Intro to Programming with R for Political Scientists

Session 3: Data Wrangling

Markus Freitag

Geschwister Scholl Institute of Political Science, LMU

July 12, 2021

Overview

1. Intro + R-Studio and (Git)Hub
2. Base R & Tidyverse Basics
3. **Data Wrangling**
4. Data Viz
5. Writing Functions
6. A complete scientific workflow with R

Workflow

- Navigate to `Session Scripts > Session 3` and open `Session_3_script.R`.
- You will see a pre-formatted Script with all the steps I do on the slides.
- Explore as you follow.
- If you have a second monitor, great! If not, split your screen.

Tidyverse Packages

- These are only the tidyverse packages loaded by default. There are **more** (e.g. **lubridate**).

The Philosophy: Tidy Data

> 1. Each variable forms a column.
> 2. Each observation forms a row.
> 3. Each type of observational unit forms a table. **(Wickham 2014)**

- Sounds pretty reasonable. 3. is related to constructing clean relational data bases (we will see an example later).

- However, we can relax this from time to time given our needs/statistical methods.

Messy data is everything else, e.g.:

```
##      race character gender age_0-100 age_100-500 age_500-100 age_>1000
## 1 hobbits     Frodo   male         1           0           0         0
## 2 hobbits       Sam   male         1           0           0         0
## 3   elves     Arwen female         0           0           0         1
## 4 hobbits     Golum   male         0           0           1         0
## 5 dwarves     Gimli   male         0           1           0         0
## 6     men     Eowyn female         1           0           0         0
```

The Philosophy: Tidy Data

Tidy:

```
##      race character gender age_cat
## 1 hobbits      Frodo   male   0-100
## 2 hobbits        Sam   male   0-100
## 3   elves      Arwen female   >1000
## 4 hobbits      Golum   male 500-100
## 5 dwarves      Gimli   male 100-500
## 6     men      Eowyn female   0-100
```

Tibbles

- Tibbles are data.frames but with some perks. For instance, subset more strictly.

- They also print differently (better):

```
as_tibble(lotr)
```

```
## # A tibble: 6 x 4
##   race     character gender age_cat
##   <chr>    <chr>     <chr>  <chr>
## 1 hobbits  Frodo     male   0-100
## 2 hobbits  Sam       male   0-100
## 3 elves    Arwen     female >1000
## 4 hobbits  Golum     male   500-100
## 5 dwarves  Gimli     male   100-500
## 6 men      Eowyn     female 0-100
```

```
# Hint: You can create tibbles just like data frames but with tibble().
```

Importing/Exporting Data

- R comes with it's own two file formats, `.rds` (single objects) and `.rda` (multiple objects/tabular data).
- However, for saving the latter you will (and should) use `.csv` or `.json` (human readable) most of the time.
- For R novices, importing and exporting can be a bit of a pain.
  - There are different functions/packages for reading different file formats (haven, data.table, readxl etc.).
- Thankfully, the **rio** package by Thomas Leeper and Chung-hong Chan et al. makes our life easier.
- Provides `export()` and `import` as wrappers fot the above mentioned packages.

```
install.packages("rio")
```

Importing/Exporting Data

```r
library(rio)
# Export
export(mtcars, "mtcars.csv") # R's built-in mtcars data-set.
export(mtcars, "mtcars.rds") # R serialized
export(mtcars, "mtcars.dta") # Stata
export(mtcars, "mtcars.json")

# Import
W <- import("mtcars.csv")
X <- import("mtcars.rds")
Y <- import("mtcars.dta")
Z <- import("mtcars.json")
```

Importing/Exporting Data

```r
# Exporting/importing several data frames: export_list()/import_list()

# Make a list of two built-in data sets.
# tibble::lst() automatically names the elements:
df_list <- tibble::lst(mtcars, iris)

export_list(df_list, file = paste0(names(df_list), ".csv"))
# export_file takes a character vector; hence, we build one from the names of our element
# With the paste0() we paste ".csv" to every element of the character vector
# produced by names(df_list).

Z <- import_list(dir(pattern = "csv$"))
# import_file takes achr vector holding file paths/files.
# With dir() we get all names of the files that match a specific pattern (regular expression).
# In this case, all files that end with csv ($ matches the end of the string).[1]
```

[1] **Fine Point:** **Regular expressions** were developed in computer science to specify search patterns/make character matching possible. They are very useful (e.g. for manipulating/cleaning textual data) but pretty hard to memorize. Even if you know the basics, you will google alot. In R, I recommend the **stringi/stringr** packages.

Pipes: %>% and |>

- Pipes are crucial to the tidyverse workflow but also in general.
- They make code more readable.

Idea: Take the output of a function and to pass it **as the first argument** of another function.

$$f(g(x)) \quad \text{becomes (in R)} \quad x \quad \%{>}\%^{[2]} \quad g() \quad \%{>}\% \quad f()$$

```
x <- c(1, 2, 3, 4)

sqrt(mean(x))

x %>%
  mean() %>%
  sqrt()
```

[2] **TIPP:** Press ctrl-shift-m to produce the **magrittr** pipe, `%>%`, in RStudio.

Pipes: %>% and |>

- We can also pass the output as the second, third, ... argument using `.` as a placeholder.

- `f(a, b = c)` can be written as `c %>% f(a, b = .)`

Example:

```
"Ceci n&#37;est pas une pipe" %>% gsub("&#37;", "'", .) # gsub() performs replacement of all
```

```
## [1] "Ceci n'est pas une pipe"
```

- They are so useful, the R core team even introduced a base pipe (`|>`), in May 2021. Something like this happen **very** rarely:

```
x |>
  mean() |>
  sqrt()
```

Pipes: %>% and |>

- The base pipe is a **tiny** bit faster.
- Does not need an extra dependency. Useful for package development (using the magrittr pipe in packages can be annoying sometimes).
- **BUT**: it does not (yet) properly support the `.` placeholder (as of June 2021).

We can hack it tho:

```
Sys.setenv("_R_USE_PIPEBIND_" = "true")
"Ceci n&#37;est pas une pipe" |> . => gsub("&#37;", "'", .)
```

```
## [1] "Ceci n'est pas une pipe"
```

- That's pretty ugly and **apparently** still buggy. We will stick to `%>%`.

Let's Wrangle

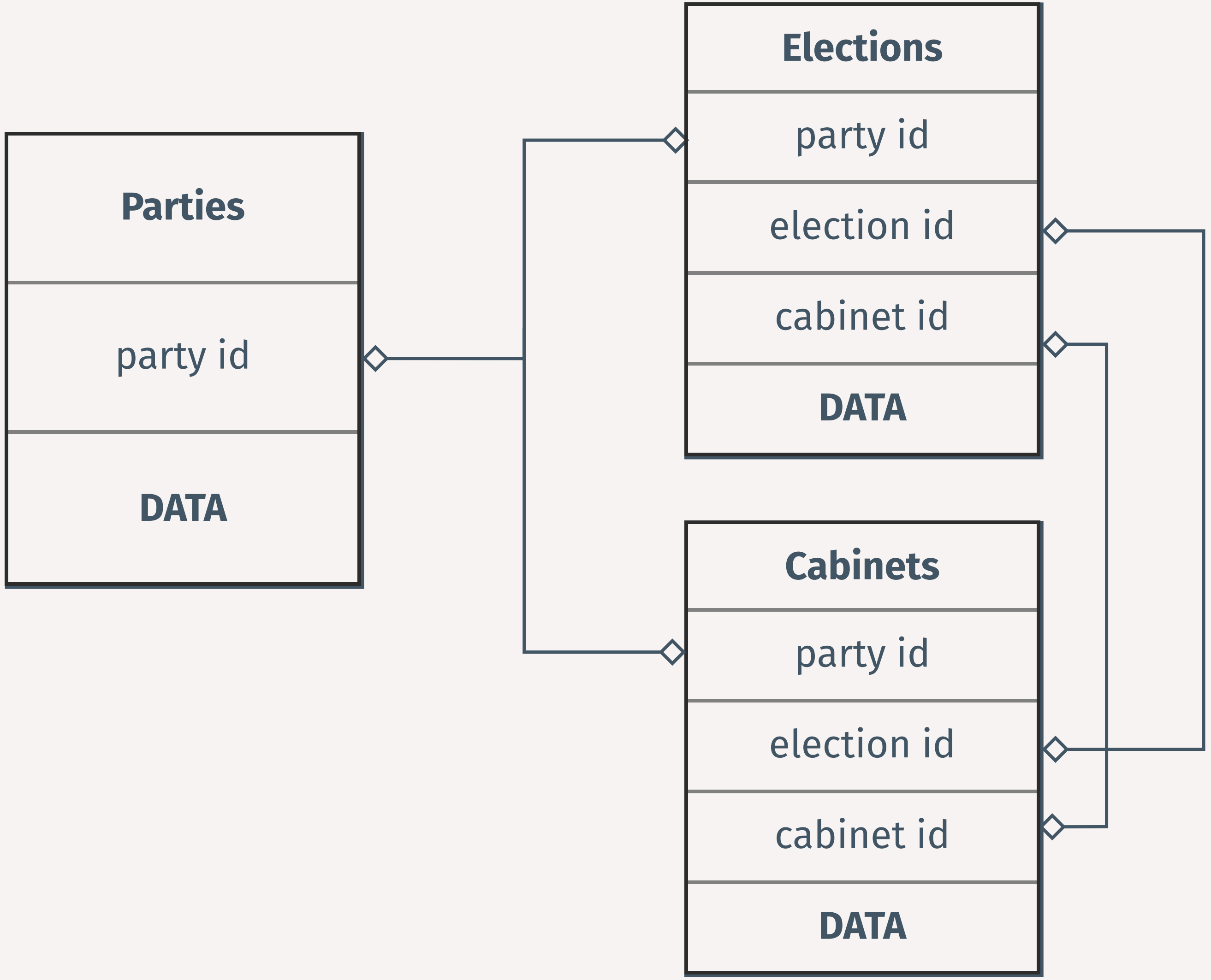
The Data

- We will use the **parlgov** database:

> ParlGov is a data infrastructure for political science and contains information for all EU and most OECD democracies (37 countries). The database combines approximately 1700 parties, 1000 elections (9400 results), and 1600 cabinets (3900 parties).

- It's relational, i.e. consists of different tables (parties, elections, cabinets) that can be **joined** using key variables. It can also be joined with the **partyfacts** dataset that provides id's for many other datasets (e.g. CLEA, ESS).

The Data



Parties
party id
DATA
Elections
party id
election id
cabinet id
DATA
Cabinets
party id
election id
cabinet id
DATA

The Data

- We will use the **parlgov** database:

> ParlGov is a data infrastructure for political science and contains information for all EU and most OECD democracies (37 countries). The database combines approximately 1700 parties, 1000 elections (9400 results), and 1600 cabinets (3900 parties).

- It's relational, i.e. consists of different tables (parties, elections, cabinets) that can be **joined** using key variables. It can also be joined with the **partyfacts** dataset that provides id's for many other datasets (e.g. CLEA, ESS).

- It makes for pretty simple examples and hence we use it.

Let's import the election data:

```
parlgov_elec <- import("http://www.parlgov.org/static/data/development-cp1252/view_election.
```

The Data: Getting an Overview

```
glimpse(parlgov_elec) # enhanced version of str()
```

```
## Rows: 8,665
## Columns: 16
## $ country_name_short                <chr> "AUS", "AUS", "AUS", "AUS", "AUS", "AU~
## $ country_name                      <chr> "Australia", "Australia", "Australia",~
## $ election_type                     <chr> "parliament", "parliament", "parliamen~
## $ election_date                     <date> 1901-03-30, 1901-03-30, 1901-03-30, 1~
## $ vote_share                        <dbl> 44.4, 34.2, 19.4, 1.4, 0.6, 29.7, 34.4~
## $ seats                             <int> 32, 26, 15, 1, 1, 26, 25, 23, 1, 0, 27~
## $ seats_total                       <int> 75, 75, 75, 75, 75, 75, 75, 75, 75, 75~
## $ party_name_short                  <chr> "PP", "FTP", "ALP", "none", "one-seat"~
## $ party_name                        <chr> "Protectionist Party", "Free Trade Par~
## $ party_name_english                <chr> "Protectionist Party", "Free Trade Par~
## $ left_right                        <dbl> 7.4000, 6.0000, 3.8833, NA, NA, 7.4000~
## $ country_id                        <int> 33, 33, 33, 33, 33, 33, 33, 33, 33, 33~
## $ election_id                       <int> 731, 731, 731, 731, 731, 730, 730, 730~
## $ previous_parliament_election_id <int> NA, NA, NA, NA, NA, 731, 731, 731, 731~
## $ previous_cabinet_id               <int> NA, NA, NA, NA, NA, 997, 997, 997, 997~
## $ party_id                          <int> 1898, 1938, 1253, 1396, 2299, 1898, 19~
```

The Data: Getting an Overview

Show 3 entries Search:

	country_name_short	country_name	election_type	election_date	vote_share	se
1	AUS	Australia	parliament	1901-03-30	44.4	
2	AUS	Australia	parliament	1901-03-30	34.2	
3	AUS	Australia	parliament	1901-03-30	19.4	

Showing 1 to 3 of 10 entries Previous 1 2 3 4 Next

Q: Is this data set tidy?

The Data: Getting an Overview

```
datasummary_skim(parlgov_elec) # from modelsummary package. Set type = "categorical" for character vars.

# Of course, not super informative in our hierarchical data set:
```

	Unique (#)	Missing (%)	Mean	SD	Min	Median	Max	
vote_share	2459	6	11.9	12.9	0.0	6.6	71.2	
seats	291	1	23.3	45.7	0.0	6.0	470.0	
seats_total	209	0	212.2	181.3	5.0	151.0	709.0	
left_right	413	10	5.1	2.4	0.0	5.7	9.8	
country_id	37	0	37.2	20.5	1.0	37.0	75.0	
election_id	998	0	569.5	320.2	1.0	583.0	1094.0	
previous_parliament_election_id	794	3	543.0	301.5	1.0	563.0	1079.0	
previous_cabinet_id	833	3	752.0	468.0	2.0	734.0	1634.0	
party_id	1559	0	1120.2	738.8	2.0	1015.0	2812.0	

TIPP: The **collapse** package also provides some data summary functions (`descr()` and `qsu()`) that work really well on huge data sets. Stata users will love it as its similar to summarize.

Tidyverse's Wrangling "Wunderkind": dplyr

© **Alison Horst**

A full list of `dplyr` functions can be found **here**.

Data Masking

- Before we jump right in, remember when we talked about R environments in the last session?
- When we wanted to pass a variable of a data frame (or element of a list) to a function, we needed to make that **explicit**, e.g., using `$` for subsetting.

```
lm(df$y ~ df$x)
```

- Some important dplyr functions use **data masking** such that you can refer to variables just like objects in the (global) environment.
- Get's rid of **$**s.

Filtering Rows

- Let's start with something very basic: filtering rows.

- For instance, we might want to obtain only the results of all Bundestag elections:

```
parlgov_elec_de <- parlgov_elec %>% # add, e.g., _de if we want to keep our original df
filter(country_name_short == "DEU")
```

- Now, say we want the mean SPD vote share across all bundestag elections (bit silly but alas):

```
parlgov_elec_de %>% # add, e.g., _de if we want to keep our original df
  filter(party_name_short == "SPD", election_type == "parliament") %>%
  mean(vote_share)
```

- That does not work...**Q**: Why?

Note that using convenience wrappers such as `datasummary()` are no substitute for learning the underlying base, tidyverse or data.table-way of achieving this.

pull()

- Base-R's summary stats functions can't take a df or tibble and we did not wrap it into a dplyr function (**in order for data masking to kick in**).

Two options:

1. We can use `pull()` to pull out a vector form a df; works like `$` but for pipes.

```
parlgov_elec_de %>% # add, e.g., _de if we want to keep our original df
  filter(party_name_short == "SPD", election_type == "parliament") %>%
  pull(vote_share) %>%
  mean()
```

```
## [1] 31.70964
```

summarise()

2. We can wrap `mean()` into `summarise()` ("data-masking" for built-in summary functions):

```
parlgov_elec_de %>%
  filter(party_name_short == "SPD", election_type == "parliament") %>%
  summarise(mean(vote_share)) # summarise() takes summary functions such as mean(), sd(), et
```

```
##   mean(vote_share)
## 1         31.70964
```

summarise()

- Puts out a tibble we can pass to other dplyr functions.
- We can also compute more summary statistics with `summarise()` **AND** also turn them into variables:

```
parlgov_elec_de %>%
  filter(party_name_short == "SPD", election_type == "parliament") %>%
  summarise(mean = mean(vote_share), sd = sd(vote_share), n = n())
```

```
##       mean       sd  n
## 1 31.70964 8.374648 28
```

```
# Aside: you can also get the number of rows (in a group) by using the base
# function nrow(.) with a placeholder in the above pipe.
```

- In that, it is somewhat similar to the `mutate()` command... but it **collapses the data**.

mutate()

- With `mutate()` you can add columns - or overwrite existing ones (with the same name) - based on transformations of other variables

- For instance, say we want to add a new column "year" based on the election date variable:

```
parlgov_elec_de <- parlgov_elec_de %>% # here, we "overwrite" our df
  mutate(year = lubridate::year(election_date))
```

- Here, we used the `year()` function of the **lubridate** package.

- We pulled out an integer vector indicating the election year from the `date` variable and assigned it to a new variable.[3]

- `transmute()` is the opposite of `mutate()`; i.e. adds a new variable and drops the rest.

[3] **NOTE** We will use some core `lubridate` functions on the fly as they are rather intuitive. If you deal with dates often, give **this** intro a read.

select() and arrange()

- Another basic wrangling operation we occasionally need to do do is selecting columns.

- And sometimes we also want to sort columns:

```
parlgov_elec_de %>%
  filter(year == 2017) %>%
  select(party_name_short, vote_share, left_right) %>%
  arrange(desc(left_right)) #default is ascending; we can wrap the masked vector with desc() to sort desc
```

```
##   party_name_short vote_share left_right
## 1              AfD       12.6     8.8000
## 2               FW        1.0     7.4000
## 3              CSU        6.2     7.2871
## 4              CDU       26.8     6.2503
## 5              FDP       10.7     5.9233
## 6              SPD       20.5     3.6451
## 7          B90/Gru        8.9     2.9308
## 8           PDS|Li        9.2     1.2152
## 9           PARTEI        1.0         NA
```

group_by()

- The last of the most fundamental dplyr "verbs" we will learn is `group_by`.
- Given the variables you supply, it converts a data frame into a grouped version of it such that you can do transformations, call functions, manipulate variables, etc.
- For instance, let's use the whole data to get the the party with the most votes ever in a parliamentary election by country...

group_by()

```
parlgov_elec %>%
  filter(election_type == "parliament") %>%
  group_by(country_name_short) %>%
  summarise(share_max = max(vote_share, na.rm = TRUE), # In many base functions, default is not to ignore missings.
            party = party_name_english[1],
            election_date = election_date[1]
            ) %>%  # note the "collapsing" I mentioned earlier
  arrange(desc(share_max))
```

```
## # A tibble: 37 x 4
##    country_name_short share_max party                     election_date
##    <chr>                  <dbl> <chr>                     <date>
##  1 CYP                     71.2 Democratic Party          1976-09-05
##  2 LVA                     68.2 Popular Front of Latvia   1990-04-29
##  3 ROU                     66.3 Democratic Party          1990-05-20
##  4 MLT                     59.9 Malta Labour Party        1947-10-27
##  5 JPN                     59   Japan Liberal Party       1946-04-10
##  6 NZL                     58.7 New Zealand Liberal Party 1902-11-25
##  7 CAN                     57   Liberal Party of Canada   1900-11-07
##  8 CHE                     56.2 Radical Democratic Party  1902-10-26
##  9 BEL                     56   Catholic Party            1900-05-27
## 10 GBR                     55.0 Conservatives             1918-12-14
## # ... with 27 more rows
```

group_by()

- Too complex?
- Ok, let's do something more simple and add a variable that tells us the maximum number of total seats for each country.

```
parlgov_elec <- parlgov_elec %>%
  group_by(country_id) %>%
  mutate(max_seats = max(seats_total)) %>%
  ungroup() # to remove the grouping
```

group_by()

```
parlgov_elec %>%
  filter(seats_total == max_seats) %>%
  select(country_name_short, election_date, max_seats) %>%
  distinct() # select distinct rows
```

```
## # A tibble: 309 x 3
##    country_name_short election_date max_seats
##    <chr>              <date>            <int>
##  1 AUS                2019-05-18          151
##  2 AUT                1920-10-17          183
##  3 AUT                1971-10-10          183
##  4 AUT                1975-10-05          183
##  5 AUT                1979-05-06          183
##  6 AUT                1983-04-24          183
##  7 AUT                1986-11-23          183
##  8 AUT                1990-10-07          183
##  9 AUT                1994-10-09          183
## 10 AUT                1995-12-17          183
## # ... with 299 more rows
```

Other usefull stuff

- `tidyr::separate()` to separate entries in columns; `tidyr::unite` to unite columns
- `dplyr::rename()` to rename variables.
- `dplyr::count()` count unique values of one or multiple variables.
- `dplyr::distinct()` or Base R `unique()` to remove duplicate rows
- `dplyr::slice()` to get a slice of rows: `parlgov_elec %>% slice(1:5)` to get the first 5 rows.
- `dplyr::across()` apply transformations to multiple columns.
- `modelsummary::datasummary_crosstab()` for easy two- or multi-dimensional cross tabulations (for Stata `tabulate` addicts).

TIPP Check out the `modelsummary` functions in more detail for easy crosstabs. ALternatively, use `janitor::tabyl`.

Session 3 - Problem Set

Advanced Wrangling

Dealing with factor variables: forcats

- Factor variables are useful, especially for plotting and modelling.
- With `factor_recode`, we can easily recode levels:

```
parlgov_elec_de %>%
  mutate(election_type = fct_recode(election_type, # Coerces the type automatically from chr
    Bundestagswahl = "parliament",
    Europawahl = "ep"
  )) %>%
  count(election_type)
```

```
##     election_type   n
## 1      Europawahl  82
## 2 Bundestagswahl 264
```

- With `fct_recorder` we can reorder factors (will be useful for plotting factors).

Complex conditions: if_else and case_when

- Often, we also want to manipulate variables by means of complex conditions
- We will go deeper into control flow statements next week, but here is a sneak preview for data wrangling.
- Say we want to create a variable, "family", that puts parties into some party family based on some arbitrary cutoff of the time-invariant left_right position:

```
parlgov_elec_de <- parlgov_elec_de %>%
  mutate(family = if_else(left_right > 5, "right", "left"))
```

- Vectorised if: `if_else(condition, true, false)`.

Complex conditions: if_else and case_when

- A generalised version of `if_else` is `case_when`.
- This is **very** useful:

```
parlgov_elec_de <- parlgov_elec_de %>%
  mutate(family = case_when(
    left_right <= 2.5 ~ "left",
    left_right > 2.5 & left_right < 5 ~ "centre-left",
    left_right > 5 & left_right < 7.5 ~ "centre-right",
    left_right >= 7.5 ~ "right"))
```

- two-sided formula: LHS = logical test; RHS = value to assign if the test is `TRUE`.
- Values that do not fall into any of the conditions become `NA` which can be prevented by adding `TRUE ~ something` as the last argument.

tidyr: reshaping data

- Reshaping data is one of the key things you need to when cleaning/analysing data.
- **Two functions**:
    - `tidyr::pivot_wider/longer()` is for reshaping from long (wide) to wide (long).

Two main arguments:

- `names_*` and `values_*`, where "*" is "to" for `pivot_wider()` and "from" for `pivot_longer()`.

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T

→

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	NA	NA

long → wide: `pivot_wider()`

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

→

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

wide → long: `pivot_longer()`

tidyr: reshaping data

Example:

- Say, we want a table of the vote shares of all major parties for each post-WW2 parliamentary election.
- Where each row is an election:

```
wide <- parlgov_elec_de %>%
  filter(election_type == "parliament", vote_share >= 5, year(election_date) >= 1945) %>%
  select(election_date, party_name_short, vote_share) %>%
  pivot_wider(names_from = party_name_short, values_from = vote_share)
```

tidyr: reshaping data

Show 5 entries　　　　Search:

	election_date	SPD	CDU	FDP	CSU	KPD	GB/BHE	B90/Gru	PDS|Li	AfD
1	1949-08-14	29.2	25.2	11.9	5.8	5.7				
2	1953-09-06	28.8	36.4	9.5	8.8		5.9			
3	1957-09-15	31.8	39.7	7.7	10.5					
4	1961-09-17	36.2	35.8	12.8	9.6					
5	1965-09-19	39.3	38.2	9.5	9.6					

Showing 1 to 5 of 19 entries

Previous 1 2 3 4 Next

tidyr: reshaping data

- We can revert back to long format:

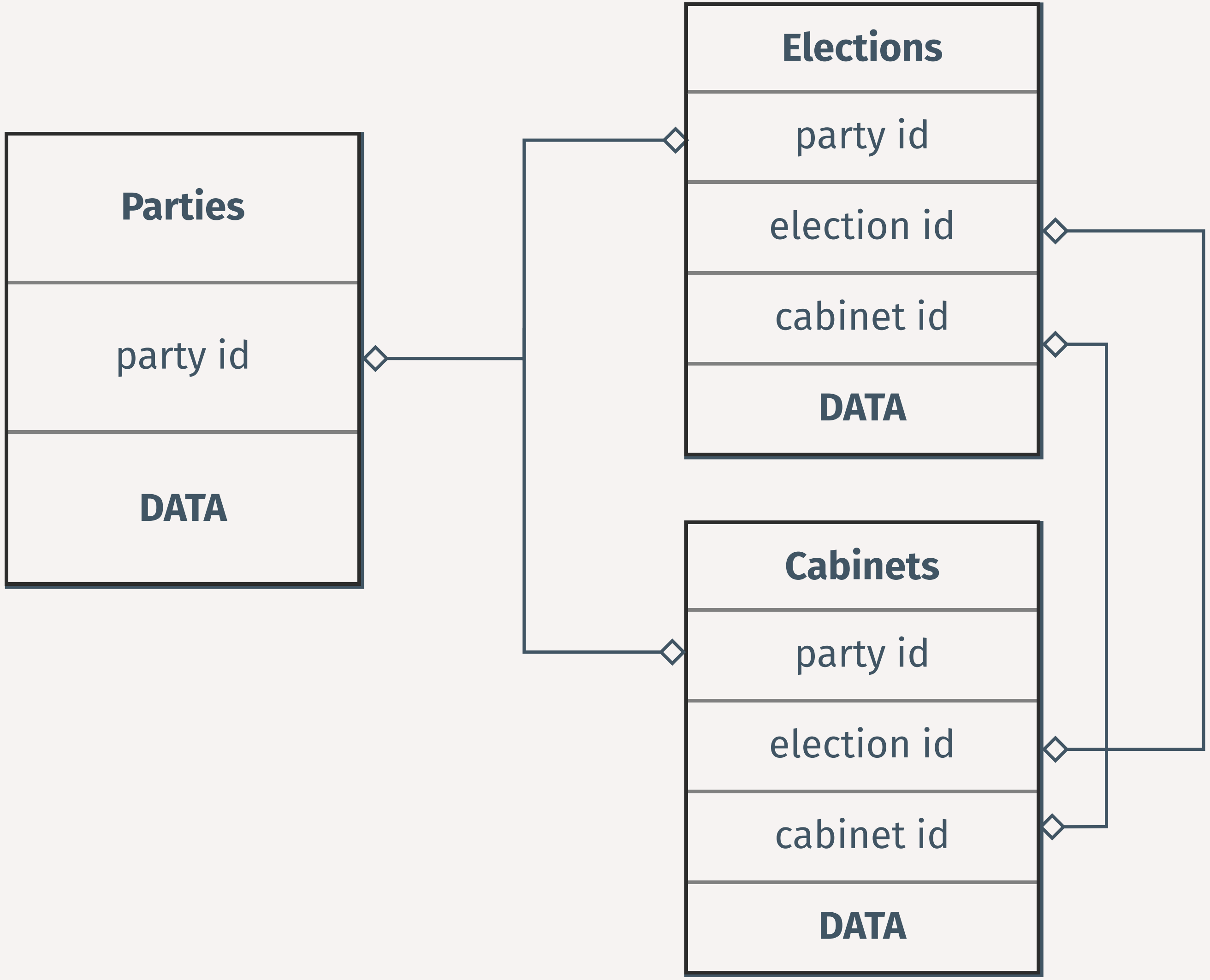
```
long <- wide %>%
  pivot_longer(!election_date, names_to = "party_name_short", values_to = "vote_share") %>%
  filter(is.na(vote_share) == FALSE) # alternatively, simply set values_drop_na to TRUE in p
head(long)
```

```
## # A tibble: 6 x 3
##   election_date party_name_short vote_share
##   <date>        <chr>                 <dbl>
## 1 1949-08-14    SPD                    29.2
## 2 1949-08-14    CDU                    25.2
## 3 1949-08-14    FDP                    11.9
## 4 1949-08-14    CSU                     5.8
## 5 1949-08-14    KPD                     5.7
## 6 1953-09-06    SPD                    28.8
```

dplyr: joins

- Let's come back to the relational nature of our data...

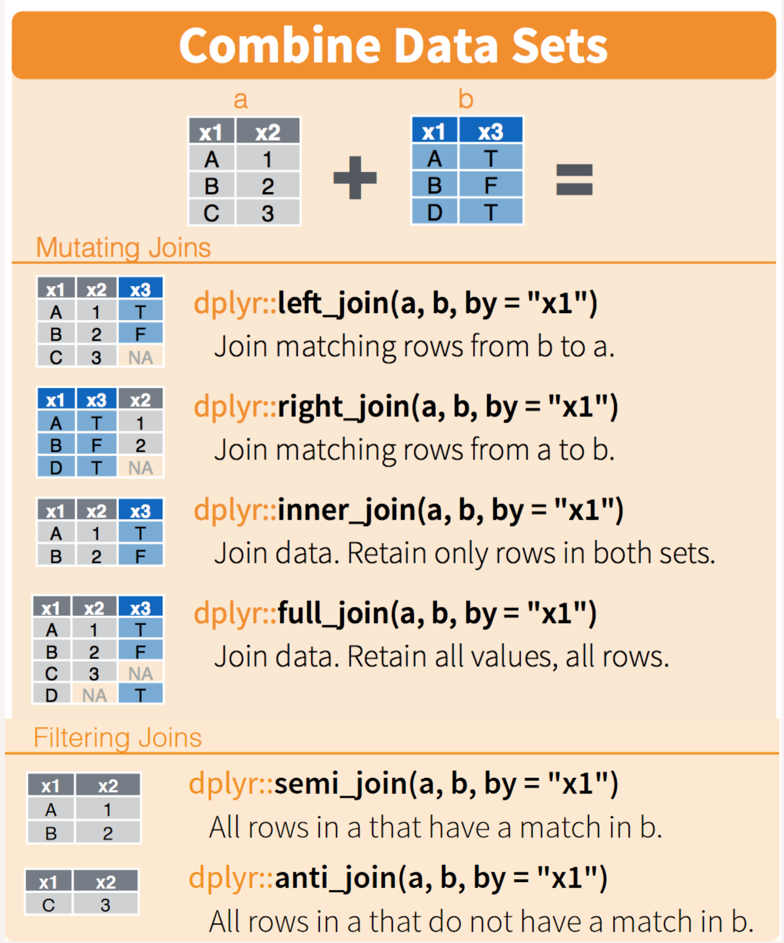
The Data


Parties
party id
DATA
Elections
party id
election id
cabinet id
DATA
Cabinets
party id
election id
cabinet id
DATA

dplyr: joins

- Let's come back to the **relational** nature of our data...
- Remember, they consist of different tables, each representing one "entity type" (c.f. the 3rd. point in Wickham's tidy data framework)
- Each table has a unique key, representing each row. This key variable is used to link/join tables
- **Suppose we want to join the party and the election table, how do we do that?**

dplyr: joins



Q Which join do we need?

dplyr: joins

We want a left join here...

```
parlgov_party <- rio::import("http://www.parlgov.org/static/data/development-utf-8/view_party.csv")

l_joined <- left_join(parlgov_elec_de, parlgov_party, by = "party_id")

head(l_joined)
```

```
##   country_name_short.x country_name.x election_type election_date vote_share
## 1                  DEU        Germany    parliament    1919-01-19      37.87
## 2                  DEU        Germany    parliament    1919-01-19      18.32
## 3                  DEU        Germany    parliament    1919-01-19      15.45
## 4                  DEU        Germany    parliament    1919-01-19      10.26
## 5                  DEU        Germany    parliament    1919-01-19       4.66
## 6                  DEU        Germany    parliament    1919-01-19       7.63
##   seats seats_total party_name_short.x
## 1   165         423                SPD
## 2    74         423                DDP
## 3    73         423                 DZ
## 4    41         423               DNVP
## 5    23         423                DVP
```

dplyr: joins

- There are multiple matching (by name) variables in both tables.
- Hence, we need to specify all keys, or let `dplyr` do its magic:

```
l_joined <- left_join(parlgov_elec_de, parlgov_party)

head(l_joined)
```

```
##   country_name_short country_name election_type election_date vote_share seats
## 1                DEU      Germany    parliament    1919-01-19      37.87   165
## 2                DEU      Germany    parliament    1919-01-19      18.32    74
## 3                DEU      Germany    parliament    1919-01-19      15.45    73
## 4                DEU      Germany    parliament    1919-01-19      10.26    41
## 5                DEU      Germany    parliament    1919-01-19       4.66    23
## 6                DEU      Germany    parliament    1919-01-19       7.63    22
##   seats_total party_name_short
## 1         423              SPD
## 2         423              DDP
## 3         423               DZ
## 4         423             DNVP
```

Alternative approaches

- For every tidyverse function ("verb"), there is, of course, **a base R way to do it**.
- There are alternatives.
- For instance, the **data.table** and **collapse** (also comes with fast versions of summary stats and models) package provide great and fast data wrangling alternatives.
- Data.table syntax is closer to the base R way of indexing/manipulating data frames. Some like that.
- Don't be dogmatic. Use whatever suits you and your context. Mix stuff.
- You can find a great comparison of `data.table` and `dplyr` **here**.

A glimpse at data.table

- Comes with its own interpretation of data frames, "data tables". Special structure to work faster.
- Looks similar to basic `df[]` indexing but with alot of twists.
- Three elements: which observations/rows COMMA transformations or other functions COMMA grouping.

Rough `dplyr` equivalent:

DT[`slice(); filter(); arrange()`, `select(); mutate()`, `group_by()`]

A glimpse at data.table

Example:

```r
parlgov_elec_de %>% # add, e.g., _de if we
  filter(party_name_short == "SPD") %>%
  summarise(mean(vote_share, na.rm = T))
```

```
##   mean(vote_share, na.rm = T)
## 1                    31.12622
```

```r
setDT(parlgov_elec_de)

parlgov_elec_de[party_name_short == "SPD",
```

```
## [1] 31.12622
```

Problem Set 03/"Homework"

- This last problem set may take a bit longer.
- Try to get as far as you can in the remaining time and finish the rest at home (if you want).

Q & A

Next Up: Data Viz (Next Week)