Fortgeschrittene Datentransformation

José C. García Alanis & Mareike Hülsemann

14. Juni, 2023

Inhalte dieser Übung

- Logische Verknüpfungen für multiple Bedingungen
- Transformation zwischen `long`- und `wide`-Format
- Zusammenfügen verschiedener Datensätzen

Logische Verknüpfungen für multiple Bedingungen

`ifelse`-Funktion

Mit der `ifelse`-Funktion können wir in sehr sparsamer Schreibweise, einen `if-else`-Block ausführen. `if-else`-Blöcke ermöglichen es, bestimmten Code nur dann auszuführen, wenn gewisse Voraussetzungen zutreffen. Falls diese nicht zu treffen, wird ein anderer Code ausgeführt.

Wir schauen uns zunächst eine einfache `if-else`-Abfrage und deren Ergebnis an:

```
Alter <- 16
if (Alter >= 18) {
  party_einlass <- TRUE
} else {
  party_einlass <- FALSE
}
party_einlass
```

```
## [1] FALSE
```

Die `if-else`-Abfrage können wir mit der Funktion `ifelse` auf eine Zeile kürzen:

```r
Alter <- 16

party_einlass <- ifelse(Alter >= 18, TRUE, FALSE)

party_einlass
```

```
## [1] FALSE
```

Wir sehen, dass wir mit der Kurzschreibweise dasselbe Ergebnis erzielen. Wir müssen uns für die `ifelse`-Funktion lediglich drei Dinge merken.

1. Das erste Argument enthält immer die Bedingungsabfrage (Voraussetzungsprüfung), also eine logische Abfrage.
2. Das zweite Argument gibt an, was passiert, wenn die Bedingung zutrifft.
3. Das dritte Argument gibt an, was passiert, wenn die Bedingung *nicht* zutrifft.

Übung 1 Überlegen Sie sich ein sinnvolles Beispiel für die Datenauswertung im psychologischen Kontext, bei der Sie eine `ifelse`-Abfrage benötigen und schreiben Sie (Pseudo-) Code für dieses Beispiel. Schreiben Sie dabei sowohl Code für die klassische Variante als auch für die `ifelse`-Funktion.

`case_when`-Funktion

Die `case_when`-Funktion ermöglicht es, die `ifelse`-Funktion in eine `dplyr`-Pipeline einzubinden. Außerdem können wir hier nicht nur zwei Fälle (`if` und `else`) definieren, sondern beliebig viele (`if`, `ifelse` und `else`). Mit der `case_when`-Funktion können beliebig viele if-Abfragen (`if` und `ifelses`) gekoppelt werden.

Wir schauen uns das anhand eines Beispieldatensatzes zum Ausprägungsgrad einer Depression an. Der Ausprägungsgrad wurde mit dem Patient Health Questionnaire (PHQ).

```r
# Benötigte Pakete
library(dplyr)

# Hier erstellen wit einen Beispieldatensatz
depressionen <- data.frame(
  id = c("a5", "h7", "q1"),
  phq = c(7, 14, 20))

# und schauen uns den an
head(depressionen)
```

```
##   id phq
## 1 a5   7
## 2 h7  14
## 3 q1  20
```

Nun wollen wir mit dem folgenden Code `case_when` einsetzen, um die Schwere der Depression der Patient:innen anhand ihrer Skalenwerte vorzunehmen.

```
depressionen <- depressionen %>%
  # hier ertellen wir eine neue variable, deren Wert über
  # `case_when` definiert wird
  mutate(
    Schweregrad = case_when(
      phq < 5 ~ "keine",
      phq >= 5 & phq < 10 ~ "leicht",
      phq >= 10 & phq < 15 ~ "mittel",
      phq >= 15 & phq < 20 ~ "ausgeprägt",
      phq >= 20 ~ "schwer"
    )
  )

head(depressionen)
```

```
##   id phq Schweregrad
## 1 a5   7      leicht
## 2 h7  14      mittel
## 3 q1  20      schwer
```

Übung 2 Schreiben Sie nun Ihr Beispiel aus der vorangegangenen Übung unter Verwendung der `case_when`-Funktion um.

Übung 3 Nutzen Sie die Funktion `case_when` und den Piping-Operator (%>%), um eine neue Spalte im Beispieldatensatz `gedaechtnis` zu erstellen. Diese soll die kognitive Leistungsfähigkeit einer jeden Person in Abhängigkeit von ihrem Alter kategorisieren. Dabei gilt:

- eine kognitive Leistung unter 10 Punkten bei Personen bis zu 20 Jahren als schlecht,
- eine kognitive Leistung ab 10 Punkten bei Personen bis zu 20 Jahren als gut,
- eine kognitive Leistung unter 8 Punkten bei Personen zwischen 20 und 40 Jahren als schlecht,
- eine kognitive Leistung ab 8 Punkten bei Personen zwischen 20 und 40 Jahren als gut,
- eine kognitive Leistung unter 5 Punkten bei Personen ab 40 Jahren als schlecht und

- eine kognitive Leistung ab 5 Punkten bei Personen ab 40 Jahren als gut.

```r
# führen Sie diesen Code aus um den Datensatz
# `gedächtnis` zu erstellen
gedaetnis <- data.frame(
  subject_id = 1:10,
  alter = c(19, 48, 33, 39, 20, 38, 40, 25, 47, 50),
  leistung = c(12, 15, 7, 2, 6, 13, 7, 4, 13, 8)
)
```

Transformation zwischen `long`- und `wide`-Format

Nun schauen wir uns zwei weitere Funktionen an: `pivot_longer` und `pivot_wider` aus dem R-Paket `tidyr`.

Oft liegen Rohdaten in einem Format vor, das pro Versuchsperson eine Zeile bereithält und alle, für diese Person, erhobenen Daten in verschiedenen Spalten abspeichert. Dieses Format nennen wir "`wide`-Datenformat". Im Falle von Messwiederholung haben wir dann z. B. zwei Spalten für denselben Fragebogenwert: eine Spalte für Messzeitpunkt 1 und eine weitere Spalte für Messzeitpunkt 2.

Wir folgen dem Lehrbuch von Maike Luhmann (*R für Einsteiger*, 2020) und betrachten den Datensatz `Minidaten_2.RData`. In diesem sind die Variablen `neuro_1`, `neuro_2`, `neuro_3` und `extra_1`, `extra_2`, `extra_3` enthalten. Es handelt sich um messwiederholte Daten, d. h. Neurotizismus (Variable `neuro`) und Extraversion (Variable `extra`) wurden zu jeweils drei Messzeitpunkten erhoben. Wir erkennen die Messzeitpunkte an den Endungen (`_1`, `_2`, `_3`).

Speichern Sie sich den Datensatz `Minidaten_2.RData` ab und lesen Sie ihn in R ein

```
load("Minidaten 2.RData")
```

Das `wide`-Datenformat hat zwei Nachteile:

1. Viele Funktionen für die Analyse von messwiederholten Daten benötigen einen Datensatz, in dem es nur eine Spalte für jede abhängige Variable gibt und in dem pro Zeile eine Beobachtung steht.
2. Alle Funktionen aus dem Paket `tidyverse` (dazu gehören auch die Pakete `dplyr` und `ggplot2`) basieren auf der Annahme, dass die Daten im Long-Format vorliegen.

Das `long`-Format zeichnet sich dadurch aus, dass - für jede Variable eine Spalte existiert, - für jede Beobachtung (z. B. Versuchspersonen oder Messzeitpunkte) eine Zeile existiert und - für jeden Wert eine Spalte existiert.

Unser Datensatz liegt im `wide`-Format vor: Jede Zeile steht für eine andere Versuchsperson und die Spalten enthalten zum Teil dieselben Variablen, für die drei Messzeitpunkte.

Das heißt, unsere Variablen enthalten nicht nur eine Information, sondern mehrere: Die Ausprägung (Neurotizismus oder Extraversion) und den Messzeitpunkt (1, 2 oder 3).

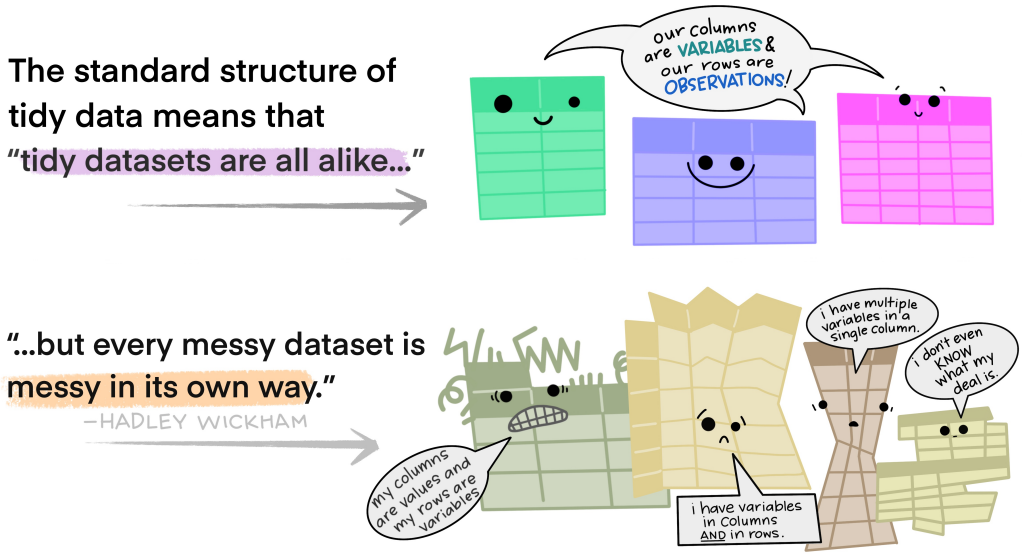

Figure 1: Illustrations from the Openscapes blog Tidy Data for reproducibility, efficiency,and collaboration by Julia Lowndes and Allison Horst, https://www.openscapes.org/blog/2020/10/12/tidy-data/

Datentransformation von `wide` zu `long`

Wir wollen nun den vorliegenden Datensatz vom `wide`-Format ins `long`-Format transformieren. Dazu nutzen wir die Funktion `pivot_longer`. Schauen wir uns den Datensatz zunächst einmal an:

```
daten2
```

```
##   id neuro_1 neuro_2 neuro_3 extra_1 extra_2 extra_3 wohnort
## 1  1       4       3       4       2       4       3    Land
## 2  4       2       3       2       2       2       2   Stadt
## 3  2       1       1       4       3       1       3    Land
## 4  5      NA       2       4       1       2       5    Land
## 5 10       3       3       2       5       5       1   Stadt
## 6 12       5       5       1       2       5       2    Land
## 7 13       1       4       2       1       3       2   Stadt
```

Schritt 1: Umstrukturierung ins Ultralong-Format Die Funktionen `pivot_longer`und `pivot_wider` gehören zu den Funktionen von `tidyr` und werden benutzt, um Datensätze zu transformieren. Wir müssen also das Paket `tidyr` laden, um diese Funktionen nutzen zu können (der Einfachheit halber können wir aber auch `tidyverse` laden, dass sowohl `tidyr` als auch `dplyr` und `ggplot2` beinhaltet).

```
library(tidyr)
```

Die Funktion `pivot_longer` transformiert einen Datensatz im `wide`-Format ins `long`-Format, indem verschiedene Spalten zu einer einzigen Spalte zusammengefasst werden und die einzelnen Werte jeweils in eine eigene Zeile geschrieben werden.

```
ultralang <- daten2 %>%
  pivot_longer(cols = -c(id, wohnort),
               names_to = "varname",
               values_to = "wert")
```

Die Funktion `pivot_longer` benötigt drei Argumente. Mit `cols = -c(id, wohnort)` sorgen wir dafür, dass die Variablen `id` und `wohnort` von der Umwandlung ausgenommen werden und diese Information in jede Zeile des neuen Datensatzes `ultralang` geschrieben wird.

Das Argument `names_to = "varname"` sorgt dafür, dass die ursprünglichen Variablennamen in einer eigenen Spalte abgespeichert werden. Wir benötigen diese noch, um unseren Datensatz ins gewünschte Long-Format zu transformieren und um die Information, was die abgespeicherten Werte repräsentieren, nicht zu verlieren.

Das Argument `values_to = "wert"` sorgt dafür, dass jeder unserer Werte in einer Spalte namens `wert` abgespeichert wird.

Schauen wir uns das Ergebnis einmal an:

```
ultralang
```

```
## # A tibble: 42 x 4
##       id wohnort varname  wert
##    <dbl> <fct>   <chr>   <int>
##  1     1 Land    neuro_1     4
##  2     1 Land    neuro_2     3
##  3     1 Land    neuro_3     4
##  4     1 Land    extra_1     2
##  5     1 Land    extra_2     4
##  6     1 Land    extra_3     3
##  7     4 Stadt   neuro_1     2
##  8     4 Stadt   neuro_2     3
##  9     4 Stadt   neuro_3     2
## 10     4 Stadt   extra_1     2
## # ... with 32 more rows
```

Unser neuer Datensatz `ultralang` hat nur noch 4 Variablen (im Gegensatz zu den ursprünglichen 8). Es gibt weiterhin die Variablen `id` und `wohnort` und die neuen Variablen

varname und wert. Unsere ursprünglichen Variablen id und wohnort sind nicht messwiderholt und nehmen nun im neuen Datensatz nicht mehr nur eine Zeile ein, sondern werden über mehrere Zeilen wiederholt. Wenn wir uns die Daten anschauen, sehen wir, dass die erste Person nun sechs Zeilen erhält und die Informationen darin identisch sind. In der dritten Spalte stehen jetzt die ursprünglichen Spaltennamen unserer abhängigen Variablen. Die Werte einer Versuchsperson für Neurotizismus und Extraversion stehen nun nicht mehr in einer Zeile und sechs Spalten, sondern in sechs Zeilen und einer Spalte, die den Namen wert trägt. Wir haben nun also eine Spalte, die die Werte für zwei verschiedenen Variablen (extra und neuro) zu drei Messzeitpunkten enthält.

Daten, die in einem übersichlichlichen long-Format forliegen, nennen wir häufig Tidy Data (was so viel heisst wie "aufgeräumte Datensätze").

Übung 4 Was passiert wenn Sie die Variable wohnort nicht ausschließen?

Übung 5 Was passiert, wenn Sie names_to = "Spaltenbenennung" anstatt des ursprünglichen Codes benutzen?

In unserem Datensatz ultralang sind in der dritten Spalte aber immer noch verschiedenen Informationen, nämlich zu Ausprägung und Messzeitpunkt, vermischt. Wir können die pivot_longer Funktion erweitern, und die Variablennamen auftrennen:

```
ultralang <- daten2 %>%
  pivot_longer(cols = -c(id, wohnort),
               names_to = c("varname", "mzp"),
               names_sep = "_",
               values_to = "wert")
```

Nun sieht das Ergebnis so aus:

```
ultralang
```

```
## # A tibble: 42 x 5
##       id wohnort varname mzp    wert
##    <dbl> <fct>   <chr>   <chr> <int>
##  1     1 Land    neuro   1         4
##  2     1 Land    neuro   2         3
##  3     1 Land    neuro   3         4
##  4     1 Land    extra   1         2
##  5     1 Land    extra   2         4
##  6     1 Land    extra   3         3
##  7     4 Stadt   neuro   1         2
##  8     4 Stadt   neuro   2         3
##  9     4 Stadt   neuro   3         2
```

```
## 10     4 Stadt   extra   1         2
## # ... with 32 more rows
```

Unsere Daten entsprechen nun zwar den `tidy`-Vorgaben, sind aber noch immer nicht im gewünschten `long`-Format, denn in diesem soll jede Variable eine eigene Spalte bekommen. In unserem Datensatz stehen die zwei Variablen Neurotizismus und Extraversion aber in ein und derselben Spalte.

Schritt 2: Umstrukturierung ins Long-Format

Wir wollen, dass unterschiedliche psychologische Merkmale jeweils ihre eigene Variable erhalten, also in verschiedenen Spalten auftauchen. Wir müssen unsere Ultralong-Daten also wieder etwas verbreitern. Dies schaffen wir mit der Funktion `pivot_wider`.

```
lang <- ultralang %>%
  pivot_wider(names_from = varname,
              values_from = wert)
```

Mit dem Argument `names_from` sagen wir `R`, wo es die neuen Variablennamen findet. Mit `pivot_wider` erstellen wir also neue Variablen und bennenen diese so, wie es in der Spalte `varname` abgespeichert ist. Das Argument `values_from` legt fest, wo die Werte für die neuen Variablen zu finden sind. Das Ergebnis sieht folgendermaßen aus:

```
lang
```

```
## # A tibble: 21 x 5
##       id wohnort mzp   neuro extra
##    <dbl> <fct>   <chr> <int> <int>
##  1     1 Land    1         4     2
##  2     1 Land    2         3     4
##  3     1 Land    3         4     3
##  4     4 Stadt   1         2     2
##  5     4 Stadt   2         3     2
##  6     4 Stadt   3         2     2
##  7     2 Land    1         1     3
##  8     2 Land    2         1     1
##  9     2 Land    3         4     3
## 10     5 Land    1        NA     1
## # ... with 11 more rows
```

Nun hat unser Datensatz die richtige Struktur um zum Beispiel eine messwiederholte ANOVA berechnen zu können oder auch um sinnvolle Grafiken zu den Variablen Neurotizismus und Extraversion erstellen zu können.

Datentransformation von `long` zu `wide`

Wir haben bereits gezeigt, wie man einen langen Datensatz wieder etwas breiter machen kann. Durch eine erneute Anwendung von `pivot_wider` können wir unseren Datensatz `lang` nun wieder ins Ausgangsformat, in ein `wide`-Format, zurücktransformieren.

```
breit <- lang %>%
  pivot_wider(names_from = mzp,
              values_from = -c(id, wohnort, mzp),
              names_sep = ".")
```

Vergleichen Sie den Ausgangsdatensatz `daten2` und das Ergebnis unserer Transformation:

```
breit
```

```
## # A tibble: 7 x 8
##      id wohnort neuro.1 neuro.2 neuro.3 extra.1 extra.2 extra.3
##   <dbl> <fct>     <int>   <int>   <int>   <int>   <int>   <int>
## 1     1 Land          4       3       4       2       4       3
## 2     4 Stadt         2       3       2       2       2       2
## 3     2 Land          1       1       4       3       1       3
## 4     5 Land         NA       2       4       1       2       5
## 5    10 Stadt         3       3       2       5       5       1
## 6    12 Land          5       5       1       2       5       2
## 7    13 Stadt         1       4       2       1       3       2
```

Das Argument `values_from` kann entweder alle messwiederholten Variablen enthalten oder wir können auflisten, welche Variablen von der Transformation ausgeschlossen werden sollen. Das `-`-Zeichen sorgt also, wie wir das auch schon von `select` kennen, zum Ausschluss der genannten Variablen.

Wir wollen die Datentransformation zwischen Wide- und Long-Format nun üben.

Übung 6 Transformieren Sie den folgenden Datensatz in das Long-Format. Erstellen Sie dafür zwei neue Spalten mit den Namen `monat` und `index`). Die Werte sollen in die Spalte `temp` geschrieben werden. Macht es einen Unterschied, ob Sie das Argument `names_sep = "_"` oder `names_sep = "."` benutzen?

```
df_wide <- data.frame(
  ort = c("Bonn", "Hamburg", "Mainz"),
  temp_jan = c(3.0, 3.5, 3.4),
  temp_feb = c(3.9, 4.4, 5.3),
  temp_mar = c(6.6, 8.0, 9.7),
```

```
  temp_apr = c(10.2, 12.3, 14.2),
  temp_mai = c(14.3, 17.5, 19.0),
  temp_jun = c(18.2, 19.9, 22.0)
)
```

Übung 7 Transformieren Sie den Datensatz `therapy.txt` in das Wide-Format. Beachten Sie, dass dieser Datensatz Tabstoppgetrennt abgespeichert ist. Finden Sie selbst heraus, welche Variablen dieser Datensatz enthält und ob diese messwiederholt sind oder nicht.

Übung 8 Transformieren Sie den Datensatz `reakionzeiten_genested.csv` in das Long-Format. Gehen Sie dabei Schritt für Schritt vor, so wie es in diesem Dokument erklärt wurde. In diesem Datensatz finden Sie simulierte Reaktionszeiten für den Attentional Network Test. Dieser erfasst unter anderem die Leistung derAufmerksamkeitssysteme "Alerting" und "Orienting". In diesem Datensatz wurden die Systeme an zwei Tagen, jeweils morgens und abends getestet.

Zusammenfügen von Datensätzen

Sie kennen das bestimmt aus Ihrer Bachelorarbeit (oder von einem anderen Analyseprojekt), dass unterschiedliche Daten in unterschiedlichen Dateien vorliegen. Häufig liegen verschiedene Angaben zu den Versuchspersonen (Alter, Abschluss, etc.) in einer Datei vor und die experimentellen Daten (Fragebögen, Reaktionszeiten, etc.) in anderen, separaten Dateien.

Für unsere Analysen ist es dann häufig hilfreich, wenn diese Dateien kurzfristig "zusammengefügt" werden. Somit können wir Beziehungen zwischen beispielsweise Personenvariablen und Kennwerten aus einer Untersuchung aufdecken. Dafür ist es notwendig, dass in beiden Datensätzen ein **Personenidentifier** (z.B. die *VPN-Nummer* oder `id`) vorhanden ist. Dieser muss für jede Person einzigartig sein, sonst können wir die Beobachtungen nicht eindeutig den Personen zuordnen.

Glücklicherweise können wir Datensätze mit `dplyr` zusammenfügen. Diese Operation nennt man **join** (verbinden). Bei `dplyr` gibt es zwei verschiedene `joins`, die wir gerne heute anschauen wollen, *inner_join* und *outer_join* (z.B. `full_join`, `left_join` und `right_join`).

`joins` mit `dyplr`

`dplyr` enthält die Funktion `inner_join()`. Die Funktion bekommt als Argumente zwei Datensätze (`x` und `y`) und den Persoenidentifier (`by`). Das Ergebnis eines `inner_join` enthält alle Spalten sowie alle Zeilen von `x` und `y`, für die es einen Match in beiden Datensätze gibt. Beobachtungen und IDs, die nicht in beiden Datensätzen vorkommen, tauchen im Ergebnis

eines `inner_join` nicht auf (sie werden nicht als `NA` eingefügt). Es ist daher immer Vorsicht angebracht.

Als einfaches Beispiel verwenden wir hier eine Tabelle mit Daten über das Alter von zehn Personen und eine Tabelle mit ihren Wohnorten:

Erstellen wir erst einmal die `altersdaten`:

```
alter <- c(18, 20, 45, 35, 17, 22, 27)
id <- c(1, 2, 3, 4, 5, 6, 7)

altersdaten <- data.frame(
  alter = alter,
  id = id
)

altersdaten
```

```
##   alter id
## 1    18  1
## 2    20  2
## 3    45  3
## 4    35  4
## 5    17  5
## 6    22  6
## 7    27  7
```

Und jetzt die `wohnortdaten`:

```
# erstellen wir ersteimal die `wohnortdaten`
wohnort <- c('Mainz', 'Frankfurt', NA, 'Berlin',
             'Wiesbaden', 'Mainz', 'München')
id <- c(1, 2, 3, 4, 5, 6, 8)

wohnortdaten <- data.frame(
  wohnort = wohnort,
  id = id
)

wohnortdaten
```

```
##     wohnort id
## 1     Mainz  1
## 2 Frankfurt  2
## 3      <NA>  3
```

```
## 4    Berlin  4
## 5 Wiesbaden  5
## 6     Mainz  6
## 7   München  8
```

Achtung!: Schauen Sie sich beide Datensätze genau an. Vielleicht ist es Ihnen aufgefallen, dass wir im Datensatz `wohnortdaten` eine Person 8 haben aber keine Person 7. Im Datensatz `altersdaten` haben wir dafür eine Person 7 aber keine Person 8. Zusätzlich haben wir in einem Datensatz fehlende Werte (kein Wohnort für Person 3).

inner_join

Benutzen wir jetzt ein `inner_join`, um die Datensätze zusammenzufügen. Das erste Argument (`x`) ist unser Datensatz `altersdaten`. Das zweite Argument (`y`) ist unser Datensatz `wohnortdaten`. Im dritten Argument (`by`) setzen wir den Personenidentifier (`id`).

```
library(dplyr)
inner_join_daten <- inner_join(
  x = altersdaten,
  y = wohnortdaten,
  by = 'id'
)

inner_join_daten
```

```
##   alter id   wohnort
## 1    18  1     Mainz
## 2    20  2 Frankfurt
## 3    45  3      <NA>
## 4    35  4    Berlin
## 5    17  5 Wiesbaden
## 6    22  6     Mainz
```

Die `inner_join()` Funktion gibt uns einen Datensatz als Ergebnis. In dem Datensatz befinden sich die Personen, die in beiden Datensätze einen Eintrag haben. Dagegen sind Person 7 und Person 8 im "Ergebnis"-Datensatz **nicht** vorhanden. Das hat den Grund, dass Person 7 nur im ersten Datensatz vorkommt und Person 8 nur im Zweiten. D.h. diese beiden Personen fliegen raus.

outer_join

Im Unterschied zum `inner_join` werden beim `outer_join` alle IDs beibehalten. Das Ergebnis ist dann ein Datensatz, in dem alle IDs enthalten sind. Beobachtungen von den IDs, die es nur in einem Datensatz gibt, werden als `NA` gekennzeichnet.

Die einfachste Version eines `outer_join` ist der `full join`, bei dem alle Zeilen und alle Spalten aus beiden Datensätze beibehalten werden. Die entsprechende Funktion heißt `full_join()`.

```
library(dplyr)
full_join_daten <- full_join(
  x = altersdaten,
  y = wohnortdaten,
  by = 'id'
)

full_join_daten
```

```
##   alter id   wohnort
## 1    18  1     Mainz
## 2    20  2 Frankfurt
## 3    45  3      <NA>
## 4    35  4    Berlin
## 5    17  5 Wiesbaden
## 6    22  6     Mainz
## 7    27  7      <NA>
## 8    NA  8   München
```

Wie Sie sehen können, enthält dieser Datensatz alle Zeilen mit ihren IDs (auch die IDs, die nur in einem der Datensätze zu finden sind).

Wrap-up

`inner_join` und `full_join` werden häufig eingesetzt, um Datensätze zusammenzufügen. Es gibt aber auch weitere `joins`, die sicherlich für spezielle Arten des "Zusammenfügens" hilfreich sein können. Dazu zählen `left_join` und `right_join`. Diese werden wir heute nicht behandeln aber der folgende Link gibt sicherlich einen guten Überblick über diese Methoden:

https://www.phonetik.uni-muenchen.de/~jmh/lehre/basic_r/_book/joining-mit-dplyr.html

Außerdem haben wir diese in der Sitzung zu `dplyr` bereits besprochen.

Übung 9 Zuvor haben wir im Argument `by` von `inner_join` einfach den Namen der ID-Variable schreiben müssen. Diese hatte im vorherigen Beispiel den gleichen Namen in beiden Datensätze. `inner_join` hat dann automatisch in beiden Datensätze geschaut, diese Variable gefunden und die Daten der Personen anhand ihrer ID zusammengefügt.

Was machen wir allerdings, wenn die ID-Variable in den beiden Datensätzen unterschiedliche Namen hat? Glücklicherweise hat `inner_join` hierfür eine Lösung parat. Wir können das Argument `by` so verändern, dass ID-Variablen mit unterschiedlichen Namen berücksichtigt werden.

Beispiel: Wir verändern `altersdaten` so, dass die ID-Variable (`id`) nun `VP` heißt.

```
altersdaten <- data.frame(
  alter = c(18, 20, 45, 35, 17, 22, 27),
  VP = c(1, 2, 3, 4, 5, 6, 7)
)

altersdaten
```

```
##   alter VP
## 1    18  1
## 2    20  2
## 3    45  3
## 4    35  4
## 5    17  5
## 6    22  6
## 7    27  7
```

Suchen Sie nun im Internet nach einer Möglichkeit `inner_join` oder `full_join` bei Datensätzen einzusetzen, die ID-Variablen mit unterschiedlichen Namen enthalten. Alternativ können Sie die `help()`-Funktion in R benutzen.

Erstellen Sie einen R-Code-Chunk und fügen Sie dann die Datensätze zusammen. Das Ergebnis sollte so aussehen:

```
##   alter VP   wohnort
## 1    18  1     Mainz
## 2    20  2 Frankfurt
## 3    45  3      <NA>
## 4    35  4    Berlin
## 5    17  5 Wiesbaden
## 6    22  6     Mainz
## 7    27  7      <NA>
## 8    NA  8   München
```

Viel Erfolg beim Coden!

Literatur

- Luhmann, M. (2020). R für Einsteiger: Einführung in die Statistik-Software für die Sozialwissenschaften. Mit Online-Material (Originalausgabe Edition). Beltz.