
- you can avoid the list in the right side

Canonical call:

```
dt[, "id" := list(1:nrow(dt))]
```

Remove quotes:

```
dt[, id := list(1:nrow(dt))]
```

Use `.N`:

```
dt[, id := list(1:.N)]
```

Remove `list`:

```
dt[, id := 1:.N]
```

Creating new variables: Clumsy?

When you have 3+ variables to create, this syntax may be error prone since you need to match the order of the left side to the order of the right side.

```
dt[, c("id", "x") := .(1:.N, Sepal.Length**2)]
```

Good news

data.table allows to create variables differently!

Creating new variables: ":=" is an operator!

Question

Is the following code legit?

```
"+"(5, 3)
```

Answer

Sure it is!

```
"+"(5, 3)
#> [1] 8
```

Weird, right?

Operators

Operators

The values `+`, `*`, `^`, `:`, and `:=` (and many others) are operators.

They are special symbols which require values on the left ***and*** on the right to work.

Each of them is in fact associated to a regular function which is defined in a regular way.★

Something more or less of the form:

```
operator = function(a, b){ etc }
```

So operators are in fact regular functions that you can summon like any other function!

★: To be clear, very often they are not regular functions but methods. But I skip this unnecessary detail.

Operators: Example

Let's create an operator that keeps the first n letters of a word.

```
"%k%" = function(x, n){
  substr(x, 1, n)
}

"bonjour" %k% 3
#> [1] "bon"
```

And since they're no different from regular functions, you can summon them just like any function (they're just quoted to avoid a parsing error):

```
"%k%"("bonjour", 3)
#> [1] "bon"
```

Creating variables: Functional form

You can use the function `":="` to create variables in the following way:

Canonical call:

```
dt[, c("id", "x") :=
     list(1:nrow(dt), Sepal.Length**2)]
```

Functional call:

```
dt[, ":="(id = 1:.N,
          x = Sepal.Length**2)]
```

Creating variables: Partial modification

Q: Say you want to modify a variable, but only for some observations. How do you do?

A: Yes, you've got it! Just use the argument i.

Let's trim large values of `Sepal.Length`:

```
dt = as.data.table(iris)
summary(dt$Sepal.Length)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   4.300   5.100   5.800   5.843   6.400   7.900

cutoff = mean(dt$Sepal.Length) + 1.5 * sd(dt$Sepal.Length)

dt[Sepal.Length >= cutoff, Sepal.Length := cutoff]
summary(dt$Sepal.Length)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   4.300   5.100   5.800   5.812   6.400   7.085
```

Creating variables: The function `set`

The function `set` is specific to data.table and allows to create variables more programmatically.

The syntax is:

```
set(dt, i = NULL, j, value)
```

Create a new variable:

```
dt = data.table(id = letters[1:3],
                x = 1:3)

set(dt, j = "x2", value = dt$x ** 2)
dt
#>    id x x2
#> 1:  a 1  1
#> 2:  b 2  4
#> 3:  c 3  9
```

Note that you cannot use variables by reference any more, you need explicit values (here `dt$x`).

Modify an existing variable:

```
dt = data.table(id = letters[1:3],
                x = 1:3)

set(dt, c(1L, 3L), j = "id",
    value = c("aa", "cc"))
dt
#>    id x
#> 1: aa 1
#> 2:  b 2
#> 3: cc 3
```

Note that `1L`, `3L` means `1` and `3` in integer format (otherwise `data.table` complains).

Selecting variables

Extraction: Basic way

The most basic way is to insert a character vector of variables' names:

```
dt = as.data.table(iris)
head(dt[, c("Species", "Sepal.Width")], 2)
#>    Species Sepal.Width
#> 1:  setosa         3.5
#> 2:  setosa         3.0
```

Very weird trick alert

You can *negate character strings!*

```
head(dt[, -c("Species", "Sepal.Width")], 2)
#>    Sepal.Length Petal.Length Petal.Width
#> 1:          5.1          1.4         0.2
#> 2:          4.9          1.4         0.2
```

Extraction: Usual way

You can extract and modify the columns of a data.table by using:

`dt[, list(newvar1 = fun(oldvars), etc)]`

```
dt = as.data.table(iris)
new_dt = dt[, list(x = Sepal.Length, id = 1:nrow(dt))]
new_dt[1]
#>      x id
#> 1: 5.1  1
```

As usual, shorthands apply:

```
new_dt = dt[, .(x = Sepal.Length, id = 1:.N)]
new_dt[1]
#>      x id
#> 1: 5.1  1
```

Extraction: Simplification

When you select just one variable, you can omit the list.

This leads to return a vector.

```
head(dt[, Sepal.Length])
#> [1] 5.1 4.9 4.7 4.6 5.0 5.4
```

If you want to return a `data.table`, use a list:

```
head(dt[, .(Sepal.Length)])
#>    Sepal.Length
#> 1:          5.1
#> 2:          4.9
#> 3:          4.7
#> 4:          4.6
#> 5:          5.0
#> 6:          5.4
```

Using variables' names in vectors

Question

Does this work?

```
var = c("Species", "Sepal.Width")
dt[, var]
```

Answer

It does not.

```
var = c("Species", "Sepal.Width")
dt[, var]
#> Error in `[.data.table`(dt, , var) : ]
#> j (the 2nd argument inside ) is a single symbol but column na
```

Q: How to make it work?

Base R's `with` mechanism

In R, you use the function `with` to attach the variables of a list (usually a data set) to the "search path", so that it works *as if* the variables of that data set were in the current environment.

Does not work because the variables `ex1` and `ex2` were never created: so we (rightfully) get an error.

```
my_list = list(ex1 = 3, ex2 = 5)
ex1 + ex2
#> Error in eval(expr, envir, enclos):
```

Now works, because we ask the variables `ex1` and `ex2` to be searched within the list `my_list`.

```
my_list = list(ex1 = 3, ex2 = 5)
with(my_list, ex1 + ex2)
#> [1] 8
```

Base R's `with` mechanism: data.table

Link to data.table

What `data.table` does is identical to this `with` mechanism: the variables' names are directly fetched in the data.

Guess what: the operator `[.data.table` has a `with` argument.

Using the argument `with` in data.table

Let's use the argument `with` and see what it does:

```
dt = as.data.table(iris)
var = c("Species", "Sepal.Width")
head(dt[, var, with = FALSE])
#>    Species Sepal.Width
#> 1:  setosa         3.5
#> 2:  setosa         3.0
#> 3:  setosa         3.2
#> 4:  setosa         3.1
#> 5:  setosa         3.6
#> 6:  setosa         3.9
```

Yeah it works!

The `with` argument

But as usual, there are shorthands. Just add two points before the name of the variable to indicate to data.table that the variable is **not** to be fetched in the data:

```
var = c("Species", "Sepal.Width")
head(dt[, ..var])
#>    Species Sepal.Width
#> 1:  setosa         3.5
#> 2:  setosa         3.0
#> 3:  setosa         3.2
#> 4:  setosa         3.1
#> 5:  setosa         3.6
#> 6:  setosa         3.9
```

To note

This is a super weird syntax trick, **absolutely specific to data.table!**

Explaining why it works is out of the scope of this course.[1]

[1]: It is thanks to R's meta programming capabilities.

Using `with` when creating variables

Remember the syntax to create variables? We can also use `with = FALSE`.

Does not work because `var` is not defined in `dt`:

```
dt = data.table(x = 1:3)
var = c("x2", "x3")
dt[, var := .(x**2, x**3)]
#> Error in `[.data.table`(dt, , `:=`(v
```

Now works with `with = FALSE`, although deprecated:

```
dt = data.table(x = 1:3)
var = c("x2", "x3")
dt[, var := .(x**2, x**3), with = FALSE
#> Warning in `[.data.table`(dt, , `:=`
#> with=FALSE together with := was depr
#> wrap the LHS of := with parentheses;
#> to column name(s) held in variable m
#> warned in 2014, this is now a warnir
dt
#>    x x2 x3
#> 1: 1  1  1
#> 2: 2  4  8
#> 3: 3  9 27
```

Using `with` when creating variables

Remember the syntax to create variables? We can also use `with = FALSE`.

Does not work because `var` is not defined in `dt`:

```
dt = data.table(x = 1:3)
var = c("x2", "x3")
dt[, var := .(x**2, x**3)]
#> Error in `[.data.table`(dt, , `:=`(v
```

And with the shorthand:

```
dt = data.table(x = 1:3)
var = c("x2", "x3")
dt[, (var) := .(x**2, x**3)]
dt
#>    x x2 x3
#> 1: 1  1  1
#> 2: 2  4  8
#> 3: 3  9 27
```

Aggregation

Aggregation

What?

Applying operations by groups of observations is one of the most common data tasks:

- average/min/max per country
- *whatever you want* per *whatever grouping*

How?

This can be done exquisitely easily★ in data.table:

Just use the argument **by**!

★: Not sure the English is correct, but that's the idea!

Aggregating data by groups

To aggregate data according to an identifier:

`dt[, list(newvar1 = fun(oldvars), etc), by = list(identifiers)]`

```
dt = as.data.table(iris)
agg_dt = dt[, list(mean_sl = mean(Sepal.Length),
                   n_obs = length(Sepal.Length)),
            by = list(Species)]
agg_dt
#>       Species mean_sl n_obs
#> 1:     setosa   5.006    50
#> 2: versicolor   5.936    50
#> 3:  virginica   6.588    50
```

Aggregating data: Shorthands

Shorthands (as usual):

- `list()` $\equiv$ `.()`
- if only one variable in `by=`, you can omit `list()`
- `.N`: the number of observations in the group

Canonical call:

```
dt[, list(mean_sl = mean(Sepal.Length),
          n_obs = length(Sepal.Length))
   by = list(Species)]
```

Using `.` for `list`:

```
dt[, .(mean_sl = mean(Sepal.Length),
       n_obs = length(Sepal.Length)),
   by = .(Species)]
```

Removing the list in `by`:

```
dt[, .(mean_sl = mean(Sepal.Length),
       n_obs = length(Sepal.Length)),
   by = Species]
```

Using `.N`:

```
dt[, .(mean_sl = mean(Sepal.Length),
       n_obs = .N),
   by = Species]
```

Aggregate: Exercize

The following data contains trade values in euros from exporting (Origin) countries to importing (Destination) countries.

```
# install.packages("fixest")
data(trade, package = "fixest")
head(trade)
#>   Destination Origin Product Year  dist_km    Euros
#> 1          LU     BE       1 2007 139.5719  2966697
#> 2          BE     LU       1 2007 139.5719  6755030
#> 3          LU     BE       2 2007 139.5719 57078782
#> 4          BE     LU       2 2007 139.5719  7117406
#> 5          LU     BE       3 2007 139.5719 17379821
#> 6          BE     LU       3 2007 139.5719  2622254
```

Exercize

Create the table containing the yearly total exportations for each exporting country.

Creating aggregate variables

Now say you want to create a new variable:

- the max `Petal.Length` for each variety of flower
- but you want to keep the same number of rows! In other words, you want the original data set with just an extra variable.

Regular way to proceed:

1. you aggregate the data at the species level
2. you merge back the information to the original database

```
dt = as.data.table(iris)[, 3:5]
agg_dt = dt[, .(max_sl = max(Petal.Length)), by = Species]
res = merge(dt, agg_dt)
# looking at some obs
res[c(1:2, 51:52, 101:102)]
#>       Species Petal.Length Petal.Width max_sl
#> 1:     setosa          1.4         0.2    1.9
#> 2:     setosa          1.4         0.2    1.9
#> 3: versicolor          4.7         1.4    5.1
#> 4: versicolor          4.5         1.5    5.1
#> 5:  virginica          6.0         2.5    6.9
#> 6:  virginica          5.1         1.9    6.9
```

Creating aggregate variables

But you can do it directly in one line with data.table!

Since it concerns the creation of a new variable, you must use `:=`:

`dt[, [character vector of names] := [list of values], by = list(identifiers)]`

We obtain the previous result with a single line:

```
dt = as.data.table(iris)[, 3:5]
dt[, max_sl := max(Petal.Length), by = Species]
# looking at some obs
dt[c(1:2, 51:52, 101:102)]
#>    Petal.Length Petal.Width    Species max_sl
#> 1:          1.4         0.2     setosa    1.9
#> 2:          1.4         0.2     setosa    1.9
#> 3:          4.7         1.4 versicolor    5.1
#> 4:          4.5         1.5 versicolor    5.1
#> 5:          6.0         2.5  virginica    6.9
#> 6:          5.1         1.9  virginica    6.9
```

Exercize: Aggregate variables

Remember the trade data:

```
# install.packages("fixest")
data(trade, package = "fixest")
head(trade, 3)
#>   Destination Origin Product Year  dist_km    Euros
#> 1          LU     BE       1 2007 139.5719  2966697
#> 2          BE     LU       1 2007 139.5719  6755030
#> 3          LU     BE       2 2007 139.5719 57078782
```

- create the data set `base_export` containing the yearly total exports between each country-pair
- add the variable `share_export`: it represents the Destination country represents in the yearly total exports of an Origin country.

> For example, take France (FR) exports to Germany (DE) in 2010. If `share_export = 50` this means that Germany receives 50% of all France exportations in 2010.

data.table: Summing up

- selecting observations
- selecting an modifying variables
- creating new variables
- creating aggregated measures

It's about 80% of your data management journey. The other 18% is merging.

Combining data sets

Merging different sets of information

Merging is the bread and butter of data management, most important and recurring operation.

First let's define data:

```
dtx = data.table(id = c("Al", "Jil", "Pablo", "Jules"))
dtx$performance = 1:4
dty = data.table(id = c("Al", "Francis", "Myriam"))
dty$age = c(34, 52, 29)
dtx
#>       id performance
#> 1:    Al           1
#> 2:   Jil           2
#> 3: Pablo           3
#> 4: Jules           4
dty
#>         id age
#> 1:      Al  34
#> 2: Francis  52
#> 3:  Myriam  29
```

Merging

Merging is an operation that combines information from different sources for a set of identifiers.

In our example we have a `performance` variable in table `dtx` and an `age` variable for table `dty.`

In both tables, the information is *unique* for a given id.

As we can see, the information is not exhaustive: `Al` is the only `id` to be in both tables.

Merging

You have 4 types of merging operations:

1. inner join: only the identifiers that are present in both tables are kept.
2. left join: all identifiers from the first table are kept. Information on identifiers of the second table that are not present in the first table are dropped.
3. right join: explicit, same logic as left join
4. outer join: All the information is kept. No identifier is lost.

Merging

This translates to the function `merge()` with the explicit:

1. inner join: default
2. left join: `all.x = TRUE`
3. right join: `all.y = TRUE`
4. outer join: `all = TRUE` (equiv. `all.x = TRUE` & `all.y = TRUE`)

Merging: Examples

```
merge(dtx, dty) # inner join
#>    id performance age
#> 1: Al           1  34
merge(dtx, dty, all.x = TRUE) # left join
#>       id performance age
#> 1:    Al           1  34
#> 2:   Jil           2  NA
#> 3: Jules           4  NA
#> 4: Pablo           3  NA
merge(dtx, dty, all.y = TRUE) # right join
#>         id performance age
#> 1:      Al           1  34
#> 2: Francis          NA  52
#> 3:  Myriam          NA  29
```

Merging: Examples

```
merge(dtx, dty, all = TRUE) # outer join
#>         id performance age
#> 1:      Al           1  34
#> 2: Francis          NA  52
#> 3:     Jil           2  NA
#> 4:   Jules           4  NA
#> 5:  Myriam          NA  29
#> 6:   Pablo           3  NA
```

Merging: Question

```
dtx = data.table(id = c("Al", "Al",
                        "Pablo", "Jule
                 performance = 1:4)
dtx
#>        id performance
#> 1:     Al           1
#> 2:     Al           2
#> 3: Pablo           3
#> 4: Jules           4
```

```
dty = data.table(id = c("Al", "Al",
                        "Myriam"),
                 age = c(34, 52, 29))
dty
#>        id age
#> 1:     Al  34
#> 2:     Al  52
#> 3: Myriam  29
```

Question

What happens if?

```
merge(dtx, dty, by = "id")
```

Answer

```
merge(dtx, dty, by = "id")
#>    id performance age
#> 1: Al           1  34
#> 2: Al           1  52
#> 3: Al           2  34
#> 4: Al           2  52
```

Merging: Cartesian product

When merging data tables, there is a check for Cartesian product to avoid performing a very costly operation by mistake.

You can bypass this with the argument `allow.cartesian`.

```
dta = as.data.table(iris)[, .(Petal.Length, Species)]
dtb = as.data.table(iris)[, .(Sepal.Width, Species)]
dtab = merge(dta, dtb, by = "Species") # cartesian product => BLOCKED!
#> Error in vecseq(f__, len__, if (allow.cartesian || notjoin || !anyDuplic
```

```
# now works => look at the size of the resulting data set!
# iris is 150 obs, 3 species of 50 obs.
# after merge we get: 3 * 50**2 obs.
dtab = merge(dta, dtb, by = "Species", allow.cartesian = TRUE)
dim(dtab)
#> [1] 7500    3
```

Merging: Keys of different names

You can use argument `by` to merge with identifiers of different names.

```
dtz = dty
names(dtz)[1] = "ID" # changing the name
merge(dtx, dtz, by.x = "id", by.y = "ID")
#>    id performance age
#> 1: Al           1  34
#> 2: Al           1  52
#> 3: Al           2  34
#> 4: Al           2  52
```

However I **strongly** discourage that.

Better keeping consistent variable names across tables right from the start of a project. But it can be useful.

Of course an identifier can consist of more than one variable (e.g. in a panel it can be individual identifier + year identifier).

Exercise: Citation counts

Let's consider patent data from the USPTO. We want to count the number of times each patent gets cited by other patents.

Based on the text files:

- uspto_sample.tsv
- uspto_cites.csv

Create a table looking as follows:

patent_id	year	nb_cites
153515	1990	12
D12345	1992	0

with `nb_cites` the number of times patent identified by `patent_id` in `uspto_sample.tsv` has been cited by the patents in `uspto_cites.csv`.

Going further

The material we've just seen should take you a very long way!

However this was just an introduction: you can do many more things with data.table!

- data.table's website
- batch manipulation of variables with .SD