Tutorial 3: Data cleaning & management

Carlo Knotz

Table of contents

Introduction	2
Setup	4
Trimming your dataset	5
select()	6
Using select()	6
Saving the result	6
Removing (de-selecting) variables	7
filter()	8
Connecting operations with the "pipe" (%>% or >)	9
Storing the result of your "pipeline"	10
The new pipe operator ($ > $)	11
Creating new variables with mutate()	12
Simple transformations with numeric variables	12
Advanced mutate(): "Dummy-coding" variables	13
Dummy-coding a numeric variable	13
Dummy-coding an ordinal or categorical variable	15
Calculating summary statistics with summarize()	18
Simple summary statistics	18
Combining summarize() & group_by()	19
Other statistics & multiple statistics in one operation	20
Other summary statistics	20
Multiple summary statistics	20
Variable types and transformations	22
Identifying variable types	22

Summing up & next steps						
More tools for working with factors	30					
Factor to character	29					
Factor to numeric	25					
Data cleaning and transformations with factors	25					
Changing storage types	23					

Introduction

In the previous tutorial, you learned how you can import a dataset and do some initial exploratory data analysis (EDA) to get familiar with it.

In a real data analysis project, these would of course only be the very first steps. Usually, you will discover some smaller and larger issues with your data during the initial EDA, for example that you have to trim and clean the dataset, and that you may need to recode some variables or create new ones. In more extreme cases, your dataset can be disorganized or contain irrelevant information, or the variables in them can be stored in the wrong way.

This process of cleaning and organizing a dataset, and of creating new variables is called *data* management (or 'munging', 'wrangling', or 'data manipulation'). It is the process of turning a raw dataset, which usually contains irrelevant observations or variables and where some variables need to be transformed or newly constructed, into the neat and tidy dataset you use in your statistical analysis.

Let's be honest for a moment: Data cleaning is often not very entertaining, and is generally the thing that is most difficult to get through when you are just starting and you do not yet have a good intuition or "muscle memory" for working with data. In that case, data cleaning can be a major hurdle and source of frustration.

Fortunately, things have improved a lot with the arrival of the tidyverse (see https://www.tidyverse.org/). As mentioned in the first tutorial, the tidyverse collection includes several packages that make data cleaning, and even otherwise difficult operations much easier and quicker.

Still: Expect to be struggling with these things at the beginning, and that this tutorial will likely be the one that feels most dull and confusing. Just hang in there, and ask if you really need help!

In this tutorial, you will first learn how to do basic data cleaning and preparation tasks with functions from the tidyverse (sections 4-6). In section 7, you will learn how to change how a particular variable is stored (e.g., from numeric to character). Here, you will use functions from 'base R'.

As in the previous tutorial, you will first practice all these operations with the small ess dataset from the bst290 package. Later, when doing the exercises, you will apply what you have learned to real-life data from the ESS.



Hvis du ønsker å lese en norsk tekst **i tillegg**: "Lær deg R", Kapittel 6. OBS: Boken bruker en annen 'dialekt' (base R) enn den vi bruker her (tidyverse).

Setup

You need to use two packages, the bst290 package and now also the tidyverse package (which you should have installed at the beginning of the course). The first step is therefore to load the two packages and then the practice dataset:

The start of your script file should look like this:

```
# Loads packages
library(bst290)
library(tidyverse)

# Loads dataset
data(ess)
```

Trimming your dataset

A typical research dataset (e.g., data from the *ESS*) will usually contain many more variables than you need for your analysis.¹ Therefore, it is very useful to know how to get rid of variables that are irrelevant for what you want to do.

You may also remember that the full ESS dataset included more than 40.000 observations for more than 20 countries. It can happen that you need to work with all of these observations, but there are also many times when you only need data for a single country (such as in this course!). In that case, you need to exclude the irrelevant observations.

Fortunately, dealing with these two problems is easy. The tidyverse (or, more precisely, the dplyr package in the tidyverse) contains two functions that make this easy:

• select(): To (de-)select variables

• filter(): To filter observations in or out

¹Think back to last week: Maybe you remember that the full ESS dataset included more than 600 variables?

select()

The small ess practice dataset includes 22 variables. Let's assume that you really need only the ID number (idno), age (agea), and gender (gndr) variables for an analysis. You therefore want to reduce your dataset to these three variables and get rid of all the others.² The select() function allows you to do that.

Using select()

Here is how you keep specific variables with select():

```
select(.data = ess, idno, agea, gndr)
```

Here you tell R:

- 1. That you want to select variables with the select() function;
- 2. That you want to select from the ess dataset (with .data = ess). It is important that you do not forget to add a dot before data;³
- 3. Then you simply list the variables you want to select, separated by commas;

Saving the result

If you only run the function, R will do the operation and then simply print the result out for you — and the result is then the "trimmed" dataset. This can be helpful if you just want to test if your code works, but you usually want to store the reduced dataset so you can use it in your analysis.

You can save the resulting "trimmed" dataset as a new object using the good old assignment operator:

```
ess_selected <- select(.data = ess, idno, agea, gndr)</pre>
```

²The idno variable does not really contain substantive information about respondents, but it is good practice to keep this variable because it can come in handy later on — for example, in case you want to add other variables.

³"Why?", you may ask? This is how the function was designed by its author, Hadley Wickham, and the detailed answer for why he did this is quite technical, see: https://design.tidyverse.org/dots-prefix.html. Not to worry, you will get around this just a few pages from here!

Removing (de-selecting) variables

Now you know how you can *keep* certain variables in a dataset and get rid of all others. But you can also use select() to *remove* specific variables but leave the rest of the dataset as it is. To do this, you simply add a minus symbol (-) before the variables you want to get rid off.

For example, to remove the agea and gndr variables — and keep all the others — you would run:

```
select(.data = ess,-agea,-gndr)
```

filter()

Filtering observations works basically the same way, the only difference is that you have to specify *how* or by which criteria you want to select observations from the data.

For example, let's assume you wanted to remove all those observations from the ess dataset where respondents were younger than 40 years. This is how you would do this with filter():

```
filter(.data = ess,agea>=40)
```

In human language:

- 1. You tell R that you want to filter observations from the ess dataset
- 2. You specify a *condition* using mathematical symbols (>=): Keep all those observations where the respondent's age is equal to 40 or greater (agea>=40)

The expression >= stands, as you probably know, for "greater than or equal to". It is one of several you can use to filter your data:

- \bullet > "greater"
- < "smaller"
- <= "smaller or equal to"
- >= "greater or equal to"
- == "must be equal to" (the double equal sign means we are extra sure here)
- != "must not be equal to" (generally, ! stands for "is not")

• %in% "is included in", usually followed by a vector (e.g., cntry %in% c("Norway", "Sweden", "Denmark"))

You can also specify *multiple conditions* in filter(). For example, to limit the data to women who are older than 35 you would do the following:

```
filter(.data = ess, agea>35 & gndr=="Female")
```

In human language:

- 1. You want respondents older than 35 (agea>35)
- 2. You want only women (gndr=="Female")
- 3. You make clear that both conditions have to be fulfilled at the same time with the & ("ampersand") symbol.

You can *save* the result with the assignment operator (<-) as shown on the previous page with select().

⁴This is a bit more advanced, but very useful in practice!

Connecting operations with the "pipe" (%>% or |>)

So far you were doing one operation at a time: First selecting variables, then filtering observations, then dropping NAs.

There is also a more efficient way of doing data cleaning with the tidyverse: You can *connect* different operations together using the "pipe" operator: %>%.

Simply put: The pipe tells R that it should take the result of one operation and then directly "feed" it into a following one — or, in other words, it "docks" two operations together. With this operator, you can build entire "data management pipelines" where you take your starting dataset, put it through a sequence of data cleaning and management operations, and get a properly cleaned and prepped dataset out at the end.

How this works will become clearer when you see the pipe in action. For example, let's say you wanted to work with the ess practice dataset, but you first wanted to trim it down to only a few variables you really need and keep only female respondents who are older than 35.

Here is how your code would look like:

```
ess %>% # 1.
select(idno, agea, gndr) %>% # 2.
filter(gndr=="Female" & agea>35) # 3.
```

In human language: We tell R that it should...

- 1. ...take the ess dataset...
- 2. ...select the variables idno, agea, and gndr from it...
- 3. ...and then filter the data so that only women (gndr=="Female") who are older than 35 (agea>35) are kept.

Of course, this could be continued even further.

Storing the result of your "pipeline"

As before, you can save the result of your "data cleaning pipeline" as a new dataset in your *Environment*. This is useful when you want to prepare a cleaned and trimmed version of the original "raw" dataset that you can then use in your statistical analysis.

Alternative 1 is to use the standard assignment operator (<-):

```
ess_clean <- ess %>% # saves the "trimmed" dataset as 'ess_clean'
select(idno, agea, gndr) %>%
filter(gndr=="Female" & agea>35)
```

There is also a second alternative, in which you use the "reversed" assignment operator (->):

```
ess %>%
  select(idno, agea, gndr) %>%
  filter(gndr=="Female" & agea>35) -> ess_clean # as above, but now at the end of the pipeli:
```

Finally, and as with the regular assingment operator, it can be a good idea to create a keyboard shortcut for the pipe operator to make typing easier (in RStudio, go to "Options", then "Code", and then "Modify keyboard shortcuts").

The new pipe operator (|>)

The "traditional" pipe operator (%>%) that we used so far is a part of the tidyverse — and it will only work if you have the tidyverse or one of the central packages (dplyr, ggplot2) loaded.

With the release of R version 4.1.0, a new pipe operator was introduced: |>. This one works like the old one, but is "native" to R — you shouldn't need to load packages to be able to use it.

While there are some differences in how they work, you can generally use either of them (or both) without running into any problems or getting wrong results.⁵

Important

The two pipe operators may not work with all functions in R. They should work fine with functions that come from the tidyverse package collection (select(), filter(), drop_na(),...) and they work also with *some* other functions, but this is not always the case. If you notice that your "pipeline" breaks when you add a particular function, then it is best if you just store the result after the last functioning step of your pipeline and then use that result with the "offending" function separately.⁶

⁵See also https://www.tidyverse.org/blog/2023/04/base-vs-magrittr-pipe/.

Creating new variables with mutate()

While your dataset often contains variables or observations you do not need, it is also frequently the case that you have to construct a new variable from one or more of those variables that are in your dataset. In this case, you "mutate" existing variables into a new shape.

This is what the mutate() function is there for. With mutate(), you can transform your variables in (almost?) any way you need to.

Simple transformations with numeric variables

The most basic way to transform a variable is to do a simple mathematical transformation. For example, let's say you wanted to work with the height variable from the ess dataset. This variable records the respondents' body heights in centimeters. When you run summary(ess\$height), you see that this variable ranges from 147cm (the shortest person) to 196cm (the tallest person).

But you decide, for some reason, that you want that variable measured in meters, and not in centimeters. To get there, you have to divide the height variable by 100.

Here is how you can do this with mutate() (plus again the pipe operator):

```
ess %>% # 1.
mutate(height_meters = height/100) # 2.
```

Once more in human language:

- 1. "Take the ess dataset..."
- 2. "...and mutate height into a new variable, height_meters, by dividing height by 100."

Obviously, this is just a very simple example and you can certainly take this further — for example by adding, subtracting, or multiplying two or more variables or doing more complex mathematical transformations. To get an overview over what you can do with mutate(), see the official help page: https://dplyr.tidyverse.org/reference/mutate.html#useful-mutate-functions

Advanced mutate(): "Dummy-coding" variables

Next to mathematical transformations, you also often have to "dichotomize" or "dummy-code" one or more of your variables. "Dummy-coding" means that you turn a more complex variable into a simple yes/no or (1/0) "dummy" variable. To dummy-code a variable, you use the if_else() function within mutate().

You can do this with numeric variables (e.g., age or years spent in education) but also categorical or ordinal variables that have more than two categories (and which are stored as *Factors*). The process differs only a bit between the two scenarios.

Dummy-coding a numeric variable

This first example shows you how you can dummy-code a variable that is numeric. In this case, we use the age-variable agea, which measures the respondents' ages in years, and we dummy-code it into a categorical variable that identifies all those respondents in our practice dataset who are older than 65 years.

In other words, the task is to create a new variable that identifies older respondents. This new variable would have the value 1 whenever a respondent in the dataset is older than 65, and it would have the value 0 for all those respondents who are younger. You can create this variable on the basis of the agea variable with the handy if else() function.

In practice, the code to do this would look like this:

The mutate-call in human language:

- 1. "Take the ess dataset and...
- 2. "...create ("mutate") a new variable called older based on the *condition* that the respondent's age is greater than 65 (agea>65)."
- 3. "If that condition is true, the new variable older gets the value of 1..."
- 4. "...and if that condition is false, older gets the value of 0."

⁷if_else() is a newer version of the similar ifelse() function that is built into R from the start. Both work essentially in the same way, but if_else() is specifically designed for mutate() and also a bit stricter—which helps you avoid errors.

To show you more clearly what a dummy-coded variable looks like and does, here is a cleaned-up result of the operation shown on the previous page:

	idno	agea	older
1	12414	22	0
2	9438	43	0
3	19782	58	0
4	18876	22	0
5	20508	84	1
6	19716	62	0
7	13476	68	1
8	6762	81	1
9	19518	59	0
10	21336	57	0

What you see here is a small part of the ess dataset with the first ten observations and only the idno and agea variables plus the new older variable. You should directly see how the older variable corresponds to the agea variable: Whenever a given respondent's age is greater than 65 years, older has the value of 1; otherwise, older is 0.

If you wanted to use the new variable in your analysis, you would obviously have to store the result with either the regular assignment operator (<-) or the reversed version (->) as shown earlier. Otherwise, R will only print out the entire ess dataset with all existing variables plus the new one (older) that the code creates.

Dummy-coding an ordinal or categorical variable

Often, you want to dummy-code a variable that is not numeric like **agea** but categorical or ordinal, and which is stored as a *Factor* in R. This is of course also possible, but the process is slightly different.

To show you how this works, we will use the health variable that is included in the ess practice dataset. This variable measures how respondents subjectively perceive their own health on an ordinal scale. The categories on that scale are "Very good", "Good", "Fair", "Bad", and "Very bad".

You can also see this when you use the attributes() function:

```
attributes(ess$health)
$levels
[1] "Very good" "Good" "Fair" "Bad" "Very bad"

$class
[1] "factor"
```

Under \$levels, you see the different categories. Under \$class, you see that it is stored as a factor-type variable (as it should be!).

Assume now that we want to create a new variable that is based on health, and the new variable should identify those respondents in our dataset that perceive their own health to be at least "good". In other words, we want to dummy-code the health variable into a new variable that identifies respondents who have a good or very good subjective health.

Here again, we use if_else() within mutate(), but we now need to specify the condition a bit differently:

Translated into "human", the code tells R to:

- 1. "Take the ess dataset and..."
- 2. "...create a new variable called health_dummy based on the *condition* that the health-variable has either the value 'Very good' or the value 'Good'. (*Notice* that we use the %in% operator here to indicate that health should be eiter"Very good" or "Good".)"
- 3. "If that condition is true, the new variable gets the value 'Good health',..."
- 4. "...and if that condition is false, the variable gets the value 'Not good health'.

And, as before, **if you wanted to use the new variable in your analysis**, you would need to store the new version of the dataset with <- or ->. Otherwise, R will only print out the entire dataset with all the variables, old and new.

And just to show you again what the new variable does in this case, here is a cleaned-up version of the result of the code shown on the previous page:

	idno	health		1	health_dummy	
1	12414		${\tt Good}$		${\tt Good}$	${\tt health}$
2	9438	Very	good		${\tt Good}$	${\tt health}$
3	19782		${\tt Good}$		${\tt Good}$	${\tt health}$
4	18876	Very	good		${\tt Good}$	${\tt health}$
5	20508	Very	good		${\tt Good}$	${\tt health}$
6	19716		Fair	Not	good	${\tt health}$
7	13476		Fair	Not	good	${\tt health}$
8	6762		Fair	Not	good	${\tt health}$
9	19518	Very	good		${\tt Good}$	${\tt health}$
10	21336		${\tt Good}$		${\tt Good}$	${\tt health}$

You see that all respondents who felt that their health was either "very good" or "good" got the value "Good health" on the new variable. The three respondents who judged their own health to be only "fair" got the value "Not good health", and this would obviously be the same for all respondents that rated their own health as "bad" or "very bad".

Good to know: When you use if_else(), you can decide what type of variable the new dummy-coded variable will be. If you use numbers (like in the first example where we dummy-coded agea), the new variable will be numeric. If you use text (like in the second example), the new variable will be a character-type variable. You can then transform this variable to a factor with as.factor(), if you like (see also below for details).

Calculating summary statistics with summarize()

A final operation that you will very often need to do is to summarize or aggregate your data. This is often the case when you want to calculate summary statistics, either over the entire dataset or for specific groups of observations.

Simple summary statistics

To start with a simple example, let's say you are interested in the average body height of your respondents. In the ess dataset, this is measured via the height variable.

You know, of course, that you can get the average of height with the mean() function:

```
mean(ess$height,na.rm = T)
[1] 173.7606
```

Here is how you would do it the tidyverse-way:

```
ess %>%
  summarize(mean_height = mean(height, na.rm = TRUE))
  mean_height
1  173.7606
```

What you do here is, in essence, the same as the above: You use the mean() function to calculate the average of the height variable. The only differences are that a) you save the result temporarily into a new variable (mean_height), and b) you do not have to use the \$ sign to tell R where to take the height variable from because you already do that in the very first step (ess %>%).

If you now think that the second option is really just a more complicated and cumbersome form of the first one: True, in principle.

But the second option has the great advantage that it can be extended — for example to calculate summary statistics over the categories of some other variable. How you do this comes on the next page.

Combining summarize() & group_by()

You can use the <code>group_by()</code> function to group your dataset by some other variable before you calculate any summary statistics.

For example, let's say you wanted to calculate the average body height for men and women separately. This is how you would do this with group_by() and summarize():

In human language:

- 1. "Take the ess dataset..."
- 2. "...group the data by gender (gndr)..."
- 3. "...and finally calculate the average height for each of the two groups. Save the result temporarily into a new variable called mean_height."

Other statistics & multiple statistics in one operation

Other summary statistics

The previous examples showed you how to calculate the mean value of a variable — but you can of course also calculate other summary statistics such as the variance, median, sum, or standard deviation.

For example, to calculate the median age across genders you would run:

Similarly, if you wanted to know the number (N) of men and women in the sample you would use the following code:

```
ess %>%
  group_by(gndr) %>%
  summarize(obs = n())
# A tibble: 2 x 2
  gndr   obs
  <fct> <int>
1 Male    75
2 Female    68
```

n() simply calculates the number of observations.

Multiple summary statistics

You can also get multiple summary statistics at the same time. All you need to do is to add to the summarize() call:

```
ess %>%
 group_by(gndr) %>%
 summarize(obs = n(),
           med_age = median(agea, na.rm = T),
           mean_weight = mean(weight, na.rm = T))
# A tibble: 2 x 4
 gndr
          obs med_age mean_weight
  <fct> <int>
               <dbl>
1 Male
                 51
                             86.8
           75
2 Female 68 42.5
                             68.9
```

Last but not least, a very useful way to extend these operations is to *directly visualize* the results in a graph using ggplot2. You will learn how to do this in the next tutorial.

A heads-up: This last part is a bit technical and you have already done quite a lot, so maybe take a quick break before doing this.

Variable types and transformations

You remember from the previous tutorials that there different ways in which variables can be stored in R:

- "Numeric" or num variables: For "pure" numbers such as age;
- "Character" or chr variables: For variables that contain text (e.g., the cntry variable);
- "Factor" variables: How R likes to store categorical or ordinal variables with distinct categories;

(There are also others, but this is a topic for another time.)

Identifying variable types

You also know already how to recognize different types, for example by looking at the description in the *Environment* tab.

In addition to the information in the *Environment* tab, you can also use specific functions to identify the type of a variable in a dataset (or, really, any other object in your workspace) with the class() function — you may remember this from the previous tutorial.

For example, running class(ess\$cntry) will tell you that the cntry variable is of type 'character' (chr).

Changing storage types

As mentioned earlier, it can happen that one or more of the variables in your dataset are not stored correctly. For example, a variable that really consists of pure numbers was somehow converted to a text variable during the data import process. In that case, you need to be able to transform your variable into its proper storage type.

What this means in practice is again easiest to see by looking at an example. Let's say that, because you are feeling silly today, you want the age variable (agea) not stored as numbers but as text. In other words, you want to convert this variable from type 'numeric' to type 'character'.

To do this, you would use the as.character() transformation function:

```
as.character(ess$agea)

[1] "22" "43" "58" "22" "84" "62" "68" "81" "59" "57" "85" "24" "43" "35" "59" [16] "52" "56" "66" "43" "26" "56" "56" "82" "18" "35" "73" "29" "56" "66" "65" "56" "17" [46] "67" "23" "50" "62" "41" "46" "62" "27" "62" "32" "69" "64" "31" "68" "58" [61] "31" "71" "79" "55" "35" "34" "62" "53" "51" "62" "42" "44" "58" "46" "38" [76] "35" "53" "54" "46" "38" [91] "41" "54" "38" "56" "39" "61" "32" "32" "32" "39" "33" "47" "41" "17" "42" "23" [106] "76" "36" "78" "19" "24" "24" "40" "75" "50" "72" "40" "70" "34" "59" "67" "17" [136] "87" "65" "71" "39" "33" "32" "16" "25"
```

You could of course also directly add this new variable to the ess dataset with the assignment operator:

```
ess$age_chr <- as.character(ess$agea)
```

Take a look at the result above: Do you notice the quotation marks around all of the numbers that R printed out? This indicate that you transformed agea into a character variable: The numbers are still there — but they are now stored as text. R will now refuse to do any calculations with this variable.

For example, if you try to calculate the mean of this new variable, you will get an error message:

```
mean(ess$age_chr, na.rm = T)
Warning in mean.default(ess$age_chr, na.rm = T): argument is not numeric or
logical: returning NA
[1] NA
```

Now you know what the problem often looks like — there is a variable that you know is supposed to be a numeric variable, but it somehow got stored as text. In that case, you have to tell R that it should treat this variable as a proper numeric variable. This works equivalently to the previous operation, but with a different function — as.numeric():

```
# Transform the age_chr variable we just created into a new one and store in ess
ess$age_num <- as.numeric(ess$age_chr)

# This should work now
mean(ess$age_num, na.rm = TRUE)
[1] 47.90909</pre>
```

This might seem like much ado about nothing, but knowing this can really save you a lot of time and headaches. **The important point:** Be conscious of how your data are stored in R, and if how it is stored really makes sense. If it does not, convert your variables into an appropriate format.

Data cleaning and transformations with factors

Categorical or ordinal variables that are stored as factors can cause headaches during the data cleaning and management phase, often simply because they are more complex than pure numeric or character variables. In this last part of the tutorial, you will learn a few tricks that can help you deal with factor variables.

Factor to numeric

Assume you were interested in people's level of satisfaction with life and you therefore wanted to do a statistical analysis with the stflife variable from the ess dataset, which measures exactly this. You also see that this variable has 11 categories — from 0 ("Extremely dissatisfied") to 10 ("Extremely satisfied") — which are enough to be used as a numeric or "metric" variable:

```
visfactor(variable = "stflife", dataset = ess)
 values
      1 Extremely dissatisfied
      2
      3
                               2
      4
                               3
      5
                               4
      6
                               5
      7
                               6
      8
                               7
      9
                               8
     10
                               9
     11
            Extremely satisfied
```

But: R will not do any math with the variable in this form. If you would try to calculate the average level of life satisfaction you get an error message:

```
mean(ess$stflife, na.rm = T)
Warning in mean.default(ess$stflife, na.rm = T): argument is not numeric or
logical: returning NA
[1] NA
```

The problem: stflife is stored as a factor, which you can see when you you check how R stored it:

```
class(ess$stflife)
[1] "factor"
```

Fortunately, you can – in principle – directly extract the numerical scores from a factor-type variable like sftlife into a new numeric variable with as.numeric().

But there is one thing you need to be careful about: Take another look at how the labels and underlying numbers of stflife correspond:

```
visfactor(variable = "stflife", dataset = ess)
values
                          labels
      1 Extremely dissatisfied
      2
      3
                               2
      4
                               3
      5
                               4
      6
                               5
      7
                               6
      8
                               7
      9
                               8
     10
     11
           Extremely satisfied
```

If you take a careful look at the result here you should notice that the **values and labels** are off by 1: "Extremely dissatisfied" – which corresponds to 0 – has the underlying value of 1, the label "1" has the underlying value of 2, and so on. In other words, the values are wrong.

This is a problem because when you use as.numeric(), R will extract the values from stflife – and because the values are wrong, all results based on them will also be wrong.

But: there is an easy way to fix this: You just subtract 1 from the result of as.numeric() and then save the result of this as a new variable:

```
ess$stflife_num <- (as.numeric(ess$stflife) - 1)</pre>
```

If this worked, then the new numeric version of stflife should have a maximum value of 10 – corresponding to the 0-10 scale that the respondents saw when they participated in the survey. And that is now indeed the case:

```
max(ess$stflife_num)
[1] 10
```

This also means that this numeric version of stflife will give you correct results, for example the average value:

```
mean(ess$stflife_num, na.rm = T)
[1] 7.86014
```

The lesson to be learned: You need to really pay attention when you convert factor-type variables to numeric ones! In general: Never run on autopilot, always remain aware of what you are doing to your data!

Here is a simple and quick **checklist** you can use to make sure that you are converting factortype variables correctly to numeric:

- 1. Use bst290::visfactor() to let R show you how the labels and underlying values correspond.
- 2. If the labels and values directly correspond a label of 0 has the value of 0, and so on then you can just directly use as.numeric()
- 3. If the labels and values do not correspond a label of 0 has the value of 1, and so on then you need to adjust the values by hand, e.g., by subtracting or adding the necessary number so that the values are correct.

Factor to character

A related problem you might have is that you want not the numerical scores but the text labels of a factor variable. For example, say you wanted to extract the labels for the different educational degrees in Norway from the edlvdno variable into a new pure character variable.⁸

To create a new variable that contains only the text labels from edlvdno, you use the as.character() function:

```
ess$edlv_chr <- as.character(ess$edlvdno)</pre>
```

You can verify that the new variable really is a character variable with:

```
class(ess$edlv_chr)
[1] "character"
```

And you can see the different levels with unique():

```
unique(ess$edlv_chr)
[1] "Fullført 3-4 årig utdanning fra høgskole (Bachelor-, cand.mag., lærerhøgsko"
[2] "Fullført 5-6 årig utdanning fra høgskole (master, hovedfag)"
[3] "Universitet/høgskole, mindre enn 3 år, men minst 2 år (høgskolekandidat, 2-"
```

- [4] "Fullført 5-6 årig utdanning fra universitet (master, hovedfag), lengre profesj"
- [5] "Vitnemål fra påbygging til videregående utdanning (fagskoleutdanning, teknisk"
- [6] "Videregående avsluttende utdanning, yrkesfaglige studieretninger/utdanningsprog"
- [7] "Ungdomsskole (grunnskole, 7-årig folkeskole, framhaldsskole, realskole)"
- [8] "Fullført 3-4 årig utdanning fra universitet (Bachelor, cand.mag.)"
- [9] "Forkurs til universitet/høgskole som ikke gir studiepoeng"
- [10] "Videregående avsluttende utdanning, allmennfaglige studieretninger/studieforber"
- [11] "Barneskole (første del av obligatorisk utdanning)"
- [12] "Vitnemål fra folkehøgskole"

⁸You already know from the previous tutorial that edvldno is a factor, but feel free to check again with class(ess\$edlvdno).

More tools for working with factors

You probably see now that working with factors can be a bit tedious, simply because they are a bit more complex than other types of variables. But you hopefully also see their structure — numbers with text labels — more clearly now that you have seen how you can extract the different elements with as.numeric() and as.character().

If you find yourself working with factors a lot, you will probably want to use the forcats package. This package is specifically designed for data cleaning and management with factors and is also included in the tidyverse collection. See then also Hadley Wickham's R for Data Science.

Summing up & next steps

Now you should know the basics of data management and data cleaning. This may have been a tough one — but understanding variable types and data transformations is absolutely critical if you want to do your own data analyses. Of course, the emphasis here was on "basics" — there is much left to learn and, if you do your first own data analysis project, you will most likely run into situations in which what you learn here is not sufficient to solve a data cleaning problem.

If you would like to learn more tricks and techniques for data cleaning, here are some resources you can use:

- For a more extensive introduction to the tidyverse approach, see Hadley Wickham's *R* for Data Science: https://r4ds.had.co.nz/).
- For an introduction (in Norwegian) to data cleaning and management using base R, see $Lar\ dig\ R$
- As always: Someone else has probably had your problem or a similar one before, and it was solved on stackoverflow.com.
- Finally, *ChatGPT* and some other "AI" chatbots can do coding, including in R, and they may be able to give you solutions to some problems but be careful, chatbots do "hallucinate" and you may still need to adapt the provided solution before it really works.

Next, some (brief, promise!) de-bugging exercises (as last time: *Tutorials*, and there choose "Data cleaning & management").