Barcelona Summer School of Demography

Module 1. Introduction to R

2. Tidy Pipelines

Tim Riffe
tim.riffe@ehu.eus

5 July 2022

Contents

1 Summary

In the first session we saw an intro to visualizing data that is tidy. Today we'll see an approach for turning messy data into tidy data: **data wrangling**. The exercises we'll do today are designed to expose you to some diversity in the functions needed to data wrangle. We'll see that complex processing chains (or pipelines) are composed of small and intuitive steps.

2 Data processing verbalized

2.1 case example: making a time use plot

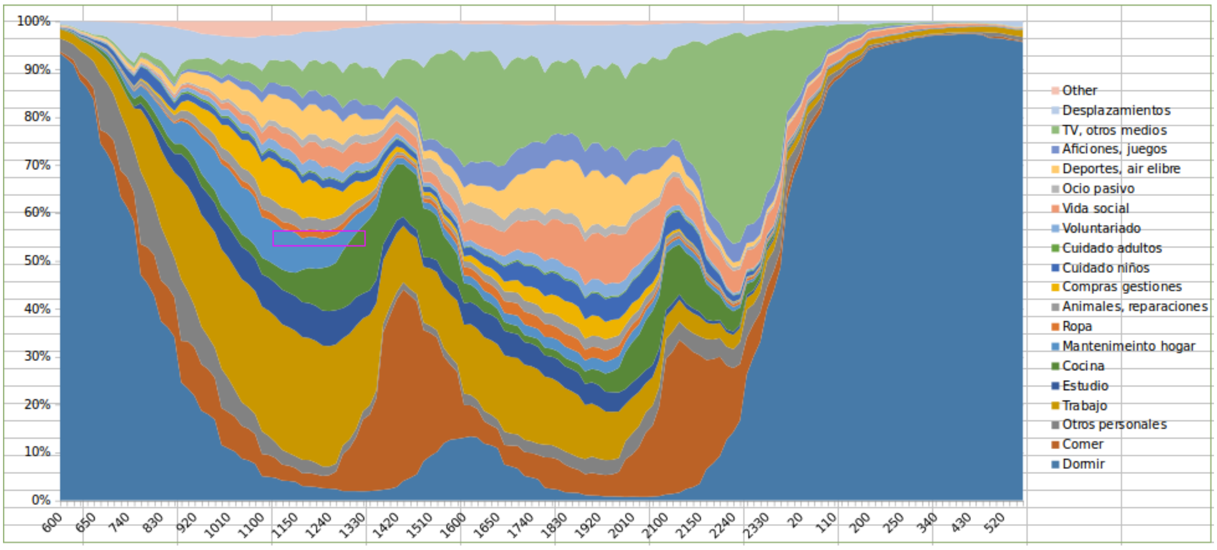
We know we'll need tidyverse functions today, so let's just get this loaded:

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --

## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble  3.1.7     v dplyr   1.0.9
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

So a friend Joan G. asked me how to make this plot in `R` some time ago:



It looks like we have time of day in x, population percentile in y, and filled areas indicate what fraction of people are doing different activities. The table underlying this was derived from survey microdata.

We can't (err, shouldn't) really go this high with qualitative colors very easy, and we may or may not avoid this issue. I've asked him for the raw data that came before the final table that went into this plot so that we can see those steps too in `R` (Thanks JG!).

The end result of getting this data wrangled will appear complex at a glance. But we won't see it until the end, and we'll see that the sequence of commands is human-readable and hopefully intuitive. We will build it this sequence incrementally together, out of relatively simple steps, and it features many of the `dplyr` heroes:

1. `pivot_longer()` make a wide range of columns tidy
2. `mutate()` make new columns using other columns, no loss of rows
3. `select()` selects columns
4. `filter()` selects rows (subsets)
5. `group_by()` allows operations on subgroups
6. `ungroup()` removes the former

7. `summarize()` aggregates over rows. Usually reduces nr of rows

and some useful utility functions that do specific things:

7. `separate()` splits a column into two
8. `remove_na()` replaces `NA`s with some value
9. `n()` counts cases, usually in grouped data

There are a couple seemingly silly but important things to note about data processing pipelines: i) most of the operations are verbs, ii) it's all a single flow, strung together with `%>%` which you can read as "and then do this". You can find a nice overview of `dplyr` functions here: https://raw.githubusercontent.com/rstudio/cheatsheets/main/data-transformation.pdf

2.1.1 download and load the time use data

First let's read in Joan's data. You can download the file from here: https://drive.google.com/open?id=1AibPQP8plaGv0MaRwOJqqOg2LR6wXqK0. Create a folder inside your `R` project for this module called `Data`, and save the file in there.

The file is in an SPSS format, so we'll use the `haven` package, which knows what to do with it. Let's take a look at this data.

```
library(haven)
# path assumes you put the file in a Data folder in your project!
JG <- read_spss("Data/caseid_aggr.sav")
```

```
# str(JG)
head(JG[,1:12])
```

```
## # A tibble: 6 x 12
##     CASEID   AGE         SEX          RACE   MARST     EDUC EMPSTAT SPOUSEPRES SPAGE
##      <dbl> <dbl>   <dbl+lbl>     <dbl+lbl> <dbl+l> <dbl+lb> <dbl+l>  <dbl+lbl> <dbl>
## 1 2.00e13    68 2 [Female] 100 [White~ 1 [Mar~ 21 [Hig~ 5 [Not~ 1 [Spouse~    71
## 2 2.00e13    79 2 [Female] 100 [White~ 3 [Wid~ 21 [Hig~ 5 [Not~ 3 [No spo~   999
## 3 2.00e13    67 1 [Male]   100 [White~ 1 [Mar~ 21 [Hig~ 5 [Not~ 1 [Spouse~    69
## 4 2.00e13    80 2 [Female] 201 [White~ 3 [Wid~ 10 [Les~ 5 [Not~ 3 [No spo~   999
## 5 2.00e13    70 2 [Female] 100 [White~ 4 [Div~ 21 [Hig~ 1 [Emp~ 3 [No spo~   999
## 6 2.00e13    67 2 [Female] 100 [White~ 3 [Wid~ 21 [Hig~ 5 [Not~ 3 [No spo~   999
## # ... with 3 more variables: SPSEX <dbl+lbl>, SPEDUC <dbl+lbl>,
## #   SPEMPNOT <dbl+lbl>
```

```
dim(JG)
```

```
## [1] 8395 1455
```

2.1.2 wrangling the time use data

The columns are similar to factors: but really they mimic the value-label style of SPSS. The actual values are just integers, and each integer stands for an activity. When we reshape to a tidy style we'll lose that information, just for the sake of being able to label activities in the plot later, we save them in an object now.

```
activities <- attr(JG$time_1000_max,"labels")
```

Here's the full pipeline:

```
JG <- JG %>%
    sample_n(1000) %>%         # reduce size for testing purposes
    pivot_longer(
      time_1_max:time_1440_max, # column range to stack
      names_to = "Time",         # new column to collect header names
        values_to = "activity"   # new column for values collected
        ) %>%
    separate(col = Time,
             into = c(NA,"time",NA),
             sep ="_",
             remove = TRUE,
             convert = TRUE) %>%
    select(CASEID, time, activity) %>% # only columns we need
    filter(time %% 10 == 1) %>%     # cut rows to every 10th
    group_by(time, activity) %>%
    summarise(n = n(),
              .groups = "drop") %>%
  group_by(time) %>%
    mutate(freq = n / sum(n, na.rm = TRUE)) %>%
    ungroup() %>%
    mutate(activity = factor(
                        activity,
                              levels = activities,
                              labels = names(activities)),
                 freq = if_else(is.na(freq), 0, freq))

     # tip for largish data:
gc() # free up memory no longer needed
```

```
##           used (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells 1137770 60.8   10758084 574.6 13447604 718.2
## Vcells 2020168 15.5   54133704 413.1 67667128 516.3
```

Let's call this a pipeline, because `JG` gets passed *through* a series of functions in the order presented. The best way to understand a pipeline like the above is to interactively see how the data gets modified at each step, so it's a good idea to do so incrementally in your `session.Rmd` file. But here's an overview of the steps in there:

1. `sample_n(1000)` The data is big and push the limits of my measly 4Gb of RAM, so this line samples 1000 rows from the data (about 1/8).
2. `pivot_longer()` This is the key step for going from wide to *tidy* (long). 1440 columns are collected. The `names_to` argument is where we stash the header names as values in a new column called `"Time"`. `value_to = "activity"` says that the contents of all the columns being gathered will be put into a new variable called `"activity"`. The column range `time_1_max:time_1440_max` says to collect columns starting from `time_1_max` all the way to `time_1440_max`, so 1440 columns with 1000 elements each will now become two columns with 1440000 elements each.
3. `separate()` is used to extract more useful information from the column `Time`: we just want the integer in the middle. `-col = Time` specifies which column to split `-into = c(NA,"time",NA)` says what the new columns will be called (`NA` placeholders are used to

discard pieces) `-sep = "_"` says to split the column on the underscores. So, in a first pass, `"time_1_max"` is turned into `"time"`, `"1"`, and `"max"`, but the `NA` placeholders throw out the first and last bits. `-"remove = TRUE"` says to delete the `"Time"` column when we're done. `-"convert = TRUE"` says to coerce newly created columns (`"time"` in our case) to numeric form. In our case `"1"` goes from character to integer, `1`. And the values of `"time"` will be integers 1 to 1440.

4. `select()` picks out just the columns we want to preserve for the rest of the pipeline.
5. `filter()` picks out just our sample of every 10th minute. This is done with the logical `time %% 10 == 1`. `%%` is the modulo operator, which tells us the remainder after division an integer number of times. Example:

```r
0:20 %% 5
```

```
##  [1] 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 0
```

```r
# which ones are evenly divisible by 5?
0:20 %% 5 == 0
```

```
##  [1]  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE
## [13] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE
```

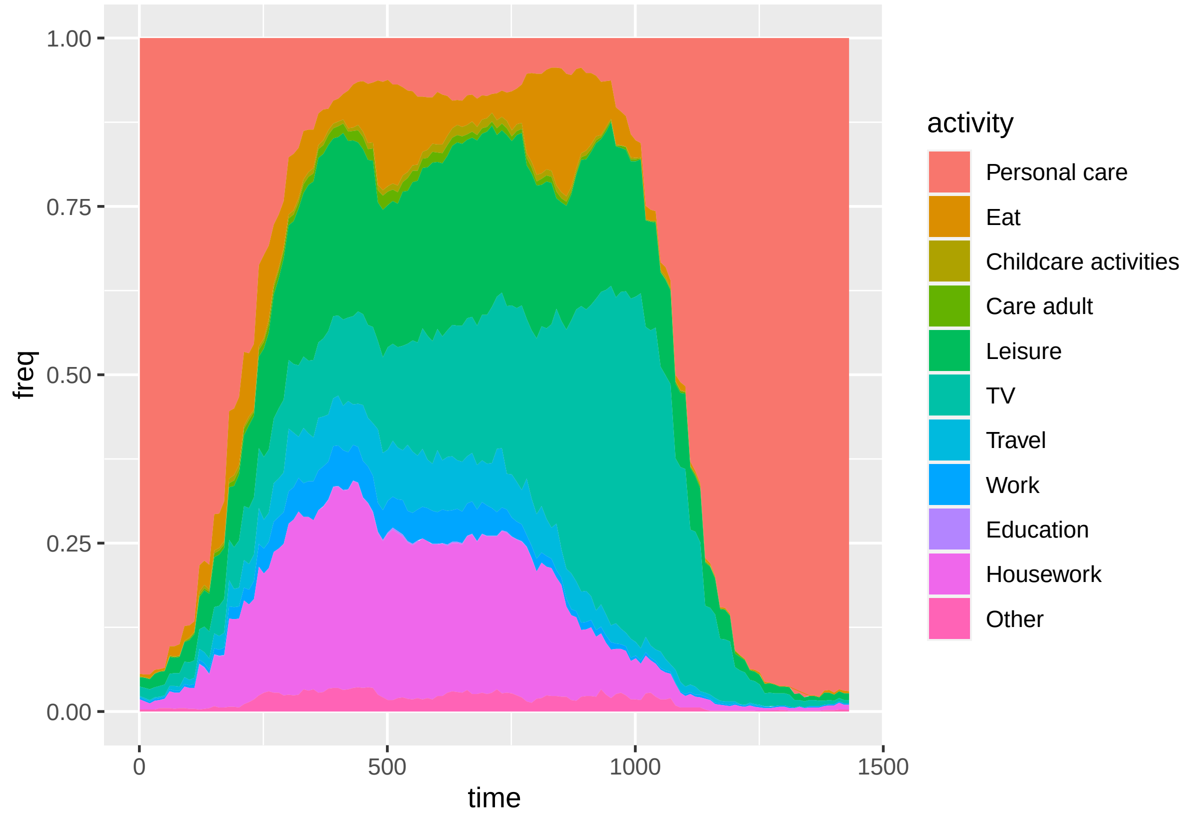
6. `group_by()` defines independent subsets of the data. Code executed within these groups is repeated for each subset.
7. `summarize()` can be used to reduce rows in the data based on some summary stats. Here we tabulate rows. Steps 6 and 7 could be combined into one with `tally(time, activity, name = "n")` with no loss.
8. `group_by() %>% mutate()` creates a new column for relative frequency (fraction or proportion) within independent subsets defined by `group_by()`.
9. Finally, in another `mutate()` we turn `activity` into a factor variable with explicit levels, and we replace `NA` values in `freq` with 0s.

2.1.3 plotting the time use data

Now we have the final dataset that we want to plot: it literally contains the fraction of time (`freq`) spent doing different activities (`activity`) in 10 minute steps through the day (`time`). Nothing less and nothing more.

In plotting this we will introduce a new `geom`: `geom_area()` to make our filled areas. Here's a first rough stab at it. You see we have the basic structure, and everything from here on out is going to be plot detail management.

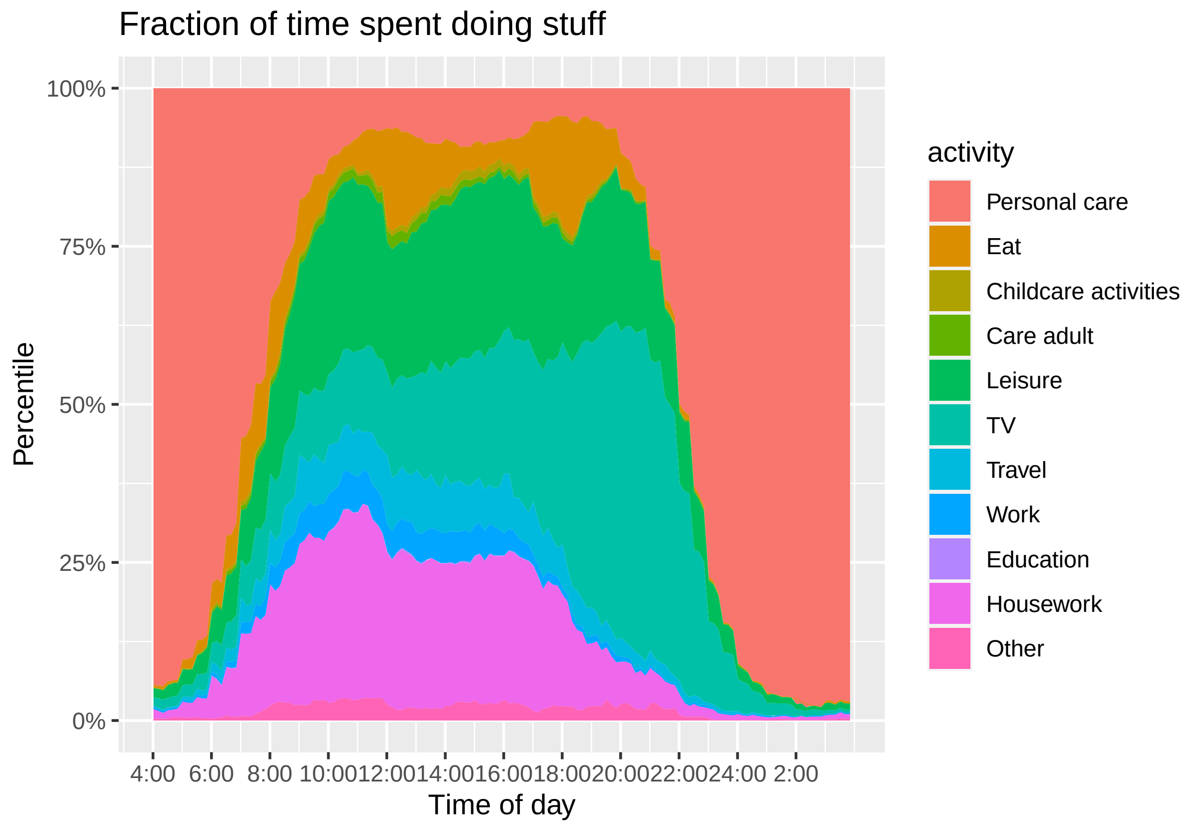
```r
ggplot(JG, mapping = aes(x = time, y = freq, fill = activity)) +
    geom_area()
```



For the sake of a semi complete solution, here's a way to get more meaningful x and y breaks and labels.

```
# Figure out time labels...
# we have minutes since 4:00AM,
# so hour = (time - 1) / 60
mins    <- seq(from = 0, to = 1440, by = 10)
hrs     <- seq(from = 0, to = 1430, by = 120)
hrslabs <- paste(c(seq(from = 4, to = 24, by = 2), 2), "00", sep = ":")

ggplot(JG, mapping = aes(x = time,
                         y = freq,
                         fill = activity)) +
    geom_area() +
    scale_y_continuous(labels = scales::percent_format(accuracy = 1)) +
    scale_x_continuous(
               breaks = hrs,
                       labels = hrslabs) +
    labs(x = "Time of day",
         y = "Percentile",
         title = "Fraction of time spent doing stuff")
```



Joan G. told me that time is expressed in minutes since 4:00 AM, so that's my reference point. I'd like a label every two hours, so 120 minutes `seq(0, 1440, by = 120)` gives me a vector of hour breaks expressed in minutes. Now I produce the labels using similar tricks, `paste()` concatenates things together with an optional separator (`:`) in our case. We just need to remember that we have 24 hours of data, so at the end we add a break for 2:00 AM. These are then set using `scale_x_continuous()` (because `time` is continuous here). To get percentile labels on the y axis I used a function from the `scales` package.

There are still several things we'd do to this plot to make it publication-worthy, but let's save some of it for other plots, or for Ilya's module.

3 Worked example number 2: fertility variation by level

I wonder how different fertility patterns can be within a given level of TFR? Note: TFR stands for total fertility rate, and it is defined as the sum of age-specific fertility rates. It is often understood as the average number of children a woman will have over the span of reproductive ages. One could of course talk lots about how this does and doesn't work well as an indicator, but you and me, we're going to make this a coding exercise! The basic idea is that fertility rates have some age-pattern. Mean ages of fertility can vary somewhat however, as can distribution symmetry vs skewness. What might be the extremes within a given TFR level? Until we look, it's sort of hard to know. Maybe there's a story here.

3.0.1 Download the data

Let's download a file from the Human Fertility Collection https://www.fertilitydata.org/cgi-bin/index.php. I'll navigate there in the browser, but this is the link straight to the file we want:

https://www.fertilitydata.org/data/HFC/HFC_ASFRstand_TOT.zip. Stick it in your Data folder we made earlier.

it's somewhat tidy already, but we'll still need to do things to it to.

```
HFC <- read_csv("Data/HFC_ASFRstand_TOT.zip",
                na = ".")
```

```
## Rows: 674621 Columns: 17
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (5): Country, AgeDef, Collection, SourceType, RefCode
## dbl (9): Year1, Year2, Age, AgeInt, Vitality, ASFR, CPFR, Note, Split
## lgl (3): Region, Urban, Origin
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
# str(HFC)
# HFC$Country %>% unique()
```

This is a rather large and underexploited data source. We have age-specific fertility rates for many countries and years. There's one inconsistency in the age definition that probably only demographers care about, can explain if you want. Also, sometimes the same year and place has more than one estimate, so we'll want to be careful picking out all the potential grouping variables if using group_by().

Let's use the HFC dataset to plot some different things.

4 Exercises

Exercise 1.1:

How about we get a glimpse of the full variety of fertility curves in there? Let's make a single line graph with all ASFR patterns plotted. To do this, you'll need to identify subsets, which is easy to trip on with this dataset, so here's a tip for that:

```
HFC <-
  HFC %>%
  mutate(YrInt = Year2 - Year1 + 1) %>%
  group_by(Country,
                   Year1,
                   Region,
                   Urban,
                   Origin,
                   AgeDef,
                   Vitality,
                   Collection,
                   SourceType,
                   RefCode,
                   YrInt) %>%
  mutate(sub_id = cur_group_id()) %>%
  ungroup()
```

Now `sub_id` can be your grouping variable. Give it a try:

```
# ggplot(HFC ...) +
#   geom_ ...
```

This is just an example to show you the full set of variety in the data that we'll want to distill somehow. It will be tricky to get this plot to work due to overplotting. Try reducing `alpha` to something small.

Exercise 1.2: Plot the time series of the total fertility rate (TFR) for 5 or so selected countries. You choose which (note they're not all there). If you're not sure about country codes used, you can check here: https://www.fertilitydata.org/cgi-bin/country_codes.php