Statistical Computing with R

Lecture 13: pipe operators; horizontal and vertical merges; long and wide format; the spaghetti plot

Mirko Signorelli
: mirkosignorelli.github.io
: statcompr [at] gmail.com

Mathematical Institute
Leiden University

Master in Statistics and Data Science (2023-2024)

Announcements

Plan for next week (lecture 14):

1. shorter - we will probably be done around 4 pm
2. more info about the final exam
3. time for questions about the contents of the 2nd block
4. practice exam so you can get a better idea of how the exam will be structured

Recap

Lecture 12:

- the `dplyr` package
- data visualization with `ggplot2`

Today:

- pipe operators
- vertical and horizontal merges
- wide and long format
- from wide to long format, and viceversa
- the spaghetti plot

Pipe operators

Vertical and horizontal merges

Long and wide format

The spaghetti plot

Pipe operators: a short (troubled) history

To date, several different pipe operators have been implemented in `R`:

- `%.%` (in the `dplyr` package) was the very first implementation, but it's not available any more
- `%>%` (in the `magrittr` package) was the second implementation (still available)
- `|>` (part of base `R` since version 4.1.0) is the latest implementation (introduced in March 2021)

What does the pipe operator do?

- The pipe operator passes an object as first argument of a function
- Two simple examples:

```
mean(1:5) # standard R
```

```
## [1] 3
```

```
1:5 |> mean() # native pipe
```

```
## [1] 3
```

```
library(magrittr) # needed for %>%
1:5 %>% mean() # magrittr pipe
```

```
## [1] 3
```

```
v = c(17, 2, NA, 5) # example 2
mean(v, na.rm = T) # standard R
```

```
## [1] 8
```

```
v |> mean(na.rm = T) # native pipe
```

```
## [1] 8
```

To pipe, or not to pipe?

Q: should I use %>%, |>, or don't even bother?

- You may use a pipe operator if you wish (but you don't have to if you don't like it) ☺
- If you decide to use a pipe operator, my suggestions are:
  1. use |> ("native pipe"; part of base `R` = no dependencies ☺) unless you need to be back-compatible with `R` versions $< 4.1.0$
  2. use %>% (introducing a dependency on `magrittr`! ☹) only if you need to be back-compatible with `R` $< 4.1.0$

⚠ Not all functions work properly with pipe operators! ⚠

- Sometimes, you may need to do additional work to be able to use pipes (see the example with `aggregate` in the next slides)

More complex piping

- Pipes are mostly used to sequentially execute multiple operations in one go
- Consider this example:

```
library(brolgar)
df.1990 = subset(heights, year == 1990)
baseRsol = aggregate(height_cm ~ continent,
                     df.1990, FUN = mean)
```

How can we rewrite this code using |>?

Rewriting using |>: option 1 (base R)

- Original statement:

```
aggregate(height_cm ~ continent, df.1990, FUN = mean)
```

- How can we pass `height_cm ~ continent` to `aggregate( )` using the pipe?
  1. Ask yourself the question: what kind of object am I passing? → check out `?aggregate`
  2. Once you figure out that you need to pass a formula object:

```
as.formula('height_cm ~ continent') |>
  aggregate(df.1990, FUN = mean)
```

```
##   continent height_cm
## 1    Africa  170.0250
## 2  Americas  170.3875
## 3      Asia  167.4502
## 4    Europe  172.7000
```

Rewriting using |>: option 2 (dplyr)

- A dplyr version of the previous aggregate is:

```
library(dplyr)
dplyrSol = heights |>
  filter(year == 1990) |>
  group_by(continent) |>
  summarize(mean = mean(height_cm))
```

- The pipe works quite well with dplyr because most dplyr functions have a data frame / tibble as first argument and output a data frame / t(s)ibble ⇒ you can write semi-long piping sequences rather easily

Did we get the same result?

- Base R solution:

```
baseRsol
```

```
##   continent height_cm
## 1    Africa  170.0250
## 2  Americas  170.3875
## 3      Asia  167.4502
## 4    Europe  172.7000
```

- dplyr solution:

```
dplyrSol
```

```
## # A tsibble: 4 x 3 [!]
## # Key:       continent [4]
##   continent  year  mean
##   <chr>     <dbl> <dbl>
## 1 Africa     1990  170.
## 2 Americas   1990  170.
## 3 Asia       1990  167.
## 4 Europe     1990  173.
```

Concluding remarks

- Up to you whether you use pipe operators or not ☺
- Unsolicited opinions (but: you do you!):
  1. personally, I find |> most useful when I am quickly trying to edit code in the console, and when I want to avoid nesting several functions into each other to make code more readable
  2. however, in many situations I don't use |> - mostly because I prefer to break down complex code involving multiple steps into separate intermediate steps (this makes it easier to double-check the intermediate outputs and debug code)
- Important to understand what |> does even if you decide to not use it at all!

Pipe operators

Vertical and horizontal merges

Long and wide format

The spaghetti plot

Example problem

Consider the following fictional situation:

1. 5 students followed a course
2. the results of their assignments are stored in this data frame:

```
assignments
```

```
##   stud_id  student assign_1 assign_2
## 1   10235    Maria      8.0      7.0
## 2   12521    David      5.5      8.0
## 3    8953    Femke      6.5      6.0
## 4    9159     Luke      8.0      9.0
## 5   13256 Fernanda      7.0      7.5
```

Example problem (cont'd)

3. 3 students passed the exam at the first opportunity:

```
exam
```

```
##   stud_id student exam
## 1   10235   Maria    8
## 2   12521   David    7
## 3    8953   Femke    6
```

3. 2 students passed the resit exam:

```
resit
```

```
##   stud_id  student exam
## 1    9159     Luke  7.0
## 2   13256 Fernanda  7.5
```

Example problem (cont'd)

- Problem: grades have been gathered separately in 3 different tables.
- How can we gather all these data frames and compute the final grade?
- To do this, we need to be able to perform:
  1. a vertical merge to join `exam` and `resit`
  2. a horizontal merge to combine information on the `assignments` and the `exam`

Vertical merge

- Vertical merge:
  1. we have two (or more) data frames containing different observations (= rows) of the same variables (= columns)
  2. we want to gather those observations into a single data frame
- Vertical merges can be done using `rbind(df1, df2)`:

```
all_exams = rbind(exam, resit)
all_exams
```

```
##   stud_id  student exam
## 1   10235    Maria  8.0
## 2   12521    David  7.0
## 3    8953    Femke  6.0
## 4    9159     Luke  7.0
## 5   13256 Fernanda  7.5
```

Vertical merge: a few remarks

A few facts about `rbind( )`:

1. the two dataframes should contain the same number of variables
2. variables should have the same names and the same types in the two data frames
3. variables could be ordered differently in the two data frames
4. you can merge > 2 data frames at the same time $\rightarrow$ `rbind(df1, df2, df3)`

$\Rightarrow$ you might need to modify the data frames before you can merge them horizontally!

An alternative to `rbind( )`: `bind_rows()`

- What if the two dataframes contain different variables?
- Handy alternative: `dplyr::bind_rows()`

```
df1 = data.frame(a = letters[1:3], b = c(8, 3, 12))
df2 = data.frame(a = letters[4:5], c = letters[8:9])
library(dplyr) |> suppressMessages()
bind_rows(df1, df2)
```

```
##   a  b    c
## 1 a  8 <NA>
## 2 b  3 <NA>
## 3 c 12 <NA>
## 4 d NA    h
## 5 e NA    i
```

Horizontal merge

- Horizontal merge:
  1. we have two (or more) data frames containing different variables for the same individuals
  2. we want to collect all variables into a single data frame
- Horizontal merges are less straightforward than vertical merges:
  1. you need a way to match observations in different data frames
  2. this is usually achieved through a unique identifier called key variable
  3. a good key variable is a variable that is unique to a single individual / observation (typical examples: Dutch BSN, LU student number, customer number used by a company. . .)
  4. sometimes, a single key is not present, but needs to be created using a combination of two or more variables (e.g., name + surname + birth date)

Back to our example

In our example, we can use the student number, called `stud_id`, as key:

```
assignments
```

```
##   stud_id  student assign_1 assign_2
## 1   10235    Maria      8.0      7.0
## 2   12521    David      5.5      8.0
## 3    8953    Femke      6.5      6.0
## 4    9159     Luke      8.0      9.0
## 5   13256 Fernanda      7.0      7.5
```

```
all_exams
```

```
##   stud_id  student exam
## 1   10235    Maria  8.0
## 2   12521    David  7.0
## 3    8953    Femke  6.0
## 4    9159     Luke  7.0
## 5   13256 Fernanda  7.5
```

The merge( ) function

- You can use merge( ) to perform a horizontal merge:

```
grades = merge(assignments, all_exams, by = 'stud_id')
grades
```

```
##   stud_id student.x assign_1 assign_2 student.y exam
## 1    8953     Femke      6.5      6.0     Femke  6.0
## 2    9159      Luke      8.0      9.0      Luke  7.0
## 3   10235     Maria      8.0      7.0     Maria  8.0
## 4   12521     David      5.5      8.0     David  7.0
## 5   13256  Fernanda      7.0      7.5  Fernanda  7.5
```

- The variable student is now repeated; two solutions in the next slide!

Removing duplicated variables

- Before taking any further action, check that the two instances of the duplicated variable are indeed the same!

```
identical(grades$student.x, grades$student.y)
```

```
## [1] TRUE
```

- In this case, `student.x` and `student.y` are exactly the same ⇒ we can remove one of the two
- Easiest solution: add `student` to the keys:

```
grades = merge(assignments, all_exams,
               by = c('stud_id', 'student'))
grades
```

```
##   stud_id  student assign_1 assign_2 exam
## 1   10235    Maria      8.0      7.0  8.0
## 2   12521    David      5.5      8.0  7.0
## 3   13256 Fernanda      7.0      7.5  7.5
## 4    8953    Femke      6.5      6.0  6.0
## 5    9159     Luke      8.0      9.0  7.0
```

Removing duplicated variables (cont'd)

- Alternative, more cumbersome solution:

```
grades = merge(assignments, all_exams, by = 'stud_id')
# remove the student.y variable:
grades$student.y = NULL
# rename the student.x variable:
names(grades)[names(grades) == 'student.x'] = 'student'
grades
```

```
##   stud_id  student assign_1 assign_2 exam
## 1    8953    Femke      6.5      6.0  6.0
## 2    9159     Luke      8.0      9.0  7.0
## 3   10235    Maria      8.0      7.0  8.0
## 4   12521    David      5.5      8.0  7.0
## 5   13256 Fernanda      7.0      7.5  7.5
```

Horizontal merge: final remarks

- Key variable could have different names in the two data frames; if so, you can use:

```
merge(x, y, by.x = '...', by.y = '...')
```

- Having exactly the same observations in both data frames is not a strict requirement:
  1. some individuals may appear in just one of the two data frames
  2. use the `all` argument to decide what to do with the non-matching rows (see next slide!)

merge(..., all = TRUE)

```
assignments_subset = assignments[c(2, 5), ]
assignments_subset
```

```
##   stud_id  student assign_1 assign_2
## 2   12521    David      5.5      8.0
## 5   13256 Fernanda      7.0      7.5
```

```
# default merge( ) behaviour:
merge(assignments_subset, all_exams, by = c('stud_id', 'student'))
```

```
##   stud_id  student assign_1 assign_2 exam
## 1   12521    David      5.5      8.0  7.0
## 2   13256 Fernanda      7.0      7.5  7.5
```

```
# ...but if you specify all = TRUE:
merge(assignments_subset, all_exams,
      by = c('stud_id', 'student'), all = T)
```

```
##   stud_id  student assign_1 assign_2 exam
## 1    8953    Femke       NA       NA  6.0
## 2    9159     Luke       NA       NA  7.0
## 3   10235    Maria       NA       NA  8.0
## 4   12521    David      5.5      8.0  7.0
## 5   13256 Fernanda      7.0      7.5  7.5
```

Pipe operators

Vertical and horizontal merges

Long and wide format

The spaghetti plot

The wide format

Data are usually stored using the wide format, where 1 individual = 1 row

```
grades
```

```
##   stud_id  student assign_1 assign_2 exam
## 1    8953    Femke      6.5      6.0  6.0
## 2    9159     Luke      8.0      9.0  7.0
## 3   10235    Maria      8.0      7.0  8.0
## 4   12521    David      5.5      8.0  7.0
## 5   13256 Fernanda      7.0      7.5  7.5
```

However, sometimes it can be handy to arrange your data in a different format...

The long format

The long format is typically used to represent longitudinal data

- Longitudinal data = data comprising repeated measurements over time of a set of variables for the same individual
- Long format:

1 row = all measurements from an individual at a given time point (e.g., 1 hospital visit)

1 individual = multiple rows (1 for each time point)

- Long format is particularly useful when dealing with unbalanced longitudinal data (different time points for different individuals)
- NB: longitudinal data analysis will be covered in the *Essentials of Mixed and Longitudinal Modelling* course. Here I will just introduce some basics about how to manage and visualize longitudinal data

Example

```
library(brolgar)
wages |> as.data.frame() |> head(10)
```

```
##    id ln_wages    xp ged xp_since_ged black hispanic
## 1  31    1.491 0.015   1        0.015     0        1
## 2  31    1.433 0.715   1        0.715     0        1
## 3  31    1.469 1.734   1        1.734     0        1
## 4  31    1.749 2.773   1        2.773     0        1
## 5  31    1.931 3.927   1        3.927     0        1
## 6  31    1.709 4.946   1        4.946     0        1
## 7  31    2.086 5.965   1        5.965     0        1
## 8  31    2.129 6.984   1        6.984     0        1
## 9  36    1.982 0.315   1        0.315     0        0
## 10 36    1.798 0.983   1        0.983     0        0
##    high_grade unemploy_rate
## 1           8          3.21
## 2           8          3.21
## 3           8          3.21
## 4           8          3.30
## 5           8          2.89
## 6           8          2.49
## 7           8          2.60
## 8           8          4.80
## 9           9          4.89
## 10          9          7.40
```

- Wage information for subject with `id = 31` recorded at the following `xp` (experience in years) values: `0.015, 0.715, 1.734, 2.773, 3.927, ...`

Converting wide and long format

- Sometimes you may need to convert wide to long format, and viceversa
- ⚠ Conversion from long to wide mostly makes sense with balanced longitudinal data, less with unbalanced designs ⚠
- There are many different functions to do this ⇒ here I will introduce on two functions from the `reshape2` package:
  1. `melt( )`
  2. `dcast( )`

. . . and mention some of the existing alternatives ☺

From wide to long format

- You can use `melt( )` from the `reshape2` package to convert from wide to long format:

```
head(grades, 2)
```

```
##   stud_id student assign_1 assign_2 exam
## 1    8953   Femke      6.5        6    6
## 2    9159    Luke      8.0        9    7
```

```
library(reshape2)
df_long = melt(grades, id.vars = c('stud_id', 'student'))
head(df_long, 7)
```

```
##   stud_id  student variable value
## 1    8953    Femke assign_1   6.5
## 2    9159     Luke assign_1   8.0
## 3   10235    Maria assign_1   8.0
## 4   12521    David assign_1   5.5
## 5   13256 Fernanda assign_1   7.0
## 6    8953    Femke assign_2   6.0
## 7    9159     Luke assign_2   9.0
```

From long to wide format

- You can use `dcast( )` from the `reshape2` package to convert from long to wide format:

```
head(df_long, 2)
```

```
##   stud_id student variable value
## 1    8953   Femke assign_1   6.5
## 2    9159    Luke assign_1   8.0
```

```
df_wide = dcast(df_long,
                stud_id + student ~ variable,
                value.var = 'value')
df_wide
```

```
##   stud_id  student assign_1 assign_2 exam
## 1    8953    Femke      6.5      6.0  6.0
## 2    9159     Luke      8.0      9.0  7.0
## 3   10235    Maria      8.0      7.0  8.0
## 4   12521    David      5.5      8.0  7.0
## 5   13256 Fernanda      7.0      7.5  7.5
```

Other packages

- In the previous examples I used `melt` and `dcast` from the `reshape2` packages
- Several packages contain similar functions:

Package	Wide to long	Long to wide
base R	`reshape`	`reshape`
`data.table`	`melt`	`dcast`
`reshape`	`melt`	`cast`
`reshape2`	`melt`	`dcast`
`tidyr`	`pivot_longer`	`pivot_wider`

- NB: `melt` is not exactly the same function across packages

reshape2::melt versus tidyr:pivot_longer

```
reshape2::melt(grades,
      id.vars = c('stud_id', 'student')) |> head(3)
```

```
##   stud_id student variable value
## 1    8953   Femke assign_1   6.5
## 2    9159    Luke assign_1   8.0
## 3   10235   Maria assign_1   8.0
```

```
tidyr::pivot_longer(grades,
      cols = c('assign_1', 'assign_2', 'exam')) |> head(3)
```

```
## # A tibble: 3 x 4
##   stud_id student name     value
##     <dbl> <chr>   <chr>    <dbl>
## 1    8953 Femke   assign_1   6.5
## 2    8953 Femke   assign_2   6
## 3    8953 Femke   exam       6
```

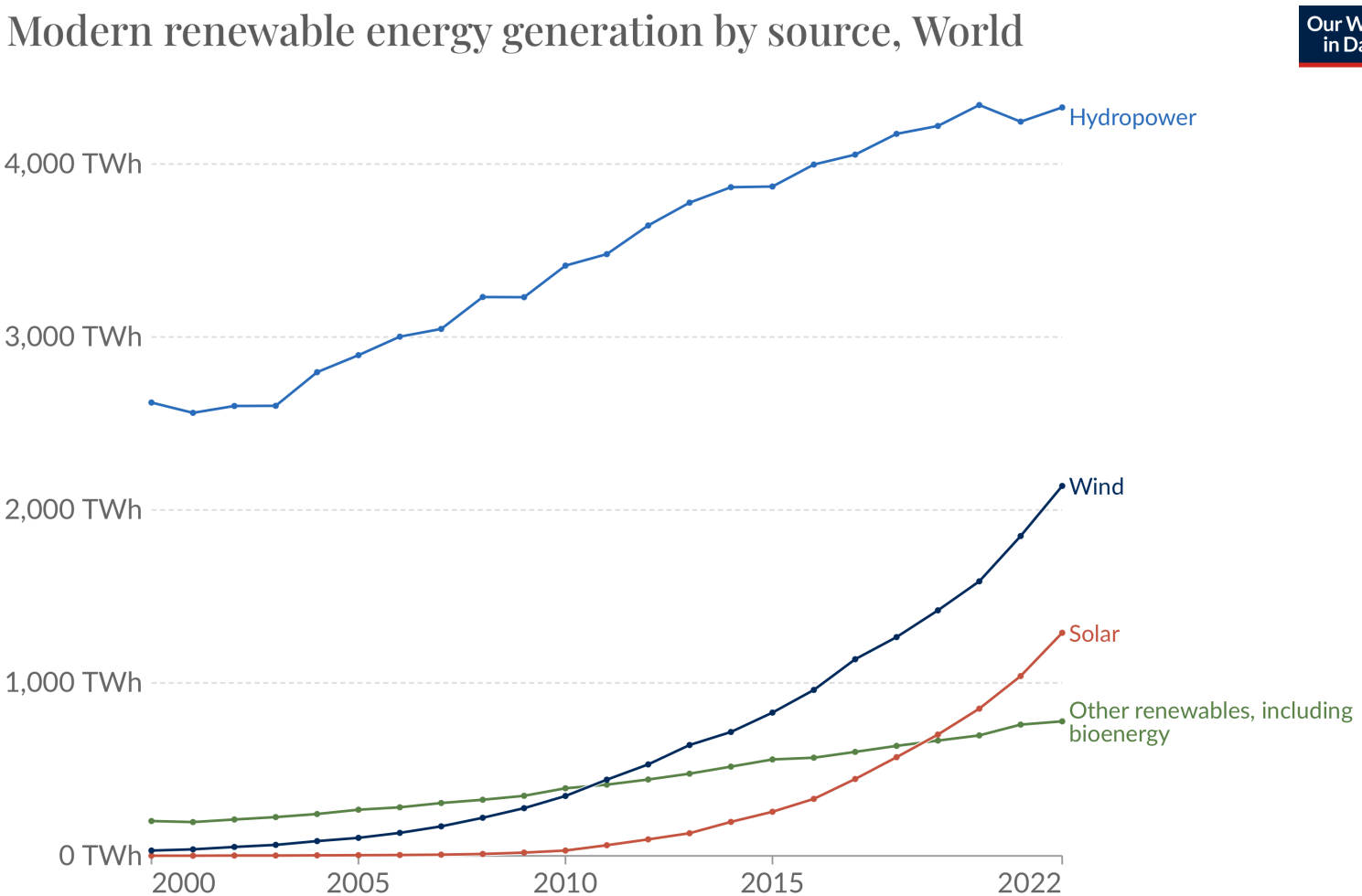
Pipe operators

Vertical and horizontal merges

Long and wide format

The spaghetti plot

Visualizing longitudinal data



Data source: Ember's Yearly Electricity Data; Ember's European Electricity Review; Energy Institute Statistical Review of World Energy
OurWorldInData.org/renewable-energy | CC BY

The spaghetti / trajectory plot

Longitudinal data are typically visualized using the so-called spaghetti plot or trajectory plot:

- x axis: a time variable (e.g., time, year, ...)
- y axis: the longitudinal variable of interest
- points denote actual measurements
- lines ("spaghetti") denote individual trajectories

In short:

1 line = trajectory of 1 subject / unit / group

make.spaghetti( )

- Several ways to draw spaghetti plots. An easy one: use the `make.spaghetti( )` function from the `ptmixed` package
- To install `ptmixed`, you first need to install the `Bioconductor` package `tweeDEseq`:

```
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
BiocManager::install("tweeDEseq")
install.packages('ptmixed')
```

- Syntax (see `?make.spaghetti` for more details):

```
library(ptmixed)
make.spaghetti(x, y, id, group, data, ...)
```

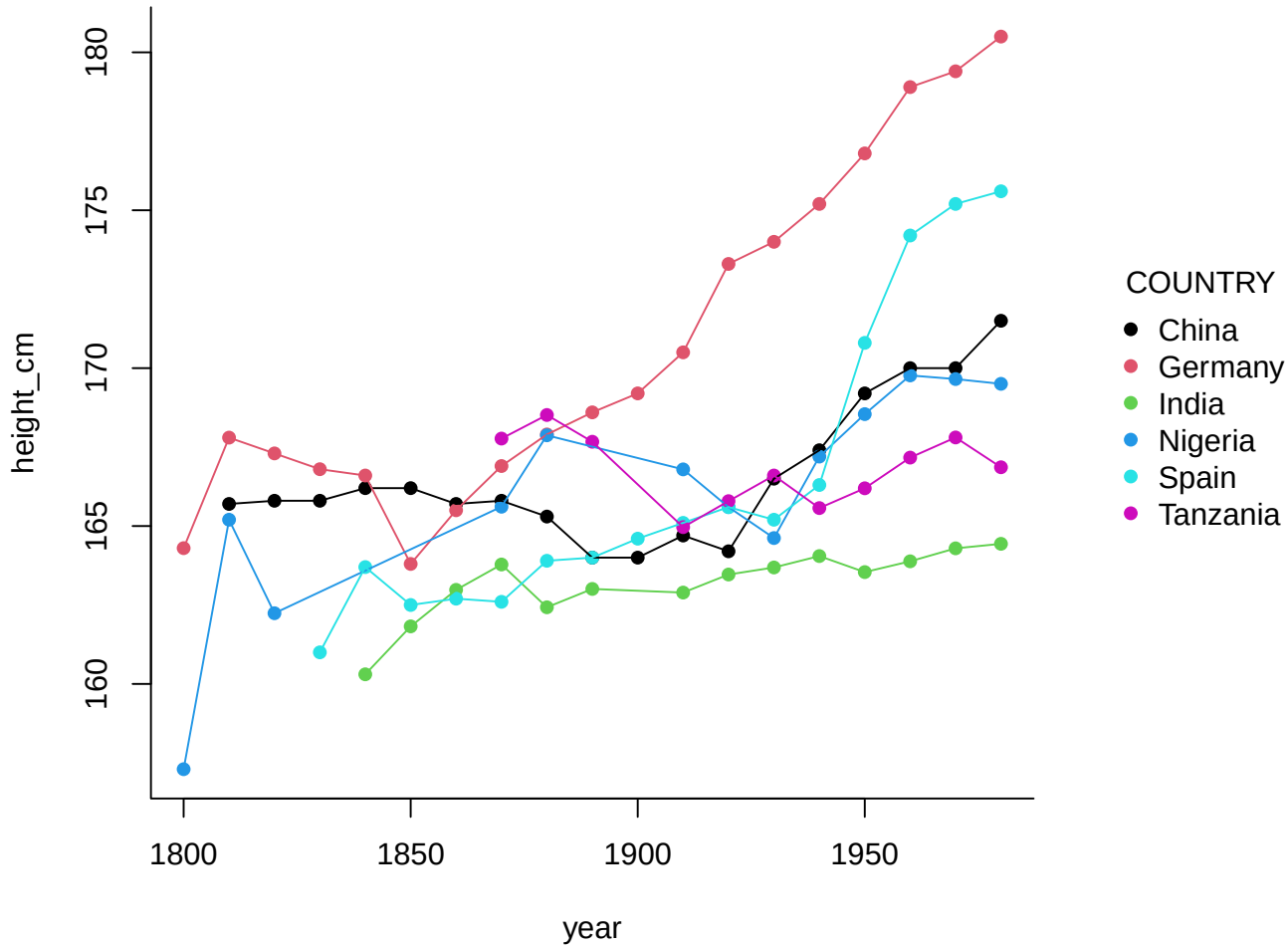
⚠ `data` should be a data frame in long format ⚠

Example

```
library(ptmixed)
library(brolgar)
df_long = as.data.frame(heights)
keep = c('China', 'India', "Germany", "Spain",
         "Nigeria", "Tanzania")
df_long = subset(df_long, country %in% keep & year >= 1800)
```

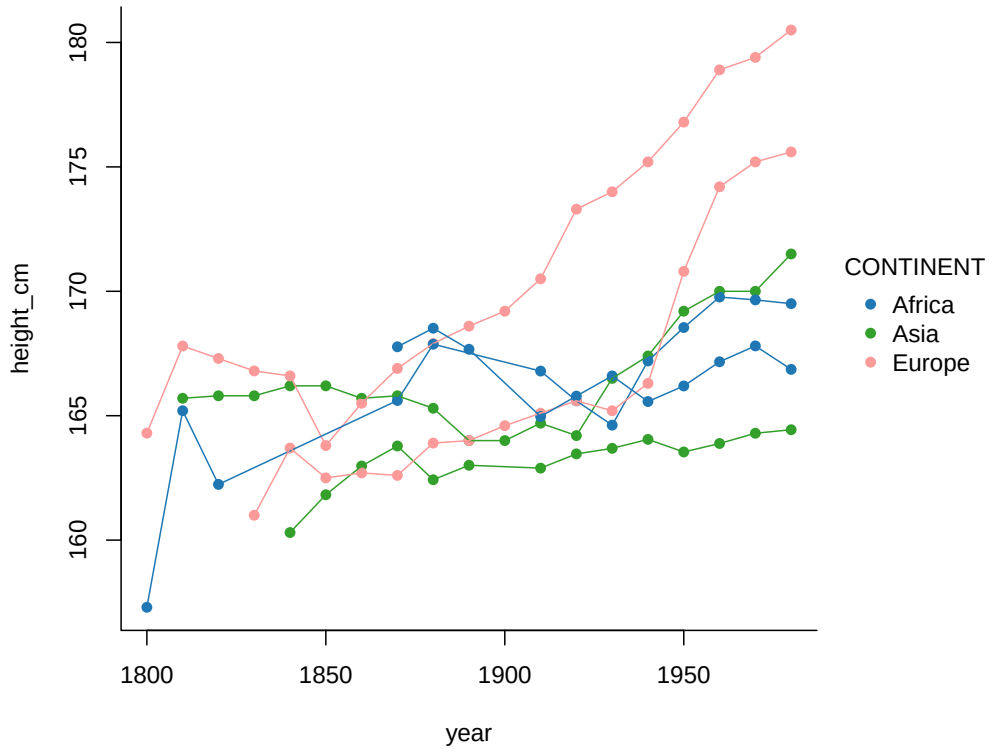
Example (cont'd)

```
make.spaghetti(year, height_cm, id = country,
               group = country, data = df_long,
               legend.title = 'COUNTRY', col = 1:6)
```



Example (cont'd)

```
make.spaghetti(year, height_cm, id = country,
               group = continent, data = df_long,
               legend.title = 'CONTINENT', legend.inset = -0.25)
```



- `group` determines line colours (e.g., by continent instead of country)
- `col` may be omitted
- `legend.inset` moves the legend to the left / right