Datenaufbereiten mit dplyr

Datenjudo

In diesem Kapitel benötigte Pakete:

```
library(tidyverse)  # Datenjudo
library(corrr)  # Korrelationen berechnen mit der Pfeife
library(stringr)   # Texte bearbeiten
library(car)  # für 'recode'
```

Mit *Datenjudo* ist gemeint, die Daten für die eigentliche Analyse "aufzubereiten". Unter *Aufbereiten* ist hier das Umformen, Prüfen, Bereinigen, Gruppieren und Zusammenfassen von Daten gemeint. Die deskriptive Statistik fällt unter die Rubrik Aufbereiten. Kurz gesagt: Alles, wan tut, nachdem die Daten "da" sind und bevor man mit anspruchsvoller(er) Modellierung beginnt.

Ist das Aufbereiten von Daten auch nicht statistisch anspruchsvoll, so ist es trotzdem von großer Bedeutung und häufig recht zeitintensiv. Eine Anekdote zur Relevanz der Datenaufbereitung, die (so will es die Geschichte) mir an einer Bar nach einer einschlägigen Konferenz erzählt wurde (daher keine Quellenangebe, Sie verstehen...). Eine Computerwissenschaftlerin aus den USA (deutschen Ursprungs) hatte einen beeindruckenden "Track Record" an Siegen in Wettkämpfen der Datenanalyse. Tatsächlich hatte sie keine besonderen, raffinierten Modellierungstechniken eingesetzt; klassische Regression war ihre Methode der Wahl. Bei einem Wettkampf, bei dem es darum ging, Krebsfälle aus Krankendaten vorherzusagen (z.B. von Röntgenbildern) fand sie nach langem Datenjudo heraus, dass in die "ID-Variablen" Information gesickert war, die dort nicht hingehörte und die sie nutzen konnte für überraschend (aus Sicht der Mitstreiter) gute Vorhersagen zu Krebsfällen. Wie war das möglich? Die Daten stammten aus mehreren Kliniken, jede Klinik verwendete ein anderes System, um IDs für Patienten zu erstellen. Überall waren die IDs stark genug, um die Anonymität der Patienten sicherzustellen, aber gleich wohl konnte man (nach einigem Judo) unterscheiden, welche ID von welcher Klinik stammte. Was das bringt? Einige Kliniken waren reine Screening-Zentren, die die Normalbevölkerung versorgte. Dort sind wenig Krebsfälle zu erwarten. Andere Kliniken jedoch waren Onkologie-Zentren für bereits bekannte Patienten oder für Patienten mit besonderer Risikolage. Wenig überraschen, dass man dann höhere Krebsraten vorhersagen kann. Eigentlich ganz einfach; besondere Mathe steht hier (zumindest in dieser Geschichte) nicht dahinter. Und, wenn man den Trick kennt, ganz einfach. Aber wie so oft ist es nicht leicht, den Trick zu finden. Sorgfältiges Datenjudo hat hier den Schlüssel zum Erfolg gebracht.

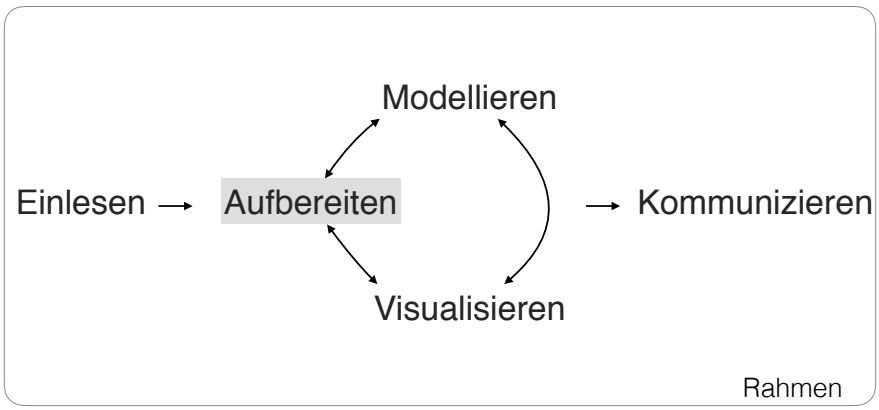


Abbildung 1: Daten aufbereiten

Typische Probleme

Bevor man seine Statistik-Trickkiste so richtig schön aufmachen kann, muss man die Daten häufig erst noch in Form bringen. Das ist nicht schwierig in dem Sinne, dass es um komplizierte Mathe ginge. Allerdings braucht es mitunter recht viel Zeit und ein paar (oder viele) handwerkliche Tricks sind hilfreich. Hier soll das folgende Kapitel helfen.

Typische Probleme, die immer wieder auftreten, sind:

- *Fehlende Werte*: Irgend jemand hat auf eine meiner schönen Fragen in der Umfrage nicht geantwortet!
- *Unerwartete Daten*: Auf die Frage, wie viele Facebook-Freunde er oder sie habe, schrieb die Person "I like you a lot". Was tun???
- *Daten müssen umgeformt werden*: Für jede der beiden Gruppen seiner Studie hat Joachim einen Google-Forms-Fragebogen aufgesetzt. Jetzt hat er zwei Tabellen, die er "verheiraten" möchte. Geht das?
- *Neue Variablen (Spalten) berechnen*: Ein Student fragt nach der Anzahl der richtigen Aufgaben in der Statistik-Probeklausur. Wir wollen helfen und im entsprechenden Datensatz eine Spalte erzeugen, in der pro Person die Anzahl der richtig beantworteten Fragen steht.

Daten aufbereiten mit `dplyr`

Es gibt viele Möglichkeiten, Daten mit R aufzubereiten; `dplyr` ist ein populäres Paket dafür. Eine zentrale Idee von `dplyr` ist, dass es nur ein paar wenige Grundbausteine geben sollte, die sich gut kombinieren lassen. Sprich: Wenige grundlegende Funktionen mit eng umgrenzter Funktionalität. Der Autor, Hadley Wickham, sprach einmal in einem Forum (citation needed), dass diese Befehle wenig können, das Wenige aber gut. Ein Nachteil dieser Konzeption kann sein, dass man recht viele dieser Bausteine kombinieren muss, um zum gewünschten Ergebnis zu kommen. Außerdem muss man die Logik des Baukastens gut verstanden habe - die Lernkurve ist also erstmal steiler. Dafür ist man dann nicht darauf angewiesen, dass es irgendwo "Mrs Right" gibt, die genau das kann, was ich will. Außerdem braucht man sich auch nicht viele Funktionen merken. Es reicht einen kleinen Satz an Funktionen zu kennen (die praktischerweise konsistent in Syntax und Methodik sind).

Willkommen in der Welt von `dyplr`! `dplyr` hat seinen Namen, weil es sich ausschließlich um *D*ataframes bemüht; es erwartet einen Dataframe als Eingabe und gibt einen Dataframe zurück (zumindest bei den meisten Befehlen).

Diese Bausteine sind typische Tätigkeiten im Umgang mit Daten; nichts Überraschendes. Schauen wir uns diese Bausteine näher an.

Zeilen filtern mit `filter`

Häufig will man bestimmte Zeilen aus einer Tabelle filtern. Zum Beispiel man arbeitet für die Zigarettenindustrie und ist nur an den Rauchern interessiert (die im Übrigen unser Gesundheitssystem retten [@kraemer2011wir]), nicht an Nicht-Rauchern; es sollen die nur Umsatzzahlen des letzten Quartals untersucht werden, nicht die vorherigen Quartale; es sollen nur die Daten aus Labor X (nicht Labor Y) ausgewertet werden etc.

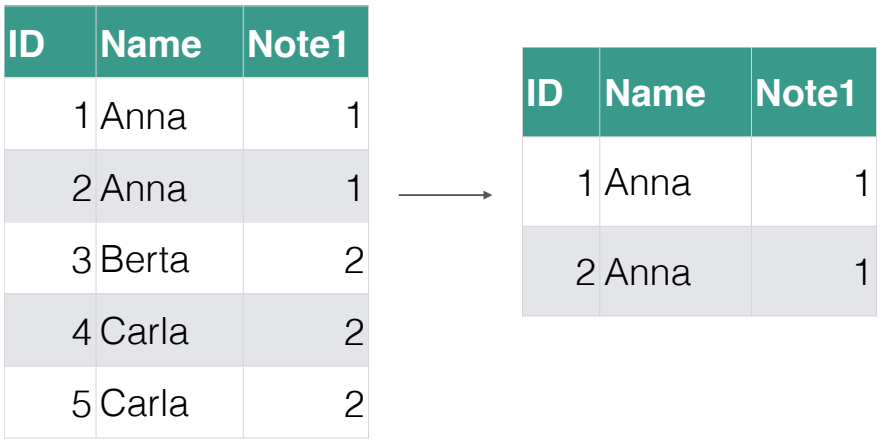
Ein Sinnbild:

Merke:

> Die Funktion `filter` filtert Zeilen aus einem Dataframe.

Schauen wir uns einige Beispiel an; zuerst die Daten laden nicht vergessen. Achtung: "Wohnen" die Daten in einem Paket, muss dieses Paket installiert sein, damit man auf die Daten zugreifen kann.

```
data(profiles, package = "okcupiddata")  # Das Paket muss installiert sein
```



ID	Name	Note1
1	Anna	1
2	Anna	1
3	Berta	2
4	Carla	2
5	Carla	2

ID	Name	Note1
1	Anna	1
2	Anna	1

Abbildung 2: Zeilen filtern

```
df_frauen <- filter(profiles, sex == "f")  # nur die Frauen
df_alt <- filter(profiles, age > 70)  # nur die alten
df_alte_frauen <- filter(profiles, age > 70, sex == "f")  # nur die alten Frauen, d.h. UND-Verknüpfung
df_nosmoke_nodrinks <- filter(profiles, smokes == "no" | drinks == "not at all")
# liefert alle Personen, die Nicht-Raucher *oder* Nicht-Trinker sind
```

Gar nicht so schwer, oder? Allgemeiner gesprochen werden diejenigen Zeilen gefiltert (also behalten bzw. zurückgeliefert), für die das Filterkriterium `TRUE` ist.

```
Manche Befehle wie `filter` haben einen Allerweltsnamen; gut möglich, dass ein Befehl mit gleichem Name
```

Aufgaben[1]

```
Richtig oder Falsch!?

1. `filter` filtert Spalten.
1. `filter` ist eine Funktion aus dem Paket `dplyr`.
1. `filter` erwartet als ersten Parameter das Filterkriterium.
1. `filter` lässt nur ein Filterkriterium zu.
1. Möchte man aus dem Datensatz `profiles` (`okcupiddata`) die Frauen filtern, so ist folgende Syntax k
```

Vertiefung: Fortgeschrittene Beispiele für `filter`

Einige fortgeschrittene Beispiele für `filter`:

Man kann alle Elemente (Zeilen) filtern, die zu einer Menge gehören und zwar mit diesem Operator: `%in%`:

```
filter(profiles, body_type %in% c("a little extra", "average"))
```

Besonders Textdaten laden zu einigen Extra-Überlegungen ein; sagen wir, wir wollen alle Personen filtern, die Katzen bei den Haustieren erwähnen. Es soll reichen, wenn `cat` ein Teil des Textes ist; also `likes dogs and likes cats` wäre OK (soll gefiltert werden). Dazu nutzen wir ein Paket zur Bearbeitung von Strings (Textdaten):

[1] F, R, F, F, R

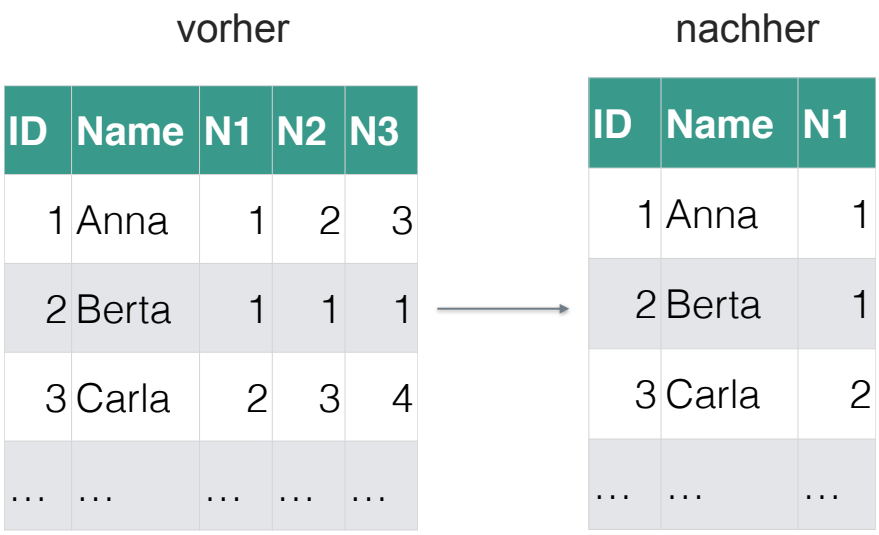


Abbildung 3: Spalten auswählen

```
filter(profiles, str_detect(pets, "cats"))
```

Ein häufiger Fall ist, Zeilen *ohne* fehlende Werte (`NA`s) zu filtern. Das geht einfach:

```
profiles_keine_nas <- na.omit(profiles)
```

Aber was ist, wenn wir nur bei bestimmten Spalten wegen fehlender Werte besorgt sind? Sagen wir bei `income` und bei `sex`:

```
filter(profiles, !is.na(income) | !is.na(sex))
```

Spalten wählen mit `select`

Das Gegenstück zu `filter` ist `select`; dieser Befehl liefert die gewählten Spalten zurück. Das ist häufig praktisch, wenn der Datensatz sehr "breit" ist, also viele Spalten enthält. Dann kann es übersichtlicher sein, sich nur die relevanten auszuwählen. Das Sinnbild für diesen Befehl:

Merke:

> Die Funktion select wählt Spalten aus einem Dataframe aus.

Laden wir als ersten einen Datensatz.

```
stats_test <- read.csv("data/test_inf_short.csv")
```

Dieser Datensatz beinhaltet Daten zu einer Statistikklausur.

Beachten Sie, dass diese Syntax davon ausgeht, dass sich die Daten in einem Unterordner mit dem Namen `data` befinden, welcher sich im Arbeitsverzeichnis befindet[2].

```
select(stats_test, score)  # Spalte `score` auswählen
select(stats_test, score, study_time)  # Splaten `score` und `study_time` auswählen
select(stats_test, score:study_time) # dito
select(stats_test, 5:6) Spalten 5 bis 6 auswählen
```

[2] der angegebene Pfad ist also *relativ* zum aktuellen Verzeichnis.

Tatsächlich ist der Befehl `select` sehr flexibel; es gibt viele Möglichkeiten, Spalten auszuwählen. Im `dplyr`-Cheatsheet findet sich ein guter Überblick dazu.

[Aufgaben]Aufgaben[3]

```
Richtig oder Falsch!?

1. `select` wählt *Zeilen* aus.
1. `select` ist eine Funktion aus dem Paket `knitr`.
1. Möchte man zwei Spalten auswählen, so ist folgende Syntax prinzipiell korrekt: `select(df, spalte1, s
1. Möchte man Spalten 1 bis 10 auswählen, so ist folgende Syntax prinzipiell korrekt: `select(df, spalte
1. Mit `select` können Spalten nur bei ihrem Namen, aber nicht bei ihrer Nummer aufgerufen werden.
```

Zeilen sortieren mit `arrange`

Man kann zwei Arten des Umgangs mit R unterscheiden: Zum einen der "interaktive Gebrauch" und zum anderen "richtiges Programmieren". Im interaktiven Gebrauch geht es uns darum, die Fragen zum aktuell vorliegenden Datensatz (schnell) zu beantworten. Es geht nicht darum, eine allgemeine Lösung zu entwickeln, die wir in die Welt verschicken können und die dort ein bestimmtes Problem löst, ohne dass der Entwickler (wir) dabei Hilfestellung geben muss. "Richtige" Software, wie ein R-Paket oder Microsoft Powerpoint, muss diese Erwartung erfüllen; "richtiges Programmieren" ist dazu vonnöten. Natürlich sind in diesem Fall die Ansprüche an die Syntax (der "Code", hört sich cooler an) viel höher. In dem Fall muss man alle Eventualitäten voraussehen und sicherstellen, dass das Programm auch beim merkwürdigsten Nutzer brav seinen Dienst tut. Wir haben hier, beim interaktiven Gebrauch, niedrigere Ansprüche bzw. andere Ziele.

Beim interaktiven Gebrauch von R (oder beliebigen Analyseprogrammen) ist das Sortieren von Zeilen eine recht häufige Tätigkeit. Typisches Beispiel wäre der Lehrer, der eine Tabelle mit Noten hat und wissen will, welche Schüler die schlechtesten oder die besten sind in einem bestimmten Fach. Oder bei der Prüfung der Umsätze nach Filialen möchten wir die umsatzstärksten sowie -schwächsten Niederlassungen kennen.

Ein R-Befehl hierzu ist `arrange`; einige Beispiele zeigen die Funktionsweise am besten:

```
arrange(stats_test, score) # liefert die *schlechtesten* Noten zuerst zurück
arrange(stats_test, -score) # liefert die *besten* Noten zuerst zurück
arrange(stats_test, interest, score)
```

```
#>     X                 V_1 study_time self_eval interest score
#> 1 234 23.01.2017 18:13:15          3         1        1    17
#> 2   4 06.01.2017 09:58:05          2         3        2    18
#> 3 131 19.01.2017 18:03:45          2         3        4    18
#> 4 142 19.01.2017 19:02:12          3         4        1    18
#> 5  35 12.01.2017 19:04:43          1         2        3    19
#> 6  71 15.01.2017 15:03:29          3         3        3    20
#>    X                 V_1 study_time self_eval interest score
#> 1  3 05.01.2017 23:33:47          5        10        6    40
#> 2  7 06.01.2017 14:25:49         NA        NA       NA    40
#> 3 29 12.01.2017 09:48:16          4        10        3    40
#> 4 41 13.01.2017 12:07:29          4        10        3    40
#> 5 58 14.01.2017 15:43:01          3         8        2    40
#> 6 83 16.01.2017 10:16:52         NA        NA       NA    40
#>     X                 V_1 study_time self_eval interest score
#> 1 234 23.01.2017 18:13:15          3         1        1    17
```

[3] F, F, R, R, F

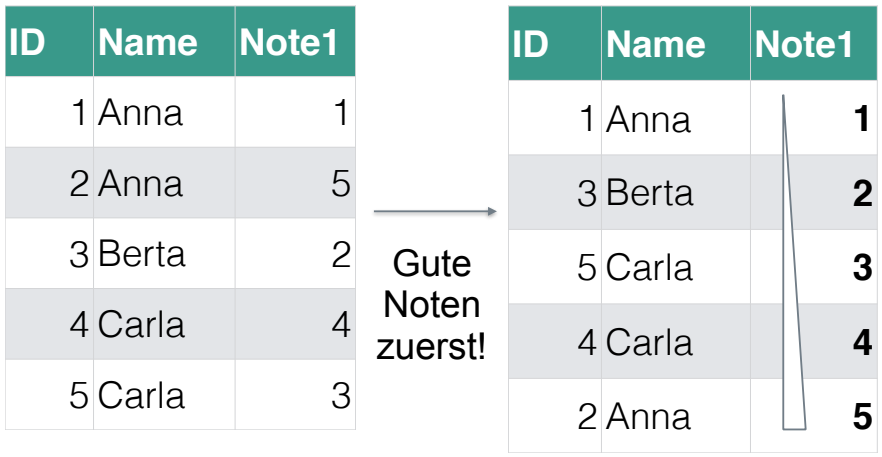


Abbildung 4: Spalten sortieren

```
#> 2 142 19.01.2017 19:02:12          3         4         1    18
#> 3 221 23.01.2017 11:40:30          1         1         1    23
#> 4 230 23.01.2017 16:27:49          1         1         1    23
#> 5  92 17.01.2017 17:18:55          1         1         1    24
#> 6 107 18.01.2017 16:01:36          3         2         1    24
```

Einige Anmerkungen. Die generelle Syntax lautet `arrange(df, Spalte1, ...)`, wobei `df` den Dataframe bezeichnet und `Spalte1` die erste zu sortierende Spalte; die Punkte `...` geben an, dass man weitere Parameter übergeben kann. Man kann sowohl numerische Spalten als auch Textspalten sortieren. Am wichtigsten ist hier, dass man weitere Spalten übergeben kann. Dazu gleich mehr.

Standardmäßig sortiert `arrange` *aufsteigend* (weil kleine Zahlen im Zahlenstrahl vor den großen Zahlen kommen). Möchte man diese Reihenfolge umdrehen (große Werte zuert, d.h. *absteigend*), so kann man ein Minuszeichen vor den Namen der Spalte setzen.

Gibt man *zwei oder mehr* Spalten an, so werden pro Wert von `Spalte1` die Werte von `Spalte2` sortiert etc; man betrachte den Output des Beispiels oben dazu.

Merke:

> Die Funktion arrange sortiert die Zeilen eines Datafames.

Ein Sinnbild zur Verdeutlichung:

Ein ähnliches Ergebnis erhält mit man `top_n()`, welches die `n` *größten Ränge* widergibt:

```
top_n(stats_test, 3)
#>      X                 V_1 study_time self_eval interest score
#> 1    3 05.01.2017 23:33:47          5        10        6    40
#> 2    7 06.01.2017 14:25:49         NA        NA       NA    40
#> 3   29 12.01.2017 09:48:16          4        10        3    40
#> 4   41 13.01.2017 12:07:29          4        10        3    40
#> 5   58 14.01.2017 15:43:01          3         8        2    40
#> 6   83 16.01.2017 10:16:52         NA        NA       NA    40
#> 7  116 18.01.2017 23:07:32          4         8        5    40
#> 8  119 19.01.2017 09:05:01         NA        NA       NA    40
#> 9  132 19.01.2017 18:22:32         NA        NA       NA    40
#> 10 175 20.01.2017 23:03:36          5        10        5    40
#> 11 179 21.01.2017 07:40:05          5         9        1    40
#> 12 185 21.01.2017 15:01:26          4        10        5    40
```

```
#> 13 196 22.01.2017 13:38:56          4        10        5    40
#> 14 197 22.01.2017 14:55:17          4        10        5    40
#> 15 248 24.01.2017 16:29:45          2        10        2    40
#> 16 249 24.01.2017 17:19:54         NA        NA       NA    40
#> 17 257 25.01.2017 10:44:34          2         9        3    40
#> 18 306 27.01.2017 11:29:48          4         9        3    40
top_n(stats_test, 3, interest)
#>     X                 V_1 study_time self_eval interest score
#> 1   3 05.01.2017 23:33:47          5        10        6    40
#> 2   5 06.01.2017 14:13:08          4         8        6    34
#> 3  43 13.01.2017 14:14:16          4         8        6    36
#> 4  65 15.01.2017 12:41:27          3         6        6    22
#> 5 110 18.01.2017 18:53:02          5         8        6    37
#> 6 136 19.01.2017 18:22:57          3         1        6    39
#> 7 172 20.01.2017 20:42:46          5        10        6    34
#> 8 214 22.01.2017 21:57:36          2         6        6    31
#> 9 301 27.01.2017 08:17:59          4         8        6    33
```

Gibt man *keine* Spalte an, so bezieht sich `top_n` auf die letzte Spalte im Datensatz.

Da sich hier mehrere Personen den größten Rang (Wert 40) teilen, bekommen wir *nicht* 3 Zeilen zurückgeliefert, sondern entsprechend mehr.

Aufgaben[4]

```
Richtig oder Falsch!?

1. `arrange` arrangiert Spalten.
1. `arrange` sortiert im Standard absteigend.
1. `arrange` lässt nur ein Sortierkriterium zu.
1. `arrange` kann numerische Werte, aber nicht Zeichenketten sortieren.
1. `top_n(5)` liefert die fünf kleinsten Ränge.
```

Datensatz gruppieren mit `group_by`

Einen Datensatz zu gruppieren ist ebenfalls eine häufige Angelegenheit: Was ist der mittlere Umsatz in Region X im Vergleich zu Region Y? Ist die Reaktionszeit in der Experimentalgruppe kleiner als in der Kontrollgruppe? Können Männer schneller ausparken als Frauen? Man sieht, dass das Gruppieren v.a. in Verbindung mit Mittelwerten oder anderen Zusammenfassungen sinnvol ist; dazu im nächsten Abschnitt mehr.

In der Abbildung wurde der Datensatz anhand der Spalte `Fach` in mehrere Gruppen geteilt. Wir könnten uns als nächstes z.B. Mittelwerte pro Fach - d.h. pro Gruppe (pro Ausprägung von `Fach`) - ausgeben lassen; in diesem Fall vier Gruppen (Fach A bis D).

```
test_gruppiert <- group_by(stats_test, interest)
test_gruppiert
#> Source: local data frame [306 x 6]
#> Groups: interest [7]
#>
#>        X                 V_1 study_time self_eval interest score
#>    <int>              <fctr>      <int>     <int>    <int> <int>
```

[4] F, F, F, F, R

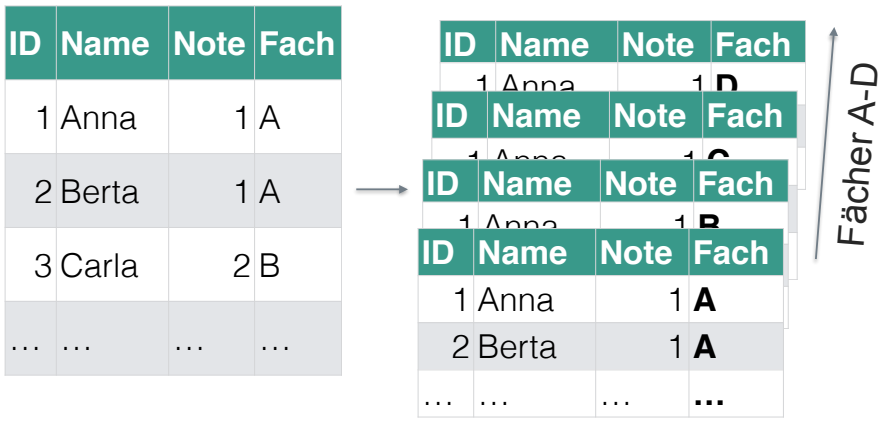


Abbildung 5: Datensätze nach Subgruppen aufteilen

```
#> 1      1 05.01.2017 13:57:01          5         8         5    29
#> 2      2 05.01.2017 21:07:56          3         7         3    29
#> 3      3 05.01.2017 23:33:47          5        10         6    40
#> 4      4 06.01.2017 09:58:05          2         3         2    18
#> 5      5 06.01.2017 14:13:08          4         8         6    34
#> 6      6 06.01.2017 14:21:18         NA        NA        NA    39
#> 7      7 06.01.2017 14:25:49         NA        NA        NA    40
#> 8      8 06.01.2017 17:24:53          2         5         3    24
#> 9      9 07.01.2017 10:11:17          2         3         5    25
#> 10    10 07.01.2017 18:10:05          4         5         5    33
#> # ... with 296 more rows
```

Schaut man sich nun den Datensatz an, sieht man erstmal wenig Effekt der Gruppierung. R teilt uns lediglich mit `Groups: interest [7]`, dass es die Gruppen gibt, aber es gibt keine extra Spalte oder sonstige Anzeichen der Gruppierung. Aber keine Sorge, wenn wir gleich einen Mittelwert ausrechnen, bekommen wir den Mittelwert pro Gruppe!

Ein paar Hinweise: `Source: local data frame [306 x 6]` will sagen, dass die Ausgabe sich auf einen `tibble` bezieht[5], also eine bestimmte Art von Dataframe. `Groups: interest [7]` zeigt, dass der Tibble in 7 Gruppen - entsprechend der Werte von `interest` aufgeteilt ist.

`group_by` an sich ist nicht wirklich nützlich. Nützlich wird es erst, wenn man weitere Funktionen auf den gruppierten Datensatz anwendet - z.B. Mittelwerte ausrechnet (z.B mit `summarise`, s. unten). Die nachfolgenden Funktionen (wenn sie aus `dplyr` kommen), berücksichtigen nämlich die Gruppierung. So kann man einfach Mittelwerte pro Gruppe ausrechnen.

Aufgaben[6]

```
Richtig oder Falsch!?

1. Mit `group_by` gruppiert man einen Datensatz.
1. `group_by` lässt nur ein Gruppierungskriterium zu.
1. Die Gruppierung durch `group_by` wird nur von Funktionen aus `dplyr` erkannt.
1. `group_by` ist sinnvoll mit `summarise` zu kombinieren.
```

[5] http://stackoverflow.com/questions/29084380/what-is-the-meaning-of-the-local-data-frame-message-from-dplyrprint-tbl-df
[6] R, F, R, R

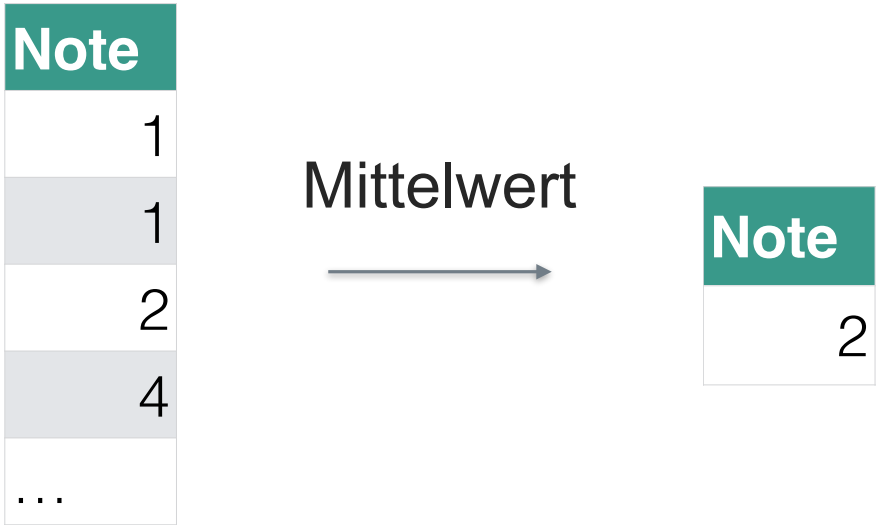


Abbildung 6: Spalten zu einer Zahl zusammenfassen

Merke:

> Mit group_by teilt man einen Datensatz in Gruppen ein, entsprechend der Werte einer mehrerer Spalten.

Eine Spalte zusammenfassen mit `summarise`

Vielleicht die wichtigste oder häufigte Tätigkeit in der Analyse von Daten ist es, eine Spalte zu *einem* Wert zusammenzufassen. Anders gesagt: Einen Mittelwert berechnen, den größten (kleinsten) Wert heraussuchen, die Korrelation berechnen oder eine beliebige andere Statistik ausgeben lassen. Die Gemeinsamkeit dieser Operaitonen ist, dass sie eine Spalte zu einem Wert zusammenfassen, "aus Spalte mach Zahl", sozusagen. Daher ist der Name des Befehls `summarise` ganz passend. Genauer gesagt fasst dieser Befehl eine Spalte zu einer Zahl zusammen *anhand* einer Funktion wie `mean` oder `max`. Hierbei ist jede Funktion erlaubt, die eine Spalte als Input verlangt und eine Zahl zurückgibt; andere Funktionen sind bei `summarise` nicht erlaubt.

```
summarise(stats_test, mean(score))
#>   mean(score)
#> 1        31.1
```

Man könnte diesen Befehl so ins Deutsche übersetzen: `Fasse aus Tabelle stats_test die Spalte score anhand des Mittelwerts zusammen`. Nicht vergessen, wenn die Spalte `score` fehlende Werte hat, wird der Befehl `mean` standardmäßig dies mit `NA` quittieren.

Jetzt können wir auch die Gruppierung nutzen:

```
test_gruppiert <- group_by(stats_test, interest)
summarise(test_gruppiert, mean(score))
#> # A tibble: 7 × 2
#>   interest `mean(score)`
#>      <int>         <dbl>
#> 1        1          28.3
#> 2        2          29.7
#> 3        3          30.8
#> 4        4          29.9
#> 5        5          32.5
```

```
#> 6        6          34.0
#> 7       NA          33.1
```

Der Befehl `summarise` erkennt also, wenn eine (mit `group_by`) gruppierte Tabelle vorliegt. Jegliche Zusammenfassung, die wir anfordern, wird anhand der Gruppierungsinformation aufgeteilt werden. In dem Beispiel bekommen wir einen Mittelwert für jeden Wert von `interest`. Interessanterweise sehen wir, dass der Mittelwert tendenziell größer wird, je größer `interest` wird.

Alle diese `dplyr`-Befehle geben einen Dataframe zurück, was praktisch ist für weitere Verarbeitung. In diesem Fall heißen die Spalten `interst` und `mean(score)`. Zweiter Name ist nicht so schön, daher ändern wir den wie folgt:

Jetzt können wir auch die Gruppierung nutzen:

```
test_gruppiert <- group_by(stats_test, interest)
summarise(test_gruppiert, mw_pro_gruppe = mean(score, na.rm = TRUE))
#> # A tibble: 7 × 2
#>   interest mw_pro_gruppe
#>      <int>         <dbl>
#> 1        1          28.3
#> 2        2          29.7
#> 3        3          30.8
#> 4        4          29.9
#> 5        5          32.5
#> 6        6          34.0
#> 7       NA          33.1
```

Nun heißt die zweite Spalte `mw_pro_Gruppe`. `na.rm = TRUE` veranlasst, bei fehlenden Werten trotzdem einen Mittelwert zurückzuliefern (die Zeilen mit fehlenden Werten werden in dem Fall ignoriert).

Grundsätzlich ist die Philosophie der `dplyr`-Befehle: "Mach nur eine Sache, aber die dafür gut". Entsprechend kann `summarise` nur *Spalten* zusammenfassen, aber keine *Zeilen*.

Merke:

> Mit summarise kann man eine Spalte eines Dataframes zu einem Wert zusammenfassen.

[Aufgaben]Aufgaben[7]

```
Richtig oder Falsch!?

1. Möchte man aus der Tabelle `stats_test` den Mittelwert für die Spalte `score` berechnen, so ist folg
1. `summarise` liefert eine Tabelle, genauer: einen Tibble, zurück.
1. Die Tabelle, die diese Funktion zurückliefert: `summarise(stats_test, mean(score))`, hat eine Spalte
1. `summarise` lässt zu, dass die zu berechnende Spalte einen Namen vom Nutzer zugewiesen bekommt.
1. `summarise` darf nur verwendet werden, wenn eine Spalte zu einem Wert zusammengefasst werden soll.
```

Zeilen zählen mit `n` und `count`

Ebenfalls nützlich ist es, Zeilen zu zählen. Im Gegensatz zum Standardbefehl[8] `nrow` versteht der `dyplr`-Befehl `n`auch Gruppierungen. `n` darf nur innerhalb von `summarise` oder ähnlichen `dplyr`-Befehlen verwendet werden.

[7] R, R, R, R, R

[8] Standardbefehl meint, dass die Funktion zum Standardrepertoire von R gehört, also nicht über ein Paket extra geladen werden muss

```
summarise(stats_test, n())
#>   n()
#> 1 306
summarise(test_gruppiert, n())
#> # A tibble: 7 × 2
#>   interest `n()`
#>      <int> <int>
#> 1        1    30
#> 2        2    47
#> 3        3    66
#> 4        4    41
#> 5        5    45
#> 6        6     9
#> 7       NA    68
nrow(stats_test)
#> [1] 306
```

Außerhalb von gruppierten Datensätzen ist `nrow` meist praktischer.

Praktischer ist der Befehl `count`, der nichts anderes ist als die Hintereinanderschaltung von `group_by` und `n`. Mit `count` zählen wir die Häufigkeiten nach Gruppen; Gruppen sind hier zumeist die Werte einer auszuzählenden Variablen (oder mehrerer auszuzählender Variablen). Das macht `count` zu einem wichtigen Helfer bei der Analyse von Häufigkeitsdaten.

```
dplyr::count(stats_test, interest)
#> # A tibble: 7 × 2
#>   interest     n
#>      <int> <int>
#> 1        1    30
#> 2        2    47
#> 3        3    66
#> 4        4    41
#> 5        5    45
#> 6        6     9
#> 7       NA    68
dplyr::count(stats_test, study_time)
#> # A tibble: 6 × 2
#>   study_time     n
#>        <int> <int>
#> 1          1    31
#> 2          2    49
#> 3          3    85
#> 4          4    56
#> 5          5    17
#> 6         NA    68
dplyr::count(stats_test, interest, study_time)
#> Source: local data frame [29 x 3]
#> Groups: interest [?]
#>
#>    interest study_time     n
#>       <int>      <int> <int>
#> 1         1          1    12
#> 2         1          2     7
#> 3         1          3     8
#> 4         1          4     2
```

Abbildung 7: La trahison des images [Ceci n'est pas une pipe], René Magritte, 1929, © C. Herscovici, Brussels / Artists Rights Society (ARS), New York, http://collections.lacma.org/node/239578

```
#> 5         1          5     1
#> 6         2          1     9
#> 7         2          2    15
#> 8         2          3    16
#> 9         2          4     6
#> 10        2          5     1
#> # ... with 19 more rows
```

Allgemeiner formuliert lautet die Syntax: `count(df, Spalte1, ...)`, wobei `df` der Dataframe ist und `Spalte1` die erste (es können mehrere sein) auszuzählende Spalte. Gibt man z.B. zwei Spalten an, so wird pro Wert der 1. Spalte die Häufigkeiten der 2. Spalte ausgegeben.

Merke:

> n und count zählen die Anzahl der Zeilen, d.h. die Anzahl der Fälle.

[Aufgaben]Aufgaben[9]

```
Richtig oder Falsch!?

1. Mit `count` kann man Zeilen zählen.
1. `count` ist ähnlich (oder identisch) zu einer Kombination von `group_by` und `n()`.
1. Mit `count` kann man nur nur eine Gruppe beim Zählen berücksichtigen.
1. `count` darf nicht bei nominalskalierten Variablen verwendet werden.
```

Vertiefung

Die Pfeife

Die zweite Idee kann man salopp als "Durchpfeifen" bezeichnen; ikonographisch mit diesem Symbol dargestellt `%>%`. Der Begriff "Durchpfeifen" ist frei vom Englischen "to pipe" übernommen. Das berühmte Bild von René Magritte stand dabei Pate.

Hierbei ist gemeint, einen Datensatz sozusagen auf ein Fließband zu legen und an jedem Arbeitsplatz einen Arbeitsschritt auszuführen. Der springende Punkt ist, dass ein Dataframe als "Rohstoff" eingegeben wird und jeder Arbeitsschritt seinerseits wieder einen Datafram ausgiebt. Damit kann man sehr schön, einen "Flow" an Verarbeitung erreichen, außerdem spart man sich Tipparbeit und die Syntax wird lesbarer. Damit das

[9] R, R, F, F

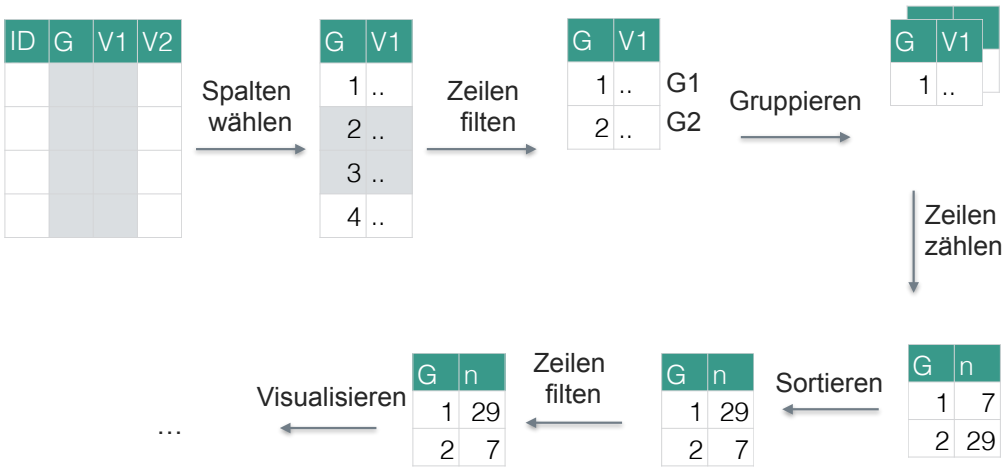


Abbildung 8: Das 'Durchpeifen'

Durchpfeifen funktioniert, benötigt man Befehle, die als Eingabe einen Dataframe erwarten und wieder einen Dataframe zurückliefern. Das Schaubild verdeutlich beispielhaft eine Abfolge des Durchpfeifens.

Die sog. "Pfeife" (pipe: `%>%`) in Anspielung an das berühmte Bild von René Magritte, verkettet Befehle hintereinander. Das ist praktisch, da es die Syntax vereinfacht. Vergleichen Sie mal diese Syntax

```
filter(summarise(group_by(filter(stats_test, !is.na(score)), interest), mw = mean(score)), mw > 30)
```

mit dieser

```
stats_test %>%
  filter(!is.na(score)) %>%
  group_by(interest) %>%
  summarise(mw = mean(score)) %>%
  filter(mw > 30)
#> # A tibble: 4 × 2
#>   interest    mw
#>      <int> <dbl>
#> 1        3  30.8
#> 2        5  32.5
#> 3        6  34.0
#> 4       NA  33.1
```

Es ist hilfreich, diese "Pfeifen-Syntax" in deutschen Pseudo-Code zu übersetzen.

```
Nimm die Tabelle "stats_test" UND DANN
filtere alle nicht-fehlenden Werte UND DANN
gruppiere die verbleibenden Werte nach "interest" UND DANN
bilde den Mittelwert (pro Gruppe) für "score" UND DANN
liefere nur die Werte größer als 30 zurück.
```

Die Pfeife zerlegt die "russische Puppe", also ineinander verschachtelteten Code, in sequenzielle Schritte und zwar in der richtigen Reihenfolge (entsprechend der Abarbeitung). Wir müssen den Code nicht mehr von innen nach außen lesen (wie das bei einer mathematischen Formel der Fall ist), sondern können wie bei einem Kochrezept "erstens ..., zweitens .., drittens ..." lesen. Die Pfeife macht die Syntax einfacher. Natürlich hätten wir die verschachtelte Syntax in viele einzelne Befehle zerlegen können und jeweils eine Zwischenergebnis speichern mit dem Zuweisungspfeil `<-` und das Zwischenergebnis dann explizit an den nächsten Befehl weitergeben. Eigentlich macht die Pfeife genau das - nur mit weniger Tipparbeit. Und auch einfacher zu lesen. Flow!

Werte umkodieren und "binnen"

car::recode

Manchmal möchte man z.B. negativ gepolte Items umdrehen oder bei kategoriellen Variablen kryptische Bezeichnungen in sprechendere umwandeln (ein Klassiker ist `1` in `maennlich` bzw. `2` in `weiblich` oder umgekehrt, kann sich niemand merken). Hier gibt es eine Reihe praktischer Befehle, z.B. `recode` aus dem Paket `car`. Übrigens: Wenn man explizit angeben möchte, aus welchem Paket ein Befehl stammt (z.B. um Verwechslungen zu vermeiden), gibt man `Paketnamen::Befehlnamen` an. Schauen wir uns ein paar Beispiele zum Umkodieren an.

```
stats_test$score_fac <- car::recode(stats_test$study_time, "5 = 'sehr viel'; 2:4 = 'mittel'; 1 = 'wenig
stats_test$score_fac <- car::recode(stats_test$study_time, "5 = 'sehr viel'; 2:4 = 'mittel'; 1 = 'wenig

stats_test$study_time <- car::recode(stats_test$study_time, "5 = 'sehr viel'; 4 = 'wenig'; else = 'Hilfe

head(stats_test$study_time)
#> [1] sehr viel Hilfe     sehr viel Hilfe     wenig     Hilfe
#> Levels: Hilfe sehr viel wenig
```

Der Befehle `recode` ist wirklich sehr prkatisch; mit `:` kann man "von bis" ansprechen (das ginge mit `c()` übrigens auch); `else` für "ansonsten" ist möglich und mit `as.factor.result` kann man entweder einen Faktor oder eine Text-Variable zurückgeliefert bekommen. Der ganze "Wechselterm" steht in Anführungsstrichen (`"`). Einzelne Teile des Wechselterms sind mit einem Strichpunkt (`;`) voneinander getrennt.

Das klassiche Umkodieren von Items aus Fragebögen kann man so anstellen; sagen wir `interest` soll umkodiert werden:

```
stats_test$no_interest <- car::recode(stats_test$interest, "1 = 6; 2 = 5; 3 = 4; 4 = 3; 5 = 2; 6 = 1; el
glimpse(stats_test$no_interest)
#>  num [1:306] 2 4 1 5 1 NA NA 4 2 2 ...
```

Bei dem Wechselterm muss man aufpassen, nichts zu verwechseln; die Zahlen sehen alle ähnlich aus...

Testen kann man den Erfolg des Umpolens mit

```
dplyr::count(stats_test, interest)
#> # A tibble: 7 × 2
#>   interest     n
#>      <int> <int>
#> 1        1    30
#> 2        2    47
#> 3        3    66
#> 4        4    41
#> 5        5    45
#> 6        6     9
#> 7       NA    68
dplyr::count(stats_test, no_interest)
#> # A tibble: 7 × 2
#>   no_interest     n
#>         <dbl> <int>
#> 1           1     9
#> 2           2    45
#> 3           3    41
#> 4           4    66
#> 5           5    47
#> 6           6    30
```

```
#> 7          NA    68
```

Scheint zu passen. Noch praktischer ist, dass man so auch numerische Variablen in Bereiche aufteilen kann ("binnen"):

```
stats_test$Ergebnis <- car::recode(stats_test$score, "1:38 = 'durchgefallen'; else = 'bestanden'")
```

Natürlich gibt es auch eine Pfeifen komptatible Version, um Variablen umzukodieren bzw. zu binnen: `dplyr::recode`[10]. Die Syntax ist allerdings etwas weniger komfortabel (da strenger), so dass wir an dieser Stelle bei `car::recode` bleiben.

Numerische Werte in Klassen gruppieren mit `cut`

Numerische Werte in Klassen zu gruppieren ("to bin", denglisch: "binnen") kann mit dem Befehl `cut` (and friends) besorgt werden.

Es lassen sich drei typische Anwendungsformen unterscheiden:

Eine numerische Variable ...

1. in k gleich große Klassen grupieren (gleichgroße Intervalle)
2. so in Klassen gruppieren, dass in jeder Klasse n Beobachtungen sind (gleiche Gruppengrößen)
3. in beliebige Klassen gruppieren

gleichgroße Intervalle

Nehmen wir an, wir möchten die numerische Variable "Körpergröße" in drei Gruppen einteilen: "klein", "mittel" und "groß". Der Range von Körpergröße soll gleichmäßig auf die drei Gruppen aufgeteilt werden, d.h. der Range (Interval) der drei Gruppen soll gleich groß sein. Dazu kann man `cut_interval` aus `ggplot2` nehmen [^d.h. `ggplot2` muss geladen sein; wenn man `tidyverse` lädt, wird `ggplot2` automatisch auch geladen].

```
wo_men <- read_csv("data/wo_men.csv")

wo_men %>%
  filter(height > 150, height < 220) -> wo_men2

temp <- cut_interval(x = wo_men2$height, n = 3)

levels(temp)
#> [1] "[155,172]" "(172,189]" "(189,206]"
```

`cut_interval` liefert eine Variabel vom Typ `factor` zurück.

gleiche Gruppengrößen

```
temp <- cut_number(wo_men2$height, n = 2)
str(temp)
#>  Factor w/ 2 levels "[155,169]","(169,206]": 1 2 2 2 2 1 1 2 1 2 ...
```

Mit `cut_number` (aus ggplot2) kann man einen Vektor in `n` Gruppen mit (etwa) gleich viel Observationen einteilen.

> Teilt man einen Vektor in zwei gleich große Gruppen, so entspricht das einer Aufteilung am Median (Median-Split).

[10] https://blog.rstudio.org/2016/06/27/dplyr-0-5-0/

In beliebige Klassen gruppieren

```
wo_men$groesse_gruppe <- cut(wo_men$height,
                             breaks = c(-Inf, 100, 150, 170, 200, 230, Inf))

count(wo_men, groesse_gruppe)
#> # A tibble: 6 × 2
#>   groesse_gruppe     n
#>           <fctr> <int>
#> 1     (-Inf,100]     4
#> 2      (150,170]    55
#> 3      (170,200]    38
#> 4      (200,230]     2
#> 5     (230, Inf]     1
#> 6             NA     1
```

`cut` ist im Standard-R (Paket "base") enthalten. Mit `breaks` gibt man die Intervallgrenzen an. Zu beachten ist, dass man eine Unter- bzw. Obergrenze angeben muss. D.h. der kleinste Wert wird nicht automatisch als unterste Intervallgrenze herangezogen.

Verweise

- Eine schöne Demonstration der Mächtigkeit von `dplyr` findet sich hier[11].
- Die GUI "exploratory" ist ein "klickbare" Umsetzung von `dplyr`, mächtig, modern und sieht cool aus: https://exploratory.io.
- *R for Data Science* bietet umfangreiche Unterstützung zu diesem Thema [@r4ds].

[11]: http://bit.ly/2kX9lvC