R Tutorial 3: Data cleaning & management

Carlo Knotz

Contents

1 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`).*

---

2 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)
```

3 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

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

3.1 `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.

3.1.1 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;

3.1.2 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)
```

3.1.3 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)
```

[2] The `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!

3.2 `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:

- `>` "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")`)[4]

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()`.

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

3.3 `drop_na()`: Getting rid of missing values (`NA`s)

If you are working with any real-life dataset, there will usually be some missing observations: For example, some survey respondents may have refused to answer some questions, or did not know what to say. Such missing observations will be stored as `NA` ("not available") in `R`.

It is often helpful to be able to get rid off `NA`s. This is for example the case when you make graphs — unless you exclude the `NA`s, they will show up in your graph, and that is often not helpful.[5]

You can exclude `NA`s from your dataset with the `drop_na()` function from the `tidyverse`. You can use this function in two ways:

1. If you use it without any further specifications, it simply drops all observations from the dataset if they have any missing information *on any variable* (in this case, the code also saves the result as `ess_noNAs`):

```
ess_noNAs <- drop_na(ess)
```

Careful: Using `drop_na()` in this general way is a kind of "nuclear option" — and, like other types of nuclear options, this approach can have serious negative consequences! For example, if you have survey data and there is one respondent who refused to state their income but answered all other questions, then that respondent will have an `NA` on the income-variable — and only there. By using `drop_na()` in its general form, *all* information about that respondent will be removed, even if only the income-variable is missing for them. *Therefore: You always need to be very careful when you exclude missing observations — and document your code, so that you can always go back and change things if you need to!*

2. An alternative way is to exclude only those observations that have missing information on a specific variable:

```
ess_noNAs <- drop_na(ess, agea)
```

In this second case, you would drop only those observations that have `NA`s on the `agea` variable (i.e, respondents who did not state how old they are) — regardless of how the other variables look like. You can also specify more than one variable here, if you like.

[5] Dropping `NA`s during data cleaning also helps you avoid having to use the `na.rm` option when you calculate summary statistics.

4 Connecting operations with the "pipe" (%>%)

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. 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.

5 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 pipeline
```

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").

Important: The pipe operator may not work with all functions in `R`. It should work fine with functions that come from the `tidyverse` package collection (`select()`, `filter()`, `drop_na()`,...) and it works 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]

---

[6] Alternatively, see this RStudio Community board discussion for a solution: https://community.rstudio.com/t/pipe-operator-does-not-work/66377

6 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.

6.1 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

6.2 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()`**.[7]

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.

6.2.1 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:

```
ess %>% # 1.
  mutate(older = if_else(condition = agea>65, # 2.
                         true = 1, # 3.
                         false = 0)) # 4.
```

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`."

[7] `if_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.

6.2.2 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:

```
ess %>%  # 1.
  mutate(health_dummy = if_else(condition = health %in% c("Very good","Good"), # 2.
                                true = "Good health", # 3.
                                false = "Not good health")) # 4.
```

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    health_dummy
1  12414      Good     Good health
2   9438 Very good     Good health
3  19782      Good     Good health
4  18876 Very good     Good health
5  20508 Very good     Good health
6  19716      Fair Not good health
7  13476      Fair Not good health
8   6762      Fair Not good health
9  19518 Very good     Good health
10 21336      Good     Good 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).

---

7 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.

7.1 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.

7.2 Combining `summarize()` & `group_by()`

You can use the `group_by()` 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()`:

```
ess %>% # 1.
  group_by(gndr) %>% # 2.
  summarize(mean_height = mean(height, na.rm = TRUE)) # 3.
# A tibble: 2 x 2
  gndr   mean_height
  <fct>        <dbl>
1 Male          179.
2 Female        168.
```

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`."

7.3 Other statistics & multiple statistics in one operation

7.3.1 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:

```
ess %>%
  group_by(gndr) %>%
  summarize(med_age = median(agea, na.rm = TRUE))
# A tibble: 2 x 2
  gndr   med_age
  <fct>    <dbl>
1 Male      51
2 Female    42.5
```

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.

7.3.2 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>       <dbl>
1 Male      75    51          86.8
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.

8 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.)

8.1 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`).

8.2 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" "69" "32" "63" "18" "56" "53" "53" "57" "40" "40" "81" "19" "21"
 [31] "43" "67" "66" "43" "26" "56" "82" "18" "35" "73" "29" "56" "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" "56" "66" "40" "44" "60" "60" "20" "71" "17" "21" "58" "90" "32"
 [91] "41" "54" "38" "56" "39" "61" "32" "32" "39" "33" "47" "41" "17" "42" "23"
[106] "76" "36" "74" "23" "55" "18" "43" "28" "44" "44" "38" "48" "38" "48" "41"
[121] "75" "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
```

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.

8.3 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.

8.3.1 *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 10 categories — enough to be used as a numeric or "metric" variable:

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

If you would now 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 extract the numerical scores from the original `sftlife` variable into a new numeric variable with `as.numeric()`:

```
ess$stflife_num <- as.numeric(ess$stflife)
```

But: we are not done yet! If you take another a close look at the result of `visfactor()` above, you will notice that there is a difference between the numerical values of the `stflife` variable and its text labels. For example, the numerical value of `3` corresponds to a text label of "2" – the text labels are shifted by `1` unit.

This is a problem, because we when you transform the `stflife` variable to `numeric`, `R` uses the underlying numbers to create the new variable. And because the underlying numbers are off by one, the new variable is too.

You can also see this when you cross-tabulate the two variables with `table()`:

```
table(ess$stflife,ess$stflife_num)
```

```
                        3  4  5  6  7  8  9 10 11
  Extremely dissatisfied  0  0  0  0  0  0  0  0  0
  1                       0  0  0  0  0  0  0  0  0
  2                       4  0  0  0  0  0  0  0  0
  3                       0  3  0  0  0  0  0  0  0
  4                       0  0  2  0  0  0  0  0  0
  5                       0  0  0  5  0  0  0  0  0
  6                       0  0  0  0  7  0  0  0  0
  7                       0  0  0  0  0 20  0  0  0
  8                       0  0  0  0  0  0 46  0  0
  9                       0  0  0  0  0  0  0 36  0
  Extremely satisfied     0  0  0  0  0  0  0  0 20
```

You see that 2 corresponds to 3, 3 corresponds to 4, and so on. This means that any calculation you do with the new numeric variable will be wrong!

But correcting this is easy to do: You just subtract `1` from the new variable and save the result:

```
ess$stflife_num <- ess$stflife_num - 1

table(ess$stflife,ess$stflife_num) # do you see the difference to the earlier cross-table?
```

```
                        2  3  4  5  6  7  8  9 10
  Extremely dissatisfied  0  0  0  0  0  0  0  0  0
  1                       0  0  0  0  0  0  0  0  0
  2                       4  0  0  0  0  0  0  0  0
  3                       0  3  0  0  0  0  0  0  0
  4                       0  0  2  0  0  0  0  0  0
  5                       0  0  0  5  0  0  0  0  0
  6                       0  0  0  0  7  0  0  0  0
  7                       0  0  0  0  0 20  0  0  0
  8                       0  0  0  0  0  0 46  0  0
  9                       0  0  0  0  0  0  0 36  0
  Extremely satisfied     0  0  0  0  0  0  0  0 20
```

With this new numeric version of `stflife` you get 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! It *can* happen that the numeric values are not the same as the actual data values. This is not always the case, but often (especially when you work with the *ESS*). In general: Never run on autopilot, always remain aware of what you are doing to your data!

8.3.2 *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)
```

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"
```

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

8.3.3 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*.

9 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 *Lær 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").