tidyr and stringr

Silvie Cinková

2025-08-13

Table of contents

1 Libraries

```
library(readr)
library(dplyr)
library(tidyr)
library(magrittr)
library(gapminder)
library(ggplot2)
library(tidytext)
library(stringr)
```

2 `tidyr`

- lumps and splits column values into new columns
- transforms several variable columns into categories of a new variable and the other way round
- completes observations with **missing values**
- drops rows with missing values in specified columns
- expands observations with all possible groups

- manages **nested columns** (vector/list inside column!)

This library is a "heavy-duty" assistant to `dplyr`. You use it when you need to make your data tidy for your purposes. Remember, tidy means that each row represents an observation and each column represents one variable. Sometimes it is a matter of perspective. Maybe we could say that the data is tidy when you can map all relevant variable to ggplot aesthetic scales.

In this lecture, we are going to focus on column transformations with **uniting**, **separating**, and **pivoting**.

`tidyr` is a library that has been evolving through the latest years. There were originally two pairs of basic verbs: `separate` with `unite` and `spread` with `gather`.

`separate` used to separate columns according to separators/delimiters provided by the user. It has been superseded by a family of `separate_wider_` functions. Use the names in the billionaires data set to play around with these functions and their arguments and extract different parts of names. It is always going to be messy, but you will get the idea of which choices you have to take when working with unstructured text data (strings).

`gather` and `spread` have been replaced with `pivot_longer` and `pivot_wider`. That is good to know, in case you are going to borrow older scripts from others. LLM copilots are likely to still occasionally use them, too.

I strongly recommend the DataCamp course Reshaping Data with `tidyr`, because `tidyr` needs a real hands-on experience to let the key concepts sink in properly. And you will benefit a lot from their `stringr` course as well (not assigned as home work).

3 Data

```
billionaires_df <- read_tsv("~/DATA.NPFL112/billionaires_combined.tsv",
 ↪  show_col_types = FALSE)
bil_unite <- billionaires_df %>% distinct(name.x, birth_comb) %>%
 ↪  drop_na(birth_comb)
bil_separate <- billionaires_df %>% distinct(name.x, person) %>%
 ↪  drop_na(name.x)
gap_cze_ger_gdp <- gapminder %>%
  filter(country %in% c("Czech Republic", "Germany")) %>%
  select(country, year, gdpPercap)
```

We are going to use the `billionaires` and `gapminder` data sets to demonstrate the most important column transformations. `billionaires` is an adapted version of a Gapminder data set from their `open-numbers` GitHub repository. In the end of this session, we will try and figure out in which business domains they accrued their wealth, using the `tidyr` functions and a `ggplot`-based library for social network analysis.

4 billionaires_df (selected columns)

```
#|echo: false
slice_head(billionaires_df, n = 5) %>%
  select(c(person, name.x, industry, countries, time)) %>%
  kableExtra::kable()
```

```
Warning in attr(.knitEnv$meta, "knit_meta_id"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")

Warning in attr(.knitEnv$meta, "knit_meta_id"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")

Warning in attr(.knitEnv$meta, "knit_meta_id"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")

Warning in attr(.knitEnv$meta, "knit_meta_id"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")

Warning in attr(.knitEnv$meta, "knit_meta_id"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")

Warning in attr(.knitEnv$meta, "knit_meta_id"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")

Warning in attr(.knitEnv$meta, "knit_meta_id"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")

Warning in attr(.knitEnv$meta, "knit_meta_id"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")

Warning in attr(.knitEnv$meta, "knit_meta_id"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
```

```
See help("Deprecated")

Warning in attr(.knitEnv$meta, "knit_meta_id"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")

Warning in attr(.knitEnv$meta, "knit_meta_id"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")

Warning in attr(.knitEnv$meta, "knit_meta_id"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")

Warning in attr(.knitEnv$meta, "knit_meta_id"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")

Warning in attr(.knitEnv$meta, "knit_meta_id"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")

Warning in attr(x, "align"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")

Warning in attr(x, "format"): 'xfun::attr()' is deprecated.
Use 'xfun::attr2()' instead.
See help("Deprecated")
```

person	name.x	industry	countries	time
a_jayson_adair	A. Jayson Adair	Automotive	usa	2021
a_jayson_adair	A. Jayson Adair	Automotive	usa	2022
a_jerrold_perenchio	A. Jerrold Perenchio	Media & Entertainment; Media;Entertainment	usa	2002
a_jerrold_perenchio	A. Jerrold Perenchio	Media & Entertainment; Media;Entertainment	usa	2003
a_jerrold_perenchio	A. Jerrold Perenchio	Media & Entertainment; Media;Entertainment	usa	2004

5 `tidyr::unite` (before)

```
bil_unite %>% slice(1:10)
```

```
# A tibble: 10 x 2
   name.x                          birth_comb
   <chr>                                <dbl>
 1 A. Jerrold Perenchio                  1931
 2 Abdulla bin Ahmad Al Ghurair          1955
 3 Abdullah bin Sulaiman Al Rajhi        1929
 4 Abdulsamad Rabiu                      1960
 5 Abhay Soi                             1973
 6 Abhay Vakil                           1952
 7 Abigail Johnson                       1962
 8 Abilio dos Santos Diniz               1937
 9 Achal Bakeri                          1961
10 Acharya Balakrishna                   1972
```

This will be a somewhat artificial use case: merge the values of the two columns into one column called `birth_comb`. Separate the name from the year of birth with three asterisks to obtain something like this: `John Brown***1970`. In this slide you see the source data.

6 `tidyr::unite` (after)

```
bil_unite %>% slice(1:10) %>%
  unite(col = ID, name.x, birth_comb, sep = "***", remove = FALSE) %>%
  kableExtra::kable()
```

ID	name.x	birth_comb
A. Jerrold Perenchio***1931	A. Jerrold Perenchio	1931
Abdulla bin Ahmad Al Ghurair***1955	Abdulla bin Ahmad Al Ghurair	1955
Abdullah bin Sulaiman Al Rajhi***1929	Abdullah bin Sulaiman Al Rajhi	1929
Abdulsamad Rabiu***1960	Abdulsamad Rabiu	1960
Abhay Soi***1973	Abhay Soi	1973
Abhay Vakil***1952	Abhay Vakil	1952
Abigail Johnson***1962	Abigail Johnson	1962
Abilio dos Santos Diniz***1937	Abilio dos Santos Diniz	1937

ID	name.x	birth_comb
Achal Bakeri***1961	Achal Bakeri	1961
Acharya Balakrishna***1972	Acharya Balakrishna	1972

This will be the function and its output. The `unite` function is very simple and seems to have been working in the same way ever since, but watch out for its counterpart `separate`, which was superseded by a family of more specialized functions.

Next slide: a necessary digression to operations on strings. Separating values into several columns is just one special context of string operations.

7 Operations on strings

- preparation for `tidyr::separate`
- Regular Expressions ("_regex_es")
- the `stringr` library (or base R `grep` & Co.)
- regex tutorial https://www.regexone.com/

The next `tidyr` function to learn would be the counterpart to `unite`, which would help you separate one column into several according to some strings in their values. For instance, you have a column structured as `Surname, Name` and you will want to separate first names from surnames into two separate columns. You will do it based on a rule that you formulate: first comes surname, then comes first name, and they are separated by a comma and a space. This is a suitable moment to discuss operations on strings.

Regular expressions: templates that you want to match in strings. Regular vs. fixed expressions: some characters (e.g. `!`, `?`, `.`, `[]`, `^`, `*`) have a special meaning in the regular expressions syntax. This is resolved in different functions in diverse ways. Sometimes there is an argument called `fixed` (fixed expression) or `pattern` (that would always be a regular expression). Sometimes there is no such argument option, but you can often say that something is a regular expression by putting the string into the `regex` function, or, the other way round, into the `fixed` function. Remember that they exist and look them up once you think you need them!

A regular expression is very often used in string functions as a test/condition: When it matches a string, it "is true" and then it depends on the function what this is used for. For instance, a function can just return a TRUE or FALSE, when a pattern gets detected. Or it gets extracted. Or it gets deleted. Or you get numerical indices of the start and the end of the matched pattern inside a longer string. Or it gets replaced with a string you have specified as a replacement string.

[!T IP] Check out the `stringr` library to find out how you can extract information from and modify unstructured textual data!

Great regex tutorial https://www.regexone.com/ https://www.regextester.com/

8 Isolated strings: prep for `tidyr::separate`

- split values into two or more new columns
- like `stringr::str_split` or `base::strsplit` on strings
- eats the separators
- define the separator as a fixed string or escape with `\`

First comes a mini use case, example follows on the next slide. You will get a string with noise and want to extract just meaningful words in a sentence.

9 Isolated strings: prep for `tidyr::separate`

```
stringr::str_split(string = "I*like*dogs", pattern = stringr::fixed("*"),
↪  simplify = TRUE)
```

```
     [,1] [,2]   [,3]
[1,] "I"  "like" "dogs"
```

```
stringr::str_split(string = "I*like*dogs", pattern = stringr::fixed("*"),
↪  simplify = TRUE)
```

```
     [,1] [,2]   [,3]
[1,] "I"  "like" "dogs"
```

10 A more general regex

```
str_split(string = "I*A3like*B3dogs", pattern = "\\*[A-Z]\\d", simplify =
↪  TRUE)
```

```
     [,1] [,2]   [,3]
[1,] "I"  "like" "dogs"
```

```
str_split(string = "I*A3like*B3pineapples83dogs", pattern = "\\*[^*]*\\d",
 ↪  simplify = TRUE)
```

```
     [,1] [,2]   [,3]
[1,] "I"  "like" "dogs"
```

Escape special characters = tell the computer that you treat them as ordinary characters. In many programming languages you do it with a backslash before. In R, you sometimes need two (or three) backslashes to get understood. This depends on how and which regular expressions search engine is implemented in which R library. I keep double-checking when debugging my regular expressions in R…

These regular expressions are here only to illustrate the power of string search. You have to learn the rules properly. So for instance: Asterisk means "no or any number of occurrences of the preceding character". Anything in square brackets is read literally (hence, an asterisk in square brackets means just an ordinary asterisk). The pair of square brackets means just one character. When you put more than one inside, like here the asterisk, it means that this position can be populated by an asterisk. But watch out for the caret before: it negates the entire selection. So it actually means: "This position can accommodate one character, but it must not be an asterisk". The asterisk after the pair of square brackets means that the previous position can occur there any number of times, including zero times. And after that comes a digit encoded as `\\d`. So this time the double backslash did not even mean escaping a special character!

Whenever this pattern gets matched in a long string, the sequence disappears and the string will be separated in this place.

11 Split into columns

```
set.seed(23)
bil_separate %>% slice_sample(n = 10) %>%
  separate_wider_delim(col = name.x, cols_remove = FALSE,
    names = c("firstnames", "middlenames", "lastnames" ),
    names_repair = "minimal",
    delim = " ",
    too_few = "align_end",
    too_many = "merge"
    )
```

```
# A tibble: 10 x 5
   firstnames middlenames lastnames   name.x              person
   <chr>      <chr>       <chr>       <chr>               <chr>
 1 <NA>       Xiong       Haitao      Xiong Haitao        xiong_haitao
 2 <NA>       Paul        Gauselmann  Paul Gauselmann     paul_gauselmann
 3 <NA>       Duan        Yongping    Duan Yongping       duan_yongping
 4 <NA>       Vladimir    Gruzdev     Vladimir Gruzdev    vladimir_gruzdev
 5 <NA>       Dinesh      Nandwana    Dinesh Nandwana     dinesh_nandwana
 6 Forrest    Mars,       Jr.         Forrest Mars, Jr.   forrest_mars_jr
 7 Wilbur     Ross,       Jr.         Wilbur Ross, Jr.    wilbur_ross_jr
 8 <NA>       Andreas     Struengmann Andreas Struengmann andreas_struengmann
 9 <NA>       Huang       Qiaolong    Huang Qiaolong      huang_qiaolong
10 <NA>       Hasmukh     Chudgar     Hasmukh Chudgar     hasmukh_chudgar
```

There are several `tidyr` functions to separate column values into individual columns. The documentation tags them as being in the experimental stage, so they might still substantially change. Anyway, here are `separate_wider_delim`, `separate_wider_regex`, and `separate_wider_position`.

12 separate_wider_delim

```
set.seed(1950)
bil_unite %>% select(!c(birth_comb)) %>%
slice_sample(n = 10) %>%
  separate_wider_delim(col = name.x,
    too_few = "debug", too_many = "debug",
    names = c("first_name", "surname"), delim = " ")
```

```
Warning: Debug mode activated: adding variables `name.x_ok`, `name.x_pieces`, and
`name.x_remainder`.
```

```
# A tibble: 10 x 6
   first_name surname  name.x           name.x_ok name.x_pieces name.x_remainder
   <chr>      <chr>    <chr>            <lgl>           <int> <chr>
 1 Peter      Spuhler  Peter Spuhler    TRUE                2 ""
 2 G.         M.       G. M. Rao        FALSE               3 " Rao"
 3 Lorenzo    Mendoza  Lorenzo Mendoza  TRUE                2 ""
 4 Vadim      Yakunin  Vadim Yakunin    TRUE                2 ""
 5 Kutayba    Alghanim Kutayba Alghanim TRUE                2 ""
 6 Minoru     Mori     Minoru Mori      TRUE                2 ""
```

```
 7 Li          Guangyu  Li Guangyu       TRUE                  2 ""
 8 Baokun      Bai      Baokun Bai       TRUE                  2 ""
 9 Luiza       Helena   Luiza Helena Tr~ FALSE                 3 " Trajano"
10 Han         Xiao     Han Xiao         TRUE                  2 ""
```

In `separate_wider_delim`, the delimiter on which you split is by default a fixed expression, but you can override it with `regex()`. Many `tidyr` functions have arguments that generate column names for new columns, and you can specify the way they do it. We will not go into this here. Instead, look at the `too_few` and `too_many` arguments: here you can specify what you want the function to do when there are more or fewer splits than you expect.
The `separate_wider_regex` function allows you to put in several patterns on which to split.

13 separate_wider_regex

```
set.seed(1950)
bil_unite %>% select(!c(birth_comb)) %>%
slice_sample(n = 10) %>%
  separate_wider_regex(col = name.x,
    too_few = "debug",
    patterns = c(first_name = "(?:^[^\\s]+)",
      middle = "(?:\\1? ?[^ ]+)?",
      surname = "(?:.*\\s[^\\s]+$)"

  ))
```

```
Warning: Debug mode activated: adding variables `name.x_ok`, `name.x_matches`, and
`name.x_remainder`.
```

```
# A tibble: 10 x 7
   first_name middle    surname name.x name.x_ok name.x_matches name.x_remainder
   <chr>      <chr>     <chr>   <chr>  <lgl>              <int> <chr>
 1 Peter      ""        " Spuh~ Peter~ TRUE                   3 ""
 2 G.         " M."     " Rao"  G. M.~ TRUE                   3 ""
 3 Lorenzo    ""        " Mend~ Loren~ TRUE                   3 ""
 4 Vadim      ""        " Yaku~ Vadim~ TRUE                   3 ""
 5 Kutayba    ""        " Algh~ Kutay~ TRUE                   3 ""
 6 Minoru     ""        " Mori" Minor~ TRUE                   3 ""
 7 Li         ""        " Guan~ Li Gu~ TRUE                   3 ""
 8 Baokun     ""        " Bai"  Baoku~ TRUE                   3 ""
 9 Luiza      " Helena" " Traj~ Luiza~ TRUE                   3 ""
10 Han        ""        " Xiao" Han X~ TRUE                   3 ""
```

This has turned out immensely difficult. Not worth pondering on without a good command of regular expressions. Probably we have to wait until this function matures and is more extensively documented.

> [!N OTE] Unclear usage of regular expressions So far, I could not find out what worked in the regexes and what didn't. For instance, intervals of occurrences did not work. It was also unclear, whether the patterns must be mutually exclusive or whether they have to mention the groups before, and whether or not it should be done directly or with look_ahead/behind expressions (where the behind would not work because intervals would not work). Choices did not work `(x|y)`. Finally, I have never seen before that groups should be marked with `(?: )` rather than just `()`.

14 Pivoting

- longer table to wider and back

Pivoting takes some time and practice. You may need it especially when you have categorical variables. Categorical variables naturally group observations according to their values, and you need different groupings for different purposes. Still unclear? We need use cases.

15 gapminder

How would you

```
gapminder <- gapminder

gapminder %>% filter(country %in% c("Germany", "Czech Republic", "Poland"))
 ↪  %>%
  slice_max(by = year, order_by = year, n = 6, with_ties = TRUE)
```

```
# A tibble: 36 x 6
   country        continent  year lifeExp      pop gdpPercap
   <fct>          <fct>     <int>   <dbl>    <int>     <dbl>
 1 Czech Republic Europe     1952    66.9  9125183     6876.
 2 Germany        Europe     1952    67.5 69145952     7144.
 3 Poland         Europe     1952    61.3 25730551     4029.
 4 Czech Republic Europe     1957    69.0  9513758     8256.
 5 Germany        Europe     1957    69.1 71019069    10188.
 6 Poland         Europe     1957    65.8 28235346     4734.
```

```
 7 Czech Republic Europe     1962    69.9  9620282    10137.
 8 Germany        Europe     1962    70.3 73739117    12902.
 9 Poland         Europe     1962    67.6 30329617     5339.
10 Czech Republic Europe     1967    70.4  9835109    11399.
# i 26 more rows
```

How would you compute correlation of `gdpPercap` between CZ and DE and the other two pairs (on all years)? You need a vector of values for each country. This would correspond to each country having its own column, if you want to extract the values directly from the data frame. Otherwise you would have to filter and pull the filtered values as vectors.

16 `tidyr::pivot_wider`

```
gap_czger_gdp_wide <- gap_cze_ger_gdp %>%
  pivot_wider(names_from = country, values_from = gdpPercap)
gap_czger_gdp_wide %>% slice(1:3) %>% kableExtra::kable()
```

year	Czech Republic	Germany
1952	6876.140	7144.114
1957	8256.344	10187.827
1962	10136.867	12902.463

```
gap_czger_gdp_wide %$%
  cor(x = `Czech Republic`, y = Germany, method = "pearson")
```

```
[1] 0.946459
```

Imagine you want to compute a correlation of the temporal development between two countries, considering an indicator, such as GDP per capita. To compute the correlation, you must have the data in two separate variables. This is how you would prepare `gapminder` to correlate GDP per capita 1952 - 2007 between Czechia and Germany.

Format changes in data frames: wider = fewer rows and more columns, longer = (sometimes) fewer columns but definitely more rows, both compared to the state before the manipulation.

17 tidyr::pivot_longer

```
gap_czger_gdp_wide %>% slice(1:10) %>%
  pivot_longer(cols = c(`Czech Republic`, Germany), names_to = "COUNTRY",
  ↪  values_to = "GDPperCap")
```

```
# A tibble: 20 x 3
    year COUNTRY        GDPperCap
   <int> <chr>              <dbl>
 1  1952 Czech Republic     6876.
 2  1952 Germany            7144.
 3  1957 Czech Republic     8256.
 4  1957 Germany           10188.
 5  1962 Czech Republic    10137.
 6  1962 Germany           12902.
 7  1967 Czech Republic    11399.
 8  1967 Germany           14746.
 9  1972 Czech Republic    13108.
10  1972 Germany           18016.
11  1977 Czech Republic    14800.
12  1977 Germany           20513.
13  1982 Czech Republic    15377.
14  1982 Germany           22032.
15  1987 Czech Republic    16310.
16  1987 Germany           24639.
17  1992 Czech Republic    14297.
18  1992 Germany           26505.
19  1997 Czech Republic    16049.
20  1997 Germany           27789.
```

… but you definitely prefer the GDP in one column and countries in the other column when you want to plot the development comparison. Wider tables are often considered to be more human-readable, and therefore you typically get them from sources that were primarily designed for print.

18 Separating into rows I

```
# A tibble: 6 x 6
  person               time countries industry       income_groups world_6region
```

```
  <chr>               <dbl> <chr>     <chr>          <chr>         <chr>
1 a_jayson_adair       2021 usa       Automotive     high_income   america
2 a_jayson_adair       2022 usa       Automotive     high_income   america
3 a_jerrold_perenchio  2002 usa       Media & Enter~ high_income   america
4 a_jerrold_perenchio  2003 usa       Media & Enter~ high_income   america
5 a_jerrold_perenchio  2004 usa       Media & Enter~ high_income   america
6 a_jerrold_perenchio  2005 usa       Media & Enter~ high_income   america
```

And now a bonus batch of separating: you can as well separate into rows, by which you also produce a longer data frame. Let's talk about a use case in the next slide.

19 Separating into rows II

```
industry_terms <- billionaires_df %>%
  select(c(person, time, countries, industry, income_groups, world_6region))
 ↪  %>%
  separate_longer_delim(cols = c("industry"), delim = ";")
set.seed(33)
industry_terms %>% slice_sample(n = 10)
```

```
# A tibble: 10 x 6
   person          time countries industry           income_groups world_6region
   <chr>          <dbl> <chr>     <chr>              <chr>         <chr>
 1 yang_huiyan     2022 chn       "Real Estate, Edu~ upper_middle~ east_asia_pa~
 2 horst_wortmann  2016 deu       "Apparel & Textil~ high_income   europe_centr~
 3 wang_wenxue     2018 chn       "Real Estate"      upper_middle~ east_asia_pa~
 4 b_wayne_hughes  2005 usa       "Service"          high_income   america
 5 daniela_herz    2013 deu       "Investments"      high_income   europe_centr~
 6 edward_lampert  2017 usa       " Finance"         high_income   america
 7 xu_jingren      2020 chn       "Healthcare"       upper_middle~ east_asia_pa~
 8 qiu_guanghe     2021 chn       "Apparel"          upper_middle~ east_asia_pa~
 9 elon_musk       2019 usa       "Automotive"       high_income   america
10 ennio_doris     2009 ita       " Finance"         high_income   europe_centr~
```

20 continuation

```
industry_terms %<>%
  separate_longer_delim(cols = c("industry"), delim = "&") %>%
  separate_longer_delim(cols = c("industry"), delim = "and") %>%
  separate_longer_delim(cols = c("industry"), delim = ",")
set.seed(10)
industry_terms %>% slice_sample(n = 10)
```

```
# A tibble: 10 x 6
   person                  time countries industry   income_groups world_6region
   <chr>                  <dbl> <chr>     <chr>      <chr>         <chr>
 1 christy_walton          2013 usa       "Fashion " high_income   america
 2 ty_warner               2004 usa       "Hospital~ high_income   america
 3 mustafa_rahmi_koc       2018 tur       " Gas"     upper_middle~ europe_centr~
 4 oleg_deripaska          2008 rus       "Metals "  upper_middle~ europe_centr~
 5 doris_fisher            2021 usa       " Fashion~ high_income   america
 6 gerald_ford             2017 usa       "Finance " high_income   america
 7 gregorio_perez_companc  2005 arg       " Food"    upper_middle~ america
 8 robert_bass             2012 usa       "Energy"   high_income   america
 9 norma_lerner            2018 usa       "Financia~ high_income   america
10 yuri_kovalchuk          2018 rus       "Finance " upper_middle~ europe_centr~
```

21 TF*IDF: domain how typical of person?

```
industry_tf_idf <- industry_terms %>% filter(nchar(industry) > 1) %>%
  group_by(industry, countries, world_6region) %>%
  count(name = "freq") %>% ungroup() %>%
  tidytext::bind_tf_idf(term = industry,
                        document = countries,
                        n = freq) %>%
  group_by(countries, world_6region) %>%
  slice_max(order_by = tf_idf, n = 3) %>% ungroup()
```

We add a column with a statistic that is called *Term Document Frequency times Inverted Document Frequency* (TF*IDF). It originates from Information Retrieval, where you aim at finding the best matching text about a topic (web search engines do that). The higher the statistic is for a combination of term and document, the better the given term distinguishes the given document from others. Here each country is a "document" and each industry has a term. And we ask for instance how well Metals characterize Russia? If each country had just one billionaire and many other countries had to do with Metals, the value would be low (=

in how many other documents this term occurred at least once? Inverted document frequency would be the lower, the more billionaires would be associated with Metals).

But many countries are associated with many billionaires, and each is associated with some industries. So, if most billionaries in Russia have the association with Metals, Metals would still characterize Russia better than the others.

TF*IDF is always a trade-off between how frequent a term is in a document and how many other documents in the collection contain that term too.

22 Social network analysis

- entities and relations
- two types of entities: bipartite network
  - countries are connected through terms
  - terms are connected through countries

You hope to visualize a network where you can see clusters of countries characterized by billionaires in similar industries.

23 Nodes and edges

```
library(tidygraph)
```

```
Attaching package: 'tidygraph'
```

```
The following object is masked from 'package:stats':

    filter
```

```
library(ggraph)

# Prepare edge list
edges <- industry_tf_idf %>%
  select(countries, industry, tf_idf)

# Create node list
nodes <- tibble(name = unique(c(edges$countries, edges$industry))) %>%
```

```
    mutate(type = if_else(name %in% edges$countries, "country", "industry"))
```

24 Graph object

```
# Create graph object
graph <- tbl_graph(nodes = nodes, edges = edges, directed = FALSE)
```

You need to create a specific R object called graph object. It looks like a list. Use the `tidygraph` library. You need two data frames, one with nodes and one with edges. You can explore the structure of `graph`. It is a list with each node being one element. The nodes are of two types: country and industry.

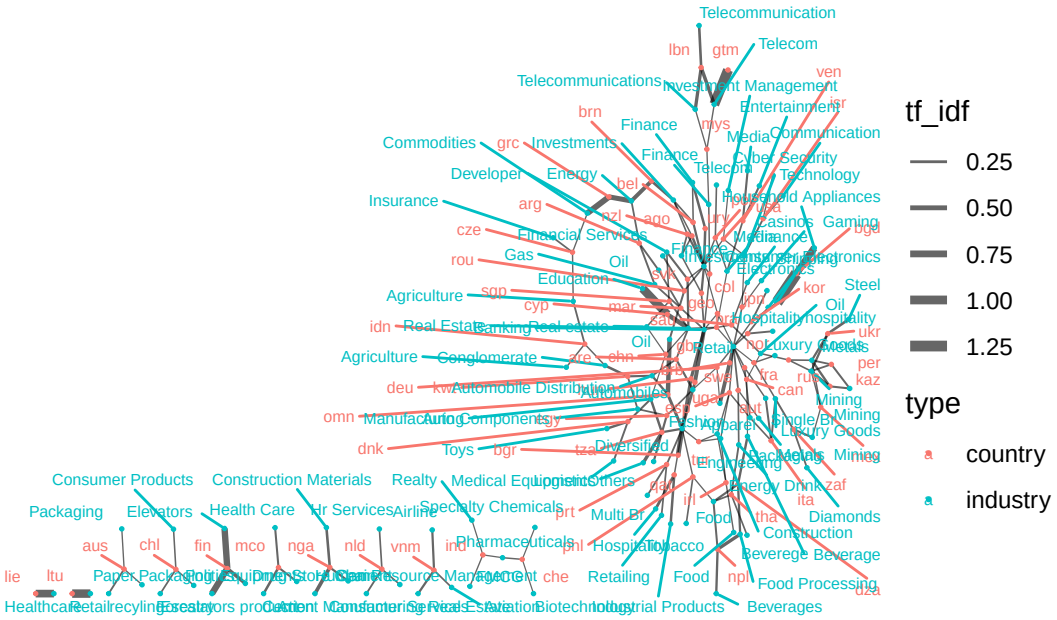
25 Plot the graph object with

```
# Plot with ggraph
p <- ggraph(graph, layout = "auto") +
  geom_edge_link(aes(edge_width = tf_idf), alpha = 0.6) +
  geom_node_point(aes(color = type), size = 0.3) +
  geom_node_text(aes(label = name, color = type), repel = TRUE, size = 2,
 ↪  max.overlaps = 100) +
  scale_edge_width(range = c(0.2, 2)) +
  theme_void() +
  labs(title = "Bipartite Network: Countries and Industries")
```

```
Using "stress" as default layout
```

```
p
```

Bipartite Network: Countries and Industries




```
ggsave(plot = p, filename =
 ↪  "~/NPFL112_2025_ZS/OUTPUT_FILES/billionairs_industry_country_ggraph.pdf",
 ↪  width = 7 * 2, height = 7 * 2.2, limitsize = FALSE)
```

```
Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'Paper Packaging ' in 'mbcsToSbcs': for   (U+3001)
```

26 Plot as image file

Bipartite Network: Countries and Industries